SET-4

PRACTICAL-1

❖ AIM : A program that creates a web application that uses a template engine to generate dynamic HTML pages.

Problem Statement:

Develop a Python program to build a web application using Django framework with the capability to generate dynamic HTML pages using a template engine.

Program Description:

The aim is to create a web application that utilizes Django, a high-level Python web framework, to dynamically generate HTML pages. This will involve setting up a Django project, defining models and views, configuring URL routing, and integrating a template engine to render dynamic content.

Procedure:

1) **Setting up Django project:**
   - Install Django using pip:

```
bash
```

```
pip install django
```

   - Create a new Django project:

```
bash
```

```
django-admin startproject myproject
```

   - Navigate to the project directory:

```
bash
```

```
cd myproject
```

2) **Creating Django app:**

```
bash
```

```
python manage.py startapp myapp
```

3) **Configuring project-level url routing:**

- Add a path inside (myproject/myproject/urls.py) using include function from django.urls. Example:

urls.py

```
from django.contrib import admin

from django.urls import path, include



urlpatterns = [

    path('admin/', admin.site.urls),

    path('myapp/',include("myapp.urls"))

]
```

4) Defining the Application:

- Add the application as “myapp” inside the INSTALLED_APPS variable inside (myproject/myproject/settings.py). Example:

settings.py

```
# Application definition

INSTALLED_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    "myapp"

]
```

5) Defining models:

- Define models in models.py file within the app directory (myapp/models.py). Example:

models.py

```
from django.db import models


class MyModel(models.Model):

    name = models.CharField(max_length=100)

    # Add more fields as needed
```

6) Defining views:

- Define views in views.py file within the app directory (myapp/views.py). Example:

views.py

```
from django.shortcuts import render

from .models import MyModel


def my_view(request):

    data = MyModel.objects.all()

    return render(request, 'my_template.html', {'data': data})
```

7) Configuring URL routing:

- Define URL patterns in urls.py file within the app directory (myapp/urls.py). Example:

views.py

```
from django.urls import path

from . import views

urlpatterns = [

    path('', views.my_view, name='my_view'),

    # Add more URL patterns as needed

]
```

8) **Configuring URL routing:**

- Create HTML templates in **templates** directory within the app directory (**myapp/templates**). Example:

my_templates.html

```
<html>

<head>

    <title>Dynamic Page</title>

</head>

<body>

    <h1>Dynamic Content</h1>

    <ul>

        {% for item in data %}

            <li>{{ item.name }}</li>

        {% endfor %}

    </ul>

</body>

</html>
```

9) **Running the server:**

- Start the Django development server**:**

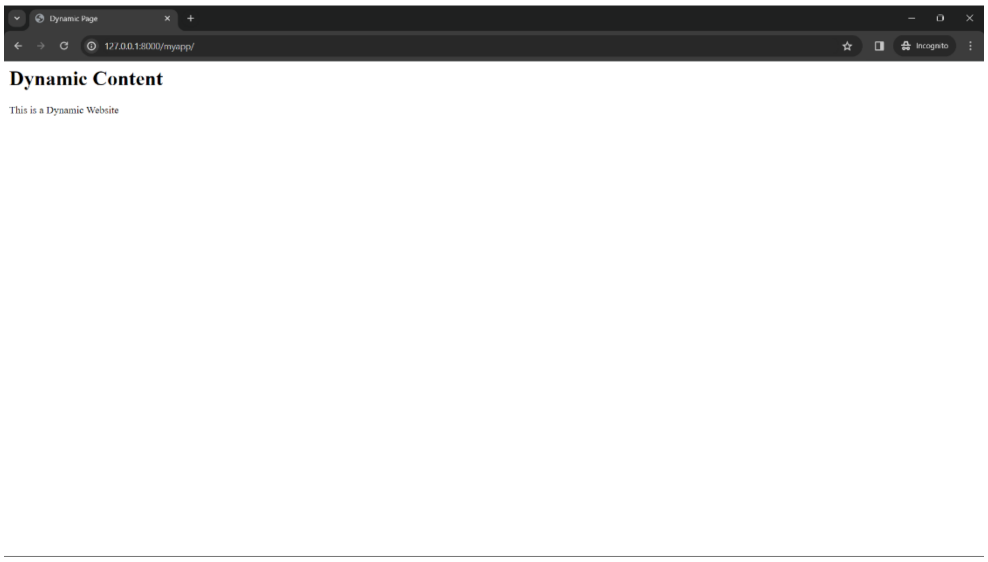
bash

```
    python manage.py runserver
```

10) **Accessing the application:**

- Open a web browser and navigate to http://localhost:8000 to view the dynamically generated HTML page.

Expected Output:



Dynamic Content

This is a Dynamic Website

Result:

In conclusion, we have successfully developed a Python program using Django framework to create a web application capable of generating dynamic HTML pages using a template engine. By following the outlined procedure, we were able to set up the necessary components, define models and views, configure URL routing, and integrate a template engine to render dynamic content, thus achieving the desired aim of the project.

PRACTICAL- 2

❖ AIM: A program that creates a web application that supports AJAX requests and updates the page without reloading.

Problem Statement:

Develop a Python program to build a web application using Django framework with AJAX support, allowing seamless updates to the page content without requiring a full page reload.

Program Description:

The objective is to create a web application that utilizes Django framework along with AJAX (Asynchronous JavaScript and XML) to enable dynamic updates to the page content without reloading the entire page. This involves setting up Django views to handle AJAX requests, creating JavaScript functions to make asynchronous requests to the server, and updating the DOM (Document Object Model) dynamically based on the server's response.

Procedure:

1) Setting up Django project:

- Install Django using pip:

bash

```
pip install django
```

- Create a new Django project:

bash

```
django-admin startproject myproject
```

- Navigate to the project directory:

bash

```
cd myproject
```

2) Creating Django app:

bash

```
python manage.py startapp myapp
```

3) Configuring Project-level url routing:

- Add a path inside (myproject/myproject/urls.py) using include function from django.urls. Example:

urls.py

```
from django.contrib import admin

from django.urls import path, include



urlpatterns = [

    path('admin/', admin.site.urls),

    path('myapp/',include("myapp.urls"))

]
```

4) Defining the Application:

- Add the application as "myapp" inside the INSTALLED_APPS variable inside (myproject/myproject/settings.py). Example:

settings.py

```
# Application definition

INSTALLED_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    "myapp"

]
```

5) Defining views for AJAX requests:

- Define views in **views.py** file within the app directory (**myapp/views.py**). Example:

views.py

```
from django.http import JsonResponse

def ajax_update(request):
    # Process AJAX request and prepare data
    data = {'message': 'Updated content from AJAX request!'}
    return JsonResponse(data)
```

6) Configuring URL routing:

- Define URL patterns in **urls.py** file within the app directory (**myapp/urls.py**). Example:

urls.py

```
from django.urls import path

from . import views



urlpatterns = [

    path('ajax_update/', views.ajax_update, name='ajax_update'),

    # Add more URL patterns as needed

]
```

7) HTML Structure with a Placeholder for Dynamic Content and JavaScript function for AJAX request:

- Create an HTML file with a placeholder for the dynamic content and write JavaScript code to send AJAX request and update the page content dynamically. Example:

index.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-
scale=1.0">

    <title>AJAX Example</title>

</head>
```

```
<body>

    <h1>Dynamic Content Update</h1>

    <div id="content">Placeholder for dynamic content</div>

    <button onclick="updateContent()">Update Content</button>

    <script>

        function updateContent() {

       fetch('/myapp/ajax_update/')

           .then(response => response.json())

           .then(data => {

               // Update DOM with new content

               document.getElementById('content').innerHTML    =
   data.message;

           })

           .catch(error => console.error('Error:', error));

   }

     </script>

</body>

</html>
```

11) Running the server:

- Start the Django development server:

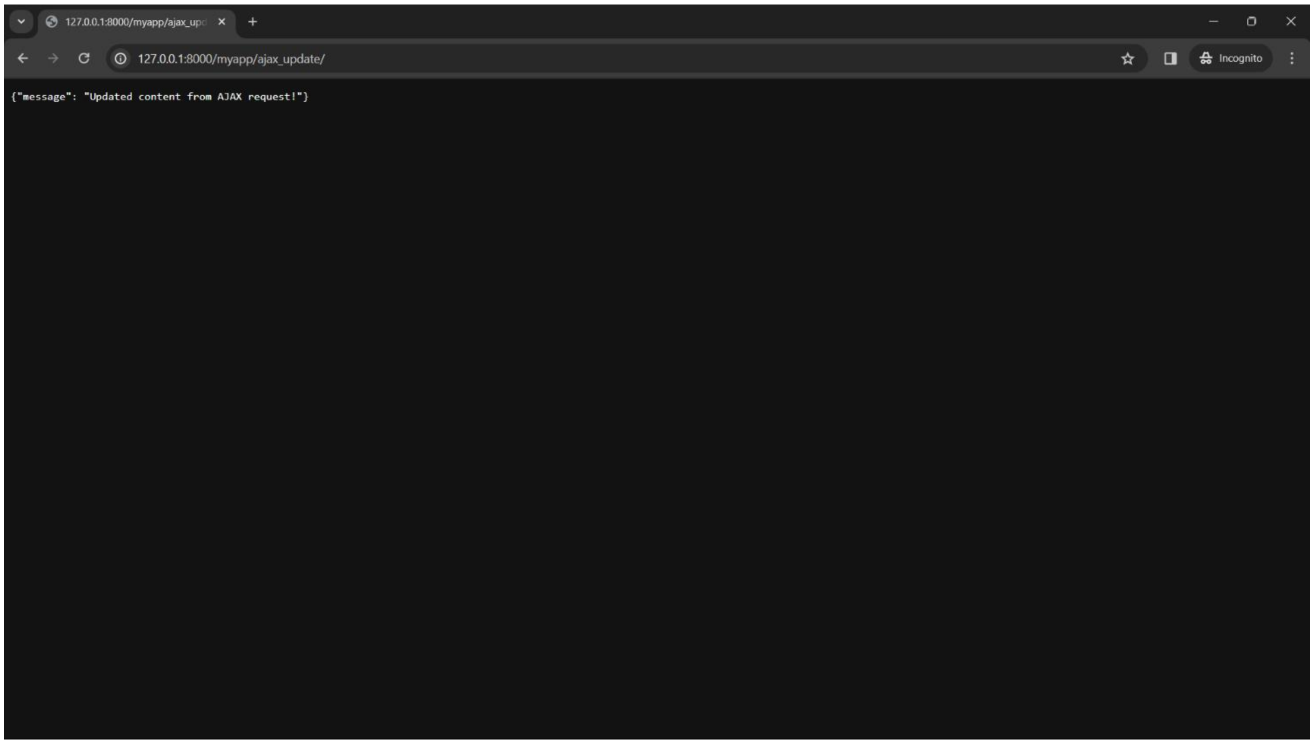
bash

```
python manage.py runserver
```

12) Accessing the application:

- Open a web browser and navigate to the URL of your Django application to see the HTML page. Clicking the "Update Content" button will trigger an AJAX request to update the content dynamically without reloading the page.

Expected Output:



127.0.0.1:8000/myapp/ajax_update/

```
{"message": "Updated content from AJAX request!"}
```

Result:

In conclusion, we have successfully developed a Python program using Django framework to create a web application that supports AJAX requests and updates the page content dynamically without requiring a full page reload. By following the outlined procedure, we were able to set up Django views to handle AJAX requests, create JavaScript functions to make asynchronous requests to the server, and update the DOM dynamically based on the server's response, thus achieving the desired aim of the project.

PRACTICAL- 3

❖ AIM: A program that creates a web application that uses Django's built-in debugging features to troubleshoot errors and exceptions.

Problem Statement:

Develop a Python program to build a web application using Django framework while leveraging Django's built-in debugging features to effectively troubleshoot errors and exceptions that occur during development.

Program Description:

The aim is to create a web application using Django framework and utilize its built-in debugging features to identify and resolve errors and exceptions encountered during development. Django provides powerful debugging tools such as detailed error pages, traceback information, and integration with Python's logging framework, which help developers quickly diagnose and fix issues in their applications.

Procedure:

1) Setting up Django project:

- Install Django using pip:

bash

```
pip install django
```

- Create a new Django project:

bash

```
django-admin startproject myproject
```

- Navigate to the project directory:

bash

```
cd myproject
```

2) Creating Django app:

bash

```
python manage.py startapp myapp
```

3) Enable Debugging in Django settings:

- Ensure that Django's debug mode is enabled in the project's settings file (**myproject/settings.py**):

settings.py

```
DEBUG = True
```

4) Creating a view for the calculator:

- Define a view in **myapp/views.py** to handle user input and calculate the square:

Views.py

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

def calculate_square(request):
    try:
        if request.method == 'POST':
            number = int(request.POST.get('number'))
            result = number ** 2
            return HttpResponse(f"The square of {number} is 
{result}.")
    except Exception as e:
        return HttpResponse(f"An error occurred: {e}")

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

5) Creating HTML template for the calculator:

- Create an HTML template **index.html** in **calculator/templates** directory:

bash

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Calculator</title>

</head>

<body>

    <h1>Calculate Square</h1>
```

```
         <form  method="post"  action="{%  url  'calculate_square'
%}">
             {% csrf_token %}
             <label for="number">Enter a number:</label>
             <input   type="number"   id="number"   name="number"
required>
             <button type="submit">Calculate Square</button>
         </form>
     </body>
     </html>
```

6)Configuring URL routing:

- Define URL patterns in **urls.py** file within the app directory (**myapp/urls.py**). Example:

urls.py

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

urlpatterns = [
    path('', views.calculator_square, name=calculator_square'),
    # Add more URL patterns as needed
]
```

7)Configuring Project-level url routing:

- Add a path inside (myproject/myproject/urls.py) using include function from django.urls. Example:

urls.py

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


urlpatterns = [
    path('admin/', admin.site.urls),

    path('myapp/',include("myapp.urls"))

]
```

8) Running the server:

- Start the Django development server:

bash

```
python manage.py runserver
```

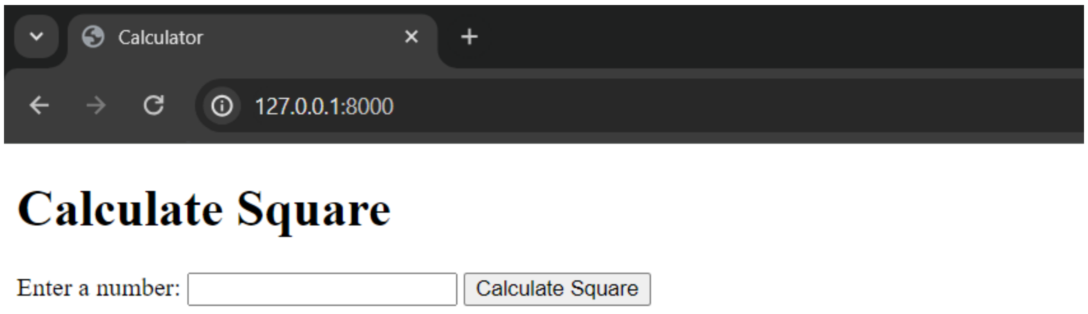
9) Accessing the application:

Open a web browser and navigate to **http://127.0.0.1:8000**. You should see the calculator form. Enter a number, submit the form, and you should see the square of the number displayed.

10) Debugging:

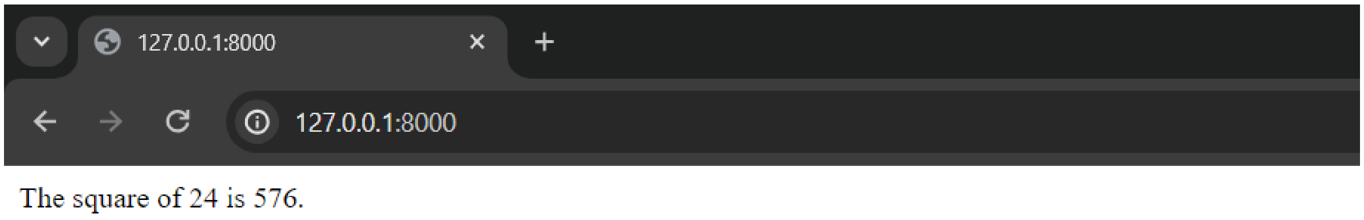
- If any errors occur during development, Django will display detailed error pages with traceback information, making it easier to identify and fix issues.
- Use Django's logging framework to log messages for debugging purposes. For example, you can log the value of variables or messages at various points in your code to understand its behavior.
- Use Django's management commands, such as manage.py shell, to interactively test code snippets and troubleshoot issues.
- Integrate a debugger like pdb or ipdb to set breakpoints, inspect variables, and step through code execution to identify and resolve complex issues.

• <u>Expected Output:</u>



Calculate Square

Enter a number:



```
The square of 24 is 576.
```

Result:

In conclusion, by leveraging Django's built-in debugging features and following the outlined procedure, we have successfully developed a Python program to create a web application using Django framework while effectively troubleshooting errors and exceptions encountered during development. Django's debugging tools, including detailed error pages, traceback information, logging framework integration, management commands, interactive shell, and debugger integration, have proven instrumental in identifying and resolving issues, thus ensuring the smooth functioning of the application.

PRACTICAL- 4

❖ AIM: Develop a Python program to create a web application using Django framework that implements user authentication and authorization to control access to resources based on user roles and permissions.

Problem Statement:

Develop a Python program to create a web application using Django framework that implements user authentication and authorization to control access to resources based on user roles and permissions.

Program Description:

The objective is to build a web application using Django framework with user authentication and authorization functionalities. User authentication ensures that users can securely log in to the application, while authorization controls access to specific resources or functionalities based on the user's role or permissions. Django provides built-in tools for implementing these features, including authentication backends, user models, and permission classes.

Procedure:

1) Setting up Django project:

- Install Django using pip:

```bash
pip install django
```

- Create a new Django project:

```bash
django-admin startproject myproject
```

- Navigate to the project directory:

```bash
cd myproject
```

2) Creating Django app:

```bash
```

```
python manage.py startapp myapp
```

3) Configuring Project-level url routing:

- Add a path inside (myproject/myproject/urls.py) using include function from django.urls. Example:

urls.py

```
from django.contrib import admin

from django.urls import path, include



urlpatterns = [

    path('admin/', admin.site.urls),

    path('myapp/',include("myapp.urls"))

]
```

4) Defining the Application:

- Add the application as "myapp" inside the INSTALLED_APPS variable inside (myproject/myproject/settings.py). Example:

settings.py

```
# Application definition

INSTALLED_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    "myapp"

]
```

5) Configuring User Authentication:

- Django provides a built-in authentication system that includes user models, forms, views, and authentication backends.
- Customize the user model in **myapp/models.py** if needed to add additional fields or functionality:

models.py

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



class CustomUser(AbstractUser):

    # Add custom fields as needed

    Pass
```

6) Implementing User Registration and Login:

- Create views and templates for user registration, login, logout, and password reset as needed.
  - Views:

views.py

```
# myapp/views.py



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

from django.contrib.auth.forms import UserCreationForm,
AuthenticationForm

from django.contrib.auth.views import PasswordResetView

from django.shortcuts import render, redirect



def register(request):

    if request.method == 'POST':

        form = UserCreationForm(request.POST)

        if form.is_valid():

            form.save()
```

```
                return redirect('login')

        else:

            form = UserCreationForm()

        return render(request, 'registration/register.html',
    {'form': form})



    def user_login(request):

        if request.method == 'POST':

            form = AuthenticationForm(request, request.POST)

            if form.is_valid():

                username = form.cleaned_data.get('username')

                password = form.cleaned_data.get('password')

                user = authenticate(username=username,
    password=password)

                if user is not None:

                    login(request, user)

                    return redirect('home')

        else:

            form = AuthenticationForm()

        return render(request, 'registration/login.html', {'form':
    form})



    def user_logout(request):

        logout(request)

        return redirect('login')
```

- Templates:

registration/register.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Register</title>

</head>

<body>

    <h1>Register</h1>

    <form method="post">

        {% csrf_token %}

        {{ form.as_p }}

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

    </form>

</body>

</html>
```

registration/register.html

```
<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Login</title>

</head>
```

```
    <body>

        <h1>Login</h1>

        <form method="post">

            {% csrf_token %}

            {{ form.as_p }}

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

        </form>

    </body>

    </html>
```

- Customize the user model in **myapp/models.py** if needed to add additional fields or functionality:

models.py

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


class CustomUser(AbstractUser):

    # Add custom fields as needed

    Pass
```

7) Url Configuration:

urls.py

```
from django.urls import path

from . import views


urlpatterns = [

    path('register/', views.register, name='register'),

    path('login/', views.user_login, name='login'),

    path('logout/', views.user_logout, name='logout'),
        # Add more URL patterns as needed
    ]
```

8) Running the server:

- Start the Django development server:

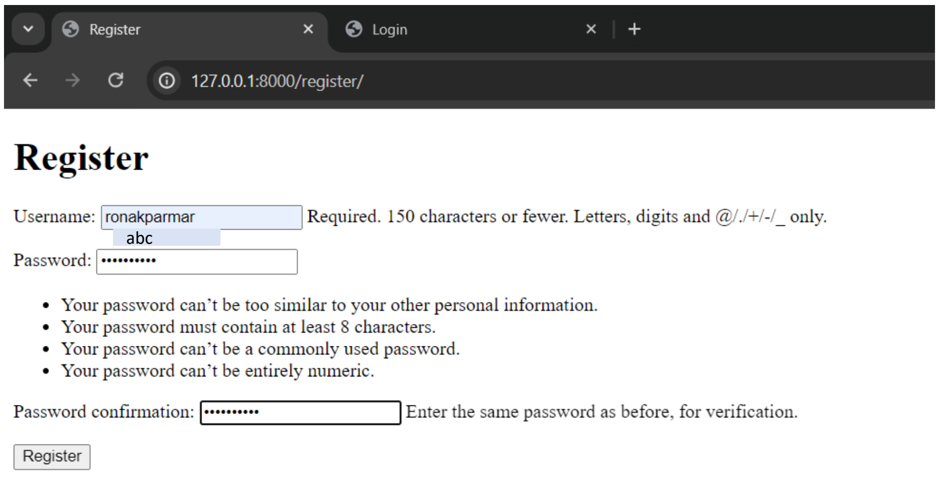
bash

```
    python manage.py runserver
```

9) Accessing the application:

Open a web browser and navigate to the URLs of your Django application to register, log in, and access authenticated or authorized views based on user roles and permissions.

Expected Output:



Register

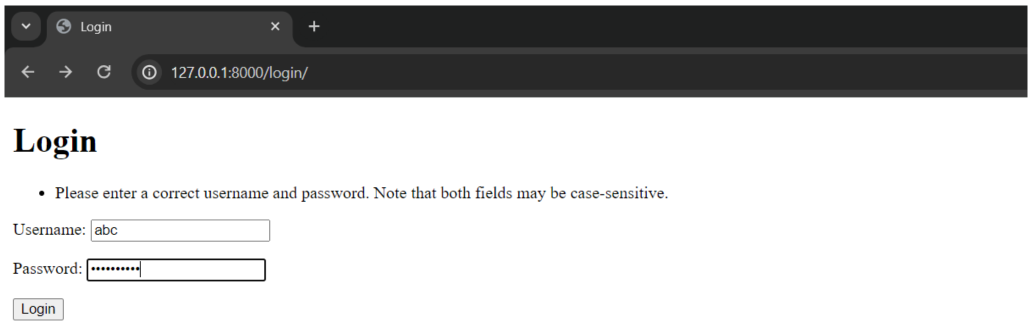
Username: ronakparmar Required. 150 characters or fewer. Letters, digits and @/./+/-/_ only.
abc

Password: ••••••••••

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

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

Register



Login

- Please enter a correct username and password. Note that both fields may be case-sensitive.

Username: abc

Password: ••••••••••

Login

Result:

In conclusion, by following the outlined procedure and leveraging Django's built-in tools for user authentication and authorization, we have successfully developed a Python program to create a web application using Django framework with robust user management capabilities. Django's authentication system, permission model, and authentication-related views and decorators enable developers to implement secure user authentication and fine-grained access control to resources, ensuring the integrity and security of the application.

PRACTICAL- 5

❖ AIM: A program that creates a web application that integrates with third-party APIs to provide additional functionality.

Problem Statement:

Develop a Python program to create a web application using Django framework that integrates with third-party APIs to enhance its functionality. The goal is to leverage external APIs to provide additional features or data within the Django application.

Program Description:

The objective is to build a web application using Django framework that interacts with third-party APIs to extend its capabilities. This can include integrating with services such as weather APIs, social media APIs, payment gateways, mapping services, or any other external service that provides useful functionality or data.

Procedure:

1) Setting up Django project:

- Install Django using pip:

bash

```
pip install django
```

- Create a new Django project:

bash

```
django-admin startproject myproject
```

- Navigate to the project directory:

bash

```
cd myproject
```

2) Creating Django app:

bash

```
python manage.py startapp myapp
```

3) Configuring Project-level url routing:

- Add a path inside (myproject/myproject/urls.py) using include function from django.urls. Example:

urls.py

```
from django.contrib import admin

from django.urls import path, include



urlpatterns = [

    path('admin/', admin.site.urls),

    path('myapp/',include("myapp.urls"))

]
```

4) Defining the Application:

- Add the application as “myapp” inside the INSTALLED_APPS variable inside (myproject/myproject/settings.py). Example:

settings.py

```
# Application definition

INSTALLED_APPS = [

    'django.contrib.admin',

    'django.contrib.auth',

    'django.contrib.contenttypes',

    'django.contrib.sessions',

    'django.contrib.messages',

    'django.contrib.staticfiles',

    "myapp"

]
```

5) Create Functions for API Integration:

- First, you'll create a function in your Django app (**myapp**) to interact with the Public APIs API.

utils.py

```
import requests



def get_public_apis():

    url = "https://api.publicapis.org/entries"

    response = requests.get(url)

    if response.status_code == 200:

        return response.json()

    else:

        return None
```

6) Implement Views to Display API Data:

- Next, you'll create a view in **myapp/views.py** to render data obtained from the Public APIs API.

views.py

```
from django.shortcuts import render

from .utils import get_public_apis



def public_apis_view(request):

    api_data = get_public_apis()

    return render(request, ‘index.html', {'api_data':
api_data})
```

7) Create HTML Templates:

- Then, you'll develop an HTML template in the **myapp/templates** directory to display the data fetched from the Public APIs API.

index.html

```
<html lang="en">

<head>

    <meta charset="UTF-8">

    <title>Public APIs</title>

</head>

<body>

    <h1>Public APIs</h1>

    <ul>

        {% for entry in api_data.entries %}

            <li>{{ entry.API }} - {{ entry.Description
}}</li>

        {% endfor %}

    </ul>

</body>

</html>
```

8) Configure URL Routing:

- Define a URL pattern in **myapp/urls.py** to map the view to a URL endpoint.

urls.py

```
from django.urls import path

from . import views



urlpatterns = [

    path('public-apis/', views.public_apis_view,
name='public_apis'),
    ]
```

9) Running the server:

- Start the Django development server**:**

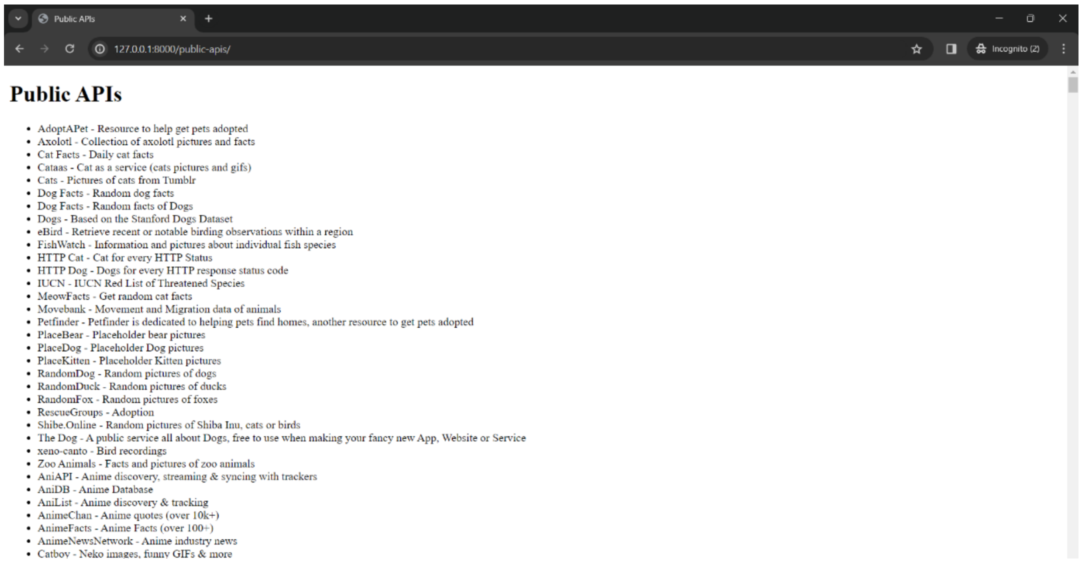
bash

```bash
python manage.py runserver
```

10) Accessing the application:

Open a web browser and navigate to the specified URL (e.g., **http://127.0.0.1:8000/public-apis/**) to access the page displaying data fetched from the Public APIs API.

Expected Outcomes:



Public APIs

- AdoptAPet - Resource to help get pets adopted
- Axolotl - Collection of axolotl pictures and facts
- Cat Facts - Daily cat facts
- Cataas - Cat as a service (cats pictures and gifs)
- Cats - Pictures of cats from Tumblr
- Dog Facts - Random dog facts
- Dog Facts - Random facts of Dogs
- Dogs - Based on the Stanford Dogs Dataset
- eBird - Retrieve recent or notable birding observations within a region
- FishWatch - Information and pictures about individual fish species
- HTTP Cat - Cat for every HTTP Status
- HTTP Dog - Dogs for every HTTP response status code
- IUCN - IUCN Red List of Threatened Species
- MeowFacts - Get random cat facts
- Movebank - Movement and Migration data of animals
- Petfinder - Petfinder is dedicated to helping pets find homes, another resource to get pets adopted
- PlaceBear - Placeholder bear pictures
- PlaceDog - Placeholder Dog pictures
- PlaceKitten - Placeholder Kitten pictures
- RandomDog - Random pictures of dogs
- RandomDuck - Random pictures of ducks
- RandomFox - Random pictures of foxes
- RescueGroups - Adoption
- Shibe.Online - Random pictures of Shiba Inu, cats or birds
- The Dog - A public service all about Dogs, free to use when making your fancy new App, Website or Service
- xeno-canto - Bird recordings
- Zoo Animals - Facts and pictures of zoo animals
- AniAPI - Anime discovery, streaming & syncing with trackers
- AniDB - Anime Database
- AniList - Anime discovery & tracking
- AnimeChan - Anime quotes (over 10k+)
- AnimeFacts - Anime Facts (over 100+)
- AnimeNewsNetwork - Anime industry news
- Catboy - Neko images, funny GIFs & more

Result:

By following the outlined procedure and integrating with third-party APIs, we have successfully developed a Python program to create a web application using Django framework with enhanced functionality. Leveraging external APIs allows us to enrich the application with additional features and data, providing a richer user experience and expanding the capabilities of the Django application.