INDEX

Expt.No:1	Simple weather application
Date: 28/07/23	

Aim:

To build a project for simple weather application using Node.js.

Project Components and Requirements:

1. Node.js Installation: Ensure that Node.js is installed and available on the system. Node.js is used to run server-side JavaScript code.

2. Get OpenWeatherMap API Key: Obtain an API key from OpenWeatherMap by signing up on their website. This API key is used to make requests to the OpenWeatherMap API to fetch weather data.

3. Create a Project Directory: Organize the project by creating a directory to contain the files.

4. Install Dependencies: Use npm (Node Package Manager) to install the required dependencies for the project. The following packages are used:
    - axios: A popular HTTP client for making API requests.
    - dotenv: A package for loading environment variables from a .env file. readline: A package for reading input from the command line.

5. Application Program (weather.js): This JavaScript file contains the server-side code for the weather application. It includes the following features:
    - Fetching weather data from the OpenWeatherMap API using the provided API key.
    - Creating an HTTP server to serve an HTML page and handle weather information requests.
    - Parsing query parameters from the URL to retrieve the user's desired location. Displaying weather information on the HTML page and handling errors.

6. HTML Template (index.html): This HTML file is the user interface for the weather application. It includes a form where users can input a city name (and optionally, a country code). The weather information is displayed below the form. The HTML file also includes JavaScript code to make AJAX requests to the server and update the page with weather data.

Running the Application:

- To run the application, use the command node weather.js in the project directory.
- The server will start listening on port 3000 by default.
- This will provide access to the application by opening a web browser and navigating to http://localhost:3000. Then enter the city name and click the "Get Weather" button to fetch and display weather information.

Output:

- When users input a valid location, the application fetches and displays the city name, temperature, and weather description.

Execution Steps:

- Node.js should be installed and available.
- Get an OpenWeatherMap API Key, https://home.openweathermap.org/users/sign_up
- Generate API key
- Create a Project Directory.
- Install Dependencies. npm install axios dotenv npm install readline
- Application Program

weather.js

```
const axios =
require('axios'); const http =
require('http'); const fs =
require('fs'); const path =
require('path');

// Function to fetch weather data async
function fetchWeatherData(city) {
  const apiKey = 'f4906f865b631341d0f3fbf49114de36';
// Replace with your OpenWeatherMap API key
  const apiUrl =
`https://api.openweathermap.org/data/2.5/weather?q=${city}&appid=${apiKey}`;

  try {
    const response = await
   axios.get(apiUrl);  const weatherData =
   response.data;  const weatherInfo = {
      city: weatherData.name,  temperature: (weatherData.main.temp
     - 273.15).toFixed(2) + '°C',  description:
     weatherData.weather[0].description
    };    return
 weatherInfo;
  }   catch   (error)   {       console.error('An    error
   occurred:', error.message);  return null;
  }
}
const url = require('url');
const server = http.createServer((req, res) => {
```

2

```
  const parsedUrl = url.parse(req.url, true); // Parse the URL including query
parameters
  if (req.method === 'GET' && parsedUrl.pathname === '/')
   {  // Read the HTML file and serve it as the response
   const filePath = path.join(__dirname, 'index.html');
   fs.readFile(filePath, 'utf8', (err, data) => {
      if (err) {
        res.writeHead(500, { 'Content-Type': 'text/plain' });
        res.end('Internal Server Error');
      } else {  res.writeHead(200, { 'Content-Type':
       'text/html' });  res.end(data);
      }
    });
  } else if (req.method === 'GET' && parsedUrl.pathname === '/weather')
   {  // Handle weather information request with query parameters  const
   location = parsedUrl.query.location;  if (location) {
      // Fetch weather data and send the response
     fetchWeatherData(location)
        .then(weatherData => {
         if (weatherData) {
            // Display weather information
            res.writeHead(200, { 'Content-Type': 'application/json' });
           res.end(JSON.stringify(weatherData));
          } else {
            // Return an error response  res.writeHead(404, {
           'Content-Type': 'application/json' });
            res.end(JSON.stringify({ error: 'Location not found' }));
         }
        })
        .catch(error => {  console.error('An error occurred:',
         error.message);  res.writeHead(500, { 'Content-Type':
         'application/json' });  res.end(JSON.stringify({ error:
         'Internal Server Error' }));
        });
    } else {  res.writeHead(400, { 'Content-Type':
     'application/json' });
      res.end(JSON.stringify({ error: 'Location parameter missing' }));
   }
  } else {  res.writeHead(404, { 'Content-Type':
   'text/plain' });  res.end('Not Found');
  }
});
// Listen on a port (e.g., 3000)
const port = 3000;
server.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
```

```
});
```

Html Template

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Weather Application</title>
    <style>
        /* Add CSS styles for formatting the weather information */
</style>
</head>
<body>
    <h1>Weather Information</h1>

    <!-- Add a form for user input -->
    <form id="location-form">
        <label for="location">Enter a city name (and optionally, a country code,
separated by a comma):</label>
        <input type="text" id="location" name="location" required>
        <button type="submit">Get Weather</button>
   </form>

    <!-- Weather information will be displayed here -->
    <div id="weather-info">
        <!-- Weather data will be displayed here -->
   </div>

    <script>
  const weatherInfoDiv = document.getElementById('weather-info');
 const locationInput = document.getElementById('location');

  document.getElementById('location-form').addEventListener('submit', async (e) =>
{
    e.preventDefault();  const location
   = locationInput.value;

    // Make an AJAX request to the server with the location as a query parameter
const response = await
```

```
fetch(`/weather?location=${encodeURIComponent(location)}`);
   const weatherData = await response.json();

    if (weatherData && !weatherData.error) {
      // Display weather information
     weatherInfoDiv.innerHTML = `
        <p>City: ${weatherData.city}</p>
```

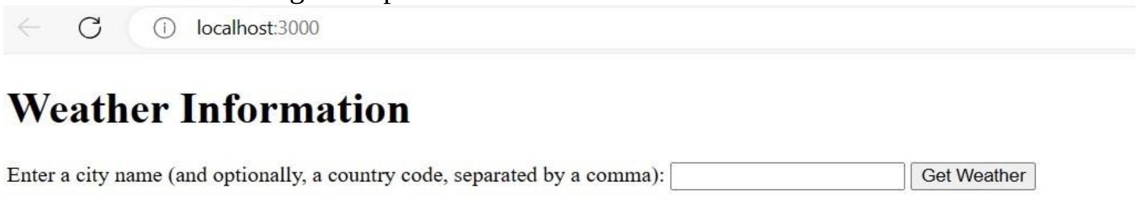
```
          <p>Temperature: ${weatherData.temperature}</p>
          <p>Weather: ${weatherData.description}</p>
        `;
      } else {
        // Display an error message       weatherInfoDiv.innerHTML =
`<p>${weatherData.error || 'An error occurred.'}</p>`;
      }
    });
    </script>
</body>
</html>
```

- Run the Application.

```
node weather.js
```

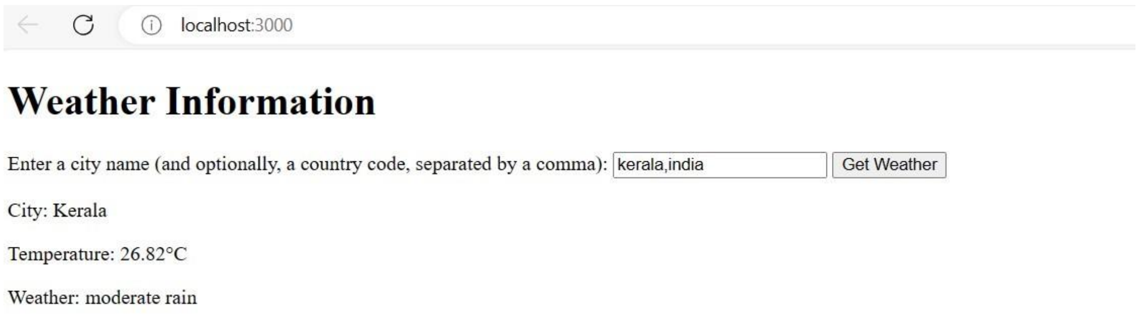
- Output

```
C:\node-one>node weather.js
Server is running on http://localhost:3000
```


localhost:3000

Weather Information

Enter a city name (and optionally, a country code, separated by a comma): Get Weather


localhost:3000

Weather Information

Enter a city name (and optionally, a country code, separated by a comma): kerala,india Get Weather

City: Kerala

Temperature: 26.82°C

Weather: moderate rain

Conclusion:

This application allows to input a city name (and optionally, a country code), and it fetches and displays the weather information for that location. Overall, this project demonstrates the creation of a basic web application using Node.js and serves as an example of how to interact with an external API (OpenWeatherMap) to fetch and display real-time weather data. Users can access the weather information through a user-friendly web interface.

Expt.No:2	URL shortener Application using SQL
Date:10/08/23	

Aim:

To build a project for URL shortener using SQL and Django.

Project Components and Requirements:

1. Python and Django: Ensure you have Python installed on your system. Also need Django, which can be installed using pip.

2. Database: Decide on the SQL database want to use (e.g., SQLite, PostgreSQL, MySQL) and ensure it is installed and running.

3. Django App: Create a Django project and an app within the project where your URL shortener functionality will reside.

4. Models: Define a Django model to represent URLs. This model should include fields for the original (long) URL, the shortened URL, and any other relevant information you want to store.

5. Views: Create views for your URL shortener application. This views is needed for the home page, URL submission form, and redirection. These views should interact with your model to retrieve and store URLs.

6. Templates: Design HTML templates for your application's pages, including the home page, URL submission form, and redirection page.

7. URL Routing: Configure URL patterns in your app's urls.py file to route requests to the appropriate views.

8. Forms: Create a Django form for the URL submission form. This form should validate and process user input.

9. Shortening Algorithm: Implement a URL shortening algorithm. Use base62 or base36 encoding, random character generation, or a combination of methods to generate short URLs.

10. Database Configuration: Configure the Django project settings to use the SQL database of your choice (e.g., SQLite or PostgreSQL). Update the DATABASES setting in settings.py.

11. Migrations: Run migrations to create the database tables based on your models.

12. Static and Media Files: Configure static and media file handling in your Django project settings if you plan to use CSS, JavaScript, or store user-uploaded files.

13. User Interface: Design and style the application's user interface using CSS or a frontend framework if desired.

14. Deployment: Choose a hosting platform (e.g., Heroku, AWS, DigitalOcean) for deploying the Django application in a production environment.

15. Domain and DNS: If needed to use a custom domain for the URL shortener, set up DNS records to point to the application's server.

Execution Steps:

1. Create Virtual Environment (Recommended):

```
python -m venv myenv

myenv\Scripts\activate
```

2. Install Django, SQLAlchemy other necessary packages:

```
pip install Django

pip install SQLAlchemy
```

3. Create a Django Project and an Application:

```
python -m django startproject urlshortener
```

If any issues Re-Install Django as follows

```
pip uninstall django

pip install Django

cd urlshortener

python manage.py startapp urlshort
```

4. Create Database and apply Migrations:

Run the following commands to create default database and apply migrations:

```
python manage.py makemigrations

python manage.py migrate
```

5. Configure Database:

In the project's settings (urlshortener/settings.py), configure the database to use SQLite:

```
INSTALLED_APPS = [
        'django.contrib.admin',
        'django.contrib.auth',
        'django.contrib.contenttypes',
        'django.contrib.sessions',
        'django.contrib.messages',
        'django.contrib.staticfiles',
        'urlshort',
]

DATABASES = {
   'default': {
       'ENGINE': 'django.db.backends.sqlite3',
       'NAME': BASE_DIR / 'db.sqlite3',
   }
}
```

6. Define Models:

In the urlshort/models.py file, define the data models using Django's ORM (ObjectRelational Mapping).

```
# urlshort/models.py
from django.contrib.auth.models import User
from django.db import models

class URL(models.Model):
    long_url = models.URLField(unique=True)
    short_url = models.CharField(max_length=20, unique=True)

class UrlModel(models.Model):  user =
   models.ForeignKey(User, on_delete=models.CASCADE)
   original_url = models.URLField()
    short_url = models.CharField(max_length=20, unique=True)
```

7. Define Views:

In the urlshort/views.py file, create views in your app's views.py to handle requests and responses.

```
# urlshort/views.py
from django.shortcuts import render, redirect,
get_object_or_404 from .models import URL from .utils
import generate_short_url

def home(request):
    # Fetch a list of all stored URLs
   urls = URL.objects.all()
```

```
    return render(request, 'urlshort/home.html', {'urls': urls})

def shorten_url(request):
    if request.method == 'POST':
        long_url = request.POST['long_url']
        short_url = generate_short_url()
        url = URL(long_url=long_url,
        short_url=short_url)  url.save()  return
        redirect('home')
    return render(request, 'urlshort/shorten_url.html')

def redirect_to_original(request, short_url):
    url = get_object_or_404(URL, short_url=short_url)
   return redirect(url.long_url)
```

<u>8. Define utils:</u>

In the urlshort/utils.py file, simple function to generate short URLs using random characters is created.

```
import random import
string def
generate_short_url():
"""
    Generate a random short URL using alphanumeric characters.
    You can customize the length and characters used as needed.
    """
    # Define the characters to choose from for the short URL
    characters = string.ascii_letters + string.digits  # Alphanumeric characters

    # Set the desired length of the short URL
    short_url_length = 6  # You can adjust this to your preference

    # Generate a random short URL
    short_url = ''.join(random.choice(characters) for _ in range(short_url_length))

    return short_url
```

<u>9. Create Templates:</u>

home.html

```
<!-- urlshort/templates/urlshort/home.html -->
<!DOCTYPE html>
<html>
<head>
    <title>URL Shortener</title>
</head>
<body>
    <h1>Welcome to the URL Shortener</h1>

    <h2>Stored URLs:</h2>
    <ul>
```

```
        {% for url in urls %}
            <li><a href="{{ url.short_url }}">{{ url.short_url }}</a></li>
        {% empty %}
            <p>No URLs stored yet.</p>
        {% endfor %}
    </ul>

    <h2>Shorten a New URL:</h2>
    <form method="post" action="{% url 'shorten_url' %}">
        {% csrf_token %}
        <input type="url" name="long_url" required>
        <input type="submit" value="Shorten">
    </form>
</body>
</html>
```

Shorten_url.html

```
<!-- urlshort/templates/urlshort/shorten_url.html -->
<!DOCTYPE html>
<html>
<head>
    <title>URL Shortener</title>
</head>
<body>
    <h1>Shorten a URL</h1>

    <form method="post">
        {% csrf_token %}
        <label for="long_url">Enter a long URL:</label>
        <input type="url" id="long_url" name="long_url" required>
        <input type="submit" value="Shorten">
    </form>
</body>
</html>
```

redirect.html

```
<!-- urlshort/templates/urlshort/redirect.html -->
<!DOCTYPE html>
<html>
<head>
    <title>URL Shortener - Redirecting...</title>
</head>
<body>
    <h1>Redirecting...</h1>
    <p>You will be redirected to the original URL in a moment.</p>
</body>
</html>
```

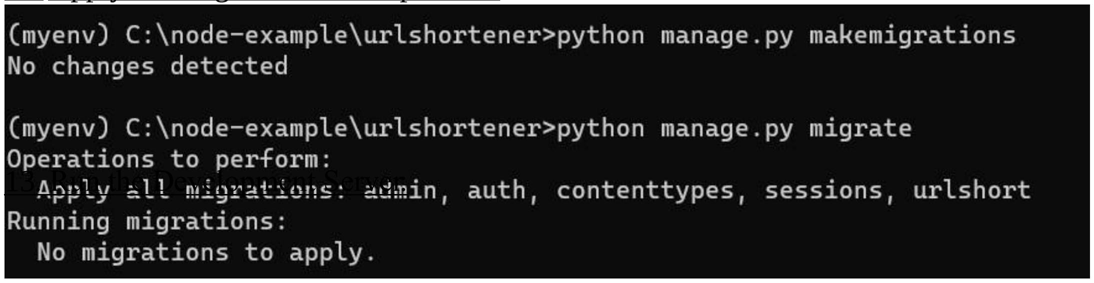
10. Update Project and App URLs:

```
# urlshort/urls.py (inside your app)
from django.urls import path from .
import views

urlpatterns = [
    path('', views.home, name='home'),
    path('shorten/', views.shorten_url, name='shorten_url'),
   path('<str:short_url>/', views.redirect_to_original,
   name='redirect_to_original'),
]

# urlshortener/urls.py (your main project's urls.py) from
django.contrib import admin from django.urls import path,
include  # Import include here urlpatterns = [
    path('admin/', admin.site.urls),  path('', include('urlshort.urls')),  #
   Include the URL patterns from your app
]
```

11. Create SuperUser(if required)
Create an admin user to access the Django admin panel:

python manage.py createsuperuser Follow
the prompts to create the admin user.

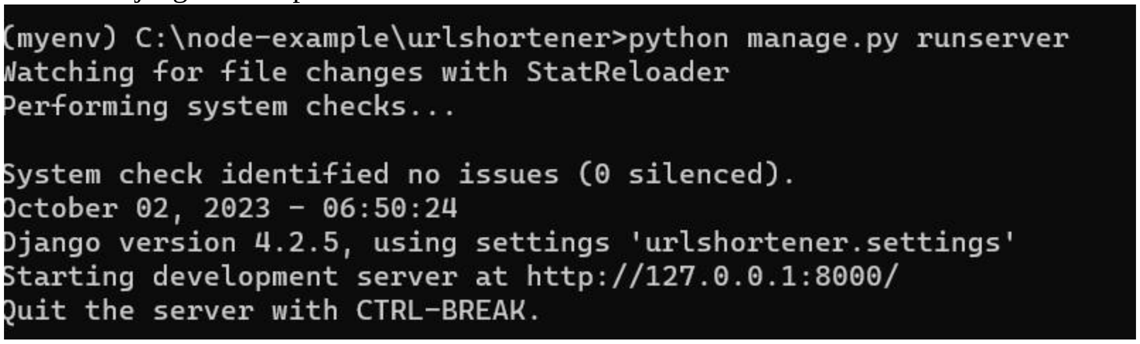
12. Apply the Migrations after updation:



```
(myenv) C:\node-example\urlshortener>python manage.py makemigrations
No changes detected

(myenv) C:\node-example\urlshortener>python manage.py migrate
Operations to perform:
  Apply all migrations: admin, auth, contenttypes, sessions, urlshort
Running migrations:
  No migrations to apply.
```

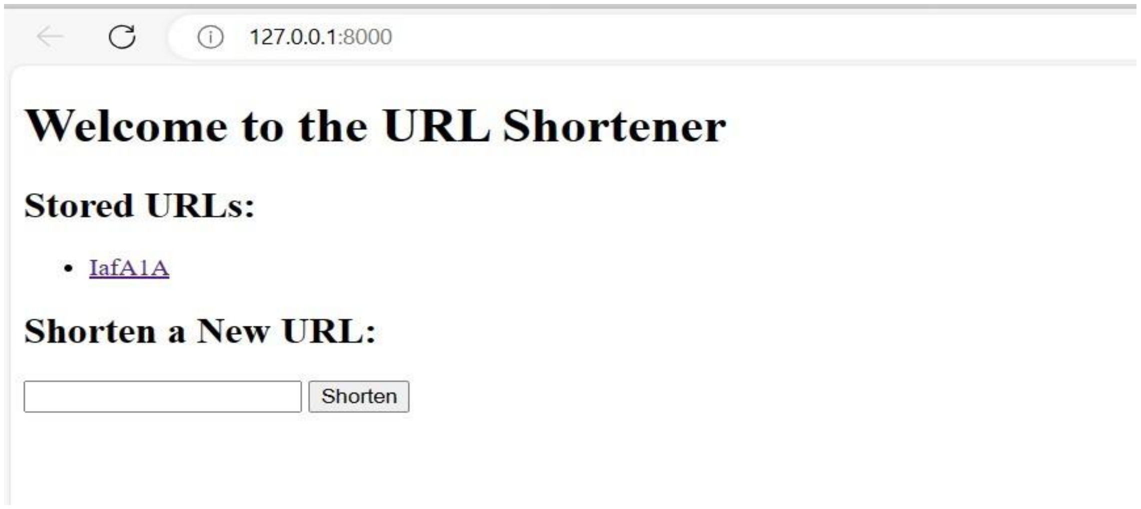
13. Run the Development Server:

Start the Django development server:



```
(myenv) C:\node-example\urlshortener>python manage.py runserver
Watching for file changes with StatReloader
Performing system checks...

System check identified no issues (0 silenced).
October 02, 2023 - 06:50:24
Django version 4.2.5, using settings 'urlshortener.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.
```

14. Outputs:



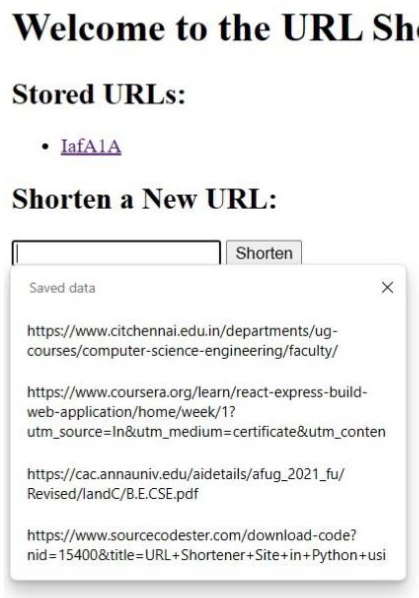
127.0.0.1:8000

Welcome to the URL Shortener

Stored URLs:

- IafA1A

Shorten a New URL:



Welcome to the URL Shortener

Stored URLs:

- IafA1A

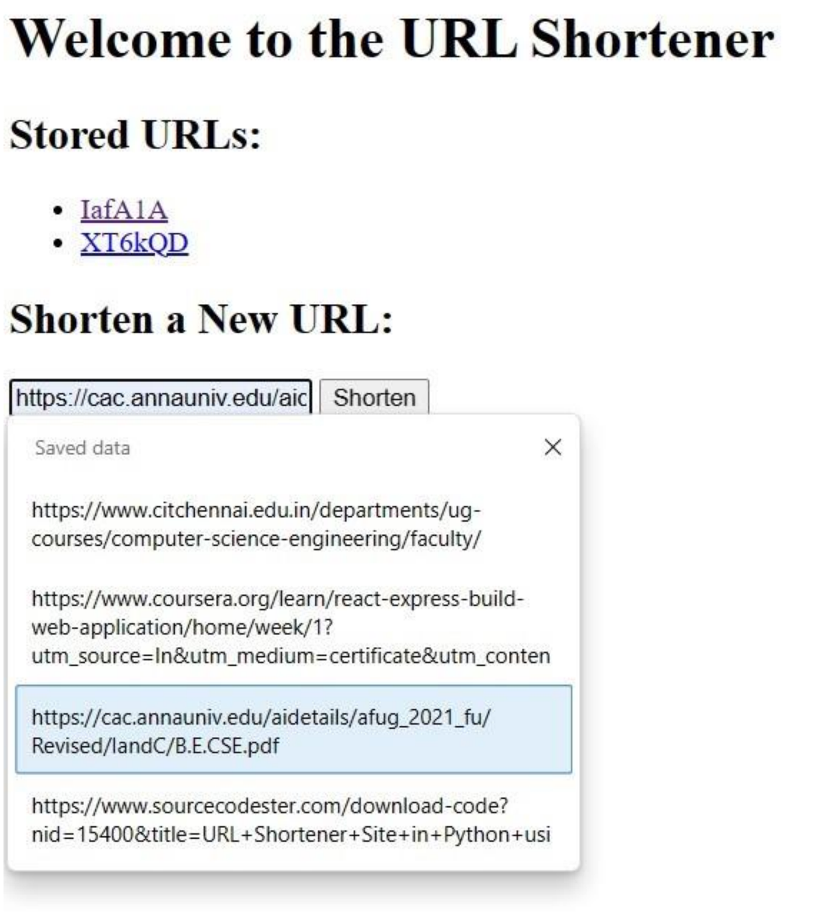
Shorten a New URL:

https://www.citchennai.edu.in/departments/ug-courses/computer-science-engineering/faculty/

https://www.coursera.org/learn/react-express-build-web-application/home/week/1?utm_source=ln&utm_medium=certificate&utm_conten

https://cac.annauniv.edu/aidetails/afug_2021_fu/Revised/landC/B.E.CSE.pdf

https://www.sourcecodester.com/download-code?nid=15400&title=URL+Shortener+Site+in+Python+usi



Welcome to the URL Shortener

Stored URLs:

- IafA1A
- XT6kQD

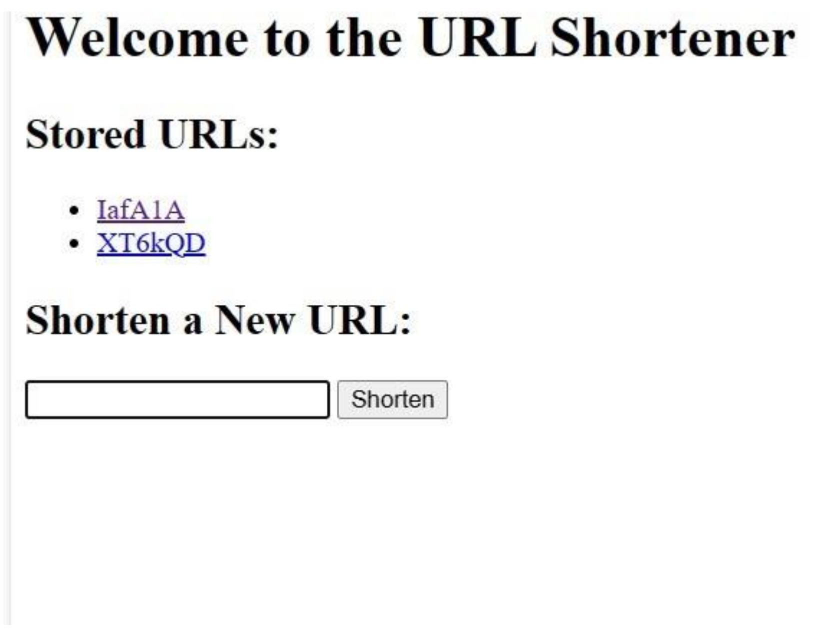
Shorten a New URL:

https://www.citchennai.edu.in/departments/ug-courses/computer-science-engineering/faculty/

https://www.coursera.org/learn/react-express-build-web-application/home/week/1?utm_source=ln&utm_medium=certificate&utm_conten

https://cac.annauniv.edu/aidetails/afug_2021_fu/Revised/landC/B.E.CSE.pdf

https://www.sourcecodester.com/download-code?nid=15400&title=URL+Shortener+Site+in+Python+usi



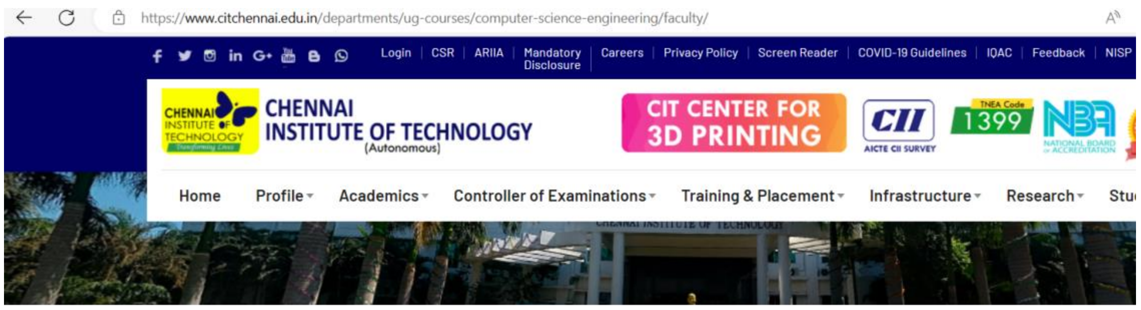
Welcome to the URL Shortener

Stored URLs:

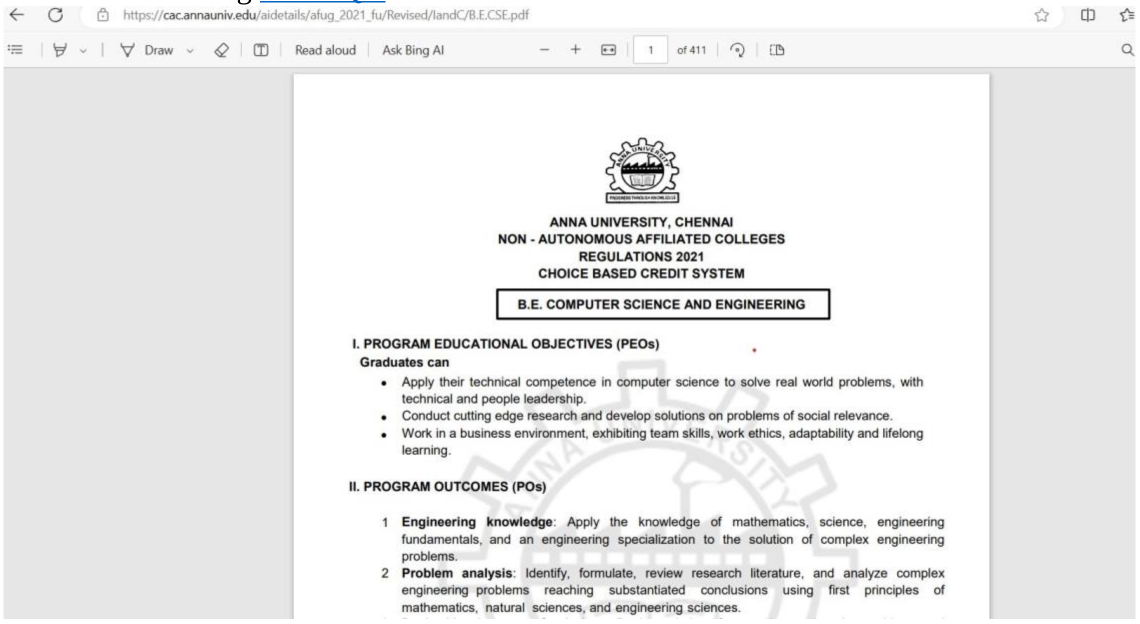
- IafA1A
- XT6kQD

Shorten a New URL:

- After Clicking IafA1A


https://www.citchennai.edu.in/departments/ug-courses/computer-science-engineering/faculty/
Login | CSR | ARIIA | Mandatory Disclosure | Careers | Privacy Policy | Screen Reader | COVID-19 Guidelines | IQAC | Feedback | NISP
CHENNAI INSTITUTE OF TECHNOLOGY (Autonomous)
CIT CENTER FOR 3D PRINTING
AICTE CII SURVEY
TNEA Code 1399
Home
Profile
Academics
Controller of Examinations
Training & Placement
Infrastructure
Research

- After Clicking XT6kQD


https://cac.annauniv.edu/aidetails/afug_2021_fu/Revised/landC/B.E.CSE.pdf
Draw
Read aloud
Ask Bing AI
1
of 411

ANNA UNIVERSITY, CHENNAI
NON - AUTONOMOUS AFFILIATED COLLEGES
REGULATIONS 2021
CHOICE BASED CREDIT SYSTEM

B.E. COMPUTER SCIENCE AND ENGINEERING

I. PROGRAM EDUCATIONAL OBJECTIVES (PEOs)
Graduates can
- Apply their technical competence in computer science to solve real world problems, with technical and people leadership.
- Conduct cutting edge research and develop solutions on problems of social relevance.
- Work in a business environment, exhibiting team skills, work ethics, adaptability and lifelong learning.

II. PROGRAM OUTCOMES (POs)

1. **Engineering knowledge**: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

Conclusion:

The URL Shortener Application using SQL and Django provides shortening of any URL and linking to the actual Webpage when clicked on the shortened URL content. It fulfils the basic requirements of URL shortening and redirection. In expansion and improvement, it can be refined and enhanced with the application based on user feedback and evolving requirements.

Expt.No:3	Flight ticket booking using any tech stack
Date:24/08/23	

Aim:

To create a web application for flight ticket booking using any tech stack for the backend and database.

Scenarios Considered:

Type of Users
a. User
b. Admin

User Use Cases
- Login
- Sign up
- Searching for flights based on date and time
- Booking tickets on a flight based on availability
- My Booking -> to list out all the bookings made by that user
- Logout

Admin Use Cases
- Login (Separate login for Admin)
- Add Flights
- Remove flights
- View all the booking based on flight number and time

User Use Cases:

1. Searching for Flights based on Date and Time
Create a new route in your Flask application to handle flight search based on date and time. Users can input their preferred date and time, and the application should return a list of available flights matching those criteria.

2. Booking Tickets on a Flight
When a user selects a flight from the search results, you can create a route for booking tickets. This route should handle the booking process, check for seat availability, and update the database with the booking information.

3. My Bookings
Create a "My Bookings" route where users can view all the bookings they've made. This route should query the database for bookings associated with the logged-in user and display the results.

4. Logout
Implement a logout functionality that clears the user's session and redirects them to the login page.
Admin Use Cases:

1. Admin Login
Create a separate login route and authentication system for administrators.

2. Add Flights
Implement a route for administrators to add new flights to the system. This route should allow the admin to input flight details, including flight number, date, time, and available seats.

3. Remove Flights
Create a route for administrators to remove flights from the system. This route should allow the admin to select a flight by its flight number and remove it from the database.

4. View All Bookings
Implement a route for administrators to view all bookings based on flight number and time. Admins should be able to filter and see the list of bookings for each flight.

Project Components and Requirements:

Tech Stack:

Backend: Python with Flask
Database: SQLite
Frontend: HTML, CSS, and JavaScript

Execution Steps:

Step 1: Set Up the Project Structure

Create a directory for your project and set up a virtual environment for Python.

```
mkdir   flight_booking_app
cd      flight_booking_app
python  -m  venv  venv
venv\Scripts\activate
```

Step 2: Install Required Packages

Install Flask and Flask extensions.

Create a text file with the name as requirements.txt and fix all these dependencies

```
blinker==1.6.3
click==8.1.7
colorama==0.4.6
itsdangerous==2.1.2
Jinja2==3.1.2
MarkupSafe==2.1.3
pip==23.2.1
setuptools==65.5.0
Werkzeug==2.0.2 flask-
bcrypt==0.7.1
```

```
mysqlclient==2.2.0
Flask==2.1.1
Flask-Login==0.5.0
Flask-SQLAlchemy==2.5.1
SQLAlchemy==1.4.26
```

Run in the following path

```
(new_venv3) C:\node-example\flight_booking_app>pip install -r requirements.txt Step 3:
```

Create source file with importing all dependencies and defining all routes app.py

```
from flask import Flask, render_template, redirect, url_for, flash, request
from flask_wtf import FlaskForm from wtforms import StringField,
PasswordField, SubmitField, DateTimeField from wtforms.validators import
DataRequired from flask_sqlalchemy import SQLAlchemy
from werkzeug.security import generate_password_hash, check_password_hash from
flask_login import LoginManager, UserMixin, login_user, login_required, logout_user,
current_user
from datetime import datetime

app = Flask(__name__)
app.config['SECRET_KEY'] = 'your-secret-key'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///flight_booking.db'
db = SQLAlchemy(app)

# Initialize Flask-Login
login_manager = LoginManager()
login_manager.login_view = 'login'
login_manager.init_app(app)

@login_manager.user_loader def
load_user(user_id):  return
User.query.get(int(user_id))

class RegistrationForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
   password = PasswordField('Password', validators=[DataRequired()])
   submit = SubmitField('Register')

class LoginForm(FlaskForm):
    username = StringField('Username', validators=[DataRequired()])
   password = PasswordField('Password',
   validators=[DataRequired()])  submit = SubmitField('Login')

class FlightForm(FlaskForm):
    flight_number = StringField('Flight Number', validators=[DataRequired()])
   departure_city = StringField('Departure City', validators=[DataRequired()])
    arrival_city = StringField('Arrival City', validators=[DataRequired()])
 departure_time = DateTimeField('Departure Time', validators=[DataRequired()],
format='%Y-%m-%d %H:%M:%S')
    submit = SubmitField('Add Flight')
```

```
class User(db.Model, UserMixin):
    id = db.Column(db.Integer, primary_key=True)  username =
   db.Column(db.String(80), unique=True, nullable=False)  password
   = db.Column(db.String(120), nullable=False)  bookings =
   db.relationship('Booking', backref='user', lazy=True)

    def __init__(self, username, password):
        self.username = username
        self.password = generate_password_hash(password, method='sha256')

class Flight(db.Model):
    id = db.Column(db.Integer, primary_key=True)  flight_number =
   db.Column(db.String(10), unique=True, nullable=False)  departure_city
   = db.Column(db.String(50), nullable=False)  arrival_city =
   db.Column(db.String(50), nullable=False)  departure_time =
   db.Column(db.DateTime, default=datetime.utcnow)  bookings =
   db.relationship('Booking', backref='flight', lazy=True)

class Booking(db.Model):
    id = db.Column(db.Integer, primary_key=True)  user_id =
   db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)  flight_id
   = db.Column(db.Integer, db.ForeignKey('flight.id'), nullable=False)
   booking_date = db.Column(db.DateTime, default=datetime.utcnow)

@app.route('/register', methods=['GET', 'POST'])
def register():
    form = RegistrationForm()
   if
   form.validate_on_submit():
        username = form.username.data  password = form.password.data
       user = User.query.filter_by(username=username).first()  if user:
       flash('Username already exists. Choose another username.', 'danger')
       else:
            new_user = User(username=username, password=password)
            db.session.add(new_user)  db.session.commit()
           flash('Registration successful. You can now log in.',
           'success')  return redirect(url_for('login'))
    return render_template('registration.html', form=form)

@app.route('/login', methods=['GET', 'POST'])
def login():
    form = LoginForm()  if
   form.validate_on_submit():
        username = form.username.data  password =
       form.password.data  user =
       User.query.filter_by(username=username).first()  if user
       and check_password_hash(user.password, password):
            login_user(user)  flash('Login
           successful!', 'success')  return
           redirect(url_for('dashboard'))
        else:
            flash('Login failed. Check your username and password.', 'danger')
    return render_template('login.html', form=form)

@app.route('/dashboard'
) @login_required def
dashboard():
    # Fetch the user's bookings  bookings =
   Booking.query.filter_by(user_id=current_user.id).all()  return
   render_template('dashboard.html', bookings=bookings)

@app.route('/logout'
) @login_required
def logout():
logout_user()
    return redirect(url_for('login'))

@app.route('/book_flight/<int:flight_id>', methods=['GET', 'POST'])
@login_required def
book_flight(flight_id):
    flight = Flight.query.get(flight_id)
   if request.method == 'POST':
        # Create a booking record for the logged-in user
        booking    =         Booking(user_id=current_user.id,        flight_id=flight.id,
booking_date=datetime.utcnow())  db.session.add(booking)  db.session.commit()
flash('Booking successful!', 'success')  return redirect(url_for('dashboard'))  return
render_template('book_flight.html', flight=flight)
@app.route('/add_flight', methods=['GET', 'POST'])
@login_required
def add_flight():
    form = FlightForm()     if
form.validate_on_submit():
# Create a new flight record
        flight = Flight(
            flight_number=form.flight_number.data,
departure_city=form.departure_city.data,
arrival_city=form.arrival_city.data,
            departure_time=form.departure_time.data
        )
        db.session.add(flight)
db.session.commit()        flash('Flight added
successfully!', 'success')         return
redirect(url_for('dashboard'))
    return render_template('add_flight.html', form=form)

@app.route('/remove_flight/<int:flight_id>', methods=['POST'])
@login_required def
remove_flight(flight_id):
    flight = Flight.query.get(flight_id)
if flight:
        db.session.delete(flight)
db.session.commit()         flash('Flight removed
successfully!', 'success')     else:
        flash('Flight not found.', 'danger')
return redirect(url_for('dashboard'))

# Define a route for viewing booked flights
@app.route('/view_bookings')
@login_required def
view_bookings():
    # Fetch the user's booked flights     booked_flights =
Booking.query.filter_by(user_id=current_user.id).all()     return
render_template('view_bookings.html', booked_flights=booked_flights)

# Define a route for viewing all flights
@app.route('/booked_flight'
) @login_required def
booked_flight():
    # Fetch all the flights booked by all users     all_booked_flights =
Booking.query.all()     return render_template('booked_flight.html',
all_booked_flights=all_booked_flights)

@app.route('/') def home():     return
"Welcome to the Flight Booking App!" if
__name__ == '__main__':     with
app.app_context():         db.create_all()
    app.run(debug=True)
```

Step 4: Define the required models

```
# models.py

from flask_sqlalchemy import SQLAlchemy
from datetime import
datetime from app import db
class Flight(db.Model):
    id = db.Column(db.Integer, primary_key=True)     flight_number =
db.Column(db.String(10), unique=True, nullable=False)     departure_city
= db.Column(db.String(50), nullable=False)     arrival_city =
db.Column(db.String(50), nullable=False)     departure_time =
db.Column(db.DateTime, default=datetime.utcnow)
    # Add more fields as needed
class Booking(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    user_id = db.Column(db.Integer, db.ForeignKey('user.id'), nullable=False)
flight_id = db.Column(db.Integer, db.ForeignKey('flight.id'), nullable=False)
booking_date = db.Column(db.DateTime, default=datetime.utcnow)
    # Add more fields as needed
```

Step 5: Design the templates required

registration.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Registration</title>
</head>
<body>
   <h1>Registration</h1>
   <form method="POST" action="{{ url_for('register') }}">
      {{ form.hidden_tag() }}
      <label for="username">Username:</label>
      {{ form.username(class="form-control") }}
      <label for="password">Password:</label>
      {{ form.password(class="form-control") }}
      {{ form.submit(class="btn btn-primary") }}
   </form>
   <p>Already have an account? <a href="{{ url_for('login') }}">Login</a></p>
</body>
</html>
```

login.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Login</title>
</head>
<body>
   <h1>Login</h1>
   <form method="POST" action="{{ url_for('login') }}">
      {{ form.hidden_tag() }}
      <label for="username">Username:</label>
      {{ form.username(class="form-control") }}
      <label for="password">Password:</label>
      {{ form.password(class="form-control") }}
      {{ form.submit(class="btn btn-primary") }}
   </form>
   <p>Don't have an account? <a href="{{ url_for('register') }}">Register</a></p>
</body>

</html>
```

dashboard.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Dashboard</title>
```

```
</head>
<body>
   <h1>Welcome to Your Dashboard, {{ current_user.username }}!</h1>
   <p>Your Bookings:</p>
   <ul>
      {% for booking in bookings %}
         <li>{{ booking.flight.departure_city }} to {{ booking.flight.arrival_city }}</li>
     {% endfor %}
   </ul>
   <p><a href="{{ url_for('logout') }}">Logout</a></p>
</body>
</html>
```

book_flight.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Book a Flight</title>
</head>
<body>
   <h1>Book a Flight</h1>
   <form method="POST" action="{{ url_for('book_flight', flight_id=flight.id) }}">
      {{ form.hidden_tag() }}
      <label>Flight Information:</label>
      <p>Flight Number: {{ flight.flight_number }}</p>
      <p>Departure City: {{ flight.departure_city }}</p>
      <p>Arrival City: {{ flight.arrival_city }}</p>
      <label for="booking_date">Booking Date:</label>
      {{ form.booking_date(class="form-control") }}
      {{ form.submit(class="btn btn-primary") }}
   </form>
</body>

</html>
```

add_flight.html

```
<!DOCTYPE html>
<html>
<head>
   <title>Add a Flight</title>
</head>
<body>
   <h1>Add a Flight</h1>
   <form method="POST" action="{{ url_for('add_flight') }}">
      {{ form.hidden_tag() }}
      <label for="flight_number">Flight Number:</label>
      {{ form.flight_number(class="form-control") }}
      <label for="departure_city">Departure City:</label>
      {{ form.departure_city(class="form-control") }}
      <label for="arrival_city">Arrival City:</label>
      {{ form.arrival_city(class="form-control") }}
```

```
        <label for="departure_time">Departure Time:</label>
        {{ form.departure_time(class="form-control") }}
        {{ form.submit(class="btn btn-primary") }}
    </form>
</body>
</html>
```

booked_flight.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Your Booked Flights</title>
</head>
<body>
    <h1>Your Booked Flights</h1>
    <ul>
        {% for booking in bookings %}
            <li>{{ booking.flight.departure_city }} to {{ booking.flight.arrival_city }}</li>
        {% endfor %}
    </ul>
    <p><a href="{{ url_for('dashboard') }}">Back to Dashboard</a></p>
</body>
</html>
```

view_bookings.html

```
<!DOCTYPE html>
<html>
<head>
    <title>View Bookings for Flight</title>
</head>
<body>
    <h1>View Bookings for Flight {{ flight.flight_number }}</h1>
    <p>Departure City: {{ flight.departure_city }}</p>
    <p>Arrival City: {{ flight.arrival_city }}</p>
    <p>Departure Time: {{ flight.departure_time }}</p>
    <h2>Bookings:</h2>
    <ul>
        {% for booking in bookings %}
            <li>{{ booking.user.username }} (Booking Date: {{ booking.booking_date }})</li>
        {% endfor %}
    </ul>
    <p><a href="{{ url_for('dashboard') }}">Back to Dashboard</a></p>
</body>
</html>
```

Step 6: Create Database Tables: Within

the Python shell type as

```
from app import db
db.create_all() exit()
```

If Mysql is used create table as such.,

```
CREATE TABLE flights (  flight_id INT
  AUTO_INCREMENT PRIMARY KEY,  flight_number
  VARCHAR(10) NOT NULL,  source VARCHAR(50)
  NOT NULL,  destination VARCHAR(50) NOT NULL,
  date DATE NOT NULL,  time TIME NOT NULL,
   seats_available INT NOT NULL
);
```

Step 7: Create __init__.py to import SQLALchemy and its dependencies

__init__.py

```
# app/__init__.py from
flask import Flask
from flask_sqlalchemy import SQLAlchemy


def create_app():
   app = Flask(__name__)
   app.config['SECRET_KEY']                    =                    'your-secret-key'
app.config['SQLALCHEMY_DATABASE_URI'] = 'sqlite:///flight_booking.db'
db.init_app(app)     return app app = create_app()
```

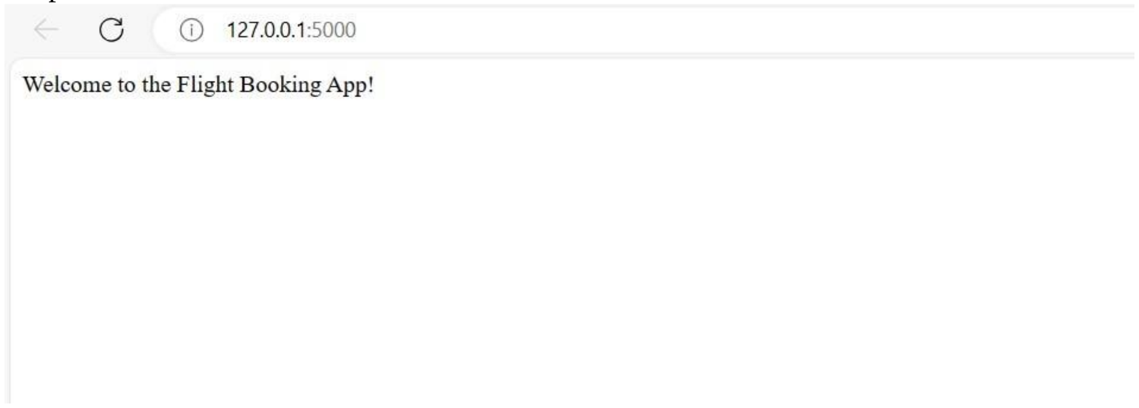
Step 8: Structure the flask application as follows

```
project_folder/
   venv/ (virtual environment)
   app/
         __init__.py
         models.py routes.py
   templates/ registration.html
         login.html
         dashboard.html
         book_flight.html
         add_flight.html
         booked_flights.html
         view_bookings.html
   app.py  requirements.txt
```

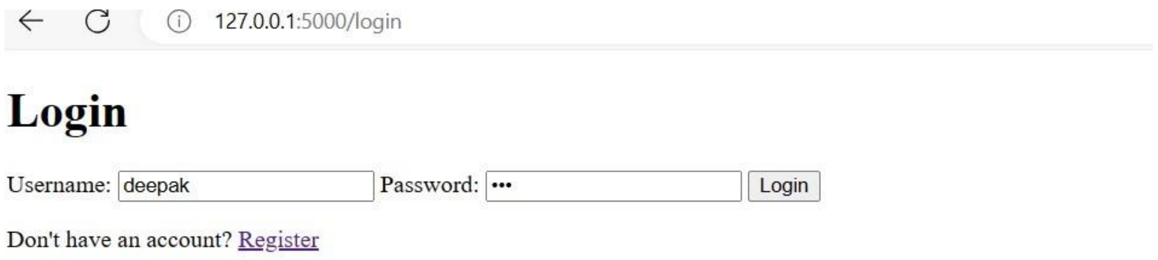
Step 9: Run the application

```
(new_venv3) C:\node-example\flight_booking_app>python app.py C:\node-
example\flight_booking_app\new_venv3\Lib\site-
```

```
packages\flask_sqlalchemy\__init__.py:872:    FSADeprecationWarning:
SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be
disabled by default in the future.  Set it to True or False to suppress this warning.
 warnings.warn(FSADeprecationWarning(
* Serving Flask app 'app' (lazy loading)
* Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
C:\node-example\flight_booking_app\new_venv3\Lib\site-
packages\flask_sqlalchemy\__init__.py:872:    FSADeprecationWarning:
SQLALCHEMY_TRACK_MODIFICATIONS adds significant overhead and will be
disabled by default in the future.  Set it to True or False to suppress this warning.
 warnings.warn(FSADeprecationWarning(
* Debugger is active!
* Debugger PIN: 795-665-397
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
```
Output:



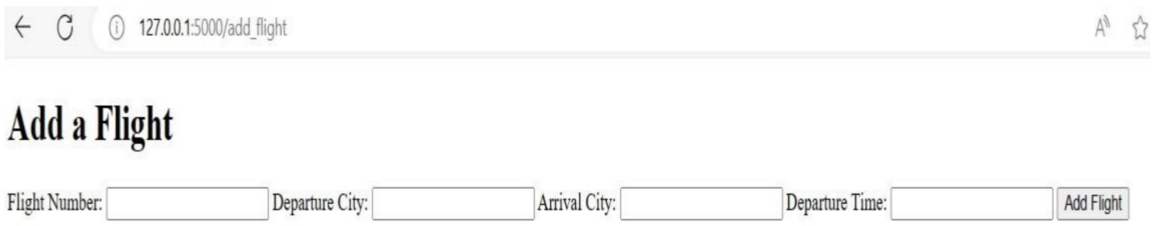
Welcome to the Flight Booking App!



Login

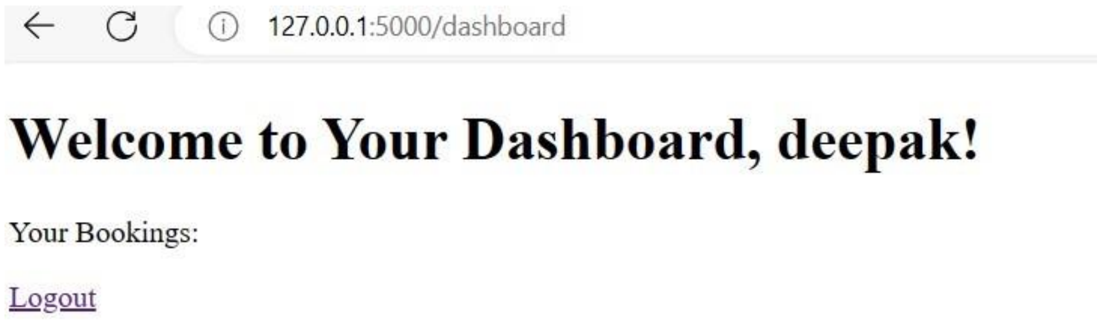
Username: deepak Password: ••• Login

Don't have an account? Register


127.0.0.1:5000/add_flight

Add a Flight

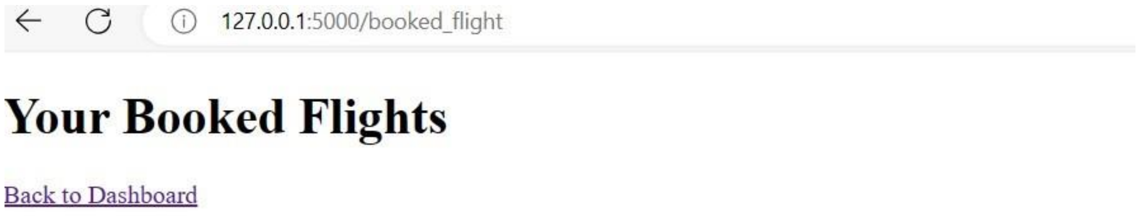
Flight Number: Departure City: Arrival City: Departure Time: Add Flight


127.0.0.1:5000/dashboard

Welcome to Your Dashboard, deepak!

Your Bookings:

Logout


127.0.0.1:5000/booked_flight

Your Booked Flights

Back to Dashboard

Conclusion:

This project is created using a Flask application with a simple flight booking form. When a user submits the form, the booking details are saved to a SQLite database. This implementation can be adapted and extended to meet any additional specific requirements, including payment integration, ticket generation, and confirmation emails. Additionally, it would be needed to create HTML templates for the form and booking confirmation pages.

This is a simplified example, and a production-grade flight booking system would require more extensive features, including user authentication, flight availability checks, and robust payment processing.

This project includes a simple flight booking system with user registration and booking credentials.

Expt.No:4	An online bookstore where books and reviews are stored in XML documents
Date: 07/09/23	

Aim:

To create a web application for online bookstore where books and reviews are stored in XML documents.

Project Components and Requirements:

1. HTML Template: This html template is the webpage to display the data stored in the xml file
2. XML Document: This file contains the data in XML format.
3. XSL: This file contains the Styles necessary for the xml document.
4. Livee Server: To view the output by hosting the html in live server. To install use npm install -g live-server

Execution Steps:

Step 1: Set Up the Project Structure

Create a directory for your project open a code editor in that file

path. Step 2: Create a books.xsl file books.xml

```
<books>
 <book>
   <title>Book 1</title>
   <author>Author 1</author>
   <price>20.00</price>
   <reviews>
     <review>Great book</review>
     <review>Must-read!</review>
  </reviews>
 </book>
 <book>
   <title>Book 2</title>
   <author>Author 2</author>
   <price>30.00</price>
   <reviews>
     <review>Superb</review>
     <review>Rare-find!</review>
  </reviews>
 </book>
 <book>
   <title>Book 3</title>
```

```
    <author>Author 3</author>
    <price>60.00</price>
    <reviews>
      <review>Adventurious</review>
      <review>Get ur hands on it asap</review>
    </reviews>
  </book>
</books>
```

Step3: Create a books.xsl file books.xsl

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:template match="/">
    <html>
      <head>
        <title>Bookstore</title>
      </head>
      <body>
        <h2>Now Trending</h2>
        <xsl:apply-templates select="//book"/>
      </body>
    </html>
 </xsl:template>

 <xsl:template match="book">
    <div class="book">
      <h2><xsl:value-of select="title"/></h2>
      <p>Author: <xsl:value-of select="author"/></p>
      <p>Price: $<xsl:value-of select="price"/></p>
      <p>Reviews:</p>
      <ul>
       <xsl:apply-templates
select="reviews/review"/>  </ul>    </div>
 </xsl:template>

 <xsl:template match="review">
    <li><xsl:value-of select="."/></li>
  </xsl:template>
</xsl:stylesheet>
```

Step 4: Create index.html to view the xml data.

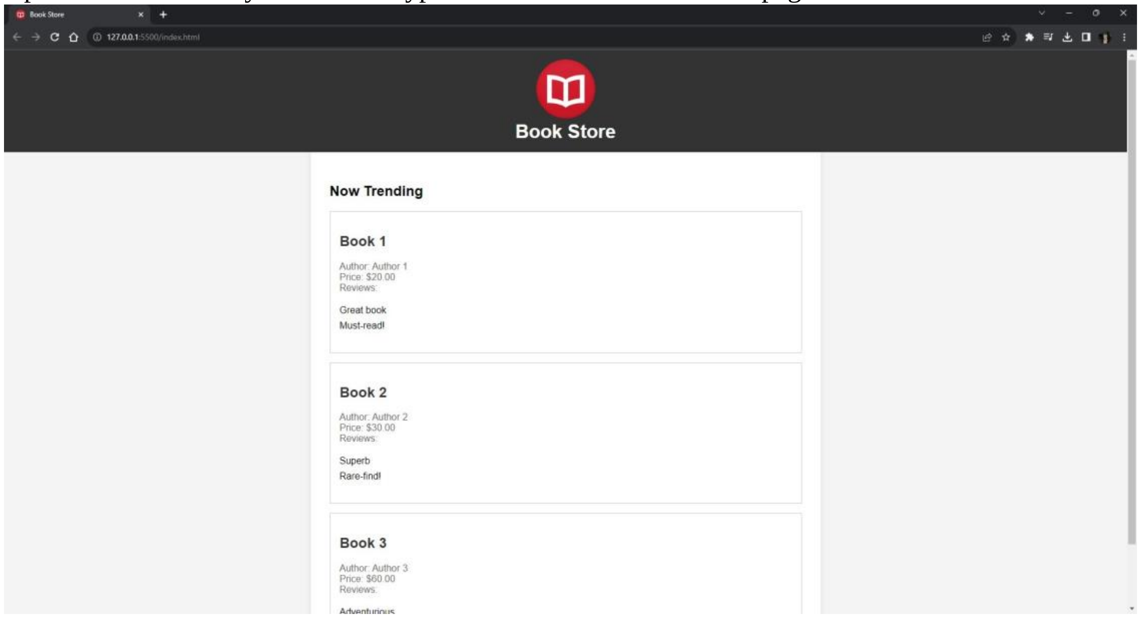
Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <link rel="icon" type="image" href="Book.png">
    <title>Book Store</title>
    <style>
      body {
           font-family: Arial, sans-serif;
          background-color: #f4f4f4;
          margin: 0;
           padding: 0;
       }

       header {
           background-color:
          #333;  color: #fff;  text-
          align: center;
           padding: 1rem;
       }

       h1 {
           margin: 0;
       }

       .container {  max-width:
          800px;  margin: 0 auto;
          padding: 2rem;
          background-color: #fff;
           box-shadow: 0 0 10px rgba(0, 0, 0, 0.1);
       }

       .book {  border: 1px
          solid #ccc;  padding:
          1rem;  margin-bottom:
          1rem;
           background-color: #fff;
       }

       .book  h2  {    color:
          #333;     font-size:
          1.5rem;
       }

       .book p {
           color: #666;
           margin: 0;
       }

       .book ul {  list-style-
          type: none;
           padding: 0;
       }

       .book li {  padding:
          0.25rem 0;
       }
    </style>
</head>
<body>
    <header>
       <img src="Book.png" width="100" height="100">
       <h1>Book Store</h1>
    </header>
    <div class="container">
      <div id="books"></div>
    </div>
    <script>  function
      loadBooks() {
           const xmlhttp = new XMLHttpRequest();
          xmlhttp.onreadystatechange = function () {  if
          (xmlhttp.readyState === 4 && xmlhttp.status === 200) {
                 const xml = xmlhttp.responseXML;
                const xsl = new XMLHttpRequest();
                xsl.onreadystatechange = function () {
                    if (xsl.readyState === 4 && xsl.status === 200) {
                        const xsltProcessor = new XSLTProcessor();
                       xsltProcessor.importStylesheet(xsl.responseXML);  const result =
                       xsltProcessor.transformToFragment(xml, document);
                       document.getElementById('books').appendChild(result);
                    }
                };
                 xsl.open('GET', 'books.xsl', true);
                xsl.send();
              }
           };
           xmlhttp.open('GET', 'books.xml', true);
          xmlhttp.send();
       }
       loadBooks();
    </script>
</body>
</html>
```

Run the application

Output:

Open the file directory in cmd and type live-server to host the html page in localhost.



Book Store

127.0.0.1:5500/index.html

Book Store

Now Trending

Book 1

Author: Author 1
Price: $20.00
Reviews:

Great book
Must-read!

Book 2

Author: Author 2
Price: $30.00
Reviews:

Superb
Rare-find!

Book 3

Author: Author 3
Price: $60.00
Reviews:

Adventurious

Conclusion:

This application displays the XML document data in html by using XMLHttpRequest.
The data can be viewed as separate products and with reviews. Users can check on the books present for sale.

Expt.No:5	A polling application that allows users to create and participate in polls using Django
Date: 22/09/23	

Aim:

To build a polling application that allows users to create and participate in polls using Django.

Project Components and Requirements:

1. Python and Django: Ensure you have Python installed on your system. Also need Django, which can be installed using pip.

2. Database: Decide on the SQL database want to use (e.g., SQLite, PostgreSQL, MySQL) and ensure it is installed and running.

3. Django App: Create a Django project and an app within the project where your URL shortener functionality will reside.

4. Models: Define a Django model to represent URLs. This model should include fields for the original (long) URL, the shortened URL, and any other relevant information you want to store.

5. Views: Create views for your URL shortener application. This views is needed for the home page, URL submission form, and redirection. These views should interact with your model to retrieve and store URLs.

6. Templates: Design HTML templates for your application's pages, including the home page, URL submission form, and redirection page.

7. URL Routing: Configure URL patterns in your app's urls.py file to route requests to the appropriate views.

8. Forms: Create a Django form for the URL submission form. This form should validate and process user input.

9. Shortening Algorithm: Implement a URL shortening algorithm. Use base62 or base36 encoding, random character generation, or a combination of methods to generate short URLs.

10. Database Configuration: Configure the Django project settings to use the SQL database of your choice (e.g., SQLite or PostgreSQL). Update the DATABASES setting in settings.py.

11. Migrations: Run migrations to create the database tables based on your models.

12. Static and Media Files: Configure static and media file handling in your Django project settings if you plan to use CSS, JavaScript, or store user-uploaded files.

13. User Interface: Design and style the application's user interface using CSS or a frontend framework if desired.

Execution Steps:

Step 1: Create an empty folder poller_project in your directory.
Step 2: Now switch to your folder and create a virtual environment in this folder using the following command.

```
python -m venv myenv

myenv\Scripts\activate
```

Step 3: A Pipfile will be created in your folder from the above step. Now install Django and SQLAlchemy in your folder using the following command.

```
pip install Django  pip

install SQLAlchemy
```

Step 4: Now we need to establish the Django project. Run the following command in your folder and initiate a Django project. django-admin startproject poller

A New Folder with name poller will be created. Switch to the poller folder using the following command. cd poller

Here you can start the server using the following command and check if the application running or not using your http://127.0.0.1:8000/ in your browser. python manage.py runserver

Step 5: Create an app ‘polls‘ using the following command

```
python manage.py startapp polls
```

Below is the folder structure after creating ”polls’ app in the project.

polls-app

Create Models:

Step 1: In your models.py file write the code given below to create two tables in your database. One is ‘Question‘ and the other one is ‘Choice‘. ‘Question’ will have two fields of ‘question_text’ and a ‘pub_date’. Choice has three fields: ‘question’, ‘choice_text’, and ‘votes’. Each Choice is associated with a Question.

```
from django.db import models

# Create your models here.

class Question(models.Model):
    question_text = models.CharField(max_length = 200)
   pub_date = models.DateTimeField('date published')
```

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


class Choice(models.Model):
    question = models.ForeignKey(Question, on_delete =
   models.CASCADE)  choice_text = models.CharField(max_length = 200)
   votes = models.IntegerField(default = 0)

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

Step 2:Go to the settings.py file and in the list, INSTALLED_APPS write down the code below to include the app in our project. This will refer to the polls -> apps.py -> PollsConfig class.

```
INSTALLED_APPS = [
    'polls.apps.PollsConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
   'django.contrib.staticfiles',
]
```

Step 3: We have made changes in our database and created some tables but in order to reflect these changes we need to create migration here and then Django application will stores changes to our models. Run the following command given below to create migrations.

```
python manage.py makemigrations polls
```

Inside polls->migrations a file 0001_initial.py will be created where you can find the database tables which we have created in our models.py file. Now to insert all the tables in our database run the command given below… python manage.py migrate

Create an Admin User:

Step 1: Run the command given below to create a user who can login to the admin site.

```
python manage.py createsuperuser
```

It will prompt username which we need to enter.

Username: Abhi
Now it will prompt an email address which again we need to enter here.

Email address: Abhi@example.com
The final step is to enter the password. We need to enter the password twice, the second time as a confirmation of the first.

Password: ******
Password (again): ******
Superuser created successfully.
Now we can run the server using the same command python manage.py runserver and we can check our admin panel browsing the URL http://127.0.0.1:8000/admin .

Step-2: In the admin.py file we will write the code given below to map each question with choices to select. Also, we will write the code to change the site header, site title, and index_title. Once this is done we can add questions and choices for the question from the admin panel.

```
from django.contrib import admin #
Register your models here. from
.models import Question, Choice

# admin.site.register(Question)
# admin.site.register(Choice)

admin.site.site_header = "Poller Admin"
admin.site.site_title = "Poller Admin Area"
admin.site.index_title = "Welcome to the Poller Admin Area"


class ChoiceInLine(admin.TabularInline):
    model = Choice
   extra = 3


class QuestionAdmin(admin.ModelAdmin):
    fieldsets = [(None, {'fields': ['question_text']}), ('Date Information', {
        'fields':    ['pub_date'],    'classes':    ['collapse']}),    ]
   inlines = [ChoiceInLine] admin.site.register(Question,
   QuestionAdmin) question-choices
```

Note: We can test the application here by adding some questions and choices for those questions.

Create Views
Now we will create the view of our application that will fetch the data from our database and will render the data in the ‘template‘ (we will create ‘template’ folder and the files inside this folder in the next section) of our application to display it to the user.

Step 1 Open views.py file and write down the code given below.

```
from django.template import loader from django.http import
HttpResponse, HttpResponseRedirect from django.shortcuts
import get_object_or_404, render from django.urls import
reverse from .models import Question, Choice # Get
questions and display them
```

```
def index(request):
     latest_question_list = Question.objects.order_by('-
    pub_date')[:5]  context = {'latest_question_list':
    latest_question_list}  return render(request, 'polls / index.html',
    context)

# Show specific question and choices


def detail(request, question_id):
   try:
        question = Question.objects.get(pk = question_id)
    except Question.DoesNotExist:
        raise Http404("Question does not exist")
     return render(request, 'polls / detail.html', {'question': question})

# Get question and display results


def results(request, question_id):
     question = get_object_or_404(Question, pk = question_id)
    return render(request, 'polls / results.html', {'question':
    question})

# Vote for a question choice
def vote(request, question_id):  #
   print(request.POST['choice'])  question =
   get_object_or_404(Question, pk = question_id)  try:
        selected_choice = question.choice_set.get(pk = request.POST['choice'])
    except (KeyError, Choice.DoesNotExist):
        # Redisplay the question voting form.
        return render(request, 'polls / detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
       selected_choice.save()
        # Always return an HttpResponseRedirect after successfully dealing  #
       with POST data. This prevents data from being posted twice if a  # user
       hits the Back button.  return
       HttpResponseRedirect(reverse('polls:results', args =(question.id, )))
```

Step 2: Create a file urls.py inside the poller->polls folder to define the routing for all the methods we have implemented in views.py file (don’t get confused with the file inside the poller->poller->urls.py file). Below is the code of urls.py file…

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

```
app_name = 'polls'
urlpatterns = [
    path('', views.index, name ='index'),
    path('<int:question_id>/', views.detail, name ='detail'),
   path('<int:question_id>/results/', views.results, name ='results'),
   path('<int:question_id>/vote/', views.vote, name ='vote'),
]
```

Create Templates

Step 1: Follow the steps given below to create the front layout of the page.

Create a folder ‘templates‘ in top-level poller folder (alongside of polls and poller) i.e. poller> templates.
Create ‘base.html‘ file inside the template folder. We will define the head, body and navigation bar of our application in this file.
In the ‘templates’ folder create another folder ‘polls‘. In ‘polls’ folder create three files ‘index.html‘, ‘results.html‘ and ‘detail.html‘.
The folder structure will look like the image given below (we have highlighted the files which we have created in ‘create views i.e urls.py’ and ‘create template’ section)…
templates

Step 2: By default Django will search the ‘template’ inside the ‘polls’ app but we have created a global ‘template’ folder which is outside the polls app. So in order to make it work, we need to define the ‘template’ folder path inside the settings.py file. Open settings.py file and add the code given below in the list ‘TEMPLATES’. In order to make the given code work add “import os” in settings.py.

```
TEMPLATES = [
    {
        # make changes in DIRS[].
        'BACKEND': 'django.template.backends.django.DjangoTemplates',
        'DIRS': [os.path.join(BASE_DIR, 'templates')],
        'APP_DIRS': True,
        'OPTIONS': {
            'context_processors': [
                'django.template.context_processors.debug',
                'django.template.context_processors.request',
                'django.contrib.auth.context_processors.auth',
                'django.contrib.messages.context_processors.messages',
            ],
        },
    }, ]
```

Step 3: Open index.html file and write the code given below. This file will display the list of questions which are stored in our database. Also, two buttons will be displayed to the user. One for the voting (we will create a detail.html file for voting) and the other one is to check the results (we will create results.html file for results).

```
{% extends 'base.html' %}
{% block content %}
<h1 class="text-center mb-3">Poll Questions</h1>
{% if latest_question_list %}
{% for question in latest_question_list %}
<div class="card-mb-3">
   <div class="card-body">
      <p class="lead">{{ question.question_text }}</p>
      <a href="{% url 'polls:detail' question.id %}" class="btn btn-primary btn-sm">Vote
Now</a>
      <a href="{% url 'polls:results' question.id %}" class="btn btn-secondary
btnsm">Results</a>
   </div>
</div>
{% endfor %}
{% else %}
<p>No polls available</p>
{% endif %}
{% endblock %}
```

Step 4: Open detail.html file and write the code given below. This file will be responsible for voting on specific questions. Whatever question a user will select for voting from the list of the question (index.html file), that specific question and the choices for the question will be displayed on this page. A user will be allowed to select one choice and give voting by clicking on the vote button.

```
{% extends 'base.html' %}
{% block content %}
<a class="btn btn-secondary btn-sm mb-3" href="{% url 'polls:index' %}">Back To
Polls</a>
<h1 class="text-center mb-3">{{ question.question_text }}</h1>

{% if error_message %}
<p class="alert alert-danger">
   <strong>{{ error_message }}</strong>
</p>
{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
   {% csrf_token %}
   {% for choice in question.choice_set.all %}
   <div class="form-check">
      <input type="radio" name="choice" class="form-check-input" id="choice{{
forloop.counter }}" value="{{ choice.id }}" />
      <label for="choice{{ forloop.counter }}">{{ choice.choice_text }}</label>
   </div>
   {% endfor %}
   <input type="submit" value="Vote" class="btn btn-success btn-lg btn-block mt-4" />
</form>
{% endblock %}
```

Step 5: Open results.html file and write the code given below. This file will display the result of total votes on a specific question whatever question the user will select (from the index.html file) to check the result.

```
{% extends 'base.html' %}
{% block content %}
<h1 class="mb-5 text-center">{{ question.question_text }}</h1>

<ul class="list-group mb-5">
   {% for choice in question.choice_set.all %}
   <li class="list-group-item">
      {{ choice.choice_text }} <span class="badge badge-success float-right">{{
      choice.votes
}}  vote{{ choice.votes | pluralize }}</span>
  </li>
   {% endfor %}
</ul>

<a class="btn btn-secondary" href="{% url 'polls:index' %}">Back To Polls</a>
<a class="btn btn-dark" href="{% url 'polls:detail' question.id %}">Vote again?</a>
{% endblock %}
```

Step 6: Let's create the navigation bar for our application. Create a folder 'partials' inside the folder 'templates' and then create a file '_navbar.html' inside the 'partial' folder. File structure will be templates->partials->_navbar.html. Write the code given below in this file.

```
<nav class="navbar navbar-dark bg-primary mb-4">
   <div class="container">
      <a class="navbar-brand" href="/">Poller</a>
   </div>
</nav>
```

Step 7: We haven't included the head and body tag in every single HTML file we have created till now. We can write these codes in just one single file base.html and we can give the layout to our page. We will also bring our navigation bar(_navbar.html file) on this page. So open base.html file inside the 'template' folder and write down the code given below.

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

<head>
 <link                                                        rel="stylesheet"
href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
      integrity="sha384-
Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
crossorigin="anonymous">
   <title>Poller {% block title %}{% endblock %}</title>
</head>
```

```
<body>
   <!--NavBar-->
   {% include 'partials/_navbar.html'%}
   <div class="container">
      <div class="row">
         <div class=".col-md-6 m-auto">
            {% block content %}{% endblock%}
         </div>
      </div>
   </div>
</body>

</html>
```

Create Landing Page

The URL http://127.0.0.1:8000/ should display a landing page for our web application. So to create a landing page we will follow the step given below.

Step 1 Switch to the top-level poller folder and run the command given below to create an app ‘pages‘.

```
python manage.py startapp pages
```

Below is the folder structure once the ‘pages’ app will be created.

pages

Step 2 Open ‘views.py‘ inside ‘pages’ folder i.e. pages->views.py. Write down the code given below to visit on landing page.

```
from django.shortcuts import render

# Create your views here.

def index(request):  return render(request,
  'pages / index.html')
```

Step 3 Create urls.py file inside the ‘pages’ folder i.e. pages->urls.py. Write the code given below to define the routing of pages->index.html file (check step-1).

```
from django.urls import path

from . import views

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

Step 4 Create a folder ‘pages‘ inside ‘template’ folder. Now inside ‘pages’ folder create a file index.html. Write down the code given below to display the landing page to the users.

```
{% extends 'base.html' %}
{% block content %}
<div class="card text-center">
   <div class="card-body">
      <h1>Welcome To Poller!</h1>
      <p> This is a Django polling app</p>
      <a class="btn btn-dark" href="{% url 'polls:index' %}">
        View Available Polls</a>
   </div>
</div>
{% endblock %}
```

Create routing inside the main urls.py file of the application

We have created two apps in our application ‘polls‘ and ‘pages‘. We need to define the routing of these two apps inside the main urls.py file which is poller->poller->urls.py file. So open the main urls.py file inside the poller folder and write down the code given below to define the routing of these two apps(‘polls’ and ‘pages’).

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

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

Testing of the Application
Admin Frontend
Step 1 Run the server using the command python manage.py runserver and browse the URL

http://127.0.0.1:8000/admin/. Now enter the username and password to login into the system.

Step 2 Click on ‘add’ button next to the ‘Questions’.

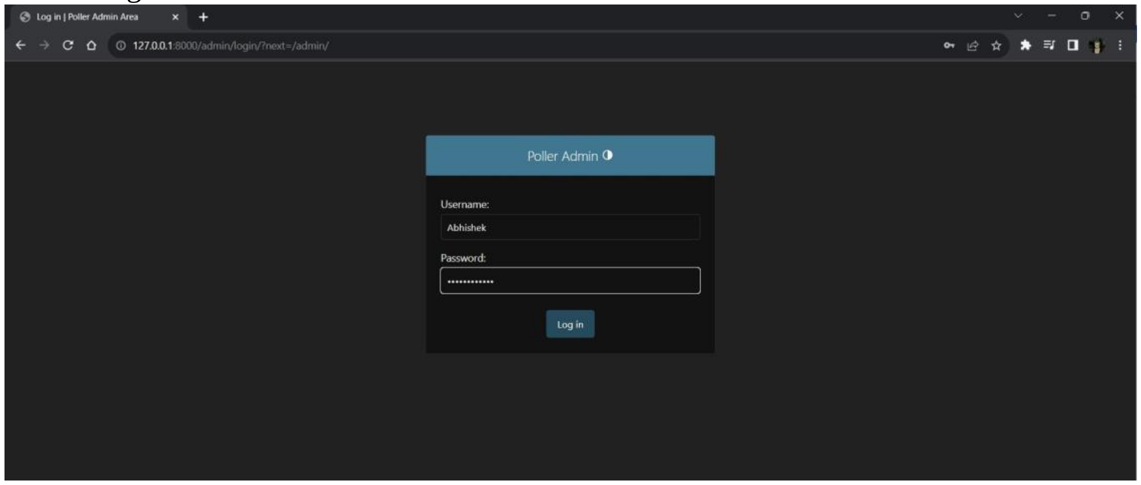
Step 3 Now add question and choices for those questions. Also, mention the date and time and then click on the ‘save’ button. You can add as many questions as you want. You will see a list of questions added in the database.

User Frontend

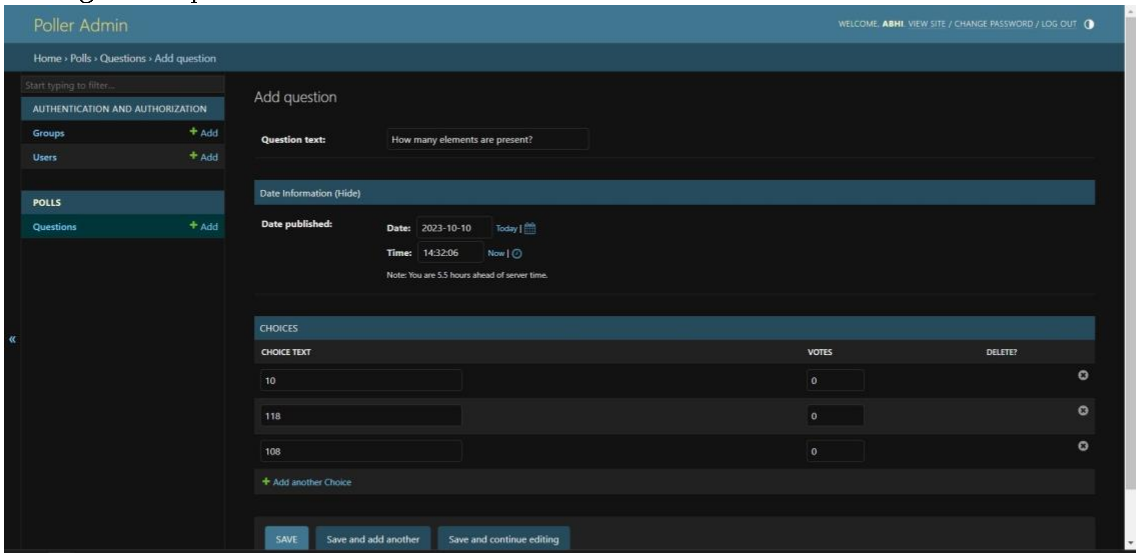
Step 1: Browse the URL http://127.0.0.1:8000/ and you will see the landing page of the application. Click on the “View Available Polls”

Step 2: You will see list of questions with two options ‘Vote Now’ and ‘Results’. From here you need to select one question and click on the ‘Vote Now’ button.
Step 3: Once this is done select any one choice and click on ‘Vote’ button. You can also go to the previous menu using the ‘Back to Polls’ button on the top.
Outputs:

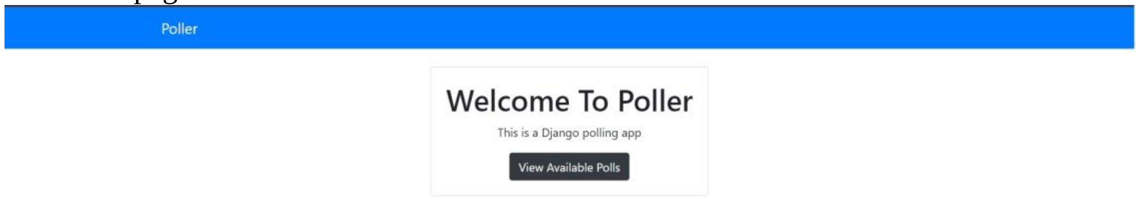
Admin Page:


Log in | Poller Admin Area
127.0.0.1:8000/admin/login/?next=/admin/
Poller Admin
Username:
Abhishek
Password:
••••••••••••
Log in

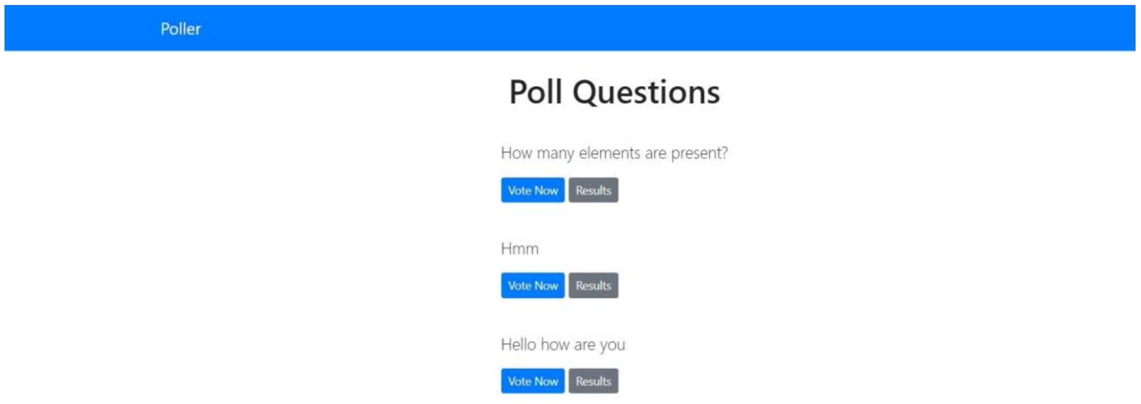
Adding a new question


Poller Admin
WELCOME, ABHI. VIEW SITE / CHANGE PASSWORD / LOG OUT
Home › Polls › Questions › Add question
Start typing to filter...
AUTHENTICATION AND AUTHORIZATION
Groups + Add
Users + Add
POLLS
Questions + Add
Add question
Question text: How many elements are present?
Date Information (Hide)
Date published: Date: 2023-10-10 Today | Time: 14:32:06 Now |
Note: You are 5.5 hours ahead of server time.
CHOICES
CHOICE TEXT VOTES DELETE?
10 0
118 0
108 0
+ Add another Choice
SAVE
Save and add another
Save and continue editing

The home page:



List of Question polls:


Poller

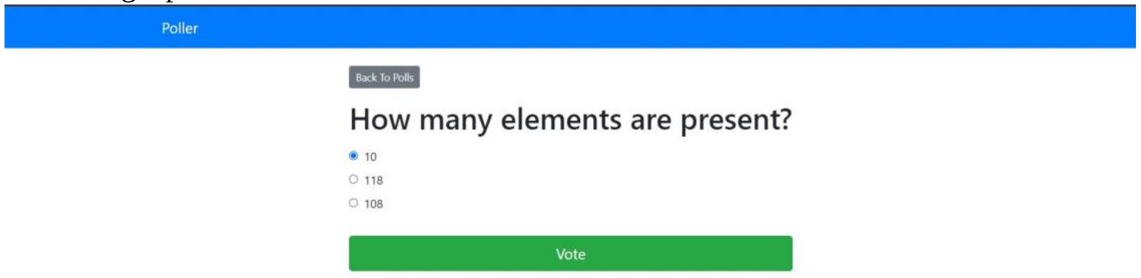
Poll Questions

How many elements are present?

Hmm

Hello how are you

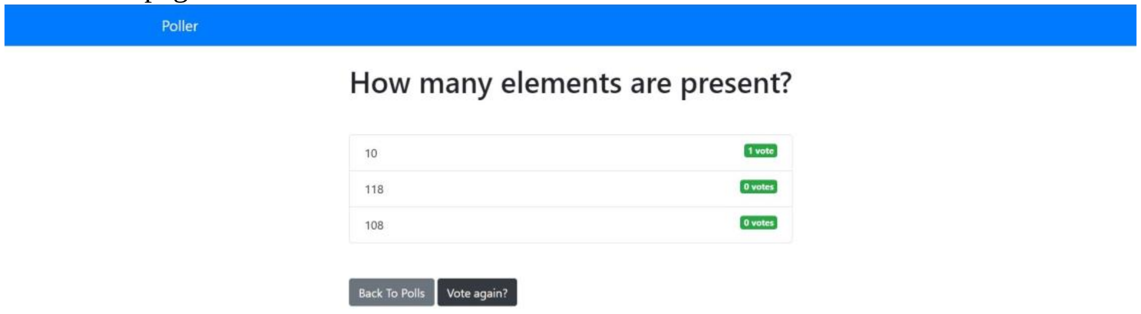
Answering a poll


Poller

How many elements are present?

- 10
- 118
- 108

The Result page:


Poller

How many elements are present?

10	1 vote
118	0 votes
108	0 votes

Conclusion:

The Poll Application using SQL and Django provides various polls users can fill in. It has a separate admin page to manage the questions. This app can be refined to use it in a more effitient manner.