USN | | | | | | | | | | |

Internal Assessment Test 2 – July 2024

Sub	**Full Stack Development**			Sub code	**21CS62**	Branch	**CSE**
Date	**08.07.2024**	Duration	**90 mins**	Max Marks	**50**	Sem /Sec	**VI Sem (B, C)**	**OBE**

Answer any FIVE FULL Questions		MARKS	CO	RBT
1(a)	Explain the various features of Django Administration website with the example? (min 5)	[5]	CO2	L2
1(b)	Explain any five customization commands with a suitable example for the model given below. Student (Stud_usn, Stud_name, Sem, marks). Incorporate customization commands in the program?	[5]	CO2	L2,L3
2	You have a Student model with fields stud_usn, stud_name, sem, and marks. Create a model form for the Student model and add custom validation to ensure that the sem field is between 1 and 8. Explain how to implement the custom validation in the form and provide code snippets for the model, form, and validation method.	[10]	CO2	L2,L3
3	Explain how to include URL configurations from two different Django apps, students and courses, in the main project's urls.py file. Provide code snippets for the urls.py files in both the apps and the main project, and describe the benefits of organizing URLs in this manner.	[10]	CO2	L2
4	Explain what are the principal advantage of Generic view in Django. Explain the Class based views and their types with example.	10	CO3	L2
5	How would you customize the ModelForm to include additional fields that are not part of the Customer model?	10	CO2	L3
6(a)	Write a python function for validating the mobile number with validation condition as 10 digits, ‘should not start with 0, + can be at the beginning.	5	CO2	L2
6(b)	Describe how to use Django's generic ListView to display a list of items from a model named Post with fields title, content, and published_date. Provide the necessary code snippets for the model, view, URL configuration, and template.	5	CO2	L2

CI CCI HOD

Internal Assessment Test 2

Solution

Sub:	**Full Stack Development**					Sub Code:	**21CS62**	Branch :	CSE
Date:	8/7/2024	Duration:	90 mins	Max Marks:	50	Sem / Sec:	VI- B & C		

Question number	Question and solution
1	**a)Explain the various features of the Django Administration website with the example? (min 5)** **Solution**: Here are five key features of the Django admin interface, illustrated with examples: 1. **User Authentication and Permissions** **Feature**: Django admin provides a built-in user authentication system. This allows administrators to create, manage, and assign permissions to users. **Example**: ● Creating Users: Admins can create new user accounts and set their passwords. ● Assigning Permissions: Admins can assign different permissions to users, such as viewing, adding, changing, or deleting data. ● Groups: Users can be assigned to groups, and permissions can be assigned to groups, making it easier to manage permissions for multiple users. **2. Model Management** **Feature**: The admin site automatically generates a management interface for any model registered with it. This includes creating, updating, and deleting records. **3. Customizing Admin Interface** **Feature**: Django allows extensive customization of the admin interface to tailor it to

specific needs.

Example:

- **Custom Admin Class**: Customizing the admin interface for a model by using a `ModelAdmin` class.

```
# admin.py

from django.contrib import admin

from .models import Author



class AuthorAdmin(admin.ModelAdmin):

    list_display = ('name', 'email')

    search_fields = ('name',)

admin.site.register(Author, AuthorAdmin)
```

4. Filter and Search Capabilities

Feature: The admin interface provides filtering and searching capabilities to make it easier to find specific records.

Example:

- **List Filter**: Adding filters to the list view to filter records based on specific fields.
- **Search Fields**: Adding search functionality to quickly locate records.

5. Custom Actions

Feature: Django admin allows the creation of custom actions that can be performed on multiple records at once. This can be used to perform batch updates or other operations.

Example:

- **Custom Action**: Creating a custom action to mark multiple books as published.

b) Explain any five customization commands with a suitable example for the model given below.

Student (Stud_usn, Stud_name, Sem, marks). Incorporate customization commands in the program?

Solution:

Five customization commands for the Django admin interface using the `Student` model **example**.

1. Displaying Fields in List View

Command: `list_display`

2. Adding Search Functionality

Command: `search_fields`

3. Adding Filters

Command: `list_filter`

4. Editable Fields in List View

Command: `list_editable`

5. Ordering Records

Command: `ordering`

Example:

```
class StudentAdmin(admin.ModelAdmin):

    list_display = ('Stud_usn', 'Stud_name', 'Sem', 'marks')
    search_fields = ('Stud_usn', 'Stud_name')

    list_filter = ('Sem',)

    list_editable = ('marks',)

    ordering = ('Sem', 'Stud_name')

admin.site.register(Student, StudentAdmin)
```

2

You have a Student model with fields stud_usn, stud_name, sem, and marks. Create a model form for the Student model and add custom validation to ensure that the sem field is between 1 and 8. Explain how to implement the custom validation in the form and provide code snippets for the model, form, and validation method.

Solution:

To create a model form for the `Student` model and add custom validation to ensure that the `sem` field is between 1 and 8, follow these steps:

1. Define the `Student` Model

```
# models.py
from django.db import models

class Student(models.Model):
    stud_usn = models.CharField(max_length=10)
    stud_name = models.CharField(max_length=100)
    sem = models.IntegerField()
    marks = models.IntegerField()

    def __str__(self):
        return self.stud_name
```

2. Create a Custom Validator

# forms.py

```
from django.core.exceptions import ValidationError
from django import forms
from .models import Student
from .validators import validate_sem

def validate_sem(value):
    if value < 1 or value > 8:
        raise ValidationError('Semester must be between 1 and 8.')

class StudentForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = ['stud_usn', 'stud_name', 'sem', 'marks']

    sem = forms.IntegerField(validators=[validate_sem])
```

3. Use the Form in a View

```
# views.py
from django.shortcuts import render, redirect
from .forms import StudentForm

def create_student(request):
    if request.method == 'POST':
        form = StudentForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('success')
    else:
        form = StudentForm()
    return render(request, 'create_student.html', {'form': form})

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

4. Create Templates for the Form and Success Page

```
<!-- create_student.html -->
<!DOCTYPE html>
<html>
<head>
    <title>Create Student</title>
</head>
<body>
    <h1>Create Student</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
        <button type="submit">Submit</button>
    </form>
</body>
</html>
```

	```
	<!-- success.html -->
	<!DOCTYPE html>
	<html>
	<head>
	    <title>Success</title>
	</head>
	<body>
	    <h1>Student Created Successfully!</h1>
	</body>
	</html>
	```

3	**Explain how to include URL configurations from two different Django apps, students and courses, in the main project's urls.py file. Provide code snippets for the urls.py files in both the apps and the main project, and describe the benefits of organizing URLs in this manner.**
	
	Solution:
	
	To include URL configurations from two different Django apps (`students` and `courses`) in the main project's `urls.py` file, follow these steps:
	
	1. Define URLs in the `students` App
	# students/urls.py
	from django.urls import path
	from . import views
	
	urlpatterns = [
	    path('', views.index, name='student_index'),
	    path('<int:id>/', views.detail, name='student_detail'),
	]
	
	2. Define URLs in the `courses` App
	# courses/urls.py
	from django.urls import path
	from . import views
	
	urlpatterns = [

	`    path('', views.index, name='course_index'),`
	`    path('<int:id>/', views.detail, name='course_detail'),`
	`]`
	3. Include App URLs in the Main Project's `urls.py`
	`# main_project/urls.py`
	`from django.contrib import admin`
	`from django.urls import path, include`
	
	`urlpatterns = [`
	`    path('admin/', admin.site.urls),`
	`    path('students/', include('students.urls')),`
	`    path('courses/', include('courses.urls')),`
	`]`
	Benefits of Organizing URLs in this Manner
	
	1. **Modularity**: Each app manages its own URLs, making the project modular and easier to maintain.
	2. **Scalability**: Adding new apps or modifying existing ones is simpler and more isolated.
	3. **Clarity**: The main `urls.py` file remains clean and clear, showing a high-level overview of the project's URL structure.
	4. **Reusability**: URL patterns can be reused across different projects or moved easily between them.

4	**Explain what are the principal advantage of Generic view in Django. Explain the Class based views and their types with example.** **Solution**: **Principal Advantages of Generic Views in Django** 1. **Speed**: Rapid development due to pre-built views for common tasks. 2. **Code Reduction**: Less code duplication and more concise implementations. 3. **Consistency**: Follows Django's design patterns, ensuring consistent code structure. 4. **Flexibility**: Customizable through mixins and class attributes. 5. **Built-in Features**: Includes built-in functionalities like pagination, form

handling, and more.

Class Based views:

- Generic Views or **Class-Based Generic views**. in web development, particularly in the Django framework, are a set of pre-built views that provide common functionalities.
- Using Class-Based views, we can easily handle the GET, POST requests for a view.
- They help developers quickly implement common patterns in web applications, such as displaying a list of objects, showing details of a single object, creating new objects, updating existing objects, and deleting objects.
- Basically we can handle CRUD operation.

Class Based Views in Django:

- **CreateView** – create or add new entries in a table in the database.
- **Retrieve Views** – read, retrieve, search, or view existing entries as a list(**ListView**) or retrieve a particular entry in detail (**DetailView**) .
- **UpdateView** – update or edit existing entries in a table in the database
- **DeleteView** – delete, deactivate, or remove existing entries in a table in the database
- **FormView** – render a form to template and handle data entered by user

5

How would you customize the ModelForm to include additional fields that are not part of the Customer model?

Solution:

Assume you have a `Customer` model with fields `name`, `email`, and `phone`. You want to add a field `message` to a form that allows customers to send a message along with their information.

# forms.py

from django import forms

```
from .models import Customer


class CustomerForm(forms.ModelForm):
    # Define additional fields not in the Customer model
    message = forms.CharField(label='Message', widget=forms.Textarea)


    class Meta:
        model = Customer
        fields = ['name', 'email', 'phone']  # Include fields from the Customer model
# views.py
from django.shortcuts import render, redirect
from .forms import CustomerForm
def customer_form(request):
    if request.method == 'POST':
        form = CustomerForm(request.POST)
        if form.is_valid():
            # Process the form data (save to database, send email, etc.)
            # For demonstration, let's just print the form data
            print(form.cleaned_data)  # This dictionary contains all form data
            return redirect('form_success')  # Redirect to a success page
    else:
        form = CustomerForm()
    return render(request, 'customer_form.html', {'form': form})

def form_success(request):

    return render(request, 'form_success.html')
```

<!-- customer_form.html -->

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Customer Form</title>

</head>

<body>

    <h2>Customer Form</h2>

    <form method="post">

        {% csrf_token %}

        {{ form.as_p }}

        <!-- Additional field -->

        <div>

            <label for="{{ form.message.id_for_label }}">Message:</label>

            {{ form.message }}

        </div>

        <button type="submit">Submit</button>

    </form>
```

```
</body>

</html>
```

6

a) **Write a python function for validating the mobile number with validation condition as 10 digits, ‘should not start with 0, + can be at the beginning**.

Solution:

forms.py

```
from django import forms

def mobile_no(value):
    mobile = str(value)

    if len(mobile) != 10:
        raise forms.ValidationError("Mobile Number should have 10 digits.")

    if mobile[0] == '0':
        raise forms.ValidationError("Mobile Number should not start with '0'.")

    if not mobile.isdigit():
        raise forms.ValidationError("Mobile Number should only contain digits.")

    if not mobile.replace('+', ").isdigit():
        raise forms.ValidationError("Mobile Number should only contain digits and optionally
a leading '+'.")

class StuForm(forms.Form):
  mob = forms.IntegerField(validators =[mobile_no])
```

b) **Describe how to use Django's generic ListView to display a list of items from a model named Post with fields title, content, and published_date. Provide the necessary code snippets for the model, view, URL configuration, and template.**

Solution:

Steps:
1. Define the `Post` Model
# models.py

```
from django.db import models

class Post(models.Model):
    title = models.CharField(max_length=200)
    content = models.TextField()
    published_date = models.DateTimeField(auto_now_add=True)

    def __str__(self):
        return self.title
```

2. Create a `ListView` in Views

```
# views.py
from django.views.generic import ListView
from .models import Post

class PostListView(ListView):
    model = Post
    template_name = 'post_list.html'
    context_object_name = 'posts'
    ordering = ['-published_date']  # Optional: Order posts by published date descending
    paginate_by = 10  # Optional: Paginate by 10 posts per page
```

3. URL Configuration

```
# urls.py
from django.urls import path
from .views import PostListView

urlpatterns = [
    path('posts/', PostListView.as_view(), name='post_list'),
    # Add other URLs as needed
]
```

4. Create a Template for Listing Posts

```
<!-- post_list.html -->

<body>
    <h1>Post List</h1>
    <ul>
        {% for post in posts %}
        <li>
            <h2>{{ post.title }}</h2>
            <p>{{ post.content }}</p>
            <p>Published on: {{ post.published_date }}</p>
        </li>
        {% empty %}
        <li>No posts found.</li>
```

	{% endfor %}
	    </ul>
	</body>