CHENNAI INSTITUTE OF TECHNOLOGY

Sarathy Nagar, Kundrathur, Chennai-600069

An Autonomous Institute Approved by AICTE and Affiliated to Anna University, Chennai

DEPARTMENT OF INFORMATION TECHNOLOGY

IT3303 – WEB DEVELOPMENT FRAMEWORK AND PRACTICES

Department of Information Technology

By
Mithun M
Reg No:23IT092

October - 2024

CHENNAI INSTITUTE OF TECHNOLOGY
CHENNAI-69

Vision of the Institute:

To be an eminent centre for Academia, Industry and Research by imparting knowledge, relevant practices and inculcating human values to address global challenges through novelty and sustainability.

Mission of the Institute:

IM1.To creates next generation leaders by effective teaching learning methodologies and instill scientific spark in them to meet the global challenges.

IM2.To transform lives through deployment of emerging technology, novelty and sustainability.

IM3.To inculcate human values and ethical principles to cater the societal needs.

IM4.To contributes towards the research ecosystem by providing a suitable, effective platform for interaction between industry, academia and R & D establishments.

CHENNAI INSTITUTE OF TECHNOLOGY
(Autonomous)

Department of Information Technology

Vision of the Department:

To Excel in the emerging areas of Information Technology by imparting knowledge, relevant practices and inculcating human values to transform the students as potential resources to contribute innovatively through advanced computing in real time situations.

Mission of the Department:

DM1: To provide strong fundamentals and technical skills through effective teaching learning methodologies.

DM2: To transform lives of the students by nurturing ethical values, creativity and novelty to become Entrepreneurs and establish start-ups.

DM3: To habituate the students to focus on sustainable solutions to improve the quality of life and the welfare of the society.

DM4: To enhance the fabric of research in computing through collaborative linkages with industry and academia.

DM5: To inculcate learning of the emerging technologies to pursue higher studies leading to lifelong learning.

CHENNAI INSTITUTE OF TECHNOLOGY
Transforming Lives

CHENNAI INSTITUTE OF TECHNOLOGY

(Approved by AICTE, New Delhi & Affiliated to Anna University Chennai), Sarathy Nagar, Kundrathur, Chennai - 600069.

Register No:

BONAFIDE CERTIFICATE

Certified that this is the bonafide record of work done by Mr./Ms..... of ..... semester B.Tech. ..... in the .....Laboratory during the academic year .....

Staff-In-Charge **Head of the Department**

Submitted for the University Practical Examination held on.....

Internal Examiner **External Examiner**

Date: ..... **Date: .....**

INDEX

Expt.No:1	SIMPLE WEATHER APPLICATION
Date: 28/07/24	

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.

  Ensure you have Node.js installed by running the following commands in the terminal:

  node -v

  npm -v

  If you don't have Node.js, download and install it from https://nodcjs.org.

- Get OpenWeatherMap API Key

  Sign up on OpenWeatherMap

  https://home.openweathermap.org/users/sign_up

1. After creating your account, navigate to the API section to generate a new API key or use the default one provided. [NOTE: Ensure your email adt11'ess is verifietl, and allow a few hours for the l‹ey to becom e active.]
2. Note this API key, which will be needed for the app.

- Generate API key
- Create a Project Directory.

Create a directory for your weather app:

mkdir weather-app

cd weather-app

- Now install the required dependencies:

In the project directory, initialize a Node.js project:

npm init -y

This will create a package.json file. Now install the required dependencies:

npm install axios dotenv readline

- Create .env File for API key

  Create a file named ’.env" in your project directory:

  In the ”.env’ file, add your OpenWeatherMap API key like this:

  API_KEY

Replace ’your_openweathermap_api_key’ with your actual API key.

Step 6: Create weather.js

This file will contain the Node.js logic for fetching the weather data and serving the HTML page.

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) => {
  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>
```

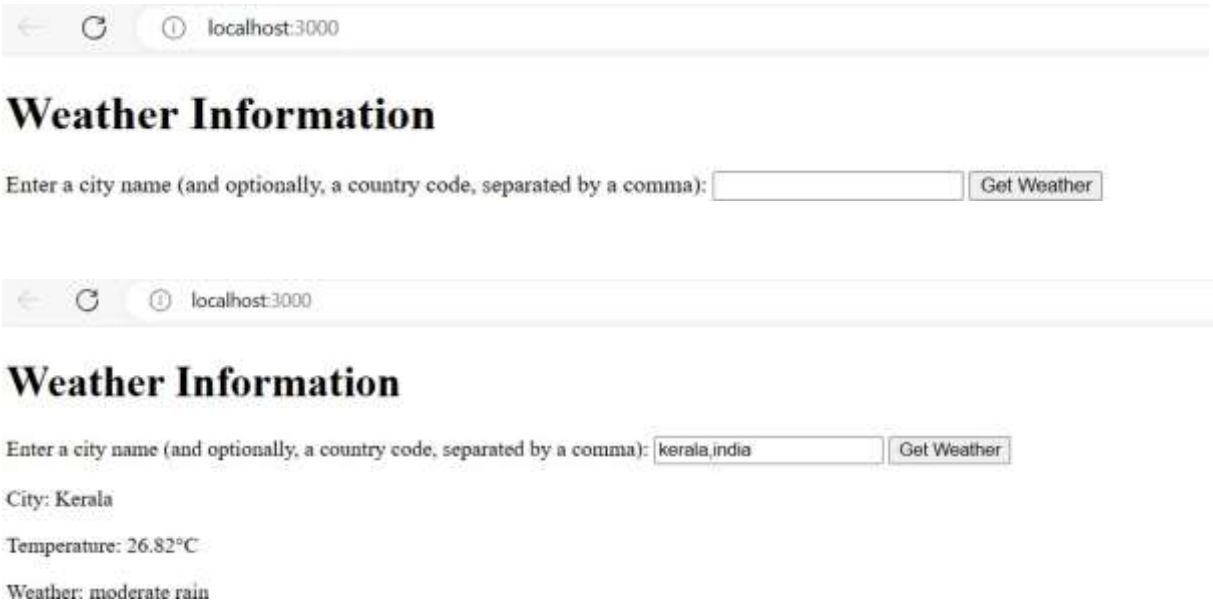
```
    </form>
    <!-- Weather information will be displayed here -->
    <div id="weather-info">
      <!-- Weather data will be displayed here -->
    </div>
    <script>
  const weatherInfoDiv = document.getElementById('weather-info');
  const locationInput = document.getElementById('location');

  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



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:10/08/24	

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 (Object-Relational 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)
```

8. Define utils:

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
```

9. Create Templates:

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:

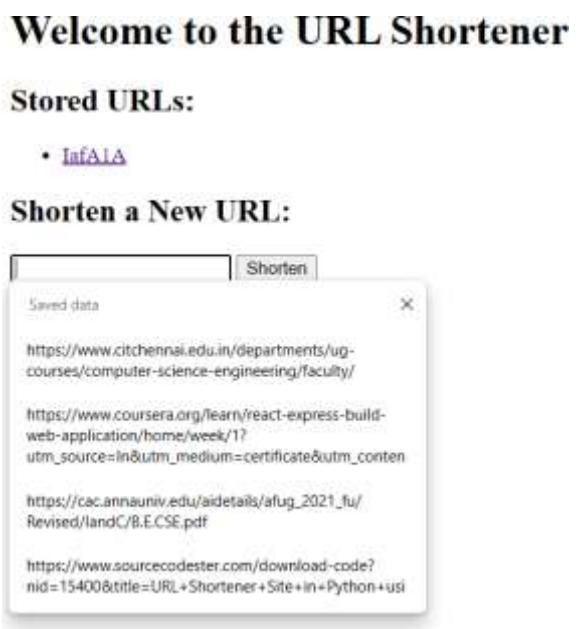
```
(myenv) C:\node-example\urlshortener>python manage.py makemigrations
No changes detected

(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:

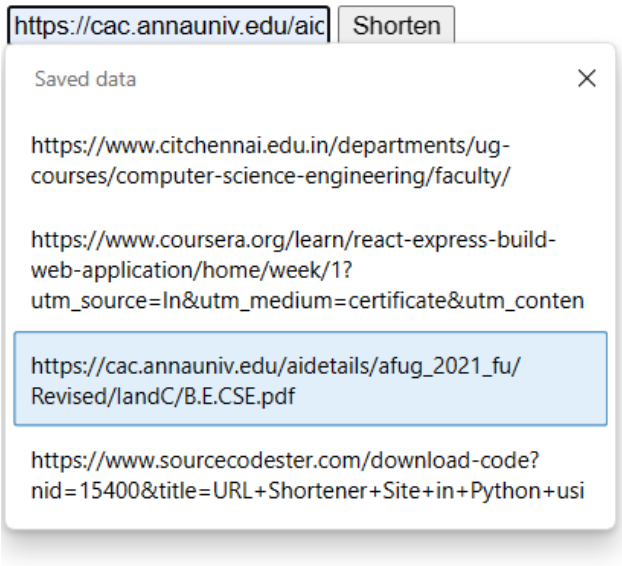


Welcome to the URL Shortener

Stored URLs:

- IafA1A
- XT6kQD

Shorten a New URL:



Welcome to the URL Shortener

Stored URLs:

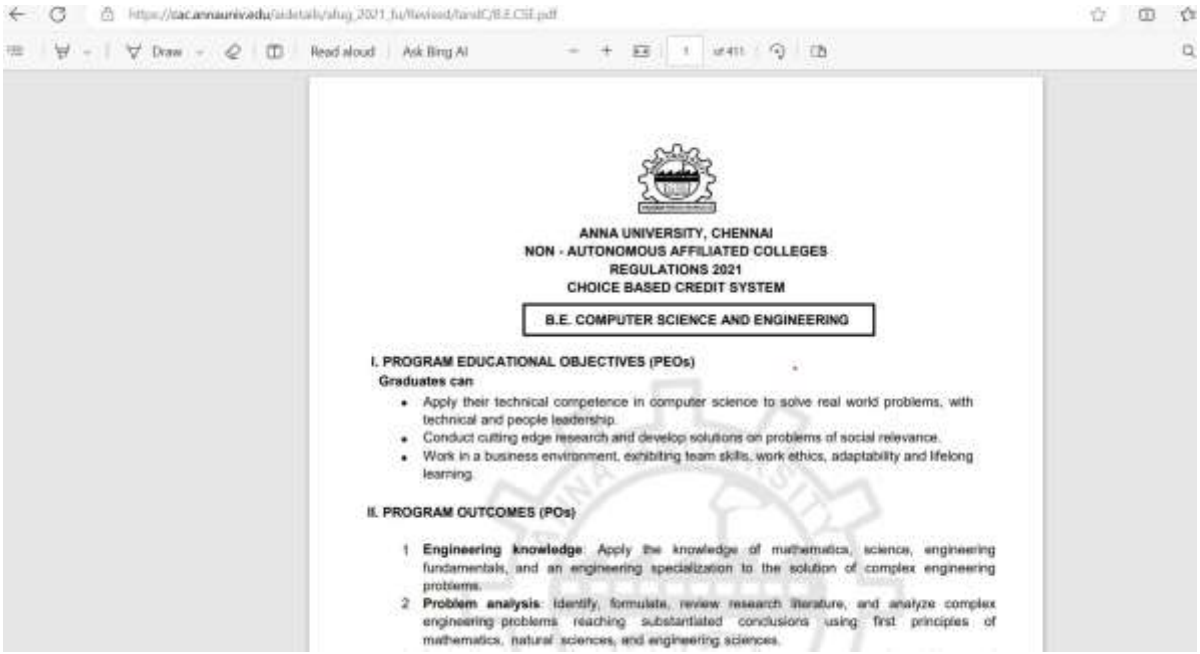
- IafA1A
- XT6kQD

Shorten a New URL:

Shorten

- After Clicking IafA1A

- After Clicking XT6kQD



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:24/08/24	

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

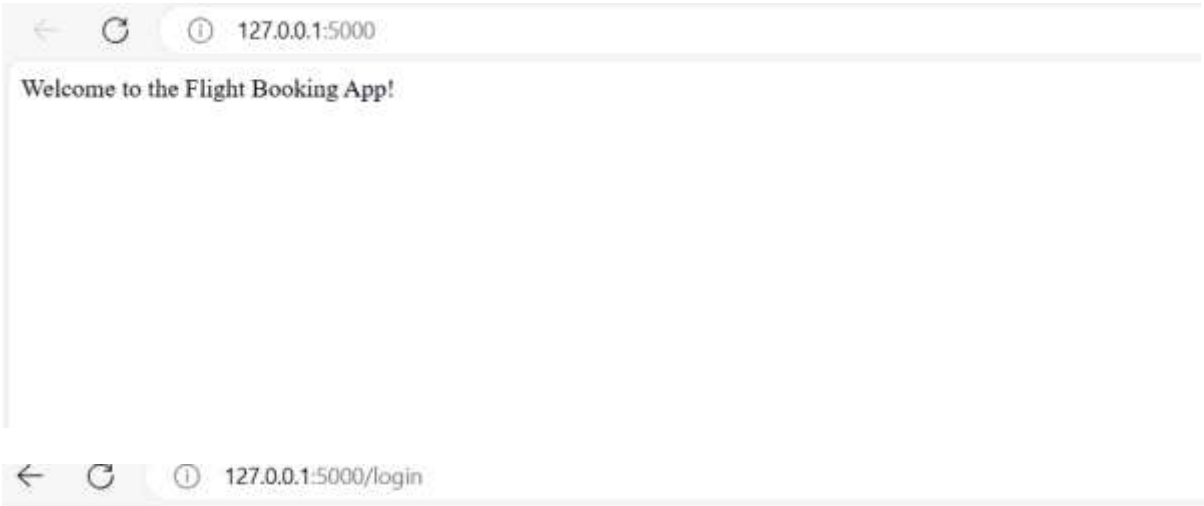
```
project_folder/
    venv/ (virtual environment)
    app/
        __init__.py
        models.py
        routes.py
    templates/
        registration.html
        login.html
        dashboard.html
        book_flight.html
        add_flight.html
        booked_flights.html
        view_bookings.html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

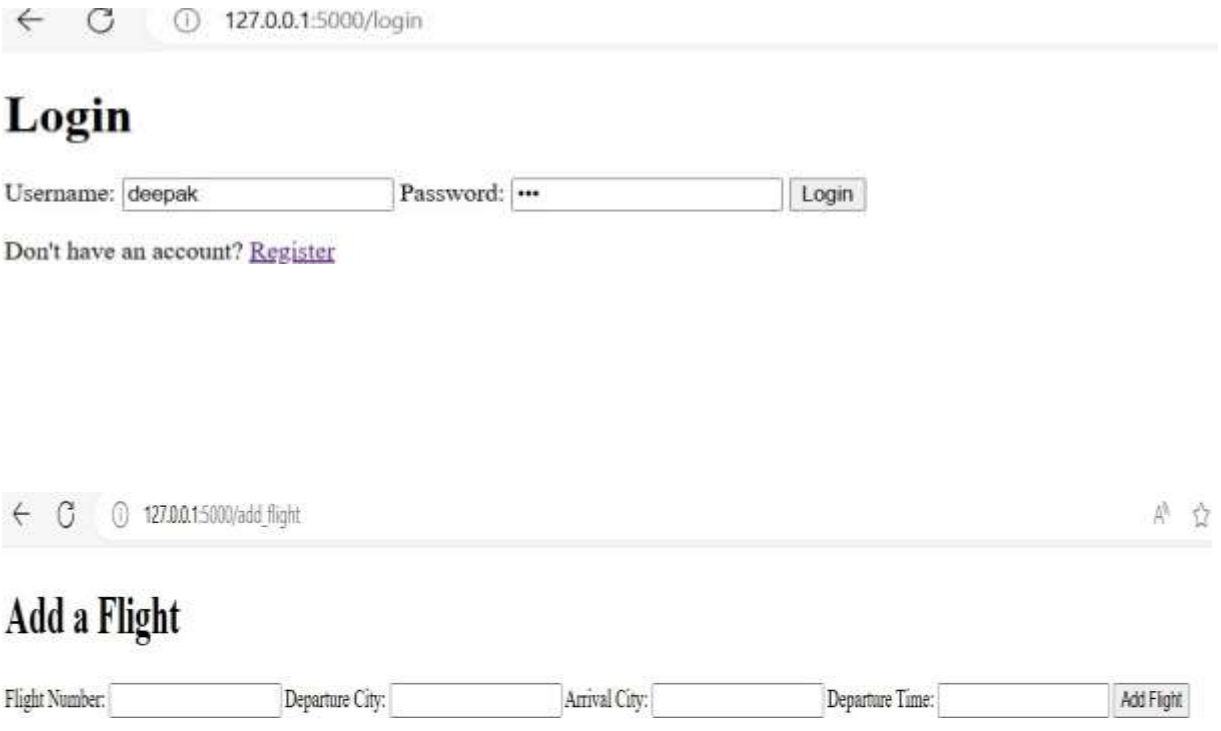
Step 9: Run the application

```
(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 Your Dashboard, deepak!

Your Bookings:

Logout

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.