A separate file that lists all of the use cases and the queries executed by them (with brief explanation). This should be well organized and readable. It should be detailed enough to give readers a good idea of how your application works, without making them dig through all the code.

Public

1. View Public Info:

```
query = (
    "SELECT flight_num, departure_datetime, airline_name,"
    " arrival_datetime, "
    + "arr_airport.airport_name, arr_airport.city,"
    " dep_airport.airport_name, dep_airport.city, base_price "
    + "FROM Flight INNER JOIN Airport as arr_airport ON"
    " Flight.arrival_airport_code = arr_airport.airport_code "
    + "INNER JOIN Airport as dep_airport ON Flight.departure_airport_code ="
    " dep_airport.airport_code "
    + "WHERE departure_datetime > CURRENT_TIMESTAMP"
)

params = ()
if dep_city != "":
    query += " and dep_airport.city = %s"
    params += (dep_city,)
if dep_airport_name != "":
    query += " and dep_airport.airport_name = %s"
    params += (dep_airport_name,)
if arr_city != "":
    query += " and arr_airport.city = %s"
    params += (arr_city,)
if arr_airport_name != "":
    query += " and arr_airport.airport_name = %s"
    params += (arr_airport_name,)
if departure_date != "":
    query += " and departure_datetime = %s"
    params += (departure_date,)

# print(query)
cursor.execute(query, params)
data_array = cursor.fetchall()
```

Return all the future flights. If certain values about the city, airport name and departure date are entered, these conditions will be applied when searching for future flights.

```
query2 = (
    "SELECT flight_num, departure_datetime, airline_name,"
    " arrival_datetime, "
    + "arr_airport.airport_name, arr_airport.city,"
    " dep_airport.airport_name, dep_airport.city, base_price "
    + "FROM Flight INNER JOIN Airport as arr_airport ON"
    " Flight.arrival_airport_code = arr_airport.airport_code "
    + "INNER JOIN Airport as dep_airport ON"
    " Flight.departure_airport_code = dep_airport.airport_code "
    + "WHERE departure_datetime > CURRENT_TIMESTAMP"
)
params2 = ()

if dep_city != "":
    query2 += " and arr_airport.city = %s"
    params2 += (dep_city,)
if dep_airport_name != "":
    query2 += " and arr_airport.airport_name = %s"
    params2 += (dep_airport_name,)
if arr_city != "":
    query2 += " and dep_airport.city = %s"
    params2 += (arr_city,)
if arr_airport_name != "":
    query2 += " and dep_airport.airport_name = %s"
    params2 += (arr_airport_name,)
if departure_date != "":
    query2 += " and departure_datetime = %s"
    params2 += (return_date,)

cursor.execute(query2, params2)
```

If the user chooses to search for two-way flights, this query will also be executed. All information will remain the same except departure city and departure airport name will be arrival city and arrival airport and vice versa.

```
  query = (
      "SELECT status FROM Flight "
      + "WHERE airline_name = %s AND flight_num = %s AND departure_datetime"
      " = %s"
  )

  cursor.execute(query, (airline_name, int(flight_num), dep_date))
  data = cursor.fetchone()
```

For checking flight status, if the targeted flight is found, return the status; if not, display an error message.

2. Register:

```
if isCustomer == "true":
    print("hi")
    query = "SELECT * FROM Customer WHERE email = %s"
    cursor.execute(query, (email))
else:
    query = "SELECT * FROM Staff WHERE username = %s"
    cursor.execute(query, (username))
```

Test if the customer or the staff already exist in the database. If so, registration fails

```
if isCustomer == "true":
    ins = (
        "INSERT INTO Customer VALUES(%s, %s, %s, %s, %s, %s, %s, %s,"
        " %s, %s, %s, %s, %s)"
    )
    cursor.execute(
        ins,
        (
            email,
            fname,
            lname,
            password,
            bldg_num,
            street,
            apt,
            city,
            state,
            passport_num,
            passport_exp,
            passport_country,
            date_of_birth,
        ),
    )
    for p in phone_num:
        ins = "INSERT INTO Customer_Phone VALUES(%s, %s)"
        cursor.execute(ins, (email, p))
```

Insert customer information & phone numbers.

```
ins = "INSERT INTO Staff VALUES(%s, %s, %s, %s, %s, %s)"
cursor.execute(
    ins,
    (username, password, airline_name, fname, lname, date_of_birth),
)

for e in email:
    ins = "INSERT INTO Staff_Email VALUES(%s, %s)"
    cursor.execute(ins, (username, e))

for p in phone_num:
    print(p)
    ins = "INSERT INTO Staff_Phone VALUES(%s, %s)"
    cursor.execute(ins, (username, p))
```

Insert staff information & emails & phone numbers.

3. Login:

```
if isCustomer == "true":
    # get the first_name of the customer
    query = (
        "SELECT fname, email FROM Customer WHERE email = %s and password"
        " = %s"
    )

else:
    # get the airline_name of the staff
    query = (
        "SELECT fname, airline_name FROM Staff WHERE username = %s and"
        " password = %s"
    )

cursor.execute(query, (username, password))

data = cursor.fetchone()
```

When a customer tries to log in, they need to enter an email and a password. When a staff tries to log in, they need to enter a username and a password.

Customer

1. View My flights: Provide various ways for the user to see flights information which he/she purchased. The default should be showing for the future flights. Optionally you may include a way for the user to specify a range of dates, specify destination and/or source airport name or city name etc.

For this use case we created 2 APIs: prev_flights and future_flights

```
890     query = (
891         "SELECT Flight.flight_num, Flight.departure_datetime, dep_airport.city"
892         " AS departure_city, dep_airport.airport_name AS departure_airport,"
893         " Flight.airline_name, Flight.arrival_datetime, arr_airport.city AS"
894         " arrival_city, arr_airport.airport_name AS arrival_airport,"
895         " Ticket.sold_price, Ticket.ticket_id FROM Ticket INNER JOIN Flight ON"
896         " Ticket.airline_name = Flight.airline_name AND Ticket.flight_num ="
897         " Flight.flight_num AND Ticket.departure_datetime ="
898         " Flight.departure_datetime INNER JOIN Airport AS dep_airport ON"
899         " Flight.departure_airport_code = dep_airport.airport_code INNER JOIN"
900         " Airport AS arr_airport ON Flight.arrival_airport_code ="
901         " arr_airport.airport_code WHERE Ticket.email = %s AND"
902         " Flight.arrival_datetime <= CURRENT_TIMESTAMP;"
903     )
904     cursor.execute(query, (customer_email))
```

This query is used to find all the past flights the customer has been on and is part of prev_flights API.

```
936    def future_flights():
937        customer_email = request.form["customer_email"]
938        cursor = conn.cursor()
939        query = (
940            "SELECT Flight.flight_num, Flight.departure_datetime, dep_airport.city"
941            " AS departure_city, dep_airport.airport_name AS departure_airport,"
942            " Flight.airline_name, Flight.arrival_datetime, arr_airport.city AS"
943            " arrival_city, arr_airport.airport_name AS arrival_airport,"
944            " Ticket.sold_price, Ticket.ticket_id FROM Ticket INNER JOIN Flight ON"
945            " Ticket.airline_name = Flight.airline_name AND Ticket.flight_num ="
946            " Flight.flight_num AND Ticket.departure_datetime ="
947            " Flight.departure_datetime INNER JOIN Airport AS dep_airport ON"
948            " Flight.departure_airport_code = dep_airport.airport_code INNER JOIN"
949            " Airport AS arr_airport ON Flight.arrival_airport_code ="
950            " arr_airport.airport_code WHERE Ticket.email = %s AND"
951            " Flight.departure_datetime > CURRENT_TIMESTAMP;"
952        )
```

This query is used to find all the future flights the customer has been on and is part of future_flights API.

2. Search for flights: Search for future flights (one way or round trip) based on source city/airport name, destination city/airport name, dates (departure or return).

The query given below is used to search for flights based on a bunch of parameters given by the user like range of dates, departure/arrival airport, departure/ arrival city and one way/round trip.

```
87      flights = []
88      query = (
89          "SELECT flight_num, departure_datetime, airline_name,"
90          " arrival_datetime, "
91          + "arr_airport.airport_name, arr_airport.city,"
92          " dep_airport.airport_name, dep_airport.city, base_price "
93          + "FROM Flight INNER JOIN Airport as arr_airport ON"
94          " Flight.arrival_airport_code = arr_airport.airport_code "
95          + "INNER JOIN Airport as dep_airport ON Flight.departure_airport_code ="
96          " dep_airport.airport_code "
97          + "WHERE departure_datetime > CURRENT_TIMESTAMP and"
98      )
99      queries = ()
00      if params["source city"]:
01          query += " dep_airport.city = %s and"
02          queries += (params["source city"],)
03      if params["destination city"]:
04          query += " arr_airport.city = %s and"
05          queries += (params["destination city"],)
06      if params["source airport"]:
07          query += " dep_airport.airport_name = %s and"
08          queries += (params["source airport"],)
09      if params["destination airport"]:
10          query += " arr_airport.airport_name = %s and"
11          queries += (params["destination airport"],)
12      if params["departure date"]:
13          query += " DATE(departure_datetime) = %s and"
14          queries += (params["departure date"],)
15      if query[-4:] == " and":
16          query = query[:-4]  # cut the trailing ' and'\
17
18      cursor.execute(query, queries)
19      data_array = cursor.fetchall()
20      for data in data_array:
21          flight = {
```

This query searches for one way flight

```
if params[
    "return date"
]:  # I assume a return date means the user wants to come back to the airport from which they departe
    return_query = (
        "SELECT flight_num, departure_datetime, airline_name,"
        " arrival_datetime, "
        + "arr_airport.airport_name, arr_airport.city,"
        " dep_airport.airport_name, dep_airport.city, base_price "
        + "FROM Flight INNER JOIN Airport as arr_airport ON"
        " Flight.arrival_airport_code = arr_airport.airport_code "
        + "INNER JOIN Airport as dep_airport ON"
        " Flight.departure_airport_code = dep_airport.airport_code "
        + "WHERE departure_datetime > CURRENT_TIMESTAMP and"
    )
    ret_queries = ()
    if params["source city"]:
        return_query += " arr_airport.city = %s and"
        ret_queries += (params["source city"],)
    if params["destination city"]:
        return_query += " dep_airport.city = %s and"
        ret_queries += (params["destination city"],)
    if params["source airport"]:
        return_query += "arr_airport.airport_name = %s and"
        ret_queries += (params["source airport"],)
    if params["destination airport"]:
        return_query += " dep_airport.airport_name = %s and"
        ret_queries += (params["destination airport"],)
    return_query += " DATE(departure_datetime) = %s"
    ret_queries += (params["return date"],)
    cursor.execute(return_query, ret_queries)
    ret_data = cursor.fetchall()
    for data in ret_data:
        flight = {
            "flight_num": data["flight_num"],
```

This query searches for return flight.

3. Purchase tickets: Customer chooses a flight and purchase ticket for this flight, providing all the needed data, via forms. You may find it easier to implement this along with a use case to search for flights.

```
# Check if ticket has already been purchased
query = (
    "SELECT * FROM Ticket WHERE flight_num = %s AND airline_name = %s AND"
    " departure_datetime = %s AND email = %s"
)
```

```
    # Calculate price of ticket
    query = (
        "SELECT seats, base_price FROM Airplane NATURAL JOIN Flight WHERE"
        " flight_num = %s AND airline_name = %s AND departure_datetime = %s"
    )
    cursor.execute(query, (flight_num, airline_name, dep_timestamp))
```

```
    # Check if the tickets were sold out
    query = (
        "SELECT COUNT(ticket_id) as total FROM Ticket WHERE flight_num = %s AND"
        " departure_datetime = %s AND airline_name = %s"
    )
    cursor.execute(query, (flight_num, dep_timestamp, airline_name))
```

The query below is used to generate a new ticket_id

```
query = "SELECT DISTINCT ticket_id FROM Ticket"
cursor.execute(query,())
data = cursor.fetchall()
if not data:
    ticket_id = 0
else:
    ticket_id = max([dct["ticket_id"] for dct in data])+1
```

```
# Insert purchase into database
ins = (
    "INSERT INTO Ticket VALUES(%s, %s, %s, %s, %s, %s, %s, %s, %s, CURRENT_TIMESTAMP, %s)"
)
cursor.execute(
    ins,
    (
        ticket_id,
        airline_name,
        flight_num,
        dep_timestamp,
        sold_price,
        card_type,
        card_name,
        card_num,
        exp_date,
        customer_email,
    ),
)
```

4. Cancel Trip: Customer chooses a purchased ticket for a flight that will take place more than 24 hours in the future and cancel the purchase. After cancellation, the ticket will no longer belong to the customer. The ticket will be available again in the system and purchasable by other customers.

The query below is used to delete the flight from Ticket table

```
# only shows flights more than 24 hours in the future so a user cannot cancel a flight that occurs in
query = (
    "DELETE FROM Ticket WHERE ticket_id = %s AND flight_num = %s AND departure_datetime = %s "
    "AND airline_name = %s AND email = %s"
)
cursor.execute(query, (ticket_id, flight_num, dep_timestamp, airline_name, customer_email))
```

5. Give Ratings and Comment on previous flights: Customer will be able to rate and comment on their previous flights (for which he/she purchased tickets and already took that flight).

The queries check if rating for flight exists, if it does then it is replaced with a new one, if it doen't exit then a new rating is added in the review table

```
1261
1262        query = 'SELECT * FROM Ticket WHERE email = %s and ticket_id = %s'
1263        cursor.execute(query, (customer_email, ticket_id))
1264        data = cursor.fetchone()
1265
1266        if not data:
1267            return jsonify({})
1268
1269        query = "SELECT * FROM Ticket INNER JOIN Reviews ON Ticket.email = Reviews.email"
1270        "WHERE Ticket.email = %s and Ticket.ticket_id = %s"
1271        cursor.execute(query, (customer_email, ticket_id))
1272        data = cursor.fetchone()
1273
1274        if data:
1275            query = "DELETE FROM Reviews WHERE ticket_id = %s AND email = %s"
1276            cursor.execute(query, (ticket_id, customer_email))
1277            conn.commit()
1278
1279        # insert rating and comment
1280        query = 'INSERT INTO Reviews VALUES (%s, %s, %s, %s)'
1281        cursor.execute(query, (customer_email, ticket_id, rating, comment))
1282        conn.commit()
1283        cursor.close()
1284        return jsonify({})
```

6.Track My Spending: Default view will be total amount of money spent in the past year and a bar chart/table showing month wise money spent for last 6 months. He/she will also have option to specify a range of dates to view total amount of money spent within that range and a bar chart/table showing month wise money spent within that range.

```
query1="SELECT SUM(sold_price) AS total_spent FROM Ticket WHERE email = %s AND purchase_datetime BETWEEN %s AND %s;"
cursor.execute(query1, (customer_email, start, end))
spending = cursor.fetchone()

query = "SELECT CONCAT(MONTH(purchase_datetime), '/', YEAR(purchase_datetime)) AS label,"
"SUM(sold_price) AS tickets FROM Ticket WHERE email = %s AND purchase_datetime BETWEEN %s AND %s "
"GROUP BY MONTH(purchase_datetime), YEAR(purchase_datetime) ORDER BY YEAR(purchase_datetime), MONTH(purchase_datetime)"
cursor.execute(query, (customer_email, start, end))
data = cursor.fetchall()
```

```
query1="SELECT SUM(sold_price) AS total_spent FROM Ticket WHERE email = %s AND "
"purchase_datetime BETWEEN DATE_SUB(NOW(), INTERVAL 1 YEAR) AND NOW();"
cursor.execute(query1, (customer_email))
spending = cursor.fetchone()

query = "SELECT CONCAT(MONTH(purchase_datetime), '/', YEAR(purchase_datetime)) AS label, SUM(sold_price) AS tickets"
"FROM Ticket WHERE email = %s AND purchase_datetime BETWEEN DATE_SUB(NOW(), INTERVAL 6 MONTH) AND NOW() GROUP BY"
"MONTH(purchase_datetime), YEAR(purchase_datetime) ORDER BY YEAR(purchase_datetime), MONTH(purchase_datetime)"
cursor.execute(query, (customer_email))
data = cursor.fetchall()

return {"spending": spending["total_spent"], "months": data}
```

7.Logout: The session is destroyed and a “goodbye” page or the login page is displayed.

```
# 10 logout
@app.route("/logout")
def logout():
    session.pop("user", None)
    return jsonify([])
```

Airline Staff use cases:

1. View flights:
   a. See passengers of a flight
      The user should be able to see the customers of a particular flight given the flight number, airline name and departure date time.

```
query = (
    "SELECT Customer.fname as fname,"
    " Customer.email as email FROM Customer INNER JOIN Ticket ON"
    " Ticket.email = Customer.email INNER JOIN Flight ON"
    " Flight.airline_name = Ticket.airline_name AND Flight.flight_num ="
    " Ticket.flight_num AND Flight.departure_datetime ="
    " Ticket.departure_datetime WHERE Flight.flight_num = %s AND"
    " Flight.airline_name = %s AND Flight.departure_datetime = %s;"
)

print(flight_num, airline_name, departure_datetime)

cursor.execute(query, (flight_num, airline_name, departure_datetime))
data = cursor.fetchall()
```

b. Display all future flights operated by the airline that the staff works for in the next 30 days

```
query = "SELECT airline_name FROM Airline WHERE airline_name = %s"
cursor.execute(query, (airline_name))
data = cursor.fetchone()
if not data:
    return jsonify([])

# display flights
query1 = (
    "SELECT flight_num, departure_datetime, airline_name,"
    + " arrival_datetime, "
    + "arr_airport.airport_name, arr_airport.city,"
    + " dep_airport.airport_name, dep_airport.city, base_price "
    + "FROM Flight INNER JOIN Airport as arr_airport ON"
    + " Flight.arrival_airport_code = arr_airport.airport_code "
    + "INNER JOIN Airport as dep_airport ON"
    + " Flight.departure_airport_code = dep_airport.airport_code "
    + "WHERE departure_datetime > CURRENT_TIMESTAMP AND "
    + "TIMESTAMPDIFF(SECOND, departure_datetime, NOW()) <= (30 * 60)"
    + "AND airline_name = %s"
)

cursor.execute(query1, (airline_name))
data_array2 = cursor.fetchall()
```

c. Display the flights that are in a particular range of date

```
query = (
    "SELECT f.flight_num, f.departure_datetime, f.airline_name,"
    " f.arrival_datetime, a1.airport_name as arrival_airport_name, a1.city"
    " as arrival_city, a2.airport_name as departure_airport_name, a2.city"
    " as departure_city, f.base_price as price FROM Flight f JOIN Airport"
    " a1 ON f.departure_airport_code = a1.airport_code AND a1.airport_name"
    " = %s AND a1.city = %s JOIN Airport a2 ON f.arrival_airport_code ="
    " a2.airport_code AND a2.airport_name = %s AND a2.city = %s WHERE"
    " f.departure_datetime BETWEEN %s AND %s AND airline_name = %s"
)

cursor.execute(
    query,
    (
        src_airport,
        src_city,
        dst_airport,
        dst_city,
        start,
        end,
        airline_name,
    ),
)

data_array = cursor.fetchall()
flights = []
```

2. Create new flights: He or she creates a new flight, providing all the needed data, via forms. The application should prevent unauthorized users from doing this action. Defaults will be showing all the future flights operated by the airline he/she works for the next 30 days.

```
query = (
    "SELECT * FROM Flight WHERE flight_num = %s AND airline_name = %s AND"
    " departure_datetime = %s;"
)

cursor.execute(query, (flight_num, airline_name, departure_datetime))
flight = cursor.fetchone()

query2 = "SELECT * FROM Airplane WHERE airplane_id = %s"
cursor.execute(query2, (airplane_id))
airplane = cursor.fetchone()

query3 = "SELECT * FROM Airport WHERE airport_code = %s;"
cursor.execute(query3, (arrival_airport_code))
airport1 = cursor.fetchone()

query4 = "SELECT * FROM Airport WHERE airport_code = %s;"
cursor.execute(query4, (departure_airport_code))
airport2 = cursor.fetchone()

if (
    flight
    or airplane == None
    or airport1 == None
    or airport2 == None
    or departure_airport_code == arrival_airport_code
    or int(base_price) < 0
):
    return {"valid": False}
```

If any of the airplane or the airport doesn't exist, or the user enters invalid information like ticket price and airports, display an error message.

```
query2 = "INSERT INTO Flight VALUES(%s, %s, %s, %s, %s, %s, %s, %s, %s)"
cursor.execute(
    query2,
    (
        flight_num,
        departure_datetime,
        airline_name,
        arrival_datetime,
        arrival_airport_code,
        departure_airport_code,
        airplane_id,
        base_price,
        status,
    ),
)

conn.commit()
```

If there's no error, then proceed to insert the flight information into the Flight table.

3. Change Status of flights: He or she changes a flight status (from on-time to delayed or vice versa) via forms.

```
query = (
    "SELECT airline_name, flight_num FROM Flight WHERE airline_name = %s"
    " AND flight_num = %s AND departure_datetime = %s"
)
cursor.execute(query, (airline_name, flight_num, departure_datetime))
data = cursor.fetchone()
if not data:
    return {"change_status": False}
```

If the flight doesn't exist, display an error message.

```
query = (
    "UPDATE Flight SET status = %s"
    + "WHERE flight_num = %s AND departure_datetime = %s AND airline_name"
    " = %s"
)
cursor.execute(
    query, (new_status, flight_num, departure_datetime, airline_name)
)
conn.commit()
```

If the flight exists, update its status to the status that user inputs.

4. Add airplane in the system:

```
query1 = "SELECT airline_name FROM Airplane WHERE airplane_id = %s"
cursor.execute(query1, (airplane_id))
data1 = cursor.fetchone()
if data1:
    return jsonify([])
```

If the airplane already exists, the update fails.

```
query2 = "INSERT INTO Airplane VALUES(%s, %s, %s, %s, %s, %s)"
cursor.execute(
    query2,
    (
        airplane_id,
        airline_name,
        seats,
        manufacturing_date,
        manufacturer,
        str(current_year - int(manufacturing_date[0:4])),
    ),
)
```

If not, then insert the values user inputs to the Airplane table.

```
query3 = "SELECT * FROM Airplane WHERE airline_name = %s"
cursor.execute(query3, (airline_name))
airplanes = cursor.fetchall()
conn.commit()
cursor.close()

return jsonify(airplanes)
```

Display all airplanes of the airline that the user works for.

5. Add new airport in the system: He or she adds a new airport, providing all the needed data, via forms. The application should prevent unauthorized users from doing this action.

```
    query = "SELECT airport_code FROM Airport WHERE airport_code = %s"
    cursor.execute(query, (airport_code))
    data = cursor.fetchone()
    if data:
        return {"add_airport": False}  # Duplicate data
```

If the airport already exists, display an error message.

```
query2 = "INSERT INTO Airport VALUES(%s, %s, %s, %s, %s)"
cursor.execute(
    query2, (airport_code, airport_name, city, country, airport_type)
)

conn.commit()
```

If not, then insert the values the user inputs to the Airport table.

6. View flight ratings:

```
query = ("SELECT Customer.fname, Reviews.rating, Reviews.comment FROM Customer INNER JOIN"
        " Ticket ON Customer.email = Ticket.email INNER JOIN Reviews ON Ticket.ticket_id = "
        "Reviews.ticket_id WHERE Ticket.flight_num = %s AND Ticket.airline_name = %s AND "
        "Ticket.departure_datetime = %s;"
)

cursor.execute(query, (flight_num, airline_name, departure_datetime))
datas = cursor.fetchall()
```

Display all customer reviews of a certain flight

```
query2 = (
    "SELECT AVG(rating) AS avg_rating FROM Reviews INNER JOIN Ticket ON"
    " Reviews.ticket_id = Ticket.ticket_id INNER JOIN Flight ON"
    " Ticket.flight_num = Flight.flight_num AND Ticket.airline_name ="
    " Flight.airline_name AND Ticket.departure_datetime ="
    " Flight.departure_datetime WHERE Flight.flight_num = %s AND"
    " Flight.airline_name = %s AND Flight.departure_datetime = %s"
)

cursor.execute(query2, (flight_num, airline_name, departure_datetime))
data2 = cursor.fetchall()
conn.commit()
cursor.close()
if data2[0]["avg_rating"] == None:
    data2[0]["avg_rating"] = 0

return jsonify({"rates": rates, "avg_rating": data2[0]["avg_rating"]})
```

Display the average rating of a flight.

7. View frequent customers:

```
query_frequent_customer = (
    "SELECT Customer.fname, Customer.lname,"
    " COUNT(Ticket.ticket_id) AS num_tickets FROM Customer INNER JOIN"
    " Ticket ON Customer.email = Ticket.email WHERE Ticket.airline_name ="
    " %s AND Ticket.purchase_datetime >= DATE_SUB(NOW(), INTERVAL 1 YEAR)"
    " GROUP BY Customer.email, Customer.fname, Customer.lname ORDER BY"
    " num_tickets DESC LIMIT 1;"
)

cursor.execute(query_frequent_customer, (airline_name))
customer_flights = cursor.fetchone()

cursor.close()
return {
    "fname": customer_flights["fname"],
    "lname": customer_flights["lname"],
}
```

Rank the customer that has the most number of flight tickets within the last year. Display the most frequent customer.

```
query = (
    "SELECT Flight.flight_num, Flight.departure_datetime, dep_airport.city"
    " as dep_city, dep_airport.airport_name as dep_airport,"
    " Flight.arrival_datetime, arr_airport.city as arr_city,"
    " arr_airport.airport_name as arr_airport, Ticket.sold_price FROM"
    " Flight INNER JOIN Ticket ON Flight.flight_num = Ticket.flight_num AND"
    " Flight.airline_name = Ticket.airline_name AND"
    " Flight.departure_datetime = Ticket.departure_datetime INNER JOIN"
    " Airport as dep_airport ON Flight.departure_airport_code ="
    " dep_airport.airport_code INNER JOIN Airport as arr_airport ON"
    " Flight.arrival_airport_code = arr_airport.airport_code WHERE"
    " Ticket.email = %s AND Flight.airline_name = %s"
)

cursor.execute(query, (email, airline_name))
data_array = cursor.fetchall()
```

Show a list of all flights a particular Customer has taken only on that particular airline.

8. View reports:

```
    query = (
        "SELECT COUNT(*) as total_tickets_sold FROM Ticket WHERE airline_name ="
        " %s AND purchase_datetime BETWEEN %s AND %s;"
    )
    cursor.execute(query, (airline_name, start, end))
    tickets = cursor.fetchone()
```

Return total amounts of tickets based on the range

```
    query2 = ("SELECT CONCAT(MONTH(purchase_datetime), '/', YEAR(purchase_datetime))"
        " as label, COUNT(*) as tickets FROM Ticket WHERE airline_name = %s AND"
        " purchase_datetime BETWEEN %s AND %s GROUP BY YEAR(purchase_datetime),"
        " MONTH(purchase_datetime) ORDER BY YEAR(purchase_datetime),"
        " MONTH(purchase_datetime);"
    )
    cursor.execute(query2, (airline_name, start, end))
    data = cursor.fetchall()
    return {"tickets": tickets["total_tickets_sold"], "months": data}
```

Return monthly wise tickets data.

9. View Earned Revenue:

```
query1 = (
    "SELECT SUM(sold_price) AS total_revenue FROM Ticket WHERE airline_name"
    " = %s AND departure_datetime BETWEEN DATE_SUB(NOW(), INTERVAL 1 MONTH)"
    " AND NOW();"
)
cursor.execute(query1, (airline_name))
month = cursor.fetchone()
```

Show the revenue within the last month

```
query2 = (
    "SELECT SUM(sold_price) AS total_revenue FROM Ticket WHERE airline_name"
    " = %s AND departure_datetime BETWEEN DATE_SUB(NOW(), INTERVAL 1 YEAR)"
    " AND NOW();"
)
cursor.execute(query2, (airline_name))
year = cursor.fetchone()
```

Show the revenue within the last year

10. Logout: The session is destroyed and a “goodbye” page or the login page is displayed.

```
# 10 logout
@app.route("/logout")
def logout():
    session.pop("user", None)
    return jsonify([])
```