# CSE183 Final Project - Pocket Planes Companion App

Andrew Trent - artrent@ucsc.edu
Yukti Malhan - ymalhan@ucsc.edu
Evelyn Johnson - evijohns@ucsc.edu

## Description:

Pocket Planes Companion is a comprehensive online hub, crafted to empower players of the popular mobile game, Pocket Planes, by facilitating collaboration and intricate in-game calculations. The platform is designed with a host of features that enhance user engagement and strategic play.

At the heart of Pocket Planes Companion, players will discover a dynamic trading post, specifically tailored to enable secure exchanges of rare in-game parts. Users can list their available parts, the items they desire, and their preferred contact information. The site initially showcases an array of information from diverse players, but users can streamline their search by entering a specific part into the search bar. The system then curates a list of potential trading partners offering the specified part, along with their requested exchanges.

To ensure an accurate trading platform, users have the ability to update or delete their information, thereby maintaining an up-to-date inventory of their available parts.
Furthering the strategic capabilities of Pocket Planes Companion, players can create and analyze flight plans. The site's advanced algorithms generate detailed flight statistics, assisting users in maximizing their in-game effectiveness.

Additionally, the platform offers a unique calculation tool that identifies viable aircraft for travel between selected cities, contributing to smarter decision-making and improved gameplay.
Every feature within Pocket Planes Companion  is readily accessible through a user-friendly sidebar, ensuring players have unrestricted access to all tools at any time. The result is a website that truly elevates the Pocket Planes gaming experience

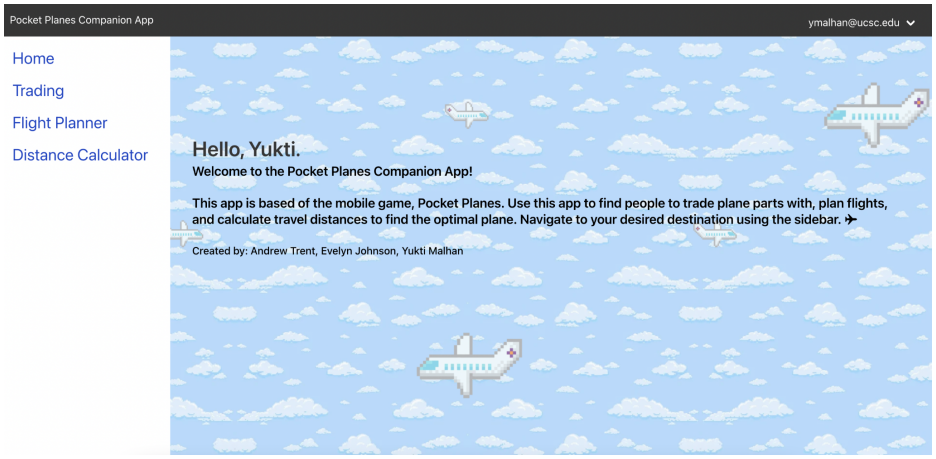
## Main Pages:

Our app has a total of 4 pages - the home page, trading page, flight planner page, and distance calculator page. Each page hosts a variety of features to assist users in all their Pocket Plane needs!

## Home Page:

*Summary:*

The home page principally functions as an introductory portal to the application, providing a succinct overview of its purpose, operational guidance, and the identity of its creators. It refrains from housing any supplementary functionalities beyond the embedded sidebar, a navigational tool purposed for facilitating access to the application's diverse pages.




## Trading:

*Summary:*

The trading page operates as a digital interface, facilitating users to input their contact information and articulate a requisite list of in-game aircraft components. The centralized table functions as a community board, presenting users with a comprehensive overview of available resources and their respective offers from the collective user base. An integrated search functionality is present to refine the scope of results, confining them to only those users who possess the item in question.

Each aircraft component registered to a user is subject to reservation by other users, provided it is not already encumbered by a pre-existing reservation. This system fosters a reciprocal exchange environment where users can dynamically respond to the supply and demand of in-game items. For monitoring transactions, each user has access to an 'Offers and Reservations' page, where they can view a catalog of bids placed by other users on their components, alongside a record of reservations they have instituted for items belonging to others. This interactive page ensures transparency in the exchange process and promotes equitable trading practices within the community.

*Data Organization:*

Backend (Controllers.py, models.py, index.js):

This page utilizes two database tables. The first stores user information. It holds their user ID, which is automatically generated, and it also has fields for username, in-game friend code, a

discord username, and a reddit username. The user can fill out the latter fields using a form, accessible from the trading page.

The second table is used for managing parts. It has fields for storing the ID of the user that owns the part, the name of the part, who its currently reserved by, and whether its a 'have' part or a 'seeking' part. There are also some fields (prepended with 'original' that are used to retrieve further reservation information.

There are multiple controllers used for managing this page. The first few are for managing the tables displayed to the user: One to get all entries for the main table (minus the current user), one to get just the current user's entry to display in their own table, and one to get just the current user's parts to display in their table. There is also a controller for each of the buttons above the user's own information table. The first is to add or edit their user information, which simply creates a form and manipulates the user info database. Next are the 'Add Have' and 'Add Seeking' buttons, which open a form for the user to add those parts. Finally, we have a controller to remove a specified part, which is called when the user clicks the 'X' next to one of the parts they own.

There are also multiple controllers for the Offers and Reservations page and the reservation system. The first is one to swap reservation status for a specified part. This happens when a user clicks the reserve button on a part, at which point it will swap the 'reserved_by' field for that part to the user's username. If they are unreserving a part, it does the opposite and sets the field to 'none' to show that no one has reserved this part. There is also a controller to grab the offers and reservations for the current user, and a few to match the 3 possible buttons on the page: Complete an offer (which removes the part from the user's 'Have' section), decline a trade (which sets the reservation status for that part to 'none'), and unreserve trade (which sets the reservation status for that part to 'none').

Within index.js there are a few methods that correspond to this page as well. Most of them are simply calls to the controllers so that they can grab and display the information, but there are a few extra ones. The first is the search method, which takes the typed query and returns true if the current part matches that query. This is then used to determine whether or not to display that part. There is also, of course, the clear_search() method that clears the search bar simply by setting the model to an empty string.

Frontend (trading.html, offers_reservations.html):
All of the index.js methods are used here to create the page, everything is pretty standard. A field for the search bar, and a table for all the user entries. We use v-if conditions to decide which buttons we should show next to each 'have' part, so only one shows up per part depending on the conditions. We also actually use another table to display the user's own information, even though there is only ever going to be one entry. This was to keep everything similar so we could use the same techniques for grabbing and displaying information throughout the app.

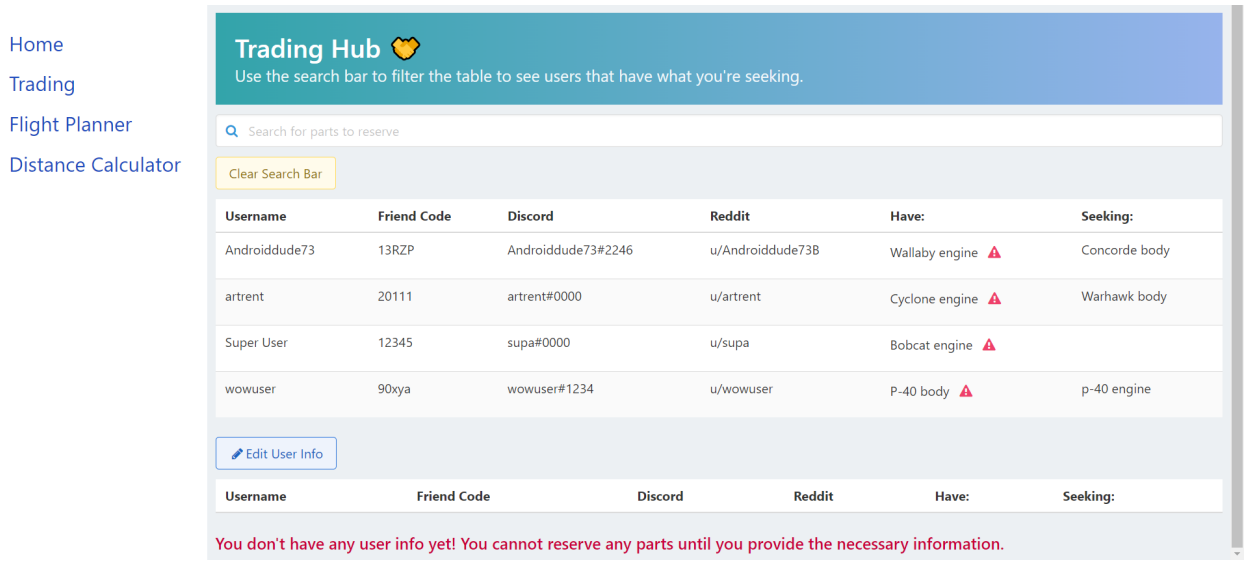
The offers and reservations html page is similar, but different in the fact that we simply use columns and divs to display the needed information. We originally had the two aspects (offers and reservations) stacked on top of eachother, but decided that having them side by side would be better for user experience.

*User Stories:*

- As a user, I want to display which parts I want and which parts I am willing to give to other users.
- As a user, I want to be able to see which parts other users are offering.
- As a user, I'd like to be able to search for specific parts that I want.
- As a user, I want to reserve the parts of other users so that they are not given to anyone else.
- As a user, I want to be able to see who has made a reservation for one of my parts, and how to contact them.
- As a user, I want to be able to see which parts I have reserved, so that I may easily follow up with the person I reserved them from, or rescind my reservation.

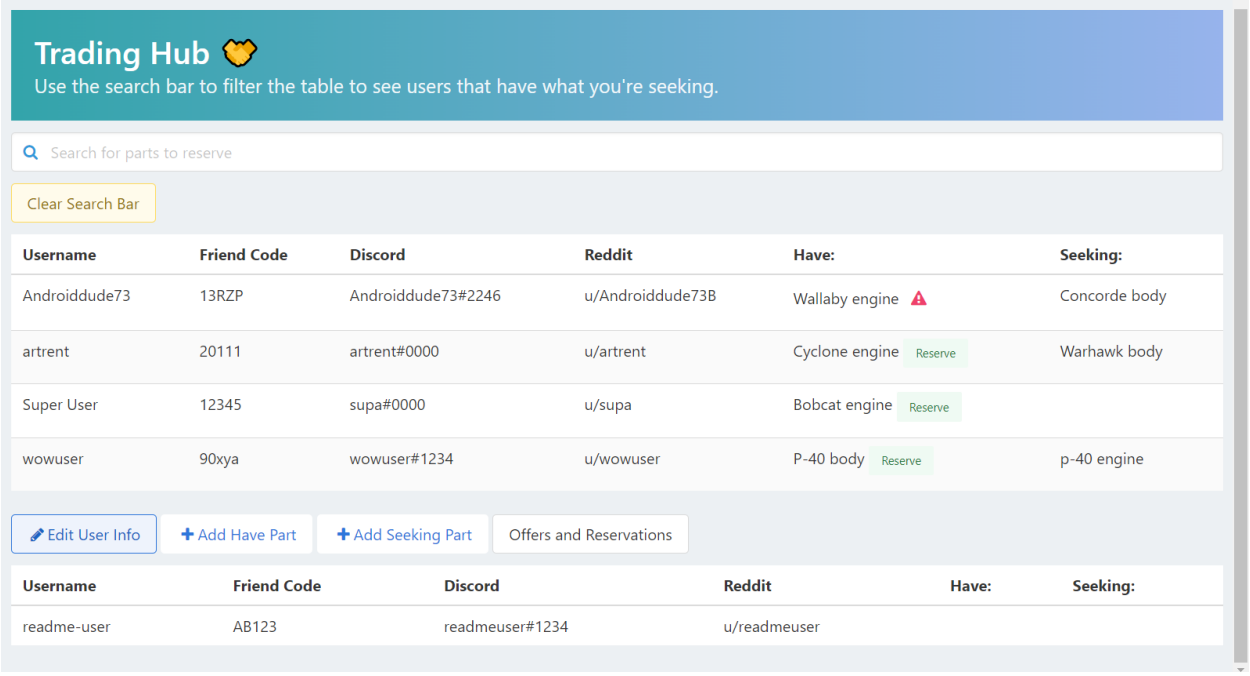
Screenshots of the process a user would generally follow on this page are as follows:

<u>Case 1:</u> The user has just created their account, and has no user information yet. They can see and search for other user's offers, but they cannot make any reservations until they provide their own information by clicking the 'Edit User Info' button.

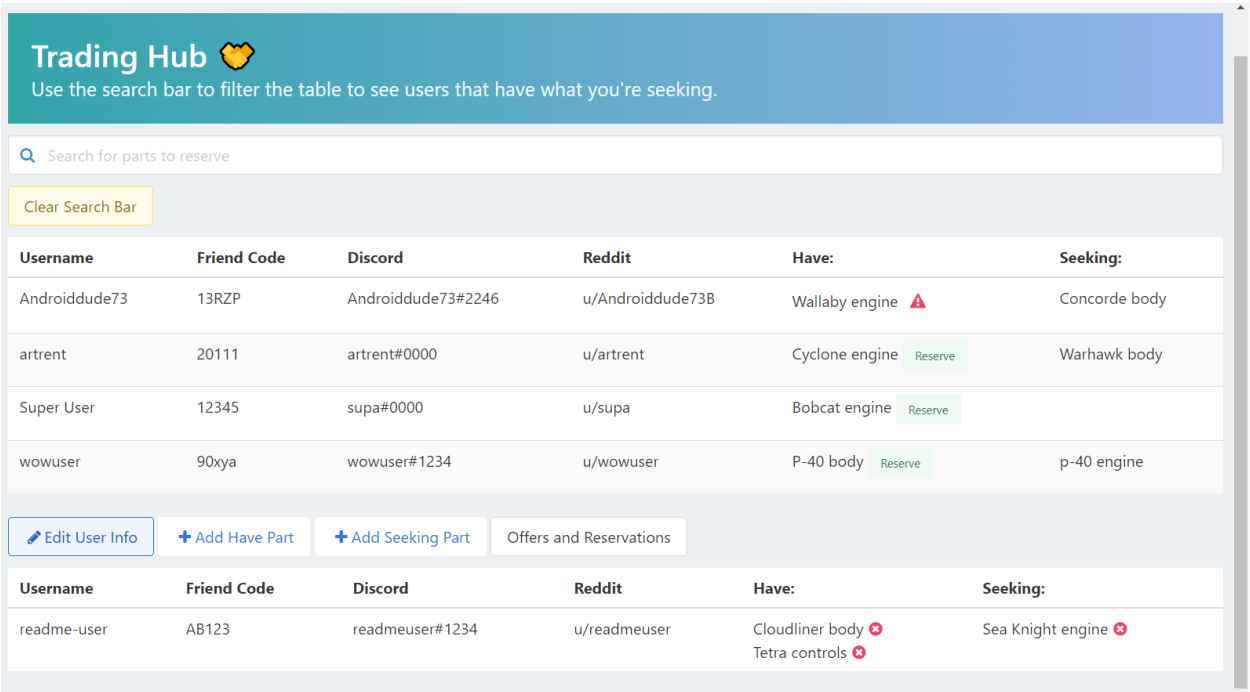



<u>Case 2:</u> After filling out the form given after clicking 'Edit User Info', the user now has their own information displayed at the bottom of the page. They can now see that most of the 'Have' parts displayed in the main table are available for reservation! The Wallaby engine

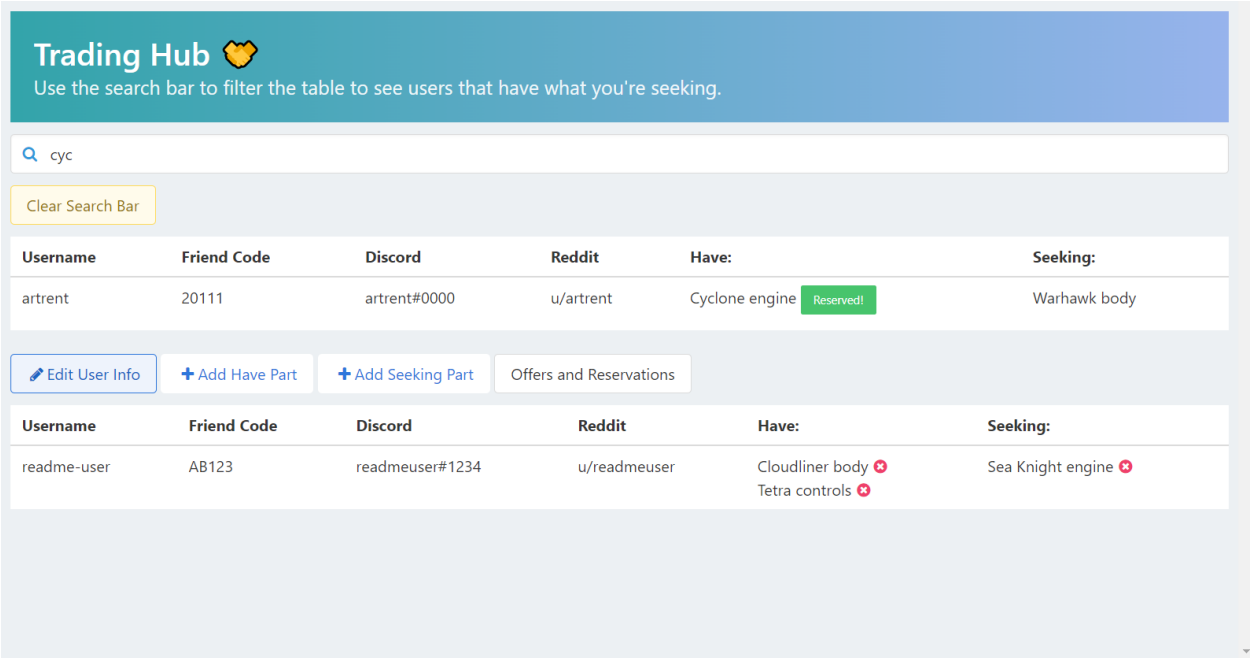
offered by Androiddude73 is reserved by another user, and thus maintains its caution symbol.



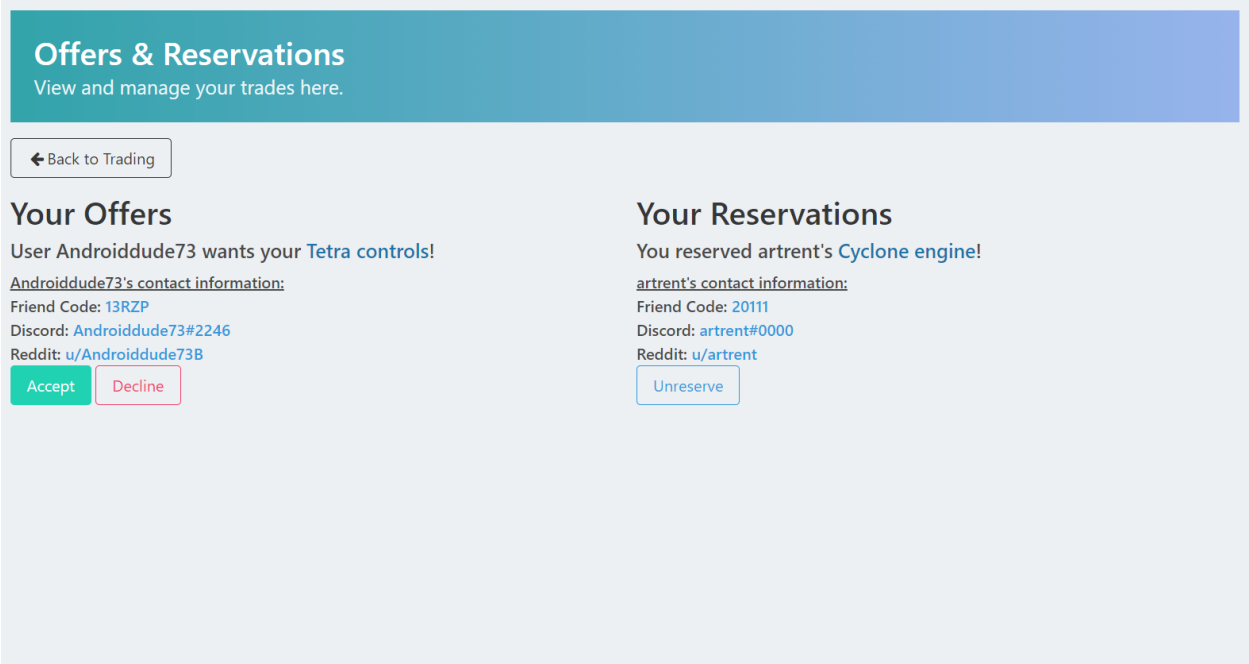

Case 3: After clicking the 'Add Have Part' and 'Add Seeking Part' buttons and filling out the respective forms, the user now has two parts they have, and a single part they're seeking. They can remove any of these parts at any time by clicking the neighboring 'X'



Case 4: The user uses the search bar to search for a part they want; in this case, they are looking for any part from a 'Cyclone'. This limits the table and only shows the users who are offering one of those parts, in this case, only one user (artrent) is. The search bar can be cleared by pressing the yellow button. Since the user would like to make an offer to artrent for their Cyclone engine, they click the reserve button. At this point, the user would want to use artrent's Discord or Reddit tag to make contact and talk about a fair deal.




Case 5: Now that the user has found their way around and made some reservations, they can check out their Offers and Reservations page by clicking the corresponding button above their personal information table. On the right, the user can see any reservations that they have made for other user's parts. As we can see, the Cyclone part they just reserved is there, along with the user's information if they didn't get it when they first reserved the part. If the user decides they no longer want to reserve this part, they can do so at any time by clicking the corresponding button. On the right, the user can see any of their own parts that have been reserved by other users. Looks like Androiddude73 wants the Tetra controls the user just added. They can make contact with Androiddude73, and either decide to make the trade or decide not to. Depending on that choice, they can either click 'Accept' to remove the part from their 'Have' section, or 'Decline' to remove the reservation status from the part.




## Flight Planner:

*Summary:*

The Flight Planner provides an intricate mechanism that enables users to formulate comprehensive journeys, deploying drop-down menus to sequentially add each destination until the final objective is reached. Each segment, encompassing "City A" to "City B", is denoted as a 'leg' of the overall journey. Upon selection of these locales, "City A" automatically assimilates the identity of "City B" as per the previous leg.

Following the accumulation of each leg within the table, users are afforded the opportunity to designate a plane from a dropdown menu, thereby revealing an analytical overview of the journey's statistics. Users wishing to alter their journey can utilize the 'Edit Leg' functionality located adjacently to each row within the table. Users are cautioned, however, to maintain the continuity of the journey by ensuring that the concluding city of each leg corresponds to the subsequent leg's starting city; failure to do so will prompt an error message until rectification. The platform provides users with the ability to eliminate either individual rows or the complete table, allowing for easy adjustments to the trip. Each leg in the table includes detailed specifications about the distance for that leg. Additionally, the platform presents an amalgamation of the journey's statistics including total distance, estimated flight time, projected profit, profit-per-hour rate, and the lengthiest leg of the journey.

For the purpose of illustration, the plane dropdown showcases the entire spectrum of in-game planes, circumventing limitations of class and range, thereby enabling users to explore a multitude of options.

*Data Organization:*
Backend (Controllers.py, models.py):
For the flight planner page, the py4web framework is used in order to handle server-side functionality. It utilizes one database table (db.flight_planner) which handles the 2 cities selected when entering part of a trip to the html table. It also contains the field planner_id which references the current logged in user to ensure that only data for the logged in user is being shown and altered, not for any other user. The flight planner has a total of 7 related actions in controller.py. These actions are used for displaying the flight_planner database, adding legs to the database, editing the database by using a form, deleting part or all of the table, calculating the distance between selected cities of each leg, and to get the list of all cities and planes for each dropdown.

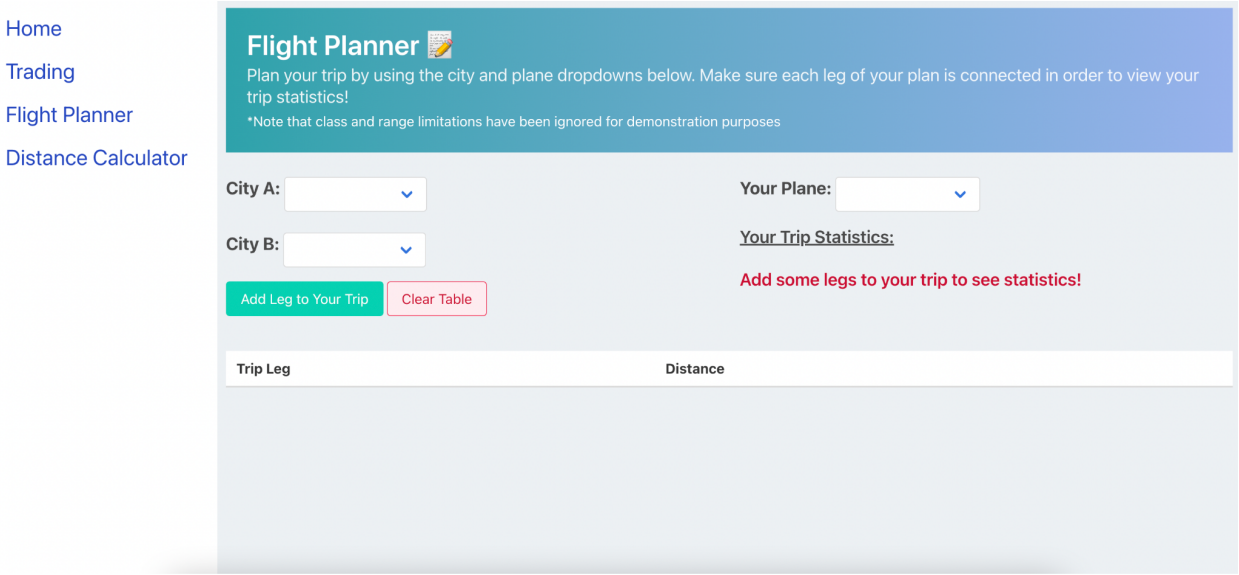
Data Files (cities.json and planes.json):
The 'cities.json' and 'planes.json' files store data about cities and planes, respectively. This data includes the coordinates and the class of each city and the range and class of each plane These files are read by the 'controllers.py' script at the startup and then served to the front end as needed.

Frontend (Index.js, planner.html, edit_plan.html):
In index.js, vue.js is used to handle all of the frontend logic. The 'get_flight_planner_entries' method uses axios to call the 'get_planner' endpoint and populate flight_planner_items as well as calling 'generate_trip_info'. The 'generate_trip_info' generates the flight statistics and also displays error messages if incorrect or insufficient data is entered. The 'build_url' and 'add_leg' methods assist in adding cities from the dropdowns into the table. The 'doFlight' method calculates all trip statistics (total distance, coins profited, flight time, profit per hour, and the longest leg) and is stored in variables to make formatting more simple when displaying in the html.
In the 'planner.html', Vue.js directives are used to render the dropdowns, buttons, and table displaying each leg of the trip and its distance. Selecting the plane dropdown triggers the 'generate_trip_info' method and is triggered again when a new plane is selected. The 'edit_plan.html' handles editing a single row of the table and updates the table once the submit is clicked.

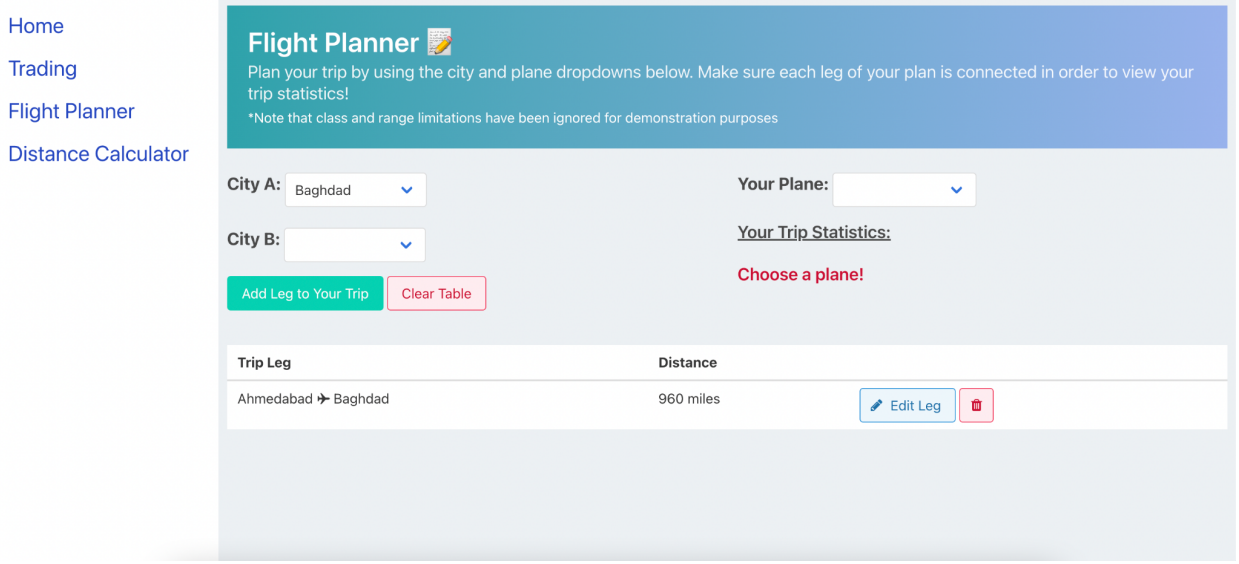
*User Stories:*
- As a user, I want to see my trip statistics from going from "City A" to "City B".
- As a user, I want to plan a long and complex trip and keep track of it by seeing each leg of my trip and how long they are.
- As a user, I want to be able to edit any leg of my trip in case I make a mistake.
- As a user, I want to delete a row of my plan table in case I make a mistake or no longer want that in my plan.
- As a user, I want to clear my entire plan easily so that I don't have to delete each individual row.
- As a user, I want to see how the statistics (total distance, flight time, flight profit, profit per hour, and longest leg) vary depending on what plane I select.

Screenshots of different cases a user can encounter are displayed below:

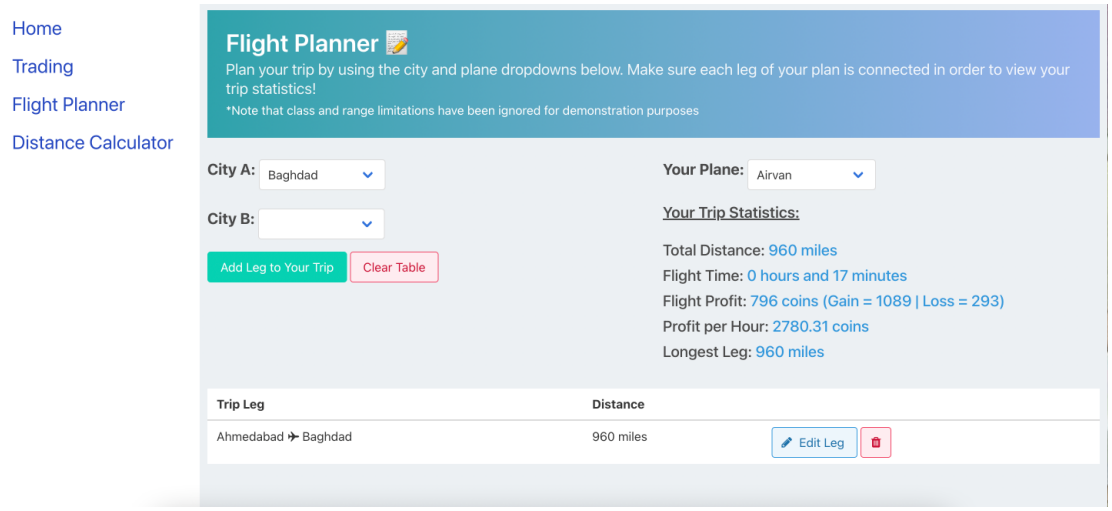
Case 1: No legs to trip added



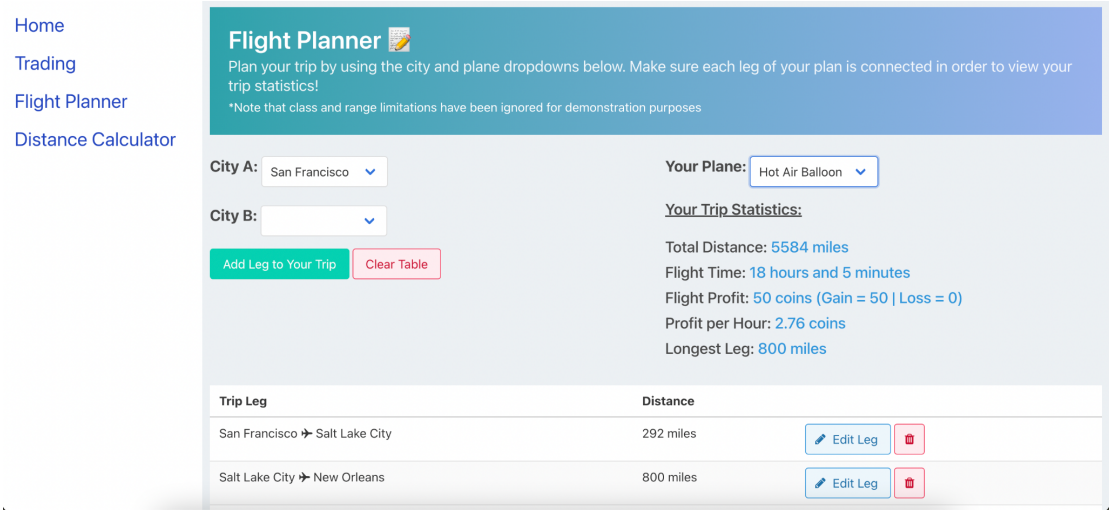

Case 2: One leg added, no plane selected

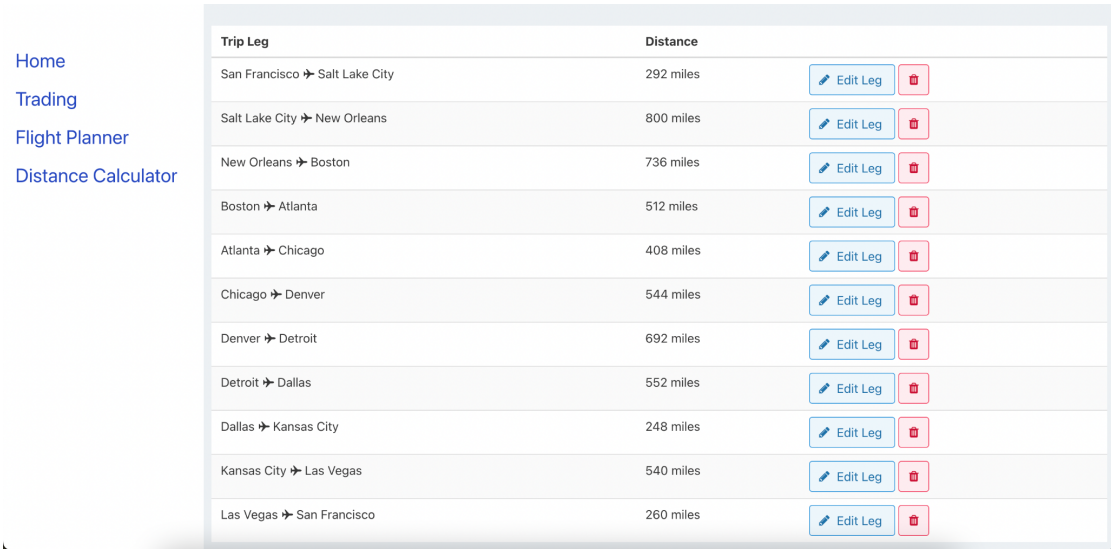


## Case 3: One leg added, plane selected, trip statistics are shown

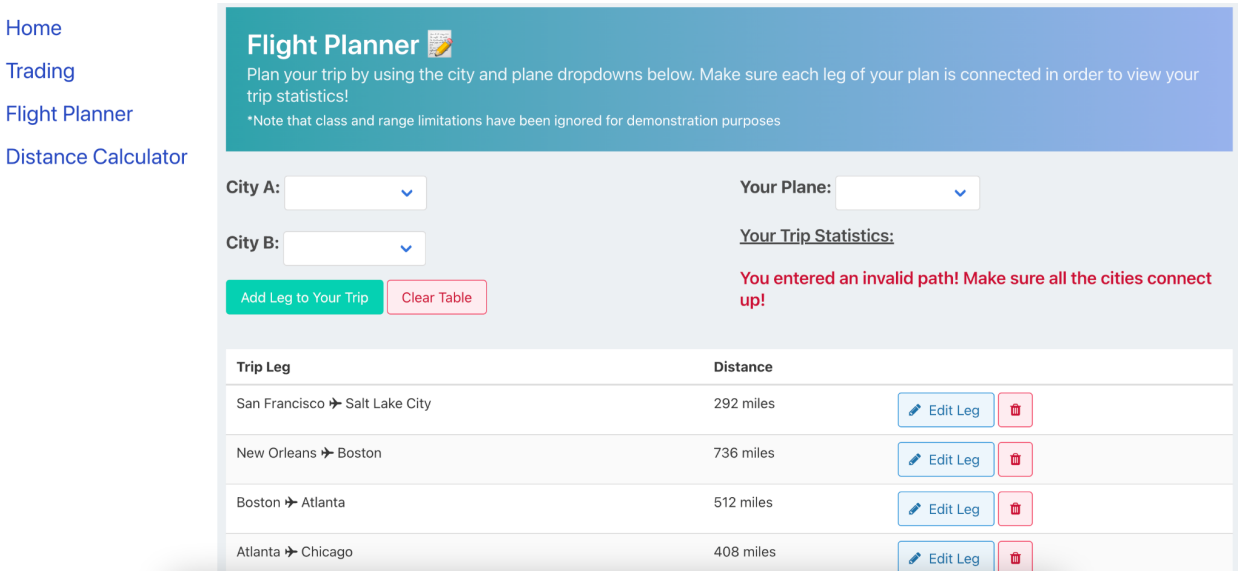



## Case 4: More complicated table, plane selected

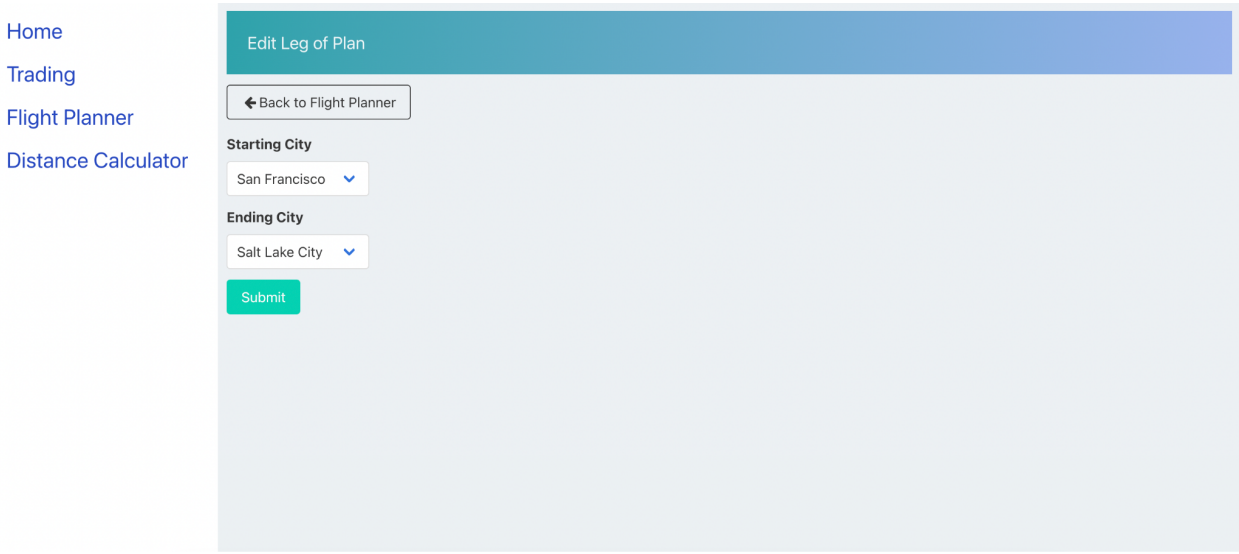


Case 5: deleting a row, cities in table no longer connect




Case 6: Editing leg of trip




## Distance Calculator:

*Summary:*

Our distance calculator is an essential tool for "Pocket Planes" enthusiasts, providing vital information on route limitations and viable planes for each specific city-to-city journey. With its intuitive dropdown menus, users can conveniently select any two cities worldwide to determine the exact distance between them. More than a mere distance measurement, this feature also offers critical data about the maximum class of planes that are permitted on that specific route. This strategic capability allows players to optimize their fleet and game performance.

In addition, the calculator presents an intelligently generated list of viable planes suitable for the selected route, tailored from the game's comprehensive assortment. This can greatly enhance a player's strategic planning, allowing them to make well-informed decisions about which aircraft to deploy based on the distance and class restrictions of their chosen routes.

*Data Organization:*

Backend (Controllers):

The backend of the project is managed using a Python script 'controllers.py' which handles all server-side functionality. It uses the 'py4web' framework. In the backend data, it reads the 'cities.json' and 'planes.json' files. It creates a list of all cities and two dictionaries for the full data on cities and planes, with the city and plane name as the keys. The backend then has two server actions: 'distance' and 'get_cities'. The 'distance' action serves the HTML file 'distance.html'. The 'get_cities' action returns the dictionary of cities and planes to the front end.

Data Files (cities.json and planes.json):

The 'cities.json' and 'planes.json' files store data about cities and planes, respectively. This data includes the coordinates and the class of each city and the range and class of each plane These files are read by the 'controllers.py' script at the startup and then served to the front end as needed.

Frontend (Index.js and distance.html):

The frontend of the project is split into a JavaScript file ('index.js') and an HTML file ('distance.html'). In 'index.js', a Vue.js instance is created to handle all the frontend logic. The Vue instance stores data and methods for handling user interactions. The 'generate_city_list' method uses axios to call the 'get_cities' endpoint and populate the 'city_list' and 'plane_list' arrays. It also populates 'allCityInfo' and 'allPlaneInfo' with the full data for each city and plane. The 'distanceCalc' method calculates the distance between two selected cities and generates a list of viable planes based on their range and class. This method also handles an error case where the two selected cities are the same. The Vue instance is mounted to the '#vue-target' HTML element. In the 'distance.html', Vue.js directives are used to render the city list and plane list to select options, to display calculated results, and to trigger the 'distanceCalc' method when the calculate button is clicked.

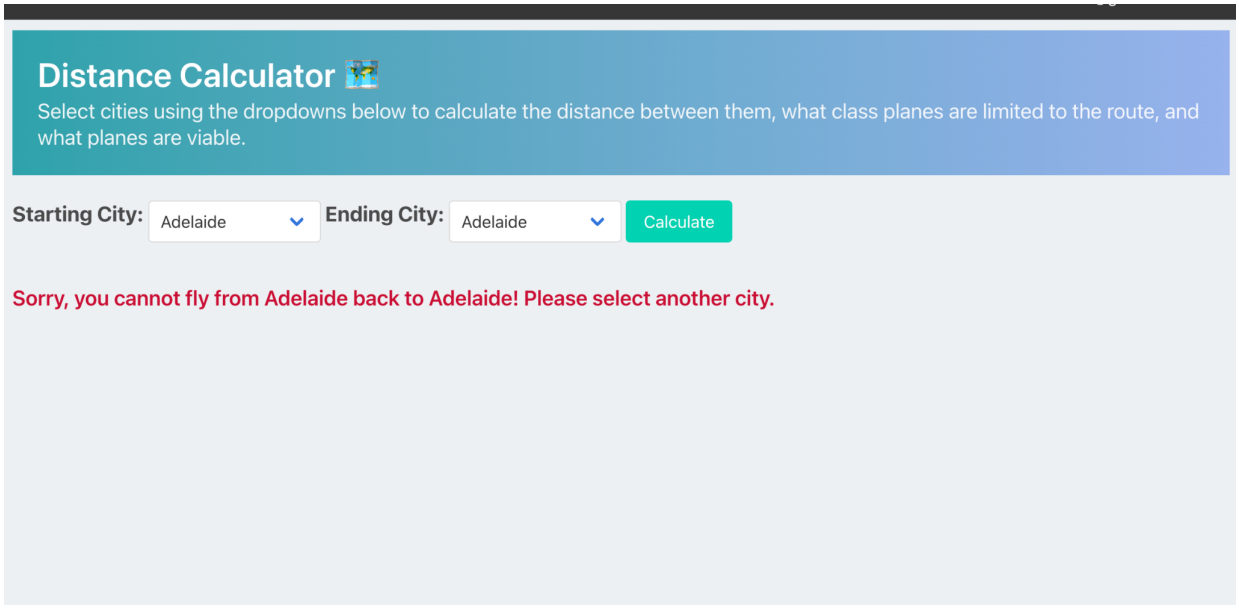
*User Stories:*

- As a user, I want to select two cities from a list, so that I can calculate the distance between them.
- As a user, I want to view the distance between two selected cities, so that I can understand the distance for my travel planning.
- As a user, I want to see viable planes for my selected route, so that I can choose the most suitable aircraft for the journey.
- As a user, I want to see class restrictions for my selected route, so that I can ensure my chosen aircraft is suitable.

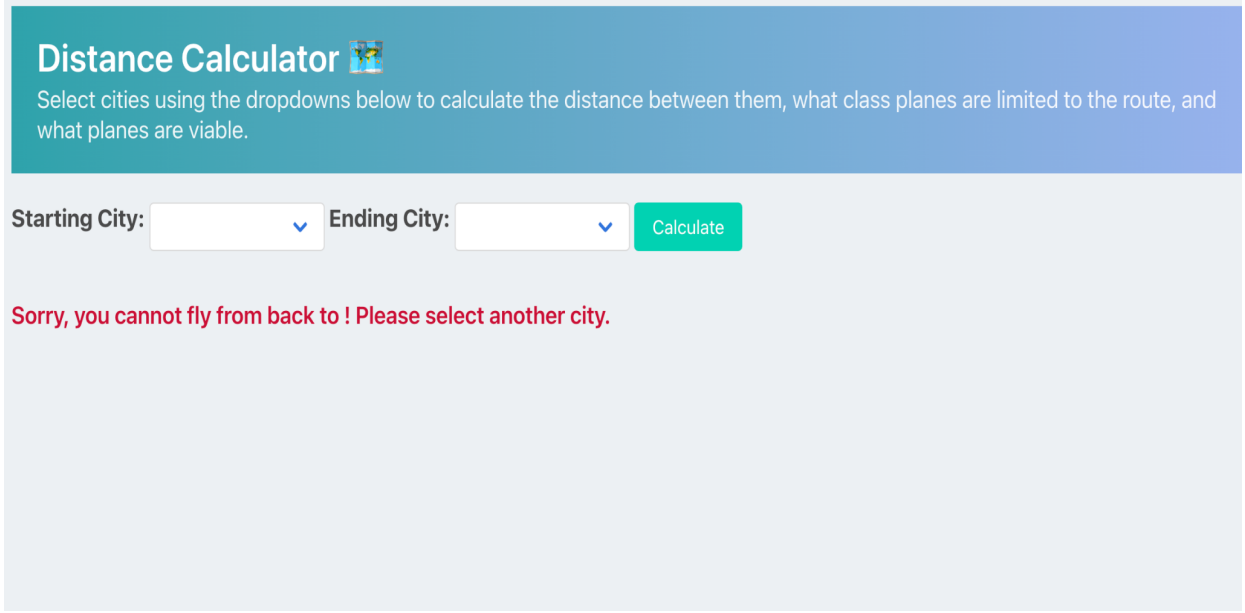
- As a user, I want to view the total travel distance for multiple selected routes, so I can plan a complex journey.
- As a user, I want to calculate and view the total flight time for my selected journey, so I can schedule my time effectively.
- As a user, I want to view the fuel consumption for my journey, so I can plan for the necessary fuel costs.

Screenshots of some of the different types of restrictions a user can encounter based on distance:

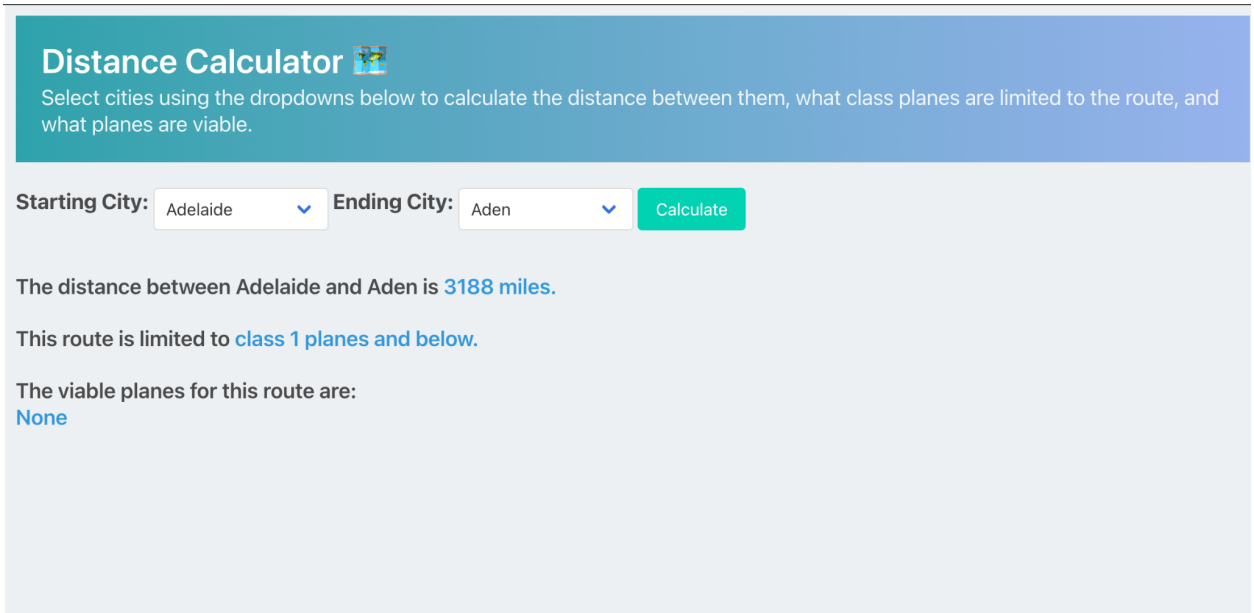
Case 1: Traveling to the same city



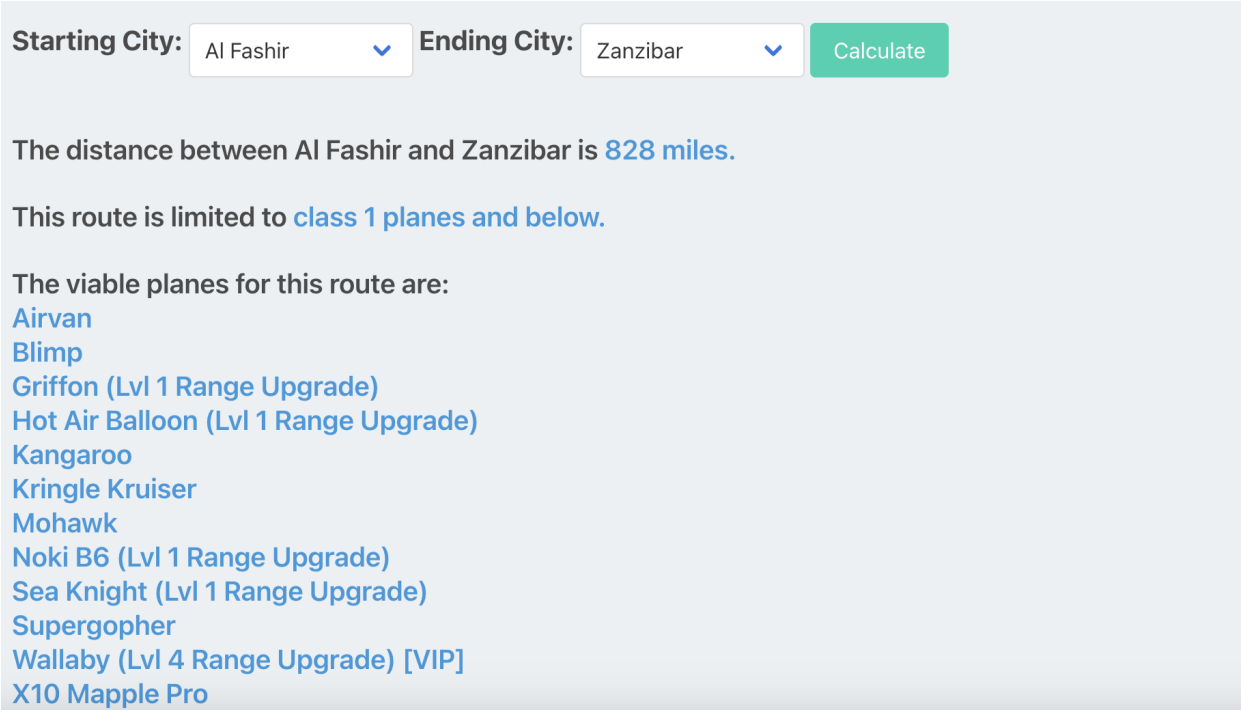

Case 2: Not listing starting city and/or ending city



Case 3: No viable planes for given route




Case 4: Many viable planes for given route



## Original Implementation Plan:

*Yukti:* Mostly working on the visuals and making things look nice!
*Evelyn:* Working primarily on the trading system and the edit form(s) for the trading system
*Andrew:* Mostly working on the backend for the flight planner and the distance calculator.

*First two weeks:* Making sure all of our pages are in place, with the top and sidebars working. Setting up py4web.

*Second two weeks:* Focus mostly on getting the trading system done, as it will probably be the most complex

*Third two weeks:* Focus on finishing up the last two pages

Making sure everything is pretty :)