"Full Stack Development Using Django"

Lab Manual

Department of

Information Science and Engineering,

City Engineering College, Bangalore

Module-1

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

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:

- Open your terminal or command prompt.
- Navigate to the directory where you want to create your Django project.
- Run the following command to create a virtual environment named "venv" (you can choose any name):
python3 -m venv venv
Activate the virtual environment:
- On Windows:
venv\Scripts\activate
Step 2: Install Django:
- Once the virtual environment is activated, install Django using pip:
pip install Django
Step 3: Create a Django Project:
- After installing Django, create a new Django project using the following command:
django-admin startproject myproject
Replace "myproject" with your desired project name.
Step 4: Navigate to the Project Directory:
- Change to the newly created project directory:
cd myproject
Step 5: Create a Django App:
- Inside the project directory, create a Django app using the following command:
python manage.py startapp myapp
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):
- If your app has models and you need to create database tables, run the following commands:
python manage.py makemigrations
python manage.py migrate
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.

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 myapp.

Step 2: Define a View: Open the **views.py** file inside your **myapp** directory and define a view to display the current date and time. Here's an example:

```
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, name='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

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

Step 5: Run the Development Server: Start the Django development server to test your 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 and go to **http://127.0.0.1:8000/myapp/current_datetime/** to see the current date and time displayed on the page.

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 date and time four hours ahead and four hours before as an offset of the current date and time on the server, you can follow these steps:

Step 1: Update the View to Include Offsets: Modify the view in your Django app (**views.py**) to calculate the current date and time along with the offsets. You'll need to import the **timedelta** class from the **datetime** module to handle the offsets. Here's an example implementation:

```
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/myapp/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 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 to display the current date and time along with the calculated offsets.

Step 3: In event_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, 'event_app/offset_time.html', {
                        'current_time': current_time,
                        'offset_time_forward': offset_time_forward,
                        'offset_time_backward': offset_time_backward,
   })
```

In **event_app/templates/event_app/offset_time.html**

```
<!DOCTYPE html>
<html>
<head>
   <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>
```

In **event_app/urls.py**, add a URL pattern for the **offset_time** view:

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

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

Module-2

1. Develop a simple Django app that displays an unordered list of fruits and ordered list of 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:

```
django-admin startproject event_manager
cd event_manager
```

Next, create a new Django app within the project:

```
python manage.py startapp events
```

Step 2:Define Models: Open the **events/models.py** file in your editor and define two models: **Fruit** and **Student**.

```
from django.db import models

class Fruit(models.Model):
    name = models.CharField(max_length=100)

    def__str_(self):
        return self.name

class Student(models.Model):
    name = models.CharField(max_length=100)
    event = models.CharField(max_length=100) # Assuming the event name is a string
    selected = models.BooleanField(default=False)

    def__str_(self):
        return self.name
```

Step 3: Register Models in Admin: Open the events/admin.py file and register the models to make them accessible via the Django admin interface.

```
from django.contrib import admin
from .models import Fruit, Student

admin.site.register(Fruit)
admin.site.register(Student)
```

Step 4: Run Migrations: Apply the migrations to create the database tables for your models:

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

Step 5: Create Views and Templates: Create views and templates to display the lists of fruits and students.

In events/views.py, define the view functions:

```
from django.shortcuts import render
from .models import Fruit, Student

def  fruit_list(request):
   fruits = Fruit.objects.all()
   return render(request, 'events/fruit_list.html', {'fruits': fruits})

def student_list(request):
   students = Student.objects.filter(selected=True)
   return render(request, 'events/student_list.html', {'students': students})
```

Create templates in the **events/templates/events/** directory:

- **fruit_list.html**:

```
<!DOCTYPE html>
<html>
<head>
   <title>Fruit List</title>
</head>
<body>
   <h1>Available Fruits:</h1>
   <ul>
      {% for fruit in fruits %}
         <li>{{ fruit.name }}</li>
      {% endfor %}
   </ul>
</body>
</html>
```

student_list.html:

```
<!DOCTYPE html>
<html>
<head>
   <title>Selected Students</title>
</head>
<body>
   <h1>Selected Students for the Event:</h1>
   <ol>
      {% for student in students %}
         <li>{{ student.name }}</li>
      {% endfor %}
   </ol>
</body>
</html>
```

Step 6: Define URLs: Create URL patterns in **events/urls.py** to map the views to URLs.

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

```
from . import views

urlpatterns = [
    path('fruits/', views.fruit_list, name='fruit_list'),
    path('students/', views.student_list, name='student_list'),
]
```

Step 7: Include URLs in Project: Include the app's URLs in the project's main **urls.py** file (**event_manager/urls.py**).

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

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

Step 8: Run the Development Server: Start the Django development server:

python manage.py runserver

Check output in **http://127.0.0.1:8000/events/fruits/** to see the list of fruits and **http://127.0.0.1:8000/events/students/** to see the list of selected students.

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 layout.html Template: Create a new HTML file named **layout.html** in your Django app's templates directory (e.g., **events/templates/events/layout.html**). This will serve as the base layout for your website.

```
<!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>
   <!-- Add your CSS and JS links here -->
</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; {{ year }} My Website. All rights reserved. Developed by {{
developer }}</p>
   </footer>
</body>
</html>
```

Step 2: **Create Additional Pages:** Now, create three additional HTML files that will inherit from **layout.html** and define the content for each page.

- **Home Page (home.html):**

```
{% extends 'events/layout.html' %}
```

```
{% block title %}Home - My Website{% endblock %}

{% block content %}
<h1>Welcome to My Website</h1>
<!-- Add home page content here -->
{% endblock %}
```

About Us Page (about_us.html):

```
{% extends 'events/layout.html' %}
{% block title %}About Us - My Website{% endblock %}

{% block content %}
<h1>About Us</h1>
<!-- Add about us page content here -->
{% endblock %}
```

Contact Us Page (contact_us.html):

```
{% extends 'events/layout.html' %}
{% block title %}Contact Us - My Website{% endblock %}

{% block content %}
<h1>Contact Us</h1>
<!-- Add contact us page content here -->
{% endblock %}
```

Step 3: Define URL Patterns: Define URL patterns in your Django app's **urls.py** file (**events/urls.py**) to map these pages to specific URLs.

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

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

Step 4: Create View Functions: Define view functions in your Django app's **views.py** file (**events/views.py**) to render the respective templates for each page.

```
from django.shortcuts import render

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

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

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

Step 5: Include URLs in Project: Include the app's URLs in the project's main **urls.py** file (**event_manager/urls.py**).

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

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include('events.urls')),  # Assuming 'events' is your app name
]
```

Step 6: Update Settings: Make sure your Django project's settings (**settings.py**) include the **'DIRS'** setting pointing to the templates directory, and define the **year** and **developer** variables for the footer.

```
TEMPLATES = [
    {
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        ...
    },
]

year = datetime.now().year
developer = "Your Name"
```

With these steps, you have created a layout template **layout.html** with a header, footer, and placeholders for content. You can now access the Home, About Us, and Contact Us pages of your website.

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: Then, create a new Django project:

django-admin startproject course_registration

Step 2: Create the models: Create models for **Student** and **Course** in a new Django app called **registration**:

cd course_registration

python manage.py startapp registration

Step 3:

In the **registration/models.py** file, define the models as follows:

```
from django.db import models

class Course(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

    def__str_(self):
        return self.name

class Student(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    courses = models.ManyToManyField(Course, related_name='students')

    def__str_(self):
        return self.name
```

Step 4:

Set up the admin interface: Register the models in the **registration/admin.py** file to manage them via the Django admin interface:

```
from django.contrib import admin
from .models import Course, Student

admin.site.register(Course)
admin.site.register(Student)
```

Step 5:

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

python manage.py makemigrations

python manage.py migrate

Step 6:

Create views and templates: Create views and templates for student registration and course listing. In **registration/views.py**, create view functions for registration and listing:

```
from django.shortcuts import render
from .models import Course

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

def register_student(request, course_id):
   course = Course.objects.get(pk=course_id)
   if request.method == 'POST':
      name = request.POST.get('name')
      email = request.POST.get('email')
      student, created = course.students.get_or_create(name=name, email=email)
      if created:
         message = f'{student.name} registered successfully for {course.name}.'
      else:
         message = f'{student.name} is already registered for {course.name}.'
      return render(request, 'registration/registration_confirmation.html', {'message':
message})
   return render(request, 'registration/student_registration.html', {'course': course})
```

Create corresponding HTML templates in **registration/templates/registration** folder:

- **course_list.html** to display the list of courses.
- **student_registration.html** for student registration form.
- **registration_confirmation.html** for displaying registration confirmation message.

Step 7:

Set up URLs: Configure URLs to route requests to the views. In **registration/urls.py**, define URL patterns:

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

urlpatterns = [
   path('', views.course_list, name='course_list'),
   path('register/<int:course_id>/', views.register_student, name='register_student'),
]
```

Include these URLs in the main project's **urls.py** file:

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

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

```
    path('', include('registration.urls')),
]
```

Step 8: Run the development server: Start the Django development server to test the app:

```
python manage.py runserver
```

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.

Solution:

To add registration functionality to the Django admin interface and perform migrations, follow these steps:

Step 1: Register the models in the admin interface: Open the **registration/admin.py** file and update it as follows to add registration functionality to the Django admin interface:

```
from django.contrib import admin
from .models import Course, Student

@admin.register(Course)
class CourseAdmin(admin.ModelAdmin):
    list_display = ['name', 'description']

@admin.register(Student)
class StudentAdmin(admin.ModelAdmin):
    list_display = ['name', 'email']
    filter_horizontal = ['courses']
```

Step 2:

Perform migrations: Apply the migrations to reflect the changes made to the admin interface and models:

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

Step 3: Illustrate data entry through admin forms: Start the Django development server if it's not already running:

```
python manage.py runserver
```

Step 4:

Data Entry:

- Click on "Courses" under the "Registration" section to add courses. Enter the course name and description, and click "Save."
- Next, click on "Students" under the "Registration" section to add students. Enter the student's name, email, and select the courses they are enrolled in from the list of available courses. You can use the filter horizontal widget to make it easier to select multiple courses.
- After adding students and courses, you can view the list of students enrolled in each course by clicking on the course name in the "Courses" list view. This will show you a detailed view of the course, including the list of enrolled students.

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

Solution:

To create a ModelForm for the Student model that includes fields for the project's topic, languages used, and duration, you can follow these steps:

Step 1: Define the Project model: First, define the Project model in your Django app's **models.py** file:

```
from django.db import models

class Project(models.Model):
    topic = models.CharField(max_length=255)
    languages_used = models.CharField(max_length=255)
    duration = models.IntegerField()  # Assuming duration is in days

    def__str_(self):
        return self.topic
```

Step 2: Create a ModelForm for the Student model: Next, create a ModelForm for the Student model with fields for the project's topic, languages used, and duration. In your app's **forms.py** file, define the StudentForm:

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

class StudentForm(forms.ModelForm):
    project_topic = forms.CharField(max_length=255)
    languages_used = forms.CharField(max_length=255)
    duration = forms.IntegerField()

    class Meta:
        model = Student
        fields = ['name', 'email']  # Include other fields from the Student model as needed
```

Step 3: Update the Student model: Update the Student model in **models.py** to include a ForeignKey field for the Project model:

```
from django.db import models
from .project import Project  # Import the Project model

class Student(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    project = models.ForeignKey(Project, on_delete=models.CASCADE, null=True,
blank=True)

    def__str_(self):
        return self.name
```

Step 4: Update the admin interface for Project: Register the Project model in the admin interface to manage projects:

```
from django.contrib import admin
from .models import Project

admin.site.register(Project)
```

Step 5: Use the StudentForm in views: In your views where you handle student data, import and use the StudentForm:

```
from django.shortcuts import render
from .forms import StudentForm

def create_student(request):
    if request.method == 'POST':
        form = StudentForm(request.POST)
        if form.is_valid():
            student = form.save(commit=False)
            # If the project data is included in the form, save it to the student object
            student.project_topic = form.cleaned_data['project_topic']
            student.languages_used = form.cleaned_data['languages_used']
            student.duration = form.cleaned_data['duration']
            student.save()
            return render(request, 'registration/student_created.html', {'student': student})
    else:
        form = StudentForm()
    return render(request, 'registration/student_form.html', {'form': form})
```

Step 6: Create HTML templates: Create HTML templates for the student form and success message. For example, **student_form.html** and **student_created.html**.

Step 7: Include URLs: Include URLs to access the views in your app's **urls.py** file.

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

urlpatterns = [
    path('create-student/', views.create_student, name='create_student'),
    # Add other URL patterns as needed
]
```

Module-4

1. For students enrolment developed in Module 2, create a generic class view which displays list of students and detailview that displays student details for any selected student in the list.

   Solution:

   To create a Django app that handles student registration for courses, displays a list of students registered for a selected course, and includes generic class-based views for listing students and displaying student details, you can follow these steps:

   Step 1: Set up the Django app and models:

   First, create a Django app for student registration and course management:

   python manage.py startapp registration_app

   Define the models for **Course** and **Student** in **registration_app/models.py**:

```
from django.db import models

class Course(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

    def__str_(self):
        return self.name

class Student(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    courses = models.ManyToManyField(Course, related_name='students')

    def__str_(self):
        return self.name
```

   Step 2: Create views using generic class-based views: Define views using Django's generic class-based views for listing students and displaying student details. Create a file named **views.py** in the **registration_app** directory and add the following code:

```
from django.views.generic import ListView, DetailView
from .models import Course, Student

class CourseListView(ListView):
    model = Course
    template_name = 'registration_app/course_list.html'

class StudentDetailView(DetailView):
    model = Student
    template_name = 'registration_app/student_detail.html'
```

   Step 3: Create URL patterns: Define URL patterns in the **urls.py** file of the **registration_app** app to map views to URLs:

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

urlpatterns = [
    path('courses/', views.CourseListView.as_view(), name='course_list'),
    path('student/<int:pk>/', views.StudentDetailView.as_view(),
name='student_detail'),
]
```

Step 4:

Create HTML templates: Create HTML templates to display the course list, student list, and student details. In the **registration_app/templates/registration_app** directory, create the following templates:

- **course_list.html** for displaying the list of courses and linking to student lists.
- **student_list.html** for displaying the list of students registered for a selected course and linking to student details.
- **student_detail.html** for displaying detailed information about a student.

Sample **course_list.html**:

```
<!DOCTYPE html>
<html>
<head>
    <title>Course List</title>
</head>
<body>
    <h1>Course List</h1>
    <ul>
        {% for course in object_list %}
            <li><a href="{% url 'student_list' course.pk %}">{{ course.name }}</a></li>
        {% endfor %}
    </ul>
</body>
</html>
```

Sample **student_list.html**:

```
<!DOCTYPE html>
<html>
<head>
    <title>Student List</title>
</head>
<body>
    <h1>Students Registered for {{ course.name }}</h1>
    <ul>
        {% for student in course.students.all %}
            <li><a href="{% url 'student_detail' student.pk %}">{{ student.name
}}</a></li>
        {% endfor %}
```

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

Sample **student_detail.html**:

```
<!DOCTYPE html>
<html>
<head>
    <title>Student Detail</title>
</head>
<body>
    <h1>{{ object.name }}</h1>
    <p>Email: {{ object.email }}</p>
    <p>Courses:</p>
    <ul>
        {% for course in object.courses.all %}
            <li>{{ course.name }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Step 5: Update settings and URLs: Add the **registration_app** to the **INSTALLED_APPS** in your project's settings file (**settings.py**). Also, include the app's URLs in the project's **urls.py** file:

```
# settings.py
INSTALLED_APPS = [
    ...
    'registration_app',
    ...
]

# urls.py (project-level)
from django.contrib import admin
from django.urls import path, include

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

Step 6: Run migrations and start the server: Apply migrations to create database tables for the models and start the Django development server:

```
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
```

Step 7: Now, you can access the app in your browser at **http://127.0.0.1:8000/registration/courses/** to view the course list, click on a course to see the list of students registered for that course (**http://127.0.0.1:8000/registration/student/<course_id>/**), and click on a student's name to view their details (**http://127.0.0.1:8000/registration/student/<student_id>/**).

2. Develop example Django app that performs CSV and PDF generation for any models created in previous laboratory component.

Solution:

To create a Django app that performs CSV and PDF generation for any models created, you can follow these steps. We'll assume you have a model named **Student** in your Django app as an example.

Step 1: Install Required Libraries: First, you need to install the necessary libraries for CSV and PDF generation. You can do this using pip:

pip install django-csv django-pdf

Step 2:

Create CSV Generation Function: In your Django app, create a function to generate a CSV file based on the data from your models. For example, in a file named **utils.py**:

```
import csv
from django.http import HttpResponse

def generate_csv_response(queryset, filename):
    response = HttpResponse(content_type='text/csv')
    response['Content-Disposition'] = f'attachment; filename="{filename}.csv"'

    writer = csv.writer(response)
    # Write header row based on model fields
    writer.writerow([field.name for field in queryset.model._meta.fields])

    # Write data rows
    for obj in queryset:
        writer.writerow([getattr(obj, field.name) for field in
queryset.model._meta.fields])

    return response
```

Step 3: Create PDF Generation Function: Similarly, create a function to generate a PDF file based on the data from your models. You can use the **reportlab** library for PDF generation. Install it using pip if you haven't already:

pip install reportlab

Then, in **utils.py**, add the PDF generation function:

```
from reportlab.pdfgen import canvas
from django.http import HttpResponse

def generate_pdf_response(queryset, filename):
    response = HttpResponse(content_type='application/pdf')
    response['Content-Disposition'] = f'attachment; filename="{filename}.pdf"'

    # Create PDF document
    pdf = canvas.Canvas(response)
    y = 800  # Initial y position for writing text
```

```
    # Write header
    pdf.setFont("Helvetica-Bold", 12)
    pdf.drawString(100, y, "Student Data")

    # Write data
    pdf.setFont("Helvetica", 10)
    y -= 20 # Move down for data rows
    for obj in queryset:
        data = f"Name: {obj.name}, Email: {obj.email}" # Customize based on your
model fields
        pdf.drawString(100, y, data)
        y -= 15 # Move down for next row

    pdf.showPage()
    pdf.save()

    return response
```

Step 4: Add URL Routes: Define URL routes in your app's **urls.py** to map to the CSV and PDF generation views:

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

urlpatterns = [
    path('generate-csv/', views.generate_csv_view, name='generate_csv'),
    path('generate-pdf/', views.generate_pdf_view, name='generate_pdf'),
]
```

Step 5: Create Views: In your app's **views.py**, create views that use the CSV and PDF generation functions:

```
from django.shortcuts import render
from django.http import HttpResponse
from .models import Student
from .utils import generate_csv_response, generate_pdf_response

def generate_csv_view(request):
    queryset = Student.objects.all() # Get data from your models
    response = generate_csv_response(queryset, 'students_data')
    return response

def generate_pdf_view(request):
    queryset = Student.objects.all() # Get data from your models
    response = generate_pdf_response(queryset, 'students_data')
    return response
```

Step 6: Link to Views in Templates or Views: In your templates or views, add links or buttons to trigger CSV and PDF generation:

```
<!-- Example in a template -->
<a href="{% url 'generate_csv' %}">Download CSV</a>
<a href="{% url 'generate_pdf' %}">Download PDF</a>
```

Step 7: Test the App: Run your Django development server and test the CSV and PDF generation functionality by navigating to the appropriate URLs (**/generate-csv/** and **/generate-pdf/**) and clicking the download links.

This setup will allow you to generate CSV and PDF files containing data from your models. Customize the functions and views as needed to suit your specific model and data requirements.

Module-5

1. Develop a registration page for student enrolment as done in Module 2 but without page refresh using AJAX.

Solution:

To create a registration page for student enrollment without page refresh using AJAX (Asynchronous JavaScript and XML), you can follow these steps:

Step 1: Set up the Django app and models: First, create a Django app for student enrollment:

python manage.py startapp enrollment_app

Define the models for **Course** and **Student** in **enrollment_app/models.py**:

```
from django.db import models

class Course(models.Model):
    name = models.CharField(max_length=100)
    description = models.TextField()

    def__str_(self):
        return self.name

class Student(models.Model):
    name = models.CharField(max_length=100)
    email = models.EmailField()
    course = models.ForeignKey(Course, on_delete=models.CASCADE)

    def__str_(self):
        return self.name
```

Step 2: Create views and templates for registration: Define views and templates for the registration form and submission handling. In **enrollment_app/views.py**, add the following code:

```
from django.shortcuts import render
from django.http import JsonResponse
from .models import Course, Student

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

def register_student(request):
    if request.method == 'POST':
        name = request.POST.get('name')
        email = request.POST.get('email')
        course_id = request.POST.get('course')
        course = Course.objects.get(pk=course_id)
```

```
            student = Student.objects.create(name=name, email=email, course=course)
            return JsonResponse({'success': True})
        return JsonResponse({'success': False})
```

Create a **registration.html** template in the **enrollment_app/templates/enrollment_app** directory:

```
<!DOCTYPE html>
<html>
<head>
    <title>Student Registration</title>
    <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</head>
<body>
    <h1>Student Registration</h1>
    <form id="registration-form">
        <label for="name">Name:</label>
        <input type="text" id="name" name="name" required><br>
        <label for="email">Email:</label>
        <input type="email" id="email" name="email" required><br>
        <label for="course">Course:</label>
        <select id="course" name="course" required>
            {% for course in courses %}
                <option value="{{ course.id }}">{{ course.name }}</option>
            {% endfor %}
        </select><br>
        <button type="submit">Register</button>
    </form>

    <div id="message"></div>

    <script>
        $(document).ready(function() {
            $('#registration-form').submit(function(e) {
                e.preventDefault();
                $.ajax({
                    type: 'POST',
                    url: '{% url "register_student" %}',
                    data: $(this).serialize(),
                    success: function(response) {
                        if (response.success) {
                            $('#message').text('Registration successful!');
                            $('#registration-form')[0].reset();
                        } else {
                            $('#message').text('Error: Registration failed.');
                        }
```

```
                }
            });
        });
    });
  </script>
</body>
</html>
```

Step 3: Define URL patterns and include them in the project's URLs: Define URL patterns in **enrollment_app/urls.py**:

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

urlpatterns = [
    path('', views.registration_page, name='registration_page'),
    path('register/', views.register_student, name='register_student'),
]
```

Include these URLs in the project's **urls.py** file:

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

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

Step 4: Run migrations and start the server: Apply migrations to create database tables for the models and start the Django development server:

```
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
```

Now, you can access the registration page at **http://127.0.0.1:8000/enrollment/** and register students without page refresh using AJAX. The form will submit the data asynchronously, and the success or error message will be displayed without reloading the entire page.

2. Develop a search application in Django using AJAX that displays courses enrolled by a student being searched.

Solution:

To create a search application in Django using AJAX that displays courses enrolled by a student being searched, follow these steps:

Step 1: Set up the Django app and models: First, create a Django app for the search application:

python manage.py startapp search_app

Define the models for **Course** and **Student** in **search_app/models.py**:

```
from django.db import models

class Course(models.Model):
    name = models.CharField(max_length=100)

    def__str_(self):
        return self.name

class Student(models.Model):
    name = models.CharField(max_length=100)
    courses = models.ManyToManyField(Course)

    def__str_(self):
        return self.name
```

Step 2: Create views and templates: Define views and templates to handle the search functionality and display the results. In **search_app/views.py**, create a view for handling the search request:

```
from django.shortcuts import render
from .models import Student

def search_courses(request):
    if request.method == 'GET' and 'student_name' in request.GET:
        student_name = request.GET['student_name']
        student = Student.objects.filter(name_icontains=student_name).first()
        if student:
            courses = student.courses.all()
        else:
            courses = []
        return render(request, 'search_app/course_list.html', {'courses': courses})
    return render(request, 'search_app/search_form.html')
```

Create two HTML templates in the **search_app/templates/search_app** directory:

- **search_form.html** for the search form.
- **course_list.html** for displaying the list of courses.

search_form.html:

```
<!DOCTYPE html>
```

```
<html>
<head>
  <title>Search Form</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/jquery/3.6.0/jquery.min.js"></script>
</head>
<body>
  <h1>Search for Courses by Student Name</h1>
  <form id="searchForm" method="GET">
    <label for="studentName">Student Name:</label>
    <input type="text" id="studentName" name="student_name">
    <button type="submit">Search</button>
  </form>
  <div id="courseList"></div>

  <script>
    $(document).ready(function() {
      $('#searchForm').on('submit', function(event) {
        event.preventDefault();
        var formData = $(this).serialize();
        $.ajax({
          url: '{% url "search_courses" %}',
          type: 'GET',
          data: formData,
          success: function(response) {
            $('#courseList').html(response);
          },
          error: function(xhr, errmsg, err) {
            console.log(xhr.status + ": " + xhr.responseText);
          }
        });
      });
    });
  </script>
</body>
</html>
```

course_list.html:

```
<!DOCTYPE html>
<html>
<head>
  <title>Course List</title>
</head>
<body>
  <h2>Courses Enrolled by {{ student.name }}</h2>
```

```
    <ul>
        {% for course in courses %}
            <li>{{ course.name }}</li>
        {% endfor %}
    </ul>
</body>
</html>
```

Step 3: Update URLs: Define URL patterns in **search_app/urls.py** to map views to URLs:

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

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

Include these URLs in the project's **urls.py**:

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

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

Step 4: Run migrations and start the server: Apply migrations to create database tables for the models and start the Django development server:

```
python manage.py makemigrations
python manage.py migrate
python manage.py runserver
```

Now, you can access the search form in your browser at **http://127.0.0.1:8000/search/** and search for courses enrolled by a student by entering their name. The results will be displayed using AJAX without refreshing the page