Competency Based Learning Material (CBLM)

Web Application Development with Python

Level-4

Module: Developing Dynamic Web Pages

Code: CBLM- OU-ICT-WADP-02-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
Level: 10-11, Biniyog Bhaban,
E-6 / B, Agargaon, Sher-E-Bangla Nagar Dhaka-1207, Bangladesh.
Email: ec@nsda.gov.bd
Website: www.nsda.gov.bd.
National Skills Portal: http:\\skillsportal.gov.bd

This Competency Based Learning Materials (CBLM) on "**Developing Dynamic Web Pages**" 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 Sheets.** 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 Sheets** 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 Sheets. 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 Django URLs and Views 2

Learning Experience 1: Use Django urls and Views 3
Information Sheet 1: Use Django URLs and Views 4
Self-Check Sheet 1: Use Django URLs and Views 38
Answer Key 1: Use Django urls and Views 39
Task Sheet-1.1: **Simple Blog Web Application Development** 40
Specification Sheet-1.1: **Simple Blog Web Application Development Items** 42
Task Sheet-1.2: T**o-Do List Web Application Development** 43
Specification Sheet-1.2: T**o-Do List Web Application Development** 45
Task Sheet-1.3: **E-commerce Product Listing Web Application** 46
Specification Sheet-1.3: **E-commerce Product Listing Web Application** 48
Task Sheet-1.4: **Recipe Website Development** 49
Specification Sheet-1.4: **Recipe Website Development** 51
Task Sheet-1.5: **Simple Library Management System** 52
Specification Sheet-15: **Simple Library Management System** 54

Learning Outcome 2: Apply Django Templates 55

Learning Experience 2: Apply Django Templates 56
Information Sheet 2: Apply Django Templates 57
Self-Check Sheet 2: Apply Django Templates 63
Answer Key 2: Apply Django Templates 64
Task Sheet-2.1: **Develop a Personal Portfolio Website** 65
Specification Sheet-2.1: **Develop a Personal Portfolio Website** 67
Task Sheet-2.2: **Develop E-commerce Product Listing Page** 68
Specification Sheet-2.2: **Develop E-commerce Product Listing Page** 70
Task Sheet-2.3: **Create Blog with Categories and Tags** 71
Specification Sheet-2.3: **Create Blog with Categories and Tags** 73
Task Sheet-2.4: **Develop Recipe Website** 74
Specification Sheet-2.4: **Develop Recipe Website** 76

Learning Outcome 3: Implement Django Application Management 77

Learning Experience 3: Implement Django Application Management 78
Information Sheet 3: Implement Django Application Management 79
Self-Check Sheet 3: Implement Django Application Management 84
Answer Key 3: Implement Django Application Management 85
Task Sheet-3.1: **Simple Library Management System** 85
Specification Sheet-3.1: **Simple Library Management System** 88
Task Sheet-3.2: Create a **Library Management System** 89
Specification Sheet-3.2: Create a **Blog with Comments** 93
Task Sheet-3.3: **Recipe Sharing Site** 94
Specification Sheet-3.3: **Recipe Sharing Site** 97
Task Sheet-3.4: Create a **Learning Management System (LMS)** 98
Specification Sheet-3.4: Create a **Learning Management System (LMS)** 102
Task Sheet-3.5: Create a **Library Management System** 103

Specification Sheet-3.5: Create a **Library Management System** 107
Task Sheet-3.6: **Create a Content Management System (CMS)** 108
Specification Sheet-3.6: **Create a Content Management System (CMS)** 112
Task Sheet-3.7: **Create a Simple Social Media Platform** 113
Specification Sheet-3.7: **Create a Simple Social Media Platform** 118
Task Sheet-3.8: **Create a Simple Project Management Tool** 119
Specification Sheet-3.8: **Create a Simple Project Management Tool** 124
Task Sheet-3.9: **Create an E-commerce Platform with Advanced** 125
Specification Sheet-3.9: **Create a Simple Project Management Tool** 130
Reference 131
Review of Competency 132
Development of CBLM 133

Module Content

Unit of Competency	**Develop Dynamic Web Pages**
Unit Code	OU-ICT-WADP-01-L4-V1
Module Title	**Developing Dynamic Web Pages**
Module Descriptor	This module covers the knowledge, skills and attitudes required to develop dynamic web pages. It includes the task of using Django urls and Views, applying Django Templates and implementing Django Application Management.
Nominal Hours	40 Hours
Lerning Outcome	After completing the practice of the module, the trainees will be able to perform the following jobs: 1. Use Django urls and Views 2. Apply Django Templates 3. Implement Django Application Management

Assessment Criteria

1. Common Url Patterns are applied
2. Url Parameters, Extra Options, and Query Strings are used
3. Url Naming and Namespaces are implemented
4. Url Method Requests are implemented
5. View Method Requests are implemented
6. View Method Responses are implemented
7. Django Template Syntaxes are applied
8. Built-In Django Filters are applied
9. Django settings.py for the Real World is set
10. ALLOWED_HOSTS is defined
11. Application is allowed
12. Static web page resources are accumulated
13. Images, CSS, JavaScript are applied

Learning Outcome 1: Use Django URLs and Views

Assessment Criteria	1. Common Url Patterns are applied 2. URLParameters, Extra Options, and Query Strings are used 3. URL Naming and Namespaces are implemented 4. URL Method Requests are implemented 5. View Method Requests are implemented 6. View Method Responses are implemented
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 Common URL Patterns 2 URL Parameters, Extra Options, and Query Strings 3 URL Naming and Namespaces 4 URL Method Requests 5 View Method Requests 6 View Method Response
Activities/job/Task	1 Simple Blog Web Application Development 2 To-Do List Web Application Development 3 E-commerce Product Listing Web Application 4 Recipe Website Development 5 Simple Library Management System
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 Django urls and Views

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 the learning materials	1. Instructor will provide the learning materials 'Use Django urls and Views'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Interpret the Project"	2. Read Information sheet 1: Use Django urls and Views 3. Answer Self-check 1: Use Django urls and Views 4. Check your answer with Answer key 1: Use Django urls and Views
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Task Sheet-1.1: Simple Blog Web Application Development Task Sheet-1.2: To-Do List Web Application Development Task Sheet-1.3: E-commerce Product Listing Web Application Task Sheet-1.4: Recipe Website Development Task Sheet-1.5: Simple Library Management System

Information Sheet 1: Use Django URLs and Views

Learning Objective:

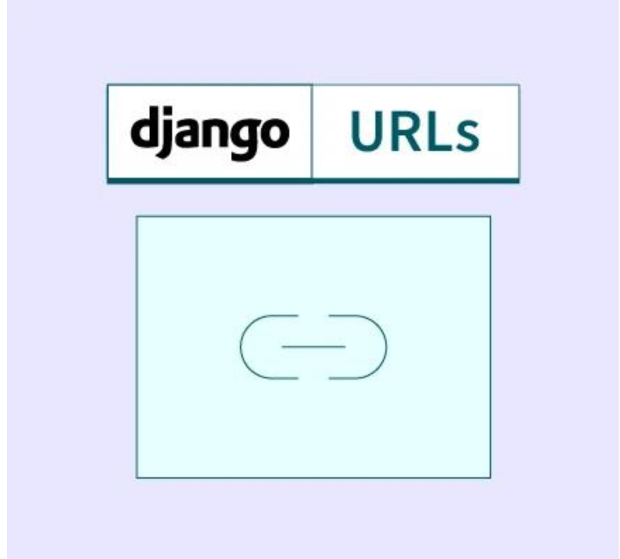
After completion of this information sheet, the learners will be able to explain, define and interpret the following contents:

1.1. Common URL Patterns are applied
1.2. URL Parameters, Extra Options, and Query Strings are used
1.3. URL Naming and Namespaces are implemented
1.4. URL Method Requests are implemented
1.5. View Method Requests are implemented
1.6. View Method Responses are implemented

1.1. Common Url Patterns

URL patterns are an important aspect of any web application, and Django provides a powerful URL routing system that allows you to map URLs to views. In this article, we will discuss some useful URL patterns in Django that can help you build robust and flexible web applications.

a. Basic URL Patterns



The path Function

The path function is the most commonly used function for defining URL patterns in Django. It takes a route and a view as arguments and returns a URL pattern. Here is an example

```
from django.urls import path

from . import views

urlpatterns = [

    path('my-url/', views.my_view, name='my-view'),

]
```

In this example, we define a URL pattern that maps the URL /my-url/ to the my_view function in views.py. The name parameter is optional, but it is a good practice to include it as it allows you to refer to this URL pattern by name in your templates and other parts of your application.

The re_path Function

The re_path function is similar to the path function, but it allows you to define URL patterns using regular expressions. Here is an example

```
from django.urls import re_path
from . import views
urlpatterns = [
    re_path(r'^my-url/(?P<slug>[\w-]+)/$', views.my_view, name='my-view'),
]
```

In this example, we define a URL pattern that maps the URL /my-url/<slug>/ to the my_view function in views.py. The <slug> parameter is a named group in the regular expression, and it will capture any alphanumeric characters, underscores, or hyphens. The captured value will be passed to the my_view function as a keyword argument named slug.

b. Advanced URL Patterns

Including Other URL Patterns

Sometimes you may want to include URL patterns from other applications in your project. You can do this using the include function. Here is an example

```
from django.urls import path, include
urlpatterns = [
    path('my-app/', include('myapp.urls')),
]
```

In this example, we include the URL patterns from the myapp application by specifying its urls.py module.

Naming URL Patterns

Naming URL patterns can be useful when you want to refer to them by name in your templates or other parts of your application. You can do this by providing a name parameter when defining the URL pattern. Here is an example

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

urlpatterns = [
    path('my-url/', views.my_view, name='my-view'),
]
```

In this example, we name the URL pattern my-view. This allows us to refer to this URL pattern by name in our templates using the {% url %} template tag:

My View

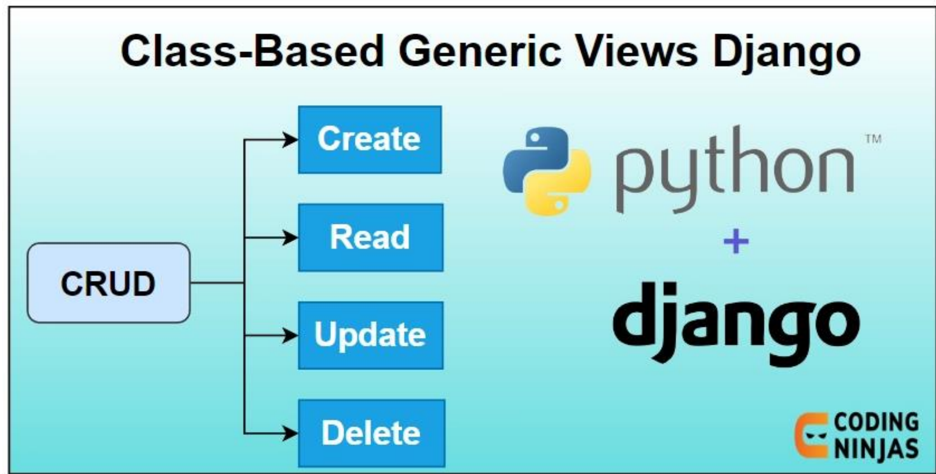
Using Regular Expressions

Django allows you to use regular expressions in your URL patterns, which can be useful for matching dynamic URL segments. Here is an example:

```
from django.urls import re_path
from . import views

urlpatterns = [
    re_path(r'^my-url/(?P<slug>[\w-]+)/$', views.my_view, name='my-view'),
]
```

In this example, we use a regular expression to match URLs that contain a dynamic <slug> segment. The captured value is passed to the my_view function as a keyword argument named slug.

Using Class-Based Views



Django also allows you to use class-based views, which can be useful for organizing your code and reusing common functionality. Here is an example of using a class-based view with a URL pattern

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

urlpatterns = [
    path('my-url/', MyView.as_view(), name='my-view'),
]
```

In this example, we use the as_view() method to convert the MyView class into a view function that can be used with a URL pattern.

Using Namespace

rjdj/**namespace-django**

Use namespace packages in Django

In large projects with multiple apps, naming conflicts can occur when multiple apps use the same view name. You can resolve this by using namespaces. Here is an example

```
from django.urls import path, include

app_name = 'myapp'

urlpatterns = [
    path('my-url/', views.my_view, name='my-view'),
]
```

In this example, we define a namespace for the myapp app by setting the app_name variable to 'myapp'. This allows us to refer to the my_view function in our templates using the {% url %} template tag with the namespace:

```
<a href="{% url 'myapp:my-view' %}">My View</a>
```

1.2. URL Parameters, Extra Options, and Query Strings

Sometimes it's helpful or even necessary to pass url information to the processing construct as a parameter. For example, if you have several urls like /drinks/mocha/, /drinks/espresso/ and /drinks/latte/, the last part of the url represents a drink name. Therefore, it can be helpful or necessary to relay this url information to the processing template to display it or use it in some other way in a view method (e.g. query a database).

To relay this information a Django url path must treat this information as a parameter.

Url parameters are declared differently for both django.urls.path and django.urls.re_path statements, with the former using Django path converters and the latter using standard Python regular expression named groups.

Url parameters with Django path converters and Python regular expression named groups

Path converters are a special Django construct specifically designed to work with django.urls.path statements, whereas named groups are a standard Python regular expression technique employed in django.urls.re_path statements which use regular expressions, as described earlier .

Url parameters can capture any part of a url, whether it's a string, a number or a special set of characters that has to be passed to a template or view method. **Example** illustrates two sets of django.urls.path and django.urls.re_path statements showing how to capture strings and numbers as url parameters.

Django url parameters to capture strings and numbers with django.urls.path and django.urls.re_path

from django.urls import path, re_path

```
# Match string after drinks/ prefix
path('drinks/<str:drink_name>/',...),
# Match one or more characters (non-digit regular expression) after drinks/ prefix
re_path(r'^drinks/(?P<drink_name>\D+)/',...),


# Match integer after stores/ prefix
path('stores/<int:store_id>/',...),
# Match one or more digits (digit regular expression) after stores/ prefix
re_path(r'^stores/(?P<store_id>\d+)/',...),
```

The first line in **example** is a django.urls.path statement to match a url prefixed with drinks/ followed by a string. The <str:drink_name> syntax represents a path converter, where str is the path converter type and drink_name is the url parameter name given to the matching value (e.g. a request for /drinks/mocha/ means drink_name=mocha). The second line is an equivalent django.urls.re_path statement that uses a regular expression named group (?P<drink_name>\D+) to perform that same match as the first line. In this case, the ?P<> syntax tells Django to treat this part of the django.urls.re_path regular expression as a named group and assign its value to a parameter named drink_name declared between <>. The final piece \D+ is a regular expression that represents one or more non-digit characters -- technically a string.

It's very important to understand that url parameters are only captured if the provided value matches a given path converter (e.g str for a string) or named group regular expression (e.g. \D+ for non-digits). If a url request doesn't match a given django.urls.path path converter or django.urls.re_path named group regular expression, Django moves onto the next django.urls.path or django.urls.re_path statement until it finds a matching django.urls.path or django.urls.re_path statement or returns a not found error, in this sense, Django's url path matching/action mechanism works just the same with or without url parameters.

Getting back to **example**, the second django.urls.path statement is used to match a url prefixed with stores/ followed by an integer. The <int:store_id> syntax represents a path converter, where int is the path converter type and store_id is the url parameter name given to the matching value (e.g. a request for /stores/1/ means store_id=1). The second line is an equivalent django.urls.re_path statement that uses a regular expression named group (?P<store_id>\d+) to perform that same match as the thrid line. In this case, the ?P<> syntax tells Django to treat this part of the django.urls.re_path regular expression as a named group and assign its value to a parameter named store_id declared between <>. The final piece \d+ is a regular expression that represents one or more digits -- technically an integer.

There's an important difference you should be aware of between django.urls.path and django.urls.re_path statements that match something other than a string, such as an integer like the last examples in example. Although the statements path('stores/<int:store_id>/',...), and re_path(r'^stores/(?P<store_id>\d+)/',...), both match urls in the form /stores/1/, the store_id parameter data type is different in each case. All parameters for django.urls.re_path statements are treated as strings, this means in a statement like re_path(r'^stores/(?P<store_id>\d+)/',...), the view or template that receives the parameter sees store_id='6', which means that in order to use store_id as an actual integer (e.g. for a mathematical operation) the store_id must be converted to an integer. On the other hand, all parameters in django.urls.path statements are automatically converted to their path convert type, which means in a statement like path('stores/<int:store_id>/',...) that uses an int path converter, the view or template that receives the parameter sees store_id=6, meaning that store_id is an actual integer data type. This last process is helpful because it avoids any additional conversion steps necessary to work with url parameters in views or templates, a process that's illustrated in the upcoming sections.

Note The only thing you shouldn't try to match as url parameters are url query strings -- snippets that are added to urls with ? followed by parameter_name=parameter_value separated by & (e.g./drinks/mocha/?type=cold&size=large). Url query strings in Django should always be processed in view methods.

Just as the url parameter matching process for strings and numbers differs for both django.urls.path and django.urls.re_path statements -- as shown in

example -- matching more complex url parameters is also different for django.urls.path and django.urls.re_path statements.

In order to create more elaborate url parameters for django.urls.re_path statements you must create more elaborate regular expressions. Since django.urls.re_path statements are based on regular expressions, it's simply a matter of incorporating a regular expression that matches a desired set of url characters as a regular expression named group (e.g. (?P<url_parameter_name>regular_expression)).

Creating elaborate url parameters for django.urls.path statements requires exploring Django path converters.

Table: Django path converters for django.urls.path supported by default

Path converter type	**Description**	**Example path converter**
str	Matches any non-empty string, excluding the path separator /	path('drinks/<str:drink_name>/',...) Matches urls like /drinks/espresso/ and /drinks/latte/, where the drink_name parameter receives a value of espresso and latte, respectively, as a standard Python str data type.
int	Matches zero or any positive integer	path('stores/<int:store_id>/',...) Matches urls like /stores/1/ and /stores/435/, where the store_id parameter receives a value of 1 and 435, respectively, as a standard Python int data type.
slug	Matches a slug. In Django, a slug is a normalized ASCII string consisting of	path('about/<slug:page>/',...) Matches urls like /about/contact-us/ and /about/serving-our-customers-since-2000/, where

	lowercase letters, numbers and hyphens (e.g. The slug representation of the string Welcome to the #1 Coffeehouse! is welco me-to-the-1-coffeehouse)	the page parameter receives a value of contact-us and serving-our-customers-since-2000, respectively, as a standard Python str data type.
uuid	Matches a Universally Unique Identifier (UUID), which is a 16 byte number displayed in 5 groups separated by hyphens, in the form 8-4-4-4-12 for a total of 36 characters (e.g. 550e8400-e29b-41d4-a716-446655440000)	path('user/<uuid:user_id>/',...) Matches urls like /user/bae88fa0-a3e8-11e9-a2a3-2a2ae2dbcce4/, where the user_id parameter receives a value of bae88fa0-a3e8-11e9-a2a3-2a2ae2dbcce4, as a standard Python uuid.UUID data type.
django.urls.p ath	Matches any non-empty string, including the path separator /	path('seo/<path:landing>/',...) Matches urls like seo/google/search/keyword/co ffee/, where the landing parameter receives a value of google/search/keyword/coffee, as a standard Python str data type.

As you can see in **Table**, in addition to the str and int path converters illustrated in **example**, it's also possible to match more elaborate sets of url characters that include a slug, a uuid and a django.urls.path. But what happens if the path converters in **Table** 2-4 aren't sufficient for your needs ? You need to create custom path converters which is the topic of the next section.

Custom path converters for django.urls.path statements

In django version 2.0 introduced new way to write url patterns using "path". In django 2.0 regular expression type "url" is removed and added 2 things to work with urls. 1. using "path" and 2. using "re_path". For writing the clean urls django

provided the path converters. It helps us to write clean and neat code and improves code readability.
default path converters provided by dango
The syntax of path converter is "". path converter type is one of the defaults str, int, slug, uuid or a ***custom path converter*** if we have one. Let's see how we can use default path converters in django.

- **str**

```
path('user/<str:username>/detail/', views.user_detail, name='user_detail')
```

In avobe code we have used "str" as a path converter and kwarg name is "username". consider the example url "user/anjaneyulu/detail/". when we access it the url dispatcher will check the string regex whether the url is pattern is matched or not after matching it will return the control to the view (views.user_detail). If url path not matched with the string pattern then it will lookfor other url paths and tries to match with other path patterns if not found it will raise 404 error.

- **int**

```
path('author/<int:pk>/detail/', views.author_detail, name='author_detail')
```

In the above code we have used "int" as a path converter and "pk" as a kwarg name. The path converter already have the "int" regex. So, when we access url path that matches the pattern then the control will we passed to the view. If not it will tries to match for other paths and if not found it will raise a 404 error.

- **slug**

```
path('blog/<slug:post_slug>/', views.blogpost_detail, name='blogpost_detail')
```

In the above code we have used "slug" as a path converter and "post_slug" as a kwarg name that will be passed view. The path converter is defined with the slug regex already. So, it will check the url with the slug regex and returns control to the view. Otherwise it will look for other url paths for matching if none matches it will raise 404 error, saying that url not found.

- **uuid**

```
path('email/<uuid:user_uuid>/confirm/', name='user_email_conform')
```

In above code we have used "uuid" as a path converter and "user_uuid" as a

kwarg name. The path converter already defined with uuid regex pattern. So, it will matches regex with given url path and if matches it will pass control to the view to return the response.

Writing custom path converters in django

A clean url is very important for any web application. Django is providing the best solution to this problem. We can write nice and clean urls using the path converters. Django already provides the default path converters but, in some cases we may need more. For example, If we want to show user details with username and we have some conditions for username like username must start with a letter, then either a-z (lowercase letters), A-Z (uppercase letters), 0-9 (numbers), underscores, periods and hyphens are allowed, and it must be between 8 and 20.

converters.py

```
class UsernamePathConverter:
    regex = '^[a-zA-Z0-9_.-]+$'

    def to_python(self, value):
        # convert value to its corresponding python datatype
        return value

    def to_url(self, value):
        # convert the value to str data
        return value
```

In above code we have created path converter for username and registered it using "register_converter" function with name "username" and we used the path converter in the url path.

1.3. Url Naming and Namespaces

In each app, there is a urls.py file that defines the list of URL patterns for that app. Each pattern is constructed by django.urls.path() function. Its arguments are a URL path string, the name of the view function to be mapped to it and an optional argument name.

The following is the urls.py of an app:

```
#demoapp/urls.py
from django.urls import path
from . import views

  urlpatterns = [
    path('', views.index, name='index'),
    path('login/', views.login, name='login')
]
```

It is included in the project's urlpatterns.

```
from django.contrib import admin
from django.urls import path, include

  urlpatterns = [
    path('admin/', admin.site.urls),
    path('demo/', include('demoapp.urls')),
]
```

Normally, the client's request URL is mapped against the function so that the application flow is directed toward it.

The URL references used internally by the application are hard-coded strings. For example, a form template's action attribute points towards /demoapp/login URL so that when the form is submitted, the login view mapped to this URL is invoked.

```
<form action='/demoapp/login', method='POST>
    # ...
</form>
```

reverse() function

However, the hard-coded URLs make the application less scalable and difficult to maintain as the project grows. In such a case, you can obtain the URL from the name parameter used in the path() function.

Start the Django shell.

```
python manage.py shell
```

Django's reverse() function returns the URL path to which it is mapped.

```
>>> from django.urls import reverse
>>> reverse('index')
'/demo/'
```

The problem comes when the view function of the same name is defined in more than one app. This is where the idea of a namespace is needed.

Application namespace

The application namespace is created by defining app_name variable in the application's urls.py and assigning it the name of the app. In the demoapp/urls.py script, make the change using the following code:

```
#demoapp/urls.py
from django.urls import path
from . import views
app_name='demoapp'
urlpatterns = [
    path('', views.index, name='index'),
]
```

The app_name defines the application namespace so that the views in this app are identified by it.

To obtain the URL path of the index() function, call the reverse() function by prepending the namespace to it.

```
>>> reverse('demoapp:index')
'/demo/'
```

To appreciate the advantage of defining a namespace, add another app in the project, for example, newapp. Provide an index() view function in it and define app_name in its URLConf file (that is urls.py).

```
#newapp/urls.py
from django.urls import path
from . import views
app_name='newapp'
urlpatterns = [
    path('', views.index, name='index'),
]
```

Update the project's urls.py.

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    path('demo/', include('demoapp.urls')),
    path('newdemo/', include('newapp.urls')),
]
```

The reverse() function is executed for index view in newapp namespace.

```
>>> reverse('newapp:index')
'/newdemo/'
```

You can see that Django differentiates between same-named URLs in multiple apps with application namespace.

Instance namespace

You can also use the namespace parameter in the include() function while adding an app's urlpattern to that of a project.

```
#in demoproject/urls.py
urlpatterns=[
    # ...
    path('demo/', include('demoapp.urls', namespace='demoapp')),
    # ...
]
```

This namespace is called the instance namespace.

Using namespace in view

Suppose you want the user to be conditionally redirected to the login view from inside another view. You need to obtain the URL of the login view and send the control to it with HttpResponsePermanentRedirect.

```
from django.http import HttpResponsePermanentRedirect
from django.urls import reverse

def myview(request):
    ....
    return HttpResponsePermanentRedirect(reverse('demoapp:login'))
```

namespace in the url tag

An HTML form is submitted to the URL specified in the action attribute.

```
<form action="/demoapp/login" method="post">

#form attributes

<input type='submit'>

</form>
```

The form will then be processed by the view mapped to this URL. However, as mentioned above, a hard-coded URL is not desired. Instead, use the url tag of the Django Template Language. It returns the absolute path from the named URL.

Use the url tag to obtain the URL path dynamically, as shown below:

```
<form action="{% url 'login' %}" method="post">
#form attributes
<input type='submit'>
</form>
```

Again, the login view may be present in multiple apps. Use the named URL qualified with the namespace to resolve the conflict.

```
<form action="{% url 'demoapp:login' %}" method="post">
#form attributes
<input type='submit'>
</form>
```

The browser shows the login form as below:

User Name:

Password:

Submit

Thus, the concept of namespace helps in resolving the conflict arising out of multiple apps under the same project having views of the same name.

1.4. Url Method Requests

In Django, URL routing is a fundamental part of handling HTTP requests. Django uses the urls.py file to map URL patterns to views that handle the corresponding requests. The URL patterns can be matched with different HTTP methods like GET, POST, PUT, DELETE, etc., using Django's view functions and class-based views.

Here's an overview of how different URL methods and requests can be handled in Django:

a. Handling URL Requests in Django

In Django, the most basic way to associate a URL pattern with a view is by using the url() function or the path() function (introduced in Django 2.0).

Using path() for URL Routing:

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

urlpatterns = [
    path('home/', views.home_view, name='home'),  # Associating a URL with a view
]
```

b. Handling Different HTTP Methods

Django provides several ways to handle different HTTP methods for the same view. The most common methods are GET, POST, PUT, and DELETE.

Using Function-Based Views (FBVs)

In function-based views, you can check the request.method to handle different HTTP methods.

Example:

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

def home_view(request):
    if request.method == 'GET':
        return HttpResponse("GET request received")
    elif request.method == 'POST':
        return HttpResponse("POST request received")
    else:
        return HttpResponse("Other request method")
```

In this example, the view checks the request.method and returns a different response based on whether the request is a GET, POST, or another method.

Using Class-Based Views (CBVs)

Django also provides class-based views to handle HTTP requests more elegantly. You can define specific methods for each HTTP request method like get(), post(), put(), and delete().

Example using View class for GET and POST:

```
from django.http import HttpResponse
from django.views import View

class HomeView(View):
    def get(self, request):
        return HttpResponse("GET request received in CBV")

    def post(self, request):
        return HttpResponse("POST request received in CBV")
```

c. **Defining URLs for Views**

Function-Based Views (FBVs)

Here's how to associate a function-based view with a URL in Django's urls.py file:

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

urlpatterns = [
    path('home/', home_view, name='home'),  # Associating the view with URL
]
```

Class-Based Views (CBVs)

For class-based views, Django's as_view() method is used to associate the URL with the class-based view:

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

urlpatterns = [
    path('home/', HomeView.as_view(), name='home'),  # Associating CBV with URL
]
```

d. Handling PUT and DELETE Requests (Advanced)

To handle PUT and DELETE requests, you typically use a class-based view (CBV) or a third-party library like Django REST Framework (DRF), which makes handling these HTTP methods more straightforward.

Using Django REST Framework

If you're building an API or want to handle more complex requests like PUT, DELETE, PATCH, etc., the Django REST Framework (DRF) is a great tool. It provides views like APIView, ModelViewSet, and GenericAPIView to simplify handling these requests.

Example using APIView for PUT/DELETE:

```
from rest_framework.views import APIView
from rest_framework.response import Response
from rest_framework import status

class RecipeView(APIView):
    def put(self, request, pk):
        # Handle PUT request logic here
        return Response({"message": "PUT request received"}, status=status.HTTP_200_OK)

    def delete(self, request, pk):
        # Handle DELETE request logic here
        return Response({"message": "DELETE request received"}, status=status.HTTP_204_NO_CONTENT)
```

e. Method-Specific Decorators

Django also provides decorators to handle specific HTTP methods in function-based views.

Using @require_http_methods decorator:

You can restrict your view to only handle certain HTTP methods using the @require_http_methods decorator.

Example:

```
from django.http import HttpResponse
from django.views.decorators.http import require_http_methods

@require_http_methods(["GET", "POST"])
def my_view(request):
    if request.method == 'GET':
        return HttpResponse("GET request received")
    elif request.method == 'POST':
        return HttpResponse("POST request received")
```

This decorator ensures that only the specified HTTP methods (GET and POST in this case) can access the view.

Using @require_GET and @require_POST:

Django also provides more specific decorators for individual HTTP methods:

```
from django.views.decorators.http import require_GET, require_POST

@require_GET
def my_get_view(request):
    return HttpResponse("GET request received")

@require_POST
def my_post_view(request):
    return HttpResponse("POST request received")
```

f. Handling URL Parameters and Query Strings

Django allows you to capture parameters from the URL or query strings. You can access these in the view via request.GET or request.POST for query parameters or form data.

Accessing URL Parameters:

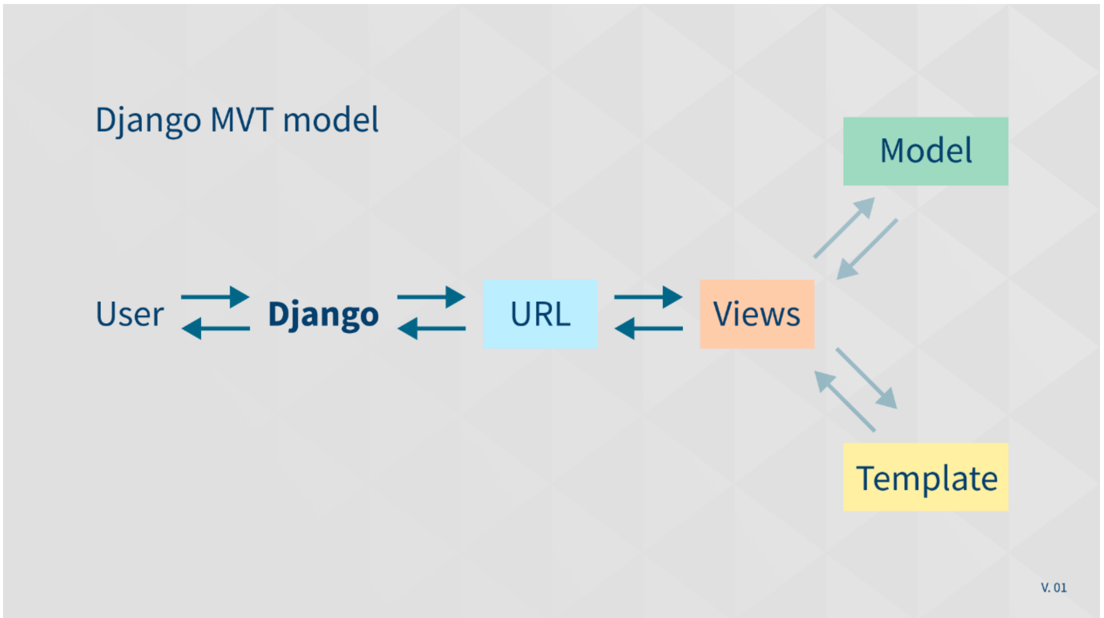
```
# In urls.py
path('recipe/<int:id>/', views.recipe_detail, name='recipe_detail')

# In views.py
def recipe_detail(request, id):
    return HttpResponse(f"Recipe ID: {id}")
```

Accessing Query Parameters:

```
def search_view(request):
    search_query = request.GET.get('q', '')  # Default to empty string if no 'q' param
    return HttpResponse(f"Search query: {search_query}")
```

1.5. View Method Requests

The request reference you've placed unquestionably in view methods up to this point, is an instance of the django.http.request.HttpRequest class[3]. This request object contains information set by entities present before a view method: a user's web browser, the web server that runs the application or a Django middleware class con**figure**d on the application.



The following list shows some of the most common attributes and methods available in a request reference

- request.method.- Contains the HTTP method used for the request (e.g. GET, POST).
- request.GET or request.POST.- Contains parameters added as part of a GET or POST request, respectively. Parameters are enclosed as a django.http.request.QueryDict[4] instance.
- request.POST.get('name',default=None).- Gets the value of the name parameter in a POST request or gets None if the parameter is not present. Note default can be overridden with a custom value.
- request.GET.getlist('drink',default=None).- Gets a list of values for the drink parameter in a GET request or gets an empty list None if the parameter is not present. Note default can be overridden with a custom value.

- request.META.- Contains HTTP headers added by browsers or a web server as part of the request. Parameters are enclosed in a standard Python dictionary where keys are the HTTP header names -- in uppercase and underscore (e.g. Content-Length as key CONTENT_LENGTH).
- request.META['REMOTE_ADDR'].- Gets a user's remote IP address.
- request.user.- Contains information about a Django user (e.g. username, email) linked to the request. Note user refers to the user in the django.contrib.auth package and is set via Django middleware, described later in this chapter.

Set up dictionary in Django view method for access in template from django.shortcuts import render

```
def detail(request,store_id=1,location=None):
    # Create fixed data structures to pass to template
    # data could equally come from database queries
    # web services or social APIs
    STORE_NAME = 'Downtown'
    store_address = {'street':'Main #385','city':'San Diego','state':'CA'}
    store_amenities = ['WiFi','A/C']
    store_menu = ((0,''),(1,'Drinks'),(2,'Food'))
    values_for_template = {'store_name':STORE_NAME,
'store_address':store_address,
                    'store_amenities':store_amenities, 'store_menu':store_menu}
    return render(request,'stores/detail.html', values_for_template)
```

Notice in **example** how the render method includes the values_for_template dictionary. In previous examples, the render method just included the request object and a template to handle the request. In **example**, a dictionary is passed as the last render argument. By specifying a dictionary as the last argument, the dictionary becomes available to the template -- which in this case is stores/detail.html.

The dictionary in example contains keys and values that are data structures declared in the method body. The dictionary keys become references to access the values inside Django templates.

Output view method dictionary in Django templates

Although the next chapter covers Django templates in-depth, the following snippet shows how to output the dictionary values in **example** using the {{}} syntax.

```
<h4>{{store_name}} store</h4>
<p>{{store_address.street}}</p>
<p>{{store_address.city}},{{store_address.state}}</p>
<hr/>
<p>We offer: {{store_amenities.0}} and {{store_amenities.1}}</p>
<p>Menu includes : {{store_menu.1.1}} and {{store_menu.2.1}}</p>
```

The first declaration {{store_name}} uses the standalone key to display the Downtown value. The other access declarations use dot(.) notation because the values themselves are composite data structures.The store_address key contains a dictionary, so to access the internal dictionary values you use the internal dictionary key separated by a dot(.).store_address.street displays the street value, store_address.city displays the city value and store_address.state displays the state value.The store_amenities key contains a list which uses a similar dot(.) notation to access internal values. However, since Python lists don't have keys you use the list index number. store_amenities.0 displays the first item in list store_amenities and store_amenities.1 displays the second item in list store_amenities.

The store_menu key contains a tuple of tuples which also requires a number on account of the lack of keys. {{store_menu.1.1}} displays the second tuple value of the second tuple value of store_menu and {{store_menu.2.1}} displays the second tuple value of the third tuple of store_menu.

1.6. View Method Responses

The render() method to generate view method responses you've used up to this point is actually a shortcut. You can see toward the top of **example**, the render() method is part of the django.shortcuts package.

This means there are other alternatives to the render() method to generate a view response, albeit the render() method is the most common technique. For starters,

there are three similar variations to generate view method responses with data backed by a template, as illustrated in **example**.

Django view method response alternatives

```
--------------------------------------------------------------------------
# Option 1)
from django.shortcuts import render

def detail(request,store_id=1,location=None):
    ...
    context = {}
    return render(request,'stores/detail.html', context)

--------------------------------------------------------------------------
# Option 2)
from django.template.response import TemplateResponse

def detail(request,store_id=1,location=None):
    ...
    context = {}
    return TemplateResponse(request, 'stores/detail.html', context)

--------------------------------------------------------------------------
# Option 3)
from django.http import HttpResponse
from django.template import loader

def detail(request,store_id=1,location=None):
     ...
     context = {}
     t = loader.get_template('stores/detail.html')
     return HttpResponse(t.render(context))
```

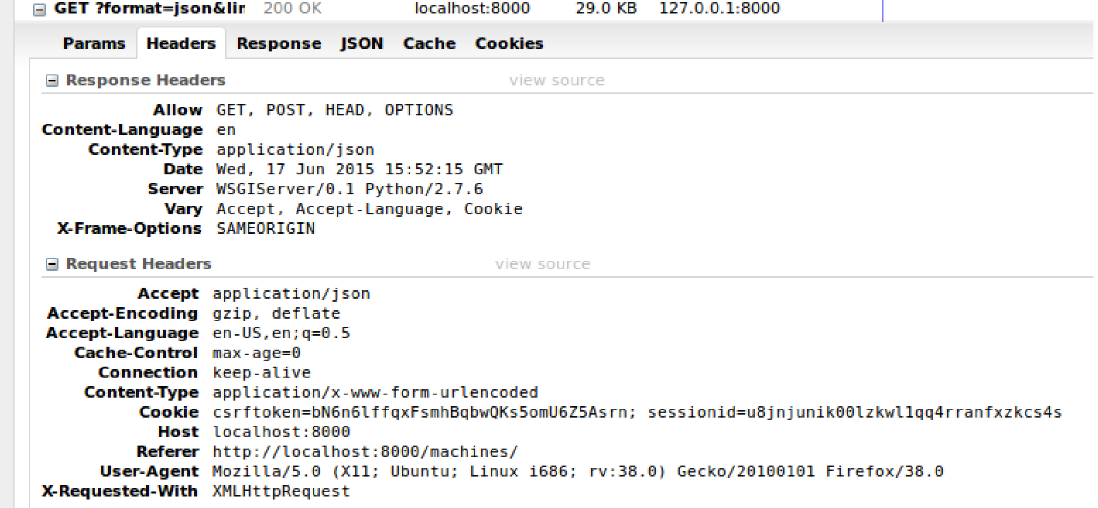
The first option in **example** is the django.shortcuts.render() method which shows three arguments to generate a response: the (required) request reference, a (required) template route and an (optional) dictionary -- also known as the context -- with data to pass to the template.

There are actually three more (optional) arguments for the render() method which are not shown in content_type which sets the HTTP Content-Type header for the response and which defaults to text/html; status which sets the HTTP Status code for the response which defaults to 200 that represents an *HTTP 200 OK* value; and using to specify the template engine -- either jinja2 or django -- to generate the response.

In the django.template.response.TemplateResponse() class, which in terms of input is nearly identical to the render() method. The difference between the two variations is, TemplateResponse() can alter a response once a view method is finished (e.g. via middleware), where as the render() method is considered the last step in the lifecycle after a view method finishes. You should use TemplateResponse() when you foresee the need to modify view method responses in multiple view methods after they finish their work, a technique that's discussed in a later section in this chapter on view method middleware.

Response options for HTTP Status and Content-type headers

Browsers set HTTP headers in requests to tell applications to take into account certain characteristics for processing. Similarly, applications set HTTP headers in responses to tell browsers to take into account certain characteristics for the content being sent out. Among the most important HTTP headers set by applications like Django are Status and Content-Type.



The HTTP Status header is a three digit code number to indicate the response status for a given request. Examples of Status values are 200 which is the standard response for successful HTTP requests and 404 which is used to indicate a requested resource could not be found. The HTTP Content-Type header is a MIME(Multipurpose Internet Mail Extensions) type string to indicate the type of content in a response. Examples of Content-Type values are text/html which is the standard for an HTML content response and image/gif which is used to indicate a response is a GIF image.

By default and unless there's an error, all Django view methods that create a response with django.shortcuts.render(), a TemplateResponse() class or HttpResponse() class -- illustrated in **example** create a response with the HTTP Status value set to 200 and the HTTP Content-Type set to text/html. Although these default values are the most common, if you want to send a

different kind of response (e.g. an error or non-HTML content) it's necessary to alter these values.

Overriding HTTP Status and Content-Type header values for any of the three options in **example** is as simple as providing the additional arguments status and/or content_type. **Example** illustrates various examples of this process.

HTTP Content-type and HTTP Status for Django view method responses from django.shortcuts import render

 # No method body(s) and only render() example provided for simplicity
 # Returns content type text/plain, with default HTTP 200
return render(request,'stores/menu.csv', context, content_type='text/plain')

 # Returns HTTP 404, wtih default text/html
 # NOTE: Django has a built-in shortcut & template 404 response, described in the next section
return render(request,'custom/notfound.html',status=404)

 # Returns HTTP 500, wtih default text/html
 # NOTE: Django has a built-in shortcut & template 500 response, described in the next section
return render(request,'custom/internalerror.html',status=500)

 # Returns content type application/json, with default HTTP 200
 # NOTE: Django has a built-in shortcut JSON response, described in the next section
return render(request,'stores/menu.json', context, content_type='application/json')

The first example in **example** is designed to return a response with plain text content. Notice the render method content_type argument. The second and third examples in **example** set the HTTP Status code to 404 and 500. Because the HTTP Status 404 code is used for resources that are not found, the render method uses a special template for this purpose. Similarly, because the HTTP Status 500 code is used to indicate an error, the render method also uses a special template for this purpose.

The fourth and last example in **example** is designed to return a response with JavaScript Object Notation(JSON) content. The HTTP Content-Type application/json is a common requirement for requests made by browsers that consume JavaScript data via Asynchronous JavaScript (AJAX).

Tip Django has built-in shortcuts and templates to deal with HTTP Status codes 404 and 500, as well as a JSON short-cut response, all of which are described in the next section and that you can use instead of the examples in **example**.

Built-in response shortcuts and templates for common HTTP Status: 400 (Bad Request), 403 (Forbidden), 404 (Not Found) & 500 (Internal Server Error)

Although you'll commonly use the response syntax samples in **example**s 2-24, they can be verbose when all you're trying to do is return a quick response without the need to create/reference a template or pass custom values to a response. For example, you can make evaluations in a Django view like if article_id < 100: or if unpayed_subscription: and based on the result respond with a one-liner HTTP 404 Not Found response or HTTP 403 Forbidden response.

Table: illustrates a series of shortcuts to trigger the most common HTTP status responses.

Table: Django shortcut exceptions to trigger HTTP statuses

HTTP status code	Python code sample
400 (Bad Request)	from django.core.exceptions import BadRequest raise BadRequest
403 (Forbidden)	from django.core.exceptions import PermissionDenied raise PermissionDenied
404 (Not Found)	from django.http import Http404 raise Http404
500 (Internal Server Error)	raise Exception

As you can see in the examples in **Table** 2-4, the shortcut syntax is straightforward. For example, for a Django view clause like if article_id < 100: you can do raise Http404 or for a Django view clause like if unpayed_subscription: you can do raise PermissionDenied.

Tip: Django automatically handles not found pages raising HTTP 404 and unhandled exceptions raising HTTP 500, so you don't need to explicilty add raise Http404 or raise Exception everywhere.

So what is the actual content sent in a response besides the HTTP status when an exception from **Table** is triggered ? The default for HTTP 403 (Forbidden) is a single line HTML page that says "403 Forbidden". For 400 (Bad Request), HTTP 404 (Not Found) and HTTP 500 (Internal Server Error), it depends on the DEBUG value in settings.py.

If a Django project has DEBUG=True in settings.py: HTTP (400) (Bad Request) generates a page with the message added to BadRequest (e.g.raise BadRequest('Invalid hours value')) as well as a traceback report -- as illustrated in **figure**; HTTP 404 (Not Found) generates a page with valid urls, hinting the user on what's available -- as illustrated in **figure**; and HTTP 500 (Internal Server Error) generates a page with a traceback report associated with the Exception -- as illustrated in **figure**.

If a Django project has DEBUG=False in settings.py: HTTP 400 (Bad Request) generates a single line HTML page that says "Bad Request (400)"; HTTP 404 (Not Found) generates a single line HTML page that says "Not Found. The requested resource was not found on this server."; and HTTP 500 (Internal Server Error) generates a single line HTML page that says "Server Error (500)".

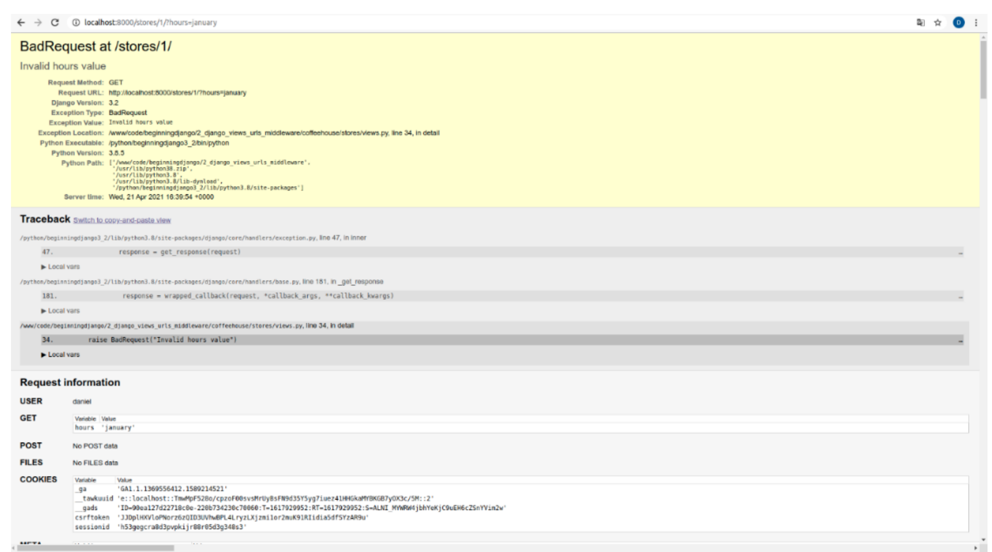


Figure :HTTP 400 for Django project when DEBUG=True

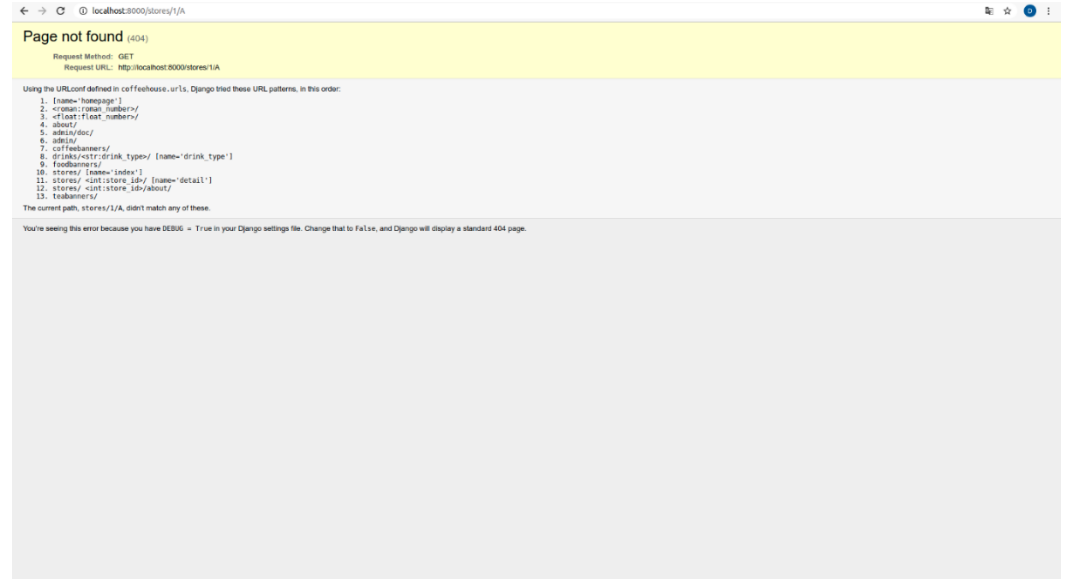


Figure - HTTP 404 for Django project when DEBUG=True

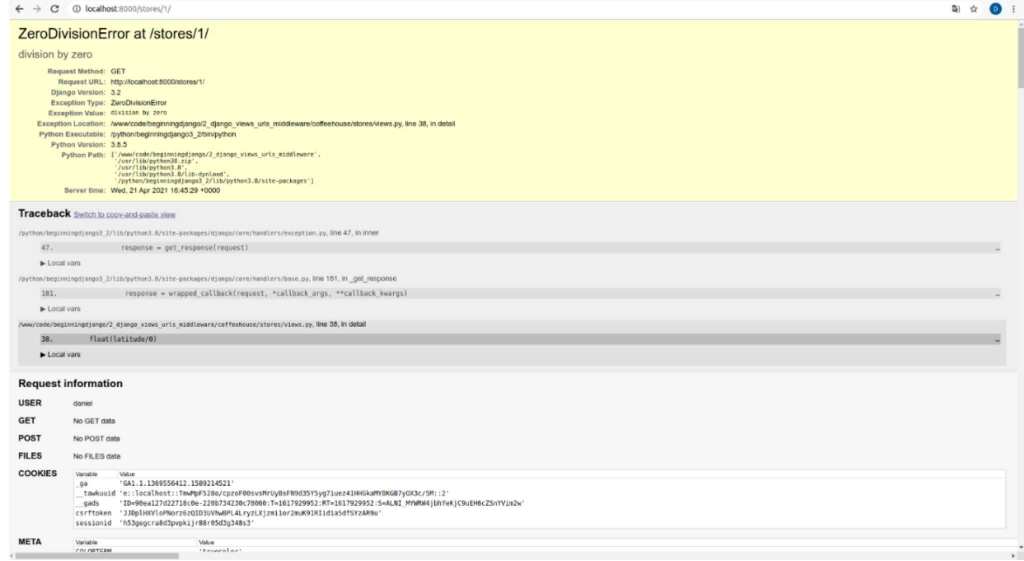


Figure - HTTP 500 for Django project when DEBUG=True

It's also possible to override the default response page for all the previous HTTP codes -- 400, 403, 404 & 500 with custom templates. To use a custom response page, you need to create a template with the desired HTTP code and .html extension. For example, for HTTP 403 you would create the 403.html template and for HTTP 500 you would create the 500.html template. All these custom HTTP response templates need to be placed in a folder defined in the DIRS list of the TEMPLATES variable so Django finds them before it uses the default HTTP response templates.

Caution: Custom 400.html, 404.html and 500.html pages only work when DEBUG=False

If DEBUG=True, it doesn't matter if you have 400.html, 404.html or 500.html templates in the right location, Django

uses the default response behavior illustrated **figure**s, respectively. You need to set DEBUG=False for the custom 400.html, 404.html and 500.html templates to work. When a custom 403.html is detected it will always be used, irrespective of the DEBUG value.

On certain occasions, using custom HTTP response templates may not be enough. For example, if you want to add context data to a custom template that handles an HTTP response, you need to customize the built-in Django HTTP view methods themselves, because there's no other way to pass data into this type of template. To customize the built-in Django HTTP view methods you need to declare special handlers in a project's urls.py file. **Example** illustrates the urls.py file with custom handlers for Django's built-in HTTP Status view methods.

Override built-in Django HTTP Status view methods in urls.py

```
# Contents coffeehouse/urls.py

# Overrides the default 400 handler django.views.defaults.bad_request
handler400 = 'coffeehouse.utils.views.bad_request'
# Overrides the default 403 handler django.views.defaults.permission_denied
handler403 = 'coffeehouse.utils.views.permission_denied'
# Overrides the default 404 handler django.views.defaults.page_not_found
handler404 = 'coffeehouse.utils.views.page_not_found'
# Overrides the default 500 handler django.views.defaults.server_error
handler500 = 'coffeehouse.utils.views.server_error'

urlpatterns = [....
]
```

Other built-in response shortcuts for in-line & streamed content

In addition to the HTTP responses presented in the last section in **Table** 2-4, there are other Django HTTP shortcut responses for less commonly used HTTP codes. **Table** illustrates the different shortcuts to trigger HTTP redirects: HTTP 301 (Permanent Redirect) or HTTP 302 (Redirect), where the response just requires a redirection url.

Table : Django shortcuts for HTTP redirects

HTTP status code	Python code sample
301 (Permanent Redirect)	from django.http import HttpResponsePermanentRedirect return HttpResponsePermanentRedirect("/")
302 (Redirect)	from django.http import HttpResponseRedirect return HttpResponseRedirect("/")

Both samples in **Table** redirect to an application's home page (i.e."/"). However, you can also set the redirection to any application url or even a full url on a different domain (e.g.http://maps.google.com/).

In addition to response redirection shortcuts, Django also offers a series of response shortcuts where you can add in-line responses. **Table** illustrates the various other shortcuts for HTTP status codes with in-line content responses.

Table: Django shortcuts for in-line and streaming content responses

Purpose or HTTP Status code	Python code sample
304 (NOT MODIFIED)	from django.http import HttpResponseNotModified return HttpResponseNotModified()*
400 (BAD REQUEST)	from django.http import HttpResponseBadRequest return HttpResponseBadRequest(" The request doesn't look right ")
404 (NOT FOUND)	from django.http import HttpResponseNotFound return HttpResponseNotFound(" Ups, we can't find that page ")
403 (FORBIDDEN)	from django.http import HttpResponseForbidden return HttpResponseForbidden("Can't look at anything here",content_type="text/plain")

405 (METHOD NOT ALLOWED)	from django.http import HttpResponseNotAllowed return HttpResponseNotAllowed(" Method not allowed ")
410 (GONE)	from django.http import HttpResponseGone return HttpResponseGone("No longer here",content_type="text/plain")
500 (INTERNAL SERVER ERROR)	from django.http import HttpResponseServerError return HttpResponseServerError(" Ups, that's a mistake on our part, sorry! ")
JSON (In-line response that serializes data to JSON, defaults to HTTP 200 and content type application/json)	from django.http import JsonResponse data_dict = {'name':'Downtown','address':'Main #385','city':'San Diego','state':'CA'} return JsonResponse(data_dict)
Streaming (In-line response that stream data, defaults to HTTP 200 and streaming content which is an iterator of strings)	from django.http import StreamingHttpResponse return StreamingHttpResponse(large_data_structure)
Files (In-line response that stream binary files, defaults to HTTP 200 and streaming content)	from django.http import FileResponse return FileResponse(open('Report.pdf','rb'))
Generic - All purpose (In-line response with any HTTP status code, defaults to HTTP 200)	from django.http import HttpResponse return HttpResponse(" Django in-line response ")

* The HTTP 304 status code indicates a 'Not Modified' response, so you can't send content in the response, it should always be empty.

As you can see in the samples in **Table**, there are multiple shortcuts to generate different HTTP Status responses with in-line content and entirely forgo the need to use a template. In addition, you can see the shortcuts in **Table** can also accept the content_type argument if the content is something other than HTML (i.e. content_type=text/html).

Since non-HTML responses have become quite common in web applications, you can see **Table** also shows three Django built-in response shortcuts to output non-HTML content. The JsonResponse class is used to transform an in-line response into JavaScript Object Notation (JSON). Because this response converts the payload to a JSON data structure, it automatically sets the content type to application/json. The StreamingHttpResponse class is designed to stream a response without the need to have the entire payload in-memory, a scenario that's helpful for large payload responses. The FileResponse class -- a subclass of StreamingHttpResponse -- is designed to stream binary data (e.g. PDF or image files).

Self-Check Sheet 1: Use Django URLs and Views

1. Which of the following is NOT a built-in Django shortcut exception to trigger a specific HTTP status code?

a) from django.core.exceptions import BadRequest

b) from django.http import Http404

c) from django.core.exceptions import PermissionDenied

d) raise Exception

2. When DEBUG=True in your Django project, what does a raised Http404 exception display?

a) A single line HTML page saying "Not Found. The requested resource was not found on this server."

b) A page with valid URLs, hinting the user on what's available.

c) A page with a traceback report associated with the Exception

d) A single line HTML page saying "404 Forbidden"

3. How can you override the default Django response page for HTTP status codes like 400 (Bad Request)?

a) Set DEBUG=False in your settings.py

b) Create a custom template with the desired HTTP code and .html extension.

c) Use a different shortcut exception like PermissionDenied.

d) There is no way to override the default response pages.

4. In which scenario would using a custom Django view method be necessary compared to a shortcut exception?

a) When you want to add context data to a custom template for an HTTP response.

b) When you want to raise a specific HTTP status code exception. c) There is no difference; both methods achieve the same result.

d) Custom view methods are only used for handling successful requests (HTTP 200).

5. Which of the following Django shortcuts is used to create a response that serializes data to JSON?

a) from django.http import HttpResponseBadRequest

b) from django.http import HttpResponseNotFound

c) from django.http import JsonResponse

d) from django.http import HttpResponse

Answer Key 1: Use Django urls and Views

1. Which of the following is NOT a built-in Django shortcut exception to trigger a specific HTTP status code?

a) from django.core.exceptions import BadRequest

b) from django.http import Http404

c) from django.core.exceptions import PermissionDenied

d) raise Exception **Correct Answer**

2. When DEBUG=True in your Django project, what does a raised Http404 exception display?

a) A single line HTML page saying "Not Found. The requested resource was not found on this server."

b) A page with valid URLs, hinting the user on what's available. **Correct Answer**

c) A page with a traceback report associated with the Exception

d) A single line HTML page saying "404 Forbidden"

3. How can you override the default Django response page for HTTP status codes like 400 (Bad Request)?

a) Set DEBUG=False in your settings.py

b) Create a custom template with the desired HTTP code and .html extension. **Correct Answer**

c) Use a different shortcut exception like PermissionDenied.

d) There is no way to override the default response pages.

4. In which scenario would using a custom Django view method be necessary compared to a shortcut exception?

a) When you want to add context data to a custom template for an HTTP response. **Correct Answer**

b) When you want to raise a specific HTTP status code exception. c) There is no difference; both methods achieve the same result.

d) Custom view methods are only used for handling successful requests (HTTP 200).

5. Which of the following Django shortcuts is used to create a response that serializes data to JSON?

a) from django.http import HttpResponseBadRequest

b) from django.http import HttpResponseNotFound

c) from django.http import JsonResponse **Correct Answer**

d) from django.http import HttpResponse

Task Sheet-1.1: **Simple Blog Web Application Development**

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Detailed Steps for Implementation:

Step 1: Set Up the Project

- 1.1 Create a new Django project named simple_blog.
- 1.2 Set up a new Django app named blog to handle the blog posts.

Step 2: Create Models

- 2.1 Define a Post model with the following fields:
  - title (CharField)
  - content (TextField)
  - published_date (DateTimeField)
  - category (ForeignKey to a Category model)
  - slug (SlugField)
- 2.2 Define a Category model with:
  - name (CharField)
  - slug (SlugField)

Step 3: Create Views

- 3.1 Define a view for the homepage (index):
  - Fetch the latest posts from the database (using Post.objects.all() or Post.objects.order_by('-published_date')).
  - Pass the posts to the template for rendering.
- 3.2 Define a view for post detail (post_detail):
  - Fetch a specific post based on its post_id.
  - Pass the post to the template for rendering.
- 3.3 Define a view for category posts (category_list):
  - Fetch posts that belong to a specific category (using Post.objects.filter(category=category)).

- Pass the posts and category details to the template.

Step 4: Configure URLs

- 4.1 Add URL patterns for the following:
    - / → Homepage view (index)
    - /post/<int:post_id>/ → Post detail view (post_detail)
    - /category/<slug:category_slug>/ → Category list view (category_list)

Step 5: Create Templates

- 5.1 Create an index.html template to render the homepage with recent posts.
- 5.2 Create a post_detail.html template to display the details of a specific post.
- 5.3 Create a category_list.html template to display posts for a specific category.

Step 6: Apply Migrations

- 6.1 Run migrations to create the necessary tables for Post and Category models.

Step 7: Testing and Debugging

- 7.1 Test the URLs and views to ensure correct functionality.
- 7.2 Debug any issues with rendering, fetching, or displaying data.

Specification Sheet-1.1: Simple Blog Web Application Development Items

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-1.2: T**o-Do List Web Application Development**

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named todo_list.
- **1.2** Set up a new Django app named tasks to manage tasks.

Step 2: Create Models

- **2.1** Define a Task model with the following fields:
    - title (CharField) - The title of the task.
    - description (TextField) - A description of the task (optional).
    - created_at (DateTimeField) - Timestamp of when the task was created.
    - completed (BooleanField) - Indicates whether the task is completed or not.
    - due_date (DateField) - Optional due date for the task.

Step 3: Create Views

- **3.1** Define a view for the task list (task_list):
    - Retrieve all tasks (use Task.objects.all()).
    - Display tasks and allow toggling between completed and pending status.
- **3.2** Define a view for adding a new task (task_add):
    - Create a form for inputting the task title, description, due date, and completion status.
    - On form submission, add the new task to the database.
- **3.3** Define a view for updating a task (task_update):
    - Fetch the task by task_id.
    - Pre-populate the form with the task's current values.
    - Allow editing and updating the task details.

- **3.4** Define a view for deleting a task (task_delete):
    - Fetch the task by task_id and delete it from the database.

Step 4: Configure URLs

- **4.1** Add URL patterns for the following:
    - / → Task list view (task_list).
    - /task/add/ → Add task view (task_add).
    - /task/<int:task_id>/update/ → Update task view (task_update).
    - /task/<int:task_id>/delete/ → Delete task view (task_delete).

Step 5: Create Templates

- **5.1** Create a task_list.html template to render the list of tasks.
- **5.2** Create a task_form.html template for both adding and updating tasks.
    - This form will have fields for the task title, description, due date, and completion status.

Step 6: Apply Migrations

- **6.1** Run migrations to create the Task table in the database.

Step 7: Testing and Debugging

- **7.1** Test the views to ensure correct task creation, update, and deletion.
- **7.2** Debug any issues with forms or task management.

Specification Sheet-1.2: To-Do List Web Application Development

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-1.3: E-commerce Product Listing Web Application

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named ecommerce_site.
- **1.2** Set up a new Django app named products to manage product listings.

Step 2: Create Models

- **2.1** Define a Category model with:
    - name (CharField) - Name of the product category.
    - slug (SlugField) - A unique slug for the category.
- **2.2** Define a Product model with:
    - name (CharField) - The name of the product.
    - description (TextField) - A detailed description of the product.
    - price (DecimalField) - The price of the product.
    - image (ImageField) - An image of the product.
    - category (ForeignKey to Category) - Link to the category of the product.
    - slug (SlugField) - A unique slug for each product.
    - created_at (DateTimeField) - Timestamp of when the product was created.

Step 3: Create Views

- **3.1** Define a view for the product list (product_list):
    - Retrieve all products if no category is selected, or retrieve products for the selected category using Product.objects.filter(category=category).
    - Display the products with pagination if needed.
- **3.2** Define a view for product details (product_detail):
    - Fetch a specific product by its product_id.

- Display the product details (name, description, price, image, and category).

Step 4: Configure URLs

- **4.1** Add URL patterns for the following:
  - /products/ → Product list view (product_list).
  - /products/<int:product_id>/ → Product detail view (product_detail).
  - /products/category/<slug:category_slug>/ → Category product list view (product_list).

Step 5: Create Templates

- **5.1** Create a product_list.html template to render the list of products. Display product names, images, and prices.
- **5.2** Create a product_detail.html template to display the details of a specific product.

Step 6: Apply Migrations

- **6.1** Run migrations to create the Category and Product tables in the database.

Step 7: Testing and Debugging

- **7.1** Test the views to ensure correct product listing and detail display.
- **7.2** Debug any issues with product retrieval, template rendering, or category filtering.

Specification Sheet-1.3: E-commerce Product Listing Web Application

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-1.4: Recipe Website Development

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named recipe_site.
- **1.2** Set up a new Django app named recipes to manage recipe data.

Step 2: Create Models

- **2.1** Define a Category model for recipe categories (optional):
  - name (CharField) - Name of the category.
  - slug (SlugField) - A unique slug for the category.
- **2.2** Define a Recipe model with:
  - name (CharField) - The name of the recipe.
  - description (TextField) - A brief description of the recipe.
  - ingredients (TextField) - A list of ingredients used in the recipe.
  - instructions (TextField) - Instructions for making the recipe.
  - created_at (DateTimeField) - Timestamp of when the recipe was added.
  - category (ForeignKey to Category) - The category the recipe belongs to (optional).
  - slug (SlugField) - A unique slug for each recipe.
  - image (ImageField) - Image of the prepared recipe.

Step 3: Create Views

- **3.1** Define a view for the recipe list (recipe_list):
  - Retrieve all recipes from the database using Recipe.objects.all().
  - Display the list of recipes with pagination if necessary.
- **3.2** Define a view for recipe details (recipe_detail):
  - Fetch a specific recipe using recipe_id.

- Display the recipe details (name, description, ingredients, instructions, image).

- **3.3** Define a view for searching recipes (recipe_search):
  - Use query parameters to search recipes by name, category, or ingredients.
  - Display search results dynamically based on user input.

Step 4: Configure URLs

- **4.1** Add URL patterns for the following:
  - /recipes/ → Recipe list view (recipe_list).
  - /recipes/<int:recipe_id>/ → Recipe detail view (recipe_detail).
  - /recipes/search/ → Recipe search view (recipe_search).

Step 5: Create Templates

- **5.1** Create a recipe_list.html template to render the list of recipes.
- **5.2** Create a recipe_detail.html template to display the details of a specific recipe.
- **5.3** Create a recipe_search.html template to handle search results and display search input.

Step 6: Apply Migrations

- **6.1** Run migrations to create the Category and Recipe tables in the database.

Step 7: Testing and Debugging

- **7.1** Test the views to ensure proper recipe listing, detail rendering, and search functionality.
- **7.2** Debug any issues related to the display of recipes, filtering, and search queries.

Specification Sheet-1.4: Recipe Website Development

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-1.5: Simple Library Management System

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named library_management.
- **1.2** Set up a new Django app named library to manage books and authors.

Step 2: Create Models

- **2.1** Define an Author model with:
  - name (CharField) - Name of the author.
  - bio (TextField) - A short biography of the author.
  - birth_date (DateField) - Date of birth of the author.
- **2.2** Define a Book model with:
  - title (CharField) - The title of the book.
  - description (TextField) - A brief description of the book.
  - author (ForeignKey to Author) - The author of the book.
  - published_date (DateField) - Date the book was published.
  - isbn (CharField) - International Standard Book Number (ISBN) for the book.

Step 3: Create Views

- **3.1** Define a view for the book list (book_list):
  - Retrieve all books from the database using Book.objects.all().
  - Display a list of books with the title and author.
- **3.2** Define a view for book details (book_detail):
  - Fetch a specific book using book_id.
  - Display the book details (title, description, author, published date, ISBN).

- **3.3** Define a view for the author list (author_list):
    - Retrieve all authors from the database using Author.objects.all().
    - Display a list of authors with their names.
- **3.4** Define a view for author details (author_detail):
    - Fetch a specific author using author_id.
    - Display the author's details (name, bio, birth date, and a list of books by that author).

Step 4: Configure URLs

- **4.1** Add URL patterns for the following:
    - /books/ → Book list view (book_list).
    - /books/<int:book_id>/ → Book detail view (book_detail).
    - /authors/ → Author list view (author_list).
    - /authors/<int:author_id>/ → Author detail view (author_detail).

Step 5: Create Templates

- **5.1** Create a book_list.html template to render the list of books.
- **5.2** Create a book_detail.html template to display the details of a specific book.
- **5.3** Create an author_list.html template to render the list of authors.
- **5.4** Create an author_detail.html template to display the details of a specific author.

Step 6: Apply Migrations

- **6.1** Run migrations to create the Author and Book tables in the database.

Step 7: Testing and Debugging

- **7.1** Test the views to ensure correct book listing, book details, author listing, and author details.
- **7.2** Debug any issues with retrieving books and authors or rendering the templates.

Specification Sheet-15: Simple Library Management System

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Learning Outcome 2: Apply Django Templates

Assessment Criteria	1. Django Template Syntaxes are applied 2. Built-In Django Filters are applied
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 Template Syntaxes 2. Django Templates are configuration 3. Template Search Paths 4. Built-In Django Filters 5. Dates 6. Strings 7. Lists 8. Numbers
Activities/job/Task	1. Develop a Personal Portfolio Website 2. Develop E-commerce Product Listing Page 3. Create Blog with Categories and Tags 4. Develop Recipe Website
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 Django Templates

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 Django Templates'
2. Read the Information sheet and complete the Self Checks & Check answer sheets on "Define Tasks of the Project"	2. Read Information sheet 2: Apply Django Templates 3. Answer Self-check 2: Apply Django Templates 4. Check your answer with Answer key 2: Apply Django Templates
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Task Sheet-2.1: Develop a Personal Portfolio Website Task Sheet-2.2: Develop E-commerce Product Listing Page Task Sheet-2.3: Create Blog with Categories and Tags Task Sheet-2.4: Develop Recipe Website

Information Sheet 2: Apply Django Templates

Learning Objective:

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

2.1. Django Template Syntaxes are applied
2.2. Built-In Django Filters are applied

2.1. Django Template Syntaxes

Django templates are used to render dynamic content on web pages. They allow you to add logic like loops, conditions, and rendering of dynamic data using placeholders and variables. Here's how you apply basic template syntax:

A. Template Variables:

Django template variables are placeholders wrapped in curly braces {}. These variables represent the dynamic data to be displayed on the web page.

Example:

```
<h1>Welcome, {{ user_name }}!</h1>
```

Here, user_name is a variable that will be replaced with its actual value from the view.

B. Template Tags:

Django template tags are enclosed in {% %} and perform logic operations like loops, conditionals, etc.

Looping with {% for %}:

```
<ul>
    {% for item in items %}
        <li>{{ item.name }}</li>
    {% endfor %}
</ul>
```

This example loops through the items list and prints each item's name.

Conditional Statements with {% if %}:

```
{% if user.is_authenticated %}
    <p>Welcome back, {{ user.username }}!</p>
{% else %}
    <p>Please log in.</p>
{% endif %}
```

Here, the page will display a welcome message only if the user is authenticated.

C. Template Filters:

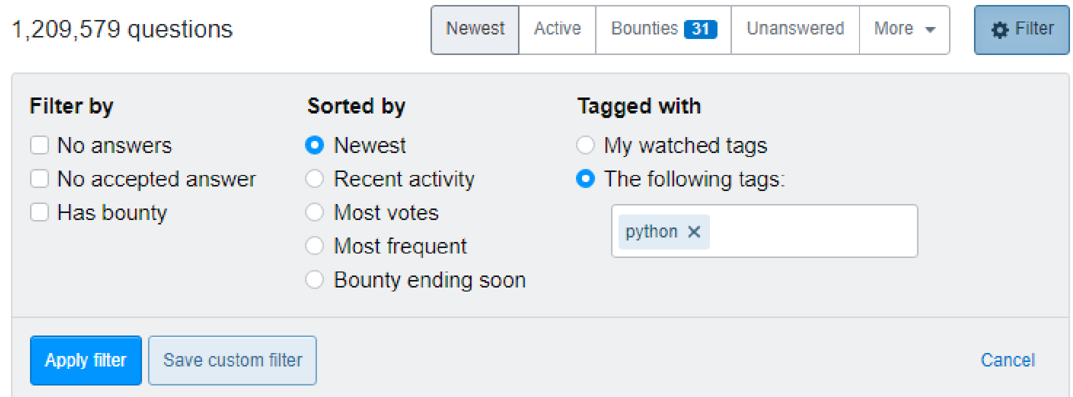
Filters allow you to modify or format template variables. Filters are applied with a pipe |.

Example using the date filter:

```
<p>Today's date is: {{ current_date|date:"F d, Y" }}</p>
```

This will format current_date to display as "February 14, 2025".

2.2. Built-In Django Filters



Django comes with several built-in filters to transform variables in templates. Some commonly used filters are:

A. date filter

Used to format date objects into a human-readable string.

```
<p>Published on: {{ blog.posted_at|date:"d M Y" }}</p>
```

If posted_at is 2025-02-14, the output will be: 14 Feb 2025.

B. lower filter

Converts a string to lowercase.

```
<p>{{ username|lower }}</p>
```

If username is "JohnDoe", the output will be: johndoe.

C. length filter

Returns the length of a list, string, or any iterable.

```
<p>The number of items is: {{ items|length }}</p>
```

If items = ["apple", "banana", "cherry"], the output will be: The number of items is: 3.

D. default filter

Provides a default value if the variable is not defined or is empty.

```
<p>{{ user_description|default:"No description available" }}</p>
```

If user_description is empty, the output will be: No description available.

E. join filter

Joins elements of a list into a string, using a separator.

```
<p>Ingredients: {{ ingredients|join:", " }}</p>
```

If ingredients = ["flour", "sugar", "eggs"], the output will be: Ingredients: flour, sugar, eggs.

Example: Django Template with Syntax and Filters

Let's look at a simple Django project that renders dynamic content using template syntax and filters.

View (views.py):

```
from django.shortcuts import render
from datetime import date

def home(request):
    context = {
        'user_name': 'Alice',
        'items': ['apple', 'banana', 'cherry'],
        'current_date': date.today(),
        'user': request.user,
    }
    return render(request, 'home.html', context)
```

Template (home.html)

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Dynamic Web Page</title>
</head>
<body>
    <h1>Welcome, {{ user_name }}!</h1>

    <h2>Your Items:</h2>
    <ul>
        {% for item in items %}
            <li>{{ item }}</li>
        {% endfor %}
    </ul>

    <p>Today's date: {{ current_date|date:"d M Y" }}</p>

    {% if user.is_authenticated %}
        <p>Welcome back, {{ user.username }}!</p>
    {% else %}
        <p>Please log in.</p>
    {% endif %}
</body>
</html>
```

Output (for an authenticated user):

```
<h1>Welcome, Alice!</h1>

<h2>Your Items:</h2>
<ul>
    <li>apple</li>
    <li>banana</li>
    <li>cherry</li>
</ul>

<p>Today's date: 14 Feb 2025</p>

<p>Welcome back, john_doe!</p>
```

Self-Check Sheet 2: Apply Django Templates

1. Which of the following is the CORRECT way to include a template variable named page_title in a Django template?

Answer:

a) <title>{{ page_title }}</title>

b) <title> {{page_title}} </title>

c) <h1>{{ page_title }}</h1>

d) echo {{ page_title }}

2. What symbol is used to apply a built-in filter to a template variable in Django?

Answer:

a) +

b) *

c) |

d) -

3. In Django templates, the date filter can be used to format a date object. What is the output of the following code snippet, assuming post.publish_date is a date object for October 21, 2024?

HTML

{{ post.publish_date | date:"Y-m-d" }}

Answer:

a) "2024-10"

b) "Monday, October 21, 2024"

c) "10-21-2024"

d) "October 21st"

4. When integrating a template in a Django view, what is the first argument passed to the render function?

Answer:

a) The context dictionary containing data for the template

b) The HTTP request object

c) The name of the template file

d) The URL pattern associated with the view

5. Which of the following is NOT a valid approach for organizing templates in a Django project?

Answer:

a) Creating a single templates directory at the project root.

b) Using subdirectories within the templates directory to group templates by functionality.

c) Storing all templates within the application directory.

d) Utilizing template inheritance with a base template and child templates.

Answer Key 2: Apply Django Templates

1. Which of the following is the CORRECT way to include a template variable named page_title in a Django template?

Answer:

a) <title>{{ page_title }}</title> **Correct Answer**

2. What symbol is used to apply a built-in filter to a template variable in Django?

Answer:

c) | **Correct Answer**

3. In Django templates, the date filter can be used to format a date object. What is the output of the following code snippet, assuming post.publish_date is a date object for October 21, 2024?

HTML

{{ post.publish_date | date:"Y-m-d" }}

Answer:

c) "10-21-2024" **Correct Answer**

4. When integrating a template in a Django view, what is the first argument passed to the render function?

Answer:

c) The name of the template file **Correct Answer**

5. Which of the following is NOT a valid approach for organizing templates in a Django project?

Answer:

c) Storing all templates within the application directory. **Correct Answer**

Task Sheet-2.1: Develop a Personal Portfolio Website

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Create a New Django Project

- Start by creating a new Django project named portfolio_website.
- Inside this project, create a Django app called portfolio to handle the content.

Step 2: Define Models in the portfolio App

- **Project Model:** For storing project information like titles, descriptions, and technologies.
- **Skill Model:** For listing your skills and proficiency levels.
- **Experience Model:** For detailing your past work experience.

Step 3: Create Templates for Each Section

- **Home Template:** Create a homepage template with a brief introduction and navigation links to other sections (About, Projects, Contact).
- **About Template:** A section where you provide your background and list your skills and work experience dynamically from the database.
- **Projects Template:** A page that displays all your projects. Use filters like truncatewords to shorten long descriptions.
- **Contact Template:** A contact form that allows users to reach out to you. You can also include links to your social media profiles.

Step 4: Set Up URL Routing

- Define URLs in urls.py to map requests to the appropriate views.
- Example:
    - / for the homepage
    - /about/ for the About page
    - /projects/ for the Projects page
    - /contact/ for the Contact page

Step 5: Create Views to Render Templates

- **Home View:** Renders the homepage template.

- **About View:** Renders the About page and passes data for skills and experience.
- **Projects View:** Renders the Projects page and passes project data.
- **Contact View:** Renders the Contact page, with optional form handling.

Step 6: Add Content and Test

- Add a few projects, skills, and experiences to the database.
- Test all the pages to make sure dynamic content is rendering correctly and the filters are applied as expected.

Specification Sheet-2.1: Develop a Personal Portfolio Website

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-2.2: Develop E-commerce Product Listing Page

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Create Django Models

1. **Product Model:**
    - Fields: Name, Description, Price, Image, Brand, Size, Color, Category.
2. **Category Model:**
    - Fields: Name, Slug (for URL).

Step 2: Design Templates

1. **Category Template:**
    - Show a list of categories (e.g., Electronics, Clothing).
    - Link each category to a product list page for that category.
2. **Product Details Template:**
    - Create a page that shows detailed information about each product (name, price, description, and images).
3. **Search Results Template:**
    - Create a page to show search results when users input keywords, along with filters.

Step 3: Implement Filters and Pagination

1. **Filter Implementation:**
    - Implement filters by price range, size, color, etc., in the product listing view.
    - Display these filters in the template.

2. **Pagination:**
    - Use Django's pagination to show a limited number of products per page and allow the user to navigate between pages of products.

Step 4: Create Views for Product List and Details

1. **Product List View:**
    - Retrieve products from the database, filter them based on user inputs (category, price, size, etc.), and pass them to the template.
2. **Product Detail View:**
    - Retrieve and display the details of a specific product when a user clicks on it.

Specification Sheet-2.2: Develop E-commerce Product Listing Page

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-2.3: Create Blog with Categories and Tags

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Create Models

1. **Blog Post Model:**
    - Fields: Title, Content, Author, Publish Date, Categories (Many-to-Many), Tags (Many-to-Many).
2. **Category Model:**
    - Fields: Name, Slug.
3. **Tag Model:**
    - Fields: Name, Slug.

Step 2: Design Templates

1. **Blog Post Template:**
    - Display individual blog posts with title, content, and metadata (author, date, tags).
2. **Category Template:**
    - List all posts within a category.
3. **Tag Template:**
    - List all posts with a specific tag.
4. **Archive Template:**
    - Allow users to view posts from a specific date range.

Step 3: Implement Filters

1. **Date Formatting:**
    - Use {{ post.publish_date|date:"F Y" }} to format the date.
2. **Create Excerpts:**
    - Use {{ post.content|truncatewords:50 }} to create an excerpt.
3. **Generate Tag Cloud:**

- Query tags from the database and display them with varying font sizes based on frequency.

4. **Display Related Posts:**
   - Use tags or categories to display related posts.

Step 4: Create Views

1. **Category View:**
   - Fetch posts from the specified category.
2. **Tag View:**
   - Fetch posts with the selected tag.
3. **Archive View:**
   - Fetch posts from a specific time range (e.g., by month, year).
4. **Post Detail View:**
   - Display details of an individual blog post.

Specification Sheet-2.3: Create Blog with Categories and Tags

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-2.4: Develop Recipe Website

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Create Models

1. **Recipe Model:**
   - Fields: Recipe name, ingredients (Many-to-Many), instructions, cuisine type, preparation time, dietary restrictions (Many-to-Many).
2. **Ingredient Model:**
   - Fields: Name of the ingredient.
3. **Cuisine Type Model:**
   - Fields: Name of the cuisine type (e.g., "Italian").
4. **Dietary Restriction Model:**
   - Fields: Name of the dietary restriction (e.g., "Vegan").

Step 2: Design Templates

1. **Recipe List Template:**
   - Display all recipes in a paginated list. Show key details like recipe name, cuisine type, and preparation time.
2. **Individual Recipe Page Template:**
   - Display detailed information for each recipe, including ingredients, preparation steps, and relevant metadata.
3. **Search Results Template:**
   - Show results based on user search criteria, with filters for ingredients, cuisine, time, and dietary restrictions.

Step 3: Implement Filters

1. **Ingredient Filtering:**
   - Use a multi-select widget or search input field that filters recipes based on ingredients.

2. **Cuisine Type Filtering:**
    - Create a dropdown or list of cuisines that users can filter by.
3. **Preparation Time Filtering:**
    - Implement a range filter or predefined time slots to filter recipes by preparation time.
4. **Dietary Restrictions Filtering:**
    - Allow users to filter based on dietary restrictions using checkboxes or multi-select options.

Step 4: Create Views

1. **Recipe List View:**
    - Fetch all recipes, apply filters, and pass the data to the recipe list template.
2. **Recipe Detail View:**
    - Fetch and display a specific recipe based on its ID.
3. **Search Results View:**
    - Handle search functionality and return results based on user-provided criteria (ingredients, cuisine, etc.).

Step 5: Implement Dynamic Content

1. **Display Recipes Based on Filters:**
    - Use Django querysets to filter and fetch recipes based on user-selected filters.
    - Implement pagination for the recipe list.

Specification Sheet-2.4: Develop Recipe Website

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Learning Outcome 3: Implement Django Application Management

Assessment Criteria	1. Django settings.py for the Real World is set 2. ALLOWED_HOSTS is defined 3. Application is allowed 4. Static web page resources are accumulated 5. Images, CSS, JavaScript are applied
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 settings.py for the Real World 2. Define ALLOWED_HOSTS 3. Static web page resources 4. Application of Images, CSS, JavaScript
Activities/job/Task	1. Simple Library Management System 2. Create a Library Management System 3. Recipe Sharing Site 4. Create a Learning Management System (LMS) 5. Create a Library Management System 6. Create a Content Management System (CMS) 7. Create a Simple Social Media Platform 8. Create a Simple Project Management Tool 9. Create an E-commerce Platform with Advanced
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 3: Implement Django Application 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 ‘Implement Django Application Management’
2. Read the Information sheet and complete the Self Checks & Check answer sheets on “Define Tasks of the Project”	2. Read Information sheet 3: Implement Django Application Management 3. Answer Self-check 3: Implement Django Application Management 4. Check your answer with Answer key 3: Implement Django Application Management
3. Read the Job/Task Sheet and Specification Sheet and perform job/Task	5. Task Sheet-3.1: Simple Library Management System Task Sheet-3.2: Create a Library Management System Task Sheet-3.3: Recipe Sharing Site Task Sheet-3.4: Create a Learning Management System (LMS) Task Sheet-3.5: Create a Library Management System Task Sheet-3.6: Create a Content Management System (CMS) Task Sheet-3.7: Create a Simple Social Media Platform Task Sheet-3.8: Create a Simple Project Management Tool Task Sheet-3.9: Create an E-commerce Platform with Advanced

Information Sheet 3: Implement Django Application Management

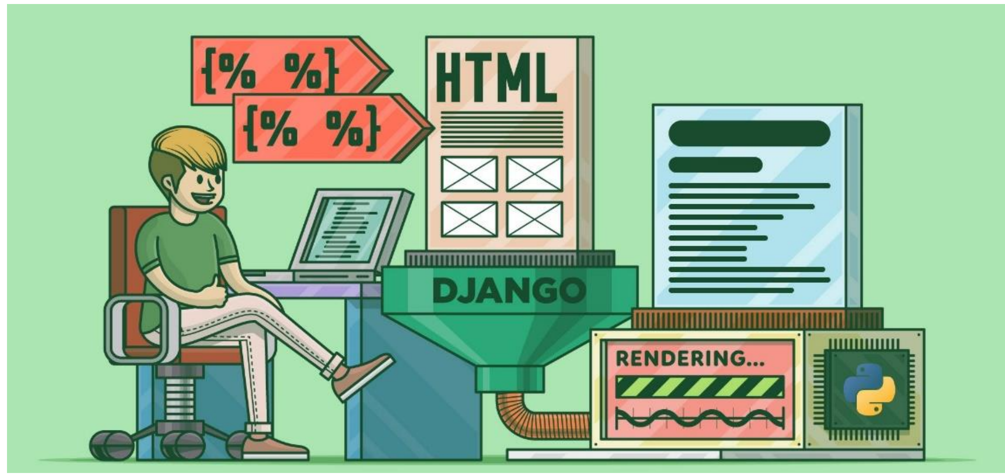
Learning Objective:

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

3.1. Django settings.py for the Real World is set
3.2. ALLOWED_HOSTS is defined
3.3. Application is allowed
3.4. Static web page resources are accumulated
3.5. Images, CSS, JavaScript are applied

3.1. Set Django settings.py for the Real World

The settings.py file in Django is where you configure your project's settings, including database configurations, middleware, installed apps, static files, and more. For a real-world deployment, several key settings need to be adjusted.



Key settings for production:

DEBUG Mode:

In production, it is crucial to set DEBUG = False to prevent the display of detailed error pages that can reveal sensitive information.

```
DEBUG = False
```

Database Settings:

You typically connect to a production database (e.g., PostgreSQL, MySQL) rather than SQLite, which is often used for development.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'my_database',
        'USER': 'my_user',
        'PASSWORD': 'my_password',
        'HOST': 'localhost',
        'PORT': '5432',
    }
}
```

Static Files Configuration:
For production, you will specify where Django should collect all static files (e.g., CSS, JavaScript, images). This is set with STATIC_ROOT:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles')
```

Security Settings:
Ensure the SECRET_KEY is kept secret and never exposed. In production, it's common to store it in an environment variable.

```
SECRET_KEY = os.getenv('DJANGO_SECRET_KEY')
```

3.2. Define ALLOWED_HOSTS

The ALLOWED_HOSTS setting defines which host/domain names your Django site can serve. It is a security feature to prevent HTTP Host header attacks.

Example of defining ALLOWED_HOSTS:

```
ALLOWED_HOSTS = ['yourdomain.com', 'www.yourdomain.com', 'subdomain.yourdomain.com']
```

- **In Development**:
  - During development, you can allow all hosts by using:
- ALLOWED_HOSTS = ['*']
  - However, in production, you should specify a list of valid domains to prevent security vulnerabilities.

3.3. Allow Application

The INSTALLED_APPS setting in Django defines which applications are enabled for your project. This setting is essential because Django needs to know which apps to include for models, views, templates, static files, etc.

Example: Adding an App to INSTALLED_APPS:
Suppose your project has an app named blog. You need to add it to the INSTALLED_APPS list:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'blog',  # Add your custom app here
]
```

This ensures Django recognizes your app and includes its models, views, and templates in the project. If the app is not listed in INSTALLED_APPS, Django will not use it.

3.4. Accumulate Static Web Page Resources

Static resources like images, CSS files, and JavaScript are stored separately from dynamic content. In Django, these resources are handled by the static files framework, and the collectstatic command is used to gather them into a central directory for production deployment.

Steps to Accumulate Static Resources:

1. **Configure Static Files**: In settings.py, set the following:
2. STATIC_URL = '/static/' # URL to serve static files

3. STATICFILES_DIRS = [os.path.join(BASE_DIR, 'static')] # Path to static files
4. STATIC_ROOT = os.path.join(BASE_DIR, 'staticfiles') # Collect static files into this directory
5. **Run the collectstatic Command**: After configuring the static file locations, run the following command to collect all static files into the STATIC_ROOT directory:
6. python manage.py collectstatic

This command copies all files from STATICFILES_DIRS and any app-level static directories into the STATIC_ROOT folder, making them ready for deployment.

3.5. Apply Images, CSS, JavaScript

Static files such as images, CSS, and JavaScript enhance the appearance and functionality of a web page. Django's static files framework allows you to include these resources in your templates using the {% static %} tag.

1. CSS (Cascading Style Sheets):

CSS is used to style the HTML elements. You link CSS files in your template as follows:

```
{% load static %}
<link rel="stylesheet" type="text/css" href="{% static 'css/styles.css' %}">
```

This will link the styles.css file located in the static/css folder.

2. JavaScript:

JavaScript is used to add interactive behavior to your web page. You can link JavaScript files in your templates:

```
<script src="{% static 'js/app.js' %}"></script>
```

This will link the app.js file located in the static/js folder.

3. Images:

You can include images in your web pages using the {% static %} tag:

```
<img src="{% static 'images/logo.png' %}" alt="Logo">
```

This will link to the image logo.png located in the static/images folder.

Example Template Using Static Files:

views.py:

```python
from django.shortcuts import render

def homepage(request):
    return render(request, 'homepage.html')
```

homepage.html (Template):

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Welcome to My Website</title>
    <link rel="stylesheet" type="text/css" href="{% static 'css/styles.css' %}">
</head>
<body>
    <header>
        <img src="{% static 'images/logo.png' %}" alt="Website Logo">
        <h1>Welcome to My Website</h1>
    </header>

    <main>
        <p>This is an example of a static web page with dynamic content.</p>
        <button onclick="showMessage()">Click Me</button>
    </main>

    <script src="{% static 'js/app.js' %}"></script>
</body>
</html>
```

Output (Rendered Web Page):

- The CSS from styles.css will style the page.
- The image logo.png will be displayed as the site's logo.
- JavaScript from app.js will enable functionality such as interactive behavior for the button.

Self-Check Sheet 3: Implement Django Application Management

1. **What is the default value of the ALLOWED_HOSTS setting in Django?**

   A. An empty list []
   B. A list containing localhost and 127.0.0.1
   C. A list containing all domain names associated with the project
   D. It depends on the Django version

2. **What is the purpose of the ALLOWED_HOSTS setting?**

   A. To configure the database connection
   B. To define static file paths
   C. To prevent HTTP Host header attacks
   D. To specify the URL prefix for static files

3. **When using environment variables like DJANGO_SETTINGS_MODULE, why might you need to define them in each shell and as a local variable for runtime environments?**

   A. Because environment variables are case-sensitive
   B. Because environment variables are not inherited by default
   C. To ensure they are available for all Django commands
   D. To prevent conflicts with other environment variables

4. **In a production environment (DEBUG=False), how do you typically serve static files?**

   A. By using Django's built-in static server
   B. By configuring your web server (e.g., Apache, Nginx) to serve them from a specific directory
   C. By including them directly in your templates
   D. By uploading them to a Content Delivery Network (CDN)

5. **What is the role of the {% static %} template tag in Django?**

   A. To define custom filters for static files
   B. To dynamically generate static file URLs based on the STATIC_URL setting
   C. To include static files from a specific app's static directory
   D. To cache static files for improved performance

Answer Key 3: Implement Django Application Management

1. **What is the default value of the ALLOWED_HOSTS setting in Django?**

   E. An empty list [] (Correct)
   F. A list containing localhost and 127.0.0.1
   G. A list containing all domain names associated with the project
   H. It depends on the Django version

2. **What is the purpose of the ALLOWED_HOSTS setting?**

   E. To configure the database connection
   F. To define static file paths
   G. To prevent HTTP Host header attacks (Correct)
   H. To specify the URL prefix for static files

3. **When using environment variables like DJANGO_SETTINGS_MODULE, why might you need to define them in each shell and as a local variable for runtime environments?**

   E. Because environment variables are case-sensitive
   F. Because environment variables are not inherited by default (Correct)
   G. To ensure they are available for all Django commands
   H. To prevent conflicts with other environment variables

4. **In a production environment (DEBUG=False), how do you typically serve static files?**

   E. By using Django's built-in static server
   F. By configuring your web server (e.g., Apache, Nginx) to serve them from a specific directory (Correct)
   G. By including them directly in your templates
   H. By uploading them to a Content Delivery Network (CDN)

5. **What is the role of the {% static %} template tag in Django?**

   E. To define custom filters for static files
   F. To dynamically generate static file URLs based on the STATIC_URL setting (Correct)
   G. To include static files from a specific app's static directory
   H. To cache static files for improved performance

Task Sheet-3.1: Simple Library Management System

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named library_management.
- **1.2** Set up a new Django app named library to manage books and authors.

Step 2: Create Models

- **2.1** Define an Author model with:
  - name (CharField) - Name of the author.
  - bio (TextField) - A short biography of the author.
  - birth_date (DateField) - Date of birth of the author.
- **2.2** Define a Book model with:
  - title (CharField) - The title of the book.
  - description (TextField) - A brief description of the book.
  - author (ForeignKey to Author) - The author of the book.
  - published_date (DateField) - Date the book was published.
  - isbn (CharField) - International Standard Book Number (ISBN) for the book.

Step 3: Create Views

- **3.1** Define a view for the book list (book_list):
  - Retrieve all books from the database using Book.objects.all().
  - Display a list of books with the title and author.
- **3.2** Define a view for book details (book_detail):
  - Fetch a specific book using book_id.
  - Display the book details (title, description, author, published date, ISBN).
- **3.3** Define a view for the author list (author_list):
  - Retrieve all authors from the database using Author.objects.all().
  - Display a list of authors with their names.
- **3.4** Define a view for author details (author_detail):

    - Fetch a specific author using author_id.
    - Display the author's details (name, bio, birth date, and a list of books by that author).

Step 4: Configure URLs

- **4.1** Add URL patterns for the following:
    - /books/ → Book list view (book_list).
    - /books/<int:book_id>/ → Book detail view (book_detail).
    - /authors/ → Author list view (author_list).
    - /authors/<int:author_id>/ → Author detail view (author_detail).

Step 5: Create Templates

- **5.1** Create a book_list.html template to render the list of books.
- **5.2** Create a book_detail.html template to display the details of a specific book.
- **5.3** Create an author_list.html template to render the list of authors.
- **5.4** Create an author_detail.html template to display the details of a specific author.

Step 6: Apply Migrations

- **6.1** Run migrations to create the Author and Book tables in the database.

Step 7: Testing and Debugging

- **7.1** Test the views to ensure correct book listing, book details, author listing, and author details.
- **7.2** Debug any issues with retrieving books and authors or rendering the templates.

Specification Sheet-3.1: Simple Library Management System

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-3.2: Create a Library Management System

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named blog_site.
- **1.2** Create the following Django apps:
  - **blog**: Manages blog posts, categories, and tags.
  - **comments**: Handles user comments on blog posts.
  - **users**: Manages user profiles, registration, and authentication.

Step 2: Set Up Models

- **2.1** Define Models for the blog app:
  - **Post** model with fields:
    - title (CharField) - Title of the blog post.
    - content (TextField) - Main content of the blog post.
    - author (ForeignKey to User) - The user who wrote the post.
    - published_date (DateTimeField) - Date and time the post was published.
    - slug (SlugField) - A unique identifier for SEO.
    - category (ForeignKey to Category) - The category of the post.
    - tags (ManyToManyField to Tag) - Tags associated with the post.
  - **Category** model with fields:
    - name (CharField) - Name of the category.
    - slug (SlugField) - Unique slug for SEO purposes.
  - **Tag** model with fields:
    - name (CharField) - Name of the tag.
    - slug (SlugField) - Unique slug for SEO purposes.

- **2.2** Define Models for the comments app:
  - **Comment** model with fields:
    - post (ForeignKey to Post) - The blog post that the comment belongs to.
    - author (ForeignKey to User) - The user who wrote the comment.
    - content (TextField) - The content of the comment.
    - created_at (DateTimeField) - Date and time when the comment was posted.
- **2.3** Define Models for the users app:
  - Use Django's built-in User model for user authentication and profile management.
  - Optionally, extend the User model to add additional fields like profile picture, bio, etc.

Step 3: Set Up Views

- **3.1 Blog Views:**
  - post_list: Displays a list of all blog posts with titles, published dates, and short excerpts.
  - post_detail: Displays the full content of a specific post along with its comments.
- **3.2 Category Views:**
  - category_list: Displays a list of all categories.
  - category_posts: Displays posts filtered by category.
- **3.3 Tag Views:**
  - tag_posts: Displays posts filtered by tags.
- **3.4 Comment Views:**
  - add_comment: Allows users to add comments to posts.
  - delete_comment: Allows users to delete their comments.
- **3.5 User Views:**
  - profile: Allows users to view and edit their profiles.
  - register: Handles user registration.
  - login: Handles user login.

- logout: Handles user logout.

Step 4: Set Up Templates

- **4.1** Create a post_list.html template to display a list of all posts.
- **4.2** Create a post_detail.html template to display the details of a specific post.
- **4.3** Create a category_list.html template to display all categories.
- **4.4** Create a category_posts.html template to display posts in a specific category.
- **4.5** Create a tag_posts.html template to display posts filtered by tags.
- **4.6** Create a comment_form.html template to allow users to add comments.
- **4.7** Create a user_profile.html template to display and edit user profiles.
- **4.8** Create a register.html template for user registration.
- **4.9** Create a login.html template for user login.

Step 5: Configure URLs

- **5.1** Define URL patterns for the following:
  - /posts/ → Post list view.
  - /posts/<slug:post_slug>/ → Post detail view.
  - /categories/ → Category list view.
  - /categories/<slug:category_slug>/ → Posts filtered by category.
  - /tags/<slug:tag_slug>/ → Posts filtered by tag.
  - /comments/add/ → Add comment view.
  - /comments/delete/<int:comment_id>/ → Delete comment view.
  - /accounts/profile/ → User profile view.
  - /accounts/register/ → User registration view.
  - /accounts/login/ → User login view.
  - /accounts/logout/ → User logout view.

Step 6: User Authentication

- **6.1** Use Django's built-in user authentication system to manage user login, registration, and logout.
- **6.2** Allow users to register, log in, and manage their profiles.

Step 7: Apply Migrations

- **7.1** Run migrations to create the necessary database tables for posts, comments, categories, tags, and user profiles.

Step 8: Testing and Debugging

- **8.1** Test all views to ensure proper functionality of post listings, post detail views, comment management, and user authentication.
- **8.2** Debug any issues related to views, templates, or authentication.

Specification Sheet-3.2: Create a Blog with Comments

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-3.3: Recipe Sharing Site

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named recipe_site.
- **1.2** Create the following Django apps:
  - **recipes**: Manages recipes, ingredients, and instructions.
  - **users**: Handles user registration, login, and profiles.
  - **reviews**: Manages user reviews and ratings for recipes.

Step 2: Set Up Models

recipes app:

- **Recipe** model with fields:
  - title (CharField) - Name of the recipe.
  - description (TextField) - Short description of the recipe.
  - author (ForeignKey to User) - The user who submitted the recipe.
  - created_at (DateTimeField) - The date the recipe was created.
  - ingredients (TextField) - Ingredients needed for the recipe.
  - instructions (TextField) - Instructions for making the recipe.
  - image (ImageField) - An optional image of the recipe.
  - slug (SlugField) - Slug for the recipe URL.
- **Category** model with fields:
  - name (CharField) - Name of the category (e.g., dessert, appetizer).
  - slug (SlugField) - Slug for category URL.

users app:

- Use Django's built-in User model for authentication.
- Optionally, extend the User model to include a profile image, bio, and other details.

reviews app:

- **Review** model with fields:
  - recipe (ForeignKey to Recipe) - The recipe being reviewed.
  - user (ForeignKey to User) - The user who wrote the review.
  - rating (IntegerField) - The rating given to the recipe (1 to 5).
  - review_text (TextField) - The actual review text.

- created_at (DateTimeField) - When the review was posted.

Step 3: Set Up Views

recipes app:

- **recipe_list**: Displays a list of all recipes with short descriptions and categories.
- **recipe_detail**: Displays detailed information about a single recipe including ingredients, instructions, and user reviews.
- **category_recipes**: Displays recipes filtered by category.

users app:

- **user_profile**: Displays user profile and allows them to update details like bio and profile image.
- **user_register**: Allows users to create an account.
- **user_login**: Allows users to log in.
- **user_logout**: Allows users to log out.

reviews app:

- **add_review**: Allows logged-in users to add a review and rating for a recipe.
- **edit_review**: Allows users to edit their own reviews.
- **delete_review**: Allows users to delete their own reviews.

Step 4: Set Up Templates

- **recipe_list.html**: Displays a list of all recipes with titles, short descriptions, and categories.
- **recipe_detail.html**: Displays detailed information for a specific recipe, including ingredients, instructions, and reviews.
- **category_recipes.html**: Displays recipes filtered by category.
- **user_profile.html**: Displays and allows editing of the user's profile.
- **register.html**: Displays the registration form for new users.
- **login.html**: Displays the login form for users.
- **add_review.html**: Displays a form for adding a review to a recipe.
- **edit_review.html**: Displays a form for editing an existing review.

Step 5: Set Up URLs

- **/recipes/** → recipe_list view.
- **/recipes/slug:recipe_slug/** → recipe_detail` view.
- **/recipes/category/slug:category_slug/** → category_recipes` view.
- **/accounts/register/** → user_register` view.
- **/accounts/login/** → user_login` view.
- **/accounts/logout/** → user_logout` view.
- **/accounts/profile/** → user_profile` view.
- **/reviews/add/slug:recipe_slug/** → add_review` view.
- **/reviews/edit/int:review_id/** → edit_review` view.
- **/reviews/delete/int:review_id/** → delete_review` view.

Step 6: User Authentication and Profile Management

- **6.1** Implement Django's built-in user authentication system for user registration, login, and logout.
- **6.2** Allow users to view and edit their profiles, including updating profile pictures and bio.

Step 7: Review System

- **7.1** Allow logged-in users to leave reviews and ratings for recipes.
- **7.2** Implement forms for adding, editing, and deleting reviews for recipes.

Step 8: Apply Migrations

- **8.1** Run migrations to create the necessary database tables for recipes, users, categories, and reviews.

Step 9: Testing and Debugging

- **9.1** Test the views and templates to ensure proper display of recipes, user authentication, and review functionality.
- **9.2** Test that reviews are properly saved, edited, and deleted by users.

Specification Sheet-3.3: Recipe Sharing Site

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-3.4: Create a Learning Management System (LMS)

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named lms_site.
- **1.2** Create the following Django apps:
  - **courses**: Manages courses, lessons, and assignments.
  - **students**: Handles student enrollment and progress tracking.
  - **instructors**: Manages instructor profiles and their courses.

Step 2: Set Up Models

courses app:

- **Course** model with fields:
  - title (CharField) - Title of the course.
  - description (TextField) - A brief description of the course.
  - instructor (ForeignKey to Instructor) - Instructor leading the course.
  - created_at (DateTimeField) - When the course was created.
  - slug (SlugField) - Slug for the course URL.
- **Lesson** model with fields:
  - course (ForeignKey to Course) - The course the lesson belongs to.
  - title (CharField) - Title of the lesson.
  - content (TextField) - Content or material for the lesson.
  - created_at (DateTimeField) - When the lesson was created.
- **Assignment** model with fields:
  - course (ForeignKey to Course) - The course the assignment belongs to.
  - title (CharField) - Title of the assignment.
  - description (TextField) - A detailed description of the assignment.

- due_date (DateTimeField) - The deadline for the assignment.
- created_at (DateTimeField) - When the assignment was created.

students app:

- **Student** model with fields:
  - user (OneToOneField to User) - Linked to the built-in User model for login.
  - courses_enrolled (ManyToManyField to Course) - Courses the student is enrolled in.
  - progress (TextField) - Tracks the student's progress in each course.

instructors app:

- **Instructor** model with fields:
  - user (OneToOneField to User) - Linked to the built-in User model.
  - bio (TextField) - A short biography of the instructor.
  - courses (ManyToManyField to Course) - Courses that the instructor teaches.

Step 3: Set Up Views

courses app:

- **course_list**: Displays a list of all courses.
- **course_detail**: Displays detailed information about a single course, including lessons and assignments.
- **lesson_detail**: Displays detailed information about a specific lesson.
- **assignment_detail**: Displays detailed information about an assignment.

students app:

- **student_dashboard**: Displays a dashboard for the student, including enrolled courses and their progress.
- **enroll_course**: Allows students to enroll in courses.
- **progress_update**: Allows students to update their progress on a course.

instructors app:

- **instructor_dashboard**: Displays the instructor's dashboard with courses they are teaching.
- **create_course**: Allows instructors to create new courses.

- **assign_grade**: Allows instructors to assign grades for assignments.

Step 4: Set Up Templates

- **course_list.html**: Displays a list of all courses with titles and descriptions.
- **course_detail.html**: Displays the course details, including lessons and assignments.
- **lesson_detail.html**: Displays detailed information for a specific lesson.
- **assignment_detail.html**: Displays detailed information for a specific assignment.
- **student_dashboard.html**: Displays a student's dashboard with their enrolled courses and progress.
- **instructor_dashboard.html**: Displays an instructor's dashboard with their courses.
- **create_course.html**: Displays the form for instructors to create new courses.

Step 5: Set Up URLs

- **/courses/** → course_list view.
- **/courses/slug:course_slug/** → course_detail` view.
- **/courses/slug:course_slug/lesson/slug:lesson_slug/** → lesson_detail` view.
- **/courses/slug:course_slug/assignment/int:assignment_id/** → assignment_detail` view.
- **/student/dashboard/** → student_dashboard` view.
- **/student/enroll/slug:course_slug/** → enroll_course` view.
- **/student/progress/slug:course_slug/** → progress_update` view.
- **/instructor/dashboard/** → instructor_dashboard` view.
- **/instructor/create-course/** → create_course` view.
- **/instructor/assign-grade/** → assign_grade` view.

Step 6: Implement Student Enrollment & Progress

- **6.1** Implement student enrollment functionality, allowing students to enroll in courses.
- **6.2** Implement a progress tracker, where students can update their progress on lessons and assignments.

Step 7: Implement Instructor Role

- **7.1** Allow instructors to create courses and manage lessons and assignments.
- **7.2** Provide instructors with a dashboard to view courses they teach and the progress of their students.

Step 8: Apply Migrations

- **8.1** Run migrations to create the necessary database tables for courses, students, instructors, lessons, and assignments.

Step 9: Testing and Debugging

- **9.1** Test the views and templates to ensure proper display of courses, lessons, assignments, student enrollment, and instructor management.
- **9.2** Test that students can update their progress, and instructors can manage courses and assignments.

Specification Sheet-3.4: Create a Learning Management System (LMS)

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-3.5: Create a Library Management System

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named lms_site.
- **1.2** Create the following Django apps:
  - **courses**: Manages courses, lessons, and assignments.
  - **students**: Handles student enrollment and progress tracking.
  - **instructors**: Manages instructor profiles and their courses.

Step 2: Set Up Models

courses app:

- **Course** model with fields:
  - title (CharField) - Title of the course.
  - description (TextField) - A brief description of the course.
  - instructor (ForeignKey to Instructor) - Instructor leading the course.
  - created_at (DateTimeField) - When the course was created.
  - slug (SlugField) - Slug for the course URL.
- **Lesson** model with fields:
  - course (ForeignKey to Course) - The course the lesson belongs to.
  - title (CharField) - Title of the lesson.
  - content (TextField) - Content or material for the lesson.
  - created_at (DateTimeField) - When the lesson was created.
- **Assignment** model with fields:
  - course (ForeignKey to Course) - The course the assignment belongs to.
  - title (CharField) - Title of the assignment.
  - description (TextField) - A detailed description of the assignment.

- due_date (DateTimeField) - The deadline for the assignment.
- created_at (DateTimeField) - When the assignment was created.

students app:

- **Student** model with fields:
  - user (OneToOneField to User) - Linked to the built-in User model for login.
  - courses_enrolled (ManyToManyField to Course) - Courses the student is enrolled in.
  - progress (TextField) - Tracks the student's progress in each course.

instructors app:

- **Instructor** model with fields:
  - user (OneToOneField to User) - Linked to the built-in User model.
  - bio (TextField) - A short biography of the instructor.
  - courses (ManyToManyField to Course) - Courses that the instructor teaches.

Step 3: Set Up Views

courses app:

- **course_list**: Displays a list of all courses.
- **course_detail**: Displays detailed information about a single course, including lessons and assignments.
- **lesson_detail**: Displays detailed information about a specific lesson.
- **assignment_detail**: Displays detailed information about an assignment.

students app:

- **student_dashboard**: Displays a dashboard for the student, including enrolled courses and their progress.
- **enroll_course**: Allows students to enroll in courses.
- **progress_update**: Allows students to update their progress on a course.

instructors app:

- **instructor_dashboard**: Displays the instructor's dashboard with courses they are teaching.
- **create_course**: Allows instructors to create new courses.

- **assign_grade**: Allows instructors to assign grades for assignments.

Step 4: Set Up Templates

- **course_list.html**: Displays a list of all courses with titles and descriptions.
- **course_detail.html**: Displays the course details, including lessons and assignments.
- **lesson_detail.html**: Displays detailed information for a specific lesson.
- **assignment_detail.html**: Displays detailed information for a specific assignment.
- **student_dashboard.html**: Displays a student's dashboard with their enrolled courses and progress.
- **instructor_dashboard.html**: Displays an instructor's dashboard with their courses.
- **create_course.html**: Displays the form for instructors to create new courses.

Step 5: Set Up URLs

- **/courses/** → course_list view.
- **/courses/slug:course_slug/** → course_detail` view.
- **/courses/slug:course_slug/lesson/slug:lesson_slug/** → lesson_detail` view.
- **/courses/slug:course_slug/assignment/int:assignment_id/** → assignment_detail` view.
- **/student/dashboard/** → student_dashboard` view.
- **/student/enroll/slug:course_slug/** → enroll_course` view.
- **/student/progress/slug:course_slug/** → progress_update` view.
- **/instructor/dashboard/** → instructor_dashboard` view.
- **/instructor/create-course/** → create_course` view.
- **/instructor/assign-grade/** → assign_grade` view.

Step 6: Implement Student Enrollment & Progress

- **6.1** Implement student enrollment functionality, allowing students to enroll in courses.
- **6.2** Implement a progress tracker, where students can update their progress on lessons and assignments.

Step 7: Implement Instructor Role

- **7.1** Allow instructors to create courses and manage lessons and assignments.
- **7.2** Provide instructors with a dashboard to view courses they teach and the progress of their students.

Step 8: Apply Migrations

- **8.1** Run migrations to create the necessary database tables for courses, students, instructors, lessons, and assignments.

Step 9: Testing and Debugging

- **9.1** Test the views and templates to ensure proper display of courses, lessons, assignments, student enrollment, and instructor management.
- **9.2** Test that students can update their progress, and instructors can manage courses and assignments.

Specification Sheet-3.5: Create a Library Management System

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-3.6: Create a Content Management System (CMS)

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named cms_project.
- **1.2** Create the following Django apps:
    - **pages**: To manage static content such as "About Us", "Contact", etc.
    - **blog**: To manage posts and articles in the CMS.
    - **media**: For uploading and managing media files (images, videos).
    - **users**: For user authentication, registration, and permissions management.
    - **search**: For implementing search functionality across the site.

Step 2: Set Up Models

pages app:

- **Page** model with fields:
    - title (CharField) - Title of the page.
    - slug (SlugField) - URL-friendly version of the title.
    - content (TextField) - Content of the page.
    - created_at (DateTimeField) - Date when the page was created.
    - updated_at (DateTimeField) - Date when the page was last updated.
    - is_active (BooleanField) - Flag to mark if the page is visible on the site.

blog app:

- **Post** model with fields:
    - title (CharField) - Title of the post.
    - slug (SlugField) - URL-friendly version of the title.
    - author (ForeignKey to User) - Author of the post.

- content (TextField) - Body content of the post.
- created_at (DateTimeField) - Date when the post was created.
- updated_at (DateTimeField) - Date when the post was last updated.
- published_date (DateTimeField) - Date when the post was published.
- tags (ManyToManyField) - Tags associated with the post.
- is_active (BooleanField) - Flag to mark if the post is published.

media app:

- **MediaFile** model with fields:
  - file (FileField) - Media file (image, video, etc.).
  - upload_date (DateTimeField) - Date when the file was uploaded.
  - file_type (CharField) - Type of media (e.g., image, video).
  - file_size (IntegerField) - Size of the file in bytes.
  - description (TextField) - Description of the media file.

users app:

- **CustomUser** model extending Django's default AbstractUser with additional fields if needed:
  - profile_picture (ImageField) - Profile picture of the user.
  - bio (TextField) - Short biography of the user.
  - is_admin (BooleanField) - Flag for admin users.
  - last_login (DateTimeField) - Last login date of the user.

search app:

- **SearchIndex** model (using django-haystack or similar) to index content (pages, blog posts):
  - content_type (CharField) - The type of content (Page or Post).
  - object_id (IntegerField) - The ID of the content being indexed.
  - text (TextField) - The searchable content from the page/post.

Step 3: Set Up Views

pages app:

- **page_list**: Displays a list of all static pages.

- **page_detail**: Displays a detailed view of a specific page.

blog app:

- **post_list**: Displays a list of all blog posts.
- **post_detail**: Displays the details of a single blog post.
- **post_create**: Allows for the creation of a new blog post (admin-only view).
- **post_update**: Allows for the editing of an existing blog post (admin-only view).
- **post_delete**: Allows for the deletion of an existing blog post (admin-only view).

media app:

- **media_list**: Displays a list of all uploaded media files.
- **media_upload**: Allows users to upload new media files (admin-only view).
- **media_detail**: Displays detailed information for a specific media file.

users app:

- **user_register**: Allows users to register for an account.
- **user_login**: Allows users to log in.
- **user_logout**: Allows users to log out.
- **user_profile**: Displays the profile information of a user.

search app:

- **search_results**: Displays the search results for pages, posts, and other indexed content.

Step 4: Set Up Templates

- **base.html**: Base template that includes common structure for header, footer, and sidebar.
- **page_list.html**: Template for listing all static pages.
- **page_detail.html**: Template for displaying a specific static page.
- **post_list.html**: Template for listing all blog posts.
- **post_detail.html**: Template for displaying a specific blog post.
- **media_list.html**: Template for displaying all media files.
- **user_profile.html**: Template for displaying the user's profile.

Step 5: Set Up URLs

- **/pages/** → page_list view (List all pages).
- **/pages/slug:slug/** → page_detail` view (Display detailed page info).
- **/blog/** → post_list view (List all blog posts).
- **/blog/slug:slug/** → post_detail` view (Display detailed post info).
- **/media/** → media_list view (List all uploaded media).
- **/media/upload/** → media_upload` view (Upload new media).
- **/users/register/** → user_register` view (User registration).
- **/users/login/** → user_login` view (User login).
- **/users/logout/** → user_logout` view (User logout).
- **/users/profile/** → user_profile` view (User profile).
- **/search/** → search_results` view (Search content on the site).

Step 6: Set Up Search Functionality

- **6.1** Implement the search functionality using django-haystack or similar.
- **6.2** Index content from the pages and blog apps for search purposes.
- **6.3** Implement a search form in the templates to allow users to search content.

Step 7: Set Up User Permissions

- **7.1** Use Django's built-in permissions system to manage user roles (admin, regular user).
- **7.2** Assign appropriate permissions for creating, editing, and deleting content (pages, blog posts, media).

Step 8: Testing and Debugging

- **8.1** Test all views to ensure pages, posts, media, and user management are functioning correctly.
- **8.2** Test the search functionality and ensure that it returns relevant results.
- **8.3** Test user registration, login, and profile management.
- **8.4** Ensure that user permissions are working as expected (e.g., only admins can create and delete posts).

Specification Sheet-3.6: Create a Content Management System (CMS)

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-3.7: Create a Simple Social Media Platform

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named social_media_platform.
- **1.2** Create the following Django apps:
  - **posts**: To manage posts, including text and media.
  - **users**: For user authentication, profiles, and relationships (following/followers).
  - **notifications**: For managing notifications sent to users.
  - **feeds**: For managing user feeds and displaying posts from followed users.

Step 2: Set Up Models

posts app:

- **Post** model with fields:
  - author (ForeignKey to User) - The user who created the post.
  - content (TextField) - Text content of the post.
  - image (ImageField, optional) - An optional image attached to the post.
  - created_at (DateTimeField) - The time the post was created.
  - updated_at (DateTimeField) - The last time the post was updated.
  - likes (ManyToManyField to User) - Users who liked the post.
  - comments (ManyToManyField to Comment) - Comments on the post.

users app:

- **User** model (extend Django's built-in AbstractUser):
  - username (CharField) - User's unique username.
  - email (EmailField) - User's email address.
  - profile_picture (ImageField) - Optional profile picture.

    - bio (TextField) - A short biography about the user.
    - following (ManyToManyField to User) - Users that this user is following.
    - followers (ManyToManyField to User) - Users following this user.
    - is_verified (BooleanField) - Flag for whether the user's account is verified.

notifications app:

- **Notification** model with fields:
    - user (ForeignKey to User) - The user who will receive the notification.
    - message (TextField) - The content of the notification.
    - created_at (DateTimeField) - The time the notification was created.
    - read (BooleanField) - Whether the user has read the notification.

feeds app:

- **FeedItem** model with fields:
    - user (ForeignKey to User) - The user for whom the feed item is being shown.
    - post (ForeignKey to Post) - The post that will be shown in the user's feed.
    - created_at (DateTimeField) - The time the feed item was created.
    - updated_at (DateTimeField) - The last time the feed item was updated.

Step 3: Set Up Views

posts app:

- **post_list**: Displays all posts from users that the current user follows.
- **post_detail**: Displays a detailed view of a specific post.
- **post_create**: Allows authenticated users to create a new post.
- **post_like**: Allows users to like a post.
- **post_comment**: Allows users to comment on a post.

users app:

- **user_register**: Allows new users to register.
- **user_login**: Allows users to log in to their accounts.

- **user_logout**: Allows users to log out.
- **user_profile**: Displays a user’s profile, including bio, posts, followers, and following.
- **user_follow**: Allows users to follow other users.
- **user_unfollow**: Allows users to unfollow other users.

notifications app:

- **notification_list**: Displays all unread notifications for the logged-in user.
- **notification_mark_as_read**: Marks a specific notification as read.

feeds app:

- **user_feed**: Displays a feed of posts from the users that the logged-in user follows.
- **global_feed**: Displays posts from all users, sorted by recency.

Step 4: Set Up Templates

- **base.html**: The base template that includes the header, footer, and common UI elements like the navigation bar.
- **post_list.html**: Template for displaying a list of posts in the feed.
- **post_detail.html**: Template for displaying the details of a single post.
- **post_create.html**: Template for creating a new post.
- **user_profile.html**: Template for displaying the user's profile, including their posts, followers, and following.
- **notification_list.html**: Template for displaying a list of unread notifications.
- **user_feed.html**: Template for displaying the user's feed with posts from followed users.
- **global_feed.html**: Template for displaying posts from all users.

Step 5: Set Up URLs

- **/posts/** → post_list view (List all posts from followed users).
- **/posts/int:post_id/** → post_detail` view (Detailed post view).
- **/posts/create/** → post_create` view (Create a new post).
- **/posts/int:post_id/like/** → post_like` view (Like a post).
- **/posts/int:post_id/comment/** → post_comment` view (Comment on a post).

- **/users/register/** → user_register` view (User registration).
- **/users/login/** → user_login` view (User login).
- **/users/logout/** → user_logout` view (User logout).
- **/users/int:user_id/profile/** → user_profile` view (User profile).
- **/users/int:user_id/follow/** → user_follow` view (Follow a user).
- **/users/int:user_id/unfollow/** → user_unfollow` view (Unfollow a user).
- **/notifications/** → notification_list` view (List unread notifications).
- **/notifications/int:notification_id/read/** → notification_mark_as_read` view (Mark notification as read).
- **/feed/** → user_feed` view (User's personalized feed).
- **/global_feed/** → global_feed` view (Global feed with posts from all users).

Step 6: Set Up User Relationships

- **6.1** Create functionality for users to follow and unfollow other users.
- **6.2** Display posts from followed users in the user's feed.
- **6.3** Send notifications when a user follows someone or likes/comments on a post.

Step 7: Implement Notifications

- **7.1** Create a notification system to alert users about interactions (likes, comments, follows).
- **7.2** Display notifications in the user interface.
- **7.3** Allow users to mark notifications as read/unread.

Step 8: Set Up Authentication and Permissions

- **8.1** Use Django's built-in authentication system for user login, registration, and logout.
- **8.2** Use Django's permission system to ensure that only authenticated users can post, follow, and comment.

Step 9: Testing and Debugging

- **9.1** Test all views and ensure correct functionality (posting, following, commenting, etc.).

- **9.2** Test the notifications and ensure they are sent correctly for follows, likes, and comments.
- **9.3** Ensure that the user feed displays posts from followed users.
- **9.4** Ensure all URLs are functioning as expected, and fix any bugs in routing or views.

Specification Sheet-3.7: Create a Simple Social Media Platform

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-3.8: Create a Simple Project Management Tool

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named project_management_tool.
- **1.2** Create the following Django apps:
  - **projects**: To manage projects and related information.
  - **tasks**: To manage tasks and subtasks within projects.
  - **teams**: For managing team members and their roles/permissions.
  - **milestones**: To track important milestones and deadlines for each project.

Step 2: Set Up Models

projects app:

- **Project** model with fields:
  - name (CharField) - The name of the project.
  - description (TextField) - A description of the project.
  - start_date (DateField) - The start date of the project.
  - end_date (DateField) - The estimated end date of the project.
  - created_by (ForeignKey to User) - The user who created the project.
  - status (CharField) - The current status of the project (e.g., "In Progress", "Completed").

tasks app:

- **Task** model with fields:
  - title (CharField) - The title of the task.
  - description (TextField) - A detailed description of the task.
  - project (ForeignKey to Project) - The project the task belongs to.
  - assigned_to (ForeignKey to User) - The user the task is assigned to.

- due_date (DateField) - The due date for the task.
- status (CharField) - The current status of the task (e.g., "Pending", "In Progress", "Completed").
- priority (CharField) - The priority level of the task (e.g., "High", "Medium", "Low").

- **Subtask** model with fields:
  - task (ForeignKey to Task) - The task the subtask belongs to.
  - title (CharField) - The title of the subtask.
  - assigned_to (ForeignKey to User) - The user assigned to the subtask.
  - status (CharField) - The current status of the subtask (e.g., "Pending", "Completed").

teams app:

- **Team** model with fields:
  - name (CharField) - The name of the team.
  - members (ManyToManyField to User) - The members of the team.
  - role (CharField) - The role of each user in the team (e.g., "Manager", "Developer", "Designer").
- **TeamRole** model to define roles with permissions:
  - role_name (CharField) - The name of the role (e.g., "Manager", "Team Lead").
  - permissions (TextField) - A description or list of permissions associated with the role.

milestones app:

- **Milestone** model with fields:
  - title (CharField) - The title of the milestone.
  - description (TextField) - A description of the milestone.
  - project (ForeignKey to Project) - The project the milestone is associated with.
  - due_date (DateField) - The due date for the milestone.
  - status (CharField) - The current status of the milestone (e.g., "Pending", "Achieved").

Step 3: Set Up Views

projects app:

- **project_list**: Displays a list of all projects with basic information.
- **project_detail**: Displays details of a specific project, including tasks, milestones, and team members.
- **project_create**: Allows users to create a new project.
- **project_edit**: Allows users to edit an existing project.
- **project_delete**: Allows users to delete a project.

tasks app:

- **task_list**: Displays all tasks within a project.
- **task_create**: Allows users to create a new task within a project.
- **task_edit**: Allows users to edit a task.
- **task_delete**: Allows users to delete a task.
- **task_assign**: Allows managers to assign tasks to team members.
- **task_progress**: Allows users to update the status of a task.

teams app:

- **team_list**: Displays a list of all teams within the project.
- **team_create**: Allows users to create a new team.
- **team_add_member**: Allows users to add a member to a team.
- **team_remove_member**: Allows users to remove a member from a team.
- **assign_role**: Assigns a role to a team member (e.g., Manager, Developer).

milestones app:

- **milestone_list**: Displays a list of all milestones within a project.
- **milestone_create**: Allows users to create a new milestone for a project.
- **milestone_edit**: Allows users to edit a milestone.
- **milestone_delete**: Allows users to delete a milestone.
- **milestone_progress**: Allows users to mark a milestone as achieved.

Step 4: Set Up Templates

- **base.html**: The base template that includes the header, footer, and common UI elements like the navigation bar.
- **project_list.html**: Template for displaying a list of all projects.
- **project_detail.html**: Template for displaying the details of a specific project.
- **task_list.html**: Template for displaying tasks within a project.
- **task_detail.html**: Template for displaying details of a specific task.
- **team_list.html**: Template for displaying team members within a project.
- **milestone_list.html**: Template for displaying milestones of a project.

Step 5: Set Up URLs

- **/projects/** → project_list view (List all projects).
- **/projects/int:project_id/** → project_detail` view (View a specific project).
- **/projects/create/** → project_create` view (Create a new project).
- **/projects/int:project_id/edit/** → project_edit` view (Edit a project).
- **/projects/int:project_id/delete/** → project_delete` view (Delete a project).
- **/tasks/** → task_list` view (List all tasks).
- **/tasks/create/** → task_create` view (Create a new task).
- **/tasks/int:task_id/edit/** → task_edit` view (Edit a task).
- **/tasks/int:task_id/delete/** → task_delete` view (Delete a task).
- **/teams/** → team_list` view (List all teams).
- **/teams/create/** → team_create` view (Create a new team).
- **/teams/int:team_id/add_member/** → team_add_member` view (Add a member to a team).
- **/teams/int:team_id/remove_member/** → team_remove_member` view (Remove a member from a team).
- **/milestones/** → milestone_list` view (List all milestones).
- **/milestones/create/** → milestone_create` view (Create a new milestone).
- **/milestones/int:milestone_id/edit/** → milestone_edit` view (Edit a milestone).
- **/milestones/int:milestone_id/delete/** → milestone_delete` view (Delete a milestone).

Step 6: Implement User Roles and Permissions

- **6.1** Assign roles (e.g., Manager, Developer) to team members using the TeamRole model.
- **6.2** Ensure that only users with the correct permissions (e.g., Managers) can create, edit, or delete projects, tasks, and milestones.

Step 7: Testing and Debugging

- **7.1** Test the creation, editing, and deletion of projects, tasks, milestones, and teams.
- **7.2** Test the task assignment functionality and ensure that team members can update their tasks.
- **7.3** Test the roles and permissions to ensure that only authorized users can perform certain actions.
- **7.4** Ensure all views and templates render correctly and function as expected.

Specification Sheet-3.8: Create a Simple Project Management Tool

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Task Sheet-3.9: Create an E-commerce Platform with Advanced

UoC Cover

OU-ICT-WADP-02- L4-V1: Develop Dynamic Web Pages

Working Procedure / Steps

Step 1: Set Up the Project

- **1.1** Create a new Django project named ecommerce_platform.
- **1.2** Create the following Django apps:
  - **products**: To manage product-related information.
  - **categories**: To manage product categories.
  - **orders**: To manage customer orders.
  - **payments**: To integrate payment gateways.
  - **shipping**: To handle shipping information.
  - **reviews**: To handle customer reviews for products.
  - **promotions**: To manage promotions, discounts, and offers.

Step 2: Set Up Models

products app:

- **Product** model with fields:
  - name (CharField) - The name of the product.
  - description (TextField) - A detailed description of the product.
  - price (DecimalField) - The price of the product.
  - category (ForeignKey to Category) - The category the product belongs to.
  - variations (ManyToManyField to Variation) - The product variations (e.g., size, color).
  - available_stock (IntegerField) - The quantity of the product available in stock.
- **Variation** model with fields:
  - product (ForeignKey to Product) - The product this variation belongs to.

- name (CharField) - The name of the variation (e.g., "Color", "Size").
- value (CharField) - The value of the variation (e.g., "Red", "Large").

categories app:

- **Category** model with fields:
  - name (CharField) - The name of the category.
  - description (TextField) - A description of the category.
  - parent_category (ForeignKey to Category, nullable) - Parent category if it's a subcategory.

orders app:

- **Order** model with fields:
  - customer (ForeignKey to User) - The customer who placed the order.
  - order_date (DateTimeField) - The date and time the order was placed.
  - status (CharField) - The status of the order (e.g., "Pending", "Shipped").
  - total_amount (DecimalField) - The total amount of the order.
- **OrderItem** model with fields:
  - order (ForeignKey to Order) - The order the item belongs to.
  - product (ForeignKey to Product) - The product in the order.
  - quantity (IntegerField) - The quantity of the product ordered.
  - price (DecimalField) - The price of the product at the time of the order.

payments app:

- **Payment** model with fields:
  - order (ForeignKey to Order) - The order the payment corresponds to.
  - payment_date (DateTimeField) - The date and time the payment was made.
  - amount (DecimalField) - The total amount paid.
  - payment_status (CharField) - The payment status (e.g., "Pending", "Completed").
  - payment_method (CharField) - The method of payment (e.g., "Credit Card", "PayPal").
  - transaction_id (CharField) - The payment transaction ID.

shipping app:

- **Shipping** model with fields:
  - order (ForeignKey to Order) - The order being shipped.
  - shipping_address (CharField) - The shipping address.
  - shipping_status (CharField) - The current status of the shipment (e.g., "Pending", "Shipped", "Delivered").
  - tracking_number (CharField) - The tracking number for the shipment.

reviews app:

- **Review** model with fields:
  - product (ForeignKey to Product) - The product being reviewed.
  - customer (ForeignKey to User) - The customer who left the review.
  - rating (IntegerField) - The rating given to the product (e.g., 1-5 stars).
  - comment (TextField) - The review comment.
  - created_at (DateTimeField) - The date and time the review was created.

promotions app:

- **Promotion** model with fields:
  - code (CharField) - The promotion code.
  - discount_percentage (DecimalField) - The percentage discount provided by the promotion.
  - valid_from (DateTimeField) - The start date of the promotion.
  - valid_until (DateTimeField) - The end date of the promotion.
  - active (BooleanField) - Whether the promotion is currently active.

Step 3: Set Up Views

products app:

- **product_list**: Displays all products in a list format.
- **product_detail**: Displays details of a specific product.
- **product_create**: Allows users to create a new product (admin only).
- **product_edit**: Allows users to edit a product's details (admin only).

categories app:

- **category_list**: Displays all product categories.
- **category_detail**: Displays products within a specific category.

orders app:

- **order_list**: Displays a list of all orders (for the admin).
- **order_detail**: Displays the details of a specific order.
- **order_create**: Allows customers to create an order (via checkout process).

payments app:

- **payment_process**: Handles payment gateway integration (e.g., Stripe, PayPal).
- **payment_success**: Displays a success message after a successful payment.
- **payment_failure**: Displays an error message in case of payment failure.

shipping app:

- **shipping_status**: Displays the shipping status of an order.
- **shipping_create**: Allows admin to create shipping information for an order.

reviews app:

- **product_reviews**: Displays all reviews for a specific product.
- **review_create**: Allows customers to leave a review for a product.

promotions app:

- **promotion_list**: Displays all active promotions.
- **promotion_apply**: Allows users to apply a promotion code during checkout.

Step 4: Set Up Templates

- **base.html**: The base template containing the header, footer, and common elements like the navigation bar.
- **product_list.html**: Template for displaying a list of all products.
- **product_detail.html**: Template for displaying the details of a specific product.
- **category_list.html**: Template for displaying a list of product categories.
- **category_detail.html**: Template for displaying products within a category.
- **order_list.html**: Template for displaying a list of orders (admin only).
- **order_detail.html**: Template for displaying the details of an order.

- **checkout.html**: Template for the checkout process, including payment details and order summary.
- **shipping_status.html**: Template for displaying the shipping status of an order.

Step 5: Set Up URLs

- **/products/** → product_list view (List all products).
- **/products/int:product_id/** → product_detail` view (View a specific product).
- **/products/create/** → product_create` view (Create a new product).
- **/categories/** → category_list view (List all categories).
- **/categories/int:category_id/** → category_detail` view (View products in a category).
- **/orders/** → order_list` view (List all orders).
- **/orders/int:order_id/** → order_detail` view (View a specific order).
- **/orders/create/** → order_create` view (Create an order).
- **/payments/process/** → payment_process` view (Process payment).
- **/payments/success/** → payment_success` view (Display success after payment).
- **/payments/failure/** → payment_failure` view (Display failure message).
- **/shipping/status/** → shipping_status` view (View shipping status).
- **/reviews/** → product_reviews` view (List reviews for a product).
- **/reviews/create/** → review_create` view (Create a new review).
- **/promotions/** → promotion_list` view (List all promotions).
- **/promotions/apply/** → promotion_apply` view (Apply a promotion code).

Step 6: Testing and Debugging

- **6.1** Test the product creation, editing, and deletion functionality.
- **6.2** Test the checkout and payment processing workflows.
- **6.3** Test the product review system and promotion application.
- **6.4** Test the shipping management system and order status tracking.

Specification Sheet-3.9: Create a Simple Project Management Tool

Necessary Personal Protective Equipment (PPE)

Sl. No	Name of PPE	Unit	Quantity
1	Ergonomic Desk Chair	Piece	1

Necessary Tools

Sl. No	Name of Tools	Specification	Unit	Quantity
1	Laptop	16GB RAM, i7 Processor	Piece	1
2	Text Editor (e.g., VS Code)	Latest Version	Piece	1
3	Git	Version Control Tool	Piece	1
4	Terminal	CLI Tool for Running Django	Piece	1

Necessary Equipment

Sl. No	Name of Equipment	Specification	Unit	Quantity
1	Web Browser (e.g., Chrome)	Latest Version	Piece	1
2	Internet Connection	High-Speed Broadband	Connection	1

Necessary Materials

Sl. No	Name of Materials	Specification	Unit	Quantity
1	Python	Version 3.8+	Installation	1
2	Django	Version 3.2+	Installation	1
3	SQLite (or other DB)	Latest Version	Installation	1
4	HTML/CSS	Standard Web Templates	Files	3

Reference

1. **Flask Web Development: Developing Python Web Applications**
   Author: Miguel Grinberg
   Year: 2018 (2nd Edition)
2. **Django for Beginners: Build Awesome Websites with Python**
   Author: William S. Vincent
   Year: 2021 (2nd Edition)
3. **Two Scoops of Django: Best Practices for the Django Web Framework**
   Authors: Daniel J. Greenfeld and Audrey Roy Greenfeld
   Year: 2020 (3rd Edition)

Review of Competency

Below is yourself assessment rating for module "Develop Dynamic Web Pages"

Assessment of performance Criteria	Yes	No
Common Url Patterns are applied		
Url Parameters, Extra Options, and Query Strings are used		
Url Naming and Namespaces are implemented		
Url Method Requests are implemented		
View Method Requests are implemented		
View Method Responses are implemented		
Django Template Syntaxes are applied		
Built-In Django Filters are applied		
Django settings.py for the Real World is set		
ALLOWED_HOSTS is defined		
Application is allowed		
Static web page resources are accumulated		
Images, CSS, JavaScript are applied		

I now feel ready to undertake my formal competency assessment.

Signed:

Date:

Development of CBLM

The Competency based Learning Material (CBLM) of ‘Developing Dynamic Web Pages’ (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