Project Scope and Plan: Flight Reservation System

The Dream Team: Jasmine Sajna, Chris Lam, Dante LoPriore

November 18, 2024

1 Defining Project Scope and Objectives

1.1 Finalized Project Scope

The aim of this project is to design and develop a user-friendly flight reservation system to simulate for customers how they would book, cancel, and view available seats across the offered flights of an airline. This system will explore how data can be dynamically stored and managed using databases, and how this dynamically stored data can be sorted using the linked list data structure. The expected outcomes include a functioning well documented web-based system, a final project report, and a powerpoint presentation.

1.2 Project Performance Requirements To Create Flight Reservation System

The project's objectives for the anticipated system are:

- To allow customers to book, cancel, and search for an available seat using an interactive web-based flight reservation system, and also easily view available flights using their starting and destination locations.
- Establish a hierarchy to differentiate the access of the systems features of the roles of users and the admin profiles.
- Provide the current booking trends and flight availability statistics to help the airline make profitable decisions based on the given data results.
- Preform features with fast response times and allow for many users to interact with the system without any changes in performance.

1.3 Project Code Implementation Objectives

The project's coding secure architecture objectives are:

- **Frond End:** The Front End component would be the team building a responsive web application using a GUI to visualize the flight reservation functions that are set for the users and admin.
- **Back End:** The Back End component would be the team building a the server side of the application that enables the transactions and processes made in a database using Flask and SQL.
- **Emphasis on Linked Lists and Sorting Algorithms:** The team's primary data structures to include are linked lists and sorting algorithms. The team plan to use the sorting algorithms to create the navigation, searching. The team will use linked lists to represent the nodes as the seats reference the seats with pointer values.

2 Project Plan

2.1 Timeline

The overall timeline for the project is divided into phases:

- **Week 1 (October 7 - October 13)**: Define scope, establish team roles, outline skills/tools.
- **Week 2 (October 14 - October 20)**: Begin development, set up project repository
- **Week 3 (October 21 - October 27)**: Coding, work on backend and frontend separately
- **Week 4 (October 28 - November 3)**: Finish backend, integrate the frontend. Start PowerPoint presentation.
- **Week 5 (November 4 - November 10)**: Finalize system, organize test scripts, continue documenting in report.
- **Week 6 (November 11 - November 17)**: Revise and finalize technical report and presentation.
- **Week 7 (November 18 - November 28)**: Final presentation, submit report.

2.2 Milestones

Key milestones include:

- Project Scope and Plan (October 7).
- GitHub Repository Setup and Initial Development (October 9).
- Showcase Backend Progress and Completion (October 28).
- Final System Testing and Report Draft (November 10).
- Final Presentation and Report Submission (November 28).

2.3 Team Roles

- **Chris Lam:** Testing, backend logic, UI
- **Dante Lopriore:** Testing, backend logic, UI
- **Jasmine Sajna:** Testing, backend logic, UI

These roles are designed such that all members are equally involved in all aspects.

3 Team Discussion Summary (October 7)

3.1 Skills Assessment

Team members need the following skills to complete the project:

- HTML/CSS/JavaScript for frontend development to build GUI and User interface
- Backend skills in Python (Flask/Django) for the server-side logic.
- Familiarity with GitHub for version control.
- Use of Overleaf for writing the final technical report.

3.2 Tools & Technologies

- **Programming Languages:** Python, JavaScript, C++
- **Database:** SQLite/MySQL.
- **Collaboration Tools:** GitHub and Overleaf.

3.3 Team Responsibilities

Each team member will focus on specific areas, but roles may adapt as the project progresses:

- **Frontend Development (Dante, Chris):** In this role, the group would build the user interface for the flight reservation system using a GUI. The GUI will allow the user to see the data visualization of the current navigation options to reserve and cancel an available flight.
- **Backend Development (Jasmine, Dante):** In this role, the group will be responsible for creating a reliable backend system that creates an universal database for users with different roles to schedule a flight, reserve an available seat, and display passengers info. The backend group are currently deciding to finalize the backend framework with using either Flask SQL or django.
- **Technical Documentation (Whole Team):** As a collective group, it is each team member's responsibility to document their progress with their role and provide feedback to improve the project's projected goals.
- **System Testing (Jasmine, Chris):** Within this role, the team members would check in with the progress of the code functionality and validating that the system works for the front end to the back end.

4 Skills & Tools Assessment (October 8)

4.1 Skills Gaps

Team Members need experience with backend development along with experience in front end development. To address these, we plan to follow online tutorials during Week 2

4.2 Tools

We will use the following tools:

- Python (Flask) for backend development.
- SQLite for database management.
- GitHub for version control and collaboration.
- Overleaf for the technical report.

5 Initial Setup Evidence (October 9)

5.1 Project Repository

The project repository has been created on GitHub, accessible by all team members. The repository can be found at `https://github.com/sajna-j/flight-reservation`.

5.2 Setup Proof

The development environment has been successfully set up. Screenshots of the setup process are provided in the Appendix.

6 Progress Review (October 10)

6.1 Progress Update after Revisions

Our team has successfully finished the initial setup of the development environment through VSCode and repository. Starting development has begin on the backend as well.

7 Algorithm Design for System's Basic Logic (October 14-20)

The current pseudcode list in this section shows all the methods that make up the flight reservation system. The bare logic of the simple system are the sorting and navigation of traverse through seats, and allowing the user to dynamically enter a seat that is available or booked. The bare logic files can be seen in this github link to the branch of the working python file `https://github.com/sajna-j/flight-reservation/blob/bare-logic-in-python`.

7.1 General Booking System Helper Functions

7.1.1 insertHead Method

Algorithm 1 insertHead:

Input: An integer representing seat , *data*
Output (Changed State): A linked list with a new node inserted at the head of the list
Begin Algorithm: insertHead
$newNode \leftarrow SeatNode(data)$
if $head == NULL$ **then**
  $head \leftarrow newNode$
  $return$;
end
$newNode.next \leftarrow this.head$
$this.head \leftarrow newNode$
End Algorithm: insertHead

Frequency Count Analysis:
SeatNode creation: 1 operation
conditional check: 1 operation
assignment of newNode at head: 1 operation
Shifting of node at head back: 1 operation
Frequency Count: F(n) = 4
Time Complexity: O(1)
Purpose: To insert a node at the head of the list

7.1.2 insertAtEnd Method

Algorithm 2 insertAtEnd

Input: An integer representing seat , *data*
Output (Changed State): A linked list with a new node inserted at the tail of the list
Begin Algorithm: insertAtEnd
$newNode \leftarrow SeatNode(data)$
if $head == NULL$ **then**
  $head \leftarrow newNode$
  $return$;
else
  $temp \leftarrow head$
  while $temp.next! = NULL$ **do**
    $temp \leftarrow temp.next$
  end
  $temp.next \leftarrow newNode$
end
End Algorithm: insertAtEnd

Frequency Count Analysis:
newNode creation: 1 operation
conditional check: 1 operation
assignment of newNode: 1 operation

shifting head to temp: 1 operation
while loop: n iterations
shifting temp: n iterations

Frequency Count: F(n) = 2n + 4
Time Complexity: O(n)
Purpose: To insert a node at the tail of the list

7.1.3 deleteNodeAtPosition Method

Algorithm 3 deleteNodeAtPosition

Input: An integer value to represent the positionIndex
Output (Changed State): Get a linked list with a node deleted at a given index position.
Begin Algorithm: deleteNodeAtPosition
$count \leftarrow 0$
if $head == NULL$ **then**
  print $The * list * is * empty$
  $return$
end
if $positionIndex == 0$ **then**
  $temp = head$
  $head \leftarrow temp.next$
  $return$
end
$currNode = head$
while $(currNode.next$ *not equal* $nullptr)$ *and* $(positionIndex - 1$ *not equal* $count)$ **do**
  $count+ = 1$
  $currNode \leftarrow currNode.next$
end
$tempNode \leftarrow currNode.next$
print $DeletingSeat : tempNode.data, Address : currNode.next$
$currNode.next \leftarrow tempNode.next$
delete $tempNode$
End Algorithm: deleteNodeAtPosition

Frequency Count Analysis:
user input of position: 1 operation
count initialization: 1 operation
check head isn't NULL: 1 comparison
check if user wants to remove the first head: 1 comparison
assign head to currNode: 1 operation
while loop iterations: n operations
while loop comparisons: 2*n comparisons
inside the while loop operations: 2*n operations
assign currNode.next to tempNode: 1 operation
print seat being deleted and the address: 1 operation
assign tempNode.next to currNode.next: 1 operation
delete tempNode: 1 operation

Frequency Count: F(n) = $5n + 9$
Time Complexity: O(n)

Purpose: To remove the seat node in a single linked list at a given position index. Iterate over each node of the list and keep a count for each pass of the node in the linked list to keep track of the index to remove the node. When the list reaches the index that the user wants to delete then delete the node and redirect the node's pointers accordingly.

7.1.4 addNodeAtPosition Method

Algorithm 4 addNodeAtPosition

Input: An integer value or index to add the node *positionIndex*, the integer values representing the data in the node *data*
Output (Changed State): The linked list with the added seat based on the user's index
Begin Algorithm: addNodeAtPosition

Initialize SeatNode* newNode = new SeatNode(data)

Set count ← 0

if *(head is equal to NULLpointer)* **then**
    print − > throw error how list is empty
    terminate program
end
if *(positionIndex is equal to 0)* **then**
    newNode− >next = head
    head = newNode
    terminate program
end
Intialize SeatNode* currNode = head
while *((currNode− >next not equal to nullptr) and (positionIndex - 1 not equal to count))* **do**
    increment count by 1
    currNode = currNode− >next
end
print− > "Add Seat:" + newNode− >data + ", Address: " + currNode-¿next
newNode− >next = currNode− >next
currNode− >next = newNode

End Algorithm: addNodeAtPosition

Frequency Count Analysis:
initialize the new node that is going to be added on to the linked list: 1 operation
set a count to keep track the nodes in the linked list: 1 operation
check if the linked list is empty using if comparison: 1 operation
if condition is true throw an error how the operation can be preformed: 1 operation
if condition is true end the program: 1 operation
check if the new node should be placed at the start of the linked list using if comparison: 1 operation
if condition is true reassign the pointers to make the next item in the list for the new node to be the start of the list: 1 operation
if condition is true make the start of the list be the new node: 1 operation
if condition is true end the program: 1 operation
iterate each element in the list via while loop until the current position is found to place the new node: 2n operations
increment the count of the current place of the current node in list by 1: 1 operation
reassign the pointers of the current node to be the next item in the list: 1 operations
print the what the new node being placed into the list: 1 operations

reasssign the pointers of the next element of the new node to be the next element of the current node : 1 comparison

reasssign the pointers of the next element of the current node to be the new node : 1 comparison

Frequency Count: F(n) = 2n + 15

Time Complexity: O(n)

Purpose: To add the seat node in a single linked list at a given position index. Iterate over each node of the list and keep a count for each pass of the node in the linked list to keep track of the index to add the node. When the list reaches the index that the user wants to add then add the node and redirect the node's pointers accordingly.

7.2 Book A Seat Function

7.2.1 bookSeat Method

Algorithm 5 bookSeat

Input: A single LinkedList, *availableSeatList*
Output: To add a seat node to the book seat list and remove the booked seat from the available seating list.
Begin Algorithm: bookSeat

$input(intuserSeatInput)$
$indexCount = 0$

while $(currentAvailable.next\ \ not\ \ equal\ \ nullptr)AND(currAvailableSeatNode.data\ \ not\ \ equal\ \ userSeatInput)$ **do**
  $indexCount \leftarrow indexCount + 1$
  $currAvailableSeatNode \leftarrow currAvailableSeatNode.next$
end
if $userSeatInput == currAvailableSeatNode.data$ **then**
  $insertAtEnd(userSeatInput)$
  $print\ seats$
  $availableSeatList.deleteNodeAtPosition(indexCount)$
  $return$
else
  $print\ UnabletoBook!$
end
End Algorithm: bookSeat

Frequency Count Analysis:
user input of seat: 1 operation
indexCount initialization: 1 operation
while loop iterations: n operations
while loop conditional checks: 2*n comparisons
indexCount increment: n operations
shift next SeatNode: n operations
if condition check: 1 comparison
insertion at end (O(n)): n operations
print seats (O(n)): n operations
deleteNodeAtPosition: n operations
Frequency Count: F(n) = 8n + 3
Time Complexity: O(n)
Purpose: To allow the user to book a seat based on the given a hand selected availability list. Make sure that the seat node exist in the list in the available list to be able to book the seat node to be stored in the booked list.

7.3 Cancel A Seat Function

7.3.1 cancelASeat Method

Algorithm 6 cancelASeat

Input: LinkedList, *availableSeatList*
Output: To remove the selected booked seat from the booked seat list and add the canceled seat back to the available seating list
Begin Algorithm: cancelASeat

$input(intuserSeatInput)$
$indexCount \leftarrow 0$

if $head == NULL$ **then**
  print− >Can't be booked
  $return$
end
$currBookedSeatNode \leftarrow head$

while $((currentAvailable.next$ *not equal* $nullptr)$ *AND* $(currAvailableSeatNode.data$ *not equal* $userSeatInput))$ **do**
  $indexCount \leftarrow indexCount + 1$
  $currBookedSeatNode \leftarrow currBookedSeatNode.next$
end
if $currBookedSeatNode.data == userSeatInput$ **then**
  $currAvailSeatNode \leftarrow availableSeatList.head$
  $indexCountAvail = 0$
  $deleteNodeAtPosition(indexCount)$

  while *(currentAvailable.next not equal nullptr) AND (userSeatInput* $>$ *currAvailSeatNode.data)* **do**
    $currAvailSeatNode \leftarrow currAvailSeatNode.next$
    $indexCountAvail \leftarrow indexCountAvail + 1$
  end
  $availableSeatList.addNodeAtPosition(indexCountAvail, userSeatInput)$
else
  print $-$ > uncancellable seat!
end
End Algorithm: cancelASeat

Frequency Count Analysis (Worst Case):
user input of seat: 1 operation
indexCount initialization: 1 operation
check head isn't NULL: 1 comparison
assign head to current booked node: 1 operation
while loop iterations: n operations
while loop comparisons: 2*n comparisons
inside while loop operations: 2*n operations
if block userinput comparison: 1 comparison
assign availableSeats head to current Node: 1 operation
indexCountAvail assignment: 1 operation
deleteNodeAtPosition(indexCount) (O(n)): n operations

while loop iterations: n operations
while loop comparisons: 2n comparisons
while loop operations: 2 operations
addNodeAtPosition (O(n)): n operations

Frequency Count: F(n) = 10n + 9

Time Complexity: O(n)

Purpose: To allow the user to cancel a seat from the booked list. Make sure that the seat node exist in the booked list to be able to cancel the seat node and then add the canceled seat back into the available seat in a sorted order.

7.4 Showing List of Available and Booked Seats Function

7.4.1 toPrint Method

Algorithm 7 toPrint

Input: The given single linked list
Output: Print the each seat from the full linked list
Begin Algorithm: toPrint

Initialize SeatNode *temp = head
if $head == NULL$ **then**
  print: There are no items
  return
end
while $temp\ not\ equal\ NULL$ **do**
  print the seat number and address
  $temp = temp.next$
end
Endline
End Algorithm: toPrint

Frequency Count Analysis (Worst Case):
temp initialization: 1 operation
if block head comparison: 1 comparison
while loop iterations: n operations
while loop comparisons: n comparisons
inside while loop operations: 2*n operations
End line to distinguish other print statements: 1 operation
Frequency Count: F(n) = $4n + 3$
Time Complexity: O(n)
Purpose: To print all the element and seat nodes in a given single linked list

7.5 Showing and Interacting With the Main Menu

7.5.1 mainMenuCatalog Method

Frequency Count Analysis (Worst Case):

userInput initialization: 1 operation
avalSeatingList initialization: 1 operation
bookedSeatingList initialization: 1 operation
maxSeatInRow initialization: 1 operation
for loop iterations: n operations
for loop comparisons: n comparisons
inside for loop operations: n operations
while loop iterations: n operations
while loop comparisons: n comparisons
inside while loop operations: 20 operations
inside while loop comparisons: 5 comparisons

Frequency Count: F(n) = $5n + 29$

Time Complexity: O(n)

Purpose: This is the UI feature in the terminal where the user can interact with different options seen in the the seat reservation system.

Algorithm 8 mainMenuCatalog

Input: userInput
Output: Linked list for booked seats or available seats, Booking a seat, canceling a booked seat, or exit
Begin Algorithm: mainMenuCatalog

Initialize string userInput

Initialize SingleLinkedSeatingList avalSeatingList

Initialize SingleLinkedSeatingList bookedSeatingList

int maxSeatInRow = 12 (host can choose number based on their need)

for $i = 0$ *to maxSeatinRow* **do**
  avalSeatingList.insertAtEnd(i)
end
while *true* **do**
  print: "Event Ticket Booking System: MAIN MENU"
  print: "Please Select an Option"
  print: "1. Book a Seat"
  print: "2. Cancel a Seat"
  print: "3. Show Available Seats (All Booked Seats List)"
  print: "4. Exit"
  print: "5. Show All Open Seats"
  print: "Insert Selection Number: "
  userInput ← user's menu choice
  if $userInput == 1$ **then**
    print: "You selected to book a seat"
    bookedSeatingList.bookSeat(avalSeatingList)
  end
  else if $userInput == 2$ **then**
    print: "You selected to cancel a seat"
    bookedSeatingList.cancelASeat(avalSeatingList)
  end
  else if $userInput == 3$ **then**
    print: "You selected display available seats. (booked Seats)"
    bookedSeatingList.toPrint()
  end
  else if $userInput == 4$ **then**
    print: "You selected to terminate the program"
    break from the loop
  end
  else if $userInput == 5$ **then**
    print: "You selected display all open seats"
    avalSeatingList.toPrint()
  end
  print: "You selected an invalid option. Please reselect your input"
end
End Algorithm: mainMenuCatalog

8 Algorithm Method Descriptions for Building Database (October 14-20)

This section discusses the algorithms and methods that build the structure of the database which in include helping store, organize, and get data. This can be seen as the GET query methods get the seats of the flight, display the available flights, and display the user information. The POST query method creates or removes the flights and seats needed for the flight reservation system. The query patch is used to describe the updates and changes of state being made on the data in the flight reservation system.

9 Back End System Architecture Data Representation (October 14-20)

9.1 Relationships of Data Entities in Back End

This section illustrates the relationships of the simulations features working together and note that these results are a work in progress. This can be seen as in Figure 1 the UML diagram displays the data entities representation while the team was testing the back end framework. This scenario describes how Flights and Seats interact with each other to organize the respective operations for each entity. The team found that each flight will have a one to many relationship with the seat as a flight can have an infinite number of seats, but only one seat can belong to a user on a single flight.

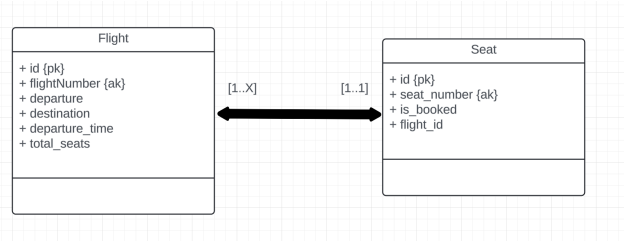

Figure 1: Current Database Respective Data Entities Representation

After the team tested the interactions between each data entities, the team want to expand the fight reservations capabilities by enabling features based on the users role like a customer or admin role. The team's finalize UML diagram to articulate this vision can be seen in Figure 2. This finalized database representation illustrates the hierarchy for a given user, which the data attributes are represented as an abstract class for the customer. The new relationship between the user and seat is also one to many as a user can reserve many seats, but there can be one seat to a user. In this new representation just separates the features for each user as the customer user are only allowed to book and cancel a flight whereas the admin have full control to manipulate the system.

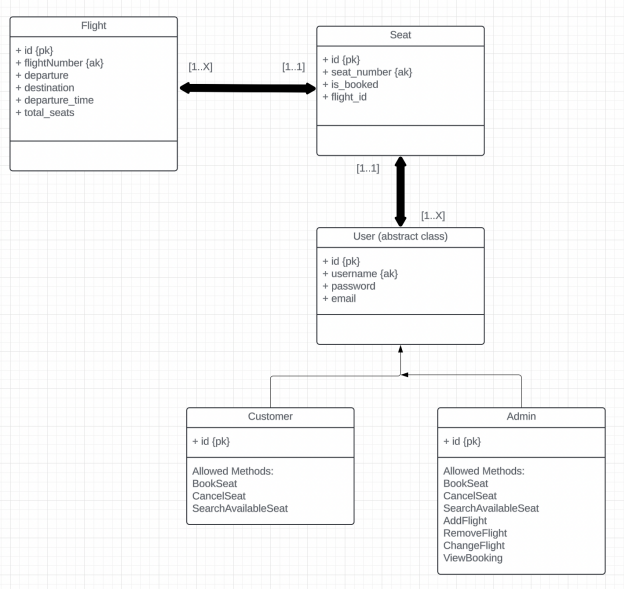

Figure 2: Final Database Representation

9.2 Evaluating Known Use Cases To Create Database

When the team started creating a database to store the data from the user's interactions to reserve a seat on a given flight, the team's initial approach was to use django as it already created the interface that were specific to an admin and user roles. Also, django organizes the ORM to be defined as python class, which the team was first comfortable to play around with. In the github link `https://github.com/sajna-j/flight-reservation/tree/django-backend-setup` illustrates the procedure to run the django backend database. In the manage python file, createsuperuser command allows the user to input their personal information and create a profile on the flight reservation system seen in Figure 3.

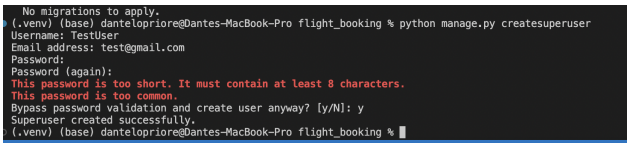

Figure 3: Confirmation on Successfully Creating A User

Then, you can populate flight to be added into the database using populateFlights flag in the manage python files seen in Figure 4.

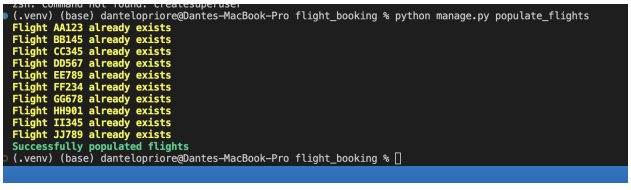

Figure 4: Creating New Flights to Database

After populating the database with flights, the team run the server command in the terminal to access the applica-

tion on the website at the url link `http://127.0.0.1:8000/admin/`. The Django built-in web interface and dashboard to represent the database can be seen in Figure 5 where the admins can directly access and manipulate flight and seating information. When having the Admin credentials, the team can display the all the models listed in the database that include the relationships between the flights and users to view and edit current records, enforce validation constraints on certain features, and provide real-time updates.

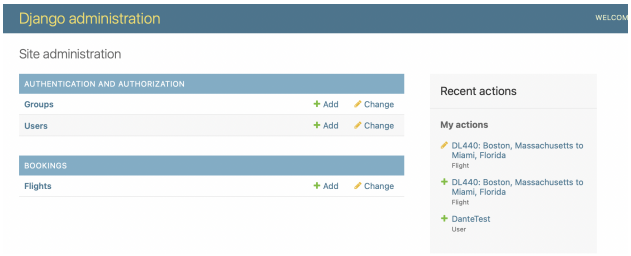

Figure 5: Django Dashboard

In Figure 6, shows how the user can add a user to interact with the database to create a user.

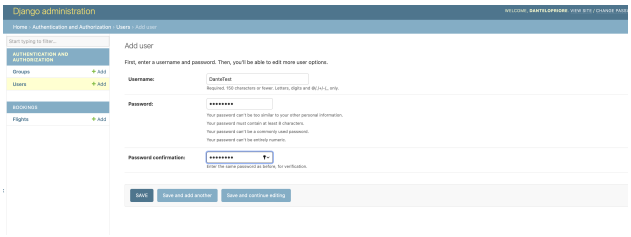

Figure 6: Photo to How the User Can Interact With Database

In Figure 7, shows how the user can successfully add a user to interact with the database to create a user.

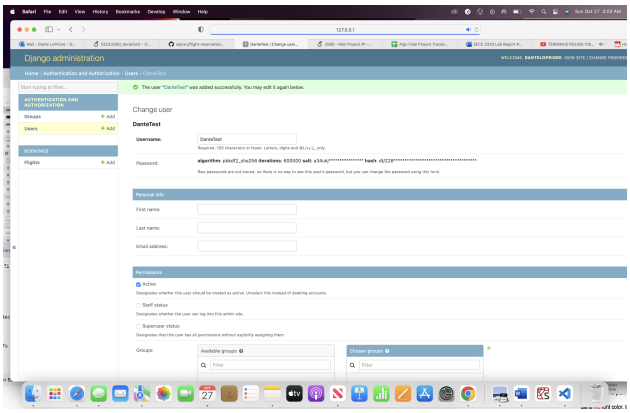

Figure 7: Photo to How the User Can Interact With Database

In Figure 8, the figure shows the all the flights dashboards where the user can reserve a seat for a given flight.

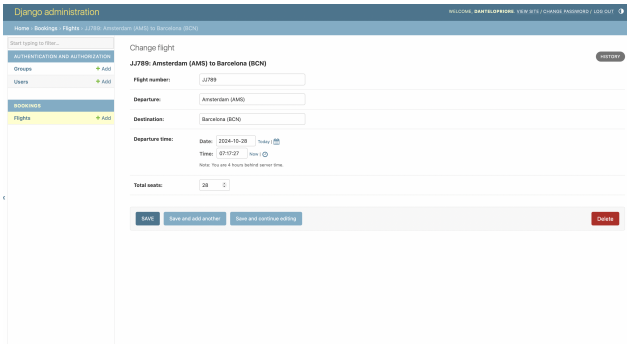

Figure 9: Adding Flight With Given Seats to Be Reserved

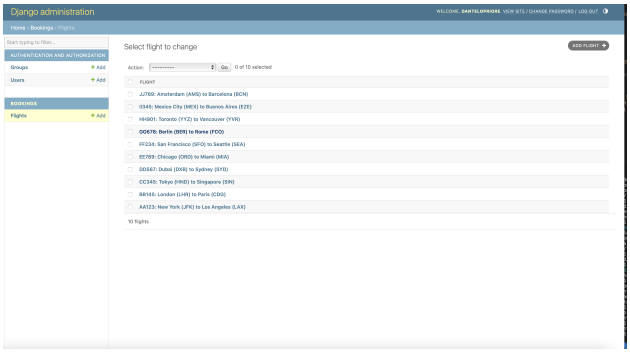

Figure 8: Reserve A Seat For Flight

In Figure 9 and 10, the figure confirms all the flights dashboard where the user can reserve a seat for a given flight can add a flight

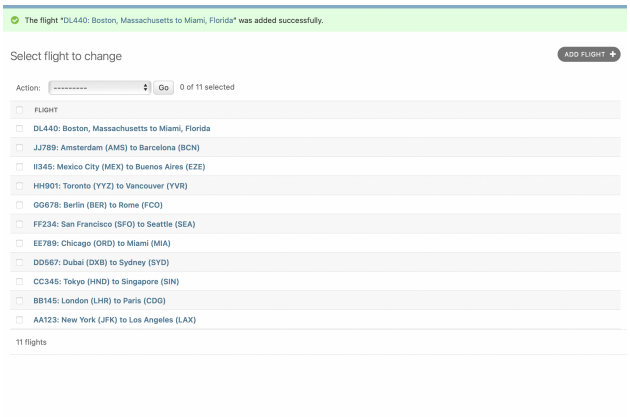

Figure 10: Confirmation on Flight Being Added

In Figure 11, shows the admin access for the group's flight reservation system.

The Flask version of this database can be seen in this github link `https://github.com/sajna-j/flight-reservation/blob/flask-backend/README.md`

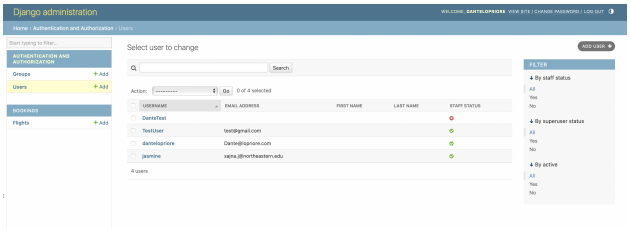

Figure 11: Admin Access Based on Each Person's Role

10 Current Challenges (October 14-20)

10.1 Django's Framework is Too Complex

After evaluating all the team experience with creating databases, the team found that creating a database with flask and sql would be very easy to learn and implement rather than using Django. Even though our team made some progress with building the database in Django, the features in the flight simulation project were simple and using Django was overkill as Django was a required more setup and structure to access certain items. The team would create a database using SQL and Flask for the data entry submissions for the flight reservation system because it is easier to represent and understand the relationships between the current data entities and the database already have a pre-defined scheme and query language to work off of compared to Django. In the team's progress for this iteration was to test and see what works and does not work, and the team identified that using SQL and Flask is best to create a database in this scenario and considering the expected timeline of the full project to be completed.

10.2 Lack of Emphasis on What the Project is About

While completing the required back end component for the flight simulation project, the team found that use database to write and read data would defeat the purpose of the project's highlighted data of applying the data structures we used in class. Therefore, the team needed to figure out a way to dynamically preform these features soley with the linked list and sorting algorithms to ensure the project requirements are meet for this type of course.

11 Anticipated New Goals For the Next Iteration (October 14-20)

11.1 Switching Main Objective to Build GUI first before Building Database

After the team put a strong emphasis on researching and testing the databases, the team decide to reorganize the project's priorities which was to highlight how the team used the data structures in class to help us produce the features needed for the simulation. The team felt that building a GUI will help enable the team to better visualize that the sorting algorithms and referencing nodes in a linked list and understand the core functionality of the simulation works based on the user's interactions with the simpler system. This supports the team's shifted focus on developing a GUI before creating a database.

12 Team Structure for Development (October 27-November 2)

12.1 Determining a Task List for Next Iteration

To better structure work for this iteration, the team decided on what stage should be reached by the end of Iteration 5–a working backend that left work for only the GUI to be made. With this goal in mind, we concluded on a list of tasks to complete:

1. The "Bare Logic" should be complete with Flight objects

a. Users can get available flights
b. Users can get all indirect and direct flights from A to B
c. Users can book a seat from a flight
2. A Script that populates all the Flights our users will have access to
3. Sorting methods should be implemented
a. Seats can be sorted by cost, class, number
b. Flights can be sorted by duration, and time departure
4. Filters should be implemented
a. Grabbing only Flights that are from A to B
5. Route the bare logic functions using the non-database structures to URLS with Flask

13 Updating Bare Logic (October 27-November 2)

13.1 Brief Summary on Changes (New Features)

After finishing the last iteration, the team expanded on the flight reservation system's functionality and navigation by creating the basic logic to support this new system. New major features added to the bare logic include:

1. **Allow User To Reserve Seats With Different Status Type:** In the new version of the bare logic allows the user to reserve or cancel a economy, business, and first class seat for a given flight. The distribution of these different types of seating are dependent on the size of the plane and user demand which can be seen as 20 percent of seatings are allocated for first class, 20 percent of seatings are allocated for business class, and the rest are set to be economy class. The pricing for the seats are set based on a formula and distribution similar to how actual airlines price out their flights by considering the flights in-demand value, number of seats, duration.

2. **Allow User To Filter and Sort Through Flights Based on their Needs:** The new codebase gives power to the user to allow them to find the desired flight they want to book by enabling new features like sorting the flights based on time, departure airport, cost, type of seat. Also, the team connected the bare logic component to the back end to make a visually appealing UI to allow the user to interact with the system.

13.2 Brief Summary on Fixing System (Fixing Major Bugs)

Also, more notably the team's new additions to the fixed any major bug fixes and optimization techniques to improve run time. New major bug fixes to improve the flight reservation system included:

1. **Improve How Indirect Flights Are Processed:** The team optimized the BFS algorithm to be an O(n^2) solution to improve run time and fix known bugs such as the searching algorithm did not consider overnight flights like a flight from 11pm one day to 2am the next day to be flight indirect flight and in the event were their are two duplicate flight paths from same departing airport at different times, it will only consider the earlier time flight to be in the computation for the indirect flight which was prove to be wrong as it should consider all known valid flights. The team fixed this issue by create special cases to encounter for these events

2. **Fixed How Each Flight's Information was Printed:** The team found a bug when the system printed out each flight as the team noticed storing the flight's address and calling the address to print each flight did not wrong properly so we fixed the printing functions to print each flight directly.

13.3 Redefining Data Entity Relationships

While considering the new bare logic enables these features, their were some small changes in how the code structure is illustrated to reflect how the new features were properly implemented. Figure 12 highlights the UML diagram to display the data entities representation to make up the bare logic.

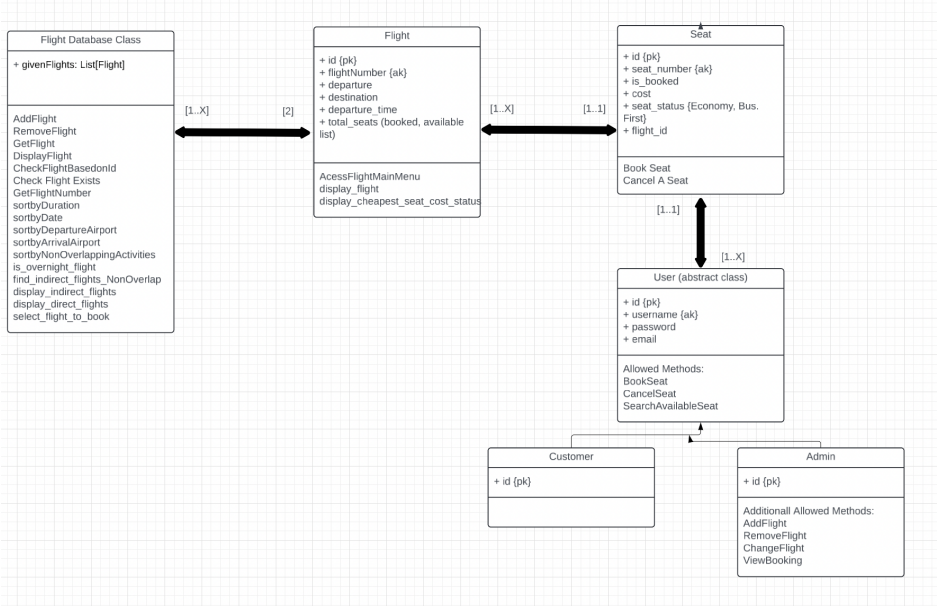

Figure 12: A UML Diagram to Describe Data Entities in the Bare Logic

The improved UML diagram includes a flight database class that manages the multiple flights stored in the system, which is practically storing a list of flight objects. This is were the interactive menu would be stored to navigate through each flight's details, adding flights, deleting flights, and filtering flights, and selecting the flights from the list. The new relationship between the Flightdatabase to the Flight can be one to many as the flight database needs to have atleast one flight to operate the system and the database can have as many flights as it wants. The relationship of the Flight to the Flightdatabase as referencing to the amount of single linked lists for each flight that are 2 one for booked flights and another for available flights. Thus, the LinkedLists and Flights relationship is deemed a composition relationship since the existence of the linked lists are dependent on if the Flight exists in the Flightbase. Also, the known changes to the attributes of the seat were altered to include the seat's status and cost based on if the seat is first class, business class, and economy class.The data structure's multiplicity seen in the between the flight and seat and the user's control over reserve the seat still remain the same with the respective changes. Overall, the team is please with the results for the bare logic to serve as a foundation to help structure the backend structure.

14 Updated Algorithm Design for System's Basic Logic (October 27-November 2)

The current pseudcode list in this section shows all the methods that make up the flight reservation system. The bare logic of the simple system are the sorting and navigation of traverse through seats, and allowing the user to dynamically enter a seat that is available or booked. The bare logic files can be seen in this github link to the branch of the working python file `https://github.com/sajna-j/flight-reservation/blob/bare-logic-in-python`.

14.1 FlightDatabase Booking System Sorting Direct and Indirect Flights Functions

14.1.1 Find Indirect Flights NonOverlap Method (BFS Algorithm)

Frequency Count Analysis:

initialize known indirect flight variable: 1
initialize known departure map dictionary variable: 1

Algorithm 9 Indirect Flights BFS NonOverlap Method :

Input: The string value of the desired airport departure location, *source*, The string value of the desired airport arrival location, *dest*

Output: A list of correct indirect flights based on the set arrival and departure airports.

Begin Algorithm: Indirect Flights BFS NonOverlap Method

Initalize $knownIndirectFlights$ = []
Initalize $departureMap$ = empty dictionary
for *(flight in self.givenFlights)* **do**
  if *(flight.departureLocation not in departureMap)* **then**
    Assign departureMap[flight.departureLocation] = empty list
  end
  departureMap[flight.departureLocation].append(flight)
end
queue = deque([(source, emptylist, NULL pointer, NULL pointer)]) Initalize visitedSet = set()
while *(queue is not NULL)* **do**
  Set currentLocation, flightPath, lastEndTime, lastDate = queue.popleft()
  if *(currentLocation == dest)* **then**
    flightNumbers = [flight.flightNumber for flight in flightPath] **if** *(flightNumbers not in knownIndirectFlights)* **then**
      knownIndirectFlights.append(flightNumbers)
    end
    continue
  end
  if *(currentLocation in departureMap)* **then**
    for *(nextFlight in departure*$_m ap[currentLocation]$) **do** **end** return know
      Set visitKey = (nextFlight.flightNumber, nextFlight.timeInterval, nextFlight.date)
      if *(visitKey in visitedSet)* **then**
        continue
      end
      visitedSet.add(visitKey)
      if *(flightPath does not exist)* **then**
        Set newFlightPath = [nextFlight] queue.append((nextFlight.arrivalLocation, newFlightPath, nextFlight.timeInterval[1], nextFlight.date))
      else
        Set lastFlight = flightPath[-1] Set lastEndTime = lastFlight.timeInterval[1] Set nextStartTime = nextFlight.timeInterval[0]
        Assign isOvernight = self.isOvernight$_f light(lastEndTime, nextStartTime) Assign dif ferenceDatesTimes = (nextFlight.date - lastFlight.date).days$
        if *((lastEndTime <= nextStartTime or isOvernight) and (lastFlight.date <= nextFlight.date) and (differenceDatesTimes <=*2)) **then**
          newFlightpath = flightPath + [nextFlight] queue.append((nextFlight.arrivalLocation, newFlightPath, nextFlight.timeInterval[1], nextFlight.date))
        end
      end
    end
  end
End Algorithm: Indirect Flights BFS NonOverlap Method

iterate through each flight in flight list : n
check if the departure location is in the dictionary if condition : 1
if condition is true assign the empty list to new departure map dictionary : 1
if condition is false assign the flight to new departure map dictionary based on departure key: 1
initialize a queue to store the known flight path: 1
iterate over a queue via a while loop : n
set variables for the queue : 1
check the if condition for the current location is the destination : 1
get a list of the known flight number iterating through for loop of known flights : n
check if the flight is located in the indirect flight list: 1 comparison
add the flight number to the knownIndirectFlights: 1 comparison
if condition to check if the current flight in departure mapping: 1 comparison
iterate through each flight in the departure map based on the current location: n^2 comparison
initialize a visited key: 1
check if a visited key in the visited set and add the key: 2
check if a current flight path exists: 1
assign and create flight path : 1
append the current flightpath to the queue : 1
initialize variables to check known constraints to define an in: 5
assign and create flight path and append the current flightpath to the queue : 2
return indirect flight list : 1

Frequency Count: F(n) = n^2+7n+22

Time Complexity: O(n^2)

Purpose: This function allocates the correct list for indirect flights based on a given source and destination airport that the user inputted. The BFS algorithm gets the all the valid indirect flight paths without excessive layovers and no overlapping flights. The BFS algorithm starts by creating a dictionary to represent a departure mapping system that stores the flights based on their departure locations to allow the search algo to pull from if needed. The queue is initialize to investigate all known possible flight paths in the BFS tree to reach from the departure location to the arrival location. The a valid indirect flight can be consider on to the ongoing list if the flight meets the requirements of being within a max layover of two days, prevents revisted already past flights using a set, check if the time of the next flight is valid and going to the correct path to the destination.

14.1.2 Display Indirect Flights

Frequency Count Analysis:

get all the known indirect flights: 1
set a count variable: 1
check if the indirect flight list has any flights using if condition : 1
iterate through each indirect flight path in indirect flight list : n
check if the departure location is in the dictionary if condition : 1
check if the the flight path in the indirect fight path has more than one flight to be considered an indirect flight and increment option : 2
iterate through each indirect flight with the valid indirect flight path via for each loop: n^2
use if condition to check if the indirect flights is not a direct flight: 1
print the current flight: 1

Frequency Count: F(n) = n^2+n+8

Time Complexity: O(n^2)

Purpose: To output all the the valid indirect flight lists

Algorithm 10 Display Indirect Flights

Input: The string value of the desired airport departure location, *sourceLocation*, The string value of the desired airport arrival location, *destLocation*

Output: A printed list of correct indirect flights based on the set arrival and departure airports.

Begin Algorithm: Display Indirect Flights

Acquire allKnownIndirectFlights = findIndirectFlightsNonOverlap(sourceLocation, destLocation) Set count = 1
if *(allKnownIndirectFlights is not NULL)* **then**
  for *(curOverallFlightPath in allKnownIndirectFlights)* **do**
    if *(len(curOverallFlightPath) > 1)* **then**
      print $->$ Indirect Flight Option count Increment count by 1
    end
    curIndirectFlight in curOverallFlightPath **if** *(len(curOverallFlightPath) > 1)* **then**
      givenFlights[self.getFlight(curIndirectFlight)].displayFlight()
    end
  end
end
print("No indirect flights available.")
End Algorithm: Display Indirect Flights

Algorithm 11 Display Direct Flights

Input: The string value of the desired airport departure location, *sourceLocation*, The string value of the desired airport arrival location, *destLocation*

Output: A printed list of correct direct flights based on the set arrival and departure airports.

Begin Algorithm: Display Indirect Flights

Set count = 1
Set printErrorFlag = True
if *(givenFlights is not NULL)* **then**
  for *(curflight in givenFlights)* **do**
    if *(curFlight.departureLocation == sourceLocation) and (curFlight.arrivalLocation == destLocation)* **then**
      print $->$ Indirect Flight Option count Increment count by 1 print $->$ currentFlight
    end
  end
end
if *(PrintErrorFlag is True)* **then**
  print "no available flights"
end
End Algorithm: Display Direct Flights

14.1.3 Display Direct Flights

Frequency Count Analysis:
set print error flag : 1
set a count variable: 1
check if the list of flights has any flights using if condition : 1
iterate through each flight in the flight list : n
check if the their are any direct flights that depart from the departure location to the arrival airport : 2
print the direct flight wither : 2
use if condition to check if the indirect flights is not a direct flight: 1
print the current flight: 1
check if the error condition is set: 1
throw error meessage: 1
Frequency Count: F(n) = n+10
Time Complexity: O(n)
Purpose: To output all the the valid direct flights by seeing if the user's requested departure and arrival airports have a flight that exists

14.2 Defining Function Seen in FlightDatabase Booking System

14.2.1 sortbyDepartureAirport

Algorithm 12 sortbyDepartureAirport method

Input: The string value of the desired airport departure location, *source*
Output: A printed list of correct direct flights based on the set departure airport.
Begin Algorithm: sortbyDepartureAirport

for *(curFight in self.givenFlights)* **do**
  if *(source == curFight.departureLocation)* **then**
    curFight.display$_flight()$**end**
  end
  End Algorithm: sortbyDepartureAirport

Frequency Count Analysis:
iterate through each flight in the flight list: n
if condition to see if the flight contains the departure airport need to be outputed : 1
print the current flight : 1
Frequency Count: F(n) = n+2
Time Complexity: O(n)
Purpose: To output all the valid direct flights filter by the user's selected departure airport

14.2.2 sortbyArrivalAirport

Frequency Count Analysis:
iterate through each flight in the flight list: n
if condition to see if the flight contains the arrival airport need to be outputted : 1
print the current flight : 1
Frequency Count: F(n) = n+2
Time Complexity: O(n)
Purpose: To output all the valid direct flights filter by the user's selected arrival airport

Algorithm 13 sortbyArrivalAirport method

Input: The string value of the desired airport departure location, *source*
Output: A printed list of correct direct flights based on the set arrival airport.
Begin Algorithm: sortbyArrivalAirport
for *(curFight in self.givenFlights)* **do**
  if *(arrival == curFight.arrivalLocation)* **then**
    curFight.display$_flight()$**end**
  end
  End Algorithm: sortbyArrivalAirport

14.2.3 sortbyDuration

Algorithm 14 sortbyDuration method

Input: The list of flights
Output: A printed list of direct flights based on acending order of their flight time and duration.
Begin Algorithm: sortbyDuration

Initialize index = 0
Initialize jindex = 0
for *(index in range(len(self.givenFlights)))* **do**
  for *(jindex in range(len(self.givenFlights)))* **do**
    if *(self.givenFlights[index].duration < self.givenFlights[jindex].duration)* **then**
      Set temp = self.givenFlights
      Set temp2 = self.givenFlights[jindex]
      Reassign givenFlights[jindex] = temp
      Reassign givenFlights[index] = temp2
    end
  end
end
End Algorithm: sortbyDuration

Frequency Count Analysis:
initialize an index to iterate over an index at i: 1
initialize an index to iterate over an index at j: 1
iterate over the the list of flight via a outer for loop: n+1
iterate over the the list of flight via a inner for loop: n^2+1
check of the duration of the current flight is smaller than the next flight using if condition: 1
sort items by swapping the two items at at index i and index j: 4
Frequency Count: F(n) = n^2+n+9
Time Complexity: O(n^2)
Purpose: To sort the flights by acending order of flight duration for the whole list.

14.2.4 sortbyDate

Frequency Count Analysis:
initialize an index to iterate over an index at i: 1
initialize an index to iterate over an index at j: 1
iterate over the the list of flight via a outer for loop: n+1

Algorithm 15 sortbyDate method

Input: The list of flights
Output: A printed list of direct flights based on acending order of their flight data and sceduled time.
Begin Algorithm: sortbyDate

```
Initialize index = 0
Initialize jindex = 0
for (index in range(len(self.givenFlights))) do
    for (jindex in range(len(self.givenFlights))) do
        if (self.givenFlights[index].date < self.givenFlights[jindex].date) then
            Set temp = self.givenFlights
            Set temp2 = self.givenFlights[jindex]
            Reassign givenFlights[jindex] = temp
            Reassign givenFlights[index] = temp2

        end
    end
end
```

End Algorithm: sortbyDate

iterate over the the list of flight via a inner for loop: n^2+1
check if the data of the current flight is earlier than the next flight using if condition: 1
sort items by swapping the two items at at index i and index j: 4
Frequency Count: F(n) = n^2+n+9
Time Complexity: O(n^2)
Purpose: To sort the flights by ascending order of flight scheduled time for the whole list.

15 Script to Populate Flights (October 27-November 2)

15.1 Adding Multiple Flight Objects

There will be 30 flights added as a script to demonstrate the functionality of choosing a flight for reserving a seat. The flights are comprised of 5 major hubs across the United States.

1. Boston, BOS
2. Los Angeles, LAX
3. New York, JFK
4. Denver, DEN
5. Chicago, ORD

From these 5 major hubs, the flights all have different dates, and time intervals to to demonstrate the ability to find direct and indirect flights. The cases that are tested are:

1. Finding direct flights
2. Overnight direct flights
3. Same-day indirect flights
4. Different-day indirect flights

15.2 Finding direct flights

Flights are populated so it has a departure and arrival destination. The users will be able to find direct flights as long as the departure and arrival locations are part of the hubs listed in section 16.1 and are not the same location.

15.3 Overnight direct flights

Some flights are populated with time intervals that start with a higher number compared to the second number. This is meant to show the first number as a late time and the second number is past midnight into the next day. The FlightDatabase is still able to find direct flights.

15.4 Same-day indirect flights

Finding the indirect flights from two different hubs is found in the FlightDatabase logic and the populated flight is able to find connecting flights. An example of this is found when the user asks for indirect flights from BOS to LAX. One flight option is BOS to LAX via a connecting flight in DEN all on a 11/18/2024.

15.5 Different-day indirect flights

To test the indirect flights, two factors are considered. That would be the maximum amount of connection to be considered an indirect flight, and the maximum time gap between flights to be considered an indirect flight. Of the 30- flights populated, the FlightDatabase logic can list connecting flights that are separated by X day(s) and do not connect to any flights that happen before the first flight. An example of this would be a flight from BOS to DEN. One of the indirect flight options listed will be a flight from BOS to ORD on 11/21/2024 and a connecting flight from ORD to DEN on 11/22/2024.

16 Revising the Flask Backend (October 27-November 2)

16.1 Removing SQL Databases

To better represent the subjects and core-concepts of the class, we removed the use of a SQL Database for our Flights in the preliminary Flask app.py script. We transitioned this to route the output of our bare logic that uses our custom LinkedLists, sorting, and filtering algorithms to URLs for the frontend.

16.2 Requests Needed for Customers

To build a working backend on our localhost, and accessible via Postman, we solidified and designed the requests we wanted users to be able to make as the following:

1. GET a flight by its ID
2. GET all direct flights for a requested source destination
3. GET all indirect flights for a requested source destination
4. GET all seats of a flight, sorted by cost, status, or ID
5. GET a specific available seat of a flight
6. POST a flight seat as booked
7. POST a flight seat as free (cancelled)

To create each request, we designed a function for each that would grab this data from the FlightDatabase instance in which all our flights are populated. Each endpoint-routed-function uses the existing functions for grabbing seats, flights, etc with adjustments so that the functions return objects that can be converted to JSON format.

16.3 Testing the Endpoints

To test each endpoint, requests are made to the running Flask app via a custom-made collection of Postman Requests, as seen in Figure 12.

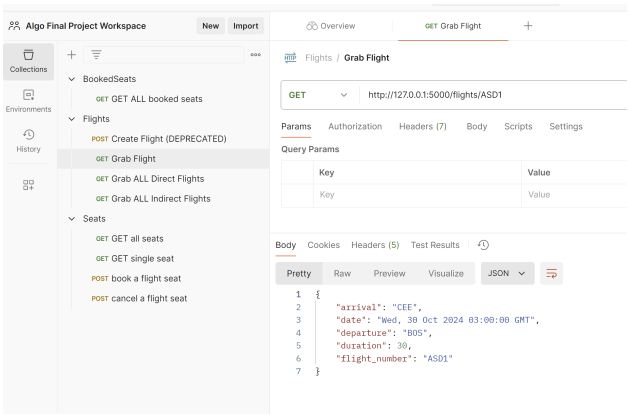

Figure 13: Working Endpoint Test Setup with Postman

17 User Interface Development (November 3-11)

17.1 Building the UI

When building the user interface, we compiled the features we wanted customers to have. These features resemble the endpoint requests we developed when building the backend. More specifically, we wanted a main search point for customers to view all the flights for their destination and preferences. Then once selecting a flight, users should have a clear window displaying all the seats of the flight they can choose from and rearrange based on cost, status, or ID. Each seat should have a button for booking/canceling.

To build the UI, we developed an HTML template using the Bootstrap framework in HTML and CSS to create a grid-screen split into four main sections–a search bar for filtering all flights, a window to view said flights, a window to view the seats of a selected flight, and a window to view the user's currently booked seats across different flights. The final results can be seen below.

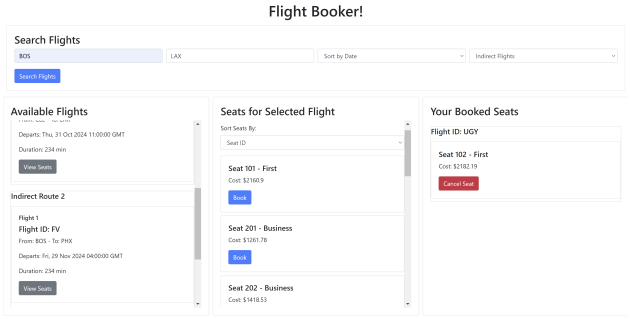

Figure 14: User Interface for Booking Flights

17.2 Additions to the Backend

While constructing the UI, we initially developed three main sections–the same as previously mentioned, excluding the currently booked seats window. This gave us insight into developing the last window when we realized one feature that was missing–users couldn't cancel a seat since booked seats disappeared from the seats view of each flight. To address this, we developed a new endpoint in Flask that accesses the booked seating list of each Flight to create a new UI window to showcase all the seats that can be canceled.

17.3 Additions to the Frontend (November 19)

To ensure users follow along with the sorting features we built into the indirect flights view, we added additional information about the duration to the title of each route card. To do so, changes were made to the backend in which the endpoint that receives the indirect flights now receives the total duration of each indirect route as a key-value pair along with the list of flights in the route. This allows users to directly verify that indirect flights are properly being sorted by duration.