Django Authentication System

COMP 8347

Slides prepared by Dr. Arunita Jaekel

arunita@uwindsor.ca

Django Authentication

- Topics
  - Introduction
  - Web Authentication
    - Login/Logout
    - Limiting access
  - Permissions and Authorization
    - Groups
    - Default permissions

Authentication System

- Django's authentication system consists of:
  - ***User*** objects
  - A configurable password hashing system
  - Forms and view tools for logging in users, or restricting content.
  - **Permissions:** Binary (yes/no) flags designating whether a user may perform a certain task.
  - **Groups**: A generic way of applying labels and permissions to more than one user.

Installation

- Add these 2 items in INSTALLED_APPS setting:
  - 'django.contrib.auth' : contains the core of the authentication framework, and its default models.
  - 'django.contrib.contenttypes': allows permissions to be associated with models you create.
- Add these 2 items in MIDDLEWARE_CLASSES setting:
  - SessionMiddleware: manages sessions across requests.
  - AuthenticationMiddleware: associates users with requests using sessions.
- By default: already included in settings.py.

User Objects

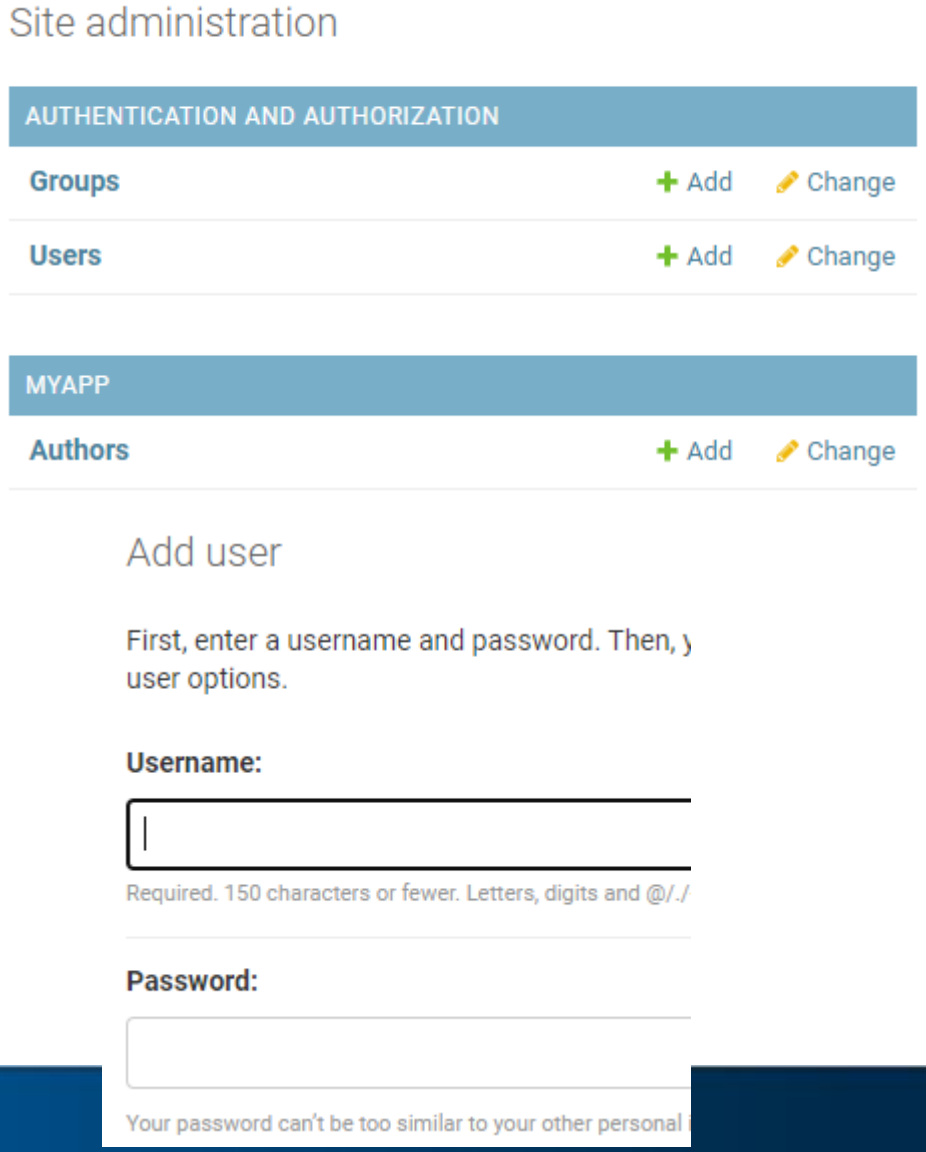
- User objects are the core of the authentication system.
  - Typically represent people interacting with your site.
  - Used to enable things like restricting access, registering user profiles etc.
  - Only one class of user exists in Django's authentication framework
    - Different user types e.g. 'superusers' or admin 'staff' users are just user objects with special attributes set
      - not different classes of user objects.

User Attributes

- The primary attributes of the default user are:
  - ***Username***: Required. 30 characters or fewer.
    - May contain alphanumeric, _, @, +, . and - characters.
  - *first_name*: Optional. 30 characters or fewer.
  - *last_name*: Optional. 30 characters or fewer.
  - *Email*: Optional. Email address.
  - ***Password***: Required.
    - A hash of, and metadata about, the password.
    - Django doesn't store the raw password. Raw passwords can be arbitrarily long and can contain any character.

Using Admin Interface

- Admin module can be used to view and manage users, groups, and permissions.
  - Both django.contrib.admin and django.contrib.auth must be installed.
  - The “Add user” admin page requires you to choose a username and password before allowing you to edit the rest of the user’s fields.
  - User passwords are **not** displayed in the admin (nor stored in the database).
    - a link to a password change form allows admins to change user passwords



Creating Users

- Use create_user() helper function:

```
from django.contrib.auth.models import User
user = User.objects.create_user('john',
   'lennon@thebeatles.com', 'johnpassword')
# At this point, user is a User object that has
# already been saved to the database. You can
# continue to change its attributes.
user.last_name = 'Lennon'
 user.save()
```

Changing Passwords

- Django does not store raw (clear text) passwords on the user model,
  - It only stores a hash.
  - Do **not** manipulate the password attribute of the user directly.
    - **user.password = 'new password' # Don't do this!**
  - Passwords can be changed using **set_password()**

```
from django.contrib.auth.models import User
u = User.objects.get(username='john')
u.set_password('new password')
u.save()
```

Authenticating Users

- ***authenticate()***: Takes credentials in the form of keyword arguments:
  - For the default configuration this is username and password
    - Returns a User object if the password is valid for the given username.
    - Returns None if password is invalid.

```
from django.contrib.auth import authenticate
 user = authenticate(username='john', password='secret')
 if user is not None: # password verified for the user
     if user.is_active:
         print("User is valid, active and authenticated")
     else:
         print("The password is valid, but the account has been disabled!")
else: # unable to verify the username and password
     print("Username and password did not match.")
```

Login

- Each **request** object provides a request.user attribute, which represents the current user.
  - If the current user has not logged in, this attribute will be set to an instance of AnonymousUser; otherwise it will be an instance of User.
  - Use is_authenticated() to differentiate between the two.
  - Example: **if request.user.is_authenticated():**

    # Do something for authenticated users.

- ***login()*** *function*: used to attach an *authenticated* user to the current session.
  - It takes an HttpRequest object and a User object.
    - Associates the **user** with the current **request** object
  - Saves the user's ID in the session, using Django's session framework.
  - Any data set during the anonymous session is retained in the session after a user logs in.
  - ***login()*** function can be called from a view.
- To manually log in a user call authenticate() before you call login().

Example

```
from django.contrib.auth import authenticate, login

def my_view(request):
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(username=username, password=password)
    if user is not None:
        if user.is_active:
            login(request, user)
                # Redirect to a success page.
        else:
                # Return a 'disabled account' error message
    else:
        # Return an 'invalid login' error message.
```

Logout

- Use django.contrib.auth.logout() within your view.
  - It takes an HttpRequest object and has no return value.
  - logout() doesn't throw any errors if the user wasn't logged in.
  - Cleans out the session data for the current request

```
from django.contrib.auth import logout

def logout_view(request):
    logout(request)
    # Redirect to a success page.
```

Login_required Decorator

- login_required() does the following:
  - If the user isn't logged in, redirect to settings.LOGIN_URL
    - Passing the current absolute path in the query string. Example: /accounts/login/?next=/polls/3/.
    - By default, the path that the user should be redirected to upon successful authentication is stored in a query string parameter called "next".
  - If the user is logged in, execute the view normally.
    - The view code is free to assume the user is logged in.

```
from django.contrib.auth.decorators import login_required
@login_required
def my_view(request):
    ...
```

Default Permissions

- 3 default permissions created for each Django model defined in one of your installed apps :
  - *Add*: Access to view the “add” form and add an object
    - limited to users with “add” permission for that type of object.
    - Example: user.has_perm(‘myapp.add_book’)
  - *Change*: Access to view the change list, view the “change” form and change an object
    - limited to users with the “change” permission for that type of object. Example: user.has_perm(‘myapp.change_book’)
  - *Delete*: Access to delete an object
    - limited to users with the “delete” permission for that type of object. Example: user.has_perm(‘myapp.delete_book’)
  - Permissions can be set not only per type of object, but also per specific object instance.

Custom Permissions

- You can create permissions programatically (in views.py or in python django console).

```
from myapp.models import Libuser
from django.contrib.auth.models import Group, Permission
from django.contrib.contenttypes.models import ContentType

content_type = ContentType.objects.get_for_model(Libuser)
perm1 = Permission.objects.create(codename='can_search',
                                  name='Can Search for Libitems',
                                  content_type=content_type)
```

  - Once created, the permission can be assigned to a User via its user_permissions attribute or to a Group via its permissions attribute.
  - The has_perm() method, permission names take the form "<app label>.<permission codename>"
    - e.g. myapp.can_search for a permission on a model in the myapp APP.

Check and Change User Permissions

- User objects have many-to-many field: **user_permissions**

  myuser = User.objects.get(pk=1)

  myuser**.user_permissions.set(**[permission_list])

  myuser**.**user_permissions**.**add(perm1, perm2, …)

  myuser**.**user_permissions**.**remove(perm1, perm2, **...**)

  myuser**.**user_permissions**.**clear()

Using Permissions in Your Code

- Restrict certain actions to specific users:
  - Check if user has permission before executing code. Example, if only certain users can search the library.

```
def searchlib(request):
    if request.user.has_perm('libapp.can_search'):
        # Code to process search request goes here
        #   …
    else:
        return HttpResponse('You do not have permission to search!')
```

Groups

- Groups allow you to apply permissions to a group of users.
  - A user in a group automatically has the permissions granted to that group.
  - Also a convenient way to categorize users to give them some label, or extended functionality.
- User objects have many-to-many field: **groups**

```
myuser = User.objects.get(pk=1)
 myuser.groups.set([group_list])
 myuser.groups.add(group1, group,2 ...)
 myuser.groups.remove(group1, group2, ...)
 myuser.groups.clear()
# Add group permissions
group = Group.objects.get(name='wizard')
group.permissions.add(permission)
```

Summary

- Authentication
  - User objects
  - Authenticate(), login() and logout()
  - Permissions and Authorization
    - Groups
    - Default permissions

- [1] https://docs.djangoproject.com/en/3.0/topics/auth/