In Django, **authentication** is the process of verifying the identity of a user who is trying to access a system. It involves confirming that a user is who they claim to be. Django provides a built-in authentication system that is robust, flexible, and easy to extend, making it a popular choice for managing users, passwords, permissions, and session-based login in web applications.

### **Key Components of Django Authentication**

Django's authentication system is built around several main components:

1. **User Model (django.contrib.auth.models.User)**:
   * The User model represents users in Django. By default, it includes fields like username, password, email, first\_name, and last\_name.
   * You can extend or replace the default user model if you need additional fields or want to customize the authentication mechanism.
2. **Authentication Backend (django.contrib.auth.backends)**:
   * Django uses *authentication backends* to determine how user credentials are validated. The default backend authenticates users against the User model, but you can define custom backends to use other credential systems or databases.
   * Backends allow for flexibility in handling different ways of authentication, such as username-password login, token-based systems, and integration with external identity providers like OAuth or LDAP.
3. **Password Hashing and Storage**:
   * Passwords are securely hashed and stored using Django's password hashing system. Django uses a secure, salted hash algorithm by default, which protects against common password attacks.
   * When a user logs in, the password they provide is hashed and compared to the stored hash. Django never stores plain-text passwords.
4. **Authentication Methods**:
   * Django provides helper methods like authenticate(), login(), and logout() to handle the login and logout process.
     + authenticate(request, username=username, password=password): Verifies a user's credentials. If correct, it returns the corresponding User object.
     + login(request, user): Attaches the authenticated user to the session.
     + logout(request): Logs out the user by clearing the session data.
5. **Permissions and Authorization**:
   * **Permissions** define what actions an authenticated user can perform within the system.
   * Django's permission system works on both a per-model and per-object level. It includes built-in permissions for creating, editing, and deleting objects in each model, and allows you to define custom permissions.
   * Permissions are often checked using the @login\_required decorator for simple access control or through the more granular has\_perm and has\_module\_perms methods.
6. **Session Management**:
   * Django uses session cookies to keep track of logged-in users across multiple requests. When a user logs in, Django creates a session and stores the session ID in a cookie.
   * The session data (such as the user's ID) is stored server-side, keeping the user’s state consistent across requests.
7. **Django Forms for Authentication**:
   * Django provides form classes like AuthenticationForm and UserCreationForm to handle user login and registration.
   * These forms automatically include validation and other security features, making it easier to build secure authentication pages.

### **Steps in the Authentication Workflow**

The authentication workflow in Django can be summarized as follows:

1. **Login**:
   * A user submits their username and password through a login form.
   * Django calls the authenticate() method to verify the credentials. If they are valid, it returns a User object; otherwise, it returns None.
   * If the credentials are valid, login() is called to start a session for the user. A session ID is stored in a browser cookie, allowing the server to recognize the user on subsequent requests.
2. **Access Control**:
   * Views can restrict access to logged-in users using decorators like @login\_required, or restrict access based on permissions using @permission\_required.
   * For custom permissions, the has\_perm method allows you to check whether a user has a specific permission.
3. **Logout**:
   * When a user logs out, logout() is called, which removes the user’s session data and cookie, effectively ending the session and clearing authentication.

***Authorization***

**Authorization** in Django is the process of determining what a user is allowed to do within the application. Once a user has been authenticated (i.e., their identity has been verified), authorization determines which resources they can access and what actions they can perform. In Django, authorization is typically managed using permissions, groups, and decorators that control access to various parts of an application.

### **Key Components of Authorization in Django**

### **Key Components of Authorization in Django**

Django’s authorization system includes several main components:

1. **Permissions**:
   * Permissions define which actions a user is allowed to perform on a given model or view.
   * Django provides three default permissions for each model: add, change, and delete, which are automatically created when the model is registered with the admin.
   * You can also define custom permissions by adding them to the model’s Meta class.

class MyModel(models.Model):

name = models.CharField(max\_length=255)

class Meta:

permissions = [

("view\_special\_report", "Can view special report"),

]

* + In the above example, a custom permission called view\_special\_report is created for MyModel.

1. **Groups**:
   * Groups allow you to organize users into roles and assign permissions to those roles collectively, rather than individually.
   * This is useful for creating roles like “Admin,” “Editor,” or “Viewer,” where each role has a specific set of permissions.
   * Users assigned to a group inherit that group’s permissions.
2. **Authorization Methods**:
   * Django provides various methods to check permissions, either at the model level or the view level.
   * These methods include has\_perm() and has\_module\_perms() to check if a user has a specific permission or any permission within a specific app.
   * Additionally, there are decorators like @permission\_required and @user\_passes\_test to enforce authorization at the view level.
3. **Django Admin Permission Controls**:
   * The Django admin interface leverages the permissions framework to control which models a user can view, add, edit, and delete.
   * Only users with the appropriate permissions will see certain models and actions in the admin interface, making it easy to limit administrative access.
4. **Object-Level Permissions (Django-Guardian)**:
   * While Django’s built-in permissions apply to all instances of a model (e.g., any Post in a blog), object-level permissions allow you to control access on a per-object basis.
   * Django doesn’t natively support object-level permissions, but packages like django-guardian make it possible to specify permissions for individual instances of a model, such as allowing a user to edit only the blog posts they own.

### **Authorization in Views**

In Django, there are several ways to implement authorization at the view level.

1. **@login\_required Decorator**:
   * This decorator restricts access to a view to authenticated users only. It does not check for specific permissions but ensures that the user is logged in.

from django.contrib.auth.decorators import login\_required

@login\_required

def my\_view(request):

# view logic here

2.**@permission\_required Decorator**:

* This decorator checks if a user has a specific permission before granting access to the view. It’s useful for views that require specific actions, like adding or deleting items.

from django.contrib.auth.decorators import permission\_required

@permission\_required('myapp.view\_special\_report')

def my\_special\_report\_view(request):

# view logic here

**3.Checking Permissions with has\_perm**:

* You can check permissions programmatically using the has\_perm() method on the User object. This allows you to conditionally display content or redirect users based on their permissions.

if request.user.has\_perm('myapp.view\_special\_report'):

# Show special report content

else:

# Redirect or show error message

**4.Class-Based Views Authorization**:

* For class-based views, you can use the PermissionRequiredMixin to restrict access based on permissions.

from django.contrib.auth.mixins import PermissionRequiredMixin

from django.views.generic import ListView

from .models import MyModel

class SpecialReportView(PermissionRequiredMixin, ListView):

permission\_required = 'myapp.view\_special\_report'

model = MyModel

template\_name = 'special\_report.html'

**5.Custom Authorization Logic with @user\_passes\_test**:

* The @user\_passes\_test decorator allows you to define custom authorization logic, such as checking for multiple conditions or attributes not covered by the built-in permission system.

from django.contrib.auth.decorators import user\_passes\_test

def is\_in\_group(user):

return user.groups.filter(name='SpecialGroup').exists()

@user\_passes\_test(is\_in\_group)

def my\_special\_group\_view(request):

# View logic here

### ***Example of Authorization Workflow***

Let’s go through a basic example of how Django authorization works in practice:

1. **Define Permissions**:
   * Suppose you have a model Document, and you want users to have specific permissions to view, edit, and delete documents.
   * You can define these permissions in the model’s Meta class or use Django’s default model-level permissions.

class Document(models.Model):

title = models.CharField(max\_length=255)

content = models.TextField()

class Meta:

permissions = [

("view\_document", "Can view document"),

("edit\_document", "Can edit document"),

("delete\_document", "Can delete document"),

]

2.**Assign Permissions to Groups**:

* In the Django admin interface, create groups (e.g., "Editors" and "Viewers") and assign permissions to these groups based on the role.
* For example, the "Editors" group can have view, edit, and delete permissions, while the "Viewers" group only has view permission.

**Check Permissions in Views**:

* In the view that renders documents, check if the user has the appropriate permissions to view, edit, or delete.

from django.shortcuts import get\_object\_or\_404, redirect

from django.contrib.auth.decorators import permission\_required

from .models import Document

@permission\_required('myapp.view\_document')

def document\_view(request, doc\_id):

document = get\_object\_or\_404(Document, id=doc\_id)

return render(request, 'document\_detail.html', {'document': document})

@permission\_required('myapp.edit\_document')

def document\_edit(request, doc\_id):

document = get\_object\_or\_404(Document, id=doc\_id)

if request.method == 'POST':

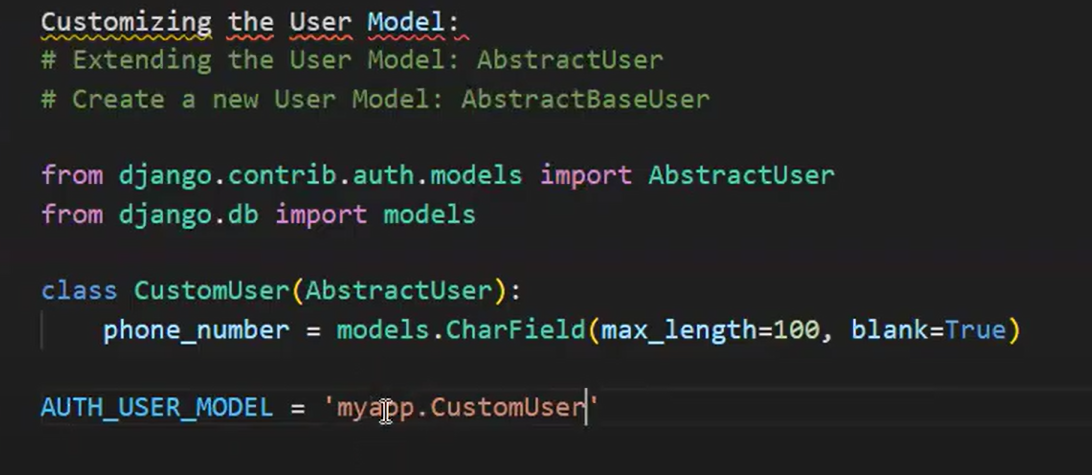
# Handle edit form submission

pass

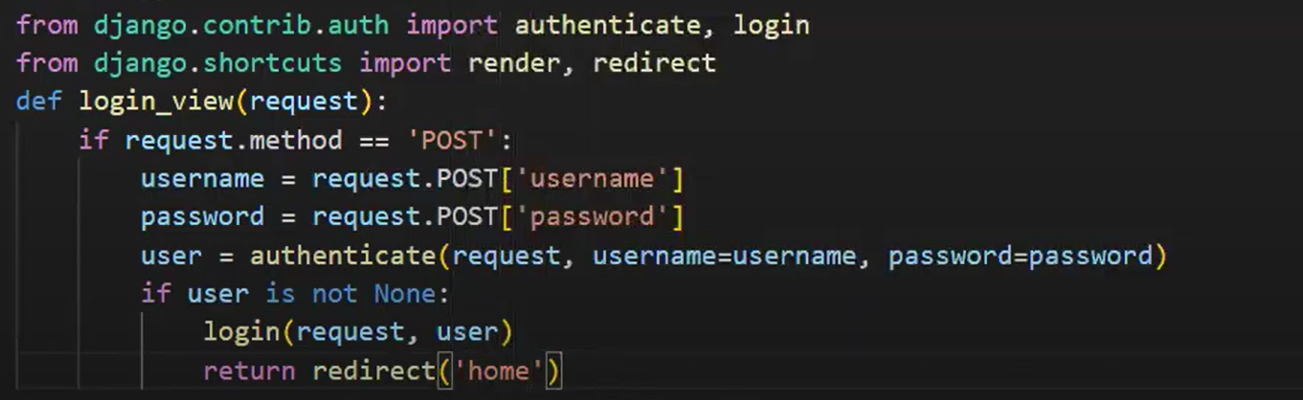
return render(request, 'document\_edit.html', {'document': document})

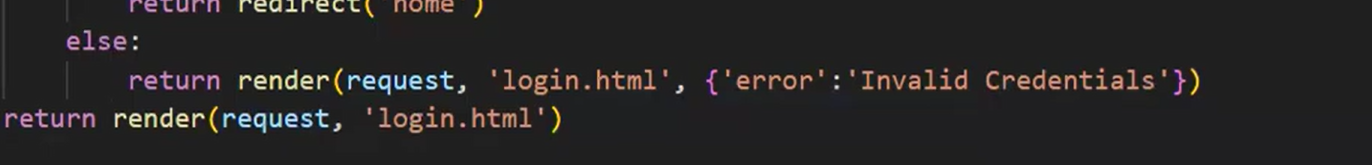
### **Conclusion**

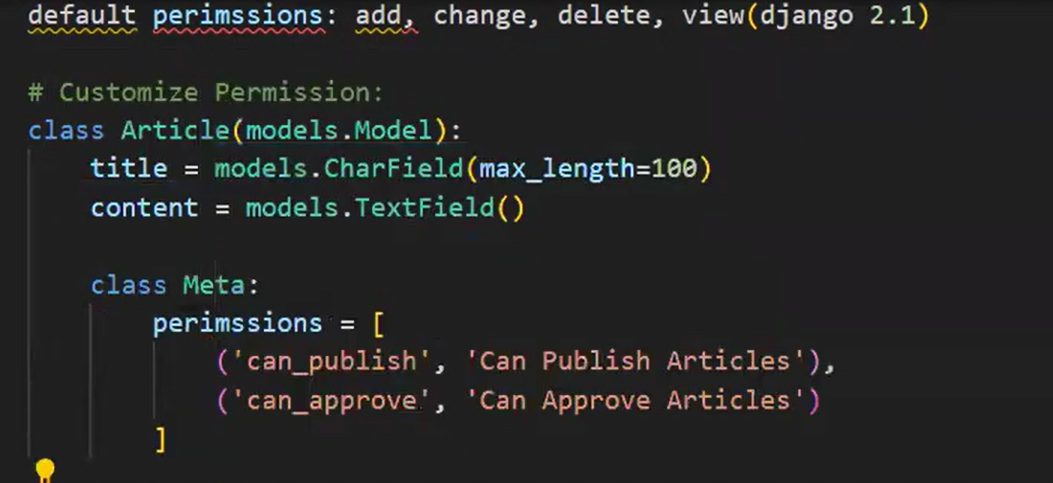
Authorization in Django is essential for controlling what users are allowed to do within an application. By leveraging permissions, groups, decorators, and custom logic, Django makes it easy to define granular access control. The built-in tools provide flexibility for most use cases, and third-party packages can further extend Django’s capabilities to handle advanced authorization requirements like object-level permissions.



views.py





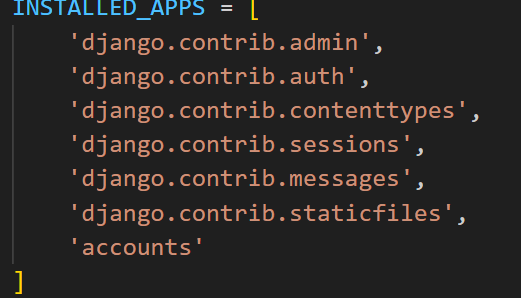




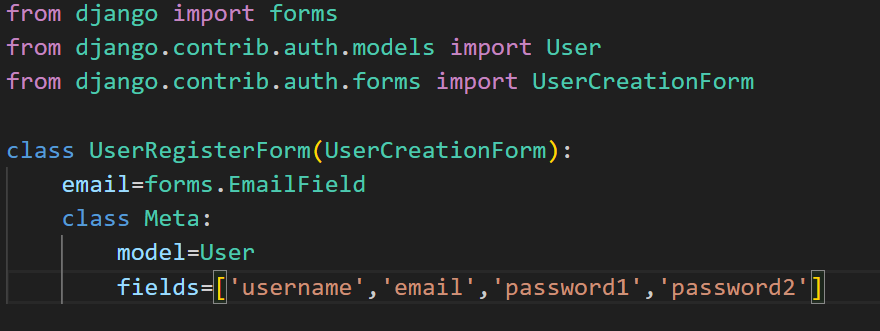


**Project steps:**

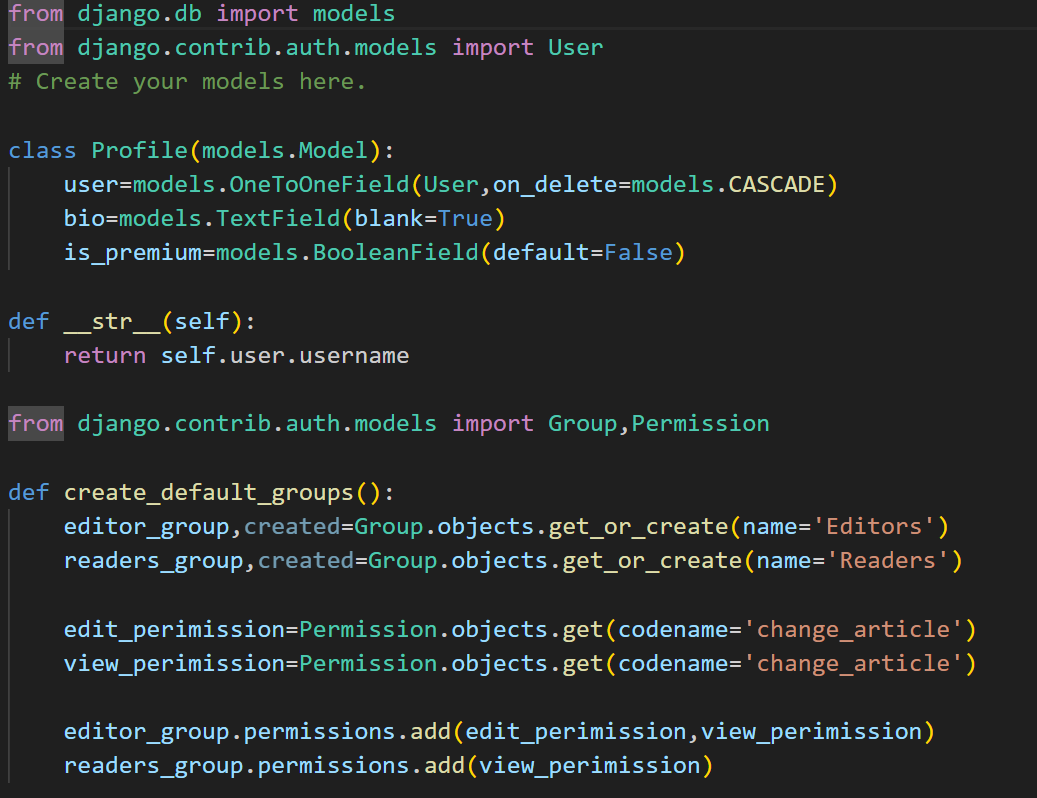
1. **Django-admin startproject auth\_project**
2. **Python manage.py startapp accounts**
3. **In settings.py**

****

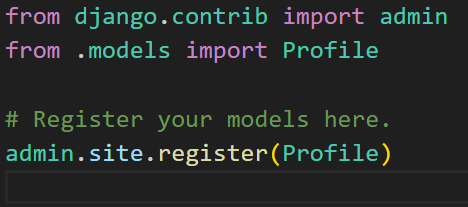
1. **Creating forms.py in application level:**

****

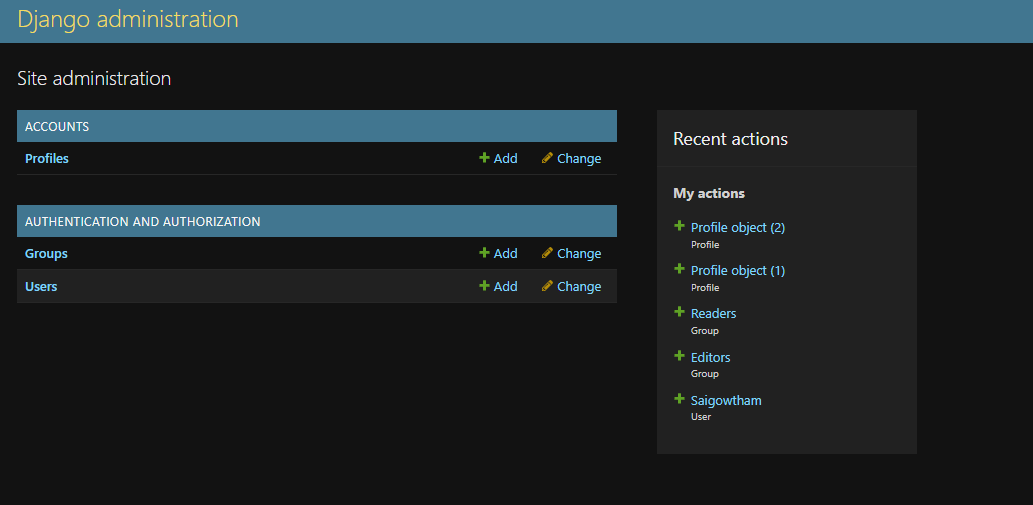
1. **In models.py**

****

1. **In admin.py**

****

1. **Python manage.py makemigrations**
2. **Python manage.py migrate**
3. **Python manage,py createsuperuser**
4. **Create username and password**
5. **In admin level—http://127.0.0.1:8000/admin/**

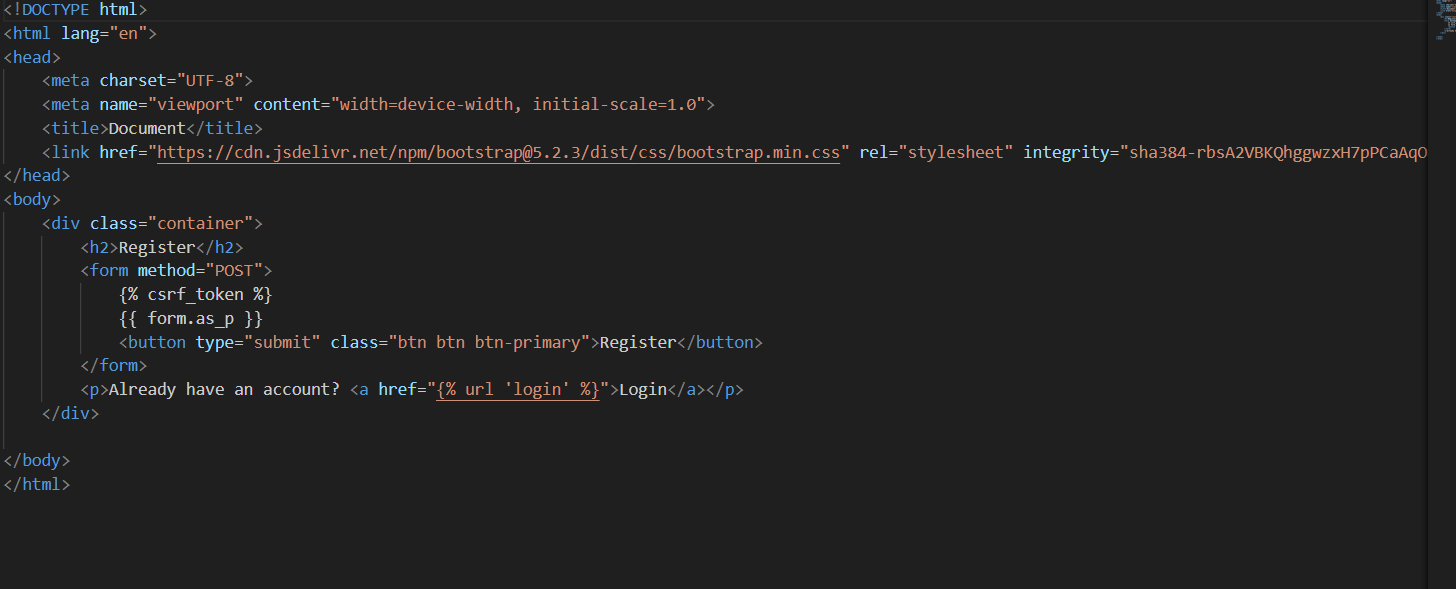
****

1. **In views.py**

****

1. **Creating html files in templates/accounts level:**

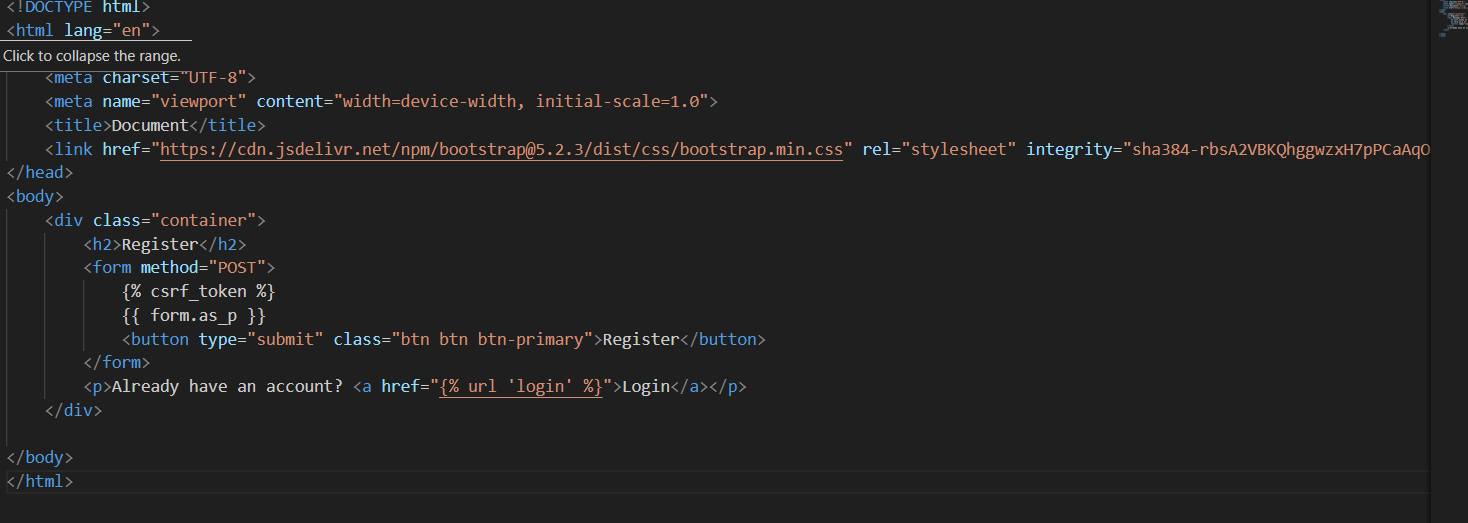
**Profile.html**

****

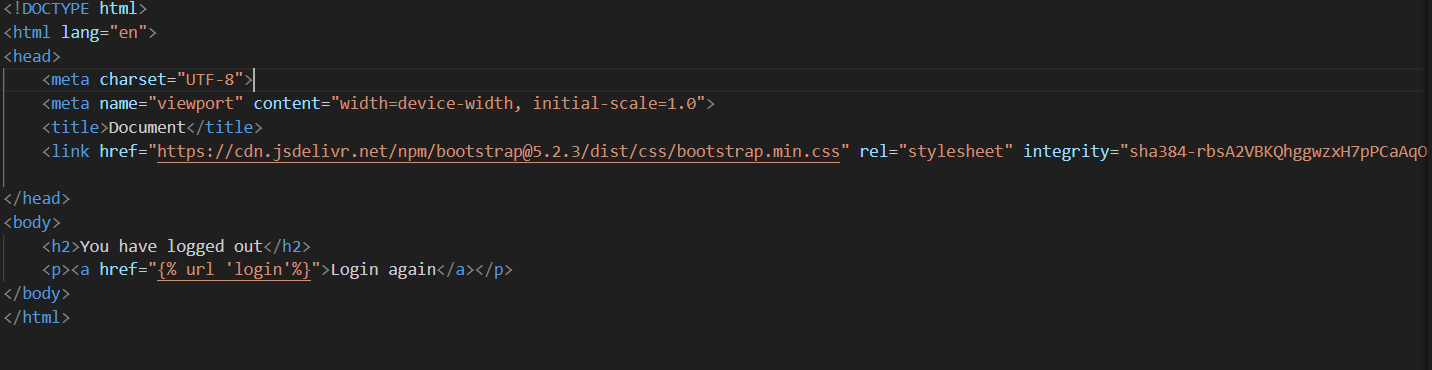
**Register.html**

****

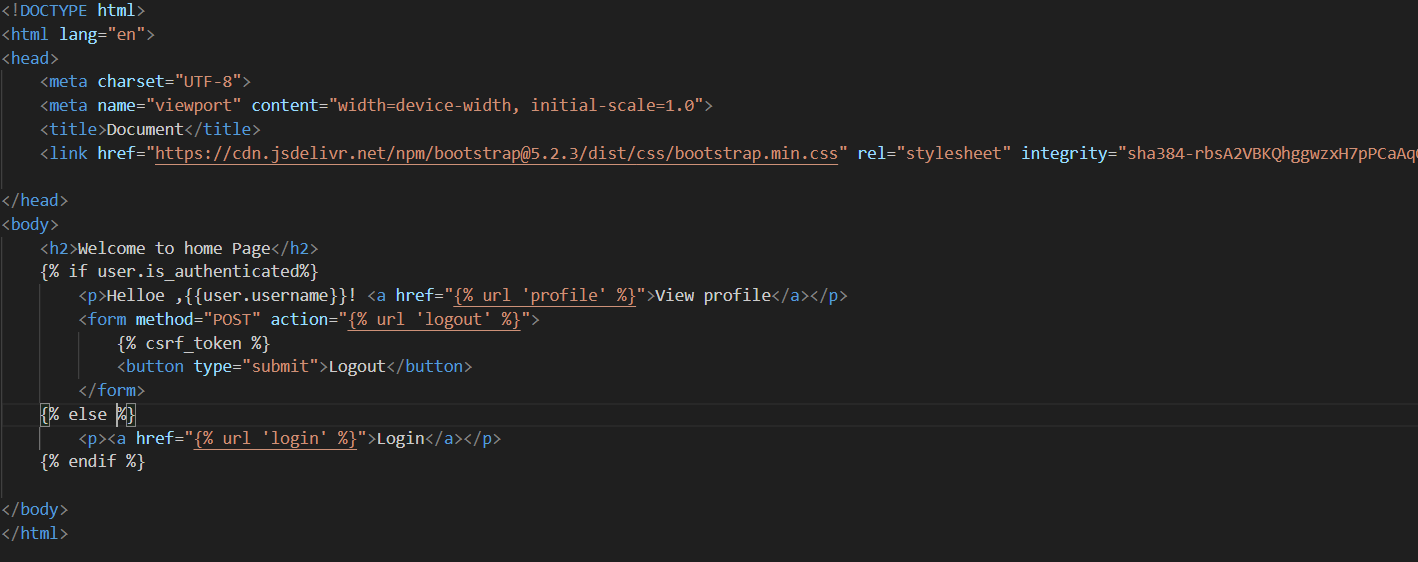
**Login.html**

****

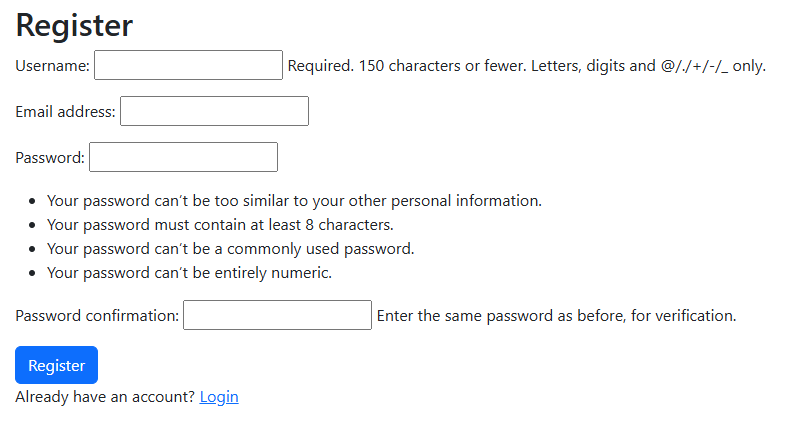
**Logout.html**

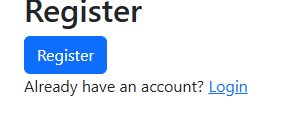
****

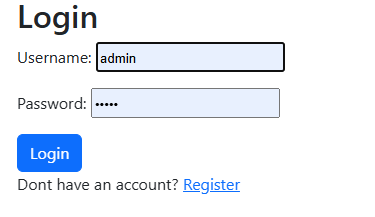
**Home.html**

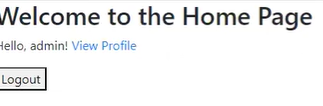
****

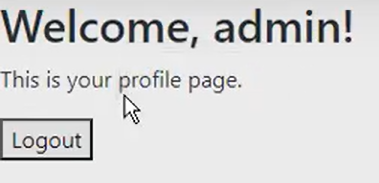
1. **Python manage.py runserver**
2. **Output**

****

****

****

****

****