User Model and Password Management

Groups in Django

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.org

Have a Question?

Software University

sli.do

#python-web

Table of Contents

The User in Django

The User

- A user is an **individual** **accessing** a **website** via a **web browser**
- Users have the ability to **interact** with the site, facilitating **functionalities** such as **access restriction**, user profile **registration**, and **association** of **content** with **creators**
- In the Django framework, **user objects** **form** the **cornerstone** of the **authentication** **system**
  - Serving as central entities for **managing** **user-related** **operations** and **authentication** **processes**

The User Model

- In Django's **authentication** framework, only **one class** of **user** exists
  - `'superusers'` or admin `'staff'` users are essentially **user objects** with **specific attributes** set to confer **special privileges**

```
from django.contrib.auth import get_user_model
UserModel = get_user_model()
```

- The Django **user model** inherits from `AbstractUser`, which itself inherits from `AbstractBaseUser` and includes the `PermissionsMixin`

The User Fields

- The **primary fields** of the default user are:
  - `username` - required, 150 characters or fewer
  - `password` - required, Django doesn't store the raw password
  - `email` - optional
  - `first_name` - optional, 150 characters or fewer
  - `last_name` - optional, 150 characters or fewer

The User Fields

- Other **fields** of the default user are:
  - `groups` - many-to-many relationship to Group
  - `user_permissions` - many-to-many relationship to Permission
  - `is_staff` - Boolean
  - `is_active` - Boolean
  - `is_superuser` - Boolean
  - `last_login` - date/time of the user's last login
  - `date_joined` - set to the current date/time by default

The User Attributes

- In Django's user model, there are **two attributes**:
  - `is_authenticated`
    - A read-only attribute that is always `True` for users who have been authenticated
  - `is_anonymous`
    - A read-only attribute that is always `False`
    - This is provided for **consistency** but **isn't practically used**

***Note:** It's recommended to use the `is_authenticated` **attribute** for checking **whether a user is authenticated**

The User Methods Examples

- `get_username()`
  - Returns the **username** for the **user**
  - Use this method **instead of referencing** the username attribute directly
- `get_full_name()`
  - Returns `"{first_name} {last_name}"`
- `get_short_name()`
  - Returns `first_name` only

All user methods: *https://docs.djangoproject.com/en/4.2/ref/contrib/auth/#methods*

The AnonymousUser Class

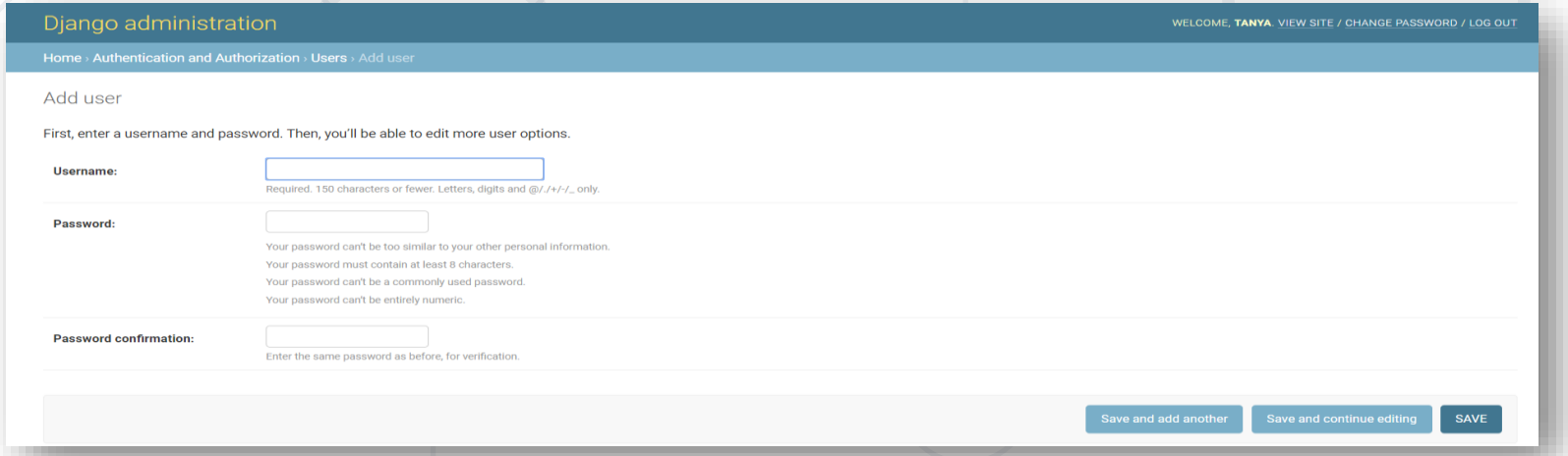
- Implements the **User interface** but with some **differences**:
  - `id` is always **None**
  - `username` is always an **empty string** ('')
  - `is_staff` and `is_superuser` are always `False`
  - `is_authenticated` always returns `False`
- `AnonymousUser` **objects** are typically used to represent **unauthenticated users** in web requests

Create User

- To create a new User, we can use the **built-in manager method** provided by the User model **`create_user()`**

```
from django.contrib.auth import get_user_model
UserModel = get_user_model()
new_user = UserModel.objects.create_user('peter', 'peter@gmail.com',
'peterpass')
```

- Or we can use the Django Admin Site



Authenticate Users

- We can use the **authenticate()** function to **verify** credentials (for login)
- If the credentials are **not valid**, **None** is returned

```
from django.contrib.auth import authenticate

user = authenticate(username='peter', password='peterpass')
if user:
    # Credentials are valid
else:
    # Credentials are not valid
```

***Note:** This is a **low-level way** to authenticate a set of credentials

Authentication in Web Requests

- In Django, the **request.user** attribute is used to **represent** the **current user** for each request
  - If the current **user is logged in**, it is **set** to an **instance** of **User**
  - Otherwise, it is **set** to an **instance** of **AnonymousUser**

```
if request.user.is_authenticated:
    # Do something for authenticated users
    ...
else:
    # Do something for anonymous users
    ...
```

Login

- To log a user in from a view in Django, you can use the **login()** function
- It takes an **HttpRequest object** and a **User object** as parameters

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

def index(request):
    UserModel = get_user_model()
    some_user = UserModel.objects.get(username='Peter')
    print(request.user.__class__.__name__) # AnonymousUser
    login(request, some_user)
    print(request.user.__class__.__name__) # User
    return render(request, 'home_page.html')
```

Logout

- To log out a user who has been logged in using **login()** function, you can use the **logout()** function within the view
- It takes an **HttpRequest object** as a parameter and **does not return** anything

```
from django.contrib.auth import logout

def logout_page(request):
    print(request.user.__class__.__name__) # User
    logout(request)
    print(request.user.__class__.__name__) # AnonymousUser
    return render(request, 'logout_page.html')
```

Registration

Built-in User Registration Form

Registration

- Django provides a **built-in** user **registration form** called `UserCreationForm`
- This form is **connected** to the **pre-built** `User` model
- It includes **three** fields
  - `username`, `password1`, and `password2`
- This form **simplifies** the **process** of **user registration** by **encapsulating** the necessary fields and validation logic
- Developers can use it **directly** or **customize** it according to their project requirements

Using UserCreationForm

- Import the **form** and create a **view**
  - You have the flexibility of using a **CBV** or **FBV**

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

def create_user_view(request):
    form = UserCreationForm(request.POST or None)
    if request.method == 'POST':
        ...
    elif request.method == 'GET':
        ...
```

Using UserCreationForm

- Create a **path** and a **template** as usual
- Start the development server

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Password:

- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

Create

Custom Registration Form

- **More** User model **fields** could be used in the **registration form**

```
from django.contrib.auth.forms import UserCreationForm
from django.contrib.auth import get_user_model

UserModel = get_user_model()

class CustomRegistrationForm(UserCreationForm):
    email = models.EmailField(required=True)
    ...
    class Meta:
        model = UserModel
        fields = ('username', 'email', 'first_name', 'last_name',)

    def save(self, commit=True):
        # clean the data and save the user
```

Custom Registration Form

- The **new fields** are now **visible** in the **form**

Username: Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.

Email address:

First name:

Last name:

Password:

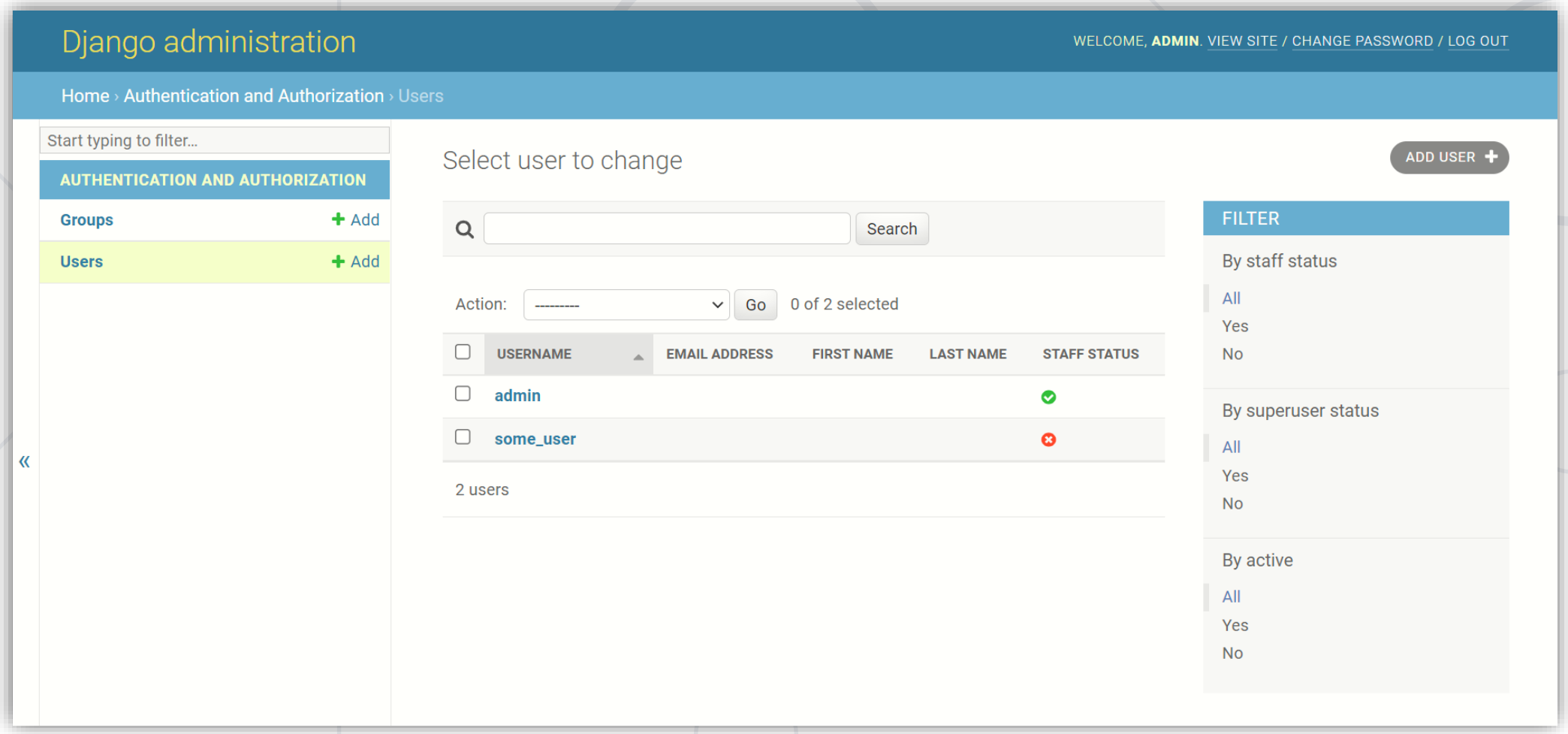
- Your password can't be too similar to your other personal information.
- Your password must contain at least 8 characters.
- Your password can't be a commonly used password.
- Your password can't be entirely numeric.

Password confirmation: Enter the same password as before, for verification.

Create

Users in the Admin Site

- View the registered users in the **Admin Site**



Login and Logout

Built-in Class-based Views

Built-in Login/Logout Views

- Once a user is registered, **ensuring** their ability to **log in** and **out** of the site is crucial
- Django **simplifies** this process with **built-in class-based views**
  - `LoginView` and `LogoutView`
- They leverage the **built-in authentication** forms, but **customization** is possible by passing your own forms
- Django **doesn't** supply **default templates** for the authentication views

Creating a Login System

- To fully utilize the Django **authentication system**, **incorporate** the provided **URLconf** into your own

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

- Alternatively, you can **directly** use specific **authentication views**

```
from django.contrib.auth import views

urlpatterns = [
    path('sign-in/', views.LoginView.as_view()),
]
```

Creating a Login System

- **Extending** or **customizing** authentication views in Django is straightforward by **subclassing** the **relevant views**

```
from django.contrib.auth.views import LoginView

class CustomLoginView(LoginView):
    # Extend or customize the view
```

- The **default redirect** after login is to `'/accounts/profile/'` **URL**
  - Modify the default behavior by adding `LOGIN_REDIRECT_URL` to your `settings.py` file

Creating a Login System

- When using the `LoginView`, the created **template** receives several **context variables**
  - `form`
    - A **Form object** representing the `AuthenticationForm`
  - `next`
    - The **URL** to **redirect** to after a **successful login**
  - `site`
    - Represents the `Site` **object** associated with the **current site**
  - `site_name`
    - Provides a convenient alias for the site `name` attribute

LoginRequiredMixin

- **LoginRequiredMixin** is a Django **class-based view mixin** that **enforces login requirements** for views
- It **ensures** that **only authenticated** users can access certain view

```
from django.contrib.auth.mixins import LoginRequiredMixin
from django.views.generic import ListView

class MyProtectedListView(LoginRequiredMixin, ListView):
    # Your view logic here
```

Creating a Logout System

- `LogoutView` in Django is **similar** to `LoginView` in terms of its usage for handling **user logout**
- For **customizing** the **post-logout redirection**, set the `LOGOUT_REDIRECT_URL` in your `settings.py`
- Template **context variables**
  - `title` - A string indicating the status ("Logged out" in this case)
  - `site` - The current `Site` object based on the `SITE_ID` setting
  - `site_name` - An alias for `site.name`

LogoutView Usage Example

- In your **urls.py**, you can **include** the **LogoutView** like this:

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

urlpatterns = [
    ...
    path('logout/', auth_views.LogoutView.as_view(),
name='logout'),
]
```

logout_then_login

- The `logout_then_login` view in Django **logs** a user **out** and then **redirects** them to the **login page**
- `login_url` – An optional **argument**, and if **not provided**, it **defaults** to the value specified in `settings.LOGIN_URL`

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

urlpatterns = [
    ...
    # Logout then login using the built-in view
    path('logout-then-login/', auth_views.logout_then_login,
name='logout_then_login'),
]
```

redirect_to_login

- The `redirect_to_login` view in Django is responsible for
  - **redirecting** users to the **login** page
  - and **subsequently back** to **another URL after** a **successful login**
- Arguments
  - `next` - This argument is **required** and specifies the **URL** to which the user should be **redirected after** a **successful login**
  - `login_url` - This is an **optional** argument and **defaults** to `LOGIN_URL` if not supplied
  - `redirect_field_name` - Another **optional** argument, this parameter is the **name** of a **GET** field containing the **URL** to **redirect** to **after** a **logout**, allowing **overriding** the `next` parameter if a specific **GET parameter** is passed

Password Management

Password Management

- Django provides a **robust** and **flexible** system for **managing user passwords**
- Following **best practices** in **password security** to **ensure** that sensitive information remains **protected**
- Using the **PBKDF2 algorithm** with a **SHA256 hash**
  - It requires massive amounts of computing time to break, making it **resistant** to **attacks**
- In summary, Django's approach to **password management** is **both secure** and **well-designed**

Hashing

- Storing **passwords** in a **hashed form** is a **crucial security** measure
  - This **prevents** storing **raw passwords** in the database, **mitigating** the **impact** of **potential data breaches**
- Hashing performs a **one-way transformation** on a password, **converting** the **original password** into **another string**, known as the **hashed password**
  - The **one-way nature** of this process **ensures** that it is computationally **infeasible** to **reverse** the **transformation** and retrieve the original password

Default PASSWORD_HASHERS

```
PASSWORD_HASHERS = [
 'django.contrib.auth.hashers.PBKDF2PasswordHasher',
 'django.contrib.auth.hashers.PBKDF2SHA1PasswordHasher',
 'django.contrib.auth.hashers.Argon2PasswordHasher',
 'django.contrib.auth.hashers.BCryptSHA256PasswordHasher',
 'django.contrib.auth.hashers.ScryptPasswordHasher',
]
```

Hashing While Testing

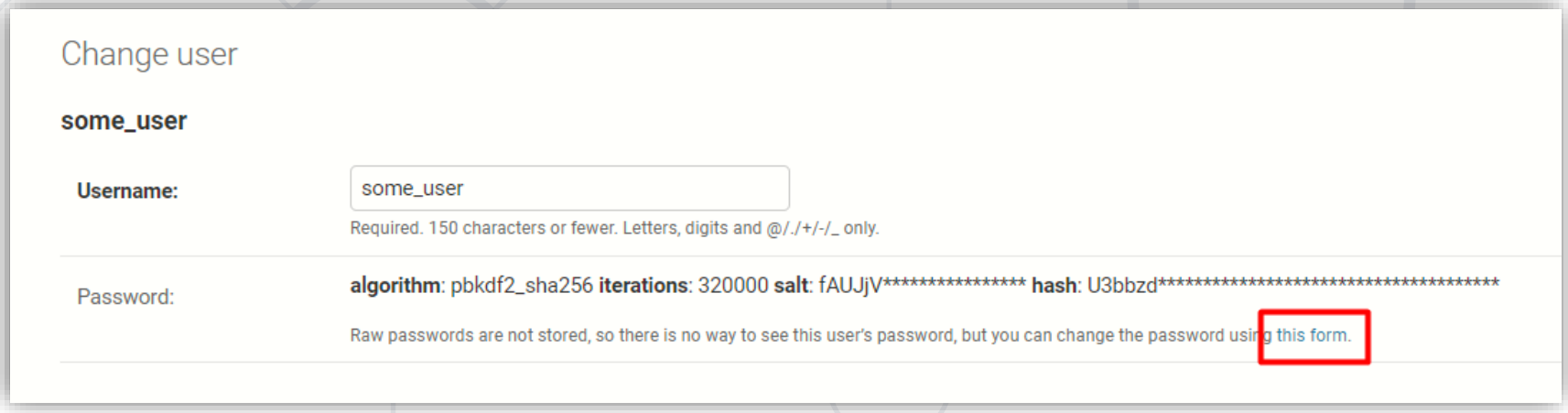
- The default password hasher is rather slow by design
- When authenticating **many users in tests**, you may want to use a custom settings file and set the **PASSWORD_HASHERS** setting to a **faster but less secure** hashing algorithm

Only for tests, not suitable for production!

```
PASSWORD_HASHERS = [
'django.contrib.auth.hashers.MD5PasswordHasher',
]
```

set_password Method

- `set_password(raw_password)`
  - A **method** provided by the **Django User model**
  - It **sets** the **user's password** to the **given raw string**, taking care of the **password hashing**, ensuring that the password is securely **hashed before** being **stored** in the database
- Set a password using the **admin page**



Other Password-related Methods

- `check_password(raw_password)`
  - Returns `True` if the given **raw string** is the **correct password** for the user (takes care of the password hashing)
- `set_unusable_password()`
  - Marks the user as having **no password** set (**not** an **empty** string)
- `has_usable_password()`
  - Returns `False` if `set_unusable_password()` has been called, indicating that the user **does not** have a **valid password** that can be used for **authentication**

Password-related Built-in Views

- Django auth system provides a **set** of **built-in views** to **handle** common **authentication-related** tasks
  - `PasswordChangeView`
  - `PasswordChangeDoneView`
  - `PasswordResetView`
  - `PasswordResetDoneView`
  - `PasswordResetConfirmView`
  - `PasswordResetCompleteView`

Documentation: *https://docs.djangoproject.com/en/4.2/topics/auth/default/#django.contrib.auth.views.PasswordChangeView*

Password-related Built-in Forms

- The Django authentication system provides a **set** of **built-in forms** that correspond to various **authentication-related** tasks
  - `PasswordChangeForm`
  - `PasswordResetForm`
  - `SetPasswordForm`
- These **forms** are designed to **simplify** the **process** of **implementing authentication features** in Django applications by providing **ready-to-use components** for common tasks

Documentation: *https://docs.djangoproject.com/en/4.2/topics/auth/default/#module-django.contrib.auth.forms*

Password Validation

- Django provides a **pluggable password validation system**, allowing developers to **customize** and **extend** password validation rules
- Several **built-in validators** are included in Django, but developers can **add their own** ones

```
AUTH_PASSWORD_VALIDATORS = [
    {'NAME': '...UserAttributeSimilarityValidator',},
    {'NAME': '...MinimumLengthValidator',},
    {'NAME': '...CommonPasswordValidator',},
    {'NAME':'...NumericPasswordValidator',},
]
```

Custom Password Validators

- When creating **custom validators** in Django, you **must** implement **two methods**
  - `validate(self, password, user=None)`
    - **Validates** a password
    - Returns `None` if the password is **valid**
    - Raises a `ValidationError` with an **error message** if the password is **not valid**
  - `get_help_text(self)`
    - Provides a **help text** to explain the **requirements** to the user

Groups

Default Permissions

- Four **default permissions**
  - **add**, **change**, **delete**, and **view**
- They are **automatically created** for each Django **model** defined in the installed applications

```
user = UserModel.objects.get(username='admin')
user.has_perm('main_app.add_employee')    # True
user.has_perm('main_app.change_employee') # True
user.has_perm('main_app.delete_employee') # True
user.has_perm('main_app.view_employee')   # True
```

Django Permissions in Groups

- Instead of **managing permissions** for each user individually, Django provides the **concept** of **groups**
- We can create a **group** (e.g., "**Users**") and **assign** the **relevant permissions** to that **group**
- Each **new user** can then be **added** to this **group**, **inheriting** the **permissions** associated with the **group**
- This approach **simplifies** the **management** of **permissions**, especially in scenarios where **multiple users** need the **same set** of **permissions**

Example: Permissions in Groups

Add group

Name: Users

Permissions:

Available permissions

Filter

admin | log entry | Can add log entry
admin | log entry | Can change log entry
admin | log entry | Can delete log entry
admin | log entry | Can view log entry
auth | group | Can add group
auth | group | Can change group
auth | group | Can delete group
auth | group | Can view group
auth | permission | Can add permission
auth | permission | Can change permission
auth | permission | Can delete permission
auth | permission | Can view permission
auth | user | Can add user

Chosen permissions

app | article | Can add article
app | article | Can change article
app | article | Can delete article
app | article | Can view article

Choose all

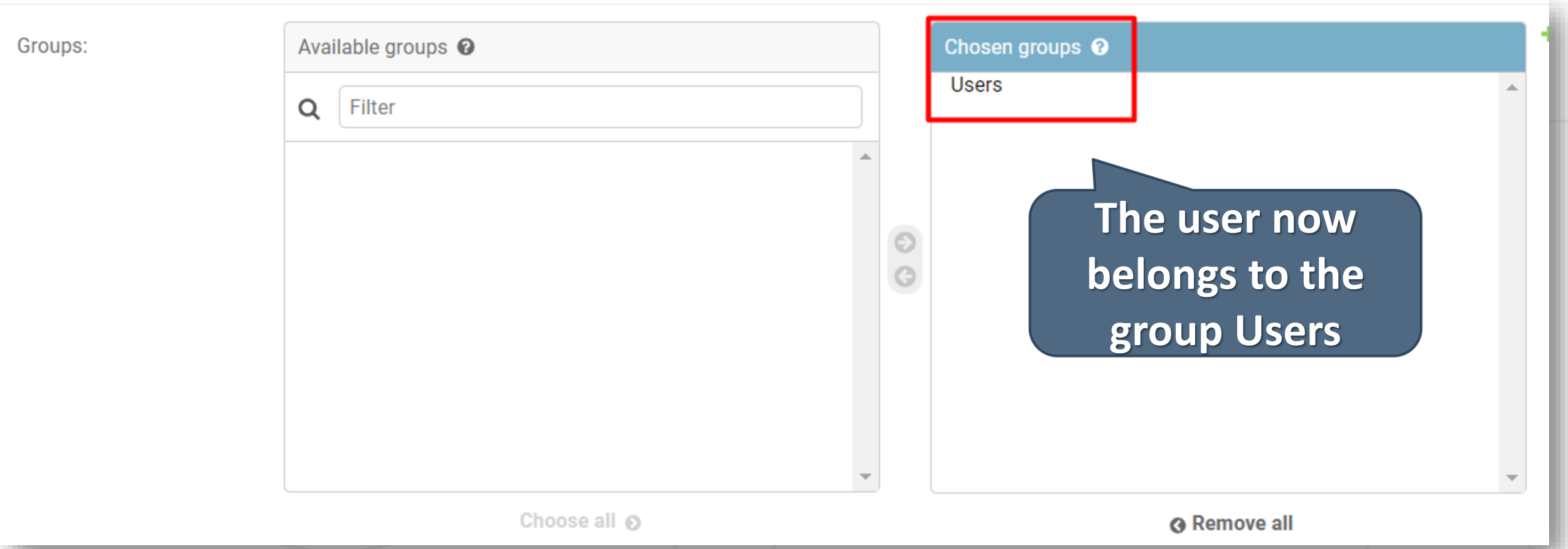
Remove all

Hold down "Control", or "Command" on a Mac, to select more than one.

Save and add another | Save and continue editing | SAVE

Assigning permissions to the group

Example: User in Users Group



Using Built-In Decorators

- Django provides **built-in decorators** that allow you to easily **enforce permission control** in your views

```python
from django.shortcuts import render
from django.contrib.auth.decorators import login_required
from app.forms.login import LoginForm

# Create your views here.
@login_required(login_url='login')
def index(req):
    return render(req, 'index.html')

def login(req):
    form = LoginForm()
```

Ensures that the user is authenticated before accessing the view

Creating Custom Decorators

- In Django, you can create **custom decorators** to **add** your **own logic** for **permission validation**
- To do this, you typically create a **`decorators.py`** file in your app and **define** your **custom decorator function(s)** there
- For example, if you want to create a **custom decorator** to **ensure** that a user has **specific permission** (e.g., belongs to the "Users" group), you could **define** your own decorator

More about permissions: *https://docs.djangoproject.com/en/4.2/topics/auth/default/#limiting-access-to-logged-in-users*

Example: Creating Custom Decorators

decorators.py

app > decorators.py

```python
from django.http import HttpResponse
from django.shortcuts import render

def allowed_groups(allowed_roles=[]):
    def decorator(view_func):
        def wrapper(request, *args, **kwargs):
            group = None
            if request.user.groups.exists():
                group = request.user.groups.all()[0].name
            if group in allowed_roles:
                return view_func(request, *args, **kwargs)
            else:
                return HttpResponse('You are not allowed to view the articles')
        return wrapper
    return decorator
```

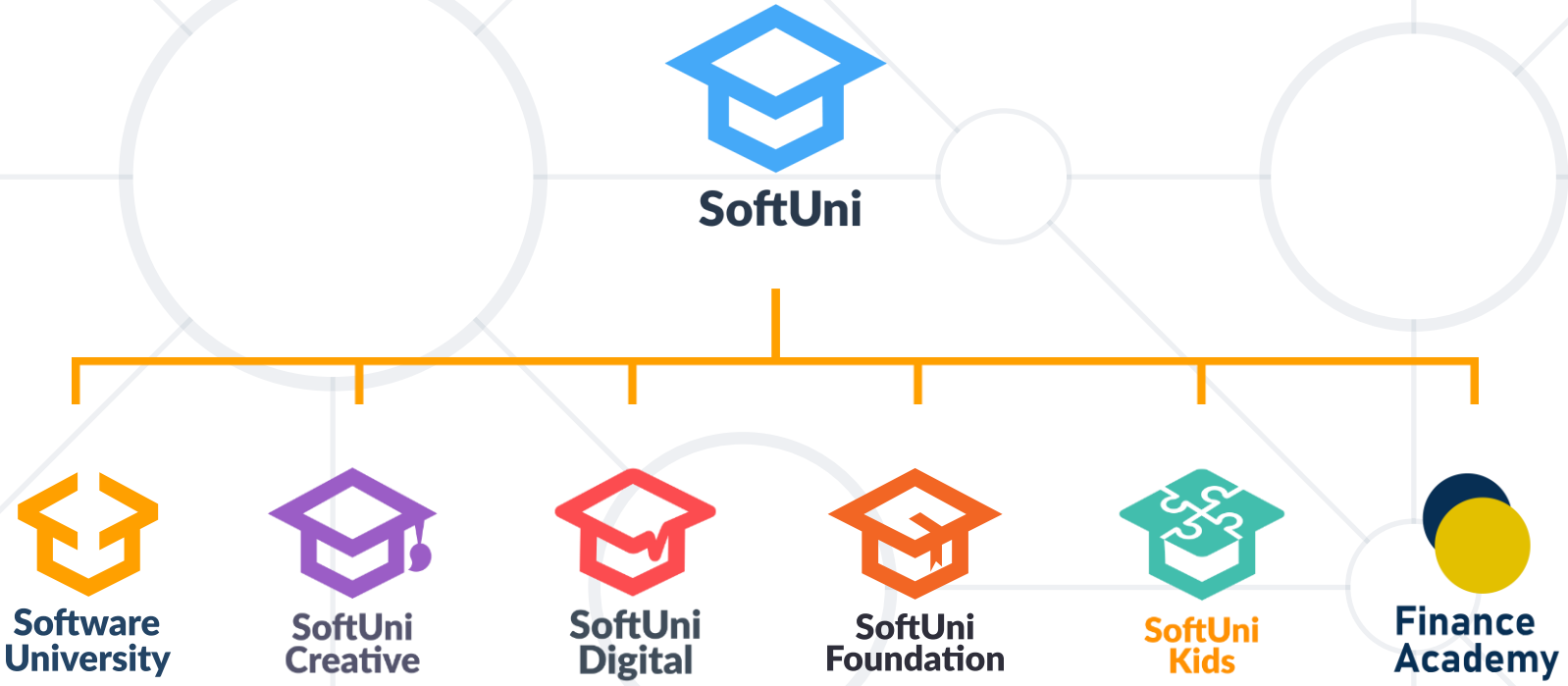
```python
from .decorators import allowed_groups

# Create your views here.
@allowed_groups(['Users'])
def index(req):
    articles = Article.objects.all()
    return render(req, 'index.html', {'articles': articles})
```

Summary

- User Model
- **Registration Form**
- **Login** and **Logout**
- **Password** Management
- **Groups**
  - **Permissions** in Groups

Questions?



SoftUni Diamond Partners

Software University

SmartIT

Coca-Cola HBC Bulgaria

SOFTWARE GROUP

DXC TECHNOLOGY

createX

Postbank
Решения за твоето утре

SUPER HOSTING .BG

DRAFT KINGS

VIVACOM

INDEAVR
Serving the high achievers

AMBITIONED

PHAR VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg, softuni.org
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

License

- This course (slides, examples, demos, exercises, homework, documents, videos, and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – https://softuni.org
- © Software University – https://softuni.bg