Competency Based Learning Material (CBLM)

Web Application Development with Python

Level-4

Module: Applying User Management

Code: CBLM- OU-ICT-WADP-04-L4-V1

National Skills Development Authority
Chief Advisor's Office
Government of the People's Republic of Bangladesh

Copyright

National Skills Development Authority
Chief Advisor's Office
Level1-11, Biniyog Bhaban,
E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.
Emailec@nsda.gov.bd
Websitewww.nsda.gov.bd.
National Skills Portalhttp:\\skillsportal.gov.bd

This Competency Based Learning Materials (CBLM) on "**Apply User Management**" under the **Web Application Development with Python , Level-4** qualification is developed based on the national competency standard approved by National Skills Development Authority (NSDA)

This document is to be used as a key reference point by the competency-based learning materials developers, teachers/trainers/assessors as a base on which to build instructional activities.

National Skills Development Authority (NSDA) is the owner of this document. Other interested parties must obtain written permission from NSDA for reproduction of information in any manner, in whole or in part, of this Competency Standard, in English or other language.

It serves as the document for providing training consistent with the requirements of industry in order to meet the qualification of individuals who graduated through the established standard via competency-based assessment for a relevant job.

This document has been developed by NSDA in association with industry representatives, academia, related specialist, trainer, and related employee. Public and private institutions may use the information contained in this CBLM for activities benefitting Bangladesh.

Approved by the Authority ……… meeting held on ....

How to use this Competency Based Learning Material (CBLM)

The module contains training materials and activities for you to complete. These activities may be completed as part of structured classroom activities or you may be required you to work at your own pace. These activities will ask you to complete associated learning and practice activities in order to gain knowledge and skills you need to achieve the learning outcomes.

1. Review the **Learning Activity** page to understand the sequence of learning activities you will undergo. This page will serve as your road map towards the achievement of competence.

2. Read the **Information sheet s.** This will give you an understanding of the jobs or tasks you are going to learn how to do. Once you have finished reading the **Information sheet s** complete the questions in the **Self-Check.**

3. **Self-**Checks are found after each **Information sheet** . **Self-Checks** are designed to help you know how you are progressing. If you are unable to answer the questions in the **Self-Check** you will need to re-read the relevant **Information sheet** . Once you have completed all the questions check your answers by reading the relevant **Answer Keys** found at the end of this module.

4. Next move on to the **Job Sheets. Job Sheets** provide detailed information about *how to do the job* you are being trained in. Some **Job Sheets** will also have a series of **Activity Sheets**. These sheets have been designed to introduce you to the job step by step. This is where you will apply the new knowledge you gained by reading the Information sheet s. This is your opportunity to practise the job. You may need to practise the job or activity several times before you become competent.

5. Specification **sheets**, specifying the details of the job to be performed will be provided where appropriate.

6. A review of competency is provided on the last page to help remind if all the required assessment criteria have been met. This record is for your own information and guidance and is not an official record of competency

When working though this Module always be aware of your safety and the safety of others in the training room. Should you require assistance or clarification please consult your trainer or facilitator.

When you have satisfactorily completed all the Jobs and/or Activities outlined in this module, an assessment event will be scheduled to assess if you have achieved competency in the specified learning outcomes. You will then be ready to move onto the next Unit of Competency or Module

Table of Contents

Copyright i

How to use this Competency Based Learning Material (CBLM) v

Module Content 1

Learning Outcome 1: Use built-in user management 2

Learning Experience 1: Use built-in user management 3

Information sheet 1: Use built-in user management 4

Self-Check Sheet 1: Use built-in user management 24

Answer Key 1: Use built-in user management 25

Job Sheet-1: Create Django Users with Different Privileges 26

Specification Sheet 1: Create Django Users with Different Privileges 27

Learning Outcome 2: Apply custom user management 28

Learning Experience 2: Apply custom user management 29

Information sheet 2: Apply custom user management 30

Self-Check Sheet 2: Apply custom user management 38

Answer Key 2: Apply custom user management 39

Job Sheet-2: Implement User Objects with Permissions and Authorization 40

Specification Sheet 2: Implement User Objects with Permissions and Authorization 41

Reference 42

Review of Competency 43

Development of CBLM 44

Module Content

Unit of Competency	**Apply User Management**
Unit Code	OU-ICT-WADP-06-L4-V1
Module Title	**Applying User Management**
Module Descriptor	This module covers the knowledge, skills and attitudes required to applying User Management It includes the task of using built-in user management, Apply custom user management, applying notification and activation process
Nominal Hours	80 Hours
Lerning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs 1. Use built-in user management 2. Apply custom user management 3. Apply notification and activation process

Assessment Criteria

1. Django default user management is applied.
2. Users, and groups are created.
3. Users are assigned to necessary groups.
4. User objects are implemented.
5. Permissions & authorization are applied.
6. Authentication in web requests is addressed.

Learning Outcome 1: Use built-in user management

Assessment Criteria	1. Django default user management is applied. 2. Users, and groups are created. 3. Users are assigned to necessary groups.
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Django default user management process 2. Users, and groups 3. Assigning users to groups
Activities/job/Task	1. Create Django Users with Different Privileges
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 1: Use built-in user management

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about about the learning materials	1. Instructor will provide the learning materials 'Use built-in user management'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Use built-in user management"	2. Read Information sheet 1: Use built-in user management 3. Answer Self-check 1: Use built-in user management 4. Check your answer 1: Use built-in user management
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-1: Create Django Users with Different Privileges

Information sheet 1: Use built-in user management

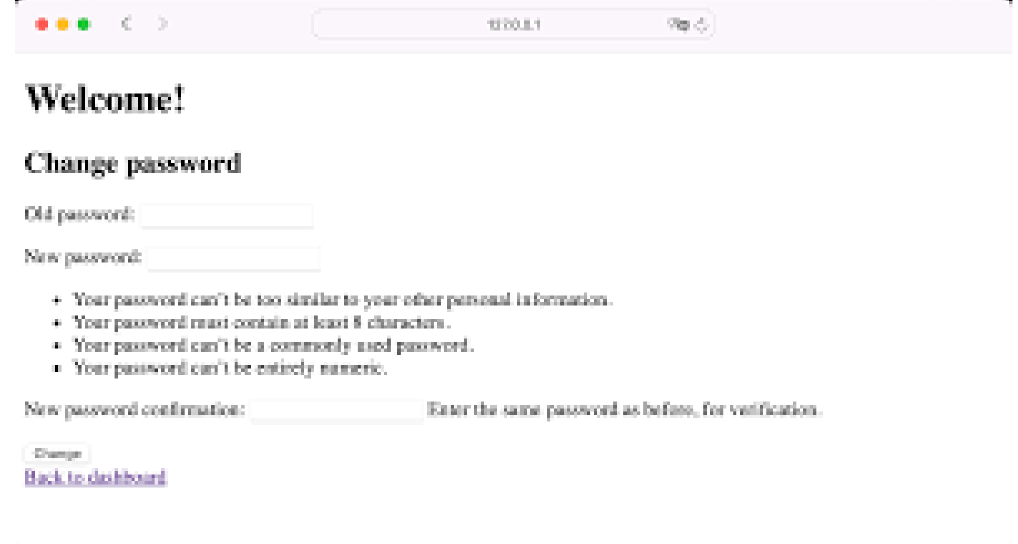
Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

1.1. Django default user management is applied.
1.2. Users, and groups are created.
1.3. Users are assigned to necessary groups.

1.1. Apply Django default user management

The Django user system is based on the django.contrib.auth package built-in to the Django framework. In this section you'll learn about the core concepts offered by this package, which is the default user system used by a variety of Django apps including the Django admin.



User types, sub-types, groups and permissions

There are two main types of Django user classes: User and AnonymousUser. If a user authenticates himself (i.e. provides a valid username/password) Django recognizes him as a User. On the other hand, if a user just surfs an application without any authentication, Django recognizes him as an AnonymousUser.

Any User can be further classified into one of various sub-types

- superuser.- The most powerful user with permissions to create, read, update and delete data in the Django admin, which includes model records and other users.

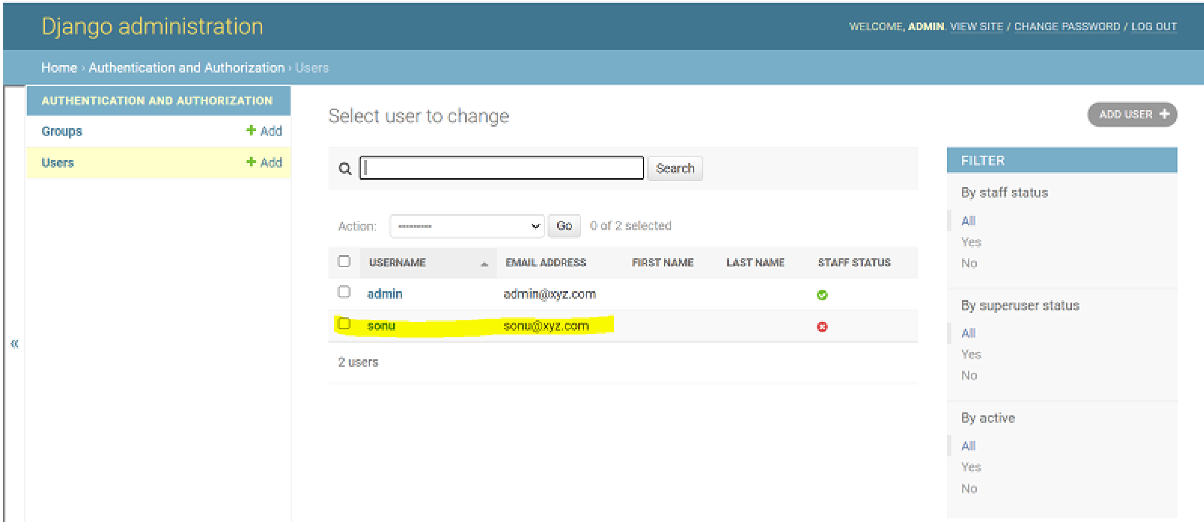
- staff.- A user marked as staff can access the Django admin. But permissions to create, read, update and delete data in the Django admin must be given explicitly to a user. By default, a superuser is marked as staff.
- active.- All users are marked as active if they're in good standing. Users marked as inactive aren't able to authenticate themselves, a common state if there's a pending post-registration step (e.g. confirm email) or a user is banned and you don't want to delete his data.

Django also offers the concept of a Group class to grant a set of users the same set of permissions without having to assign them individually. For example, you can grant permissions to a group and then assign users to the group to make permission management easier. In this manner, you can revoke or add permissions in a single step to a set of users, as well as quickly give new users the same permissions.

In addition, you can assign Django permissions granularly to a User or Group in order for them to do CRUD(Create-Update-Delete) records on Django models, a process which is done through Permission model records. Or you can also assign coarser grained Django permissions on url/view methods or template content to grant access to a User, Group or even Permission assignee.

Create users

Example illustrates the various ways in which you can create a superuser.



Example: Create Django superuser

```
[user@coffeehouse ~]$ python manage.py createsuperuser
Username (leave blank to use 'admin'):
Email address: admin@coffeehouse.com
Password:
Password (again):
Superuser created successfully.
```

```
[user@coffeehouse ~]$ python manage.py createsuperuser --username=bigboss
                    --email=bigboss@coffeehouse.com
Password:
Password (again):
Superuser created successfully.

[user@coffeehouse ~]$ python manage.py shell
Python 2.7.3 (default, Apr 10 2013, 06:20:15)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_superuser(username='angelinvestor',
                              email='angelinvestor@coffeehouse.com',
                              password='seedfunding')
```

In addition, it's also possible to create a superuser through the Django shell using the User model class with the create_superuser() method. Notice how the create_superuser() method requires the same username, email and password arguments.

When you create a superuser in any of these ways, the user is also automatically set as a staff member and marked as active, so you don't need to take any additional steps to access the Django admin -- which requires staff member permissions -- or proceed with authentication, which requires a user to be marked as active.

Sometimes you just want to create a regular user, in which case you can use Django's shell utility and create a user directly through the User model. This process is illustrated in Example.

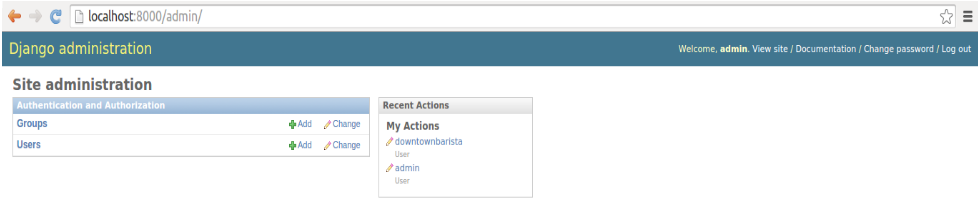
Example. Create regular Django user through shell

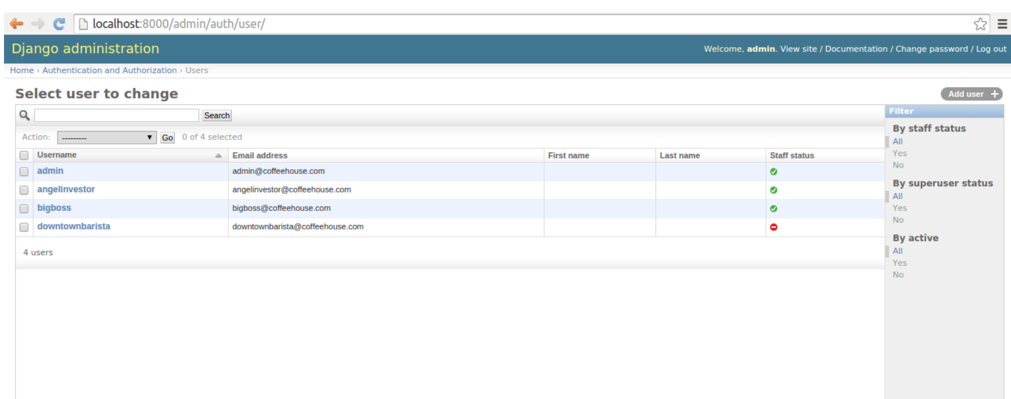
```
[user@coffeehouse ~]$ python manage.py shell
Python 2.7.3 (default, Apr 10 2013, 06:20:15)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from django.contrib.auth.models import User
>>> user = User.objects.create_user(username='downtownbarista',
                              email='downtownbarista@coffeehouse.com',
                              password='cappuccino')
>>> user.is_staff
False
>>> user.is_active
True
>>> user.is_superuser
False
```

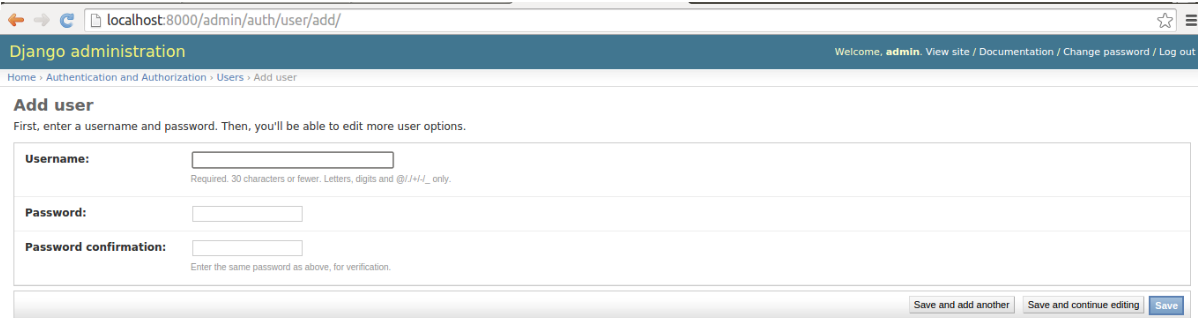
As you can see in Example, after you gain access to the Django shell you import the User model class and invoke the create_user() method with the username, email and password arguments. The result of the create_user() method contains the newly created user.
To confirm the create_user() method generates a regular user, you can see in Example a call to the various User model attributes, confirming the user is neither staff or superuser and is just marked as active.

Finally, it's also possible to create users through the Django admin. To do this you'll first need to make sure you have superuser access to the Django admin. Once you access the Django admin, you'll see a screen like the one Figure 1-1, click on the 'Users' link. The 'Users' link takes you to a screen like the one in Figure 1-2, click on the button 'Add User+' in the top right. The 'Add User+' button takes you to a screen like the one in Figure 6-3 where you can introduce the credentials for a new user.

If you wish to change the sub-type (i.e. superuser, staff) of a user created in this manner, you can also do it in the Django admin, a process that's described in the next section.







Manage users

Once a user is in a Django application, you'll end up managing him. This management can be either revoking his privileges, adding to his privileges or even editing his profile information. You can manage Django users in two ways, in the Django admin or by manipulating a User model in the Django shell or directly in your application.

The easiest way to manage users is directly in the Django admin. Once you access the Django admin you'll see a screen like the one in figure 1-1, if you click on the 'Users' link you'll be taken to a screen like the one in figure 1-2 which contains a list of Django users. Each Django user presented in figure 1-2 has his username as a link.

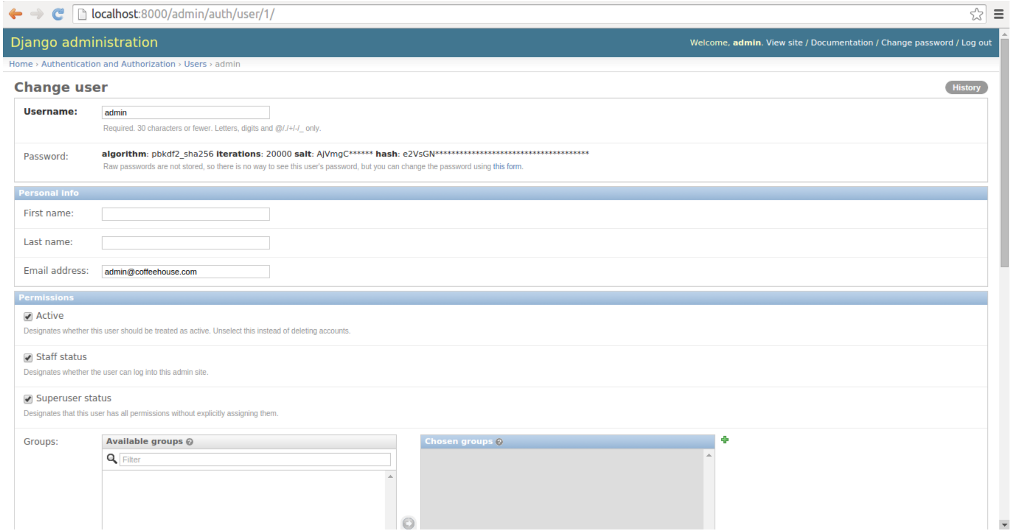


Figure: Django admin change user page - Part 1

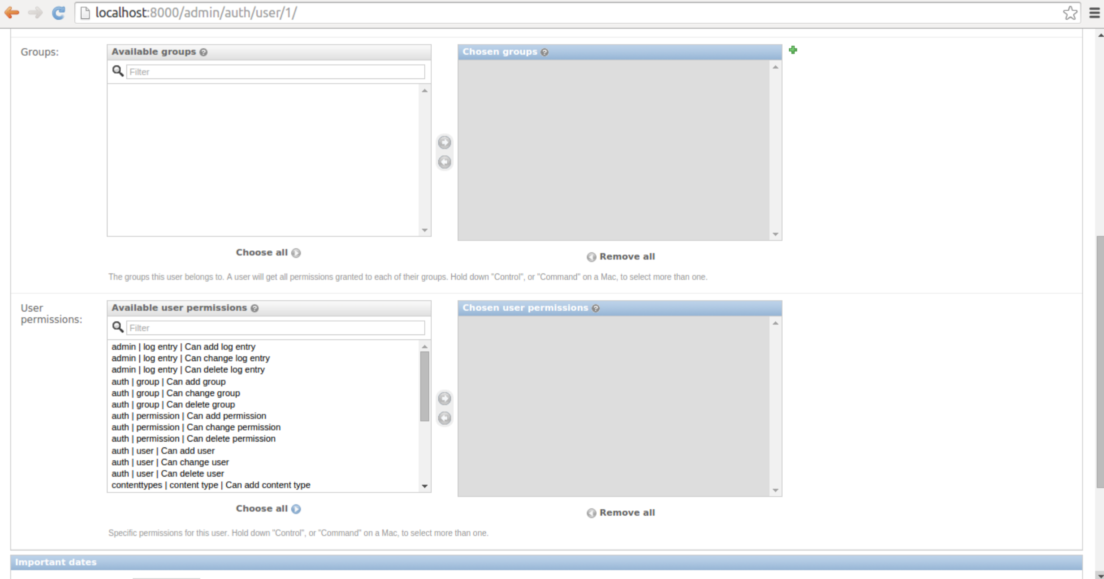


Figure: Django admin change user page - Part 2

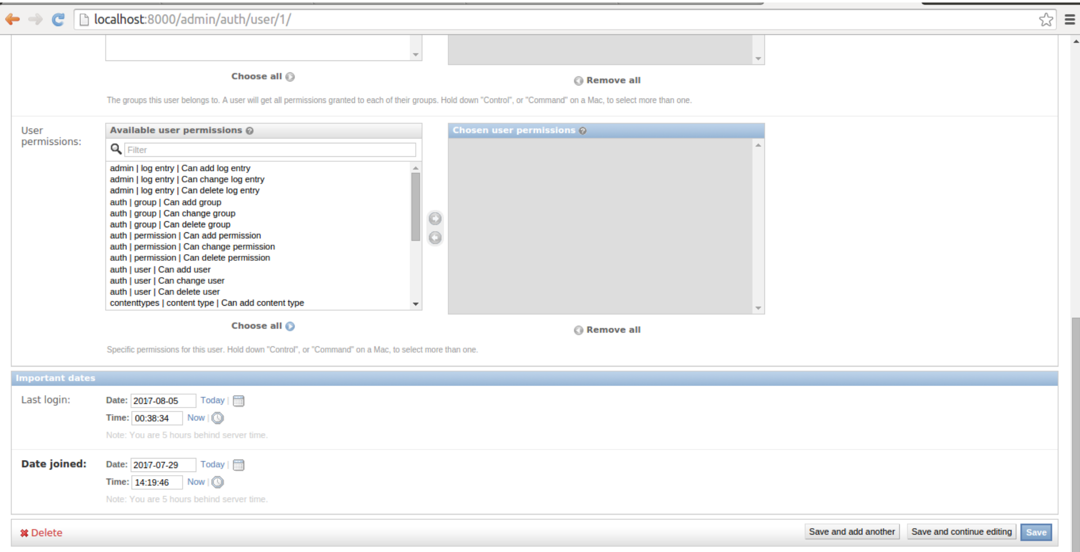


Figure: Django admin change user page - Part 3

The first part of a user's profile you can edit is illustrated in figure 6-4. Here you can edit his username, his password -- by clicking on the 'small form' link at the end of the small text -- his first name, his last name, as well as his email. In addition, you can see there are three check-boxes where it's possible to change a user's active, staff and superuser status.

If you scroll down, you'll see the second part of a user's profile you can edit which is illustrated in figure 1-5. Here you can assign a user to different groups, as well as assign a user individual CRUD permissions over Django models. Here I would advise you to carefully evaluate the need to assign individual CRUD permissions to a user, a more flexible approach is to create groups and assign them CRUD permissions and then assign users to groups, this way the permissions become easier to track and reusable for other users.

If you scroll further down to the end, you'll see the third part of a user's profile you can edit which is illustrated in figure 6-6. Here you can view and update a user's last login, as well as the date a user was created. At the bottom right of the page, you can see the various save buttons to store any changes made to the page. And in addition, at the bottom left there's a 'Delete' button to remove a user completely, however, I would advise you to consider just unchecking a user's active status to restrict access. This last step is sufficient to block a user from accessing an application again and it keeps his other data untouched in case you want to undo the action.

Another option to modify a user's profile is to directly manipulate his User model record. As illustrated in listing 6-3, you first make a query for the desired user and then modify the model attributes or execute one of the User model helper methods.

Example. Manage Django user through shell

```
[user@coffeehouse ~]$ python manage.py shell
Python 2.7.3 (default, Apr 10 2013, 06:20:15)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>> from django.contrib.auth.models import User
>>> user = User.objects.get(id=1)
>>> user.username = 'superadmin'
>>> user.save()
>>> userbig = User.objects.get(username='bigboss')
>>> userbig.is_superuser
True
>>> userbig.superuser = False
>>> userbig.first_name = 'Big'
>>> userbig.last_name = 'Boss'
>>> userbig.save()
>>> userbig.is_superuser
False
>>> userbig.get_full_name()
u'Big Boss'
>>> userbarista = User.objects.get(email='downtownbarista@coffeehouse.com')
>>> userbarista.email ='barista@coffeehouse.com'
>>> userbarista.save()
>>> userbarista.set_password('mynewpass')
>>> userbarista.check_password('oldpass')
False
>>> userbarista.check_password('mynewpass')
True
```

As you can see in Example, you can modify the same User profile values as those presented in the Django admin in figure 1-4, figure 1-5 & figure 1-6. Just be aware that because you're making a query, any changes made to model fields must be followed by

a call to the save() method on the reference for fields to be persisted. Table and Table contain a full list of fields and methods available on the User model.

Table Django django.contrib.auth.models.User fields

Field	Description
username	(Required) 30 characters or fewer and can 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. Note that Django doesn't store the raw password.
groups	A many-to-many relationship to django.contrib.auth.models.Group
user_permissions	A many-to-many relationship to django.contrib.auth.Permission
is_staff (Boolean)	Designates whether a user can access the admin site.
is_active (Boolean)	Designates whether a user is considered active.
is_superuser	(Boolean) Designates whether a user has all permissions without explicitly assigning them.
last_login	A datetime of the user's last login, set to NULL if the user has never logged in
date_joined	A datetime designating when the account was created. Is set to the current date/time by default when the account is created.

Table. Django django.contrib.auth.models.User methods

Method	Description
get_username()	Returns the username for the user. Since the User model can be changed for another, this method is the recommended approach instead of referencing the username attribute directly.
is_anonymous()	For a User this method always returns False, it's only used as a way to differentiate between User and AnonymousUser.
is_authenticated()	For a User this method always returns True, it's only used to to find out whether the user has gone through the AuthenticationMiddleware (representing the currently logged-in user).
get_full_name()	Returns the first_name and the last_name fields, with a space in between.
get_short_name()	Returns the first_name.

set_password(raw_password)	Sets the user's password to the given raw string, taking care of the password hashing. Note that when the raw_password is None, the password is set to an unusable password, as if set_unusable_password() were used.
check_password(raw_password)	Returns True if the given raw string is the correct password for the user, talking care of the password hashing for making the comparison.
set_unusable_password()	Marks the user as having no password set. Note this isn't the same as having a blank string for a password. check_password() for this user will never return True. This is helpful if authentication takes place against an existing external source (e.g.LDAP directory).
has_usable_password()	Returns False if set_unusable_password() has been called for the user.
get_group_permissions(obj=None)	Returns a set of group permission strings for the user. If the obj is passed, only returns the group permissions for the specific object.
get_all_permissions(obj=None)	Returns a set of group and user permission strings for the user. If the obj is passed, only returns the group permissions for the specific object.
has_perm(perm, obj=None)	Returns True if the user has the specified permission, where perm is in the format <app label>.<permission codename>". Note if the user is inactive, this method always returns False. If the obj is passed, the check ocurrs on the specific object and not on the model.
has_perms(perm_list, obj=None)	Returns True if the user has each of the specified permissions, where each perm is in the format <app label>.<permission codename>". Note if the user is inactive, this method always returns False. If the obj is passed, the check ocurrs on the specific object and not on the model.
has_module_perms(package_name)	Returns True if the user has permissions in the given package (i.e. the Django app label). If the user is inactive, this method always returns False.
email_user(subject, message, from_email=None, **kwargs)	Sends an email to the user. If from_email is None, Django uses the DEFAULT_FROM_EMAIL in settings.py. Also note this method relies on Django's send_mail() method to which it passes the **kwargs argument. See the Django email

	shortcut methods for more details on the send_mail() method and **kwargs values

By default, Django enforces password follow certain rules, such as: not being similar to a username, containing a minimum amount of characters, avoiding common words and forcing passwords to consist of more than numbers. These password rules are defined in a project's settings.py file in the AUTH_PASSWORD_VALIDATORS variable, as follows

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

This list of validators can be edited to suit the needs of a project, by either removing certain rules or inclusively making them more strict with options[1]

Create and manage groups

Django groups can be created in the Django admin. With a superuser account access the Django admin and you'll see a screen like the one figure 1-1, click on the 'Groups' link. The 'Groups' link takes you to a screen like the one in figure 1-7, click on the button 'Add Group+' in the top right. The 'Add Group+' button takes you to a screen like the one in Figure where you can create a new group introducing its name.

Figure: Django admin Groups list

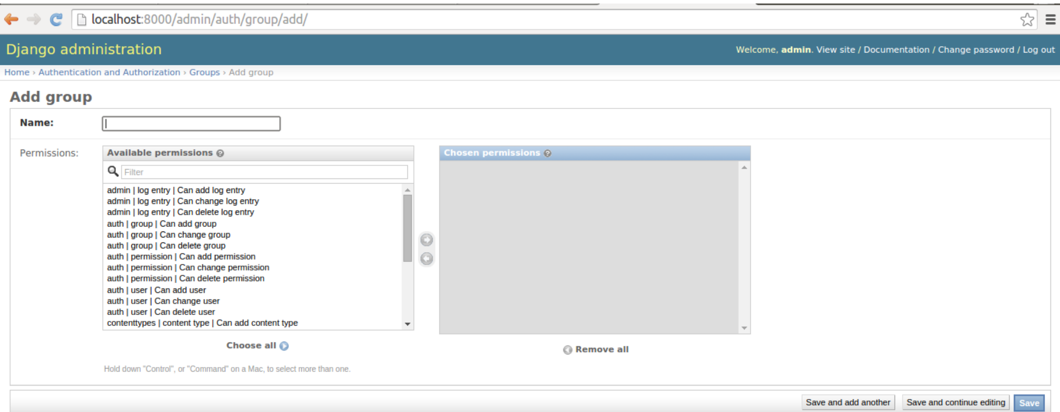


Figure. Django admin to create new group

As you can see in Figure, all that's need to create a group is a name and you can optionally specify permissions given to the group to do CRUD operations on Django models in the application.

The management of groups is simpler than users and can also be completely done from the Django admin. What you'll end up doing most of the time is assigning users to groups.

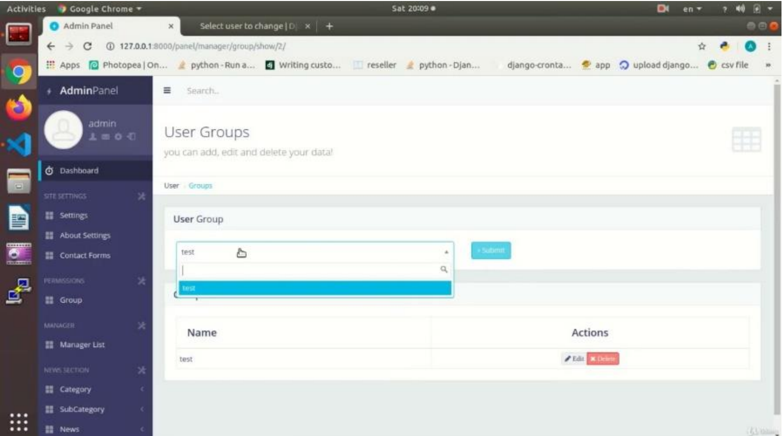
To assign a user to a group, when you're editing a user you'll see a selection grid for just this purpose, which is illustrated in figure 1-5. To edit a group's properties -- name & Create-Delete-Update Django model permissions -- you can do so from the same page where you created it illustrated in Figure. To delete a group from the Groups list illustrated in figure 1-7, you select the group you wish to delete and select the action from the drop-down list, as illustrated in figure.

Figure. Django admin to delete group

Tip Group model data is stored in the database table auth_group. And User-Group relationships is stored in the database table auth_user_groups.

1.2. Create Users, and groups.

In Django, a profile is an optional feature that allows you to store additional information about a user beyond what is included in the built-in User model. The User model provides basic information about a user, such as their username, password, first name, last name, and email address.

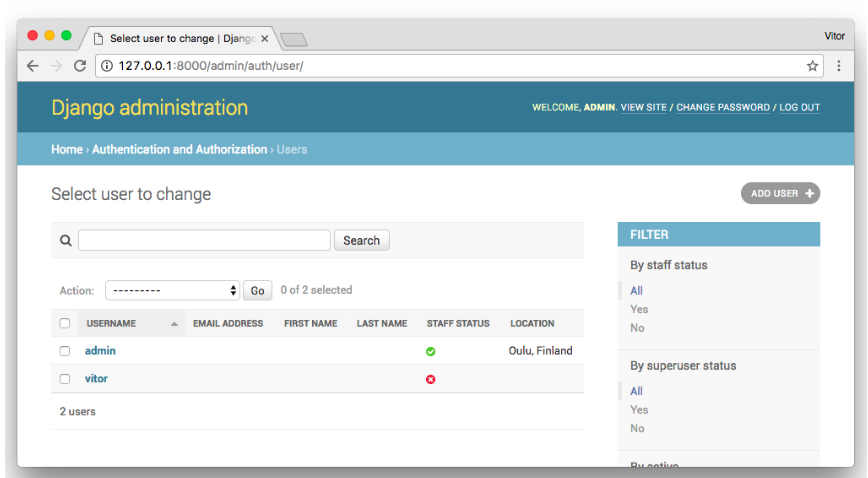


To implement profiles in Django, you can create a separate model to store the additional information. This model is usually associated with the User model through a one-to-one relationship. This means that each user has exactly one profile, and each profile belongs to exactly one user. Profiles are implemented as a separate model linked to the User model through a one-to-one relationship, making it easy to manage the additional information about a user.

In this information about Profiles and Groups in Django, we'll see more about how profiles and groups are created and implemented the Django applications.

a. Profile

Creating a User profile



To create a user profile in Django, you can use Django's built-in User model and extend it with an additional profile model that stores additional information about the user. Here's an example of how you can do this:

- Create a new app for the user profile, for example, "profiles".
- In the models.py file of the profiles app, create a new model for the profile

```
from django.contrib.auth.models import User
from django.db import models

class Profile(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    bio = models.TextField(max_length=500, blank=True)
    location = models.CharField(max_length=30, blank=True)
    birth_date = models.DateField(null=True, blank=True)
    profile_pic = models.ImageField(upload_to='profile_pics', blank=True)
```

- Run the following command to create the database tables for the new models:

```
pythge.py makemigrations
python manage.py migrate
```

- In the views.py file of the profiles app, create a view that displays the user profile:

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from .models import Profile

@login_required
def profile(request):
    profile = request.user.profile
    return render(request, 'profiles/profile.html', {'profile': profile})
```

- In the urls.py file of the profiles app, create a URL pattern for the profile view:

```
from django.urls import path
from .views import profile

app_name = 'profiles'

urlpatterns = [
    path('profile/', profile, name='profile'),
]
```

- In the template folder of the profiles app, create a template file named profile.html that displays the user's profile information:

```
<h1>{{ profile.user.username }}'s Profile</h1>
<p>Bio: {{ profile.bio }}</p>
<p>Location: {{ profile.location }}</p>
<p>Birthdate: {{ profile.birth_date }}</p>
<img src="{{ profile.profile_pic.url }}" alt="Profile Picture">
```

- Finally, in the settings.py file of your project, add the profiles app to the INSTALLED_APPS list:

```
INSTALLED_APPS = [    ...    'profiles',    ...]
```

This should be enough to create a basic user profile in Django. You can extend this example by adding additional fields, customizing the view, or adding validation to the model. Let's see further in Profiles and Groups in the Django information, how we can update the user profile.

Updating a User Profile

To update a user profile in Django, you can create a form that allows users to edit their profile information. Here's an example of how you can do this:

In the forms.py file of the profiles app, create a form for editing the user profile:

```
from django import forms
from .models import Profile

class ProfileForm(forms.ModelForm):
    class Meta:
        model = Profile
        fields = ['bio', 'location', 'birth_date', 'profile_pic']
```

- In the views.py file of the profiles app, create a view for updating the user profile:

```
from django.shortcuts import render, redirect
from django.contrib.auth.decorators import login_required
from .forms import ProfileForm

@login_required
def edit_profile(request):
    profile = request.user.profile
    if request.method == 'POST':
        form = ProfileForm(request.POST, request.FILES, instance=profile)
        if form.is_valid():
            form.save()
            return redirect('profiles:profile')
    else:
        form = ProfileForm(instance=profile)
    return render(request, 'profiles/edit_profile.html', {'form': form})
```

- In the urls.py file of the profiles app, create a URL pattern for the edit profile view:

```
from django.urls import path
from .views import edit_profile

app_name = 'profiles'
```

```
urlpatterns = [
    ...
    path('edit_profile/', edit_profile, name='edit_profile'),
    ...
]
```

In the template folder of the profiles app, create a template file named edit_profile.html that displays the form for editing the user profile:

```
<h1>Edit Profile</h1>
<form method="post" enctype="multipart/form-data">
  {% csrf_token %}
  {{ form.as_p }}
  <button type="submit">Save</button>
</form>
```

Finally, in the template file for the profile view (profile.html), add a link to the edit profile page:

```
<h1>{{ profile.user.username }}'s Profile</h1>
<p>Bio: {{ profile.bio }}</p>
<p>Location: {{ profile.location }}</p>
<p>Birthdate: {{ profile.birth_date }}</p>
<img src="{{ profile.profile_pic.url }}" alt="Profile Picture">
<a href="{% url 'profiles:edit_profile' %}">Edit Profile</a>
```

This should be enough to allow users to edit their profile information in Django. You can further customize the form and views to meet your needs.

What are the Groups in Django?

As we saw above how profiles can be implemented and used in Django, now we'll look at how Profiles and Groups in Django are associated with each other.

A group is a way to categorize user accounts based on their permissions. It is a way to manage authorization for a set of users, rather than managing individual user accounts. By grouping users with similar permissions together, you can simplify the management of user access control for your Django application. For example, you may have a group of users that are responsible for editing content on your website. Instead of granting editing permissions to each user, you can create a "editors" group and add the relevant users to this group. By doing so, you can grant editing permissions to the entire group, rather than to each user.

In Django, the Group model is used to manage groups and their permissions. The Group model is part of the built-in authentication system in Django, and it is stored in the database. The Group model is associated with the User model, which represents individual user accounts, through a many-to-many relationship. This means that a user can belong to multiple groups and a group can have multiple users.

To use groups in Django, you will

- first need to create the groups, which you can do through the Django admin interface or the command line.
- Once you have created the groups, you can add users to them and assign the appropriate permissions.
- In Django, permissions are defined by the Django apps that you have installed and the models that you have created. You can use the Django admin interface or the command line to assign permissions to a group.

b. Working with Django Groups

Creating Django User Groups

To create Django user groups, you can follow these steps

Create a Group model: The first step is to create a model to represent the groups. You can do this by creating a new model in your Django app and using the Group model from the built-in Django authentication system as the base class. You can also add any additional fields that you need to store information about the groups.

#we added a description field to store a text description for each group.
from django.contrib.auth.models import Group

```
class CustomGroup(Group):
    description = models.TextField(blank=True)
```

Register the model with the admin interface: To manage groups through the Django admin interface, you will need to register the Group model with the admin site. You can do this by adding a line of code in the admin.py file of your app.

```
from django.contrib.auth.admin import GroupAdmin
from django.contrib.auth.models import Group
from django.utils.translation import gettext_lazy as _

admin.site.unregister(Group)

@admin.register(CustomGroup)
class CustomGroupAdmin(GroupAdmin):
    fieldsets = (
        (None, {'fields': ('name', 'permissions')}),
        (_('Description'), {'fields': ('description',)}),
    )
```

This code unregisters the built-in Group model from the admin interface and replaces it with the CustomGroup model. The CustomGroupAdmin class extends the built-in GroupAdmin class and adds the description field to the admin interface

- **Create the groups**: Once you have created the Group model and registered it with the admin interface, you can create the groups. You can do this through the Django admin interface or the command line. To create a group through the admin interface, navigate to the Groups section, click the "Add Group" button, and enter a name for the group.
- **Assign permissions to the groups**: In Django, permissions are defined by the models in your Django apps. To assign permissions to a group, you can use the Django admin interface or the command line. To assign permissions through the admin interface, navigate to the Groups section, select the group you want to assign permissions to and use the "Permissions" section to select the permissions you want to grant to the group.
- **Add users to the groups**: To add users to a group, you can use the Django admin interface or the command line. To add users through the admin interface, navigate to the Groups section, select the group you want to add users to and use the "Members" section to select the users you want to add to the group.

Creating Django User Groups with Custom Permissions

Moving further in this information, Profiles and Groups in Django, let's learn about what is custom permissions with context to Django applications and how we can create user groups with custom permissions. **What are Custom permissions?** Custom permissions are user-defined permissions in Django that allow you to control access to specific actions within your application. Custom permissions are implemented using the Permission model in Django, which allows you to define a specific name and code name for the permission, and associate it with a specific model in your application. These custom permissions can then be assigned to users or groups, which can be used to control access to specific parts of your application.

Create User Groups with Custom permissions To create Django user groups with custom permissions, you can use the Django built-in Group model and the Permission model. Here is a general outline of the steps you would need to follow:

- Create your custom permissions using the Permission model. You can do this through the Django admin interface, or by using Django's migrations.
- Add the custom permissions to a group using the Group model. Again, you can do this through the Django admin interface or by using Django's migrations.
- Assign the group to a user using the User model. You can do this by updating the user's groups field.

Here is an example of how you might do this in code

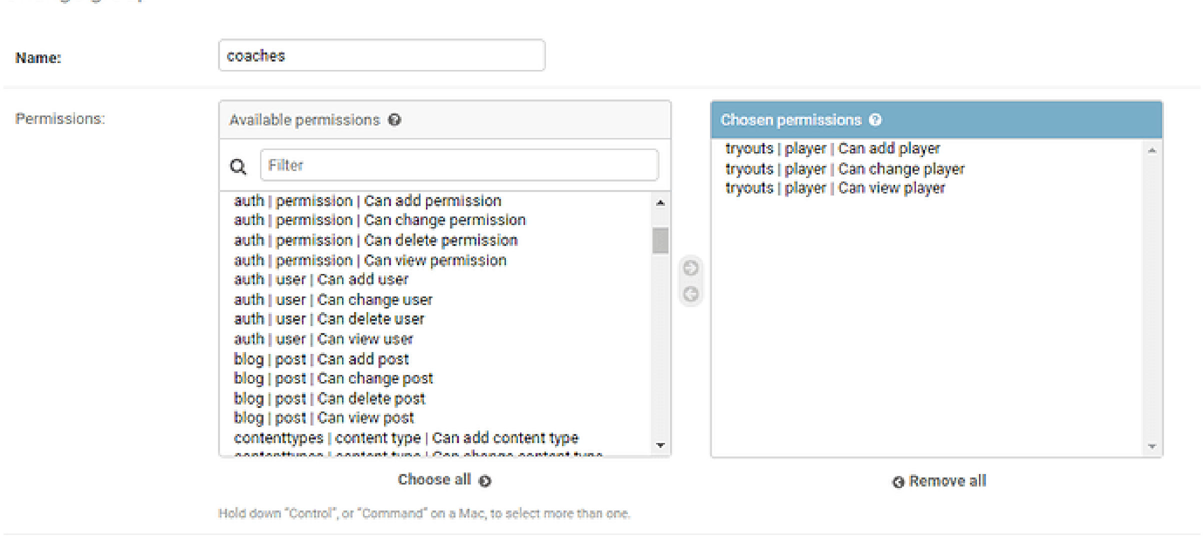
```
from django.contrib.auth.models import User, Group, Permission
from django.contrib.contenttypes.models import ContentType

# Step 1: Create custom permissions
content_type = ContentType.objects.get_for_model(MyModel)
permission = Permission.objects.create(
```

```
    codename='custom_permission',
    name='Can access custom feature',
    content_type=content_type
)

# Step 2: Add permissions to a group
group = Group.objects.create(name='My Group')
group.permissions.add(permission)

# Step 3: Assign the group to a user
user = User.objects.get(username='john')
user.groups.add(group)
```

Now, the user john should have the custom permission Can access custom feature as a result of being in the My Group group.

1.3. Assigned Users to necessary groups.

Django's built-in group and permission system allows you to assign users to groups and grant them permissions based on their group membership. Here's how to achieve this:



Creating Groups: You can create groups through the Django admin interface or using management commands.

Admin Interface

1. Login to your Django admin panel.
2. Navigate to the "auth" app section (usually under "Users").
3. Click on the "Groups" link.
4. Click the "Add group" button.

5. Enter a name and optional permissions for the group (we'll cover permissions later).
6. Click "Save".

Management Command

Bash
```bash
python manage.py creategroup <group_name>
```

Replace <group_name> with the desired name for your group (e.g., editors, admins).

Adding Users to Groups: Once you have groups created, you can assign users to them.

Admin Interface

1. In the Django admin, navigate to the "auth" app section and select "Users".
2. Click on the specific user you want to edit.
3. Under the "Groups" section, select the groups you want the user to belong to using the checkboxes.
4. Click "Save".

Python Code:

```python
from django.contrib.auth.models import Group, User

user = User.objects.get(username="johndoe")  # Replace with actual username
group = Group.objects.get(name="editors")  # Replace with actual group name
user.groups.add(group)
```

Assigning Permissions to Groups (Optional)

- Permissions define specific actions users can perform on Django models.
- By assigning permissions to groups, you can control access based on group membership.

Admin Interface

1. In the Django admin, navigate to the "auth" app section and select "Permissions".
2. Select the specific group you want to edit permissions for.
3. Under "Permissions granted to this group," check the boxes for the permissions you want to grant the group.
4. Click "Save".

Python Code

```
from django.contrib.auth.models import Permission, Group

group = Group.objects.get(name="editors")
permission = Permission.objects.get(name="Can edit information")  # Replace with actual permission name
group.permissions.add(permission)
```

By following these steps, you can effectively manage user groups and permissions in your Django application, ensuring users have the necessary access based on their assigned groups.

Self-Check Sheet 1: Use built-in user management

1. Which of the following statements is TRUE about Django's default user management system?

Ans:

a) It requires a separate app to be installed.
b) It is built-in to the Django framework.
c) It cannot be customized.
d) It only supports anonymous users.

2. What are the different user types defined in Django's default user system? (Choose all that apply)

Ans:

a) Superuser

b) Staff user

c) Regular user

d) Guest user

3. How can you create a superuser in Django?

Ans:

a) Through the Django admin interface (if you have superuser access already).
b) Using the createsuperuser management command.
c) By creating a regular user and granting them superuser permissions.
d) There is no built-in way to create a superuser.

4. What is the difference between a superuser and a staff user in Django?

Ans:

a) There is no difference.

b) A superuser has full administrative privileges, while a staff user can only access the admin panel.

c) A staff user can create other staff users, while a superuser cannot.

d) A superuser is created by default, while a staff user needs to be created manually.

5. How can you check if a user has a specific permission in Django?

Ans:

a) By comparing the user's group membership.

b) By using the user.is_staff attribute. c) By using the user.has_perm("app.permission_codename") method.

d) There is no built-in way to check user permissions.

Answer Key 1: Use built-in user management

1. Which of the following statements is TRUE about Django's default user management system?

Ans: b) It is built-in to the Django framework.

2. What are the different user types defined in Django's default user system? (Choose all that apply)

Ans: a) Superuser b) Staff user c) Regular user

3. How can you create a superuser in Django?

Ans: b) Using the createsuperuser management command.

4. What is the difference between a superuser and a staff user in Django?

Ans: b) A superuser has full administrative privileges, while a staff user can only access the admin panel.

5. How can you check if a user has a specific permission in Django?

Ans: c) By using the user.has_perm("app.permission_codename") method.

Job Sheet-1: Create Django Users with Different Privileges

UoC Cover

OU-ICT-WADP-02- L6-V1: Apply User Management

Working Procedure / Steps

1. **Access Django Shell:**
    - Open a terminal window and navigate to your Django project directory.
    - Run the command python manage.py shell to access the Django interactive shell.
2. **Import User Model:**
    - Inside the shell, import the User model from django.contrib.auth.models.
3. **Create Superuser:**
    - Use the User.objects.create_superuser method to create a superuser account. This user will have full administrative access.
        - Provide arguments for username, email, and password.
4. **Create Staff User:**
    - Use the User.objects.create_user method to create a regular user account with staff privileges. This user can access the Django admin panel but won't have full administrative permissions.
        - Provide arguments for username, email, and password.
        - Set is_staff to True.
5. **Create Regular User:**
    - Use the User.objects.create_user method to create a regular user account without any special privileges.
        - Provide arguments for username, email, and password.
        - Set is_staff to False (default).
6. **View User Information:**
    - Use the created user objects to access their information, such as username, email, and staff status.

Specification Sheet 1: Create Django Users with Different Privileges

Technical Requirements

- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later
- **Terminal:** A command-line interface for executing commands.

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.

Learning Outcome 2: Apply custom user management

Assessment Criteria	1. User objects are implemented. 2. Permissions & authorization are applied. 3. Authentication in web requests is addressed
Conditions and Resources	1. Real or simulated workplace 2. CBLM 3. Handouts 4. Laptop 5. Multimedia Projector 6. Paper, Pen, Pencil, Eraser 7. Internet facilities 8. White board and marker 9. Audio Video Device
Contents	1. Working with user objects 2. Permissions & authorization 3. Authentication in web requests
Activities/job/Task	1. Implement User Objects with Permissions and Authorization
Training Methods	1. Discussion 2. Presentation 3. Demonstration 4. Guided Practice 5. Individual Practice 6. Project Work 7. Problem Solving 8. Brainstorming
Assessment Methods	Assessment methods may include but not limited to 1. Written Test 2. Demonstration 3. Oral Questioning 4. Portfolio

Learning Experience 2: Apply custom user management

In order to achieve the objectives stated in this learning guide, you must perform the learning steps below. Beside each step are the resources or special instructions you will use to accomplish the corresponding activity.

Learning Activities	Recourses/Special Instructions
1. Trainee will ask the instructor about about the learning materials	1. Instructor will provide the learning materials 'Apply custom user management'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Apply custom user management"	2. Read Information sheet 2: Apply custom user management 3. Answer Self-check 2: Apply custom user management 4. Check your answer with 2: Apply custom user management
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Job Sheet-2: Implement User Objects with Permissions and Authorization

Information sheet 2: Apply custom user management

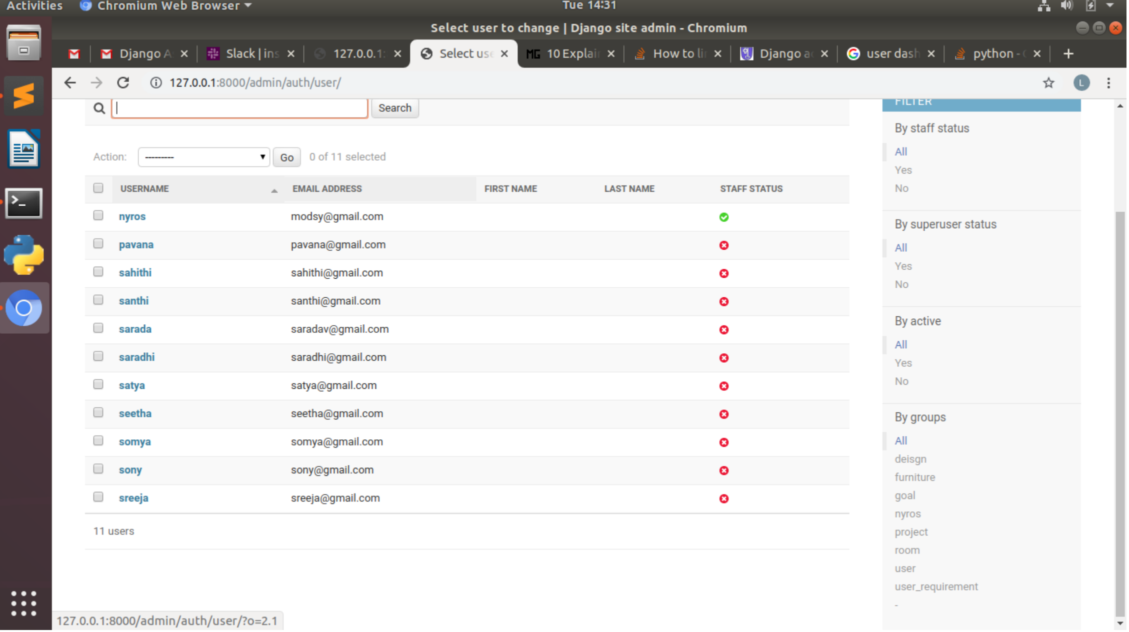
Learning Objective:

After completion of this Information sheet , the learners will be able to explain, define and interpret the following contents:

2.1. User objects are implemented.
2.2. Permissions & authorization are applied.
2.3. Authentication in web requests is addressed

2.1. Implement User objects.

The authentication that comes with Django is good enough for most common cases, but you may have needs not met by the out-of-the-box defaults. Customizing authentication in your projects requires understanding what points of the provided system are extensible or replaceable. This document provides details about how the auth system can be customized.



Authentication backends provide an extensible system for when a username and password stored with the user model need to be authenticated against a different service than Django's default.

You can give your models custom permissions that can be checked through Django's authorization system.

You can extend the default User model, or substitute a completely customized model.

Other authentication sources
There may be times you have the need to hook into another authentication source – that is, another source of usernames and passwords or authentication methods.

For example, your company may already have an LDAP setup that stores a username and password for every employee. It'd be a hassle for both the network administrator and the users themselves if users had separate accounts in LDAP and the Django-based applications.

So, to handle situations like this, the Django authentication system lets you plug in other authentication sources. You can override Django's default database-based scheme, or you can use the default system in tandem with other systems.

See the authentication backend reference for information on the authentication backends included with Django.

Specifying authentication backends
Behind the scenes, Django maintains a list of "authentication backends" that it checks for authentication. When somebody calls django.contrib.auth.authenticate() – as described in How to log a user in – Django tries authenticating across all of its authentication backends. If the first authentication method fails, Django tries the second one, and so on, until all backends have been attempted.

The list of authentication backends to use is specified in the AUTHENTICATION_BACKENDS setting. This should be a list of Python path names that point to Python classes that know how to authenticate. These classes can be anywhere on your Python path.

By default, AUTHENTICATION_BACKENDS is set to:

["django.contrib.auth.backends.ModelBackend"]
That's the basic authentication backend that checks the Django users database and queries the built-in permissions. It does not provide protection against brute force attacks via any rate limiting mechanism. You may either implement your own rate limiting mechanism in a custom auth backend, or use the mechanisms provided by most web servers.

The order of AUTHENTICATION_BACKENDS matters, so if the same username and password is valid in multiple backends, Django will stop processing at the first positive match.

If a backend raises a PermissionDenied exception, authentication will immediately fail. Django won't check the backends that follow.

Note

Once a user has authenticated, Django stores which backend was used to authenticate the user in the user's session, and reuses the same backend for the duration of that session whenever access to the currently authenticated user is needed. This effectively

means that authentication sources are cached on a per-session basis, so if you change AUTHENTICATION_BACKENDS, you'll need to clear out session data if you need to force users to re-authenticate using different methods. A simple way to do that is to execute Session.objects.all().delete().

Authorization for inactive users

An inactive user is one that has its is_active field set to False. The ModelBackend and RemoteUserBackend authentication backends prohibits these users from authenticating. If a custom user model doesn't have an is_active field, all users will be allowed to authenticate.

You can use AllowAllUsersModelBackend or AllowAllUsersRemoteUserBackend if you want to allow inactive users to authenticate.

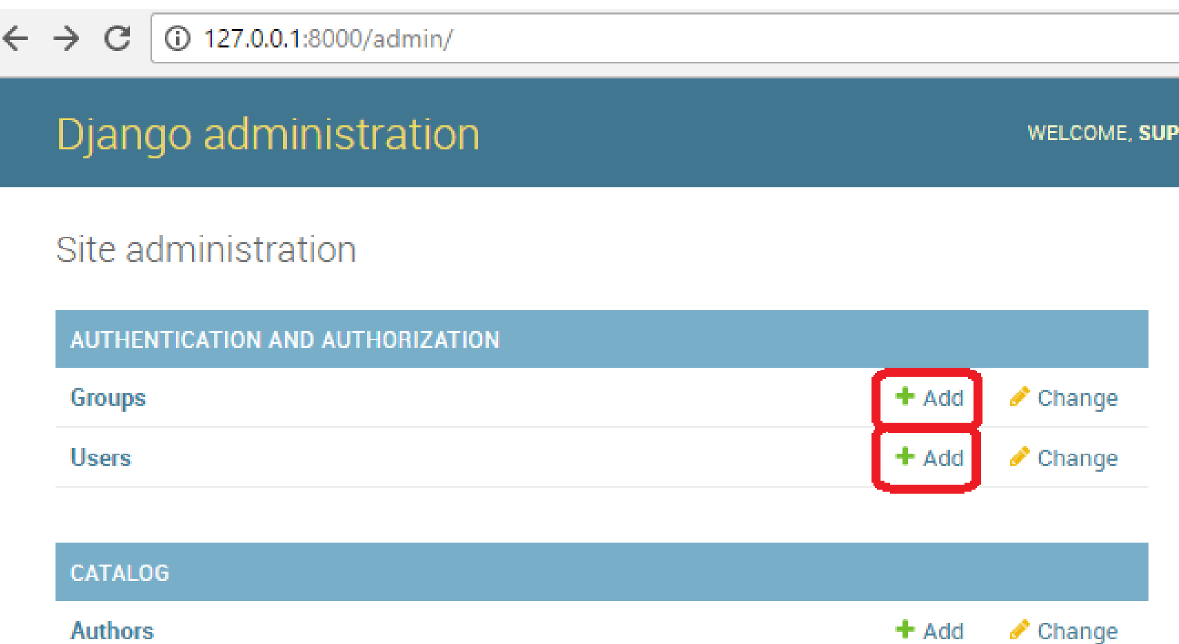
The support for anonymous users in the permission system allows for a scenario where anonymous users have permissions to do something while inactive authenticated users do not.

Do not forget to test for the is_active attribute of the user in your own backend permission methods.

2.2. Apply Permissions & authorization

Custom permissions

To create custom permissions for a given model object, use the permissions model Meta attribute.



This example Task model creates two custom permissions, i.e., actions users can or cannot do with Task instances, specific to your application:

```
class Task(models.Model):
    ...

    class Meta:
        permissions = [
            ("change_task_status", "Can change the status of tasks"),
            ("close_task", "Can remove a task by setting its status as closed"),
        ]
```

The only thing this does is create those extra permissions when you run manage.py migrate (the function that creates permissions is connected to the post_migrate signal). Your code is in charge of checking the value of these permissions when a user is trying to access the functionality provided by the application (changing the status of tasks or closing tasks.) Continuing the above example, the following checks if a user may close tasks:

```
user.has_perm("app.close_task")
```

Extending the existing User model

There are two ways to extend the default User model without substituting your own model. If the changes you need are purely behavioral, and don't require any change to what is stored in the database, you can create a proxy model based on User. This allows for any of the features offered by proxy models including default ordering, custom managers, or custom model methods.

If you wish to store information related to User, you can use a OneToOneField to a model containing the fields for additional information. This one-to-one model is often called a profile model, as it might store non-auth related information about a site user. For example you might create an Employee model:

```
from django.contrib.auth.models import User


class Employee(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    department = models.CharField(max_length=100)
```

Assuming an existing Employee Fred Smith who has both a User and Employee model, you can access the related information using Django's standard related model conventions:

```
>>> u = User.objects.get(username="fsmith")
>>> freds_department = u.employee.department
```

To add a profile model's fields to the user page in the admin, define an InlineModelAdmin (for this example, we'll use a StackedInline) in your app's admin.py and add it to a UserAdmin class which is registered with the User class:

```
from django.contrib import admin
from django.contrib.auth.admin import UserAdmin as BaseUserAdmin
from django.contrib.auth.models import User

from my_user_profile_app.models import Employee


# Define an inline admin descriptor for Employee model
# which acts a bit like a singleton
class EmployeeInline(admin.StackedInline):
    model = Employee
    can_delete = False
    verbose_name_plural = "employee"


# Define a new User admin
class UserAdmin(BaseUserAdmin):
    inlines = [EmployeeInline]


# Re-register UserAdmin
admin.site.unregister(User)
admin.site.register(User, UserAdmin)
```

These profile models are not special in any way - they are just Django models that happen to have a one-to-one link with a user model. As such, they aren't auto created when a user is created, but a django.db.models.signals.post_save could be used to create or update related models as appropriate.

Using related models results in additional queries or joins to retrieve the related data. Depending on your needs, a custom user model that includes the related fields may be your better option, however, existing relations to the default user model within your project's apps may justify the extra database load.

Django's built-in User model can be replaced with a custom one via the AUTH_USER_MODEL setting (e.g., "myapp.MyUser"). For new projects, creating a custom User model (inheriting from AbstractUser) from the start is recommended. Changing mid-project is complex and requires manual database schema adjustments. Reusable apps should *not* implement custom User models. Use get_user_model() and settings.AUTH_USER_MODEL to reference the active User model. Custom User models should inherit from AbstractBaseUser and define USERNAME_FIELD, REQUIRED_FIELDS, and other necessary attributes. A custom manager (extending BaseUserManager) is also required. Built-in auth forms may need adjustments, and django.contrib.admin integration requires specific attributes (is_staff, has_perm, etc.) and registration. PermissionsMixin can be used to integrate Django's permission

framework. Custom User models can break proxy models extending the default User. A full example demonstrating a custom User model with email as username and date of birth is provided.

2.3. Address Authentication in web requests

Authentication in Django web requests is a crucial process that verifies a user's identity and grants them access to specific functionalities within your application. Here's a breakdown of the key concepts and steps involved

A. User Model:

- The foundation of authentication is the User model, typically located in the auth app.
- This model stores user credentials (username, password) and other relevant information.
- You can customize the User model by adding additional fields using custom user models.

B. Authentication Backend:

- Django utilizes a pluggable backend system for authentication.
- By default, Django uses a model backend that relies on the built-in User model for verification.
- You can create custom authentication backends for integrating with external authentication providers (e.g., social login with Facebook, Google).

C. Authentication Views:

- Django provides pre-built views for common authentication tasks like login, logout, and password reset:
  - LoginView: Handles user login attempts and redirects to a success page upon successful authentication.
  - LogoutView: Logs out the current user and redirects to a designated logout page.
  - PasswordResetView and related views: Initiate and manage password reset workflows.

D. Authentication Process:

- A user submits login credentials (usually username and password) through a login form.
- The form data is sent to the LoginView (or a custom login view).
- LoginView authenticates the user using the configured backend (usually the model backend).
- If credentials are valid, the user is authenticated, and a session is created.
- The user is then redirected to the designated success page.

E. Request.user Attribute:

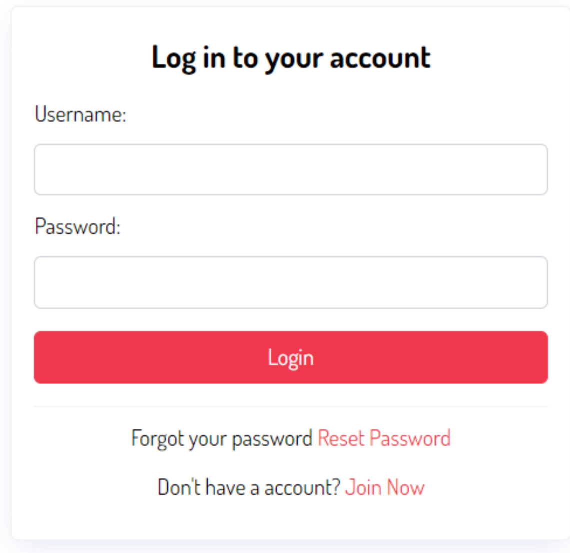
- Once a user is authenticated, a request.user attribute becomes available in every subsequent request object.
- This attribute contains information about the authenticated user, allowing you to personalize content or restrict access based on user roles or permissions.

F. Permissions (Optional):

- While authentication verifies identity, permissions define what actions a user can perform.
- Django's permission system allows you to control access to views, models, and other functionalities based on user groups or specific user attributes.

Here's an example of a basic login view using Django's LoginView::

```
from django.contrib.auth import login
from django.contrib.auth.views import LoginView
from django.shortcuts import render, redirect

def my_login(request):
  if request.method == 'POST':
    username = request.POST['username']
    password = request.POST['password']
    user = authenticate(request, username=username, password=password)
    if user is not None:
      login(request, user)
      # Redirect to success page
      return redirect('home')
    else:
      # Invalid credentials error handling
      return render(request, 'login.html', {'error_message': 'Invalid username or
password'})
  return render(request, 'login.html')

# Using LoginView (class-based view)
class MyLoginView(LoginView):
  template_name = 'login.html'
  success_url = '/'  # Redirect to homepage on successful login
```



By understanding these concepts and implementing authentication mechanisms, you can ensure secure access control and personalized experiences for users in your Django web application.

Additional Considerations

- Secure password storage with hashing algorithms (e.g., bcrypt) is essential.
- Implement CSRF (Cross-Site Request Forgery) protection to prevent unauthorized form submissions.
- Consider using session-based authentication or token-based authentication (e.g., JWT) depending on your application's requirements.

Self-Check Sheet 2: Apply custom user management

1. What is the primary purpose of authentication backends in Django?

Answer:

a) To store user passwords securely.
b) To extend the default user model.
c) To implement authentication against different sources.
d) To manage user permissions.

2. How does the order of authentication backends in the AUTHENTICATION_BACKENDS **setting affect authentication?**

Answer:

a) It has no effect; all backends are checked simultaneously.
b) Django stops processing backends after the first successful authentication.
c) Django checks backends in reverse order.
d) The order only affects permission checks.

3. What does the get_user(user_id) **method of an authentication backend do?**

Answer:

a) It retrieves the user object associated with the provided ID.
b) It verifies the validity of a username and password.
c) It creates a new user object.
d) It grants permissions to a user.

4. How can a custom authentication backend handle authorization for anonymous users?

Answer:

a) By checking the user.is_active attribute.
b) By defining custom permission checks in the backend.
c) There is no built-in way to authorize anonymous users.
d) Anonymous users are always granted all permissions.

5. What are two ways to extend the default User model in Django?

Answer:

a) By creating a custom authentication backend and a profile model.
b) By using a proxy model or a OneToOneField to a profile model.
c) By overriding the User class directly.
d) There is no way to extend the default User model.

Answer Key 2: Apply custom user management

1. What is the primary purpose of authentication backends in Django?

Answer: c) To implement authentication against different sources.

2. How does the order of authentication backends in the AUTHENTICATION_BACKENDS **setting affect authentication?**

Answer: b) Django stops processing backends after the first successful authentication.

3. What does the get_user(user_id) **method of an authentication backend do?**

Answer: a) It retrieves the user object associated with the provided ID.

4. How can a custom authentication backend handle authorization for anonymous users?

Answer: b) By defining custom permission checks in the backend.

5. What are two ways to extend the default User model in Django?

Answer: b) By using a proxy model or a OneToOneField to a profile model.

Job Sheet-2: Implement User Objects with Permissions and Authorization

UoC Cover

OU-ICT-WADP-02- L6-V1: Apply User Management

Working Procedure / Steps

1. **Define Custom User Model (Optional):**
    - Django provides a default User model for authentication. However, you can create a custom model if you need additional user information or different identification methods (e.g., email instead of username).
    - Consider using AbstractUser or AbstractBaseUser as a base class depending on your requirements.
    - Remember to update the AUTH_USER_MODEL setting in your project's settings.py to point to your custom model class.
2. **Create User Objects:**
    - Use Django's user management functionalities to create user accounts:
        - User.objects.create_superuser(username, email, password): Creates a superuser with full administrative privileges.
        - User.objects.create_user(username, email, password, is_staff=True): Creates a staff user with access to the admin panel but not full administrative permissions.
        - User.objects.create_user(username, email, password): Creates a regular user without any special permissions.
3. **Define Custom Permissions:**
    - Use the permissions meta option within your custom model (or the Task model in the provided example) to define custom permissions specific to your application.
    - Each permission is a tuple containing a codename (e.g., "change_task_status") and a human-readable description (e.g., "Can change the status of tasks").
4. **Implement Authorization Checks:**
    - Use the user.has_perm("app.permission_codename") method to check if a user has a specific permission before allowing them to perform an action.
    - Integrate these checks into your application logic to restrict access based on user permissions.

Specification Sheet 2: Implement User Objects with Permissions and Authorization

Technical Requirements

- **Programming Language:** Python 3.7 or later
- **Framework:** Django 3.2 or later

Tools and Equipment

- **Computer:** A computer with sufficient processing power, RAM, and storage.
- **Text Editor/IDE:** Visual Studio Code, PyCharm, Sublime Text (or any preferred code editor).

Reference

1. https://forum.djangoproject.com/
2. https://www.webforefront.com/django/

Review of Competency

Below is yourself assessment rating for module "Apply User Management"

Assessment of performance Criteria	Yes	No
Django default user management is applied.	☐	☐
Users, and groups are created.	☐	☐
Users are assigned to necessary groups.	☐	☐
User objects are implemented.	☐	☐
Permissions & authorization are applied.	☐	☐
Authentication in web requests is addressed.	☐	☐

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of ‘Applying User Management’ (Occupation: Web Application Development with Python , Level-4) for National Skills Certificate is developed by NSDA with the assistance of SAMAHAR Consultants Ltd.in the month of June, 2024 under the contract number of package SD-9C dated 15th January 2024.

SL No.	Name and Address	Designation	Contact Number
1	Khan Mohammad Mahmud Hasan	Writer	Cell: 01714087897 Email: kmmhasan@gmail.com
2	A K M Mashuqur Rahman Mazumder	Editor	Cell: 01676323576 Email : mashuq.odelltech@odell.com.bd
3	Khan Mohammad Mahmud Hasan	Co-Ordinator	Cell: 01714087897 Email: kmmhasan@gmail.com
4	Md. Saif Uddin	Reviewer	Cell:01723004419 Email: engrbd.saif@gmail.com