INTRODUCTION

➢ DEFINITION OF FULL STACK WEB DEVELOPMENT

The process of planning, building, testing, and launching an entire web application from beginning to end is known as full stack development. Full stack development is the combination of both the front end (client side) and back end (server side) portions of a web application.

➢ CLIENT SIDE (FRONT END)

The front end is also known as the client side; it is a part of the website that the user sees and interacts with. It consists of a user interface (UI) of the website, and it runs on the user's local system, instead of where it is hosted.

➢ SERVER SIDE (BACK END)

The back end is also known as the server side; it is responsible for all the logic and functions of any website. It includes things like database routing and API creation. The back end ensures that the server is up all the time and also manages the incoming traffic to the website.

Full stack developers require different skills and tools to develop the front end and back end of any web application. Let's get into the role of a full stack developer.

➢ WHO IS A FULL STACK DEVELOPER?

A full stack developer is an engineer who is proficient in working with both server-side and client-side programming software. They manage everything from front end development languages, back-end development methodologies, server handling, and Application Programming Interface (API) designing to working with version control systems.

➢ WHY WOULD YOU NEED A FULL STACK DEVELOPER?

Following are the multiple reasons why organizations require a Full Stack Developer:

- Full Stack developers ensure to keep the end-to-end web application running without any hiccups.
- Full stack developers can fit into multiple roles in the application development process, thereby greatly reducing the costs and time required to solve problems.
- Full Stack developers can actively debug applications alongside development and also extend help in testing and actively developing contingency protocols for the application.

➢ ADVANTAGES OF FULL STACK DEVELOPMENT

1. **Versatility and Efficiency**

One primary benefit of having full stack development is its versatility. A full stack developer is proficient in multiple programming languages and frameworks, which helps them handle multiple aspects of a project. A single developer can manage the front end as well as the back end, making the development process seamless and efficient.

2. **Rapid Prototyping and MVP Development**

Prototyping and the minimum viable product (MVP) are easy to achieve when we use full stack development. Full stack developers have a wide range of skills that enable them to rapidly produce end-to-end working web applications.

3. **Enhanced Problem Solving**

Full stack developers have a broad understanding of the entire tech stack used to build the whole application, which makes it easy to find and resolve bugs.

➢ FULL STACK DEVELOPMENT TECHNOLOGIES

- A full stack developer needs to know certain tools and technologies based on the application they are creating. Mainly, the Full Stack Development Technologies are divided into two parts: one for front end creation and the other for the back end.
- The front end majorly focuses on the design part of the website. There are some very popular front end programming languages to build the UI for websites.
    - HTML (Hyper Text Makeup Language): It is used to create the structure or skeleton of any website. By using HTML, you can add data and visual content to your website.

- CSS (Cascading Style Sheets): This is used for styling the HTML components to make them more attractive and create a more appealing layout by adding colors to them.
- JavaScript: Javascript can be used as a front end and back end programming language. In the front end, we use it to make dynamic and interactive components such as a responsive navbar that turns into a hamburger menu when the screen size changes.

TECHNOLOGIES TO DEVELOP BACK END

- The back end handles the data transfer between the front end and the database or server; it connects the webpage to the main server and establishes the connection between them. The most commonly used back end development languages are PHP, Java, Python, and Node.js.
- The back end can be further divided into 3 sub-layers:
  - **API Layer:** It is responsible for connecting the back end to the front end.
  - **Storage Layer:** It stores all the data coming from the front end; it also manages access to data based on the conditions provided by the developer.
  - **Logic Layer:** This layer consists of all the logic for the website. It is the most important part because it ensures the working of any website.

➢ **SKILLS REQUIRED TO BECOME A FULL STACK DEVELOPER**

- A full stack developer requires a specialty in computer science and should have a high level of proficiency in both back end and front end programming languages and frameworks. They are skilled in PHP, Django, Node.js, Express.js, JavaScript, and HTML.

- Full stack developers are also knowledgeable about a wide range of DBMS (database management systems), including MongoDB, PostgreSQL, and MySQL.

➢ WHAT ARE THE POPULAR FRAMEWORKS FOR FULL STACK DEVELOPMENT?

There are more than 100 full stack frameworks available out there, but you don't need to learn all of them to become a full stack developer.

Here are the top 5 full stack frameworks:

- **Node.js and Expreess.js (Javascript frameworks)**

Node.js and Express are the two most popular frameworks for full stack development. According to W3Tech, around 3.1% of websites are developed using Node.js, which is around 6.3 million websites.

Node.js is a runtime Javascript framework that runs on the server side and Express.js works on top of node.js, providing robust features for building websites. It provides simplicity, flexibility, and cross-platform functionality, and it also provides great community support.

- **Ruby on Rails**

Ruby on Rails offers many features, like continuous updates and a vast open-source library. Apart from that, it is very easy to learn and implement. Some of the most popular websites are built using Ruby on Rails, which includes Github, Airbnb, Shopify, and many more. It also offers security, and fast processing, and the architecture is based on a model view controller, making it one of the best full stack development frameworks.

- **Django**

One of the greatest frameworks for full stack web development is Django, which is used by companies like Google, Instagram, YouTube, and NASA. It enables

programmers to rapidly construct online apps without having to worry about tasks like creating HTML templates or database management. A template engine for producing HTML views, an object-relational mapper (ORM) for communicating with databases, and a multitude of tools and libraries for common tasks are all included in Django.

- **Spring Boot**

Spring boot is one of the most flexible and compatible Java frameworks. It helps in developing a production-ready application, which means that you can directly deploy the application without worrying about bugs and errors.

Another benefit of using spring-boot is that it offers an extensive range of integrations and customizations, so you can make it work exactly how you want. Spring Boot is a great option, whether you're searching for an easy approach to building web applications that are ready for production or need a very configurable framework.

- **Laravel**

Laravel is a thoroughly documented PHP web application framework that offers a clean, intelligent syntax that helps you create web applications quickly and easily. Larvel comes with lots of functionalities, including ORM (Object Relational Mapping), routing, and authentication. Larvel also offers a huge collection of libraries and built-in methods that are useful in maintaining and developing web applications.

Laboratory Component - 1:

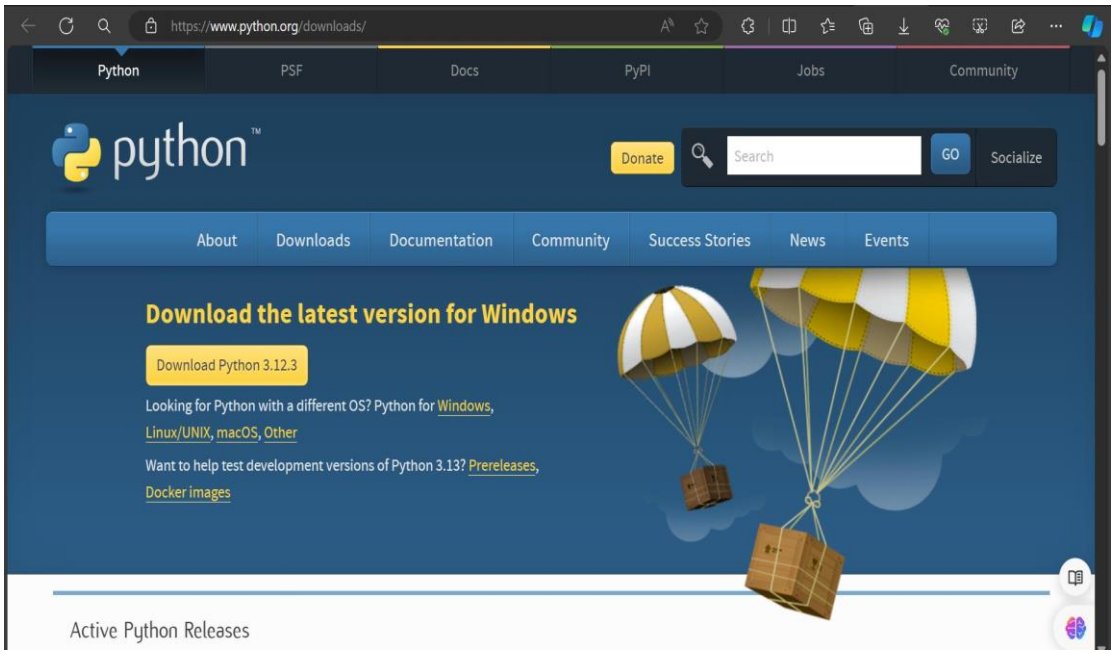
1. **Installation of Python, Django and Visual Studio code editors can be demonstrated.**
2. **Creation of virtual environment, Django project and App should be demonstrated**
3. **Develop a Django app that displays current date and time in server**
4. **Develop a Django app that displays date and time four hours ahead and four hours before as an offset of current date and time in server.**

1.1 Installation of Python, Django and Visual Studio code editors can be demonstrated.

Installation of Python, Django and Visual Studio code editors can be demonstrated.

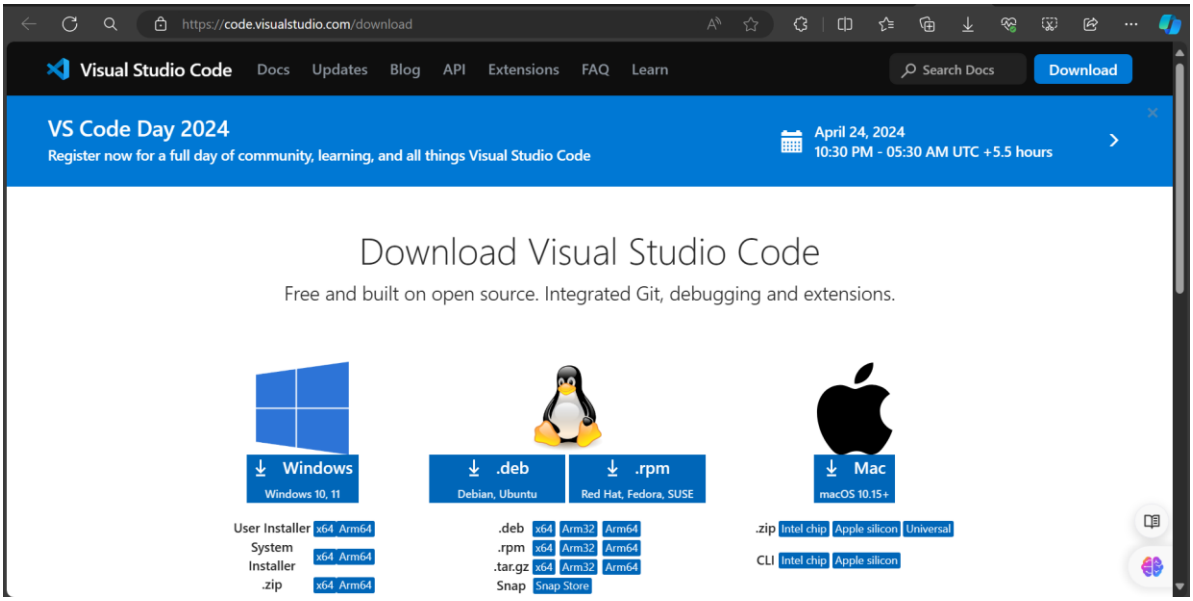
a) Python:

1. Download the latest Python installer from **https://www.python.org/downloads/**. (Python 3.11.5preferred).
2. Run the installer and follow the on-screen instructions. Ensure "**Add Python to PATH**" is checked foreasy access from the command line.
3. Open a command prompt or terminal and type **python --version** to verify installation.



b) Visual Studio Code:

1. Download and install Visual Studio Code from **https://code.visualstudio.com/download**.
2. Install the Python extension for code completion and debugging within VS Code.



3. On your file system, create a folder, such as django_lab
4. In that folder, use the following command (as appropriate to your computer) to create a virtual environment named **myenv** based on your current interpreter (in cmd prompt).

```
# Linux
sudo apt-get install python3-
venvpython3 -m venv myenv
source myenv/bin/activate



# macOS
python3 -m venv myenv
source
myenv/bin/activate

# Windows
py -3 -m venv myenv
myenv\scripts\activate
```

5.Install Django in the virtual environment by running the following command in the cmd prompt Terminal: python -m pip install django

c) To create a Django project in Visual Studio, you can follow these steps:

Step 1: Open Visual Studio: Launch Visual Studio and ensure you have the necessary Python development tools installed. You can install them during the Visual Studio installation process or later via the Visual Studio Installer.

Step 2: Create a New Django Project:

- Go to File > New > Project....
- In the "New Project" dialog, search for "Django" using the search box at the top.
- Choose "Django Web Project" template.
- Enter the project name and location.
- Click on the "Create" button.

Step 3: Define Models, Views, and Templates: Inside your Django app folder (usually named app or projectname), you can define models, views, and templates as described in the previous Python code snippets.

Step 4: Run the Server: You can run the Django server directly from Visual Studio. You'll typically find a green "play" button to start the server.

Step 5: Access Your App: Once the server is running, you can access your Django app by navigating to the provided URL, usually http://127.0.0.1:8000/, in your web browser.

Step 5: Code and Debug: Visual Studio provides a rich environment for coding and debugging Django projects. You can set breakpoints, inspect variables, and utilize other debugging features to troubleshoot your code.

1.2 Creation of virtual environment, Django project and App should be demonstrated.

Solution: The process of creating a virtual environment, setting up a Django project, and creating a Django app. Here are the steps:

Step 1: Create a Virtual Environment:

2 Open your terminal or command prompt.

3 Navigate to the directory where you want to create your Django project.

4 Run the following command to create a virtual environment named "venv" (you can choose any name): **python -m venv env1**

5 Activate the virtual environment: **On Windows: env1\Scripts\activate**

Step 2: Install Django:

6 Once the virtual environment is activated, install Django using pip: pip install Django

Step 3: Create a Django Project:

7 After installing Django, create a new Django project using the following command: **django-admin startproject lab1**

8 Replace "lab1" with your desired project name.

Step 4: Navigate to the Project Directory:

- Change to the newly created project directory: **cd lab1**

Step 5: Create a Django App:

9 Inside the project directory, create a Django app using the following command: **python manage.py startapp pg1**

Step 6: Register the App in Settings:

- Open the settings.py file inside the myproject directory.
- Find the INSTALLED_APPS list and add your app's name ('myapp') to the list.

  INSTALLED_APPS = [… 'myapp',]

Step 7: Run Migrations (Optional):

10 If your app has models and you need to create database tables, run the following commands: **python manage.py makemigrations**

python manage.py migrate

11 Now you have successfully created a virtual environment, a Django project, and a Django app. You can start working on your Django app by defining models, views, templates, and URLs as needed.

1.3 Develop a Django app that displays current date and time in server

Solution:

To develop a Django app that displays the current date and time on the server, follow these steps:

Step 1: Create a Django App named pg1.

Step 2: Define a View:

12 Open the views.py file inside your **pg1** directory and define a view to display the current date and time.

13 Here's an example:

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

def current_datetime(request):
    now = datetime.now()
    html = f"<html><body><h1>Current Date and
            Time:</h1><p>{now}</p></body></html>"
    return HttpResponse(html)
```

Step 3: Create a URL Configuration:

- Open the **urls.py** file inside your **myapp** directory and create a URL configuration to map the view you just defined to a URL.
- Here's an example:

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

urlpatterns = [
    path('current_datetime/', views.current_datetime,
         ='current_datetime'),
]
```

Step 4: Update Project URL Configuration:

- Open the **urls.py** file inside your project directory (the directory containing **settings.py**) and include your app's URLs by importing them and adding them to the **urlpatterns** list.
- Here's an example:

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

urlpatterns = [
    path('admin/', admin.site.urls),
     path('pg1/', include('pg1.urls')), # Include your app's URLs
]
```

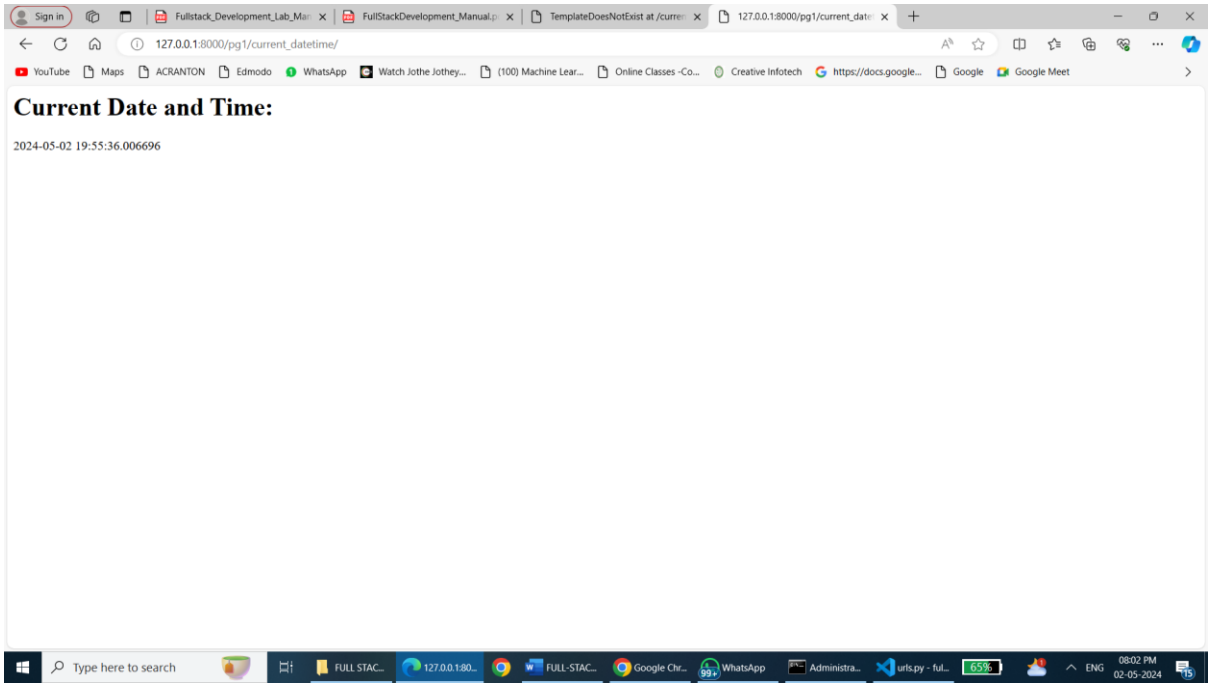
Step 5: Run the Development Server:

- Start the Django development server to testyour app.
- In your terminal, make sure you're in the project directory (where **manage.py** is located) and run the following command:

```
python manage.py runserver
```

Step 6: Access the App:

- Open a web browser
- go to **127.0.0.1:8000/pg1/current_datetime/** to see the current date and time displayed on the page.



1.4 Develop a Django app that displays date and time four hours ahead and four hours before as an offset of current date and time in server.

Solution:

To develop a Django app that displays the current date and time along with the dateand time four hours ahead and four hours before as an offset of the current date andtime on the server, you can follow these steps:

Step 1: Update the View to Include Offsets:

- create a app by the name pg2: python manage.py startapp pg2
- You'll need to import the **timedelta** class from the **datetime** module to handle the offsets. Here's an example implementation:
- In pg2/views.py

```
from django.shortcuts import render
from django.http import HttpResponse
from datetime import datetime, timedelta

def datetime_with_offsets(request):
    now = datetime.now()
    offset_hours = 4

    # Calculate dates with offsets
    four_hours_ahead = now + timedelta(hours=offset_hours)
    four_hours_before = now - timedelta(hours=offset_hours)

    html = f"<html><body><h1>Current Date and Time with
Offsets:</h1>" \
    f"<p>Current: {now}</p>" \
    f"<p>Four Hours Ahead: {four_hours_ahead}</p>" \
    f"<p>Four Hours Before: {four_hours_before}</p></body></html>"
    return HttpResponse(html)
```

Step 2: Update URL Configuration:

- Update the URL configuration (**urls.py**) for your app to include the new view.
- You can create a new URL pattern to map the view to a specific URL. Here's an example:

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

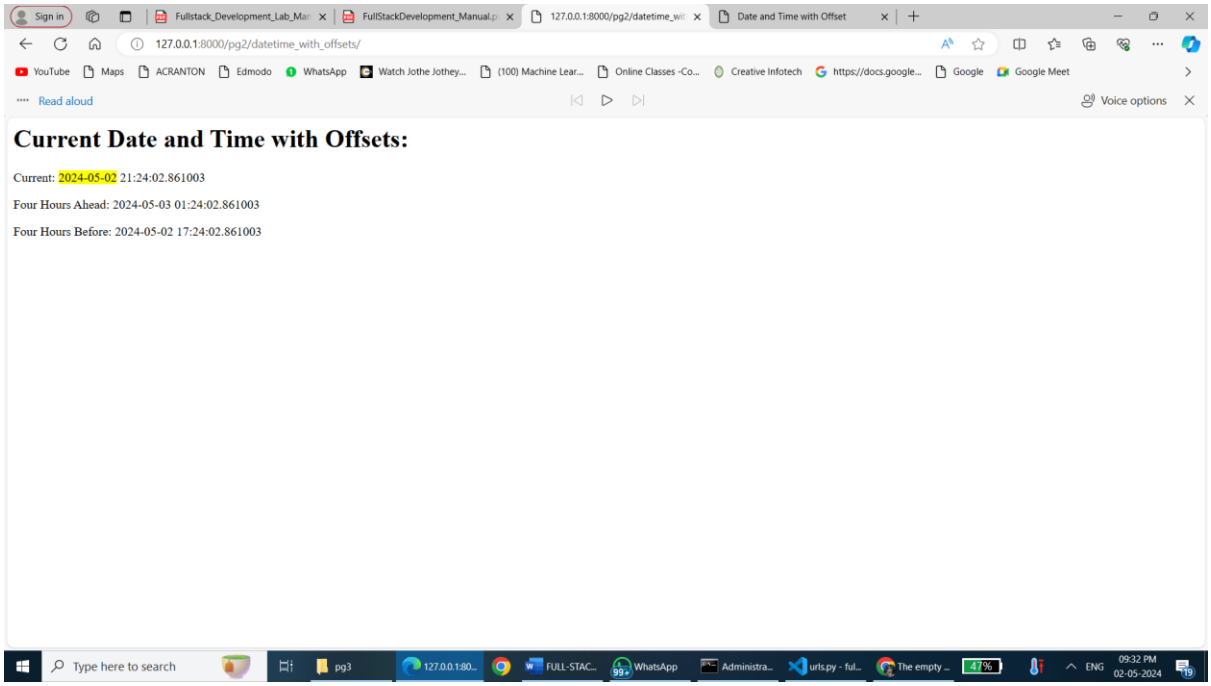
```
from . import views

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

Step 3: Test the App:

- Run the Django development server using the command:

  python manage.py runserver in your project directory.

- Then, navigate to **http://127.0.0.1:8000/pg2/datetime_with_offsets/** in your web browser to see the current date and time with the specified offsets displayed on the page.



OR

Step 1: Define the View:

- Create a view in your Django app **"pg3"** that calculates the current date and time, adds and subtracts four hours, and then renders a template with the results.

Step 2: Create the Template:

- Create a template folder inside pg3 app and subfolder named pg3 inside template
- Create a offset_date.html file inside pg3 subfloder to display the current date and time along with the calculated offsets.
- In pg3 app-views.py

```
from django.shortcuts import render

from datetime import datetime, timedelta

def offset_time(request):
 current_time = datetime.now()
 offset_time_forward = current_time + timedelta(hours=4)
 offset_time_backward = current_time - timedelta(hours=4)

 return render(request, 'pg3/offset_date.html', {
 'current_time': current_time,
           'offset_time_forward': offset_time_forward,
           'offset_time_backward': offset_time_backward,
    })
```

Step 3:define urls

In app pg3/urls.py

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

urlpatterns = [
    path('offset_date/', views.offset_time, name='offset_time'),
]
```

Step 4: Define html

In **pg3/templates/pg3/offset_date.html** file

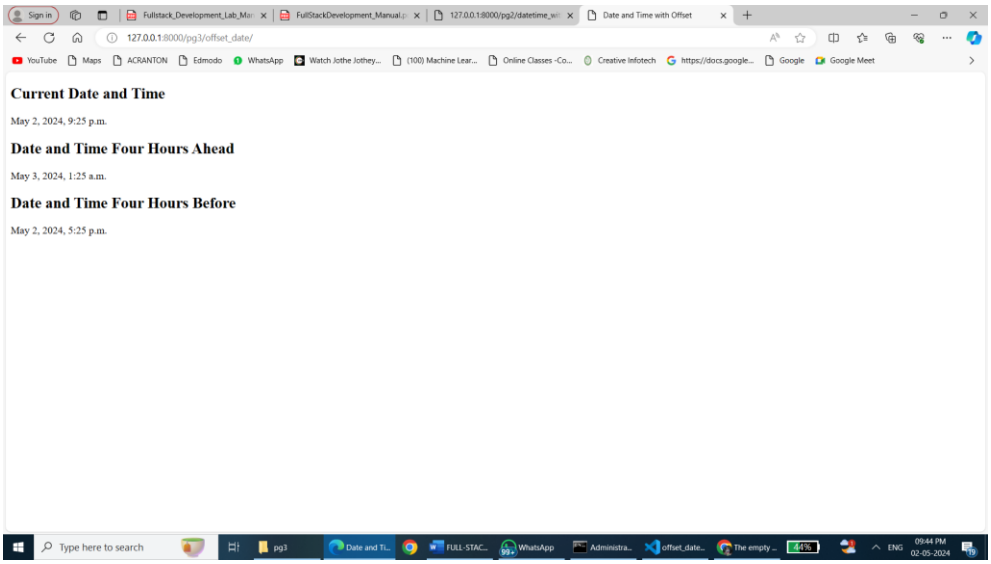
```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
scale=1.0">
    <title>Date and Time with Offset</title>
</head>
<body>
    <h2>Current Date and Time</h2>
    <p>{{ current_time }}</p>

    <h2>Date and Time Four Hours Ahead</h2>
    <p>{{ offset_time_forward }}</p>

    <h2>Date and Time Four Hours Before</h2>
    <p>{{ offset_time_backward }}</p>
</body>
</html>
```

Step5: Test the App:

- Run the Django development server using the command:

  python manage.py runserver in your project directory.

- Then, navigate to **http://127.0.01:8000/pg3/offset_date/** in your web browser to see the current date and time with the specified offsets displayed on the page.



(MODULE 2)

Laboratory Component - 2:

1. Develop a simple Django app that displays an unordered list of fruits and ordered list of selected students for an event
2. Develop a layout.html with a suitable header (containing navigation menu) and footer with copyright and developer information. Inherit this layout.html and create 3 additional pages: contact us, About Us and Home page of any website.
3. Develop a Django app that performs student registration to a course. It should also display listof students registered for any selected course. Create students and course as models with enrolment as ManyToMany field.

2.1 Develop a simple Django app that displays an unordered list of fruits and ordered listof selected students for an event

Solution:

To create a Django app that displays an unordered list of fruits and an ordered list of selected students for an event, follow these steps:

Step 1:

- **Set Up Django Project and App**. Then, create a new Django project and navigate into the project directory:

```
(env1) D:\fullstack-lab-prg>django-admin startproject lab2
```

- Next, create a new Django app (in VS Code) within the project:

```
PS D:\fullstack-lab-prg\>cd lab2
PSD:\fullstack-lab-prg\lab2>python manage.py startapp fruits_and_students
```

Step 2:

- **Define views:** Open the **fruits_and_students/views.py** file in your editor and define two models: Fruit and Student.
- **In lab2/fruits_and_students/views.py**

```
from django.shortcuts import render
def fruits_and_students(request):
    #print(request.build_absolute_uri())
    fruits = ['Apple', 'Banana', 'Orange', 'Grapes']
    students = ['Alice', 'Bob', 'Charlie', 'David']
    return render(request, 'fruits_and_students/
fruits_and_students.html', {'fruits': fruits, 'students':students})
```

Step 3:

Define urls: create a urls.py file in the app and open **lab2/fruits_and_students/urls.py** file.

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

urlpatterns = [
path('', views.fruits_and_students, name='fruits_and_students'),
]
```

Step 4:

The lab2 folder also contains a urls.py file, which is where URL routing is actually handled. Keep in mind the lab2/urls.py will be used to handle all of the laboratory component apps' that will be built. Just add the path url routing line of code to the urlpatterns list every time in the already existing code.

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

urlpatterns = [

path("admin/", admin.site.urls),
path('fruits_and_students/', include('fruits_and_students.urls')),

]
```

Step 5:

Run Migrations: Apply the migrations to create the database tables for your models:

```
python manage.py migrate
```

Step 6:

In the lab2/settings.py file, locate the INSTALLED_APPS list and add the following entry, which makes sure the project knows about the app so it can handle templating: **'fruits_and_students',**

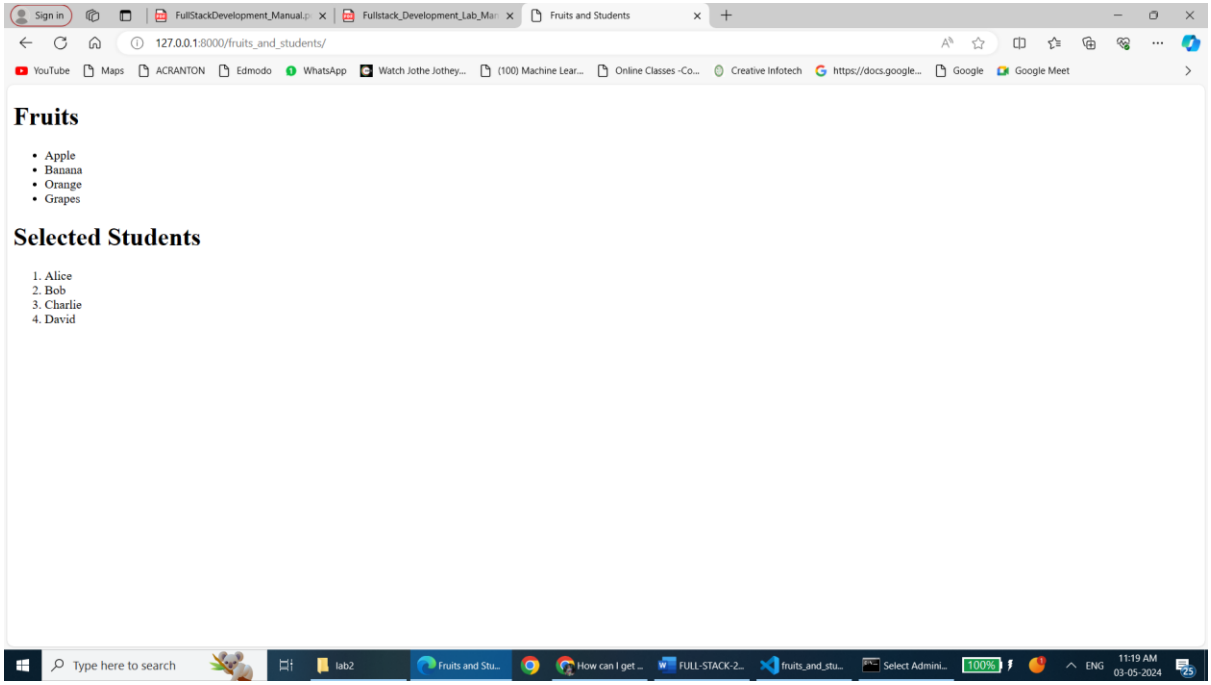
Step7:

- Inside the fruits_and_students folder, create a folder named **templates**, and then another subfolder named **fruits_and_students** to match the app name (this two-tiered folder structure is typical Django convention).
- In the **templates/fruits_and_students** folder, create a file named **fruits_and_students.html** with the contents below. (**fruits_and_students/templates/fruits_and_students/ fruits_and_students.html)**

```
<!DOCTYPE html>
<html lang="en">

<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-
scale=1.0">
<title>Fruits and Students</title>
</head>

<body>
<h1>Fruits</h1>
<ul>
{% for fruit in fruits %}
<li>{{ fruit }}</li>
{% endfor %}
</ul>

<h1>Selected Students</h1>
<ol>
{% for student in students %}
<li>{{ student }}</li>
{% endfor %}
</ol>

</body>

</html>
```

Step8: test the App

- In the VS Code Terminal, again with the virtual environment activated, run the development server with python manage.py runserver and open a browser to **http://127.0.0.1:8000/**
- In the url box of the browse, navigate to **http://127.0.0.1:8000/fruits_and_students** to view the output.



2.2 Develop a layout.html with a suitable header (containing navigation menu) and footer with copyright and developer information. Inherit this layout.html and create 3 additional pages: contact us, About Us and Home page of any website.

Solution:

To create a layout template layout.html with a header containing a navigation menu and a footer with copyright and developer information, and then inherit this layout for creating additional pages, follow these steps:

Step 1: Create a Django app

In the VS Code Terminal with your virtual environment activated, run the administrative utility's startapp command in your lab2 folder (where manage.py resides):

```
python manage.py startapp website_pages
```

Step 2: Define Views

Modify **website_page/views.py** to match the following code.

```
from django.shortcuts import render

def home(request):
    return render(request, 'website_pages/home.html')

def about_us(request):
    return render(request, 'website_pages/about_us.html')

def contact_us(request):
    return render(request, 'website_pages/contact_us.html')
```

Step 3: Define urls

Create a file, website_page/urls.py, with the contents below. The urls.py file is where you specify patterns to route different URLs to their appropriate views.

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

urlpatterns = [
path('home/', views.home, name='home'),
path('about_us/', views.about_us, name='about_us'),
path('contact_us/', views.contact_us, name='contact_us'),
]
```

The lab2 folder also contains a urls.py file, which is where URL routing is actually handled. #urls.py (lab2/urls.py)

Step 4: Setup the setting.py file

In the **lab2/settings.py** file, locate the INSTALLED_APPS list and add the following entry, which makes sure the project knows about the app so it can handle templating:

'website_pages',

Step 5: Define html files

➔ Inside the **website_page** folder, create a folder named **templates.** Inside the **templates** folder, create a file named **layout.html.**

#templates/layout.html (The CSS is optional)

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>{% block title %}My Website{% endblock %}</title>
<style>
body {
font-family: Arial, sans-serif;
margin: 0;
padding: 0;
}
header {
background-color: #333;
color: #fff;
padding: 10px;
}
nav ul {
list-style-type: none;
padding: 0;
}
nav ul li {
display: inline;
margin-right: 20px;
}
nav ul li a {
text-decoration: none;
color: #fff;
}
main {
padding: 20px;
}
```

```
footer {
background-color: #333;
color: #fff;
text-align: center;
padding: 10px;
position: fixed;
bottom: 0;
width: 100%;
}
</style>
</head>
<body>
<header>
<nav>
<ul>
<li><a href="{% url 'home' %}">Home</a></li>
<li><a href="{% url 'about_us' %}">About Us</a></li>
<li><a href="{% url 'contact_us' %}">Contact Us</a></li>
</ul>
</nav>
</header>
<main>
{% block content %}
{% endblock %}
</main>
<footer>
<p>&copy; 2024 My Website. All rights reserved. </p>
<p>Developed by CIT</p>
</footer>
</body>
</html>
```

➔ Inside the **templates** folder, create another subfolder named **website_page** to match the app name (this two-tiered folder structure is typical Django convention).

➔ In the **templates/website_page** folder, create files named **home.html, about_us.html, and contact_us.html** with the contents below.

Lab2/website_page/templates/website_page/home.html

```
{% extends 'layout.html' %}
{% block title %}Home{% endblock %}
{% block content %}
<h1>Welcome to Sapthagiri college of engineering</h1>
<p>Empowering students with a blend of knowledge and innovation.</p>
<p>Nestled in the bustling city of Bengaluru, our campus is a hub of academic
excellence and
cutting-edge research.</p>
<h2>Discover Your Potential</h2>
<ul>
<li><strong>Undergraduate Programs:</strong> Dive into our diverse range of
engineering
courses designed to fuel your passion and drive innovation.</li>
<li><strong>Postgraduate Programs:</strong> Advance your expertise with our
specialized
master's programs and embrace leadership in technology.</li>
</ul>
<p>Join our vibrant community where ideas flourish and inventions come to life
in our state-of-the
art labs and research centers.</p>
<p>Benefit from our strong industry ties and placement programs that open
doors to exciting career
opportunities.</p>
{% endblock %}
```

Lab2/website_page/templates/website_page/about_us.html

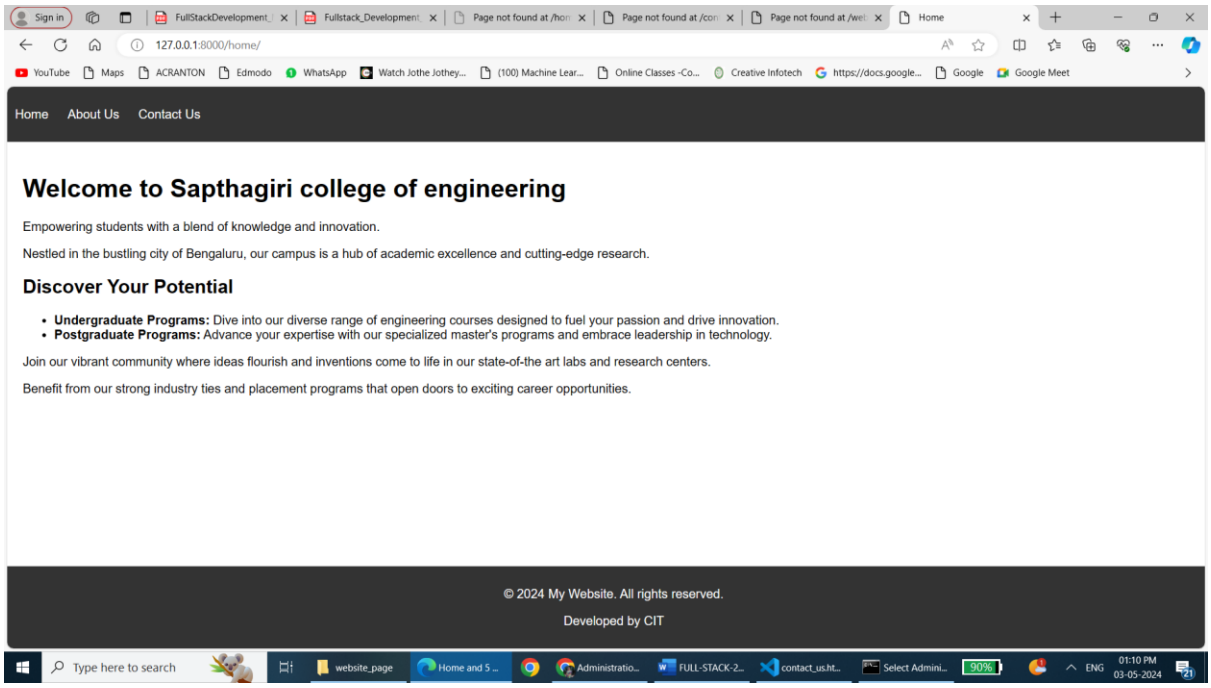
```
{% extends 'layout.html' %}
{% block title %}About Us{% endblock %}
{% block content %}
<h1>Our Legacy</h1>
<p>Founded on the principles of quality education and societal contribution,
we've been at the
forefront of technological education for over four decades.</p>
<h1>Vision and Mission</h1>
<p>Our vision is to be a beacon of knowledge that lights the way for aspiring
minds, and our mission
is to nurture innovative thinkers who will shape the future of technology.</p>
<h1>Campus Life</h1>
<p>Experience a dynamic campus life enriched with cultural activities,
technical clubs, and
community service initiatives that foster holistic development.</p>
{% endblock %}
```

Lab2/website_page/templates/website_page/contact_us.html

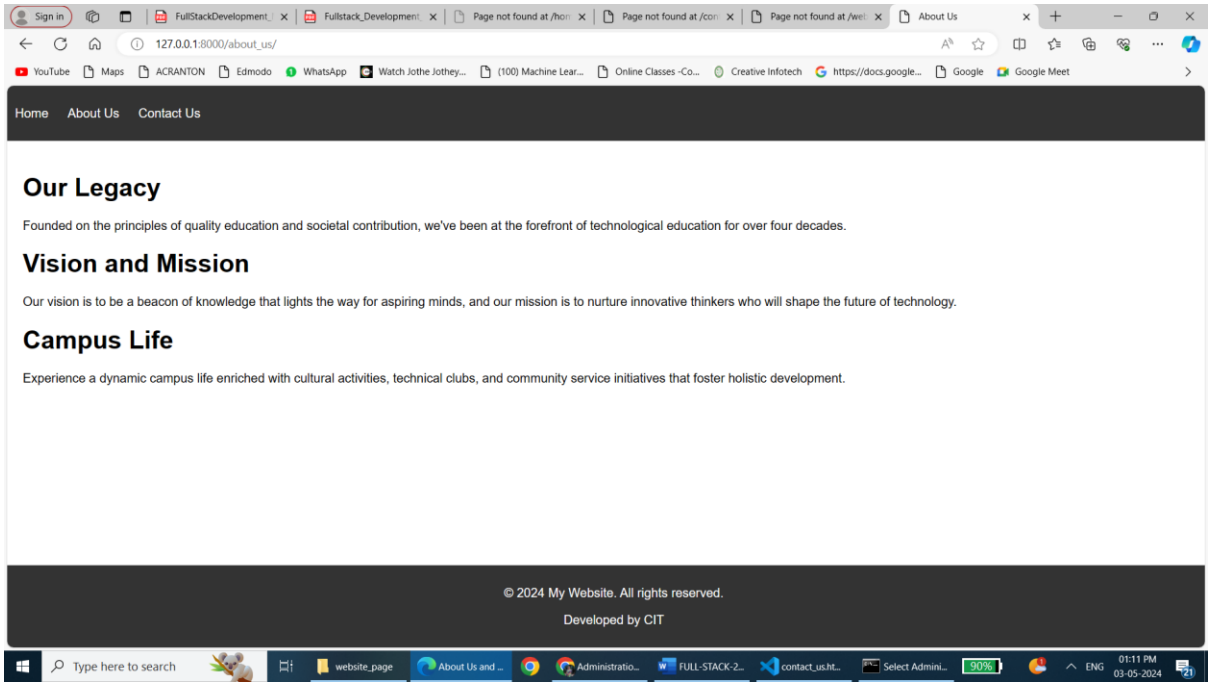
```
{% extends 'layout.html' %}
{% block title %}Contact Us{% endblock %}
{% block content %}
<h1>Get in Touch</h1>
<p>For admissions and inquiries, reach out to us at:</p>
<ul>
<li><strong>Email:</strong> admissions@cambridge.edu.in</li>
<li><strong>Phone:</strong> +91-9731998888</li>
</ul>
<h1>Visit Our Campus</h1>
<p>Cambridge Institute of Technology - Main Campus</p>
<p>KR Puram, Bengaluru - 560036</p>
<p>We welcome you to be a part of our thriving community that's dedicated to
creating a better
tomorrow through technology and innovation.</p>
{% endblock %}
```

Step 6: Test the App

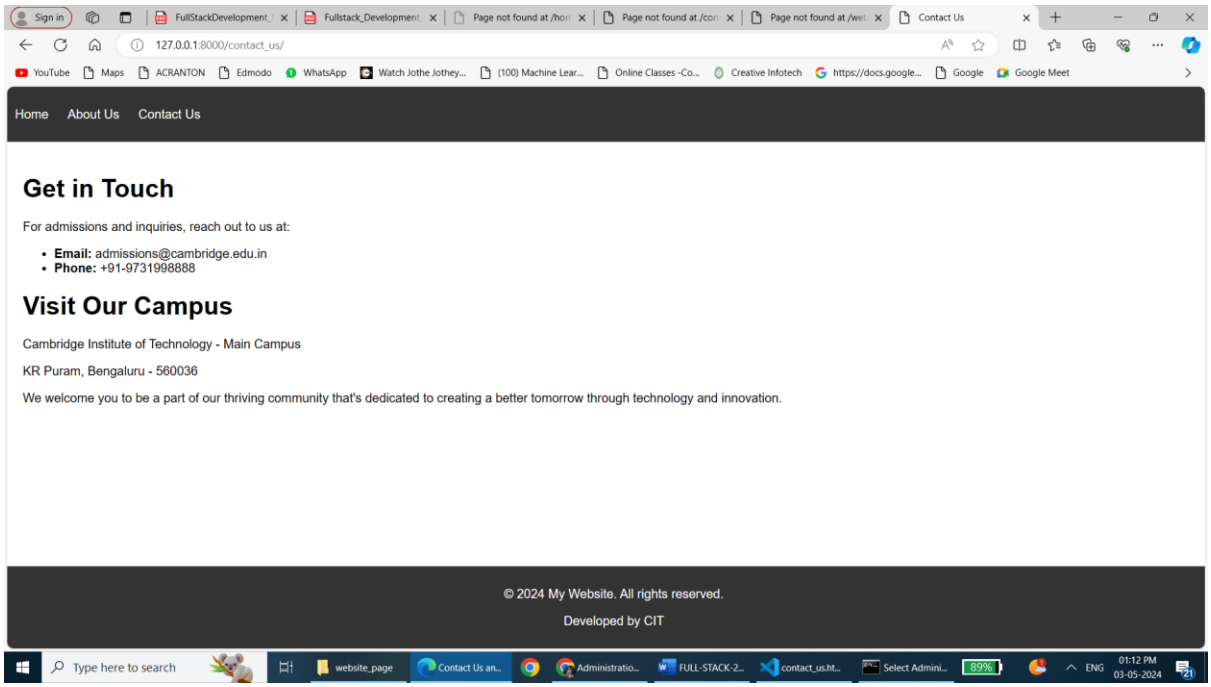
In the VS Code Terminal, again with the virtual environment activated, run the development server with python manage.py runserver and open a browser to http://127.0.0.1:8000/



http://127.0.0.1:8000/home/



http://127.0.0.1:8000/about_us/



http://127.0.0.1:8000/contact_us/

2.3 Develop a Django app that performs student registration to a course. It should also display list of students registered for any selected course. Create students and course as models with enrolment as ManyToMany field.

Solution:

To create a Django app for student registration to a course and display a list of students registered for a selected course, you can follow these steps:

Step 1: Creation of App

In the VS Code Terminal with your virtual environment activated, run the administrative utility's startapp command in your lab2 folder (where manage.py resides):

```
PS D:\fullstack-lab-prg\lab2> python manage.py startapp course_registration
```

Step 2: Define Views

Modify **course_registration/views.py** to match the following code.

```python
from .forms import StudentForm, CourseForm
from .models import Student, Course
from django.shortcuts import render, redirect, get_object_or_404

def add_student(request):
    if request.method == 'POST':
        form = StudentForm(request.POST)
        if form.is_valid():
            form.save()
            # Redirect to a view that lists all students
            return redirect('student_list')
    else:
        form = StudentForm()
    return render(request, 'course_registration/add_student.html', {'form': form})

def add_course(request):
    if request.method == 'POST':
        form = CourseForm(request.POST)
        if form.is_valid():
            form.save()
            return redirect('course_registration')
    else:
        form = CourseForm()
    return render(request, 'course_registration/add_course.html', {'form': form})
```

```
def register_student(request):
    if request.method == 'POST':
        student_name = request.POST.get('student_name')
        course_id = request.POST.get('course_id')
        # Validate that both student_name and course_id are provided
        if not student_name or not course_id:
            return render(request, 'course_registration/register_student.html',
{'courses': Course.objects.all(),'error_message': 'Please provide both student name
and select a course.'})
        try:
            # Retrieve the course based on course_id or return 404 if not found
            course = get_object_or_404(Course, pk=course_id)
            # Check if the student already exists in the database
            student = Student.objects.filter(name=student_name).first()
            if not student:
                # If the student does not exist, return an error message
                return render(request, 'course_registration/register_student.html',
{'courses': Course.objects.all(), 'error_message': 'Student does not exist in the
database.'})
            # Add the student to the course
            course.students.add(student)
            return redirect('course_registration')
        except Course.DoesNotExist:
            return render(request, 'course_registration/register_student.html',
{'courses': Course.objects.all(), 'error_message': 'Invalid course ID. Please select
a valid course.'})
        # If not a POST request, render the registration form with all courses
    return render(request, 'course_registration/register_student.html', {'courses':
Course.objects.all()})

def course_registration(request):
    courses = Course.objects.all()
    return render(request, 'course_registration/course_registration.html.html',
{'courses': courses})

def students_list(request, course_id):
    # Retrieve the course based on course_id or return 404 if not found
    course = get_object_or_404(Course, course_id=course_id)
    # Retrieve the students associated with the course
    students = course.students.all()
    return render(request, 'course_registration/students_list.html', {'course':
course, 'students': students})
```

Step 3: Define urls

1. Create a file, **course_registration/urls.py**, with the contents below. The urls.py file is where you specify patterns to route different URLs to their appropriate views.

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

urlpatterns = [
    path('add_student/', views.add_student, name='add_student'),
    path('add_course/', views.add_course, name='add_course'),
    path('register/', views.register_student, name='register_student'),
    path('courses/', views.course_registration, name='course_registration'),
    path('students_list/<int:course_id>/', views.students_list,
name='students_list'),
]
```

2. The lab2 folder also contains a urls.py file, which is where URL routing is actually handled. #urls.py **(lab2/urls.py)**

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

urlpatterns = [

path("admin/", admin.site.urls),
path('fruits_and_students/', include('fruits_and_students.urls')),
path('', include('website_page.urls')),
path('registration/', include('course_registration.urls')),

]
```

Step 4: Define models

Modify course_registration/models.py to match the following code.

```
from django.db import models

class Student(models.Model):
    name = models.CharField(max_length=100,unique=True)
    date_of_birth = models.DateField( default='1900-01-01', blank=False,
null=False)  # Set a default date
    email = models.EmailField( default='example@example.com', blank=False,
null=False)  # Set a default email

def __str__(self):
    return self.name

class Course(models.Model):
    name = models.CharField(max_length=100,unique=True)
    students = models.ManyToManyField(Student, related_name='courses')
    course_id = models.IntegerField(default=0,unique=True)

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

Step 5: Define forms

Modify **course_registration/forms.py** to match the following code.

```
from .models import Student
from django import forms
from .models import Course

class CourseForm(forms.ModelForm):
    class Meta:
        model = Course
        fields = ['name', 'course_id']

class StudentForm(forms.ModelForm):
    class Meta:
        model = Student
        fields = ['name', 'date_of_birth', 'email']
```

Step 6: Define settings

In the **lab2/settings.py** file, locate the INSTALLED_APPS list and add the following entry, which makes sure the project knows about the app so it can handle templating:

'course_registration',

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'fruits_and_students',
    'website_page',
    'course_registration'


]
```

Step 7: Define templates

Create a templates folder inside **course_registration** folder, create inside templates folder, a subfolder named **course_registration** to match the app name (this two-tiered folder structure is typical Django convention). In the **templates/course_registration** folder, create files named **add_student.html, add_course.html, register_student.html, course_registration.html, and students_list.html** with the contents below.

1. **#templates/course_registration/add_student.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Add Student</title>
</head>
<body>
    <h1>Add Student</h1>
    <form method="post">
        {% csrf_token %}
        {{ form.as_p }}
    <button type="submit">Submit</button>
    </form>
</body>
</html>
```

2. #templates/course_registration/add_course.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Add Course</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f7f7f7;
            margin: 0;
            padding: 20px;
        }
        h1 {
            color: #333;
        }
        form {
            background-color: #fff;
            padding: 20px;
            border-radius: 5px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }
        label {
            display: block;
            margin-bottom: 10px;
            font-weight: bold;
        }
        input[type="text"], input[type="number"] {
            width: 100%;
            padding: 10px;
            margin-bottom: 20px;
            border: 1px solid #ccc;
            border-radius: 5px;
            box-sizing: border-box;
            font-size: 16px;
        }
        button[type="submit"] {
            background-color: #007bff;
            color: #fff;
            padding: 10px 20px;
            border: none;
            border-radius: 5px;
            cursor: pointer;
            font-size: 16px;
        }
    button[type="submit"]:hover {
    background-color: #0056b3;
```

```
    }
    </style>
</head>
<body>
    <h1>Add Course</h1>
    <form method="POST">
        {% csrf_token %}
        <label for="course_name">Course Name:</label>
        <input type="text" id="course_name" name="name" required>
        <label for="course_id">Course ID:</label>
        <input type="number" id="course_id" name="course_id" required>
        <button type="submit">Add Course</button>
    </form>
</body>
</html>
```

3. #templates/course_registration/register_student.html

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="UTF-8">
<meta name="viewport" content="width=device-width, initial-scale=1.0">
<title>Register Student</title>
<style>
body {
    font-family: Arial, sans-serif;
    background-color: #f8f9fa;
    margin: 0;
    padding: 0;
}
.container {
    max-width: 600px;
    margin: 50px auto;
    background-color: #fff;
    padding: 20px;
    border-radius: 5px;
    box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
    }
.form-group {
    margin-bottom: 20px;
    }
.form-control {
    width: 100%;
    padding: 10px;
    border: 1px solid #ccc;
```

```
    border-radius: 5px;
    box-sizing: border-box;
    }
.btn {
    padding: 10px 20px;
    background-color: #007bff;
    color: #fff;
    border: none;
    border-radius: 5px;
    cursor: pointer;
    }
.btn-primary {
    background-color: #007bff;
    }
    </style>
</head>
<body>
    <div class="container">
        <h1>Register Student to Course</h1>
        <form method="POST" class="form">
            {% csrf_token %}
            <div class="form-group">
                <label for="student_name">Student Name:</label>
                <input type="text" id="student_name" name="student_name"
class="form-control" required>
            </div>
            <div class="form-group">
                <label for="course_id">Select Course:</label>
                <select name="course_id" id="course_id" class="form-
control">
                    {% for course in courses %}
                    <option value="{{ course.id }}">{{ course.name
}}</option>
                    {% endfor %}
                </select>
            </div>
            <button type="submit" class="btn btn-primary">Register</button>
            </form>
        </div>
    </body>
</html>
```

4. **#templates/course_registration/course_registration.html**

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Course Registration</title>
    <style>
        body {
            font-family: Arial, sans-serif;
             background-color: #f8f9fa;
            margin: 0;
            padding: 0;
        }
        .container {
            max-width: 600px;
            margin: 50px auto;
            background-color: #fff;
            padding: 20px;
            border-radius: 5px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }
        .list-group {
            list-style-type: none;
            padding: 0;
        }
        .list-group-item {
            margin-bottom: 10px;
        }
        .list-group-item a {
            text-decoration: none;
            color: #333;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Course Registration</h1>
        <ul class="list-group">
            {% for course in courses %}
            <li class="list-group-item">
            <a href="{% url 'students_list' course.course_id %}">{{
course.name }} (ID: {{course.course_id }})</a>
            </li>
            {% endfor %}
        </ul>
    </div>
```

```
</body>
</html>
```

5. #templates/course_registration/students_list.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Students List</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            background-color: #f8f9fa;
            margin: 0;
            padding: 0;
        }
        .container {
            max-width: 600px;
            margin: 50px auto;
            background-color: #fff;
            padding: 20px;
            border-radius: 5px;
            box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
        }
        .list-group {
            list-style-type: none;
            padding: 0;
        }
        .list-group-item {
            margin-bottom: 10px;
        }
    </style>
</head>
<body>
    <div class="container">
        <h1>Students Registered for {{course.name}}</h1>
        <ul class="list-group">
            {% for student in students %}
            <li class="list-group-item">{{student.name}}</li>
            {% empty %}
            <li class="list-group-item">No students registered for this
course.</li>
            {% endfor %}
        </ul>
```

```
    </div>
</body>
</html>
```

Step 8: save all the files

Step 9: Database Migration:

In the VS Code Terminal, again with the virtual environment activated, run the below commands to migrate changes.

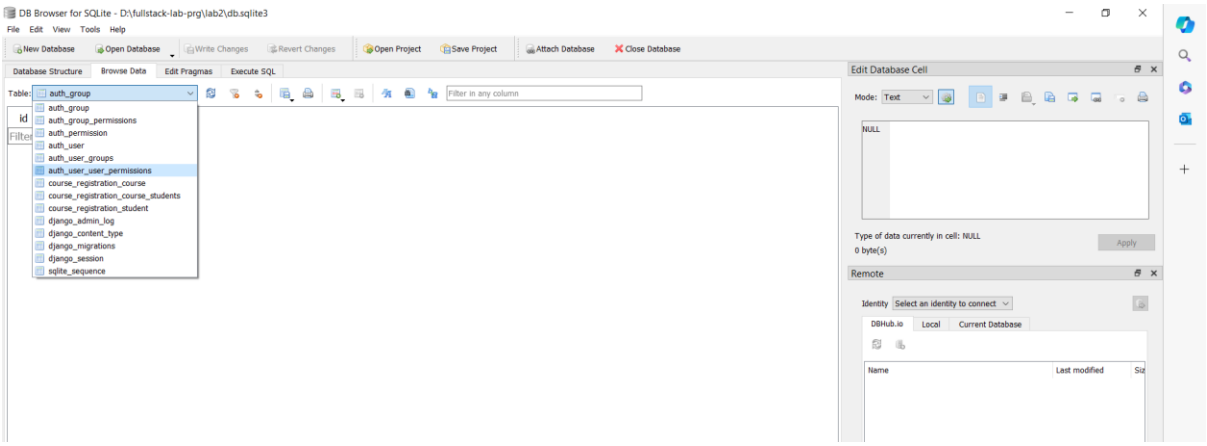
python manage.py makemigrations

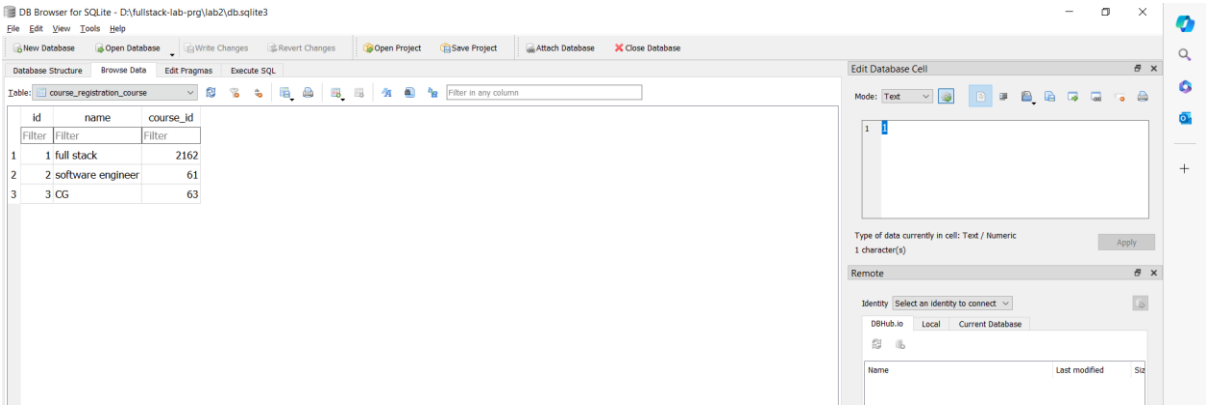
python manage.py migrate

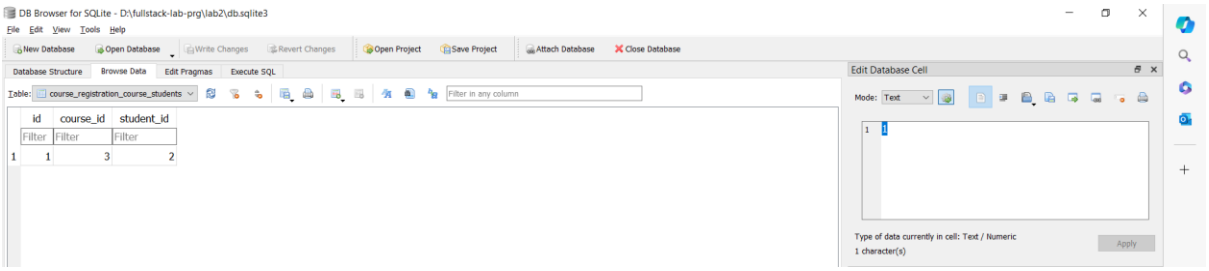
Step 10: In order to view the database and its tables, you can use SQLite DB Browser. https://sqlitebrowser.org/dl/

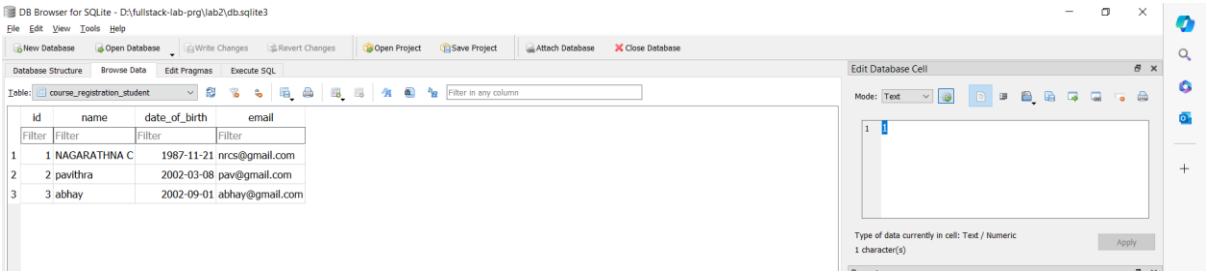
ManyToMany field application can be checked through the fact that multiple students can be enrolled to multiple courses.

Go to db browser- file-open database-select your database(lab2/course_registration/db)





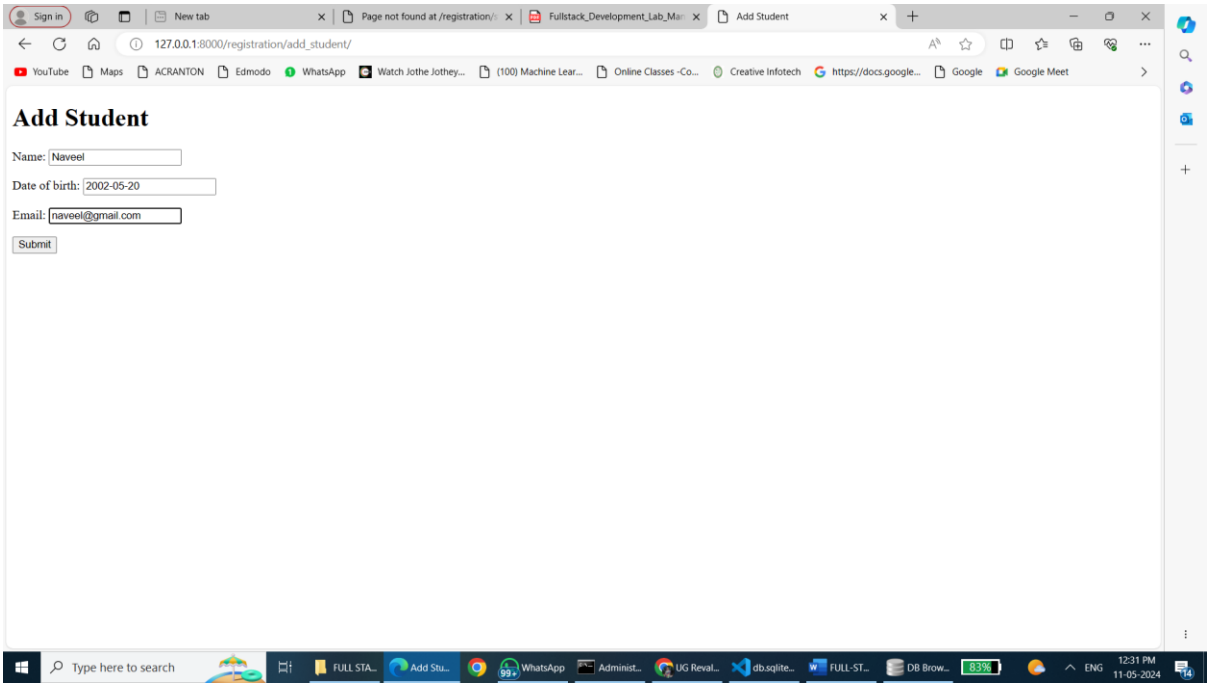




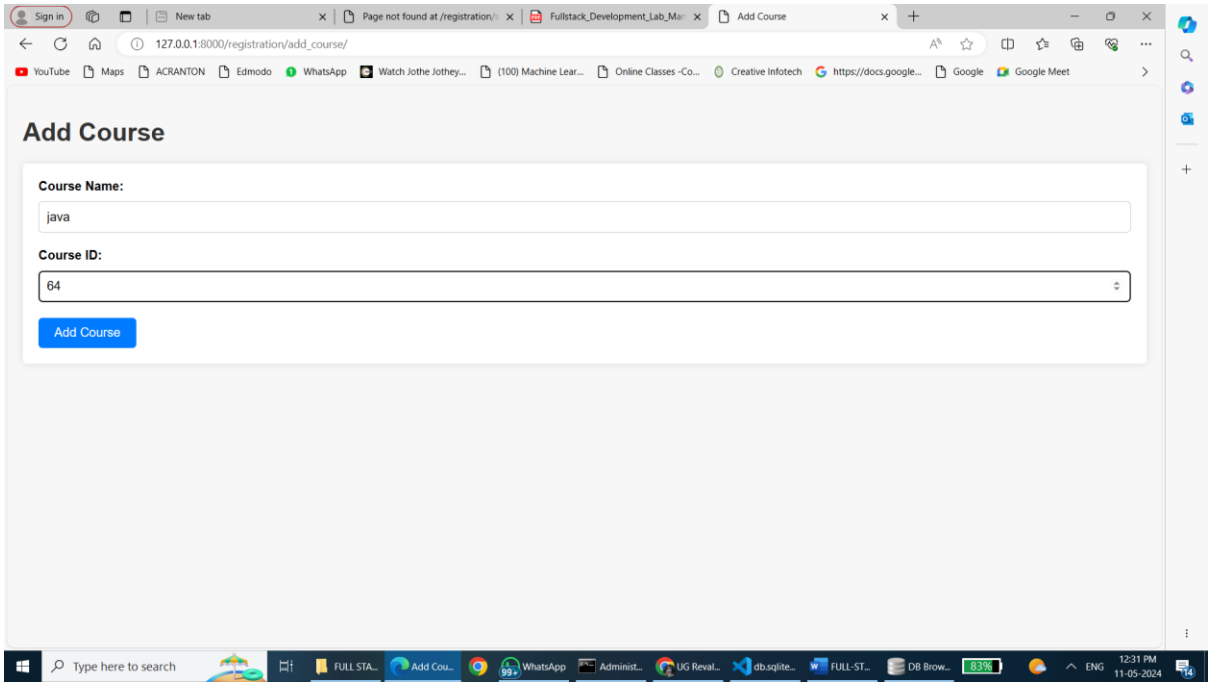
Step 11: run the server

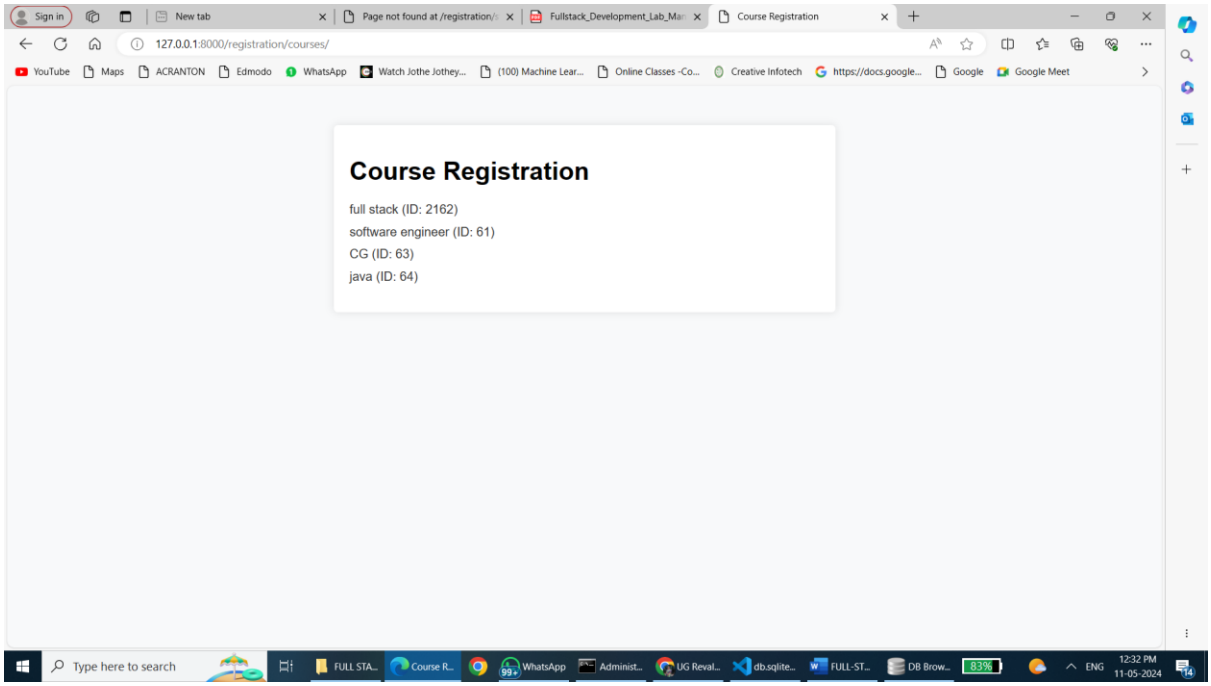
In the url box of the browse, navigate to

http://127.0.0.1:8000/registration/add_student

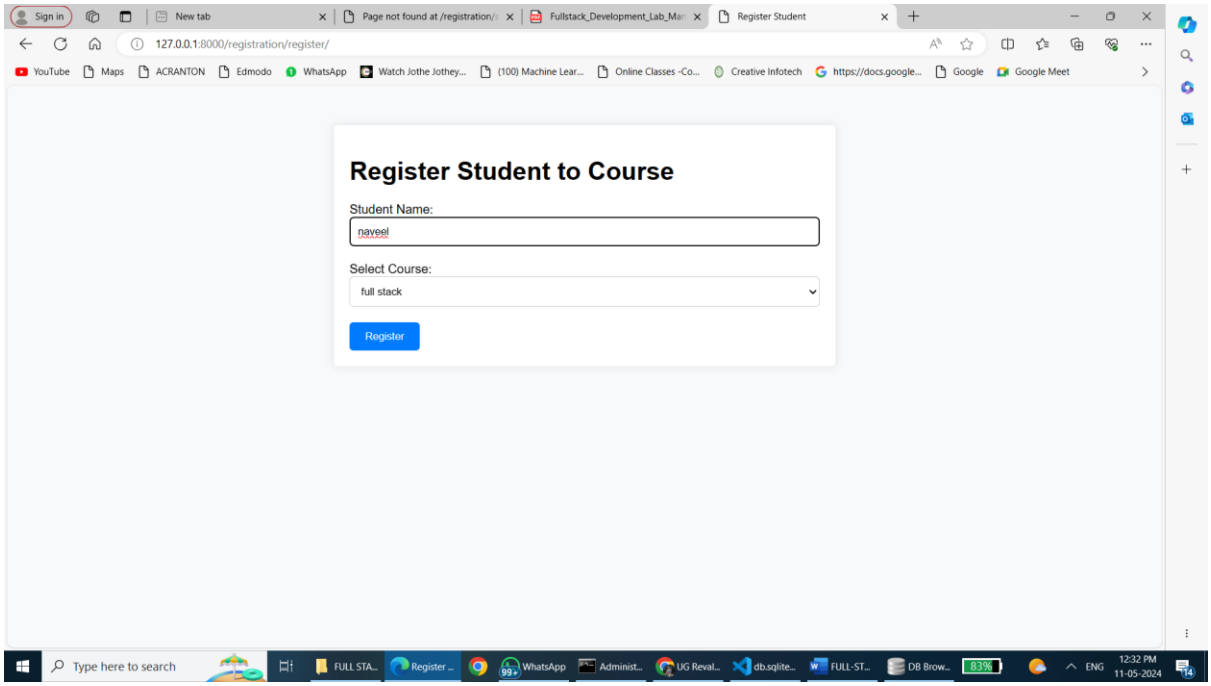


http://127.0.0.1:8000/registration/add_course

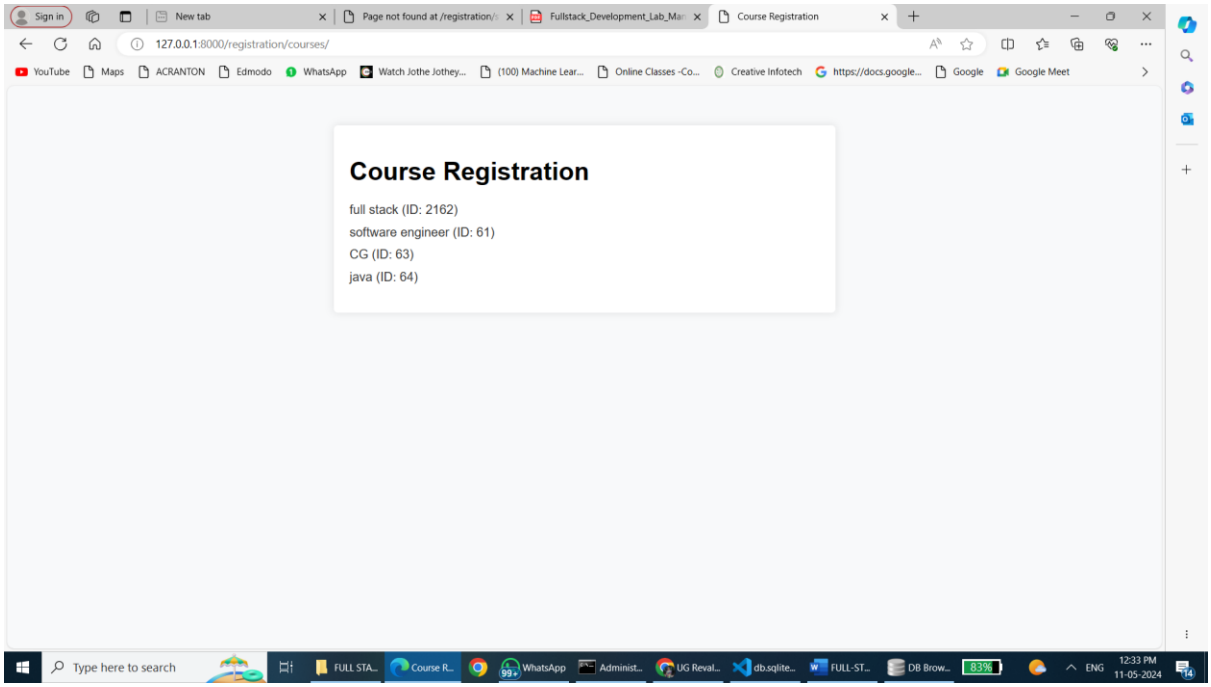




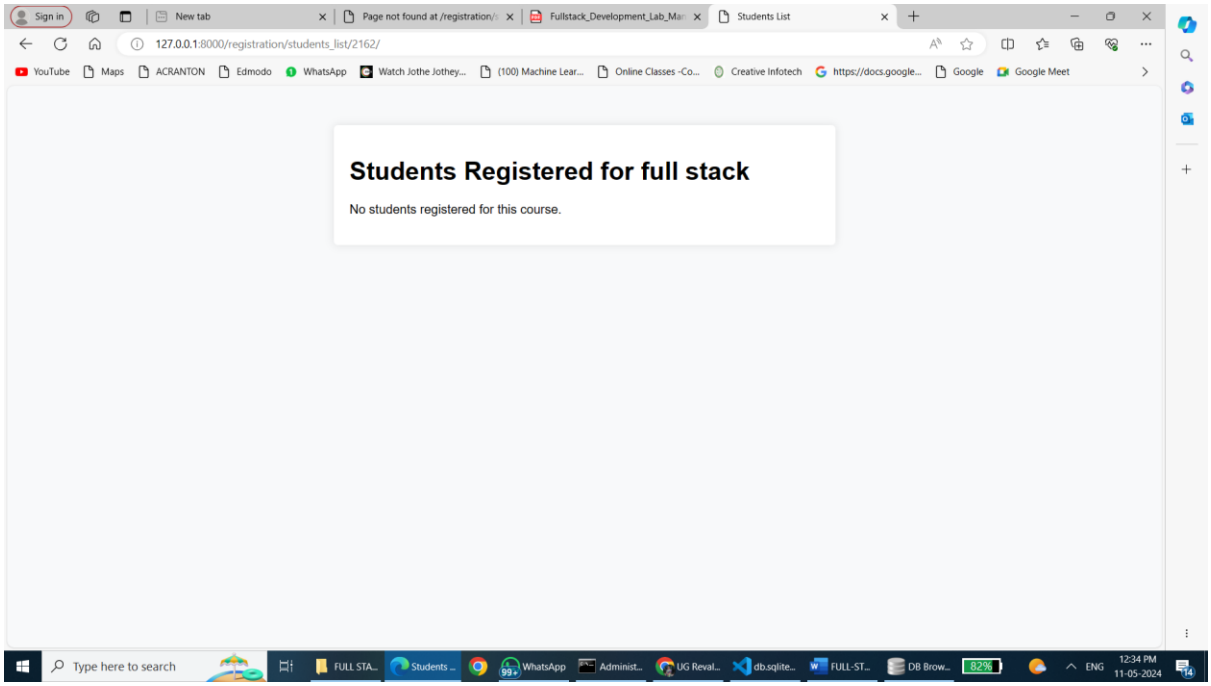
http://127.0.0.1:8000/registration/register



http://127.0.0.1:8000/registration/courses



http://127.0.0.1:8000/registration/students_list/



MODULE 3

1. For student and course models created in Lab experiment for Module2, register admin interfaces, perform migrations and illustrate data entry through admin forms.

2. Develop a Model form for student that contains his topic chosen for project, languages used and duration with a model called project.

3.1 For student and course models created in Lab experiment for Module2, register admin interfaces, perform migrations and illustrate data entry through admin forms.