Contact.html:

{% extends 'base.html' %}

{% block content %}

<div class="contact-section">

<h1>Get in Touch</h1>

<p>We're always here to help you on your journey. If you've got a question, a tale to tell, or you just want to say hi, drop us a line!</p>

<div class="contact-info">

<div class="contact-item">

<h2>Call Us</h2>

<p><i class="fas fa-phone"></i> +123 456 7890</p>

</div>

<div class="contact-item">

<h2>Email Us</h2>

<p><i class="fas fa-envelope"></i> hello@offbeat-travels.com</p>

</div>

<div class="contact-item">

<h2>Visit Us</h2>

<p><i class="fas fa-map-marker-alt"></i> 123 Explorer's Lane, Adventure City</p>

</div>

</div>

<form id="contactForm" action="/submit-contact" method="post" class="contact-form">

<div class="form-group">

<label for="name">Your Name:</label>

<input type="text" id="name" name="name" placeholder="Jane Doe" required>

</div>

<div class="form-group">

<label for="email">Your Email:</label>

<input type="email" id="email" name="email" placeholder="jane.doe@example.com" required>

</div>

<div class="form-group">

<label for="message">Your Message:</label>

<textarea id="message" name="message" placeholder="I would love to know more about your trips to the Sahara." rows="6" required></textarea>

</div>

<div class="form-group">

<button type="submit" class="btn-submit">Send Message</button>

</div>

</form>

</div>

{% endblock %}

button {

background-color: #4CAF50; /\* Green \*/

border: none;

color: white;

padding: 10px 20px;

text-align: center;

text-decoration: none;

display: inline-block;

font-size: 16px;

border-radius: 5px;

cursor: pointer;

}

Update\_booking\_form

<div class="content-container">

Starting your project with Flask involves a series of steps that will guide you from setting up your environment to deploying your application. Here’s a detailed step-by-step guide tailored for "Offbeat Travels," focusing on discovering unique travel destinations, integrating with MongoDB and MySQL, and ensuring a user-friendly interface.

Step 1: Environment Setup

Before diving into code, set up a proper Python development environment.

* Create a Virtual Environment: This isolates your project's libraries from the system. Use python -m venv offbeat-env in your project directory.
* Activate the Virtual Environment:

On macOS/Linux, use source offbeat-env/bin/activate

* Install Flask: Run pip install Flask to install Flask in your virtual environment.

my\_conn = sqlalchemy.create\_engine(("mysql+mysqldb://root:Dsci-551@localhost/dsci551")

Step 2: Project Structure

Organize your project structure for maintainability and scalability.

* Create a project directory, e.g., OffbeatTravels.

Inside, create the following:

* **app.py:** Your main Flask application file.
* **/templates**: A directory for HTML templates.
* **/static**: A directory for CSS, JavaScript, and images.
* **/models**: (Optional) For your data models if using an ORM.

my\_conn = sqlalchemy.create\_engine("mysql+mysqldb://root:Dsci-  
551@localhost/dsci551")

scp -i [ubuntu@172.31.24.156:~/project/src /](mailto:ubuntu@172.31.24.156:~/project/src%20/) Users/pardibedirian/Desktop/test/

### Install Flask Extensions for Database Integration

* **For MongoDB**: Use **Flask-PyMongo**, which provides PyMongo functionalities wrapped for Flask.
* **For MySQL**: Use **Flask-MySQLdb** or **Flask-SQLAlchemy**. **Flask-MySQLdb** provides

Integrating MongoDB and MySQL into your Flask application involves several steps, from installing the necessary extensions to configuring your databases and defining your data models. Here's how to do it step by step:

**Step 1: Install Flask Extensions for Database Integration**

* **For MongoDB**: Use **Flask-PyMongo**, which provides PyMongo functionalities wrapped for Flask.
* **For MySQL**: Use **Flask-MySQLdb** or **Flask-SQLAlchemy**. **Flask-MySQLdb** provides MySQL connection capabilities, while **Flask-SQLAlchemy** is an ORM that can interface with MySQL, offering more abstraction.

Open your terminal, make sure your virtual environment is activated, and run:

bashCopy code

pip install Flask-PyMongo Flask-MySQLdb Flask-SQLAlchemy

app = Flask(\_\_name\_\_)

app.config['SECRET\_KEY'] = '1998'

app.config['SQLALCHEMY\_DATABASE\_URI'] = 'sqlite:///offbeat\_travels.db' # Ensure you have this

app.config['SQLALCHEMY\_TRACK\_MODIFICATIONS'] = False

* You're initializing your Flask app.
* You're setting a secret key for security purposes.
* You're configuring your app to use an SQLite database.
* You're disabling a feature of SQLAlchemy that you probably don't need, in order to be more resource-efficient.

db.init\_app(app)

csrf = CSRFProtect(app)

### Step 1: Flight Model Creation

create a model for flights in your **models.py**

class Flight(db.Model):

id = db.Column(db.Integer, primary\_key=True)

origin = db.Column(db.String(100), nullable=False)

destination = db.Column(db.String(100), nullable=False)

depart\_date = db.Column(db.Date, nullable=False)

return\_date = db.Column(db.Date)

economy\_fare = db.Column(db.Float)

business\_fare = db.Column(db.Float)

first\_fare = db.Column(db.Float)

# ... any other fields you need

### Step 2: Search Results View

In your **search\_results** route, use the form data to query your **Flight** model for flights that match the search criteria.

@app.route('/search\_results', methods=['GET'])

def search\_results():

# Get form data

trip\_type = request.args.get('trip\_type')

from\_code = request.args.get('from')

to\_code = request.args.get('to')

departure\_date = request.args.get('departure\_date')

return\_date = request.args.get('return\_date') if trip\_type == 'round\_trip' else None

seat\_class = request.args.get('class')

# Query the database for flights

flights = Flight.query.filter\_by(origin=from\_code, destination=to\_code, depart\_date=departure\_date).all()

# Further filter for return flights if it's a round trip

# Render search results

return render\_template('search\_results.html', flights=flights, ...)

### Step 3: Search Results Template

Create a **search\_results.html** template that will display the flights. Iterate over the **flights** passed to the template from your **search\_results** view.

### Step 4: Booking Functionality

Implement a function to handle the booking process. This could involve creating a **Booking** model, associating it with the **User** model, and handling POST requests where a user confirms their flight choices. First, you'll need a model to represent bookings in your **models.py** file. This model will be associated with both the **User** model and the **Flight** model.

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)

status = db.Column(db.String(100), default='Pending')

user = db.relationship('User', backref='bookings')

flight = db.relationship('Flight', backref='bookings')

This model includes a reference to the **User** who made the booking and the **Flight** that was booked. The **booking\_date** records when the booking was made, and **status** could be used to track if a booking is pending, confirmed, or canceled.

### Step 4.2: Booking Route and View Function

Next, you'll need to create a route and view function that can handle the booking process. Users will be directed here when they choose to book a flight.

@app.route('/book/<int:flight\_id>', methods=['GET', 'POST'])

@login\_required

def book\_flight(flight\_id):

# Get the flight the user wants to book

flight = Flight.query.get\_or\_404(flight\_id)

if request.method == 'POST':

# Create a new booking instance

booking = Booking(user\_id=current\_user.id, flight\_id=flight.id)

# Save the booking to the database

db.session.add(booking)

db.session.commit()

# Redirect to a confirmation page, booking management page, or wherever you see fit

flash('Your flight has been booked successfully!', 'success')

return redirect(url\_for('manage\_bookings'))

# For a GET request, render a template where the user can confirm their booking

return render\_template('confirm\_booking.html', flight=flight)

### Step 4.3: Booking Confirmation Template

You should create a **confirm\_booking.html** template where the user can review their flight choice and confirm their booking.

<!-- confirm\_booking.html -->

{% extends 'base.html' %}

{% block content %}

<div class="booking-confirmation-container">

<h2>Confirm Booking</h2>

<div class="flight-details">

<p>Flight from {{ flight.origin }} to {{ flight.destination }}</p>

<!-- Add more flight details here -->

</div>

<form action="{{ url\_for('book\_flight', flight\_id=flight.id) }}" method="POST">

{{ csrf\_token() }} <!-- Include CSRF token for security -->

<button type="submit" class="btn btn-primary">Confirm Booking</button>

</form>

</div>

{% endblock %}

### Step 4.4: Manage Bookings View

Finally, you'll want a page where users can see their current and past bookings. You have a route **@app.route('/my\_bookings')** already defined for this.

### <!-- my\_bookings.html -->

### {% extends 'base.html' %}

### {% block content %}

### <div class="my-bookings-container">

### <h2>My Bookings</h2>

### <ul class="bookings-list">

### {% for booking in current\_user.bookings %}

### <li>

### <p>Booking ID: {{ booking.id }}</p>

### <p>Flight: {{ booking.flight.origin }} to {{ booking.flight.destination }}</p>

### <p>Date: {{ booking.booking\_date }}</p>

### <p>Status: {{ booking.status }}</p>

### <!-- Add options like cancel booking if you want -->

### </li>

### {% endfor %}

### </ul>

### </div>

### {% endblock %}

### Step 5: User Dashboard