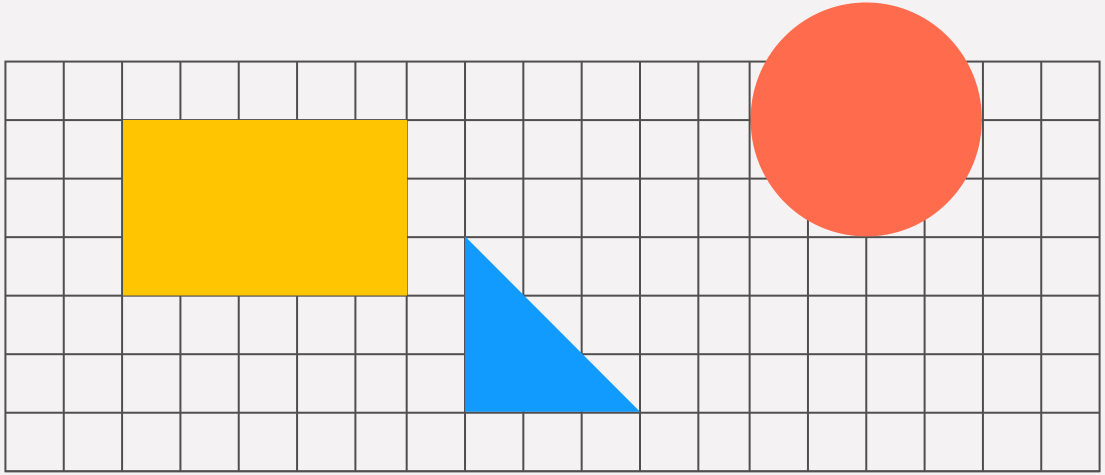
Django Session & Authentication

Ali Abrishami



In this Lecture You will...

a

Get to know Django Session

b

Explore Django's Authentication

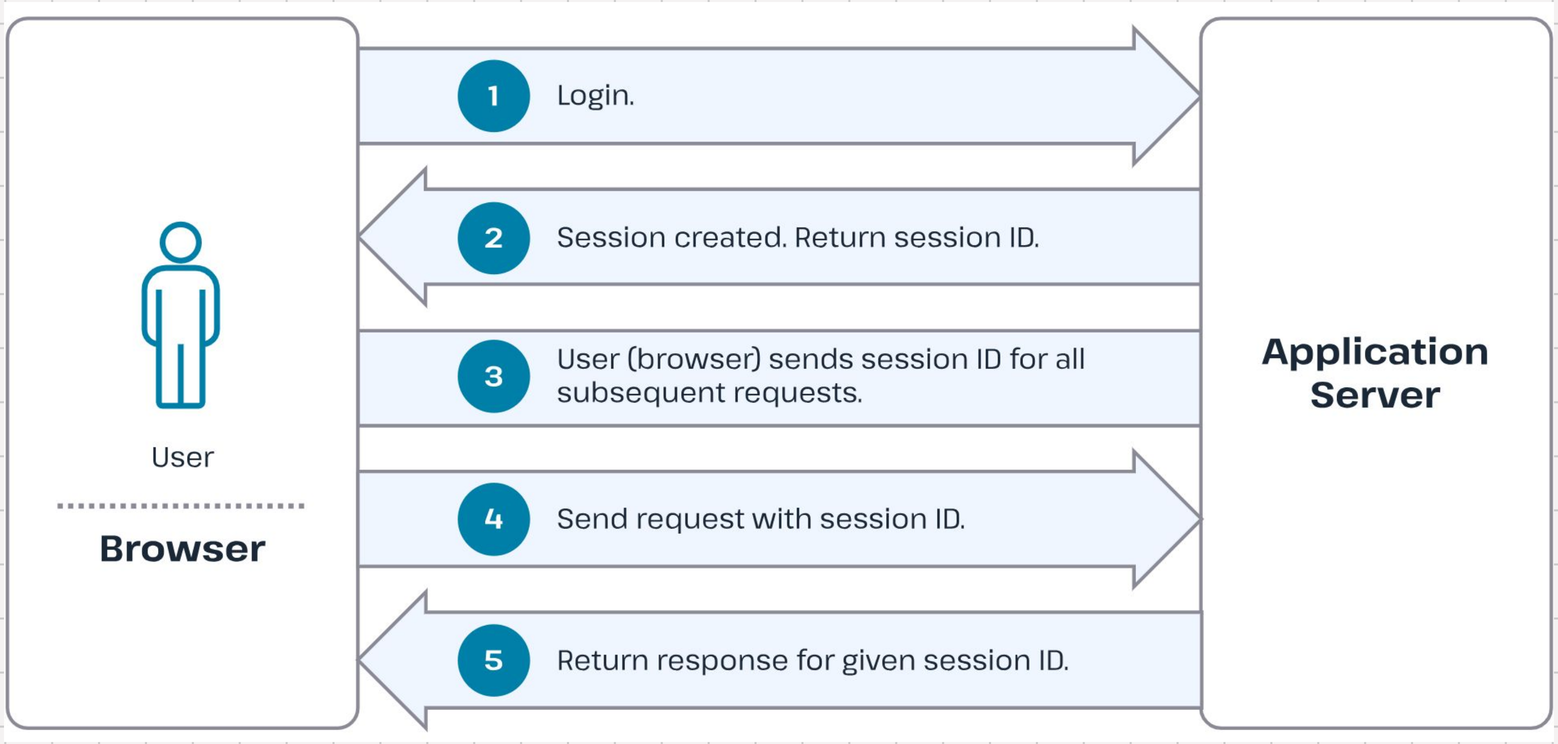
c

Explore Django Authorization

What are Sessions?

- All communication between web browsers and servers is via HTTP, which is **stateless**.
- It means that messages between the client and server are completely **independent** of each other.
- There is no notion of "sequence" or behavior based on previous messages.
- Sessions are the mechanism for keeping track of the **"state"** between the site and a particular browser.
- Sessions allow you to store arbitrary data per **browser**, and have this data available to the site whenever the browser connects.

Web Sessions


User
Browser
Application Server
1 Login.
2 Session created. Return session ID.
3 User (browser) sends session ID for all subsequent requests.
4 Send request with session ID.
5 Return response for given session ID.

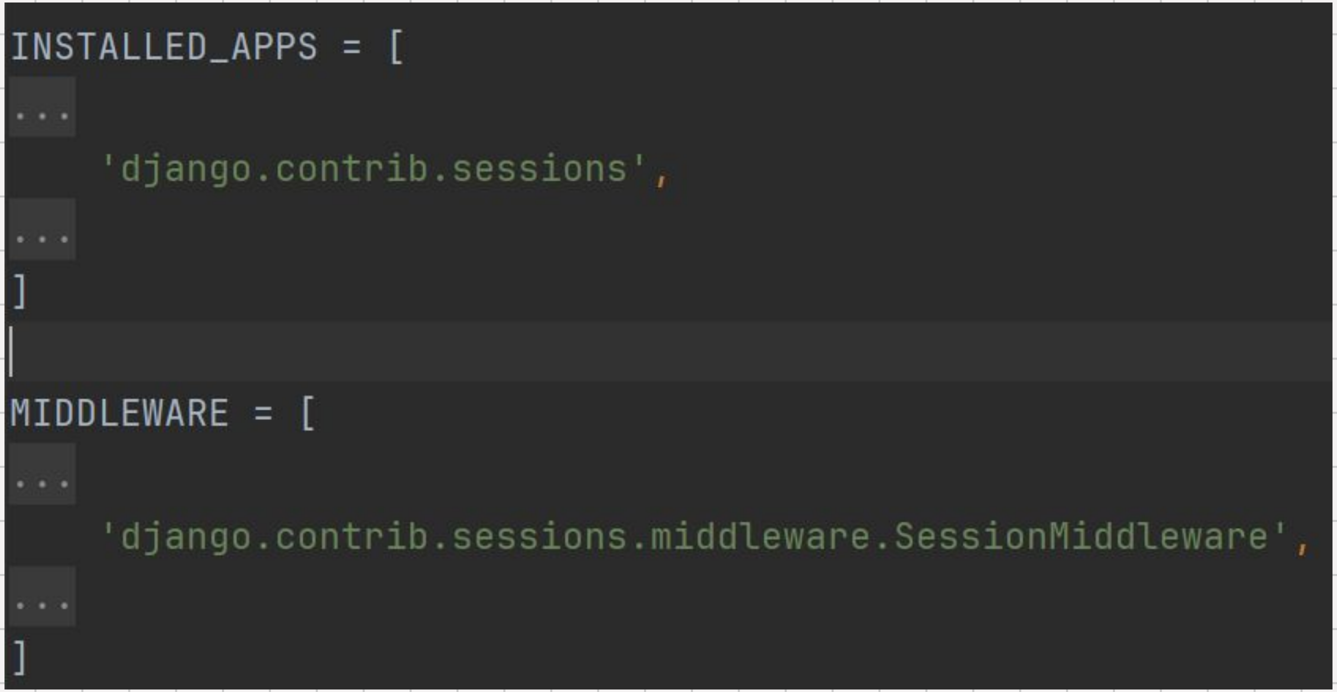
Session in Django

- Django uses a **cookie** containing a special session id to identify each browser and its associated session with the site.
- The actual session data is stored in the **site database** by default.
- You can configure Django to store the session data in other places (cache, files, "secure" cookies).
- But the default location is a good and relatively secure option.

Enabling Sessions

- The configuration is set up in the **INSTALLED_APPS** and **MIDDLEWARE** sections of the project setting file.
- You can access the session attribute within a view from the **request parameter** (the HttpRequest).
- This session attribute represents the specific connection to the current user.
- To be more precise, the connection to the current browser, as identified by the session id in the browser's cookie for this site).

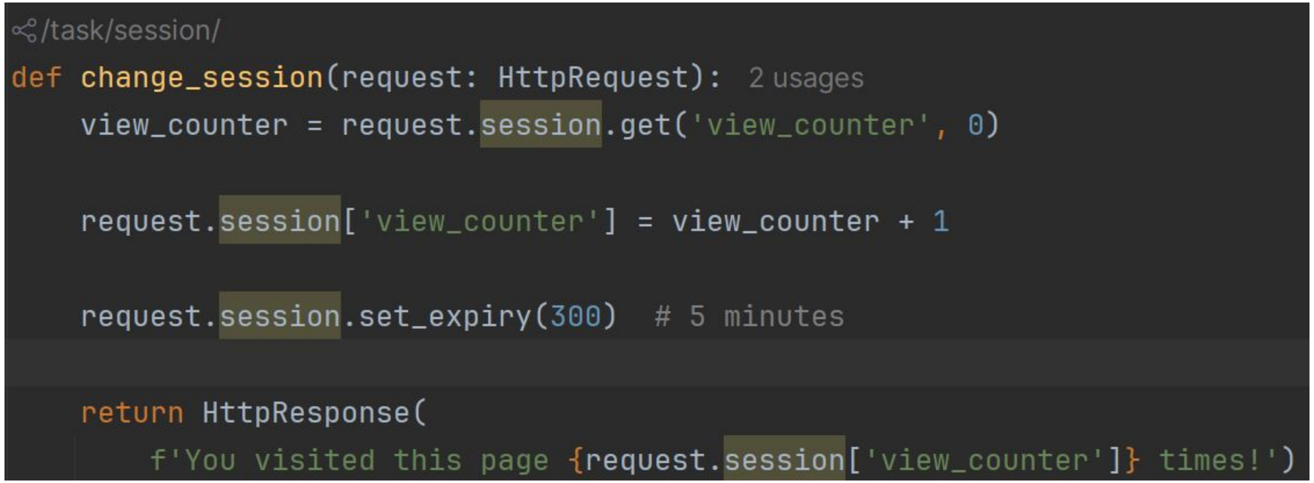
Enabling Sessions (cont.)



```
INSTALLED_APPS = [
...
    'django.contrib.sessions',
...
]

MIDDLEWARE = [
...
    'django.contrib.sessions.middleware.SessionMiddleware',
...
]
```

Saving Session Data

- The session attribute is a **dictionary-like** object that you can read and write as many times as you like in your view, modifying it as wished.



```
/task/session/
def change_session(request: HttpRequest):  2 usages
    view_counter = request.session.get('view_counter', 0)

    request.session['view_counter'] = view_counter + 1

    request.session.set_expiry(300)  # 5 minutes

    return HttpResponse(
        f'You visited this page {request.session['view_counter']} times!')
```

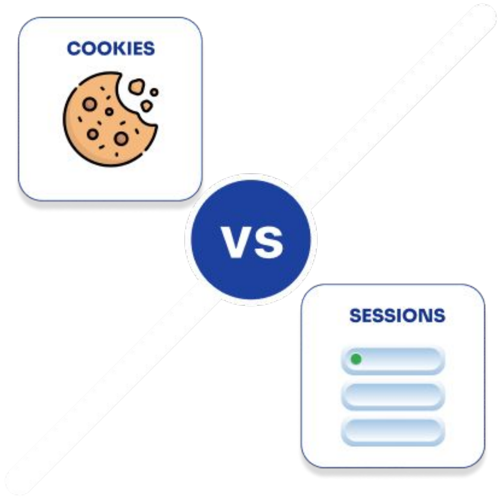
Session Use Cases

- User Authentication / Login Systems: To keep users logged in.
- Shopping Carts (E-commerce): Store items added to a shopping cart before checkout.
- User Preferences / Personalization: Language preference, theme, … .
- Tracking User Activity: Pages visited, last visited time, … .
- Rate Limiting / Flood Control: Track how often a user performs an action to prevent abuse.

Session vs Cookie

- Cookies:
  - Stored on client-side (browser)
  - Limited size (4KB typically)
  - Can be viewed/modified by user
  - Sent with every request

- Sessions:
  - Data stored on server-side
  - Only session ID sent to client
  - More secure for sensitive data
  - Larger storage capacity
  - Cannot be directly viewed by user



Django Built-in Auth.

- Django provides an **authentication** and **authorization** ("permission") system.
- System is built on top of the **session** framework.
- The authentication system is very flexible, and you can build up your URLs, forms, views, and templates from scratch.
- `django.contrib.auth` - Authentication framework
- `django.contrib.contenttypes` - Content type system

Authentication vs Authorization

- Authentication
  - Verifies **who the user is**
  - Uses credentials (password, biometrics, OTP)
  - Happens **first**
  - Example: Logging into an account

- Authorization
  - Determines **what the user can access**
  - Uses roles and permissions
  - Happens **after** authentication
  - Example: Accessing admin features

AUTHENTICATION
USERNAME
****
PASSWORD
****
LOGIN
WHO ARE YOU?
VS
AUTHORIZATION
USER SCOPES:
ABILITY:ONE
ABILITY:TWO
ABILITY:THREE
CAN YOU DO THAT?

Enabling Authentication

```
INSTALLED_APPS = [
...
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
...
]

MIDDLEWARE = [
...
    'django.contrib.sessions.middleware.SessionMiddleware',
    'django.contrib.auth.middleware.AuthenticationMiddleware',
...
]
```

Django User Model

- The **Django User Model** is Django’s built-in way to represent and manage users in a web application.

- It handles authentication, authorization, and user-related data such as usernames, passwords, and permissions.

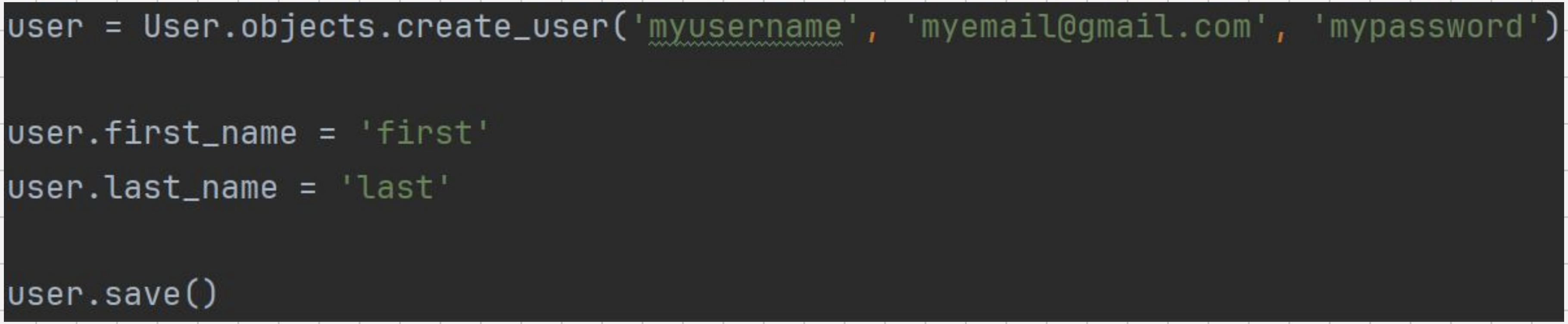
- `django.contrib.auth.models.User`

- This model is tightly integrated with Django’s authentication system, including:
  - Login / logout
  - Password hashing
  - Permissions and groups
  - Admin panel user management

Django User Model

- Default `User` model fields:
    - `username` - Unique username
    - `password` - Hashed password
    - `email` - Email address
    - `first_name` / `last_name` - Name fields
    - `is_active` - Account status
    - `is_staff` - Admin access
    - `is_superuser` - Full permissions
    - `date_joined` - Registration timestamp
    - `last_login` - Last login time

Create User

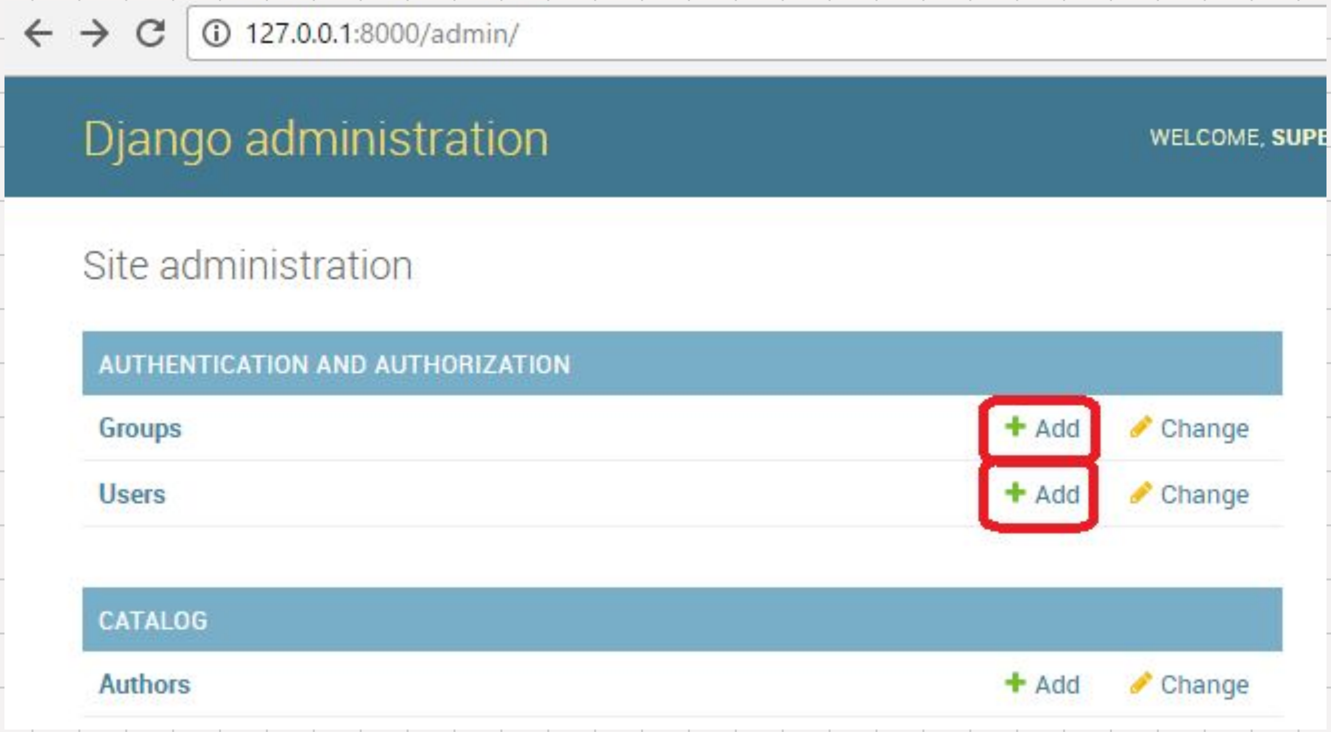


```python
user = User.objects.create_user('myusername', 'myemail@gmail.com', 'mypassword')

user.first_name = 'first'
user.last_name = 'last'

user.save()
```

Create User (cont.)

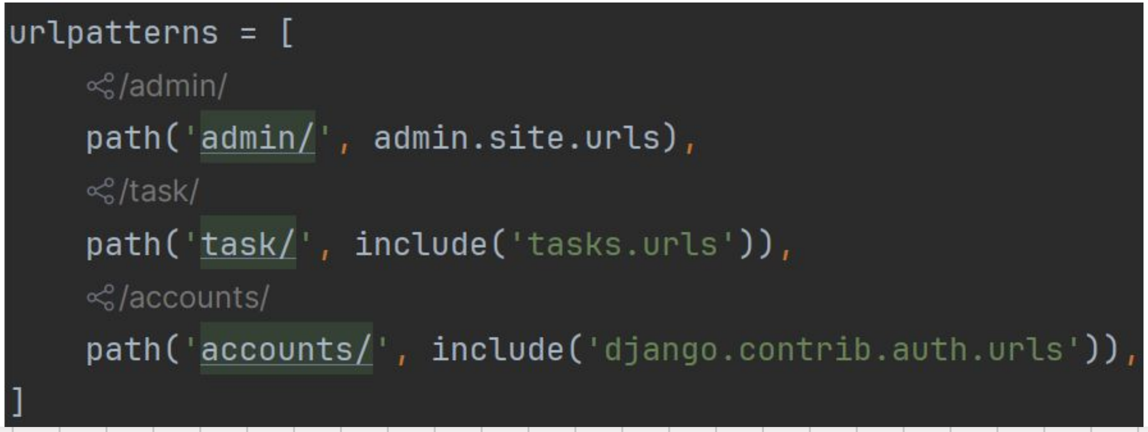
- `create_user()` is the recommended Django method for creating users safely.
- It is provided by Django’s UserManager and handles important security steps automatically.
- Always use `create_user()` OR `user.set_password()`
- Django hashes passwords automatically
- If the database is compromised, attackers cannot see real passwords and hashes cannot be reversed.

Explore Django User and Group Admin



Authentication Views

- Django provides almost everything you need to create authentication pages to handle login, log out, and password management "out of the box".
- This includes a URL mapper, views and forms, but it does not include the templates — we have to create our own!



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

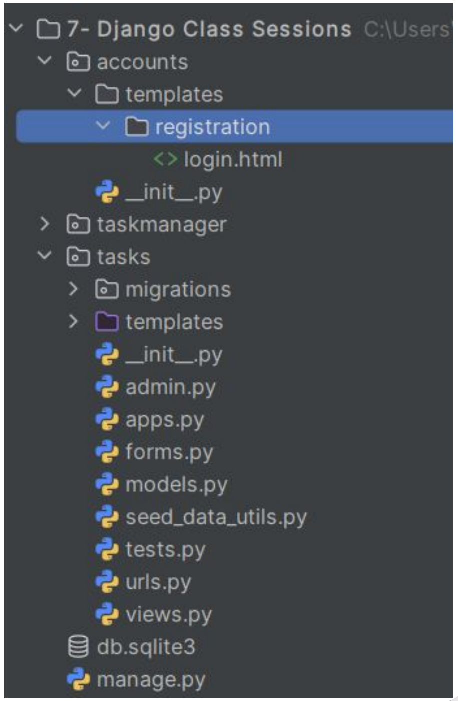
Authentication Views (cont.)

- 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']
```

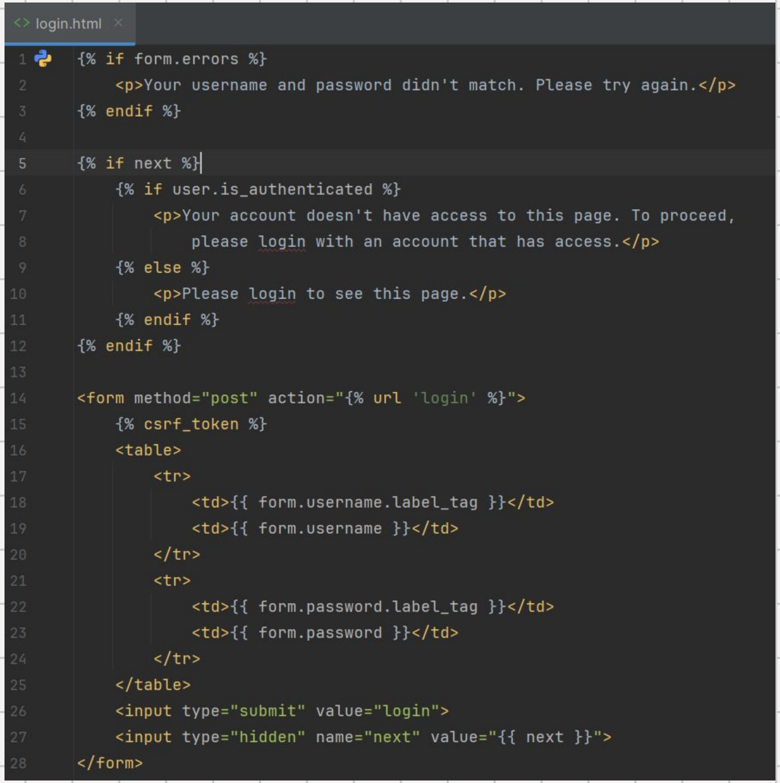
Authentication Templates

- The URLs (and implicitly, views) that we just added expect to find their associated templates In a directory **/registration/** somewhere in the **templates search path**.
- Navigate back to the login page (http://127.0.0.1:8000/accounts/login/)
- You should write other templates yourself!



```
7- Django Class Sessions C:\Users
  accounts
    templates
      registration
        login.html
    __init__.py
  taskmanager
  tasks
    migrations
    templates
    __init__.py
    admin.py
    apps.py
    forms.py
    models.py
    seed_data_utils.py
    tests.py
    urls.py
    views.py
  db.sqlite3
  manage.py
```

Simple Login Page



login.html

```
{% if form.errors %}
    <p>Your username and password didn't match. Please try again.</p>
{% endif %}

{% 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 %}

<form method="post" action="{% url 'login' %}">
    {% csrf_token %}
    <table>
        <tr>
            <td>{{ form.username.label_tag }}</td>
            <td>{{ form.username }}</td>
        </tr>
        <tr>
            <td>{{ form.password.label_tag }}</td>
            <td>{{ form.password }}</td>
        </tr>
    </table>
    <input type="submit" value="login">
    <input type="hidden" name="next" value="{{ next }}">
</form>
```

Checking Authenticated Users

- Selectively control content the user sees based on whether they are logged in or not.
- You can get information about the currently logged in user in templates with the `{{ user }}` template variable.
- This is added to the template context by default when you set up the project as we did in our skeleton.
- Typically you will first test against the `{{ user.is_authenticated }}` template variable.

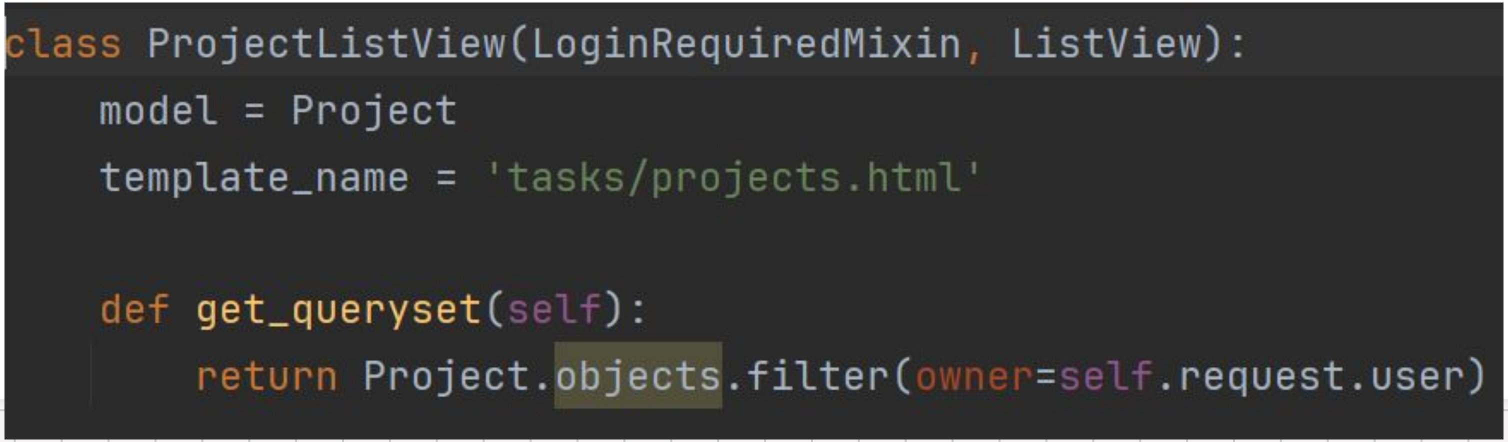
Testing in Views

- In function-based views, the easiest way to restrict access to your functions is to apply the login_required decorator to the view function.
  - `@login_required`
  - `def my_view(request):`

- Similarly, the easiest way to restrict access to logged-in users in your class-based views is to derive from LoginRequiredMixin.
  - `class MyView(LoginRequiredMixin, View):`

Filter User’s Objects

- We re-implement `get_queryset()` as shown to filter objects specific to the logged-in user.
- `self.request.user` is used to tie objects to the currently authenticated user.
- For security, `LoginRequiredMixin` ensures only logged-in users can access the view.



```python
class ProjectListView(LoginRequiredMixin, ListView):
    model = Project
    template_name = 'tasks/projects.html'

    def get_queryset(self):
        return Project.objects.filter(owner=self.request.user)
```

Django Permissions

- A Django permission is a rule that defines what actions a user is allowed to perform on a model or in the system.
- Permissions are a core part of authorization (not authentication).
- For each Django model, Django automatically creates four default permissions:

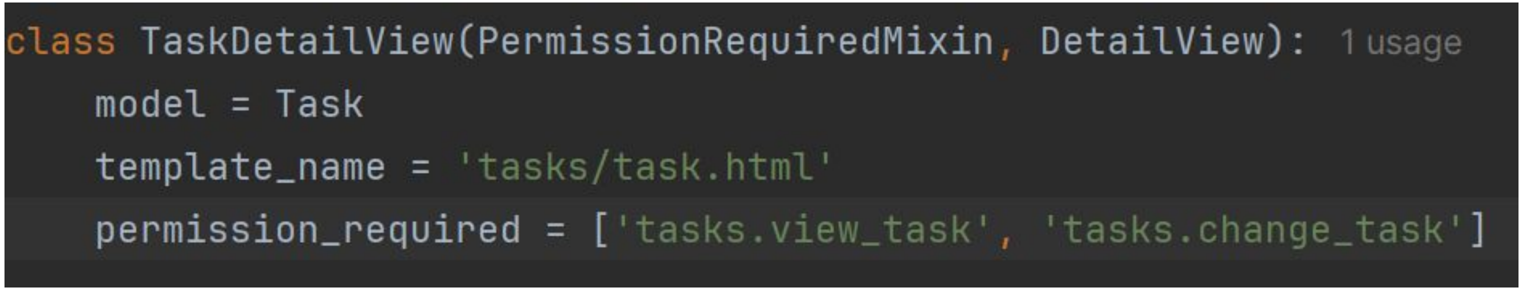
Permission	**Meaning**
`add_modelname`	Can create objects
`change_modelname`	Can edit objects
`delete_modelname`	Can delete objects
`view_modelname`	Can view objects

Django Group & Permission

- A Group is a collection of permissions.
  - Example: Editors group → add, change, view articles
- Instead of assigning permissions to users one by one, you assign them to a group, and users inherit all group permissions.
- Permissions are associated with models and define the operations that can be performed on a model instance by a user who has the permission.

Checking Permissions

- Permissions can be tested in function view using the `permission_required` decorator.
- Or in a class-based view using the `PermissionRequiredMixin`.
- You might reasonably have to add multiple permissions.



```
class TaskDetailView(PermissionRequiredMixin, DetailView):  1 usage
    model = Task
    template_name = 'tasks/task.html'
    permission_required = ['tasks.view_task', 'tasks.change_task']
```