

AIVFR



AI ASSISTED FLIGHT PLANNING

AN AI-ASSISTED FLIGHT PLANNING WEBTOOL FOR UK PILOTS.

v1.0.0-beta

Development Documentation

As written for the 2026 OCR A-level Computer Science
Non-Exam Assessment. For Reference only.

A passion project By
@TheEasyPilot

NOTE: Any diagrams or images related to flight planning or aircraft used in this document are for REFERENCE ONLY. Nothing in this document is suitable for real-world flight planning.



CONTENTS

ANALYSIS.....	8
Problem Identification.....	8
Why is it suitable for a computational approach?.....	8
Abstraction.....	9
Decomposition.....	9
Algorithmic Thinking.....	10
The use of APIs.....	10
Stakeholders.....	11
Who could qualify as a stakeholder?.....	11
Stakeholder interviews/questionnaires.....	11
Responses.....	13
Analysis of responses.....	15
Existing solutions.....	17
1 - SkyDemon.....	17
2 - AeroWeather.....	19
3 - METAR-TAF.com.....	20
4 - E6BX.....	22
5 - Notion AI.....	24
Essential features.....	25
Limitations.....	27
The use of Artificial Intelligence.....	27
Technical limitations.....	27
Time constraints.....	27
Resources and experience.....	28
Hardware/Software Requirements.....	28
Success Criteria.....	29
DESIGN.....	33
Breakdown of the problem - Hierarchy Charts.....	33
Main breakdown.....	33
Main Menu.....	34
Dashboard.....	34
Planning Module 1 (Navigation & Routing).....	35
Planning Module 2 (Flight performance, weights and fuel).....	35
Justification of the process.....	35
Decomposition.....	35
Chart features & Design.....	36

Detailed breakdown.....	37
Main Menu.....	37
Dashboard.....	37
Planning Module 1 (Navigation & Routing).....	39
Planning Module 2 (Flight performance, weights and fuel).....	41
User Interface.....	42
Colours.....	42
Screen designs.....	44
Title Screen (Main Menu).....	44
Disclaimer Popup.....	45
Settings Menu (Nothing selected).....	46
Settings Menu (Item selected).....	46
Dashboard - light mode.....	47
Dashboard - dark mode.....	48
Selecting another tab.....	48
Route tab.....	49
Weather tab.....	50
Navigation log tab.....	51
Fuel tab.....	51
Mass & Balance tab.....	52
Performance tab.....	52
Usability features.....	53
Learnability.....	53
Error Tolerance.....	54
Feedback.....	54
Consistency.....	55
Algorithms.....	55
Algorithm 1: CheckSaved().....	55
Algorithm 2: verifyICAO().....	56
Algorithm 3: fetchAirportName().....	56
Algorithm 4: FetchMetar/TAF().....	57
Algorithm 5: updateJSON().....	58
Algorithm 6: fetchRole().....	58
Algorithm 7: prompt().....	59
The Issue with fetching NOTAM reports.....	59
Key Variables & Data structures.....	60
Justification of how the algorithms form a complete solution.....	63
Application Walkthrough.....	63
Testing.....	64
Iterative test data.....	64
Post development test data.....	66

DEVELOPMENT.....	68
Section 1: Initial Flask Setup.....	68
Aims of this section.....	68
Iteration 1 - Setting up and testing the basic Flask template.....	69
Iteration 2 - Creating and testing all the pages.....	70
Section Review.....	72
MENU AND CONFIGURATION MODULE.....	72
Section 2: Main Menu.....	72
Aims of this section.....	72
Iteration 1 - Fetching Font and displaying main attributes.....	73
Iteration 2 - Styling background and logo.....	75
Iteration 3 - Styling logo completion.....	79
Iteration 4 - Settings button styling.....	81
Iteration 5 - Continue/load/save/settings buttons styling.....	82
Iteration 6 - Main Menu styling completion.....	86
Iteration 7 - The Disclaimer.....	88
Iteration 8 - Disclaimer popup completion.....	90
Iteration 9 - Scripting the buttons: Main Menu -> Disclaimer.....	93
Iteration 9.5 - Fixing TemplateNotFound Error.....	95
Iteration 10 - Scripting the buttons: Main Menu -> Disclaimer.....	95
Iteration 11 - Scripting the buttons: Settings button, Continue button.....	98
Iteration 11.5 - Fixing the 404 Error.....	101
Section Review.....	103
Section 3: Settings Menu.....	104
Aims of this section.....	104
Iteration 1 - Header.....	104
Iteration 2 - Main content: Elements dump and organisation.....	107
Iteration 3 - Settings Menu Styling.....	109
Iteration 4 - The light/dark toggle.....	111
Iteration 4.5 - Fixing the .svg color issue.....	113
Iteration 5 - Choosing units: Hiding/Showing options.....	114
Iteration 5.5 - Adding a toggle for the options display.....	116
Iteration 6 - Choosing units: Closing other options and adding colours.....	117
Iteration 7 - Back button.....	118
Section Review.....	119
Section 4: The Base Template.....	120
Aims of this section.....	120
Iteration 1 - base.html setup and inheritance test.....	122
Iteration 2 - Base template header.....	125
Iteration 3 - The Time and home.....	128

Iteration 4 - Navigation bar.....	130
Iteration 5 - Navbar: Buttons styling.....	134
Iteration 6: Selecting page overlay.....	136
Iteration 7 - Gradient and toggling.....	138
Iteration 8 - Selection overlay completion.....	142
Iteration 9 - Coding buttons, page creation and button coloring.....	143
Iteration 10 - Linking pages, coding buttons and color styling.....	145
Section Review.....	147
Section 5: The flight data and the Session.....	147
Aims of this section.....	147
Iteration 1 - Initialising flight data and the session.....	148
Iteration 2 - Updating data.....	151
Iteration 3 - Light/Dark mode integration.....	153
Iteration 3.5 - Fixing unstyled content flash.....	155
Iteration 4 - Settings Units Selection.....	156
Iteration 5 - Main menu New Flight button.....	159
Iteration 6 - Inputting, reading and outputting text data.....	162
Iteration 7 - Outputting text data from the session.....	164
Iteration 8 - Changing the units of an input number value.....	166
Iteration 9 - Refining the update units algorithm.....	169
Iteration 10 - (Actually) Finalising the Units conversion algorithm.....	171
Iteration 11 - Saving and loading a flight plan: Save button.....	173
Iteration 12 - Saving the flight data to the user's device.....	177
Iteration 13 - Load Flight Popup.....	179
Iteration 14 - Replacing the session with the uploaded session.....	183
Section Review.....	187
THE PLANNING MODULE.....	188
Working Backwards.....	188
Section 6: Route Tab.....	188
Aims of this section.....	188
Iteration 1 - HTML initial layout and styling.....	189
Iteration 2 - Route setup: Adding and positioning elements.....	193
Iteration 3 - Route setup: Styling the elements.....	195
Iteration 4 - Route setup: Validating input data.....	198
Iteration 5 - Route setup: Searching for the airport inputted.....	201
Removal of 'Via' route data.....	205
Iteration 6 - Route map: Initialising the map.....	205
Iteration 7 - Route map: Adding the OpenAIP layer.....	206
Iteration 8 - Route map: Adding a satellite view.....	209
Iteration 9 - Styling ICAO route and route generation sections.....	211
Iteration 10 - Route Plan: Drawing on the map.....	215

Iteration 10.5 - Eliminating the TypeError.....	217
Iteration 11 - Route Plan: Drawing the Departure-Arrival line.....	219
Iteration 11.5 - Eliminating the IndexError in the API fetch.....	223
Iteration 12 - Route Plan: Adding cities and aviation features to the route.....	225
Iteration 13 - Route Plan: Crafting an entire route with indexing.....	232
Iteration 14 - Route generator: Initial Test.....	238
Iteration 15 - Route generator: Prompt and output making.....	241
Iteration 15.5 - Fixing AI output errors.....	246
Iteration 16 - Route generator: Updating the route using the response.....	249
Iteration 17 - Tooltip Box.....	251
Section Review.....	253
Section 7: Weather Tab.....	254
Aims of this section.....	254
Iteration 1 - HTML initial layout and styling.....	254
Iteration 2 - Aerodrome Search: Searching for an aerodrome.....	257
Iteration 3 - Aerodrome Search: Fetching and outputting reports.....	261
Iteration 4 - Aerodrome Search: Parsing the METAR data.....	266
Iteration 4.5 - Fixing METAR formatting error.....	270
Iteration 5 - Aerodrome Search: Parsing the TAF data.....	271
Iteration 6 - Departure & Arrival Reports.....	275
Iteration 6.5 - Fixing minor formatting issues.....	279
Iteration 7 - Validation and styling for feedback and missing inputs.....	280
Section Review.....	283
Section 8: Navlog Tab.....	284
Aims of this section.....	284
Iteration 1 - Calculating Route distance.....	284
Iteration 1.5 - Fixing TypeError in distance calculation.....	287
Iteration 2 - Navlog header elements and styling.....	288
Iteration 3 - Navigation log table styling.....	292
Iteration 4 - JS: Variation, adding rows dynamically.....	295
Iteration 4.5 - Fixing positioning errors in the navlog.....	299
Iteration 5 - Updating data in the table.....	301
Iteration 6 - Performing Calculations.....	306
Iteration 7 - The Alternate Aerodrome.....	310
Removal of AI autofill.....	314
Iteration 8 - Live input refreshing on navlog.....	314
Iteration 9 - Downloading the navlog as PDF.....	316
Iteration 10 - Adding details to the PDF.....	318
Section Review.....	321
Section 9: Fuel Tab.....	321
Aims of this section.....	321

Iteration 1 - HTML initial layout and styling.....	322
Iteration 2 - Adding fuel policy table.....	326
Iteration 3 - Adding keys for values & Units.....	330
Iteration 4 - Input handling and Calculations.....	333
Iteration 4.5 - Fixing unit conversion inaccuracies.....	337
Iteration 5 - Fuel policy calculations & stopovers.....	339
Iteration 6 - Fixing unit conflicts.....	342
Iteration 7 - AI Autofill.....	346
Section Review.....	348
Section 10: Animations.....	349
Iteration 1 - Animations.....	349
Section 11: Mass & Balance Tab.....	351
Aims of this section.....	351
Iteration 1 - HTML initial layout and styling.....	351
Iteration 2 - Table HTML and styling.....	354
Iteration 3 - User Input handling.....	359
Iteration 3.5 - Fixing session saving error.....	363
Iteration 4 - Calculating moments and CG.....	364
Iteration 5 - Unit conversion fixing.....	367
Iteration 6 - Pre-inputted values & symmetric arm autofill.....	369
Iteration 7 - Manoeuvring speeds and table highlighting.....	374
Iteration 8 - AI change suggestion.....	377
Section Review.....	381
Section 12: Performance Tab.....	381
Aims of this section.....	381
Iteration 1 - Performance Tab.....	381
Section Review.....	385
Adding a Changelog.....	386
Section 13: Dashboard.....	389
Aims of this section.....	389
Iteration 1 - HTML initial layout and styling.....	389
Iteration 2 - Finishing page design and styling.....	392
Iteration 3 - Filling data not requiring JS.....	397
Iteration 4 - JS required values: POB, Cargo, Cruise, Average GS.....	400
Iteration 5 - Current weather reports.....	403
Iteration 6 - Adding Expenses.....	406
Iteration 7 - Removing expenses.....	411
Iteration 8 - AI Brief.....	414
Switching from Session to a Database.....	416
Iteration 9 - Downloading the flight plan as a PDF.....	419
Section Review.....	425

EVALUATION.....	426
Post Development testing.....	426
Structure of the tests.....	426
Functionality test: Tatenhill (EGBM) to Cambridge (EGSC) [DEMO].....	427
Usability Test: North Weald (EGSX) to London Southend (EGMC).....	428
Evaluation of Usability Features.....	428
Learnability.....	428
Error Tolerance.....	430
Feedback.....	431
Consistency.....	431
Robustness tests: North Weald (EGSX) Local flight.....	432
Robustness Test 1.....	432
Robustness Test 2 (Manual).....	432
Comments on Robustness.....	434
Evaluation of Success Criteria.....	435
Addressing Unmet Features in Further Development.....	437
Unmet Success Criteria.....	437
Unmet Usability Features.....	438
Evaluation of Limitations.....	438
Inability to add a waypoint from any index/point in the route.....	438
Entire route must be reset to change departure/arrival airports.....	439
AI-related limitations.....	439
Slowness with calculations.....	440
Split text in the brief pdf.....	440
How were initial limitations handled?.....	441
Maintenance issues.....	442
Deployment issues.....	442
Server maintenance.....	442
API reliability and cost.....	442
User feedback.....	442
RESOURCES & CREDITS.....	444
Full Code.....	444
Testing Videos.....	444
APIs.....	444
Mapping.....	445
Useful Tools.....	445
libraries.....	445
Tutorials and aids.....	445
AI use within development.....	446
Disclaimer (As seen in webapp).....	447

ANALYSIS

Problem Identification

As a licensed [UK Private Pilot](#), I have real-time experience of what it is like to plan a flight. Regarding calculations such as planned airspeed for the flight, fuel calculations and so on, traditional navigation methods involved using analogue instruments such as the [flight computer](#), which, although reasonably accurate and produced reliable results, was more prone to human error and can also consume a lot of time as you brief your flight. Obviously, in the modern world we live in now, better solutions have come up, such as electronic flight computers, which have rightfully become the norm.

Things get more challenging when we start to consider the route we plan to fly. For a cross-country flight to be executed flawlessly, many considerations must be made, namely the weather, the route, the necessity to enter controlled airspace, obstructions along the route, alternate aerodromes in the case of an emergency diversion- the list (really does) go on. Sometimes it's easy, let's say, if the route is straight forward, has been flown several times before, and (most importantly) there aren't many complex procedures involved at the arrival aerodrome- but this is almost never the case. Things change, even across well-known routes that could fly past the head of an experienced pilot and could lead to increased workload in the aircraft. [My solution aims to mitigate this and have the result of making flight planning a lot simpler for a pilot, as extensive thought doesn't have to go into considering every detail at every point along the route.](#)

The purpose of this project isn't simply to make flight planning something like an 'automated' process. This is very (very) dangerous as at no point should a decision in a flight plan be made on behalf of the actual pilot in command, as this could lead to serious consequences in flight if said decision was not practical but was disregarded by the pilot. Annex 2 of the International Civil Aviation Organisation (ICAO) stresses: "The pilot-in-command of an aircraft shall, whether manipulating the controls or not, be responsible for the operation of the aircraft in accordance with the rules of the air..." This indeed includes flight planning as without one the other doesn't happen. [So, to cooperate with this, the purpose of AI in this solution is to offer different suggestions and options that best suit the chosen flight plan and identify any potential threats along the route and again suggest workarounds to mitigate them. I will therefore stress that **the decision to follow any of these suggestions are at the sole responsibility of the pilot in command** and careful consideration of any risks involved is essential before doing so.](#)

In respect to this, I will ensure the suggestions produced by AI are as accurate and useful as possible to the best of my ability.

Why is it suitable for a computational approach?

The solution requires access to many features that can only be accessed via computational methods- in other words, the internet. Examples include from airport data, weather reports, airspace information and map data. [Given that in a real-life flight planning scenario access to a device is essential to be able to obtain such data, it will be no different with this solution. A solution that doesn't involve a computational approach in this case would simply be impossible.](#)

Furthermore, the solution intends to use Artificial Intelligence tools to assist in finding problems/suggestions to make to the user based on the aforementioned data, and to also present this in a friendly, conversational manner. This inevitably will require access to an internet connection to be able to run, and a computational approach is the only way to achieve this. Finally, the User Interface is also needed to provide a means for the user to interact with the website properly- the use of a computer will allow me to achieve this.

Abstraction

For standardisation and computer efficiency reasons, a lot of the data to be collected from different databases are all in a coded format. Abstraction will be required to present this data to the user in a better, more useful representation of reality. *Although the trained pilot will have studied how to decode such data, as it is in fact faster to read it in this way, having it represented in a pure, easily readable English format will avoid any errors when interpreting the data.* Below is an example (at the most complex level) of a weather report from Heathrow airport (code EGLL) in both its raw form and decoded forms.

RAW FORM

```
EGLL 221250Z AUTO VRB03KT 9999 BKN036 15/05 Q1019 NOSIG
```

DECODED FORM

```
Conditions at: 1250 UTC 22 Thu May 2025
Temperature: 15°C (59°F)
Dewpoint: 5°C (41°F) (RH = 51%)
Pressure (altimeter): 30.09 inHg (1019 hPa)
Winds: variable at 3 kt (1.5 m/s, 3.5 mph)
Visibility: 6+ mi (10+ km)
Clouds: broken clouds at 3,600 ft
Ceiling: 3,600 ft
Flight Category: VFR
```

Above: Raw and Decoded forms of a Meteorological Aerodrome Report (METAR) [Source: aviationweather.gov]

Other data like aerodrome forecasts, Met-Office weather data, Notices to Airmen (used to notify any temporary changes that could affect a flight) are all coded in some way, some more than others. Abstraction is therefore important here to ensure that these reports are presented in a comprehensive, user-friendly manner.

Decomposition

The form of decomposition to be used in this project will be **modular programming**, in which each subproblem is dealt with within self-contained modules, leaving other sections of code unaffected in the process. This will essentially allow me to code each of the project's functions, such as the AI Algorithms, the decoding algorithms, API commands and calculation functions as separate modules and at the end link them to each other to create a fully-fledged system. This is the most efficient way to do this as it avoids errors that could damage the entire code/program structure at once rather than just the section currently being worked on. It also means that, since each module is essentially its own program, I can perform testing on each of them individually and be confident that if this works independently it should also work once integrated with the main program. *This has the result of making my code more robust and less prone to unexpected bugs or errors that, if I*

hadn't incorporated modular programming, would be extremely difficult to fix without directly affecting some other part of the program at the same time.

On a very basic level, which shall be developed in the Design section of this document, the modules that make up this project will be something like that shown below:

FRONTEND

- Web module (HTML/CSS/JS)

BACKEND (Python)

- Decoder
- Prompter
- TEMAnalyser (Threat, Error and Management)
- FlightComputer
- FuelCalculator
- MassAndBalanceCalculator
- ExpenseCalculator

As you can see, having things laid out in this way allows for a much easier development process, and even makes the code easier to navigate for those accessing the source code for this program. Each module has no reliance on another to operate, which is key in ensuring that the development process remains as robust as possible. Furthermore, these modules act as checkpoints – important modules can be prioritised and those not completed won't affect the completed modules in any way.

Algorithmic Thinking

The use of algorithms in this program must be thought out extremely carefully to produce more accurate and reliable outputs to the user. This is especially important in my case given that my program plans to provide pilots with real-world information that is supposed to help in real world flight. Whilst in some modules, must-be-accurate data is essential (calculations for example), others may not need this and will be more focused on outputting suggestions based on analysis of data chosen only by the user. This will define which algorithms can and cannot incorporate things like Large Language Model APIs for advanced output, or those that must rely solely on hard-coded pre-written calculation functions. The way I intend to differentiate between these algorithms will be explained in further detail in the design section of this document. Without the proper incorporation of algorithmic thinking in my solution, I run the risk of giving incorrect data for the pilot to use in the real world, which could result in several problems despite a disclaimer being present.

The use of APIs

When considering the suitability of this solution being computational, APIs are an essential part of this program as there are countless amounts of data that are updated constantly throughout the day (weather reports are the most obvious example in this case) and the only way to ensure such data is accurate at the time of using the webtool is to use APIs available to fetch data and display it to the user. Luckily, a lot of aviation-related APIs are free since this data has always been made free to obtain from government entities and so this is respected when third-party companies provide reports using data accessed from such sites. If I choose not to use any APIs, then no reports necessary for flight planning can be obtained for the user and they will have to search for them elsewhere, which is not ideal and could potentially be time consuming.

Stakeholders

Who could qualify as a stakeholder?

A Licensed pilot

This is the more obvious stakeholder as it is the main target audience for this webapp. A pilot who has held at least a PPL will know what is required to plan a flight to the legal and suitable standards, and would be able to interact with and utilise the webtool to the best of its ability, and this software should be of aid to them the most. Whilst this may be obvious, what is not so obvious is how useful this tool could be from an educational perspective.

A Pilot in training

A pilot in training also makes for an excellent stakeholder, as, looking back on my own training, there was a lot of theory involved in learning how to plan a flight correctly, but hardly ever were there opportunities to actually plan one from start to finish, considering everything necessary, until the solo land-aways and the single qualifying cross-country flight you would do as part of your practical training. After that, you are on your own, and without the pressure and aid from flight instructors, it's much easier to miss something out. However, if tools like this were used during training to *inform* the decisions made by the student pilot, this could teach them valuable skills such as what type of weather to look out for, specific things to look out for in route planning, or, and most importantly, risk assessment (which the aviation industry will refer to as Threat and Error Management), the lack of which is undoubtedly a common cause of aviation accidents worldwide.

A Layperson

Having someone who has no experience in the field at all would also serve as a good stakeholder as they could use the tool to educate themselves on how a flight might be planned. It would be a brilliant learning resource, as there is no risk attached and they can play around with any features they would like and also use AI tools like the brief and route generators to see how a pilot might think when planning a route to be as safe and as efficient as possible, all without being stuck behind a paywall, which is common in most sophisticated flight planning software out there, as will be seen in a few sections.

AI Researchers and Data Scientists

Since the aim of this project is an experiment on how AI can be applied in General Aviation flight planning practices, the data generated in this project could potentially be used by these people to train their models to be better suited for GA flight planning, which opens the doors for more reliable AI integration into the currently available flight planning software, which brings new innovative possibilities to flight planning that we may have never seen, or thought of, before. Even better, this data could be pushed to machine learning engineers, who could go as far as developing AI models tailored specifically to aviation flight planning, providing it with access to large aviation databases etc. This could then be used by tools like mine and official software to provide what could be state-of-the-art software that could completely change the way flights are planned in the future; perhaps, to the point where it becomes fully automated! Who knows!

Stakeholder interviews/questionnaires

Of course, this will only have a positive impact on student pilots and qualified pilots if it actually solves their specific problems they may have during flight planning, and I will identify these problems through a questionnaire. Unfortunately, none of my fellow school students are qualified or student pilots, and a good grasp of aviation and general navigation theory would be required to not only understand the questions I ask but to also identify problems affecting the general flight planning experience. Luckily, I am proudly part of a community (shout out to *Young Sky Pioneers!*) of hundreds of aviation enthusiasts

and professionals at all different levels, some of which I have met in person because of this community. I will ask members of the community to fill in a form containing a few questions that should help in tailoring what this solution should really do for users to feel satisfied.

Questions and Justification

Q1 – In general, how much effort would you say must be put in to plan a long-distance (100 NM+) cross-country flight to an unfamiliar area/destination?

This question allows me to see what unassisted flight planning looks like for a pilot. In foresight, it will most definitely differ with experience, but it allows me to generalise how people see flight planning and will inform how much assistance the AI provides for them.

Q2 – What tools do you use to make flight planning easier for you? How do they help?

There are a range of tools available that make flight planning a lot easier than the traditional methods, namely an app called ‘*SkyDemon*’ which is currently the most popular in the UK (I predict this will be mentioned a lot). This question allows me to see how many people use such tools and what about them makes them so convenient. I will then investigate these features and see how I can make something similar in this AI solution.

Q3 – If you are a License Holder, how would you compare your flight planning experience during training and now? Would you have preferred to have these tools in training?

This question is essential as it allows me to see what student pilots find difficult whilst in training with traditional flight planning methods. I will use this information to inform how the AI can still assist the students greatly whilst also giving them the ability to become proficient in these traditional methods (As these are, nowadays, supposed to act as a backup if modern methods fail, which they do)

Q4 – What is/are the most challenging aspect(s) of flight planning for you and why?

The obvious purpose of this question is to see what makes different people struggle when it comes to flight planning. In an ideal situation, the interviews should reveal a common problem that is faced among many pilots and thus the solution can put more focus on solving said problem. I also acknowledge that this might not be the case, and instead I will rank them based on how common they appeared/how much of a problem I personally thought it would be regarding flight planning.

Q5 – What would make these challenges easier to handle?

This acts as a follow up to Q4, and this time it allows me to see how *they* see the problem being fixed. This is important, because if a developer doesn’t understand what the ideal solution looks like from the user’s point of view, it is very easy for the developer to produce a solution that solves the problem in a way that is not entirely useful for the user.

Q6 – What is/are the simplest aspect(s) of flight planning for you and why?

This question allows me to see what I should focus less on finding an alternative to and tells me what I should have included in my solution as without these simple aspects flight planning would be a difficult task, according to the user.

Q7 – The outbreak of AI has seen AI-Assisted flight planning as useful in commercial aviation. Do you think modern tools like this have potential in General Aviation? If so, what would you like to see from it?

This is overall the most important question as it defines how the AI in this solution will act. By knowing how the user would like to see AI assist them in flight planning, I can tailor the algorithms/prompts used to make it as close as possible to how the user would like it, which ensures proper user satisfaction with my solution.

Form link:

https://docs.google.com/forms/d/e/1FAIpQLScaBfyLnMAnRS0uLnt_X3sIX_7QqmJeEEVjp_29DBZYk0TfKw/viewform?usp=header

Responses

I took 5 responses in total, *as more than this would increase the analysis burden and was also simply unnecessary*. There were some responses that did not provide useful data so were either removed completely or disregarded due to them not being relevant enough to the purpose of my solution.

Glossary

TFR – Temporary Flight Restriction

NOTAM – Notices To Airmen

Q1 – Effort needed for flight planning

- It takes a while so a lot of effort does have to go into it! Especially if you're landing at another airfield. You have to take into account weather, landing permission, types of airspace, time, so on. It's a fun process though! 9/10, 1 point knocked off for stress of it lol, when it all comes together for a good flight it's very satisfying
- About an hour of work including weather, winds aloft, Notams, airport charts and I always make a frequency list in a bright color so I can grab it quick. I'd rate it 7.5 out of ten
- My companies navigation department builds routes for us so it takes seconds. However if there is an issue enroute (TFR, NOTAM) etc it can take a little longer to edit.
- It takes some effort, there's a few resources to look at when planning and that does take some time to go through, also making sure ur aware of what airspace ur going through and what ur minimum and maximum altitudes are
- Easy

Q2 – Tools used to make flight planning easier

- For me I used a mix of online tools and paper-based, always good to have a backup incase either go wrong. I use a Pooleys CRP flight computer with a chart and SkyDemon.
- Sky demon and preflight
- ~~LIDO, NMP, CHMI, NOP portal~~ **[disregarded – relevancy]**
- SkyDemon, AIP, AeroWeather, pooleys plate, NOTAMs
- Sky demon to remove the wind work. CRP takes ages.

Q3 – Comparison of flight training aspects to now

- I would have loved to use sky demon. But still need the skills of chart work when the iPad dies.

- I used these tools during my training so not much different! I also had the guidance of an instructor which is always handy if you want security of another pair of more experienced eyes.
- Had them during training.

Q4 – Challenging aspects

- Getting the hang of using flight computers was easily the hardest part, after that, everything just sort of pieced itself together for me! Flight planning I would say is one of the hardest bits of flight training overall, so just the process of understanding and getting used to it was challenging.
- The winds and ground speed calculations. Fuel calculations are difficult
- Large danger areas, or times of mass disruption aka Spanish power cut
- Airspace - it's figuring out the best altitude to fly at without having to constantly go up and down

Q5 – What would make them easier to handle

- A more resilient ATC system across Europe and more up to date flight planning software. Old TFRs and NOTAMs also need to be removed from the system.
- Practice and sky demon.
- Again, having guidance from instructors and mentors helped a hell of a lot. One instructor ran a class purely to learn how to use the flight computers which was SO useful. With flight planning I think it's important to use your initiative, go home do some studying, use any resources you can.
- Not sure - don't think much can be done about this unless there's something that could help with figuring out best altitude to fly at for an entire journey

Q6 – Simplest aspects

- Actually flying the route I've planned lol! I think after that it would be planning the basic route, getting distances and times, before doing the harder bits of taking weather into account.
- It's not really simple to be honest.
- ~~Slot avoidance, if there is routes available round the slot area.~~ **[disregarded – relevancy]**
- Working out Headings to fly - it's repetitive
- Planning the route.

Q7a – AI assisted flight planning – place for it in general aviation?

40% of responses answered no. 60% Answered Yes.

Q7b – If so, what would you like to see from it? If not, why?

- **[YES]** More innovative route solutions and optimised Flight Planning that irons out inefficient routings and offers cost savings.
- **[NO*]** A better integration like being able to file an automatic flight plan would be great.

**(I believe this responder meant to say yes based on this response)*

- **[NO]** Not for me. I think flight planning is a really key bit of learning to be a pilot, and I think by using AI it could put people off making the effort to learn how to do it 'properly', which could potentially make it much harder if something goes wrong in the sky and you have to reroute for example!
- **[YES]** AI to help with diversions on updated conditions - best route to fly and airfield to land at AI to help analyse weather, NOTAMS, airspace restrictions and recommend safest and most efficient routes
- **[YES]** AI NOTAM integration.

Analysis of responses

Q1 – As predicted, there was a huge disparity in how people saw flight planning as a simple task. It seems like there is a routine procedure for it and some just prefer it more than others, but there are no significant stresses or difficulties that make the process require more effort than what's needed. Notably, it appears that the long duration of a flight planning session serves many things that must be considered such as weather, airspace, altitudes along the route, and such. Luckily, this serves good for my own requirements, as I want the AI to be responsible for analysing such things throughout a particular route and providing results for the pilot to use, which greatly reduces the workload and, as a by-product, reduces planning time, which solves another potential problem.

Q2 – It is clear that SkyDemon was among the most used solutions for making flight planning an easier process, and for a good reason. I will elaborate its use in the next subsection and what I would like to take from it. Other applications included AeroWeather, which is an app that provides airport weather data – but this is simple to implement as its primary purpose is to decode weather reports which is already in the plans for this solution too. 'Pooleys Plate' presents useful airport information and charts, but I will only be able to produce the former because charts can only be pulled from one reliable source which is constantly updating and could also have some legality issues that I obviously would not want to risk getting myself into. Pilots can still access these charts for free on an official website anyway, so it's definitely not worth trying to implement it here given the aforementioned risks.

Q3 – These results really surprised me, because I didn't have much access to modern navigation tools during training, whilst it seems it's the norm for others. Either way, the results show just how essential modern navigation tools are at the moment compared to traditional methods, although, as one of the responses mentioned, knowing how traditional methods of navigation work is essential in-case technology fails. Overall, nothing significant was gained from this question.

Q4 – Winds, ground speed, fuel, planning good altitudes that avoid certain airspace, and avoiding danger areas were the key takeaways from this question. Calculations such as winds, and groundspeed can be easily calculated with modern technology, whose formulae I can include in my program to also give the user ability to make these calculations. The AI

will also have the ability to look at the airspace along the route and present a route that requires the least amount of altitude changes and avoids danger areas.

Q5 – It was clear in this question that most people wanted some sort of assistance that would help them during their flight planning, which is what my solution aims to provide. Other problems involved outdated NOTAM's being shown, which could be fixed by having some method of refreshing (and behind the scenes, fetching) and gaining the latest information from the selected aerodrome. The altitude problem is mentioned again here, which signifies its importance and I can thus prioritise it as a key fix in my solution.

Q6 – Calculating the distances between points, and the headings (bearings essentially) to fly to get to them seem to be the simplest aspects of flight planning due to their repetitive nature. Such calculations can also be done using modern tools thus making things a lot easier to manage. I may disregard this in my own program as the purpose is to provide advice along the route and not to perform extensive calculations for the user. If I had done this it could take a lot of time and add an unnecessary burden to my programming.

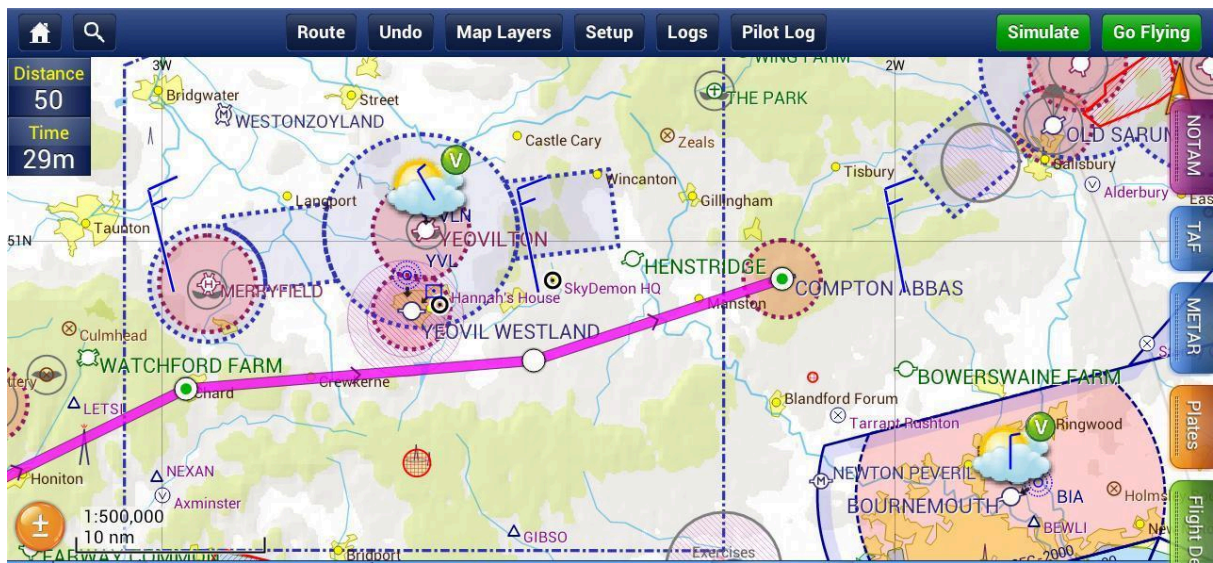
Q7 – Almost all responses here were supportive of the use of AI, and wanted to see things like route optimization, fuel considerations, weather analysis and the like. Fortunately, these are some of the things that I intend to have present in my solution. *Conversely, dealing with updated conditions en-route and diversions cannot be included, as they simply don't form part of pre-flight planning and also require complex information and instruments to operate in real-time. Attempting this would see me wasting a lot of time on something that really and truly would be impossible to implement with my level of expertise.*

Existing solutions

PREFACE

Surprisingly, there were no exact solutions that followed the concept of 'AI-assisted flight planning for general aviation'. They were in fact, far away from that, and the focus was mainly on the airlines (i.e. at a commercial level rather than general aviation) and how they were using AI to assist in route making, fuel planning (reducing costs, that is) and the like. Additionally, because this concept of 'AI-assisted flight planning' is relatively new, many private companies don't provide live demos on their websites and instead ask for company/airline details to negotiate its use for them (which is valid, as a lot of money is to be earned should it be done right). As a consequence of no focus on GA customers and AI tools that aid flight planning on a national level rather than an intercontinental level, I will focus on flight planning software that is common among private pilots that make flight planning easy for them. This will ensure that I can focus more on GA users and focus on the assisted planning of cross-country flights rather than long-haul flights made by aircraft which are far more capable of doing so than a small GA light aircraft.

1 – SkyDemon





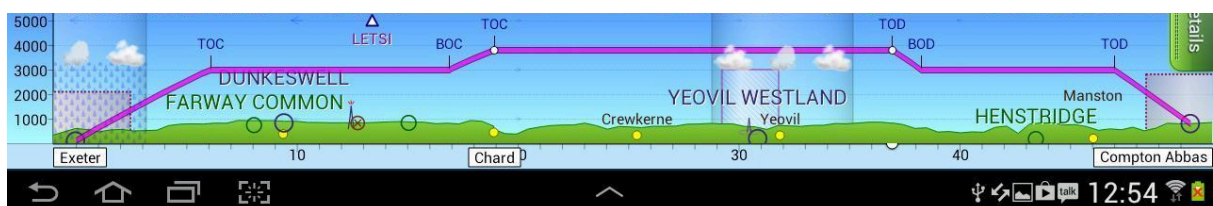
SkyDemon is a navigation software that allows planning of flights, with live data and information, and also provides live data along the route when you are actually flying it. It is an extremely useful piece of software that is used all over the UK, ensuring safer flights and a huge reduction in pilot workload. Its use is preferred by many other than the traditional methods, which is why its use is often present during training as well.

As seen in the questionnaire responses, SkyDemon is easily the best navigation tool for pilots that fly cross-country using visual reference to points on the ground (or in aviation, Visual Flight Rules). It is rightfully trusted by many, and its software is intensely complex but very easy to use. There are several things I'd like to take and use to inform how I would like my solution to look like.

Aspects to include:

Route overlays on the map is one of them, also included in other navigation software, and this simply means showing all the airspace and the actual route **line** that the aircraft will be following. This will allow the user to visually see what their route will look like and make a decision on whether to take use it or refine it themselves. Alternatively, I could simply have a map showing a direct line from point A to point B, but this does not help the user to know how their surroundings will look whilst in flight and how their flight is impacted by the different airspace.

Quick access to weather tools and information (on the right hand side of the above image) is also something that would be beneficial, as it will allow the user to check reports as often as they need in a quick manner as they are constantly changing throughout the day. If such data is placed far away from the user, more clicks would be required to navigate there and back all the time which wastes time and is inconvenient for the user.



Vertical profiles of the route (see above) to allow the user to see what the route looks like relative to the surrounding airspace and the ground. It also allows the user to see how long each segment of the route will be. This is extremely useful when trying to avoid accidentally infringing airspace and also keeping vertical separation from ground features. **It's not an essential feature per-say, but without it it can make planning altitudes a little more difficult than it could be, thus it being something definitely worth adding.**

Aspects not to include:

Weather like clouds on the vertical profile cannot be implemented because clouds are actively moving and this would require me to constantly update the map. This also isn't necessary because I won't be making a tool to help pilots whilst flying but only whilst on the ground pre-flight. **Including this will require live changing data which cannot be accessed solely from weather reports used for flight planning, making things unnecessarily much harder for me.**

2 - AeroWeather

The screenshot shows the AeroWeather app interface on an iPad. The left panel displays a list of aerodromes with their names, ICAO codes, and current weather data (wind, temperature, pressure, and cloud height). The right panel provides detailed weather information for Waddington, including a METAR report, TAF forecast, and various weather parameters like wind direction, speed, visibility, and cloud cover.

Aerodrome	ICAO	Wind	Temp	Pressure	Clouds
Waddington	WTN EGXW	220° 19 kts	5°C	988 hPa	few clouds at 1600 feet
Scampton	EGXP	220° 20 kts	5°C	987 hPa	broken clouds at 1600 feet
Coningsby Royal Air Force B...	EGXC	210° 16 kts	6°C	988 hPa	few clouds at 1800 feet
Cranwell	EGYD	210° 24 kts	5°C	988 hPa	few clouds at 1800 feet
Humberside	HUY EGNJ	210° 22 kts	6°C	987 hPa	scattered clouds at 1900 feet
Doncaster/Sheffield	EGCN	210° 19 kts	6°C	988 hPa	scattered clouds at 3700 feet
East Midlands	EMA EGNX	220° 18 kts	6°C	989 hPa	broken clouds at 1600 feet

Waddington Weather Details:

- METAR:** at: 09:50 LT (7.) 37 minutes old. Wind: 220° (SW) at 19 knots. Visibility: 10 km or more. Clouds: few clouds at 1600 feet. Temp.: 5°C, Dewpoint 2°C, RH 81%. Pressure: 988 hPa. MIL-Code: BLU TEMPO SCT018 WHT.
- TAF:** issued at: 07:41 LT (7.). Forecast from 09:00 (7.) to 18:00 (7.): 200° (SSW) at 18 knots gusting to 28 knots. 10 km or more. few clouds at 1400 feet.
- Probability 30%:** Temporary 09:00 (7.) to 11:00 (7.): light showers rain. scattered clouds at 1400 feet.
- Becoming 10:00 (7.) to 13:00 (7.):** 240° (WSW) at 25 knots gusting to 35 knots.
- Temporary 12:00 (7.) to 16:00 (7.):** 6000 m. light showers rain. scattered clouds at 2000 feet.

AeroWeather is a mobile app that allows you to obtain detailed weather information from aerodrome reports decoded in a very nice and simple format. This will form the basis of how I will decode weather reports and present them, as this app achieves them in a user friendly way which is essential when it comes to making flight planning as less of a complex task as possible.

Aspects to include:

Wind data (direction & speed) at different aerodromes/places along the chosen route, which gives the pilot a good understanding of how the weather changes along the route. I really like the way this app displays this data as a side menu, as you can find data from

many different places in a detailed but readable format. Not including such a system would mean that the pilot would have to constantly search for airport data again and again to identify changes and expectations along the route. This wastes time and is undesirable for flight planning.

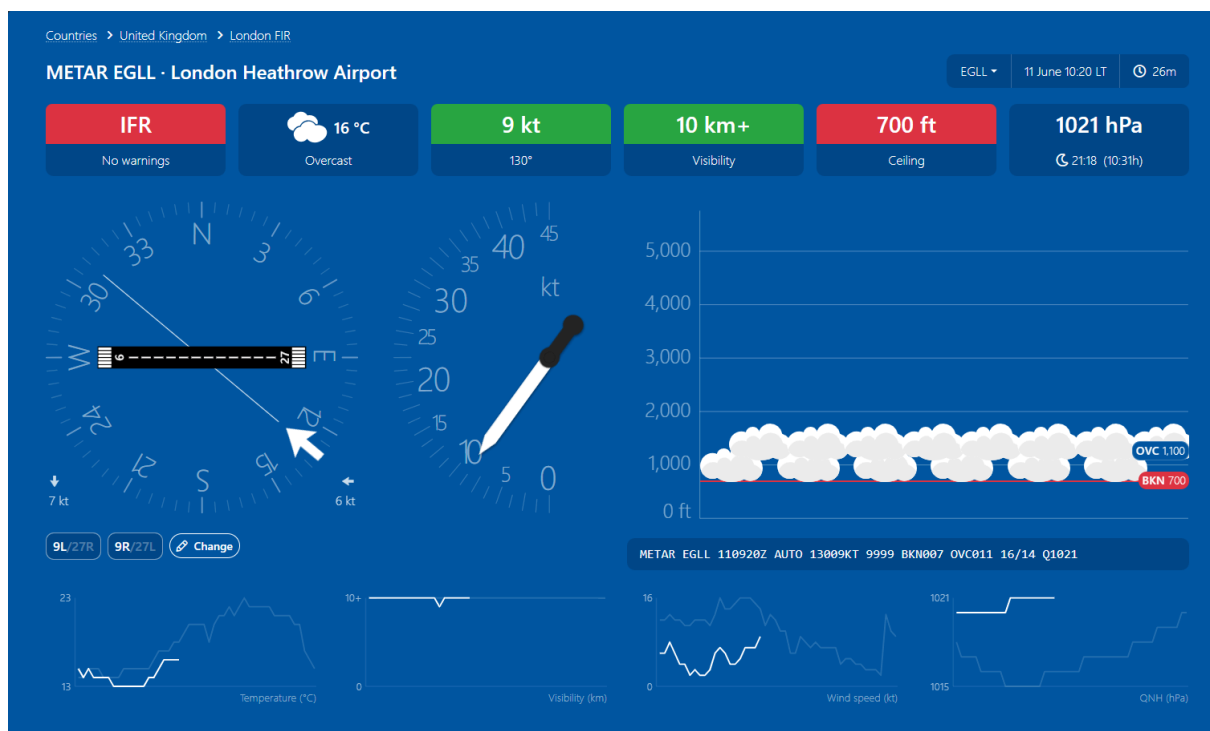
Sunrise and sunset times to allow for flights being planned around these specific times (pilots must be separately qualified to fly at night- some may need this data to avoid these conditions if they don't have these qualifications yet). Not including this could cause both the pilot and the AI who uses this data to make suggestions; to miss out on this crucial information and cause the pilot to inadvertently fly at night unintentionally, whether they're qualified or not. Either way it's very unideal.

The age of the reports is also a key feature, as a sensible pilot would disregard a METAR report that is 55 minutes old since they are updated every 60 minutes - it would be better to wait 5 minutes to obtain a fresh new report. Without this data, the pilot may plan a flight with reliance on data not suitable enough and could possibly lead to miscalculations and in-flight errors.

Aspects not to include:

Weather maps: Whilst they could theoretically be pulled from weather API's they are often not so much of use for the pilot. I say this because there is already a way to obtain actual UK geographical weather data on a large scale (which is essentially what the weather maps display) for free, and that is from the Met Office's Aviation Briefing Service! It's the only service approved by the Civil Aviation Authority. In respect to this, I believe it is better to allow the pilots to obtain such data from official sources which will always be accurate than from other sources that could potentially contain errors that could prove detrimental en-route.

3 - METAR-TAF.com



Time	wednesday 10:00	wednesday 11:00	wednesday 12:00	wednesday 13:00	wednesday 14:00	wednesday 15:00	wednesday 16:00	wednesday 17:00	wednesday 18:00	wednesday 19:00	wednesday 20:00	wednesday 21:00	wednesday 22:00	wednesday 23:00	thursday 00:00
Code ⓘ	IFR	VFR	VFR	VFR	VFR	VFR	VFR	VFR	VFR	VFR	VFR	VFR	VFR	VFR	VFR
Weather	 Overcast	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few clouds	 Few cloud
Visibility	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+	10 km+
Ceiling	900 ft	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Wind	 150°	 150°	 150°	 150°	 150°	 150°	 150°	 150°	 090°	 090°	 090°	 090°	 090°	 090°	 090°
Speed	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt	10 kt
Gusts															
Temp/Sun												☁ 21:18			

Original

TAF EGLL 110830Z 1108/1212 15010KT 9999 OVC009
BECMG 1108/1110 FEW020
BECMG 1117/1120 09010KT
TEMPO 1209/1212 7000 SHRA
PROB30 TEMPO 1210/1212 10015G25KT 3000 +SHRA +TSRA

This website provided a great example of what I would like the weather reports to look like in their decoded forms. You can see that all this information is obtained from just the METAR or TAF, which is just a single line or few lines of coded text.

Aspects to include:

METAR EGSX - North Weald Airport						EGSS ▾	12 June 13:50 LT	🕒 21m
VFR	 21 °C	14 kt	10 km+	None	1011 hPa			
No warnings	Clear	100°	Visibility	Ceiling	🕒 21:18 (7:07h)			

A similar UI to this website in terms of showing the decoded METAR and TAF information is the main thing I would like to include. Especially the way the report is summarised precisely at the top of the page in the METAR report (the first picture). Having this information displayed in this manner will make viewing weather reports much more intuitive and helpful to the reader.

For instance, the above shows, using the data from a report, whether the aerodrome and its surroundings are suitable for Visual Flight Rules (VFR), Marginal VFR (MVFR), Instrument flight rules (IFR) or Low IFR (LIFR) - which is the worst possible weather condition. These aren't explicitly stated in the report but we can use the data from the report to find out which category it falls under.

This is useful to pilots using my solution as they will be focused on areas with VFR and MVFR conditions, and simply seeing it on the weather report saves time reading the details and remembering the category boundaries to figure out whether or not they can fly to a specific aerodrome.

4 - E6BX

E6BX

Flight Scheduler^{Free!}WeatherE6BE6B SimNavLogHoldsCalculations ▾Other ▾

E6BX App is now available. Download Now: [Apple Store](#) or [Android](#)

Online E6B Computer / E6B Emulator

E6BX E6B Simulator ^{New}

World's most popular E6B Calculator, aka CR, CRP-5

Heading, Ground Speed, & Wind Correction Angle

Course :

320

True Air Speed :

100

Wind Direction :

90

Wind Speed :

23

Wind Correction Angle :

10

Heading :

330

Ground Speed :

113

Illustration

Wind: 23 @ 90

MCA : +10

Course : 320

Heading: 330

GS : 113

TAS: 100

The [E6B flight computer](#) is the American version of the flight computer, which is ever so slightly different from the 'CRP-5' flight computer I linked in the very first section of this document (used in the UK primarily). As I've mentioned several times before, these manual instruments, whilst can produce accurate results, are still subject to significantly more human error as it's much easier for someone to read off a dial incorrectly than to read off a set given output, like that shown above. [E6BX.com](#) is the modernised version of the flight computer and can perform all calculations that could also be performed using the traditional flight computer. It's extremely reliable, I've used it myself in practice, and it's even used by the US Military!

Aspects to include:

Basic flight computer functions such as that shown in the above image (Heading, ground speed, and wind correction angle). These calculations are the bare minimum of what is required for the safe planning of a route whilst accounting for the wind also pushing the aircraft from side to side. Luckily, all that is involved in creating these functions is the knowledge of the formula to calculate it- then the rest is just parameters and output. **Not including such essential calculations would cause the pilot to unnecessarily have to go to other websites like this one to do it and would cause them to have to constantly switch tabs or something similar, which again wastes time and will frustrate the user.**

NavLog Calculator, VFR and IFR Flight Planner

World's most popular Navlog Calculator

☐ Calculate TAS using IAS

Save Load Reset

True Course	Alt.	True Air Spd	True Wind Dir	Wind Spd	Wind Corr. Ang.	True Hdg	Mag Var	Mag Hdg	Mag Dev	Comps Hdg	Check Point Name	Dist	Grnd Spd	Time	Fuel Per Hour	Fuel Used	Fuel Rem.
												0.00	0			0.00	0.00

Add New Line

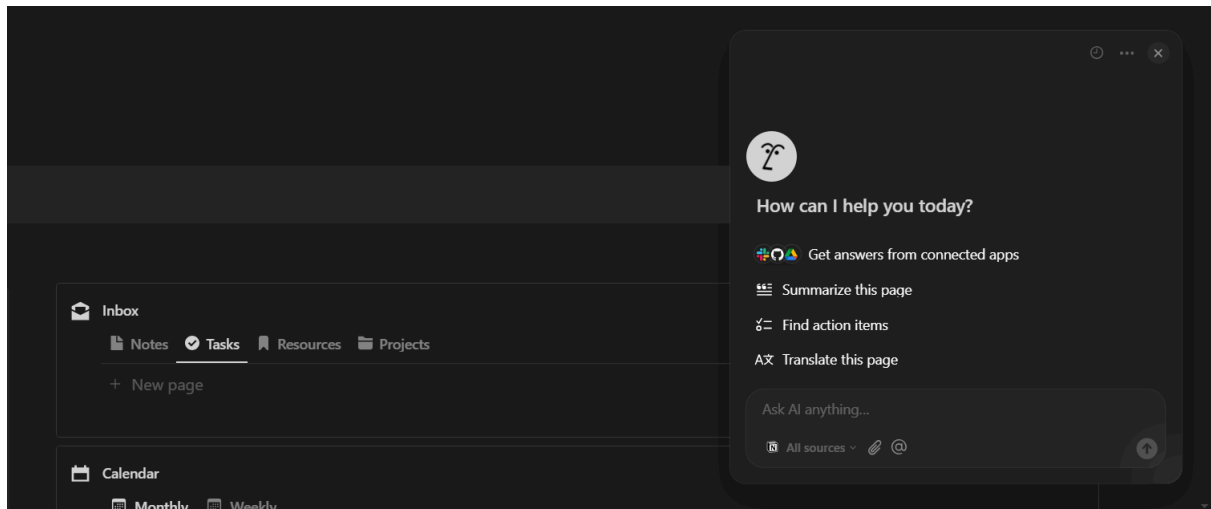
A **pilot navigation log** such as the one shown in the picture above will allow the user to visibly see this data, along with other information such as waypoints across the route, in one neatly consolidated log. Every pilot flies with one, whether virtually or as a paper log, which, from what I've seen, is more common. Thus, I may consider making the completed log into a printable format to allow the user to adjust it whilst actually in flight. Alternatively I could most definitely keep it digital, but typing with hands whilst flying is harder than you can imagine it, and I say this from experience: it's much easier to write with one hand and have the other on the control yoke (which, although still requires some high level of situational awareness, is the expectation).

Aspects not to include:

Other calculations such as pressure altitude, density altitude, cloudbase altitude, Holding pattern calculator, Mach and True airspeed calculations. These calculations, whilst beneficial, do not form the foundation of flight planning and therefore aren't essential in my solution. Furthermore, this is not supposed to be a tool that is used to construct an entire flight plan, rather, I would encourage users to use other apps along with mine to ensure that they get the most out of their flight plan. It's what we do in real life, after all; And, from a computational point of view, adding these functions could unnecessarily take a lot of time and could lead to me not being able to fully complete other more essential items in my program.

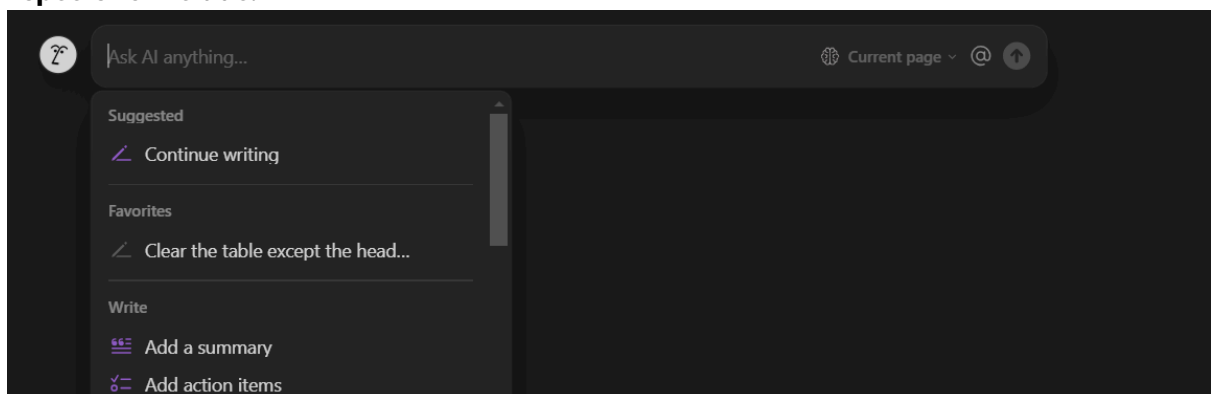
Unit conversions are also unnecessary to add because they don't often have to be performed as we often work on set units already here in the UK, and given that this solution is made only for UK licensed pilots it'd be unnecessary to allow conversions such as from degrees Celsius to degrees Fahrenheit. Also, because these conversions aren't frequent, there would be no problem with switching to other tabs like this one (E6BX) to perform a quick conversion. If I do include them, (which is a relatively simple job if you have the formulas) it could be a waste of time, so I will only consider doing it if everything is finished but time is still present.

5 - Notion AI



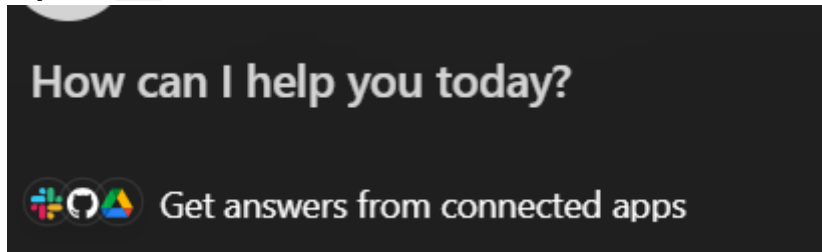
It's important to consider existing solutions related to the AI part of this project. For context, Notion is a note-taking app that allows you to create entire personal databases, and workspaces, integrate things from other websites, and basically allow you to have one hub that you can access certain things from. Many might use it to manage finances, school studies, or similar. *Notion AI* is a recent but powerful addition that allows it to perform actions on your behalf, such as making tables, filling data, or even answering general questions.

Aspects to include:



UI Elements such as the dropdown menu shown above (and in the title picture). I wish to follow a similar style because it looks neat and easy to understand for the reader. I also like the concept of having pre-written options ready for the user related to what they are currently doing. Having a complex UI that is harder for the reader to understand will degrade the user's overall experience as they take their time to understand every aspect of the interface.

Aspects not to include:



Getting answers from ‘connected’ apps is something that I do not plan to include in my solution. I don’t intend to give the user the ability to connect to any external apps that are related to flight planning as my research shows me that there isn’t a lot of integration between them (They all essentially do the same thing).

I will, however, if I have time, perhaps give the user the ability to send their flight log to their google drive or similar, but this is more of a like than a need. *It is much easier to give the user the ability to download a printable file onto their own computer than connecting to external apps, which is why I prefer it more and it will be what I attempt first.*

Essential features

Feature	Explanation	Justification
Disclaimer	A disclaimer stating that the pilot in command is responsible for all decisions made in flight planning and thus must not rely solely upon the outputs provided by my solution.	This statement must be present simply to keep my product safe in a legal sense. Not including this disclaimer would put me as responsible for anything that occurs as a result of the pilot using my solution, which is not ideal.
Ability to enter departure/Arrival aerodromes	The user should have the ability to enter the departure and arrival airports to provide a foundation for route generation/analysis	A flight must have a departure and arrival aerodrome to be a flight. (Even if departure airport = arrival airport)
Integration of aerodrome forecasts and reports	This allows the user to see the weather conditions along the route, at the departure and at the arrival aerodromes.	This is essential for the safe execution of a flight in Visual Flight Rules. Without it the pilot runs the risk of entering unsuitable flight conditions.
Route generation	A route should be generated that avoids controlled airspace and has optimum distance, fuel and altitude values.	This will ensure the pilot has reduced fuel costs, a smoother ride, and avoids unnecessary workload caused with controlled airspace.

Route map (Vertical profile not essential)	A map showing the generated route from the departure to arrival aerodrome.	This will allow the user to visualise their route across the different airspace and zones, which helps to identify areas of higher workload (due to busy airspace, parachuting, skydiving or gliding activity, for example)
Pilot navigation log	A pilot log containing flight information related to distances, waypoints, airspeed, altitude, heading, etc. To be in a printable format such as png/pdf.	This flight log can be then taken on the aircraft and written on in flight if any edits need to be made. This is useful as it allows the user to have their planned flight consolidated in one sheet and can actually be used in flight.
Basic flight calculations	Calculations such as groundspeed, heading, airspeed, and flight time.	These calculations are essential in flight and are used to determine critical features such as ETA, which is often required by the arrival aerodrome. Without this, therefore, flying would not be possible.
Fuel Calculations	Fuel calculations for the duration of the flight, including extra fuel in case of delays which require the aircraft to be in the air a little longer. (Happens more often than not)	Fuel is of course an essential consideration for any flight. If not considered, and the tank was just filled without consideration, the mass and balance is affected and this could easily lead to a forced (emergency) landing!
Integration of AI for analysis & suggestions	This is the feature that differentiates my solution from others. The AI is supposed to analyse everything from the route, to the fuel to the weather and everything in between, identifying any dangers to look out for if any, and suggesting alternatives.	This feature is required in order for the solution to work as intended. Of course, without this feature then this becomes a simple flight planning app, which no-one would use as there are much better solutions out there that do so too, and to a better standard.
Mass and balance, and flight performance calculations	The solution should calculate the certain weights and weight limits for the flight based on user input. Pre-calculated flight	These are the most critical calculations and no flight can be performed without them. The performance calculations involve

	performance calculations can then be combined to form a robust system for the flight.	complex mathematical formulae that only estimate the required values so I allow the user to consult their manual instead and input pre-calculated values.
--	---	---

Limitations

The use of Artificial Intelligence

Of course, using AI in this project comes with its associated limitations. The first of them being the fact that it is AI, and, as you will see all the disclaimers say: 'AI can make mistakes. Check important info'. To prevent this limitation from having an effect on the accuracy of my solution, I will have to decide which parts would I require a LLM to analyse, and which parts of my project I would have to write my own algorithms for. For example, I would use an LLM to analyse and suggest a route to the user, but I would have to create my own decoding algorithms to decode weather reports and NOTAMs, before sending *those* to an LLM to analyse and change things based on that data. *Doing it like this would ensure that must-be-accurate data is always accurate and completely eliminates the danger of AI giving incorrect data to the user. Whilst I fully believe AI is capable of analysing a simple weather report, if even a single mistake is made, so many factors could change, which could seriously affect certain aspects of the flight plan.*

The second limitation related to AI is the fact that it's not usually a free service when it comes to LLM APIs. Although relatively cheap (for example, GPT 3.5 turbo), with extended use, funding could become a problem. For the moment I don't plan to open this fully to the public anyway as, even if the idea is somewhat original, this is a small-scale project afterall.

Technical limitations

The main technical limitation is the fact that this is a web-application, so it will not work without an internet connection. Fortunately, it is difficult to plan flights without the internet in general, as sites such as the Met Office Aviation Briefing service, and the CAA Aeronautical Information Publication (AIP) are all online access and contain essential data for flight planning. Therefore, it is an expectation that all users who decide to use this web-tool already have a device with some sort of access to an internet connection. *I could alternatively make this web-tool available offline, but this would mean that many things wouldn't be present in it, such as the reports I mentioned above. It wouldn't be feasible as a webtool therefore.*

Time constraints

This project is without a doubt an extremely ambitious project, and I am fully aware of the fact that the time constraints may hold me back in terms of what I will be able to complete and what I won't. To combat this, I will have to have a minimum viable product, or MVP, in mind to ensure the project remains successful to some degree. *Not having something like this in place will cause me to simply begin blindly working on the project and not being able to finish with at least the essential features included, and the success criteria satisfied.*

Resources and experience

Another limitation that could affect this project is the availability of certain resources and how experienced I am in using them. I intend to use Flask, which I will use to integrate python into the website, and allow me to make and link each page of my webapp. I plan to watch tutorials on how it works, and I trust that they will give me a decent foundation to build up on to enable me to develop this project effectively. In the case that they don't, another simpler solution will have to be searched for instead, such as simply relying on basic styles and JavaScript. Being familiar with how all these tools operate is essential as without this knowledge I will struggle to make a decent start to the coding, styling and integration of different API's in my project which could potentially render it infeasible.

Hardware/Software Requirements

</> - Developer-Specific Requirements

Requirement	Justification
Device with Internet Access	Since this is a web application, it could be on mobile (i.e. via a tablet) as well, but it must be able to connect to the internet to receive live weather reports and data relevant to the flight, and to access AI.
</> Visual Studio Code	This will be used to code the python and HTML/CSS/JS required for this web-tool. It's useful as there are several addons that make coding easier such as a live updating view of the website. Furthermore, the popularity of VS code means that most tutorials will use this application which makes following them a much easier process.
</> Flask Web Framework	This is required for myself to enable me to develop a user-friendly UI and integrate python into my website.
Windows 10/11	Either version of windows will allow for efficient coding of this project with the latest software available for each version. (although windows 10 support is ending, it has no effect on the website itself).
Modern Web Browser	The user would need any modern browser to be able to run the website. Older versions of Internet Explorer are not adapted to modern website standards and may not work properly for the user.
Screen resolution of at least 1280x720px	This ensures the display is easily readable and can adapt to smaller screens like a tablet. Lower resolutions may make the website hard to interpret as things cannot be read so clearly.

Stable Internet connection (5Mbps+)	This ensures the smooth running of the website in general (pages don't take excessively long to load, server-side code execution etc), but most importantly it is so that the API's do not take excessive time to fetch data, as they may already do so on a stable internet speed depending on the API, and errors could occur if the internet fluctuates in strength, such as a failed fetch.
Minimum 4GB RAM	For smooth performance at least 4GB of RAM should be enough for the user, and this mainly applies for lower end devices as most devices will have more than this.
</> Deployment Environment	If I decide to deploy this as a public website I would have to have a suitable deployment environment. This ensures it is deployed with HTTPS, as well as having secure environment variables for things like API keys. Without this there would be extensive security issues with my webtool and could pose a risk to its users.

Success Criteria

Criteria (13)	Explanation	Justification
Dashboard	This will show all relevant flight details, such as the route, the expenses, the fuel, the planned mass and balance, weather, and full flight brief, All in one place.	This will make it much easier to view the details of a flight without having to cycle through all the different pages, which is time consuming and not intuitive for the user.
Route generation	Using AI and the departing and arrival aerodromes inputted by the user, a route should be generated considering factors such as avoiding controlled airspace, fuel quantities, danger areas, TFRs, NOTAMs and more.	This is the basis of this AI web-tool and it gives the user some initial values to work with so that they can discuss alternatives with the AI and the AI can then generate another route based on that. Having the user make their own route could also serve to be useful, and I could even include that as an alternative, but the aim here is to lessen workload and

		get routes that consider many factors that a pilot could potentially miss out.
Use of APIs to fetch weather data	The use of APIs to fetch live weather data from any aerodrome. These will be presented to the user, and will also be used to inform how the route will be or how the full flight briefing is at the end. For example, the AI may suggest the flight is at high risk if thunderstorms are in the vicinity and the pilot has a chance of getting caught in them.	This is a must, as it is the only way the user (and the AI) can receive live weather data. Alternatively, the user could use other sites to access such data, but this is counter-intuitive and just makes the flight planning process a much harder one for the user, especially when it's not that hard to fetch this data as it's free and not extremely large.
Use of API's to fetch NOTAM and TFR data	The same applies with warnings and temporary flight restrictions. These will be analysed by the AI and could influence the outcome of the final route/flight briefing.	Again, the user could use other applications to obtain this important data, but it saves them time just presenting it and decoding it here for easy access.
Use of AI (LLM)	Use of a large language model such as GPT-3.5 turbo to analyse route, weather, NOTAM data, etc. This will also be responsible for providing a full flight brief to the user at the end of a flight.	It is very much possible for me to create my own AI algorithm that can do this as well, but the amount of parameters that must be considered means that it would take me an unimaginable amount of time to code. Therefore I only plan to use my own algorithms in things that must be accurate, such as the decoding algorithm explained below.
Decoding algorithm	A set algorithm that looks at certain keywords of a report and decodes them into readable user-friendly english language, that can then be presented to the user or sent to an LLM to analyse and inform the flight brief.	Things such as the Notices To Airmen are sometimes extensive in data, and so everything must be accurate to avoid disregarding warnings that could detrimentally affect the flight. Therefore, it would be unsuitable for a large language model to decode this because, whilst minimal, a chance still exists that it could decode a NOTAM or weather report incorrectly.

Map with route Visualisation	Map showing the route lines, including surrounding airspace, Visual Reference points, etc.	Makes visualising the route easier in regards to airspace and restrictions along the route.
Flight computer	A flight computer algorithm covering basic calculations such as heading, airspeed and groundspeed	These values are usually present on any pilot navigation log and are used especially when calculating distances/times and ETA. Asking the user to use another application is counter-intuitive and unnecessary.
Fuel, Mass and balance, and Expense calculators	These calculators should use data selected by the user to provide reliable values that can then be used to inform the route suggested by AI.	Not including any calculations could cause the AI to plan 'impossible' flights as things such as the fuel capacity and burn rate, the weight and the performance of the selected aircraft was not considered.
Full flight brief	The AI should output a full flight brief containing details such as weather, the departure/arrival airports, things/threats/warnings to look out for en-route, airspace and more.	This flight brief will give the user a good summary of all the planning performed and the tips received by the AI will help in Threat and Error Management. Not including this could potentially result in the pilot missing out on key information related to the flight.
Printable pilot Navigation log	A log containing details about the route, Minimum Safety Altitudes (MSA), airspeed, groundspeed, headings, ETA, etc, in a printable format such as png or pdf.	Not having a flight log to store all this data isn't extremely detrimental as the pilot can simply use their own paper log to write it down, but having it already done online is useful and quicker for the pilot. It can also be accessed offline which is a major difference.
Settings menu	For changes such as units used for measurement, and other minor tweaks that are dependent on the user	The absence of a settings menu would make it only suitable for users who agree with the way I like to plan my own flights, which is of course extremely different and so a variety of options are necessary if I want to get a large user base.

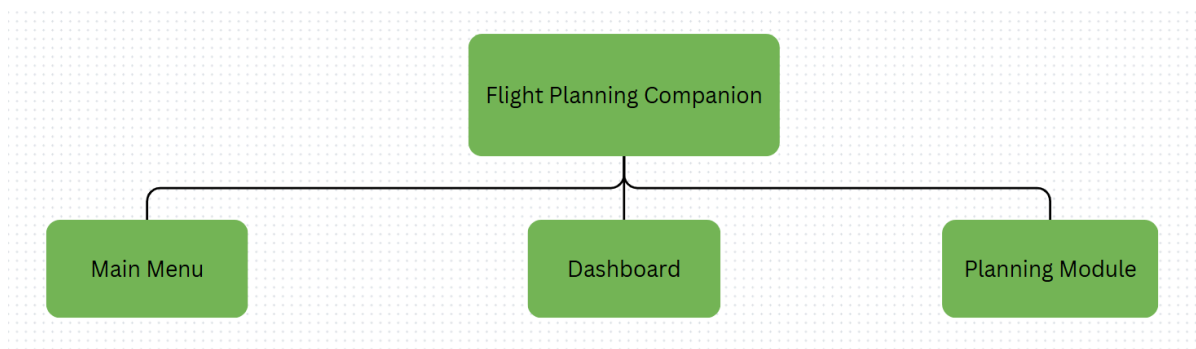
Ability to save, edit and start new flight plans	The user can save the current flight plan in the cache, externally save flights in the computer and log new ones, allowing them to log several flights and edit older ones, etc.	If I decided to not save user data then everything can be lost simply by reloading the page, which is very very disadvantageous when done accidentally and could be a deal-breaker when it comes to users using this web-tool.
---	---	---

DESIGN

Breakdown of the problem - Hierarchy Charts

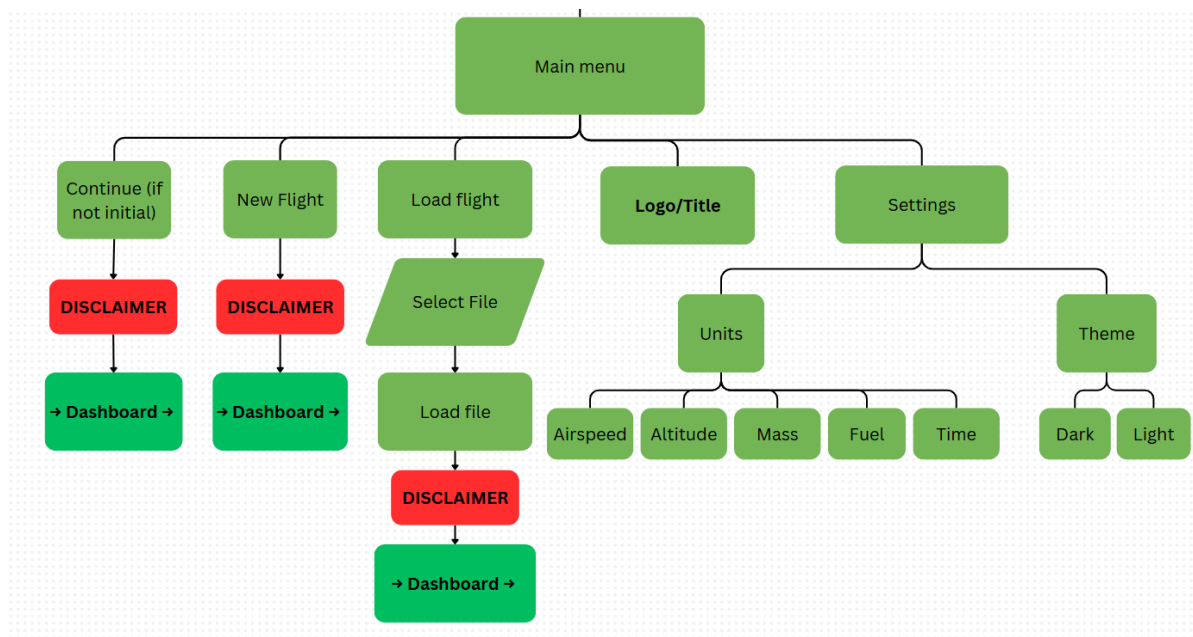
To best break-down this problem in an understandable and readable manner, I have used a hierarchy chart with some embedded flowchart mechanics within. The entire application is separated into 3 main sections: the **Main menu**, **Dashboard**, and **Planning Module**, as shown in the the chart below.

Main breakdown

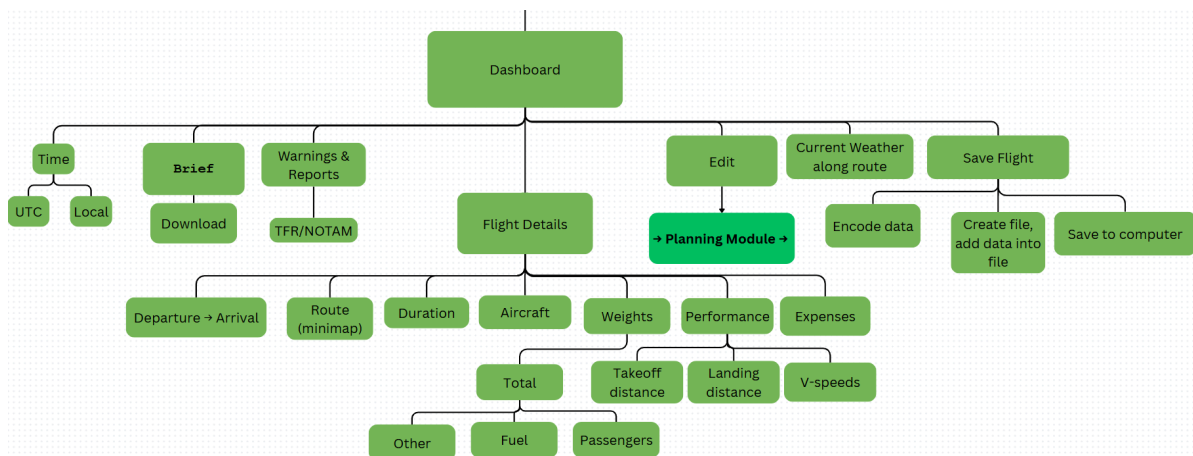


It was important that I did not make an entire chart as one to then display here as there is simply not enough space for such an image to appear legible on this document. I also believe that this is simpler as it is easy to recognise where each section of the chart comes from.

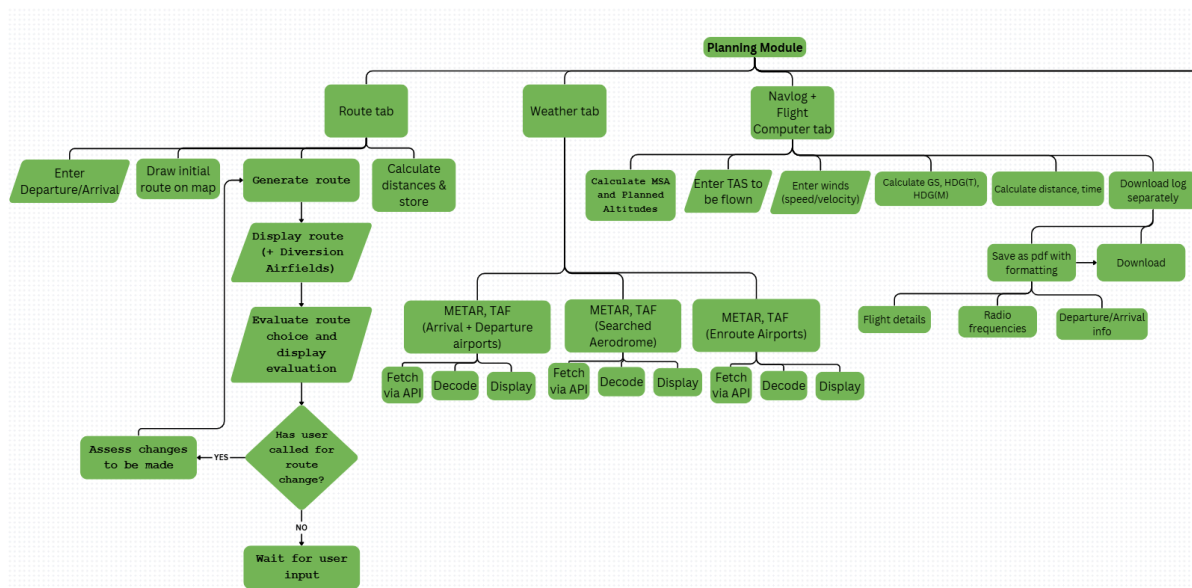
Main Menu



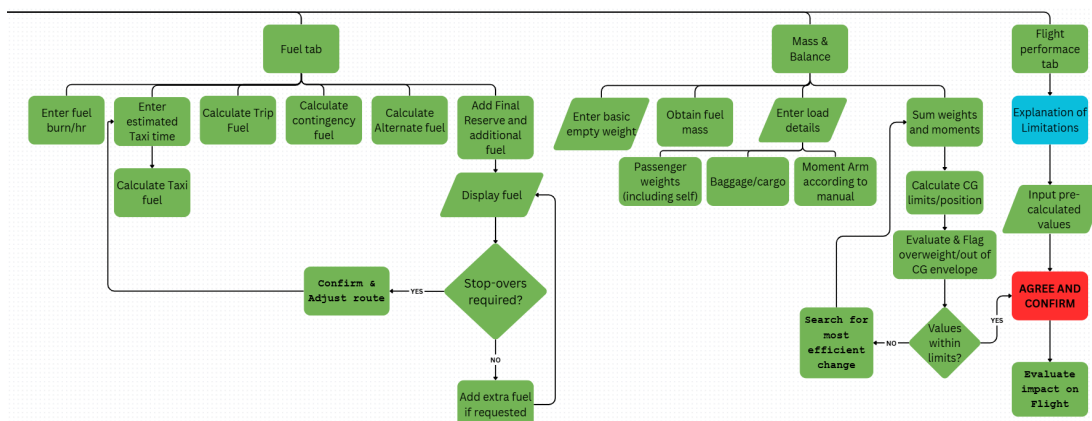
Dashboard



Planning Module 1 (Navigation & Routing)



Planning Module 2 (Flight performance, weights and fuel)



Justification of the process

Decomposition

The main reason for me decomposing the problem this way is to organise the way I develop the solution. The structure gives me a clear view of what happens when and how it will happen, what things depend on it and (in my case) what things must happen immediately after a subsequent action has happened. It also allows me to visualise the modules of the program in their final format, and how they interact with each other. Breaking it down like this will make development much faster and easier as I can keep track of what part of the solution I'm working on, and, by referencing this chart throughout the process, can remain aware of parts that must be worked on alongside another part. I could also not structure it like this and I could instead, for instance use a list, with each category and its features as separate lists. This will make the job much harder for me because I cannot visually see what each part consists of and whether it is dependent on another part. Of course, I could have it written down in words, but It can easily be misinterpreted as I have to separately read each subsection and move to other sections just to confirm a relationship. This hierarchy

chart shows it immediately and therefore reduces the effort required to interpret the design of the solution.

Chart features & Design

In this chart specifically, I have utilised some flow-chart mechanics and also indicated clearly parts that must happen before continuing to an action, parts that lead to another module, and parts that are to be executed by the use of AI:

Having the flow-chart mechanic embedded allows me to easily identify parts of the project that must occur in a specific order or sequence. This key change to the chart was so important for my case for a few reasons:

- The first (and most important) being the fact that there are certain parts, particularly in *Planning Module - Part 2* that have strict links with other parts, which cannot be ignored. Because the calculations in such sections are so critical, outputs from one section must agree with the other, and visually representing it like this makes this relationship clear and visible to me as the developer.
- The second is that it saves me having to draw separate flow charts for each section which will be separate from this hierarchy chart. By having both forms in one single chart, referencing from it becomes much simpler and makes the overall agile development process easier. This doesn't mean there will not be any separate flowcharts at all for other sections, it just means that only sections without critical links will have their own flowchart algorithm shown later in the Algorithms section.

I'd like to note that I am aware that having flowcharts/hierarchy charts embedded like this means that they aren't necessarily shown in the correct layout of a traditional flowchart or hierarchy chart. I am willing to sacrifice the integrity of my chart if it makes it easier to understand for both me and future readers of this document, especially because my project is a bit more complex and I must take effort to simplify things as much as possible.

The **red boxes** are indicators of items that cannot be missed and must be done to full consideration before continuing to the next item, and this only consists of the disclaimer and the confirmation that the mass and balance details agree with the aircraft performance calculations. If either of these are not done, This makes the solution unsuitable for use by my intended audience and the project is incomplete. The single **blue** box is less significant but is also important enough to be singled out as a special item- this will be elaborated further in the detailed breakdown.

Darker green boxes indicate sections leading to another module. Whilst simply a design choice, it makes the chart look more organised and easier to read for me and other potential users of this document.

Parts that use AI are identified using a separate font to distinguish them from normal parts that will be executed without it. Again this is a design choice and I could've indeed simply labelled them with '[AI]', but doing it this way just makes it even easier to distinguish between what is and isn't AI, and anything that would make my chart easier to understand is valuable in my view.

I would also note that, as seemingly complex as this chart may seem already, not all processes and details are included in it. This would make the charts very hard to interpret even for me and the several different branches could potentially be overwhelming.

Detailed breakdown

Main Menu

Logo/Title: To be displayed on the main page, ideally to the left. The title of this solution will be something along the lines of 'AI VFR', with a subtitle of 'AI - Assisted Flight planning', and I have made the decision to have it in a logo-like style as will be shown in the next section. I've chosen to include a stylised title because it gives a sense of originality and assures the user of some sense of legitimacy for the solution and the developer who created it - although subtle.

Settings: This will include some changes that may be of use to the user. I see such settings in most solutions similar to mine and so I have made the decision to include them too as it shows they're of necessity to some. I will note as well that the settings shown in the chart may not be all the settings that will be in the actual solution and I anticipate that it may change through the course of development.

The settings include the ability to change the default units for some values, and the ability to change from a light theme to a darker theme, to avoid eye strain.

Load Flight: The flights should be saved as .json files in the users computer I've chosen to do it this way instead of using a database as I can format the files in such a way that they can actually integrate with other tools that accept such files, which will most certainly be of use to the user. Moreover, a database requires me to make a sign in feature so that users can access their data. This is something that I wanted to avoid as this is just simple flight planning software that doesn't have extensive personalisation to the extent that a user-login is required. The only data to be stored not in a file is the current flight being planned, via the cache.

New Flight: This will clear everything in the cache for the current flight (if anything exists) and will load the empty dashboard.

Continue: The user can continue the current flight being edited (if anything exists).

Disclaimer: The disclaimer, as mentioned in the essential features, is to ensure that I don't get myself into trouble if something goes wrong within my solution and inaccurate results are outputted for any reason. I've already acknowledged this possibility and therefore I must warn the user that this project is for educational purposes only and they should abstain from using it in real flight, despite the intention of the solution itself. pUltimately, they are the pilot in command and are therefore responsible for everything that happens in the course of a flight. I will also emphasise that based on the above I will therefore not be responsible for any errors that my solution may output. The user must accept this by pressing a continue button, which I will have mentioned the conditions of should that user decide to press it.

Ultimately, this disclaimer is an essential part of this project as portrayed by its appearance on the chart. Not including this simply puts me at legal risk if any AI output causes error in flight. In hindsight of this, I am also obliged to ensure that the AI does not, under any circumstances, perform critical calculations related to the flight. This could completely invalidate the disclaimer because I have not put any effort into actually making sure accurate values are shown to the user.

Dashboard

Time: The time should be shown at all times (even whilst in the Planning module- although not seen in the chart) beginning from the dashboard view. This should display both the local

time and the UTC (GMT) time, as that is the time used globally for aviation. In some cases it will be needed and other times it won't, so it is useful to have both displayed.

Warning and reports: I anticipate that these details may be hard to fetch because of the lack of API's that supply this information. If I manage to do so, this section will be dedicated to displaying this information.

Flight Details: All the details This will include all the relevant information for the flight as shown in the chart. This will allow the user to do a quick scan of important information and check if anything needs to be changed. The reason for having this information displayed in this way is due to these details being impacted by different sections of the Planning Module.

Edit: This will be a large button saying 'Edit Flight Plan' which will immediately take the user to the planning Module, where everything can be changed. No changes to the flight plan can be made on the dashboard, as this will be limited due to space constraints and will mean I will have to embed the algorithms used to make normal changes, into the dashboard screen, which will require more than the single group of algorithms that I might use to edit only, for example, the flight plan. Overall it is an unnecessary feature that doesn't affect the user significantly if I don't include it.

Current weather along route: Displays the current weather along the route. There will be an option to refresh this data, which is especially useful when the pilot is just about to fly, as the new data will influence the brief and other figures like aircraft performance.

Save flight: Will be a button that saves all the current flight details as a .json format onto the computer. I will attempt to save it in an ICAO (International Civil Aviation Organisation) standard layout for integration with other flight planning software. This can then be re-loaded into the webtool and read so that the flight can be edited again if required. There was a major alternative to doing this - this was to simply use a database to store data for a particular user's flight details. After some thought I rejected this idea, because I simply didn't want to be responsible for handling the user's data, which will now have to include logins and passwords because user authentication will be required to secure user data. If I just give the user the ability to download it as a file, reading data in a format I set shouldn't necessarily be problematic, and some level of integration like I mentioned before will also be very useful in the case of using data from other flight planning software and editing them here, or vice-versa.

Brief: Finally, the flight brief is essentially a summary of everything on the dashboard screen as we have covered, but on a single sheet (or sheets) of paper. In light of this, It would make sense to include this information with the nav-log, for future reference in flight. There is a lot of information when it comes to a flight plan but having it consolidated so one can quickly reference certain parts of your plan is essential can greatly, greatly decrease the workload of the pilot in flight. It also saves the pilot from having to write down these details- it has to be done and sometimes there's a lot to note down that might not be consolidated properly, not to note that room for human error exists too.

Particular parts of the flight brief, as suggested by the font in the chart, are to be made by AI, and will require a precise prompt for the best possible output.

In essence, the brief should contain the following:

- **A Header section** including the departure and arrival aerodromes, the date, empty space for the pilots name and aircraft registration and type, and the purpose of the flight.

- **The navigation overview** contains the route, the planned altitudes, cruise speed, total distance to cover, and the alternate aerodrome(s).
 - Below this the navigation log could then be inserted.
- **A weather summary** summarising the weather conditions expected throughout the flight, and at the departure and arrival aerodromes. Summaries of the current METAR and TAF could also be included if time allows, as this would mean I will need to have a way of constantly updating these values.
- **Aircraft performance** values such as endurance fuel time, fuel required time, the total planned payload, takeoff and landing distances with confirmations of whether these are within limits or are marginally within limits.
- **Frequencies** for the relevant air traffic control stations that will need to be accessed throughout the flight.
- **Admin checklist** to tick off things like whether you have your licenses and certificates on your person, aircraft documents, NOTAMs checked, briefed passengers, etc.

AI-Generated sections*

- **Notes on airspace** along routes and how they impact the flight, and NOTAMs as well (if they can be fetched)
- **Threat and Error Management**, which involves listing potential threats for the flight and suggesting mitigation measures to avoid them from happening and potentially causing an accident or un-ideal conditions. Examples could be a very tight airspace boundary above and below the aircraft, unfamiliar destinations, turbulence, engine issues on departure/landing.

* The prompts for these entries vary heavily on what I think should be considered for the most optimal response, and I believe the best prompt will be when I've reached the section to actually make the prompt, closer to the end of development. This will allow me to consider all the things that might be different than what was initially planned here in the design section. I have therefore decided not to think of what these prompts will be now and only have a general idea, because the things I consider to not be important now may be very important by the time I get to actually writing the prompt, and so by delaying it until the end of development will allow me to consider all factors that could affect the safety of the flight to the best of my ability.

Planning Module 1 (Navigation & Routing)

Route Tab: This is where the flight's route will be generated. First, the user will enter the departure and arrival aerodromes. An initial straight line from the departure to the arrival will be drawn, to give the user some perspective of where they will be going and to anticipate any changes the AI will make to generate the final route. The user can change this many times as they want to see different destinations and what they might look like.

Then, upon user command, the AI will generate a route. It will suggest a planned altitude, and the route will initially avoid all controlled airspace. TFRs and danger areas will also be considered. The AI will also justify its reasoning for the decisions for the route and display this to the user, which prevents the Black-Box nature of a LLM from having a major impact on the routing and the flight as a whole.

The user can then either:

1. Prompt the AI to change the route according to the user's suggestion
2. Adjust the route themselves.

An alternative approach would see the user make the route first, then have AI refine it. This makes sense in many cases outside of aviation, for example, writing an essay, building/designing something, developing an idea, and so on. Within aviation boundaries however, this doesn't make sense to be done this way, because the AI works on a prompt and is not specifically trained in flight planning, but the pilot will be, making them more capable of producing a route that can be used in flight. It could work if the user decides to make the route and then prompts the AI as previously discussed but I don't want that to be the primary way, because I want to save the user from thinking and considering all the things required to initially make the route, when the AI can do that and the user can refine it, with less of a workload on their shoulders.

A more sensible approach in this regard would be for the user to have some input on the initially generated route. After entering the departure and arrival aerodromes, they can prompt the AI to what they would like the route to look like if they already have an idea. The AI can then generate a route based on this information. If the user doesn't have a particular idea for how they would like their route to look like, they can just press generate and the AI will do the initial work for them, giving a useful foundation for the pilot to work on. This is particularly useful for pilots intending to perform sightseeing flights which means the route needs to pass over certain locations and distance doesn't necessarily matter as much.

Weather Tab: Here, the user can see live weather details of any aerodrome of their choice, or, the weather details of the reports based on the route they have chosen, if they have.

The main key features are the Meteorological Aerodrome Report (METAR) and the Terminal Aerodrome Forecast (TAF) providing a live report and a weather forecast for that airport respectively. These will obviously come in a coded format (fetched from the relevant API), and therefore it will be necessary, match the essential features, to decode them, in which a decoding algorithm will be necessary. After this, the results will be displayed to the user.

Initially, I was only going to show the weather details relevant to the selected route, meaning that if no route was selected, no data can be shown. Whilst this in theory works, it's useful to any pilot to be able to see the weather reports for aerodromes unrelated to the route, because they could be planning to use it in an emergency, or many other reasons I won't go into. It's generally more like a 'why not' scenario as it does not require extensive effort and has a useful result.

Navlog + Flight Computer tab: This will be where the pilot's navigation log will be formulated, then later available to download.

Using data from the route, a Minimum Safety Altitude (MSA) and Planned Altitudes to cruise on for the flight are suggested by AI *initially*, but can be changed by the user if they want to. As before, the user will also be able to see the AI's evaluation on deciding these values.

The user will then input their own True Airspeed that they will be flying (which is highly dependent on their aircraft) and the current winds according to the Met Office Aviation Briefing Service, which I cannot get any access to via an API. (Hence the need to enter this manually). From these values, mathematical algorithms will be used to calculate the

appropriate speed along the ground (GS), the True Heading to be flown, and the Magnetic heading to be flown (difference being one uses true North and the other, magnetic North). The fetched distances from the route, and the GS we just calculated can then be used to calculate the time it should take to reach that certain waypoint, and this repeats until a full route is formed.

The log will then be formatted into a downloadable pdf file, so that it can be downloaded and perhaps printed if required. If the user prefers to have this log as part of their brief, then a checkbox that says 'include in brief' will be visible and they can print it from there along with the brief instead. The log will include details such as basic flight details, Radio frequencies and Departure/Arrival information for easy access.

I've given the user the ability to just have the log downloaded by itself as only the log is essential to carry out the flight. In reality, the brief could just be read digitally before the flight and only the log could be taken on board, but the printing out of *that* on paper had the same general reasoning as the log as well- to allow the pilot to write adjustments and extra notes on the paper.

Planning Module 2 (Flight performance, weights and fuel)

*Items and processes in this Module are somewhat complex to understand for the inexperienced reader. Therefore, such items will be summarised in a way that will be understandable to the reader but still covers all aspects of the section.

Fuel tab: As displayed in the chart, several aspects make up what the total fuel of the aircraft will be. The user will enter the fuel burn/hr, and the time they believe it will take for them to taxi to the runway (as the engine is running then and thus still uses fuel). The fuel required for that period is then calculated.

Using values from the route, algorithms will be used to calculate the fuel for the whole trip, along with all the extra fuel values listed in the chart that are necessary, but I won't go so far into detail about these values.

Mass & Balance: The aim of this phase of flight planning is to ensure that the aircraft is safely loaded for flight. No decisions regarding mass & balance will be made by me (through an algorithm) or AI, as these calculations are so critical that it's actually safer to help the users calculate basic values using the relevant formulas, and allow the pilot themselves to evaluate whether these are within limits by referencing the graphs in their flight manuals provided by the manufacturer of their aircraft.

Moreover, if I did decide to do these calculations myself, they would be extremely complex because I must include data from these physical flight manuals, which of course vary across different aircraft. Collecting this data from each of these manuals is a grueling task, and is unnecessary. Even putting the graphical data into comparable data in code is not a simple task, and even modern solutions such as E6BX who actually managed to implement this, have a disclaimer in their website telling users to consult their manuals and further verify the provided data themselves.

Important: The values and charts provided on this page might vary between model numbers. This is for educational purposes only. You, as pilot in command, are solely responsible for assuring correct data and proper loading of your aircraft prior to flight.

The user will input necessary details to calculate the centre of gravity (CG) position along the aircraft on takeoff and landing as illustrated in the chart (It's not important to know what each of these mean exactly):

- Basic Empty Weight of the aircraft
- The weight of any pilots and passengers
- The weight of any baggage in their respective baggage positions
- The fuel Load (obtained from the fuel calculations)
- The taxi fuel (obtained from the fuel calculations)
- The fuel burned during flight (obtained from the fuel calculations)
- The moment arm from the datum point for all these values. (found in the flight manual)

With this data, a simple formula can be used to calculate the moment for each of these. Then, by adding the total weight and the total moment then we can calculate the centre of gravity position, which the user can then take and compare with their flight manual to see if it is within limits. If so, the user confirms it via a button.

If it isn't, the current load information is made into a prompt and sent to the AI so that they can suggest a suitable change to make the CG/Gross Weight in limits. Again, the user can just make this decision themselves and change the values accordingly.

Flight Performance tab: Unfortunately, there are no calculations related to flight performance that I can calculate, even with my own algorithms. The values we are looking for are usually things like takeoff distance required, the landing distance required, the achievable climb gradient based on the mass of the aircraft, and other factors. These depend on many factors which are completely infeasible to implement in my solution, as this data is obtained from graphs and tables in the aforementioned flight manual, which have been formulated using real-world test data and so are very difficult to replicate in code without just taking the data from the book- but then again you must account for all the aircraft that could be flown as they all have different tables and graphs- it's a long process, and even E6BX doesn't have a performance tab at all to do these calculations, likely for the same reason.

With that, these limitations will be explained to the user on this page and they will be responsible for calculating these values themselves. When this is done, they can put their calculations in the provided spaces which will be added to the brief for reference. In making these calculations, the user would have to reference the mass and balance values calculated prior, and it is imperative that they agree with one another. If they do, this correct data can be passed to the AI to evaluate the overall impact on the flight. No changes should be suggested here unless urgent, only alerts and warnings that may require user intervention. This is because the mass and balance/performance values are calculated last when everything has been considered to the last calculation, and changes after these values are finally calculated are relatively rare.

User Interface

Colours

A good GUI uses good colours. However, whilst I wanted my UI to look good, I didn't want to spend too much time making it look good, as functionality is more important in this regard, especially with my project specifications. I decided to pick four simple colours for the design, namely shades of white, black and blue, as shown below for both light and darker themes, and their HSL equivalents:

LIGHT 



DARK 



```
:root {
  --bg: hsl(0 0% 96.1%);
  --primary: hsl(190 100% 42.5%);
  --highlight: hsl(0 0% 100%);
  --text: hsl(0 0% 0%);
  --onhover: hsl(0 0% 80.4%);
  --popupbg: hsla(0, 0%, 0%, 0.33);
}

.dark-mode {
  --bg: hsl(0 0% 12.5%);
  --primary: hsl(190 100% 27.6%);
  --highlight: hsl(0 0% 20.8%);
  --text: hsl(0 0% 100%);
  --onhover: hsl(0 0% 32.9%);
  --popupbg: hsla(0, 0%, 0%, 0.33);
}
```

The use of **HSL** color codes instead of **Hex**, or **rgb** colour codes was a choice made to make adjusting the colouring a lot easier, because HSL colour values actually have a link to one another, whilst others don't. HSL stands for Hue, Saturation and Lightness (and Alpha for opacity if included) and so do their numbers, meaning I can adjust these particular values simply by adjusting the numbers, and I will be able to predict the output colour, which is not achievable with hex or rgb. An example is shown below, where the lightness is adjusted to make a darker blue.

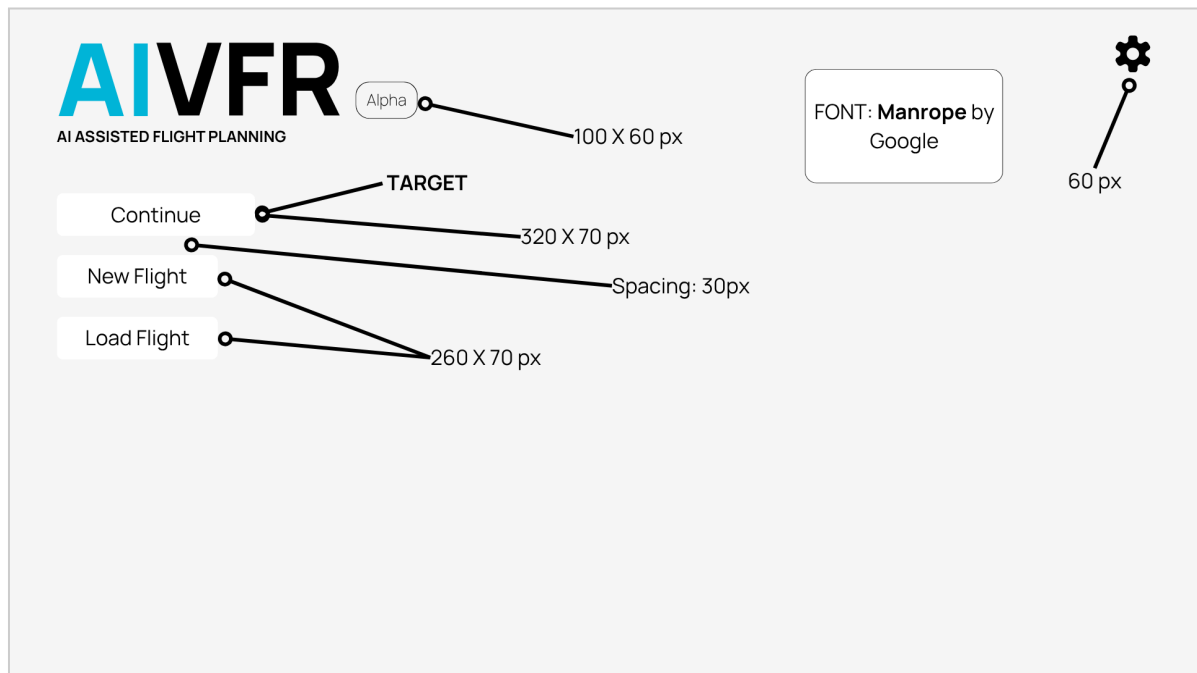
```
test {
  --demo: hsl(207, 100%, 50%);
  --demo: hsl(207, 100%, 20%);
}
```

If I decided not to use HSL colours, and used rgb or hex instead, I will have to manually pick colours by looking up their hex/rgb value online every time, or using a selector wheel that some VS code extensions provide, which can sometimes be frustrating even though it can be somewhat not a long process. Now, however, this shouldn't need to happen at all and I will very rarely have to look for a new colour manually- I can just plug in the numbers with an expected output, making it a much quicker process.

Screen designs

I've made the decision to only show the dark mode appearances for certain screens only. I have created dark mode designs for almost all screens but only the light mode versions of them are shown here for the most part because dimensions and other details have only been annotated on the light mode versions of each screen.

Title Screen (Main Menu)



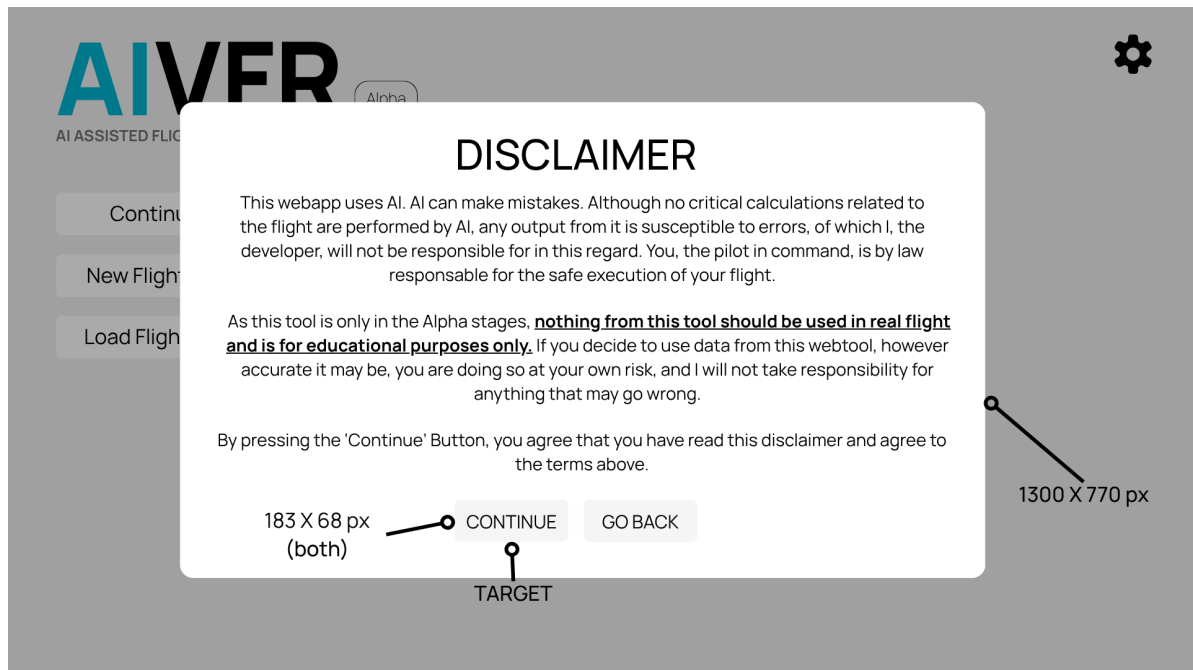
Menu positioning: I decided to align most of the menu features to the left of the screen for a few reasons. The first being that the 'AIVFR' logo's subtitle is aligned to the left by design, and that would mean that placing the logo elsewhere, for example, in the centre or to the right would ruin this design, as I would have to align that piece of text accordingly so it doesn't look weird to the user. The other reason was because it meant that the extra space could be utilized for something else in the future, for example, snapshots of what the webtool looks like and some of its features, which can engage the user into using it.

Font: Manrope by Google was a very good option for my webtool as its letters are extremely clear and minimalistic. This reduces the chances of the user misreading a text element, which could have systemic effects. If I used hard-to-read fonts like **Impact** and the like, the chances of human error increase, and mistakes are more likely; it just doesn't look good either.

Settings button position: After the previous decisions made above, placing the settings button here seemed most practical. It fits with what most applications do if the settings button is not a physical button with the word 'settings' on it, but just an icon- so I decided to adapt this so it feels natural for the user. Making it a worded button as described before would be counterintuitive because it makes it seem as though it has something to do with loading or starting the flight planning process, which is what is suggested by the options 'New flight', 'Load Flight' etc.

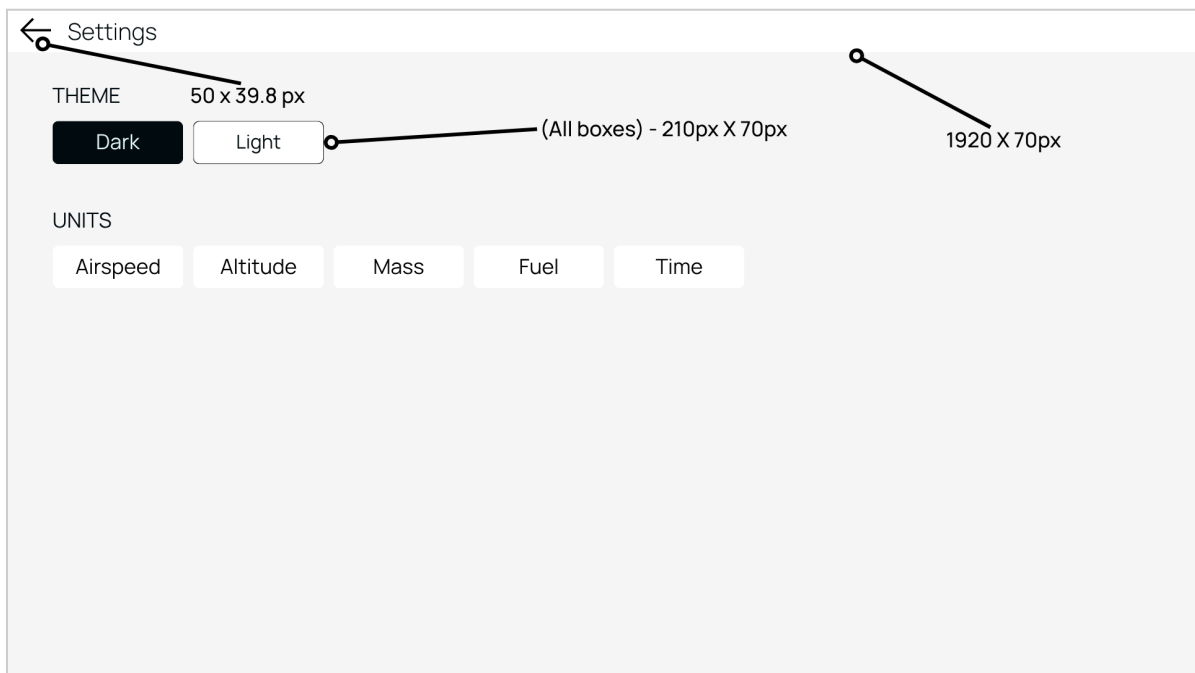
TARGET buttons: I've defined these 'special' buttons as buttons that the user is most likely to want to press on a screen. The primary characteristic of a target button is that when the cursor hovers over it, it will have the **primary** colour instead of the grey on-hover colour, which is to indicate that it is a more important button. This is used in quite a few other web apps and websites, and sometimes the colour change is permanent and not only an on-hover colour. Not including hierarchy doesn't have any significant effect on user experience, so this is simply a design choice. Given that the user is most likely to press the continue button in this scenario, I've selected it to be a target button.

Disclaimer Popup



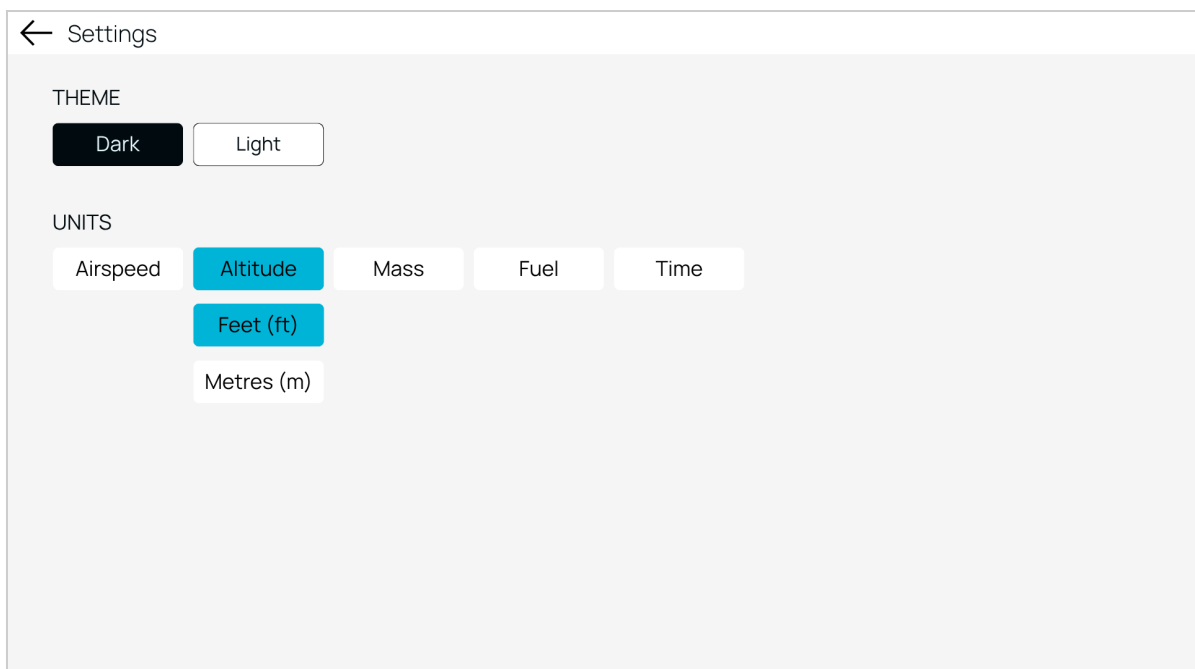
Centre popup display: It seemed obvious to me that the disclaimer should cover as much of the screen as possible, so as to catch the attention of the user. It was evident that it had to therefore be in the centre, as this is the only positioning that looks natural for the user. Any alternative position would seem off-centered and would not look appealing to the user.

Settings Menu (Nothing selected)



When an item is selected, it will turn to the blue primary accent colour, whilst other options related to the setting pressed are displayed, with the active option also in blue, as shown below:

Settings Menu (Item selected)

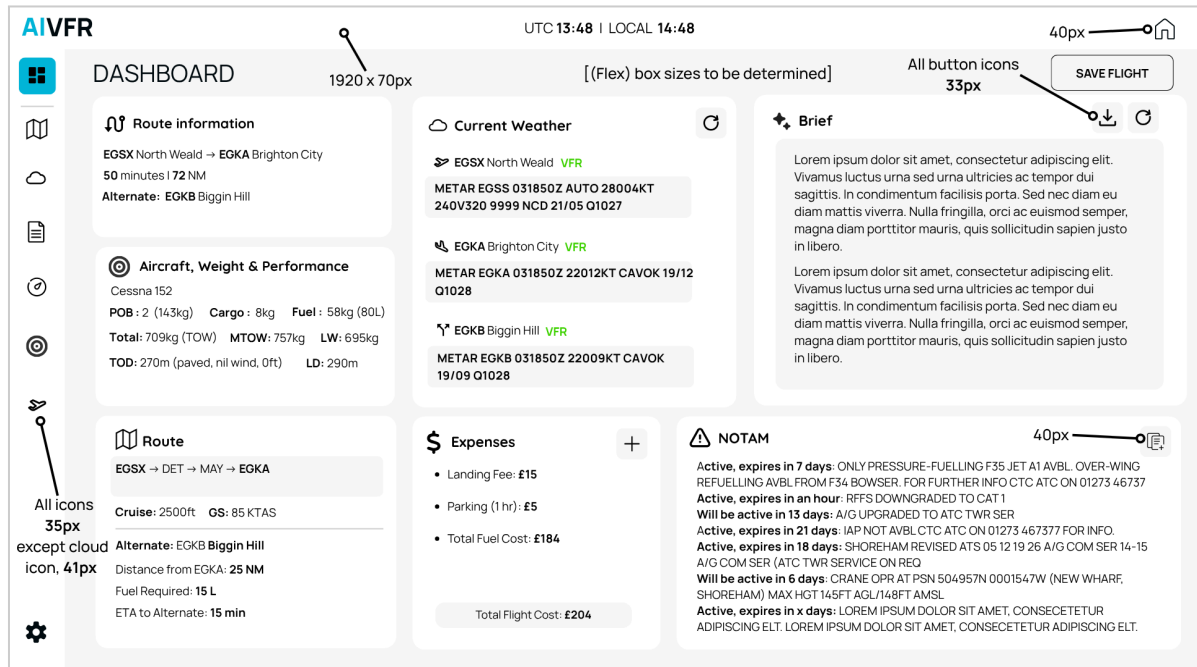


This will be similar to the other setting options as well.

Dropdown design: This is a relatively uncommon way of displaying options in a settings menu, but I specifically picked this one because it meant that I can use the same buttons

and colours and not have to worry about creating other elements, such as switches, which is what I would have included if I followed the design of modern settings menus we see across many apps (i.e. a list with toggle switches). My design doesn't compromise user experience either, as the options are clear and can immediately be interpreted by the user.

Dashboard - light mode



Boxes: I'm attempting to utilize the flex-box layout in HTML to create these separate sections that display different information, which will ensure they can adapt to other screen sizes. Without this, this would only work properly on a PC and some tablet models, but it won't work on mobile devices. This reason applies for all the other page designs below.

The settings icon has been moved to the bottom left corner of the screen. I made this decision because many other similar UI designs do the same already and thus it would be more intuitive for the user if it is located there. I also wanted to avoid a lot of clutter in the top bar part, which only includes the logo, the time and the home button. Moreover, the home button leads to the settings menu too (just with an extra click), so I didn't want to place the settings menu next to it, and instead place it away which will distinguish their functions.

Dashboard - dark mode

AIVFR UTC 13:48 | LOCAL 14:48

DASHBOARD

Route information

EGSX North Weald → EGKA Brighton City
50 minutes | 72 NM
Alternate: EGKB Biggin Hill

Aircraft, Weight & Performance

Cessna 152
POB: 2 (143kg) Cargo: 8kg Fuel: 58kg (80L)
Total: 709kg (TOW) MTOW: 757kg LW: 695kg
TOD: 270m (paved, nil wind, 0ft) LD: 290m

Route

EGSX → DET → MAY → EGKA

Cruise: 2500ft GS: 85 KTAS

Alternate: EGKB Biggin Hill
Distance from EGKA: 25 NM
Fuel Required: 15 L
ETA to Alternate: 15 min

Current Weather

EGSX North Weald **VFR**
METAR EGSS 031850Z AUTO 28004KT 240V320 9999 NCD 21/05 Q1027

EGKA Brighton City **VFR**
METAR EGKA 031850Z 22012KT CAVOK 19/12 Q1028

EGKB Biggin Hill **VFR**
METAR EGKB 031850Z 22009KT CAVOK 19/09 Q1028

Expenses

- Landing Fee: £15
- Parking (1 hr): £5
- Total Fuel Cost: £184

Total Flight Cost: £204

Brief

Active, expires in 7 days: ONLY PRESSURE-FUELLING F35 JET A1 AVBL. OVER-WING REFUELLING AVBL FROM F34 BOWSER. FOR FURTHER INFO CTC ATC ON 01273 46737

Active, expires in an hour: RFFS DOWNGRADED TO CAT 1

Will be active in 13 days: A/G UPGRADED TO ATC TWR SER

Active, expires in 21 days: IAP NOT AVBL CTC ATC ON 01273 467377 FOR INFO.

Active, expires in 18 days: SHOREHAM REVISED ATS 05 12 19 26 A/G COM SER 14-15 A/G COM SER (ATC TWR SERVICE ON REQ

Will be active in 6 days: CRANE OPR AT PSN 504957N 0001547W (NEW WHARF, SHOREHAM) MAX HGT 145FT AGL/148FT AMSL

Active, expires in x days: LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELT. LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELT.

Whilst designing this I thought of what would feel more natural for the user. In this case, I felt like having an 'edit' button that leads to the sort of navigation bar you can see on the left was not as intuitive because the user now has to go through other buttons to get to and from the dashboard. What made more sense was to simply put the dashboard tab button in the nav bar too, along with the other tab buttons, which feels more natural for the user. Additionally, this makes all the pages easier to code as the same template for the navigation and top bar can be used for each tab. The result is what is seen above in the dashboard layout.

Selecting another tab

AIVFR UTC 13:48 | LOCAL 14:48

DASHBOARD

Route information

EGSX North Weald → EGKA Brighton City
Weather 172 NM
Alternate: EGKB Biggin Hill

Aircraft, Weight & Performance

Cessna 152
POB: 2 (143kg) Cargo: 8kg Fuel: 58kg (80L)
Total: 709kg (TOW) MTOW: 757kg LW: 695kg
TOD: 270m (paved, nil wind, 0ft) LD: 290m

Route

EGSX → DET → MAY → EGKA

Cruise: 2500ft GS: 85 KTAS

Alternate: EGKB Biggin Hill
Distance from EGKA: 25 NM
Fuel Required: 15 L
ETA to Alternate: 15 min

Current Weather

EGSX North Weald **VFR**
METAR EGSS 031850Z AUTO 28004KT 240V320 9999 NCD 21/05 Q1027

EGKA Brighton City **VFR**
METAR EGKA 031850Z 22012KT CAVOK 19/12 Q1028

EGKB Biggin Hill **VFR**
METAR EGKB 031850Z 22009KT CAVOK 19/09 Q1028

Expenses

- Landing Fee: £15
- Parking (1 hr): £5
- Total Fuel Cost: £184

Total Flight Cost: £204

Brief

Active, expires in 7 days: ONLY PRESSURE-FUELLING F35 JET A1 AVBL. OVER-WING REFUELLING AVBL FROM F34 BOWSER. FOR FURTHER INFO CTC ATC ON 01273 46737

Active, expires in an hour: RFFS DOWNGRADED TO CAT 1

Will be active in 13 days: A/G UPGRADED TO ATC TWR SER

Active, expires in 21 days: IAP NOT AVBL CTC ATC ON 01273 467377 FOR INFO.

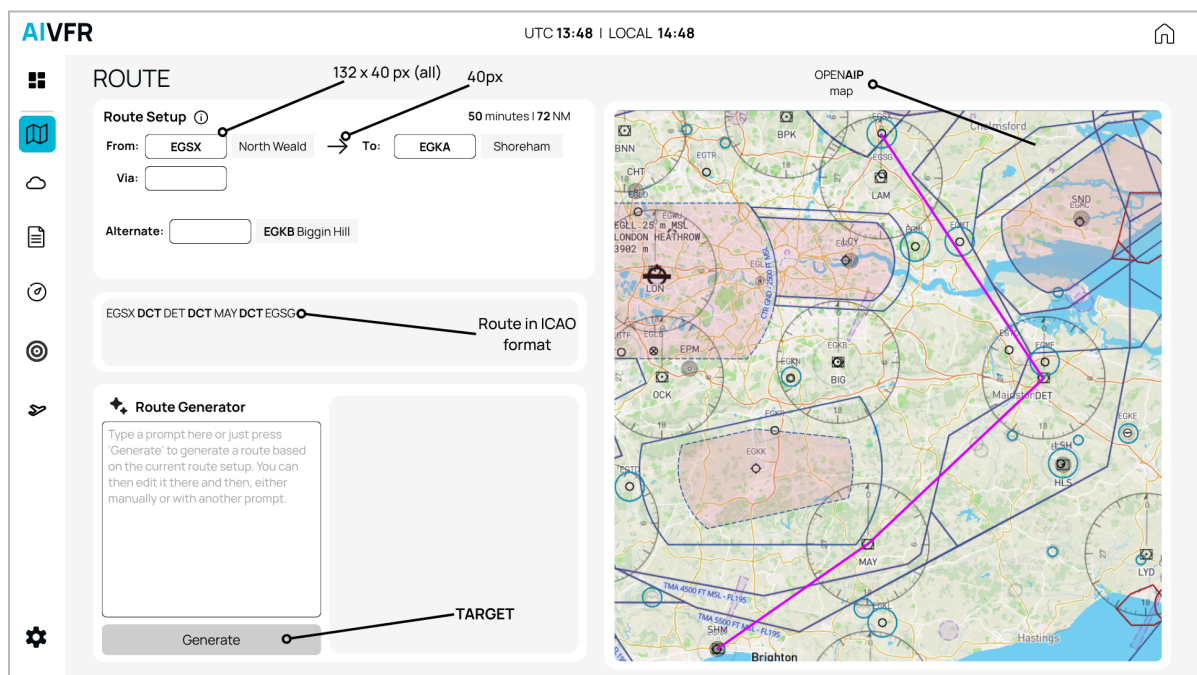
Active, expires in 18 days: SHOREHAM REVISED ATS 05 12 19 26 A/G COM SER 14-15 A/G COM SER (ATC TWR SERVICE ON REQ

Will be active in 6 days: CRANE OPR AT PSN 504957N 0001547W (NEW WHARF, SHOREHAM) MAX HGT 145FT AGL/148FT AMSL

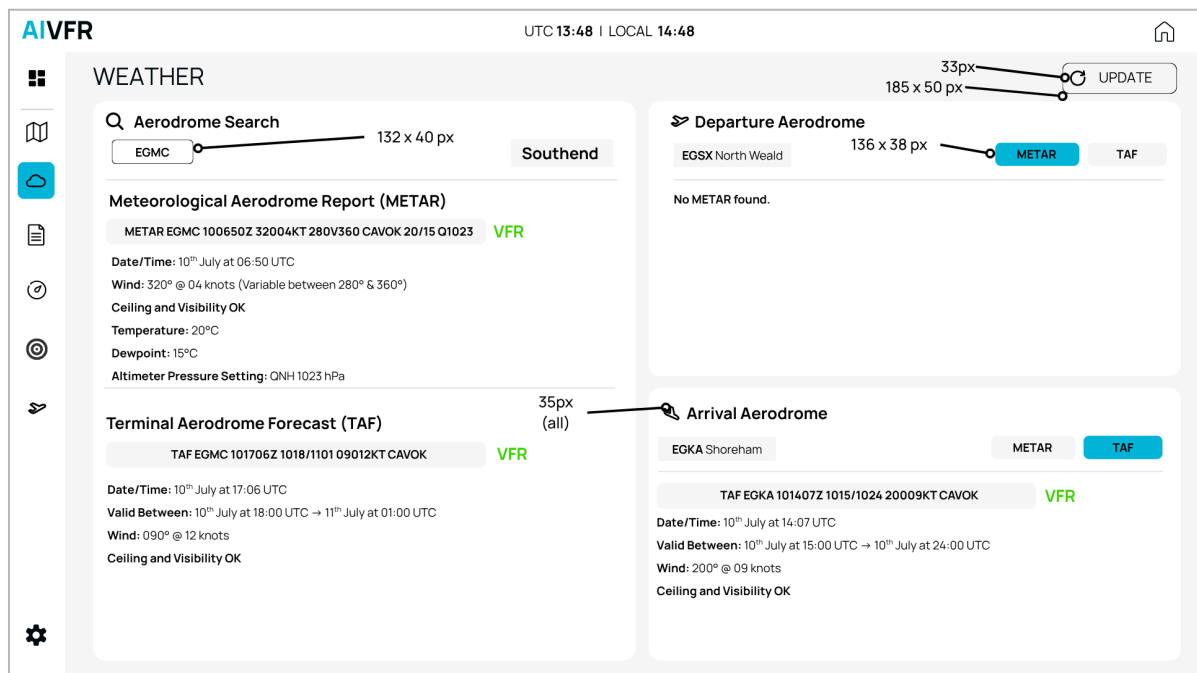
Active, expires in x days: LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELT. LOREM IPSUM DOLOR SIT AMET, CONSECTETUR ADIPISCING ELT.

When selecting other tabs, I needed a way for the user to know what they're clicking without needing to extend the whole sidebar, because that didn't seem to look nice when I looked at it. My ultimate decision was the above, where there is a gradient block that has a low opacity, with the item name showing as a small box whenever the cursor hovers over the tab button. That gradient block is necessary because it ensures that the button is actually visible to the user; it uses the same colours as the boxes below it. In doing it like this, the design is improved because I no longer have to extend the whole sidebar and include all the different names of the tabs when only the relevant tab should be shown to the user. I would also have a hard time finding out how to make sure the extended navigation bar is readable because it also uses the same colours as the boxes below it.

Route tab



Weather tab



The 'No METAR found' message here is true for this scenario, and is placeholder text to represent the output when an aerodrome does not have any published METARs. This will of course be a form of testing in the development of this project.

UPDATE button: Putting this button in the top-right corner of the screen was the better alternative to having the same button (or maybe just the icon) on each of the sections (search, departure aerodrome and arrival aerodrome). The reasoning being that at no point will a pilot require an out-of-date METAR or TAF, and so not giving them the ability to refresh just a particular report such as, for example, specifically the departure aerodrome, is not detrimental to the user. Instead, whilst the user may be focused on seeing the reports for an aerodrome they've searched, the reports for the departure and arrival aerodromes would update simultaneously revealing any changes in the weather.

Navigation log tab

The screenshot displays the AI VFR Navigation Log interface. The top section includes a title bar with 'AI VFR' and a timestamp 'UTC 13:48 | LOCAL 14:48'. Below the title bar is a navigation bar with a sidebar containing icons for home, list, cloud, document, checkmark, target, and settings. The main content area is titled 'NAVIGATION LOG' and features a form for flight parameters: TAS (90 KTS), VARIATION (1E), and WIND (210 / 10). A blue 'AUTOFILL' button is present, with annotations indicating its width (32px) and height (33px). Below the form is a table with flight data. The table has columns: FROM / TO, MSA, ALT PLAN, TAS, TRK (T), WIND, HDG (T), HDG (M), GS, DIS, and TIME. The table contains four rows of data. The first three rows have a light gray background, and the last row has a yellow background. An annotation points to the 'TAS' cell in the yellow row, showing the CSS rule 'hsl(49 83.2% 79%)'.

UTC 13:48 | LOCAL 14:48

NAVIGATION LOG

TAS: 90 KTS VARIATION: 1E WIND: 210 / 10

33px 32px

AUTOFILL

FROM / TO	MSA	ALT PLAN	TAS	TRK (T)	WIND	HDG (T)	HDG (M)	GS	DIS	TIME
EGSX - DET	1900	2200	90	146	210/10	152	151	85	30	21
DET - MAY	1900	2200	90	227	210/10	225	224	80	25	19
MAY - EGKA	1900	2200	90	235	210/10	232	231	81	19	14
EGKA-EGKB	1900	2200	90	023	210/10	22	21	100	32	19

Any item on this pilot log can be changed at your discretion.

hsl(49 83.2% 79%)

The different colour used on the bottom row of the table is to signify that this is not part of the main route and is simply the direct route to the alternate aerodrome. This scheme is used in many logs already such as SkyDemon and even physical logs, and is well known and easily understood by pilots. Hence, not having it coloured this way will confuse the pilot into thinking it is part of the route which is not useful in the cockpit.

Fuel tab

AIVFR

UTC 13:48 | LOCAL 14:48

FUEL

General Fuel Policy

Element	General Fuel Policy	Time Allowed
Trip	Take-off to Landing. Should consider steep climbs, descents and arrival procedures.	
Contingency	Either: • 5% of Trip • 3% if Enroute alternative aerodrome available • Whatever is sufficient for 20min of flight. Whichever higher.	
Alternate	For Missed approaches / landing at alternative aerodromes	
Final Reserve	Enough for 45 min Flight time	
Additional	Enough for 15 minutes holding time	
Extra	At PIC discretion	

The general fuel policy is a guide and does not have to strictly be adhered to.

Fuel Burn:

Litres/hr

Taxi time:

mins

Fuel Required

Taxi Fuel:

Trip Fuel:

Contingency:

Alternate:

Final Reserve:

Additional:

Extra:

TOTAL:

Max Tank Capacity:

No stop-overs are required for this route.

Mass & Balance tab

AIVFR

UTC 13:48 | LOCAL 14:48

MASS & BALANCE

54px

(Table continues)

MTOW:

MLW:

Condition	Weight	Moment	C/G
Basic			
Zero-Fuel			
Ramp			
Takeoff			
Landing			

Manoeuvring Speeds

V_A @ MGW:

MGW:

V_A Takeoff:

V_A Landing:

Any Weight:

Manoeuvring speed (V_A):

Item	Weight	Arm	Moment
Basic Empty Weight			
Oil			
Pilot 1			
Pilot 2			
PAX 1			
PAX 2			
Baggage section 1			
Baggage section 2			

Use your aircrafts weight & balance schedule to check if your CG is in limits.

Not in limits?

Suggest a change

Performance tab

AIVFR

UTC 13:48 | LOCAL 14:48

PERFORMANCE

Take-off Distance Required:

Landing Distance Required:

Take-off Distance Available:

Landing Distance Available:

Due to the critical nature of performance values in flight, the ability to calculate performance values here is not possible. Instead, please consult the official charts to calculate these values yourself.

As previously mentioned, the ability to calculate performance figures is not possible in the webtool and the responsibility is given to the pilot. The boxes above are essentially the results of those calculations- whose purpose is simply to be included in the brief for the user as well.

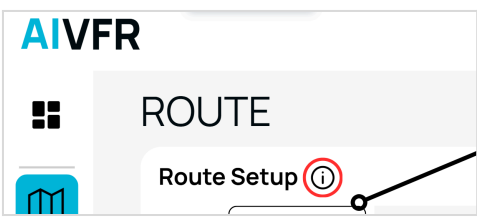
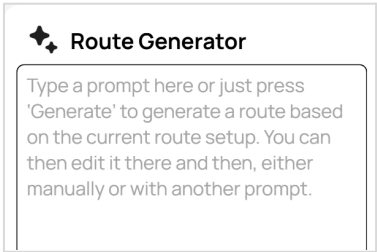
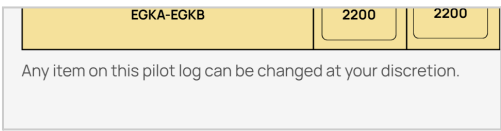
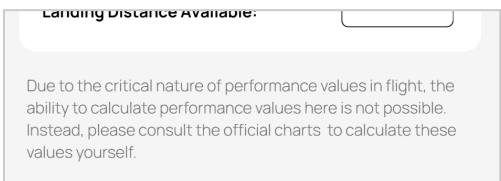
Usability features

Learnability

It's important that the user finds it easy to accomplish basic tasks the first time they use this webtool. Items used to do this include:

Icons: Each of the icons used relate to its purpose in the program, and are accompanied with the actual name of the page/item it is referring to, for absolute clarity. This is important, because *using icons alone that aren't commonly known can lead to confusion to some degree*. However, icons such as the search icon, settings icon, and home icon don't need any explanation as they are used globally across different applications. *On the contrary, simply using words alone without icons diminishes the overall readability and design of the system as the user must read everything on the page to understand what is actually going on and where they might want to go, which is obviously never intuitive in any situation.*

Tooltips: Tooltips are an essential part of my program, especially due to how complex it is. Whilst it is expected that a pilot should know what something is for, when it comes to the actual design of the UI they rely on me to show them how a complex system works, or why they don't. A few examples are shown below.

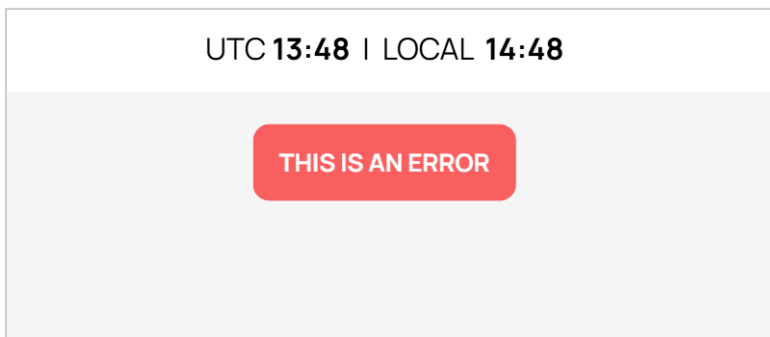
	This is the first time the user interacts with the aerodrome search. As I've used a slightly different way to do this, this info button can be pressed for a brief explanation on how to search for an aerodrome.
	Just leaving this box empty will most definitely confuse the user, no matter how experienced they are, as what to do is not apparent but extremely vague. This helps a lot, telling the user the purpose of the prompt box and how they can use it to their advantage.
	This tooltip reassures the user that values can be changed at any point. Often when things like this aren't customisable it becomes useless to the user, and this ensures it isn't.
	This tooltip explains why a feature doesn't work as expected, or why it's limited. If the user understands why a feature isn't included, they will be much more appreciative than if I just showed a page with limited or no useful features, which will easily frustrate the user.

The absence of these tooltips could make using certain features of the webtool more confusing than it needs to be, as there is no clear guide on how they operate and how the user can interact with them correctly. Clarifying things and complex systems eliminates any potential issues when it comes to their usage and will greatly improve user experience.

Error Tolerance

The ability for users to be able to quickly recover from mistakes is also important in my solution. The main thing that is likely to cause error is when the user is trying to start a new flight without having saved the current one at all. To tolerate this, a **confirmation message** such as *'You have not saved the current flight. Continue without saving?'* Can warn the user to first save their current flight so that they do not lose all their progress. *Without such a warning the user is at risk of accidentally clearing what is stored in the cache at the time and will lose all their flight details that could have taken a while to plan. The user will then get frustrated and will probably not plan another flight with this software as a result.*

Error tolerance wouldn't be needed if the user has no way of even knowing that an error has occurred. To incorporate this, I'll need to include some sort of way of **alerting the user when an error occurs**, through some sort of popup that appears to the user and in its contents will be the nature of the error, as shown in the image below. *Without this there will be no sort of transparency of the system to the user which greatly increases the difficulty of using its features effectively.*



Feedback

Feedback involves the system providing clear responses to user actions. This is more defined at times when the user is expecting a particular output and wants confirmation that something is either happening behind the scenes, or that it has successfully completed. The way I would like to implement this is by **changing the wording of buttons** that will trigger a loading process to say 'Working...' to give the user feedback that their action has been registered and an output is being fetched. An example of this is with updating the weather reports:



At the completion of the loading process, the user may also want to see a visible confirmation that the process has been completed successfully, or if an error has occurred. This is where confirmation messages would come in, which will, in the case of this webtool, simply be a popup at the top of the screen telling the user that such and such details have (or have not) successfully been loaded/fetched/etc.

In some instances, like searching for an airport, just showing the airport name after it's fetched is enough to show the user that the search was successful.

Route Setup ⓘ

From:

→

To:

Route Setup ⓘ

From:

North Weald

→

Whilst I could have the user simply press a button and wait, with no way of seeing what is happening, and no confirmation message either, the system will all of a sudden become less transparent to the user, with the user having to interpret for themselves what is happening and when, or whether something has happened correctly or not. This lack of transparency degrades user experience as it shows that the developer behind it has no consideration of how the user might interact with their system.

Consistency

This involves handling similar tasks and operations in a consistent manner. A few examples of how I will achieve this are the following (not an exhaustive list):

- Navigation menu is always on the left hand side of the screen
- Button colours and sizes are consistent across pages
- The methods for searching for an aerodrome is the same for both the route and weather tab
- The method to input a value is the same throughout
- Colour scheme is consistent throughout
- Same general style throughout

Having repeating features show up consistently throughout the webtool improves user experience by making the user familiar with how a particular system works, then using the same system elsewhere where the user doesn't have to re-learn how to perform another operation of a similar nature. Inconsistent layouts and different designs for similar operations will give the user a much harder time getting to terms with how to use each and every feature the webtool offers. It also makes it more difficult to remember how to use these features again if the user has not used the software for a prolonged period of time.

Algorithms

Algorithm 1: CheckSaved()

The purpose of this algorithm is to check if the user has saved their flight plan, or in other words, has the .json file downloaded onto their device. This is important because once they start to plan a new flight, the cache will be cleared to store this new flight data and any unsaved data left in the cache will be deleted. This algorithm checks that this data is saved first by checking the relevant data in the .json file that holds all the flight data.

```
FUNCTION CheckSaved():
BEGIN
    with open("data.json", "r") as file:
        data = json.load(file)
    IF data["saved"] == False THEN
        RETURN False
    ELSE
        RETURN True
```

```
ENDIF
END
```

Here we check the key “saved” and check if it is true. If not, then the function returns false, triggering a popup to tell the user that they have not yet saved their current flight and that continuing will cause them to lose all their data unless they save it.

Algorithm 2: verifyICAO()

Each airport has a unique code assigned to them by the International Civil Aviation Organisation, or ICAO. For example, London Heathrow airport has the code EGLL, and Los Angeles International has the code KLAX. Since all airports have this unique code, this is the most efficient way to search for an airport and it is what I will be implementing. This algorithm simply validates the input of the user to make sure that it has 4 unique letters with no numbers.

```
FUNCTION verifyICAO(code):
BEGIN
    IF len(code) == 4 THEN
        FOR letter in code:
            IF isdigit(letter) THEN
                RETURN False
            ENDIF
        NEXT code
    ELSE
        RETURN False
    ENDIF
    RETURN True
END
```

If an incorrect input is brought, then the function returns false and a popup will appear to tell the user to input a proper ICAO code with 4 letters and no numbers.

Algorithm 3: fetchAirportName()

After the code is verified to be a good input, we can then use this to search for an airport. The best way to do this is to get its name and display it on the screen. This directly tells the user that the system has identified the input and it can be used in their flight planning process. This process requires using the openAIP API (which holds airport, airspace and other navigational information) to find the airport name that matches the code given by the user.

Luckily, this API has a defined way of calling their API to fetch data, which is shown on their [website](#). I’ve used the same structure in the pseudocode for clarity purposes as there was no other way to display it without including some of the proper code required.

```
FUNCTION fetchAirportName(code):
BEGIN
    IF verifyICAO(code) THEN

        headers = {
```

```

        "x-openaip-api-key": api_key_openaip
    }
    params = {
        "search": str(code)
    }

    url = "https://api.core.openaip.net/api/airports"
    response = requests.get(url, headers=headers, params=params)
    data = response.json()
    airport = data["items"][0]
    Code = airport["ICAOcode"]
    Name = airport["name"]

    IF error occurred OR Code != code THEN
        RETURN False

    ELSEIF Code == code THEN
        RETURN name
    ENDIF
ENDIF
END

```

If a name is successfully found it is returned, and this will be what is presented to the user on the screen to indicate a successful search. If it fails, a popup will again be triggered saying the selected code wasn't found in the database.

Algorithm 4: FetchMetar/TAF()

These algorithms are linked to the weather reports API and simply fetch the latest METAR for a specific aerodrome, also identified by a code. I was very lucky to find this specific API because they also provide decoded versions of the weather reports, as a .json file. This saves a lot of time as all I need to do is parse this data and append it to the flight data file and display it to the user, with no decoding algorithm required.

```

FUNCTION fetchMETAR(code):
BEGIN
    IF verifyICAO(code) THEN
        url = (f"https://api.checkwx.com/metar/{code}/decoded")
        response = request.get(url).json()
        print(response)
        *to be parsed and appended to .json data file*
    ENDIF
END

FUNCTION fetchTAF(code):
BEGIN

```

```

    IF verifyICAO(code) THEN
        url = (f"https://api.checkwx.com/taf/{code}/decoded")
        response = request.get(url).json()
        print(response)
        *to be parsed and appended to .json data file*
    ENDIF
END

```

Algorithm 5: updateJSON()

At several occasions data might need to be updated in the large data file, for example, if a setting is changed, or a new route is generated, or the weather reports have been updated. This simple algorithm allows me to do this by just calling the function and inputting the key and values that I want to update. New values will be added manually via the file itself to maintain a defined structure for each and every single flight data file by the end of the development process.

```

FUNCTION updateJSON(key, value):
BEGIN
    with open("data.json", "r") as file:
        data = json.load(file)
        data[key] = value
    with open("data.json", "w") as file:
        json.dump(data, file, indent=4)
END

```

Algorithm 6: fetchRole()

When it comes to the AI part of this project, the user must prompt the AI to generate a particular piece of data. The API allows users to give the AI a role, along with the actual prompt. This is great because I can tailor a specific model to manage briefs, and another for the navigation log, and so on. It also means I can tailor exactly how I want them to output their response, which is important when it comes to reading these outputs in HTML, for example. The purpose of this algorithm is to fetch the specific role for the AI to follow.

```

FUNCTION fetchRole(type):
BEGIN
    IF type == "Route" THEN
        role = (
            "Your role is to generate the best possible route.
            Display in ICAO format.
            Current flight data is as follows: {input relevant data from .json
            file}.
            Justify your response.
            Your output should strictly have the following layout: LAYOUT"

```



```

    )
    RETURN role

ELSEIF type == "Brief" THEN
    role = (
        "Your role is to generate a complete flight brief.
        Current flight data is as follows: {input relevant data from .json
file}.
        All Flight data should be displayed in your response.
        consider Threat Error and Management in your response.
        Your output should strictly have the following layout: LAYOUT"
    )
    RETURN role

#and so on...
ENDIF
END

```

The basic representation of what it would look like is what is shown above, with each role dependent on the type of AI being used. Each role (e.g. route, navigation, brief, etc) will have its very own full complete prompt and this function fetches the correct role prompt to be used in the actual user prompting algorithm, shown below.

Algorithm 7: prompt()

This algorithm serves to prompt the AI to output a response, with the parameter 'type' being the type of AI they're asking from (such as the route generator) and the text being the actual user prompt. Sometimes there will not be a particular prompt from the user- for example when autofilling the navigation log, in which case a custom user prompt written by me will be used.

```

FUNCTION prompt(type, userText):
BEGIN
    model = "gpt-3.5-turbo",
    instructions = [
        "role": fetchPrompt(type),
        "user's prompt": userText
    ]
    RETURN response
END

```

The Issue with fetching NOTAM reports

One of the main algorithms that has not been shown in this section is the fetching and decoding algorithms for Notices To Airmen (NOTAM) reports, which essentially are just warnings in place that airports publish that could potentially affect a flight.

During the feasibility study of this project, whilst I confirmed that the weather and airport data APIs existed and were operative, I never found an API that I could use to fetch NOTAMs. Nevertheless, I decided to include the feature, and have the user paste in a NOTAM and the plan was to decode them and display them on the dashboard, which is what is shown in the UI design. However whilst formulating the pseudocode for this decoding algorithm, I ran into a serious issue that meant I had no choice but to discontinue the feature of showing NOTAM reports to the user.

To keep it brief, NOTAMs, like a METAR or TAF, have a defined structure. This should in theory make it easy to decode, however the line-by-line format of a NOTAM differs greatly depending on who is providing it. This meant that I couldn't code a set way to decode every sort of NOTAM that the user could paste into the webtool. This wouldn't have been a problem if there was a set API that I could get them from without strict licensing and such.

If I decided to continue this feature, It would have the user pasting line-by-line the NOTAM in the format I would like them to, before I can decode it. Keep in mind that the sources that they get these NOTAMs from all provide decoded versions already, and they're just pasting in the raw version. This was enough for me to discontinue this feature, because there was no way of including this feature in a way that would be beneficial to the user, Especially when they can get neatly decoded versions of these reports from other sources.

Key Variables & Data structures

Data (15)	Data Type	Explanation	Justification
flight_data["saved"]	Boolean	This boolean value will indicate to the system whether the flight is saved or not, and will change whenever the user successfully saves the flight data file onto their device. It is a key found in the flight data file.	This is important in running the checkSave() algorithm. Without this variable, there would be no way for the system to check if user data has been saved and more often than not, user data will be lost after several minutes of planning.
verifyICAO()	Function	This function verifies the ICAO code inputted into search bars to allow an aerodrome search to occur. It prevents bad inputs from being sent to the API.	This layer of verification makes the program much faster, because it's quicker for the program to run this code to verify the contents of a string, than to run the API every time and have that indicate a bad input instead.
code	String	This is the string allocated to airport codes. Most functions that utilise airport reports or data will utilise this string to identify which airport the user is referring to.	This is key for the operation of any function involving an airport, because the function must know what airport to fetch data for before it can work, therefore without this, many essential features become inoperative.
prompt()	Function	This function prompts the AI to create a response for the user. It is accompanied with the role of	This works in conjunction with the fetchRole() algorithm and specifically accounts for the user's

		the AI to allow it to produce a response.	input, and is also responsible for producing a response. Without this feature the user has no way of interacting with any AI features of this webtool.
type	String	This variable refers to the type of AI being requested, as each AI will have to have a different role to facilitate the user's needs. Examples include "brief", "route", and so on.	This was an important addition because it would be very difficult to use just one model with one role to perform all AI actions across the program. Having different types of models with different roles meant that each LLM can be tailored and become 'experts' in their roles, allowing for more reliable responses.
userText	String	This is used as a parameter in the prompt() function and will contain the user's custom input to the LLM.	Without this the user again has no way of accessing the LLM and cannot personalise their route/brief/navigation log.
model	String	This is a requirement by the ChatGPT API that I'm using to provide AI features to the webtool. In this case, I will be using 'gpt-3.5-turbo' and this is what this variable will be assigned.	Without specifying the model that I want to use, the API will not operate and no response output.
fetchRole()	Function	This function fetches the relevant text that describes the role of the type of AI the user wants to access.	This function is necessary to hold all the role information for each AI. Without it, I would have to find a way of creating a role from scratch every time the user wishes to generate a response from the AI, which is extremely difficult.
role	String	This is what is returned from the fetchRole() algorithm and holds the role of a specific AI. This is passed onto the prompt() algorithm to be used in generating a response.	Necessary in order for the webtool's AI features to be accessible to the user.
fetchAirportName()	Function	This function is responsible for fetching the airport name from the code given as a parameter. It will use the airport data API (OpenAIP) to search for the airport that matches the code and return its name.	This is important in providing feedback to the user that a search was successful. By showing the name of the airport the user has searched, it shows that the code they inputted was correct. This is more efficient than coding up an entire popup message just to tell the user the same thing, because

			it also gives the user details about the airport they've searched.
Code	String	This is linked to the airport ICAO code for the airport fetched from the airport data API. This is stored and used to display the code and names of airports that are searched as alternate airports.	Because a route can have more than one airport, the code cannot remain in the search box and the user needs that input space. This will fetch that code the user put in and it will be displayed next to the name instead. Without this there would be no way of searching for more than one alternate aerodrome.
name	String	This is linked to the airport ICAO code for the airport fetched from the airport data API. This is stored and used to display the names of airports that are searched.	This is what is returned from the fetchAirportName() algorithm and is used to confirm that a search was completed successfully. Without it this form of user feedback would not be achievable.
fetchMETAR()	Function	Used to fetch a Meteorological Aerodrome Report from the aerodrome searched by the user. (taken as the code parameter)	This is responsible for one of the essential features of this project which is weather reports. The absence of this function would have the user go to other sources to find these reports which is obviously not my intention.
fetchTAF()	Function	Used to fetch a Terminal Aerodrome Forecast from the aerodrome searched by the user. (taken as the code parameter)	This is responsible for one of the essential features of this project which is weather reports. The absence of this function would have the user go to other sources to find these reports which is obviously not my intention.
flight_data[key]	Variable	When searching for a value from the flight data file, as it is a .json file it will be called in this format. I've added this here because it plays a key role in allowing several features access to the current flight data.	Without this variable there would be no way for different features (especially the dashboard, which relies solely on data from the data file) to access current flight data. It is also needed to allow data to be updated accordingly when required.

Justification of how the algorithms form a complete solution

Application Walkthrough

The best way to justify how these algorithms will form a complete solution is to walk through the expected flight planning process the user will take when using the webtool, and explain where each of the main algorithms come into place and how they apply in their specific situations. I will also take this opportunity to outline exactly how the complete solution should turn out to be, and this is what will be followed to be achieved when in development.

So, to start, the user enters the webtool and is shown the main menu. Let's say, for the sake of argument, the user has used this webtool before and wants to plan a new flight. This is where **Algorithm 1: Checksave()** comes into play, checking and prompting the user (if required) to save their previous flight onto their device before continuing. The same goes with if the user selects 'load flight' with items still in the cache.

The user will then be presented with an empty dashboard, with a tooltip prompting the user to start planning to get detailed information about their flight on the dashboard, reading this information, the user would go to the route tab, and begin to plan their flight, first by entering the departure, arrival and alternate aerodromes of their choice. **Algorithm 2: verifyICAO()** verifies the airport code inputted into the relevant boxes, and if they pass the verification, then **Algorithm 3: fetchAirportName()** searches the API to find the aerodrome the user is looking for. If successful, the name of the aerodrome will appear beside the box to indicate a successful search, and the flight data file is updated accordingly using **Algorithm 5: updateJSON()**.

At this point, the user could generate a route directly by pressing the 'Generate' button, or they could tailor their route by inputting a prompt first. In either case, both **Algorithm 6: fetchRole()** and **Algorithm 7: prompt()** will be used here to generate the appropriate response. This route will be reflected on the map where they can observe if their route crosses any particular airspace or something similar. Again, the flight data file is updated with all this new information and is stored in the cache.

After planning their route they should move onto the weather section, where they can view the latest reports for their specific route or any airport of their choice. At first all boxes will not have any data until the user first presses the 'update' button, which will trigger **Algorithm 4: fetchMetar()** or **fetchTaf()** depending on what the user has selected to generate. All reports will then be updated and displayed to the user. **Algorithm 5: updateJSON()** once again will add this new weather report data to the flight data file to allow its use in the flight brief.

The user continues to the navigation log tab. At this point the only values that should be filled are the distance and the track values, as they are stored in the route already. If the user decides to do this manually, the user will enter the relevant data such as true airspeed (TAS), the spotwinds forecasted, and the largest Maximum Elevation figure along their route, which is found on their aeronautical chart. From this, all other values will fill in as they rely on calculations from the wind. The user however must select their own planned altitude.

The alternative to filling out the log is to use the 'autofill' button on the page. The prompt will read the current flight data stored in the data file and make a (justified) judgement on the winds, MSA and the planned altitude. **Algorithm 6: fetchRole()** and **Algorithm 7:**

prompt() will be used again here to generate this. Moving onto the final 3: fuel tab, mass & balance, and flight performance. No AI is used here as these are critical calculations that I do not believe my current AI model will be able to produce suitable results for, considering every circumstance.

Finally, in the dashboard, all the flight details should have been updated for the user to see. **Algorithm 4: fetchMetar()** will be used to fetch the current METARs, and **Algorithm 6: fetchRole()** and **Algorithm 7: prompt()** will once again generate the brief for the user. At this point, the user saves the flight and exits the webpage, at this point the current flight data will still be cached if a new flight is yet to be created.

Testing

Iterative test data

Test description (11)	Test data	Justification of test data
Initial Flask web page testing	"[Page name]", e.g. "Route Tab"	To ensure that each web page loads correctly and displays HTML content, this string will be used. It is important that the string is different for each page, as having the same string, such as "Hello World" will be identical for each page and distinguishing between pages will be difficult.
The first page that loads up after the main menu is the dashboard.	Main menu input by pressing either of the buttons	The dashboard showing up first is key in prompting the user to come back to it after planning their flight because they will see that it is empty and the tooltip will tell them to plan a flight for information to show. Users may forget to look at the dashboard and save their flight if this feature was not available.
Disclaimer pops up <u>before</u> opening dashboard, and there is no available way to bypass it.	Input via menu buttons	This test is important in keeping my solution safe to use. If the disclaimer fails to show up and the user has not agreed to any terms then it puts me in a bad situation if any results from this is used in real flight and a problem occurs.
Buttons are operative. If colour is to change they do, and when clicked they trigger the correct output.	Mouse hover Mouse input (click)	All buttons must be working properly for most functions of the webtool to work. With inoperative buttons the user won't be able to use the menu, access different tabs, the settings menu, and several other features that are essential to the webtool.
Base template inheritance testing: Once the base template is made, other pages that must inherit from it should do so with no discrepancies between pages.	Dashboard and planning module tab buttons	To ensure consistency throughout the webtool, pages inherited from a base template must have the same throughout. Any minor change in detail as a result of not testing this forms a major design flaw and may not look appealing to the user.

User should be able to switch between tabs seamlessly and correct hover effect to be shown	Mouse inputs	As shown in the UI design, the user should be able to hover over the image for a specific tab and its name be shown adjacent to it. Then, if the user clicks on it, then they should be taken to that specific tab. The absence of this would not allow the user to see or switch between tabs to access other parts of the webtool other than the dashboard.
Data stored in the .json flight data file should be able to be saved for future use	The flight data file (flight_data.json), A test variable (test_var: "testing")	This is probably one of the most important tests to be conducted, as this ensures that the user does not lose all their data simply by exiting the webpage. The test data for this is simply a test string because no real data will be used until after a successful test.
AI should output suitable responses in accordance to their given role and user prompt	Multiple mockup user prompts Multiple mockup roles	Many of the AI responses must be written in a specific way and this testing will consist of trial-and-error to generate a response in the appropriate format. Therefore, several mockups of user prompts and roles must be used here, so that once the tests are successful the real flight data can be made into a user prompt much more quickly. If I just used one user prompt and adjusted the role only or did vice versa, this would greatly limit what can be outputted by the AI and obtaining a successful output will take significantly longer.
APIs are operative and display data in the correct way on the relevant page	API function returns, weather tab, route tab, dashboard	The main tests for this will be the Weather and Route tabs as these utilize the different APIs the most, with a lot of data from the API needed to be displayed in very particular ways. This allows for me to test formatting and output in ways I otherwise wouldn't be able to should I have chosen other test data.
Light/dark Mode switches correctly with the correct colours	All screens	It's important that every single screen is checked in dark mode before moving onto the next. This is to ensure that the colours have been selected correctly to present the correct opposite for dark mode when it is selected. If I choose to only rely on one page to check if it's correct, other pages could look much different and several design errors would be present across the webtool. Moreover, if I wait until post-development to do this, the initiative to fix all the colour errors will likely be very little as I would have to make edits across every screen.
Dashboard displays correct data and updates accordingly	flight_data.json	The .json data file is what will be changed to check if the dashboard updates correctly. It's also the source file that it will read to display this data, and so it is the only suitable test data for this test and nothing else is available as an alternative.

Post development test data

Test description (12)	Test Data	Justification of test data
Webpage loads	Running Flask	Flask will be the core tool behind the webtool. It is through running this that the HTML pages will run, and so this should be run first. Running each HTML page may not give me the ability to navigate between pages, which would not be ideal.
Settings changes work, appropriate variable in data file changes	Settings buttons	The settings buttons (and only these buttons) should trigger the relevant variable to change in the flight data file, to change how something is displayed, such as units of a measurement. In this case this is the only suitable test data for this.
Disclaimer operates as expected	Continue/New/Load flight buttons, disclaimer buttons	The disclaimer should act like a barrier between the planning module and the main menu, by which the terms must be accepted in order to pass through. It is important as well that the disclaimer itself is tested, so that if the user presses 'Go Back', the system does exactly that and does not progress to the planning module. Focusing on just the main menu buttons would pose a danger if I hadn't considered the buttons in the message itself.
Dashboard is empty on first look	New flight button	On a new flight the dashboard has no data from the flight data file to be shown and so nothing should be shown, prompting the user to move on and plan the flight. The 'New Flight' button must be used here instead of just adjusting the flight data file itself because at this stage we are in post-development and no internal code should therefore be edited as test data.
User can navigate between pages	Cursor input	As the user hovers over tab icons, tab names should be shown and the user should be able to switch between them seamlessly. Such a test that incorporates mouse-hover mechanics cannot be conducted without a mouse cursor.
User can plan a route using the Route Tab (either with AI or Manually)	Three test routes: EGSX - EGMC EGSX - EGX EGBM - EGSC	<i>(Only airport codes written here for simplicity but full routes will be very clear to understand in the actual post-development testing section.)</i> Test routes are the best way to check if the Route planning system and algorithms work properly. Each route has something different about it that should bring up different routes and tests new difficulties that the AI will have to work through. If I chose to test only one route, I will have no idea what other more complex/less complex routes will look like and this could be detrimental to the functionality of the webtool. Therefore it is

		important that several different routes are tested.
Weather displays and updates correctly	Test route inputs, and 3 random airports of my choice (that don't form any routes from the above). At least one must not provide either a METAR or TAF report.	Since the route tab includes the weather reports for the departure and arrival aerodromes, the aerodromes inputted as test inputs will facilitate this. For the aerodrome search, 3 different airports will be used. At least one must not provide either a METAR/TAF or both, to test the output of 'No METAR/TAF Found'. Such an input is the only way to test this feature live. The aerodrome search airports must not form part of the test routes to allow me to see how fresh reports look like. If I chose those from the test routes it would just be the exact same report as the departure/arrival airports which is not useful for testing purposes.
Navigation log can be edited either by AI or Manually	User input 'AUTOFILL' button	User input here will facilitate the manual addition of data into the navigation log, whilst the AUTOFILL button triggers the AI to do the same. No alternative methods should be available to edit the navigation log other than these two methods.
Fuel can be added with correct calculations for all fuel policies	User Input (int values only)	Since we are dealing with fuel quantities, only integer values can be used here so that calculations actually work. The testing here is for a complete successful fuel plan and not for correct calculations because that will be checked already in the development of the fuel planning section itself.
Correct mass and balance calculations	User input (int values only)	The same applies in this case as well. A successful test will see the different values for CG available to the user for each stage of flight, allowing them to compare with their charts to see if it is within limits. Integer values again here to ensure that the calculations can be made.
Dashboard shows all the flight details in a clear format and a brief generated on command	Flight_data.json (reading only) Generate brief button	The test data from the current flight plan being tested will be the input for the dashboard, as it should only be able to read from this file. The Generate brief button is required to trigger the AI generating algorithm to generate the best suitable brief for the user.
Saving a flight to the device is possible	'Save flight' button input flight_data.json	The data being saved is the json file, and the save flight button is the only suitable input to trigger this.

A successful post-development test in the case of this project would ideally be a test of all these features **3 different times**, using the 3 different routes outlined in the route tab tests. If all 3 plans can be made without any significant errors or problems, then I as a developer can happily take my webtool as a reliable and robust system, and deem the project as successful.

DEVELOPMENT

This chapter was documented in real time rather than after the entire development process.

Section 1: Initial Flask Setup

Aims of this section

- Code the basic flask template
- Create and test all webpages

Collecting relevant API keys

Three API keys will be required for this project. They are for:

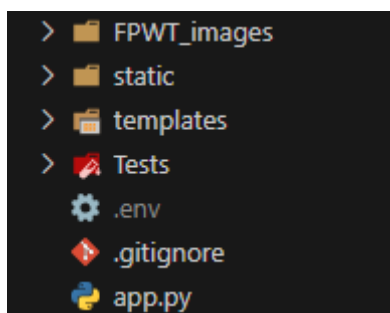
1. **OpenAI** - for the ChatGPT LLM
2. **OpenAIP** - (Stands for Open Aeronautical Information Publication). For the airport data
3. **CheckWX** - CheckWeather API, for the weather reports.

Once I created the API keys from their respective websites I stored them in a **.env** file. This allowed me to call the python dotenv function in my python code to call the variables storing the keys, and not have to paste in the keys into the actual code.

It meant that I could ignore the **.env** file using **.gitignore**, because when you commit on github it detects API keys and refuses to commit because it won't save sensitive data onto its system.

File management

Flask requires specific file names for certain folders, so when creating necessary files I have named them accordingly:



Tests contains any test pages that I may create to test something without interfering with my actual project- a common thing in most development projects. This will of course be deleted at the end of development.

FlightPlanningWebtool_images holds all the image icon files used in the UI. I saved them one by one as I searched and implemented them in the design, so by the time I had finished the screen designs I had all the icons I was going to use in one file, and imported it straight into VSCode.

Static (Flask requirement) Contains any CSS and JavaScript files to be used.

Templates (Flask requirement) Contains any HTML files to be used.

Finally, **app.py** contains the main python flask algorithms.

This is the initial setup, and things are subject to change when more files/features are added throughout the development process.

Iteration 1 - Setting up and testing the basic Flask template

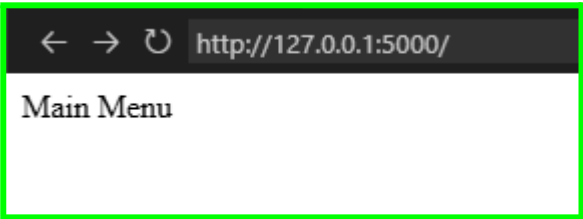
```
AIVFR > _init_.py > create_app
1  from flask import Flask
2
3  def create_app():
4      app = Flask(__name__)
5
6      from .app import main as app_blueprint
7      app.register_blueprint(app_blueprint)
8
9
10     return app
```

```
AIVFR > app.py > index
1  from flask import Blueprint, render_template
2
3  main = Blueprint('app', __name__)
4
5  @main.route('/') #this will be the first to load up
6  def index():
7      return 'Main Menu'
```

At this point, I was just following a basic python flask setup tutorial to get it up and running. Thus no errors in the code were present.

On a basic level, when the program first runs it will run whatever is in `@main.route('/')` first, and is why the main menu will be on this page. We can now test this to see if it successfully returns 'Main Menu'.

`__init__.py` was created to hold the `create_app()` function that is a specific function (also must be named this way) that flask will run to initialise the program.

Test No.	Test Description	Expected Outcome	Actual Outcome
1	Initial Flask web page testing	Test string displays ('Main Menu')	

Now that we know that flask actually works as expected, we can continue to create all the other pages like so:

Iteration 2 - Creating and testing all the pages

```
#settings menu
@main.route('/settings')
def settingsMenu():
    return 'settings Menu'

#dashboard
@main.route('/dashboard')
def dashboard():
    return 'dashboard'

#Route tab
@main.route('/route')
def routeTab():
    return 'Route Tab'

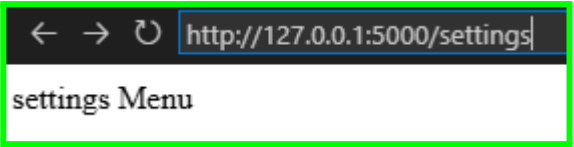
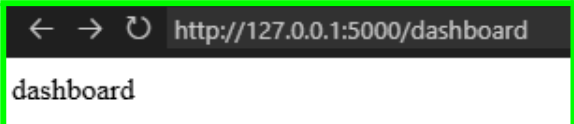
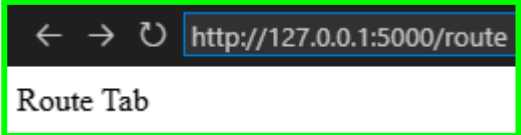
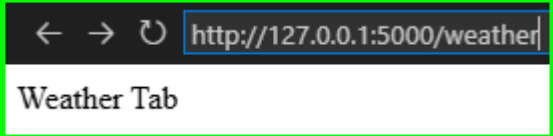
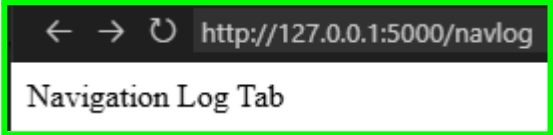
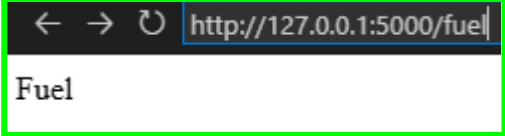
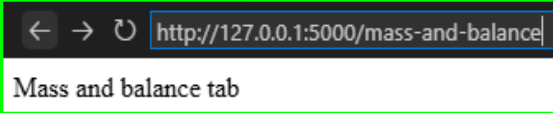
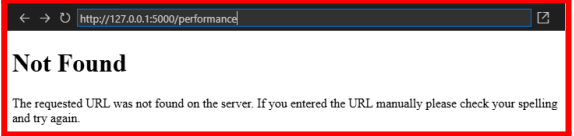
#weather tab
@main.route('/weather')
def weatherTab():
    return 'Weather Tab'

#navigation log tab
@main.route('/navlog')
def navlogTab():
    return 'Navigation Log Tab'

#fuel tab
@main.route('/fuel')
def fuelTab():
    return 'Fuel'

#mass and balance tab
@main.route('/mass-and-balance')
def massAndBalanceTab():
    return 'Mass and balance tab'

#performance tab
@main.route('/peformance')
def performanceTab():
    return 'performance tab'
```

Test No.	Test Description	Expected Outcome	Actual Outcome
2	Initial Flask web page testing	All other pages load as expected	       

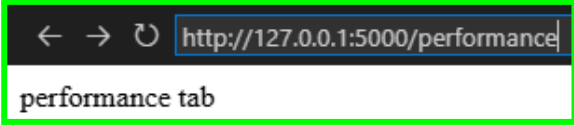
Error Analysis & Fixes

Test No: 2

Failure Description: URL Not found error

Cause: spelling mistake: 'peformance' instead of 'performance'

Fix (immediate) : corrected spelling mistake

<pre>#performance tab @main.route('/performance') def performanceTab(): return 'performance tab'</pre>	
--	--

Justification: syntax error is the source and so the fix must involve a syntax edit. (there are no alternatives). This was a simple fix and another full documented iteration to show this fix would therefore be unnecessary. This will be the main reason for any 'immediate' fixes to occur anywhere else in the development process.

Section Review

Aims completed in this section:

- Coded the basic flask template
- Created and tested all webpages

Aims not completed in this section:

N/A

Tests complete:

- Initial Flask web page testing

Remarks: Nothing to address.

MENU AND CONFIGURATION MODULE

Section 2: Main Menu

Aims of this section

- Implement the logo and appropriate colouring & styling of the page
- Make the three main menu buttons with appropriate hover effects
- Implement the first essential feature: **disclaimer**
- Test the buttons such that the settings and continue/new/load flight buttons do what they're supposed to.

All buttons will initially be coded to popup the disclaimer when they're pressed. However, only the 'Continue' button will take you to the dashboard as the load/new flight buttons require action involving the cache (for the most part) which is another section in this development process.

If I decided to still attempt to code them now, I will spend unnecessary time trying to test the cache and working with the .json data file and this section will take significantly longer than it needs to be.

Iteration 1 - Fetching Font and displaying main attributes

```
<!DOCTYPE html>
<html lang='en'>
  <head>
    <!--Getting the Manrope font from google fonts-->
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Manrope:wght@200..800&display=swap" rel="stylesheet">

    <meta charset='UTF-8' name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>AIVFR - Home</title>
    <link rel="stylesheet" href="../static/main-menu.css"/>
  </head>

  <!--logo-->
  <div>AIVFR</div>
  <div>AI ASSISTED FLIGHT PLANNING</div>
  <div>Alpha</div>

  <!--Buttons-->
  <div><button>Continue</button></div>
  <div><button>New Flight</button></div>
  <div><button>Load Flight</button></div>

   <!--settings icon-->

  <body>
</html>
```

What I did

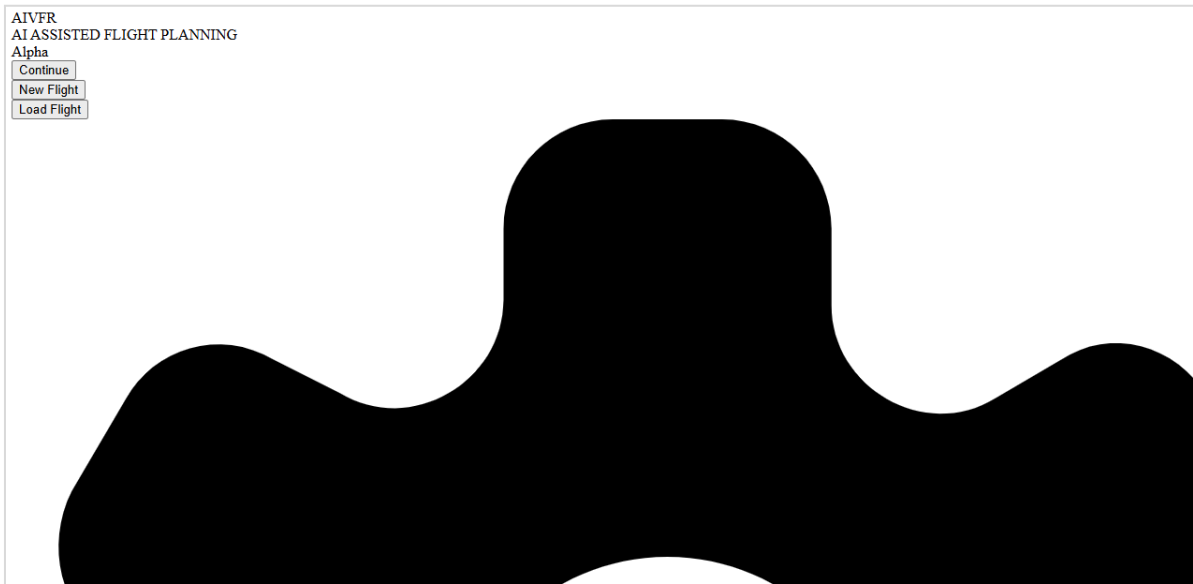
It was best to start with the essentials and only after add the styling and details to make it look like the final product back in the UI design. First, I coded everything I'll need and just put it on the page.

Why I did it this way

If I continued to work with the styling and other aspects of the code, not much of the iterative development of this project would have been shown. This will be the case for many other iterations so this reasoning will only be explained once, here.

Being specific, Having everything laid out first allows me to simply test the HTML page, see if the image loads, and there are no issues with viewing the webpage. This saves time because I can combat any errors that occur there and then, rather than figuring out if it has something to do with the styling, or not.

Program output (Not the entire viewport - too large)



Known Limitations / Missing features

- There is no styling whatsoever in this iteration and that is expected, and this includes the background, settings image, the logo elements and the buttons.
- Font has not been actually applied to the text, just fetched.
- No buttons work in this iteration.
- Disclaimer popup

Test No.	Test Description	Expected Outcome	Actual Outcome
3	HTML output testing	All elements display as expected	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Styling background and logo

Iteration 2 - Styling background and logo

menu.html

```
<link rel="stylesheet" href="../static/base.css"/>
</head>

<!--logo-->
<span id="logoAI">AI</span> <!--having these separated allows me to style them differently-->
<span id="logoVFR">VFR</span>
<span id="logoAlpha">Alpha</span>
<div>AI ASSISTED FLIGHT PLANNING</div>
```

Base.css

```
:root { /*these colours will be used by default...*/
--bg: hsl(0 0% 96.1%);
--primary: hsl(190 100% 42.5%);
--highlight: hsl(0 0% 100%);
--text: hsl(0 0% 0%);
--onhover: hsl(0 0% 80.4%);
--popupbg: hsla(0, 0%, 0%, 0.33);
}

/*unless dark mode is toggled, then these become the root
and are used instead*/
.dark-mode {
--bg: hsl(0 0% 12.5%);
--primary: hsl(190 100% 27.6%);
--highlight: hsl(0 0% 20.8%);
--text: hsl(0 0% 100%);
--onhover: hsl(0 0% 32.9%);
--popupbg: hsla(0, 0%, 0%, 0.33);
}

body {
background-color: var(--bg);
font-family: 'Manrope';
padding-left: 30px;
padding-top: 10px;
}

.text{
font-family: 'Manrope';
}
```

main-menu.css

```
#logoAI, #logoVFR {
  font-size: 122px;
  font-weight: 650;
}

#logoAI {
  color: var(--primary)
}

#logoAlpha {
  border-width: var(--text) 100px;
}
```

What I did

The first thing I did was to create a CSS file called **base.css** because there will be several pages that will have some basic styling features such as the background colour, font, text colour, and so on. These form the 'base' of many of the pages and so were included here as one single file.

Base.css contains the colours I decided to use and the background and font styling, as well as padding, as these are what will be reoccurring in most of my pages. However I may reconsider including padding in base.css because within the flight planning module the padding will be different due to the navigation elements on the sides of the viewport.

The colours use a simple mechanism utilising the `:root` feature in CSS. It's done in the way that when dark mode is toggled on, all the variables in **.dark-mode** will be used instead of those in light mode. Since they both have the same variable names for each colour, the colours will immediately be replaced.

The final change was to styling the logo. As per my design, the logo reads 'AIVFR' with the 'AI' part of it being blue. I quickly ran into an issue because if I wanted to style them separately they couldn't just be one `<div>` element. The obvious solution was to make them `<span>` elements instead, which made them line up with each other but could still receive separate id's, which meant I could style them appropriately as seen in **main-menu.css**.

Why I did it this way

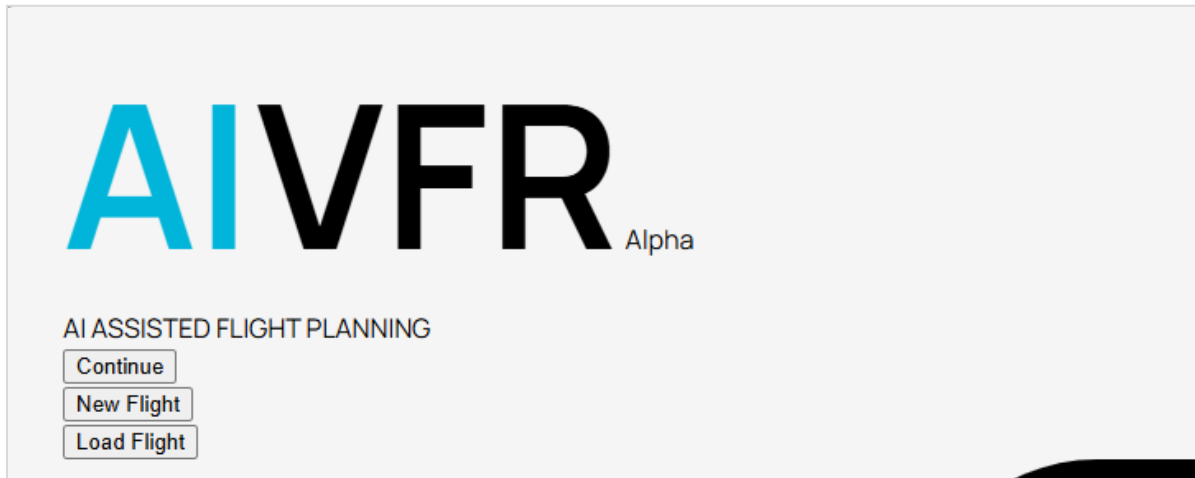
Base.css is an essential addition because it allows for more efficient code. Almost every page will inherit the styling from base.css meaning I don't have to write the same styling code for every single page, which is unnecessary in every regard.

Keeping the main padding in base.css would mean that all the other pages would inherit the same padding and would have to be changed as per the reason I gave above. This change will only have to be made once as there will be a template for pages in the planning module too, but in attempt to make my code as efficient as possible it is better to take something that doesn't fit the use case of the file that it is in (being present in all other files, in the case of **base.css**) out and implement it separately.

The root feature is the only feasible way of switching between colour schemes. Of course, having to set all the colours separately in each file (or base template) would make it impossible to switch because of variable naming, different file locations, and so on. Finally, the decision to change the element used in the logo was an essential requirement to allow

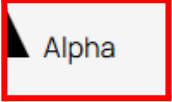
the successful styling of the logo. Without it the initial design would have to be abandoned as there was no other way of having them separated but also aligned as desired.

Program output



Known Limitations / Missing features

- Buttons and settings icon yet to be styled.
- No buttons work in this iteration.
- Disclaimer popup

Test No.	Test Description	Expected Outcome	Actual Outcome
4	Background colour test	Background is the appropriate colour: <code>--bg: hsl(0 0% 96.1%);</code>	
5	'AI VFR' logo colouring test	'AI' is blue (primary colour) and 'VFR' is black (normal text colour)	
6	Subtitle formatting testing	Subtitle of the logo appears as per UI design: 	
7	'Alpha' formatting testing (Alpha is the stage of this projects development)	Should appear in a curved border as per design: 	

Error Analysis & Fixes

Test No: 6

Failure Description: Words spaced too far from logo, also not bold enough

Cause: Currently unknown, most likely an issue with the element size and spacing. The boldness issue is due to a wrong **font-weight** value.

Fix (proposed): Test other elements and play with the positioning of the subtitle. I will attempt to make it relative/absolute and see if I can move it upwards. The boldness can be adjusted via adjusting the font weight.

Justification: 'relative' positioning allows the item to move relative to itself. That way I may be able to move it upwards manually. Any alternative element won't be able to do this as efficiently in my knowledge. 'Absolute' might have the same effect but is less likely.

Test No: 7

Failure Description: no border around alpha, and font a little too thick.

Cause: need to encase it in a div before creating a border, or the code I wrote is just insufficient. The boldness issue is due to a wrong **font-weight** value.

Fix (proposed): Research how to create a proper border around text and adjust the font weight value to make it less bold.

Justification: It's best to research something that I'm not familiar with than to simply try several methods without avail. Of course, there is an extent to trial-and-error that can be applied when developing but at this point I'm completely unsure and searching for the solution will save me time constantly brute-forcing a solution.

Iteration Review

Met requirements: N/A

Changes: Padding will be changed to be in **main-menu.css** instead of in **base.css**.

Next steps: Correcting the outstanding 2 errors to come up with the full AIVFR logo as per design.

Iteration 3 - Styling logo completion

Main-menu.css

```
#logoAlpha {
  color: var(--text);
  position: relative;
  bottom: 10px;
  left: 10px;
  font-weight: 100;
  padding: 6px;
  padding-right: 10px;
  padding-left: 10px;
  border: 0.1px solid var(--text);
  border-radius: 14px;
  border-top-color: var(--bg) 100px;
  border-spacing: 10px;
}

#subtitle {
  position: relative;
  color: var(--text);
  bottom: 25px;
  font-weight: 700;
}

body {
  padding-left: 30px;
  padding-top: 10px;
}
```

menu.html

```
<div id="subtitle">AI ASSISTED FLIGHT PLANNING</div>
```

What I did

After some research, the problem with the styling of the 'Alpha' became very apparent. It seemed that I had not given it a **border style**, (**solid**, in this case), so nothing showed up. The line **border: 0.1px solid var(--text)** is what initiates the border styling, and so this fixes the problem. After discovering this, I continued to play around with the border sizes until I got what looked very similar to what was in the design

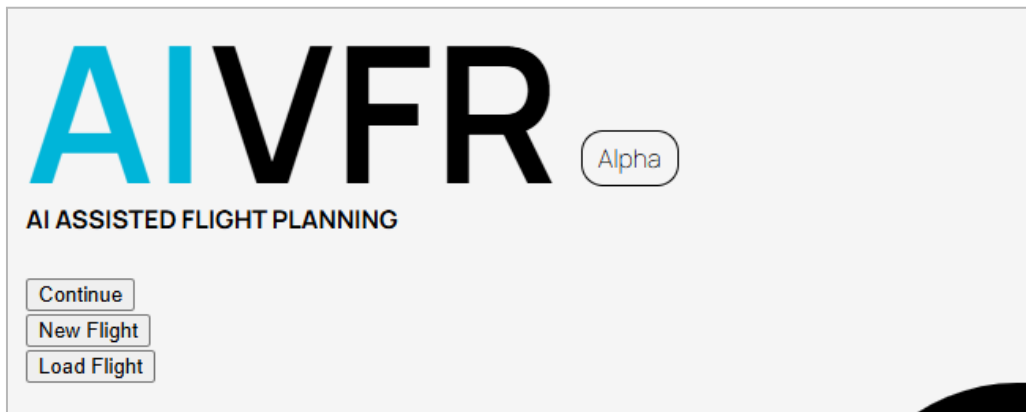
Then it got to moving around. It turned out I indeed needed to make its position 'relative', so that it can move relative to itself. Essentially, this attribute makes any object movable relative to itself. I used this to move that same part of the logo to match the design, and also moved the logo's subtitle using the same logic.

Finally, I changed the **font-weight** values of both of these elements as appropriate, so that they also matched the design in terms of thickness.

Why I did it this way

Another positioning alternative I could have used was 'absolute', which in my case aligns an object relative to the body. In my knowledge, Using this would require more movement from the end of the screen to its position, and may not be as suitable because the position may change when the viewport also changes.

Program output



Known Limitations / Missing features

- Buttons and settings icon yet to be styled.
- No buttons work in this iteration.
- Disclaimer popup

Test No.	Test Description	Expected Outcome	Actual Outcome
8	Subtitle formatting testing	Subtitle of the logo appears as per UI design: 	
9	'Alpha' formatting testing (Alpha is the stage of this projects development - very very early stages)	Should appear in a curved border as per design: 	

Iteration Review

Met requirements: (Aim complete) Implement the logo and appropriate colouring & styling of the page

Changes: N/A

Next steps: Settings button styling

Iteration 4 - Settings button styling

```
#settings {  
  position: absolute;  
  width: 50px;  
  height: 50px;  
  top: 30px;  
  right: 30px;  
}
```

What I did

To style the settings icon size, all that was needed was to change its width and height values appropriately. I deviated from the design a little in regards to size, because 60px x 60px was a little large and didn't look good in my opinion.

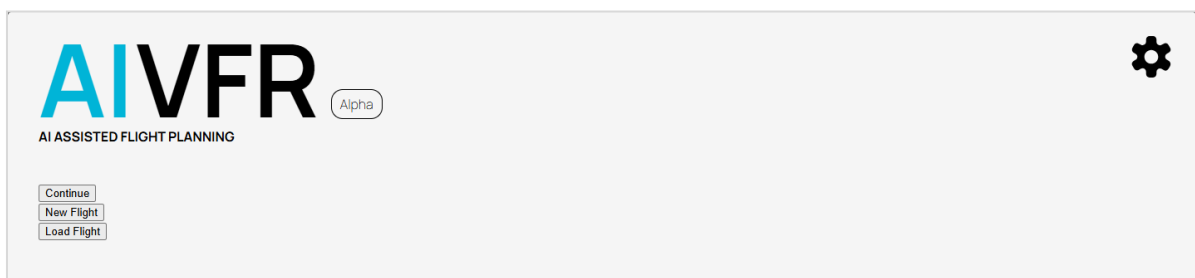
The positioning of the settings menu was different to normal, as I used `position: absolute`, which meant that the positioning of the icon would be relative to the viewport in this case. This was what I needed because the settings menu needed to be able to move with the viewport. I will elaborate this further.

After that, I realised that the logo didn't align with the positioning of the settings menu, and so I made a few adjustments to the **top/bottom** attributes of the logo parts to match the top of the settings logo. This way they were both aligned as per design and looked neater.

Why I did it this way

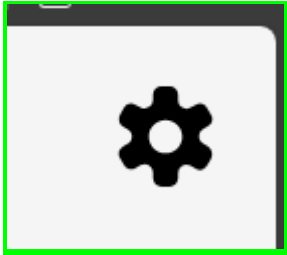
Using position absolute instead of relative was because I needed the settings icon to move with the viewport and maintain its distance from it as it moved. This means that when someone accesses this application with a tablet, the settings icon will be the same amount of distance away from the viewport and no change in appearance will occur. *If I used relative positioning, this wouldn't be the case because the reference point is not the viewport but rather itself, and it will maintain it, which could change the position of the icon when using smaller screens.*

Program output



Known Limitations / Missing features

- Buttons yet to be styled.
- No buttons work in this iteration.
- Disclaimer popup

Test No.	Test Description	Expected Outcome	Actual Outcome
10	Settings icon testing	Settings icon has the correct size and position	

Iteration Review

Met requirements: N/A

Changes: Logo positioning to match with settings position

Next steps: Continue/load/save/settings Button styling

Iteration 5 - Continue/load/save/settings buttons styling

components.css

```
.mainMenuButton {
  position: relative;
  font-family: 'Manrope';
  font-size: 20px;
  background-color: var(--highlight);
  border: none;
  padding: 10px 26px;
  border-radius: 5px;
}

/* applying normal button hover effects*/
.mainMenuButton:hover {
  background-color: var(--onhover);
  cursor: pointer;
}

/* TARGET buttons have a blue hover effect*/
.TARGET:hover {
  background-color: var(--primary);
}
```


main-menu.css

```
#settings:hover {  
    cursor: pointer;  
}
```

menu.html

```
<!--Buttons-->  
<div><button class="mainMenuButton TARGET" id="continue">Continue</button></div>  
<div><button class="mainMenuButton">New Flight</button></div>  
<div><button class="mainMenuButton">Load Flight</button></div>
```

What I did

A new file called **components.css** was created to store the code for the buttons. The reason for the creation of this file is the same as base.css in that its styling code will be used for other elements of similar nature throughout the programme and so it's nice to have it here coded once in one place. The only difference is that **components.css** contains styling for UI elements, whilst base.css contains styling for base elements, like the text and background.

Next, In **menu.html** I gave each button a class of **mainMenuButton** to then style all three at once as per design, as they all should look the same. The continue button of course, was given another class **TARGET** because it needs to have a different hover effect and so another class was necessary. I then went over to **components.css** to apply the appropriate styling.

The last thing I did in this iteration was working on the hover effects. I needed to make sure there was both a colour change and that the cursor changed itself to a pointer, to show that a button is selectable. The settings menu specifically needed this because it doesn't change colour and so a pointer is the only indication here.

For the next iteration I've decided to make the **.mainMenuButton:hover** to a separate **button:hover** as normal buttons will have the same hover effect anyway, not specifically the main menu button. This would involve making a new class called **button** and the hover effect will apply to this class. That way the mainMenu buttons are different only in that the style is different.

Why I did it this way

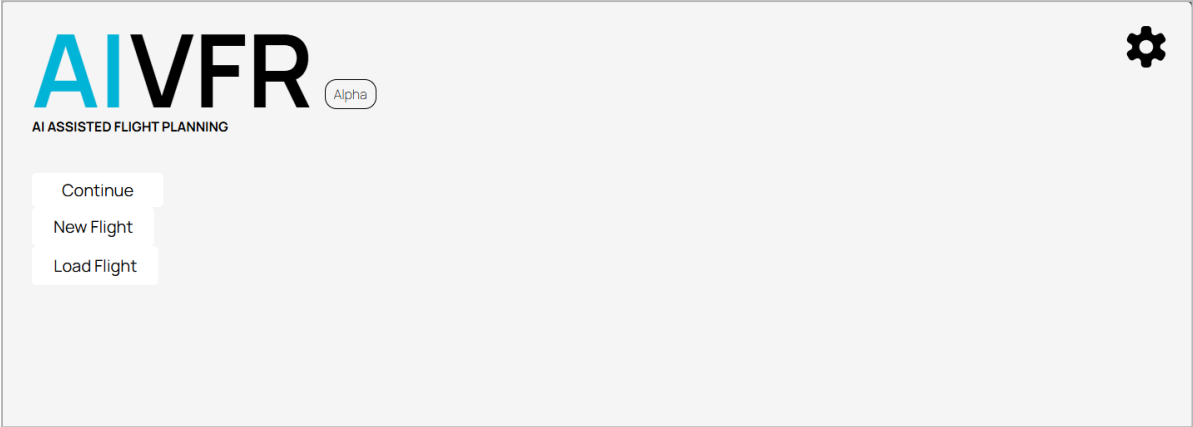
Since components.css and base.css have the same purpose, I could most definitely make them one file. In fact, one might argue that this is easier because it's only one stylesheet to link the HTML to. The reason this alternative is not feasible is because if I did it this way, files will contain simply too much code and finding things will be difficult, even though commented. If I have separate files for different groups of styling however, It makes the code more organised and understandable by anyone who looks at the source code or even me as the developer.

The cursor change to a pointer is extremely important in indicating to the user what they should and should not click, especially for the settings button. There would of course be no other way of indicating that the settings button actually works other than making an entire background effect just for that, which is a waste of time.

The mainMenuButton change is simply for better organisation of code, and efficiency. Given that all non-target buttons will have a single :onhover effect, it only makes sense that

they should all be in one class. Otherwise, I would have to write this hover effect code several times for different buttons if they have even the slightest difference between a normal button, even though the hover effect will be the same. This is of course inefficient and changing it so it is a single class avoids this and greatly improves code quality.

Program output

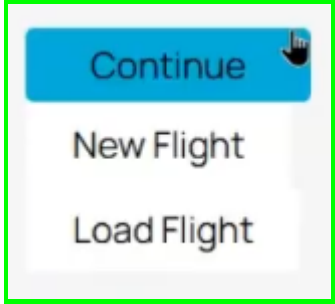
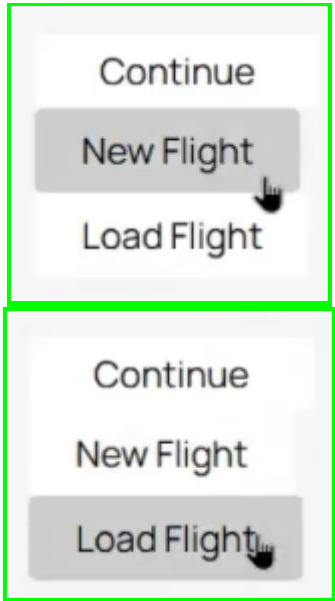
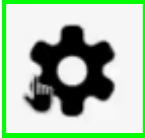


 main_menu onhover test.mp4

Known Limitations / Missing features

- No buttons work in this iteration.
- Disclaimer popup

Test No.	Test Description	Expected Outcome	Actual Outcome
11	Button position and spacing testing	Buttons are in correct position and have correct spacing as per design: Continue New Flight Load Flight	AIVFR AI ASSISTED FLIGHT PLANNING Alpha Continue New Flight Load Flight

12	TARGET Hover testing	The continue button has a blue (primary) colour when hovered over because it's a TARGET. Cursor changes to pointer.	
13	Normal button Hover testing	Button colour changes to onhover colour. Cursor changes to pointer.	
14	Settings button hover testing	Cursor changes to pointer.	

Error Analysis & Fixes

Test No: 11

Failure Description: Incorrect alignment and positioning of menu buttons

Cause: After some research the most probable cause for the spacing is a margin error, as I have not included any margin value to space them from each other. The size of the buttons must be an error with the padding or the button width.

Fix (proposed): increase the margin value for all buttons and try to adjust the button width to make each button have the appropriate appearance.

Justification: Increasing the margin value is the best way to solve this issue rather than changing the position of each button manually. It requires less code and also allows them to have equal spacing between the buttons, whilst moving them manually would see a lot of trial and error in trying to have all of them equidistant from each other.

Iteration Review

Met requirements: N/A

Changes: mainMenuButton will hold a styling difference only. A new class called button will be made to accommodate the normal hovering effect of normal buttons.

Next steps: Making the above change and fixing the positioning error.

Iteration 6 - Main Menu styling completion

```
#continue {  
  width: 260px;  
}  
  
#settings: hover {  
  cursor: pointer;  
}  
  
.mainMenuButton {  
  margin-bottom: 20px;  
  width: 200px;  
  height: 45px;  
}
```

```
/* applying normal button hover effects*/  
.button: hover {  
  background-color: var(--onhover);  
  cursor: pointer;  
}
```

```
<!--Buttons-->  
<div><button class="mainMenuButton TARGET" id="continue">Continue</button></div>  
<div><button class="mainMenuButton button">New Flight</button></div>  
<div><button class="mainMenuButton button">Load Flight</button></div>
```

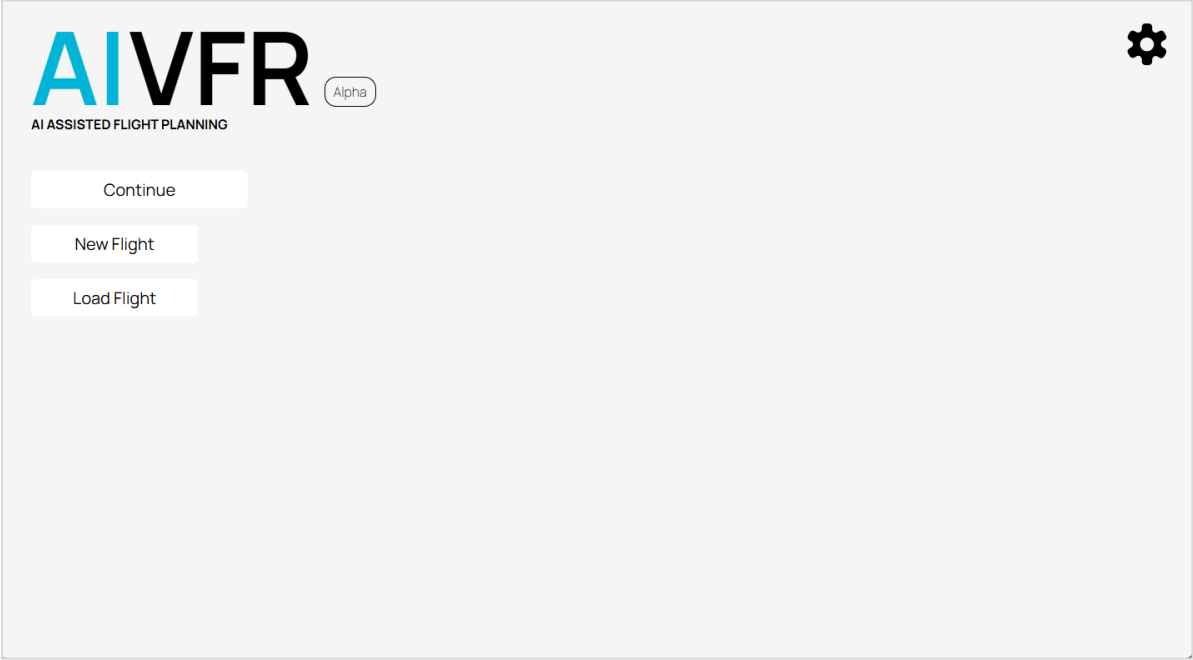
What I did

To finish off the main menu styling, I made the class change so that the hover effect applies to all buttons with class 'button', and I adjusted the margin to make equidistant spacing between the buttons. I then adjusted the width of the buttons to fit the design, with the continue button larger than the main save/load buttons.

Why I did it this way

The justification and reasoning for the above changes were mentioned in the previous iteration. To save time and space I will not re-iterate them here.

Program output (Entire viewport)



Known Limitations / Missing features

- No buttons work in this iteration.
- Disclaimer popup

Test No.	Test Description	Expected Outcome	Actual Outcome
15	Button position and spacing testing	Buttons are in correct position and have correct spacing as per design: Continue New Flight Load Flight	Continue New Flight Load Flight

Iteration Review

Met requirements: (Aim complete) Make the three main menu buttons with appropriate hover effects

Changes: N/A

Next steps: Making the Disclaimer

Iteration 7 - The Disclaimer

menu.html

```
<!--Disclaimer Modal-->
<div id="disclaimer" class="modal">
  <div class="disclaimerModal">
    <p>This webapp uses AI. AI can make mistakes. Although no critical calculations related to the flight are performed by AI, any output from it is susceptible to errors, of v
    <p> As this tool is only in the Alpha stages, nothing from this tool should be used in real flight and is for educational purposes only. If you decide to use data from this
    <p>By pressing the [Continue] Button, you agree that you have read this disclaimer and agree to the terms above.</p>
  </div>
</div>
```

components.css

```
/*modals (popups)*/
.modal {
  /*display: none; makes it hidden by default */
  position: fixed;
  z-index: 1; /*this makes it appear ontop*/
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  overflow: auto; /* enables scrolling if needed (shouldn't be the case) */
  background-color: var(--popupbg)
}
```

main-menu.css

```
/* Disclaimer modal Content/Box */
.disclaimerModal {
  background-color: var(--highlight);
  margin: 15% auto;
  padding: 20px;
  width: 80%;
  border-radius: 20px;
}
```

What I did

The disclaimer is supposed to pop up on the centre of the screen. To do this, 2 *div* elements were needed, one for the dimmed background and one for the text box. Since there may be some other popups that may appear throughout the programme, the class '**modal**' was given to the modal to allow this to be used in the future, hence it also being in components.css. It was important to comment out the `display: none;` line otherwise I would have not been able to see what the popup looks like. It will be re-enabled afterwards for testing.

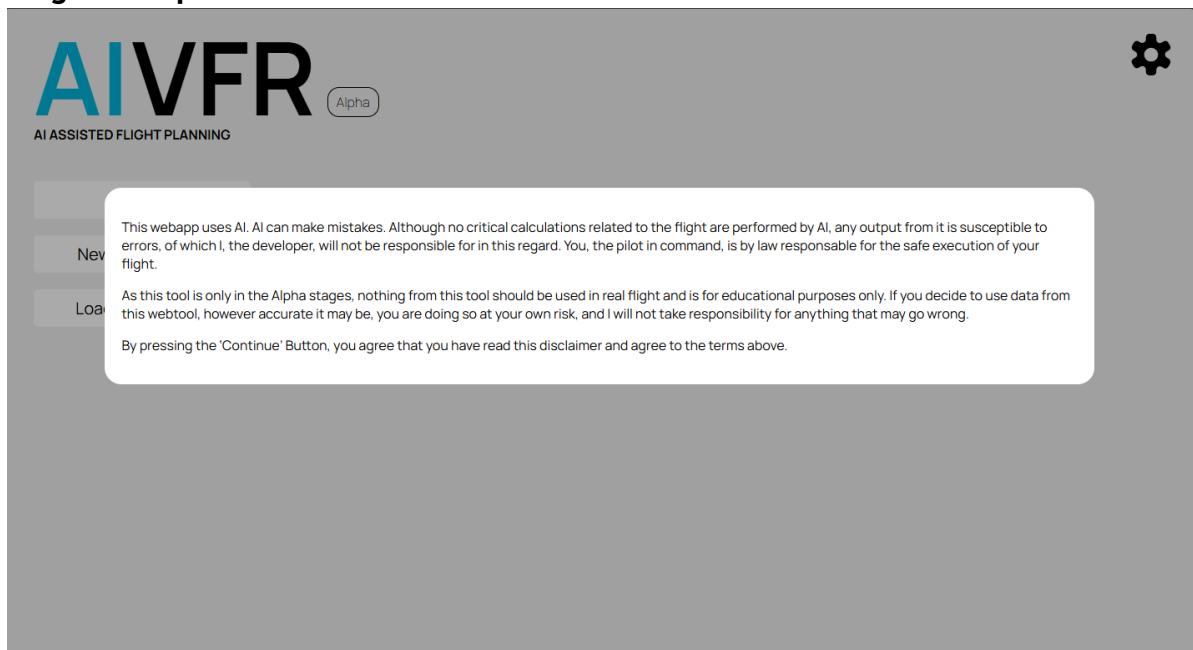
I then went on to paste in the text (from the design) and then edited the actual disclaimer popup (specific to the main menu) in main-menu.css, only adjusting the centering and the corners, before stopping to document my progress thus far.

Why I did it this way

Having a single modal class for all modals makes coding future modals much easier as I don't have to bother styling all of them. If I chose to simply keep the styling specific to main-menu.css, I will spend unnecessary time coding this exact same few lines of code again and again, rather than me being able to just utilise this existing code and perhaps modify it specifically for other popups.

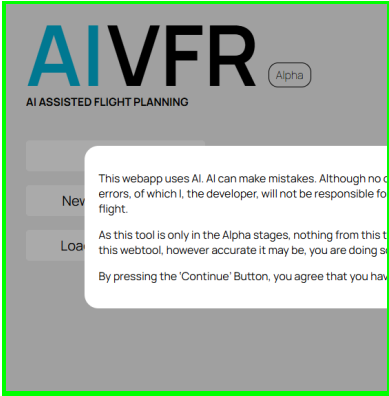
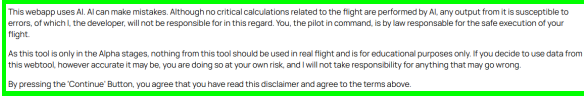
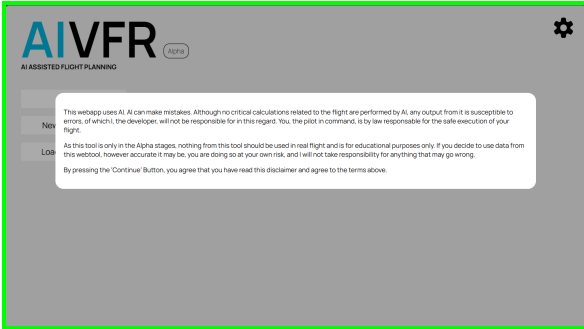
I could have definitely decided to code the entire modal in one iteration, as the tutorial I am using is quite clear and concise. Nevertheless I made the decision to stop here and document my progress as it would make things easier to explain in the long run than If I had coded it to completion immediately. It's also beneficial to you, the reader, to view the iterative development of each feature, rather than its full implementation in one go.

Program output



Known Limitations / Missing features

- No header (to say 'DISCLAIMER')
- No buttons to exit/accept disclaimer
- Formatting of text not yet completed
- Box dimensions are incomplete
- No buttons work in this iteration.

Test No.	Test Description	Expected Outcome	Actual Outcome
16	Modal background test	Background is dimmed as appropriate	
17	Disclaimer content	Disclaimer content matches the design	
18	Disclaimer positioning	Popup appears in the centre of the screen	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Disclaimer popup completion (i.e. implementation of all known missing features)

Iteration 8 - Disclaimer popup completion

menu.html

```
<p> As this tool is only in the Alpha stages, <u><em><b>nothing from this tool should be used in real flight and is for educational purposes only.</em></b></u>
<p>By pressing the <span><button class="disclaimerButton button TARGET">CONTINUE</button></span>
<span><button class="disclaimerButton button">GO BACK</button></span>
```


main-menu.css

```
/*'DISCLAIMER' header at the top*/
#disclaimerHeader {
  font-weight: 500;
  font-size: 36px;
  margin-top: auto;
}

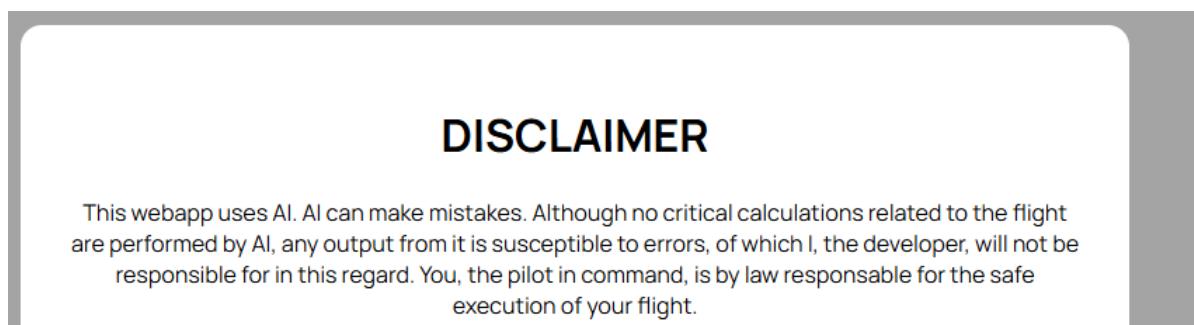
/*Continue and go back buttons*/
.disclaimerButton {
  position: relative;
  font-family: 'Manrope';
  font-size: 20px;
  background-color: var(--bg);
  border: none;
  padding: 6px 20px;
  border-radius: 5px;
  margin-left: 5px;
  margin-right: 5px;
}
```

```
/* Disclaimer modal Content/Box */
.disclaimerModal {
  position: relative;
  background-color: var(--highlight);
  margin: 15% auto;
  padding: 35px;
  width: 720px;
  border-radius: 15px;
  text-align: center;
  bottom: 50px;
}
```

What I did

To finish the disclaimer popup, I first adjusted the box dimensions to make the text look a little more compact. I settled with `width: 720px`, which is what is shown in the program output. It's important to note that this (and other aspects such as the menu buttons, disclaimer buttons, etc) deviate from the actual sizes given in the design. It seemed like I oversized all the buttons whilst designing on Canva and so in the real thing many elements appear smaller than they actually are.

Then I added the 'DISCLAIMER' title at the top, and then aligned everything to the centre as per design. The heading initially brought some issues with formatting...



...so I gave the `<h1>` an id `disclaimerHeader` to edit its top margin so there is less space between it and the top. I also took this opportunity to reduce the boldness and increase the size of the header to match the design. The result is in the final output.

Lastly, I formatted the important part of the disclaimer 'nothing from this tool should be used in real flight and is for educational purposes only.' to be bold and underlined. I also decided to make it italic via emphasis tags because why not.

That was the disclaimer popup complete, but I had one more problem.

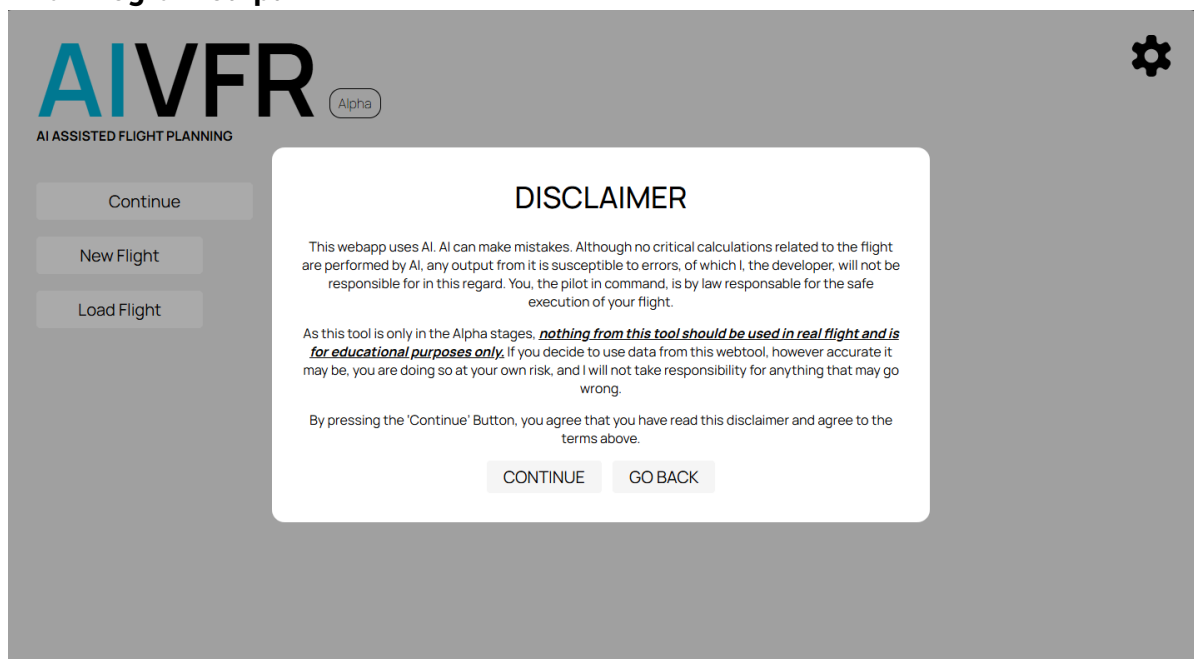
The tutorial I used utilised an **overflow** attribute that allowed scrolling if required. Initially I thought this wouldn't need to happen and so I left it as it was, but after adjusting the disclaimer the scrollbar did appear on the side. It was at this point that I realised that the popup wasn't centred properly because by default the box was quite low on the screen. I decided to remove this and manually centre the popup to match the design myself, as this wouldn't have had too much of an impact on smaller screens, keeping in mind this webapp is not targeted at small mobile phones.

Why I did it this way

Breaking from the design was necessary simply because things were too large in the mockup design, and it felt unusual, as though the screen was zoomed in. If I strictly followed the design then this will be persistent throughout the webtool and this unusual sizing will force the user to zoom out manually all the time, which is heavily inconveniencing and is bad for user experience.

If the overflow value was still set to auto (as it was before), then the popup (and potentially future popups) will not be centered and this could degrade the design consistency of the webtool, which again isn't good for user experience as a user prefers to see a design that is kept consistent throughout. This would be different on a mobile phone, however, but I am not considering them as their screens are too small and so there would have to be some major upgrades to the webapp to allow it to be responsive enough to handle such screen sizes- not something I plan to implement into the final solution now because it takes too much time.

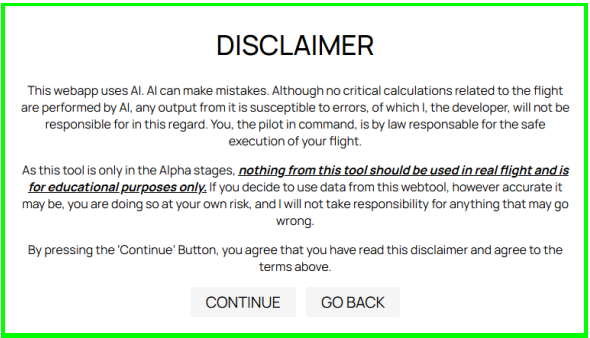
Final Program output



DisclaimerModal onhover test.mp4

Known Limitations / Missing features

- No buttons work in this iteration.

Test No.	Test Description	Expected Outcome	Actual Outcome
19	Disclaimer content and formatting test	Disclaimer content and formatting matches that of the design	
20	Hover button tests	'Continue' button turns blue on hover 'Go back' button turns grey on hover	

Iteration Review

Met requirements: **Essential feature complete:** Disclaimer

Changes: removed overflow:auto to ensure proper centering of popup box

Next steps: Scripting the buttons

Iteration 9 - Scripting the buttons: Main Menu -> Disclaimer

```
#main menu
@main.route('/') #this will be the first to load up
def index():
    return render_template('menu.html')

#settings menu
@main.route('/settings')
def settingsMenu():
    return render_template('settings.html')
```

What I did

The first thing I did was to link **menu.html** to replace the 'menu page' message that showed up when we ran flask. This meant that instead, the first page that loads up is the main menu. I did the same for the settings menu as well as at one point we will need to make the settings button load up the settings menu too.

I tested this immediately to make sure that menu.html shows up first as expected.

Why I did it this way

There was no way to test the buttons to make sure they open another page unless I linked them to a flask which holds the code for all of the other pages. Without this link then the scripting part of the project would be impossible.

Program output

This was not what I expected at all, as the linking code was indeed written as per the tutorial.

Web

Internal Server Error

The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.

Terminal

```
File "/workspaces/Flight_Planning_Companion/.venv/lib/python3.11/site-packages/flask/_get_source_fast.py", line 99, in _get_source_fast
    raise TemplateNotFound(template)
jinja2.exceptions.TemplateNotFound: menu.html
127.0.0.1 - - [09/Aug/2025 08:51:33] "GET / HTTP/1.1" 500 -
```

Known Limitations / Missing features

- No buttons work in this iteration.

Test No.	Test Description	Expected Outcome	Actual Outcome
20	Flask running test	Menu.html runs and shows up first on the screen.	Internal Server Error The server encountered an internal error and was unable to complete your request. Either the server is overloaded or there is an error in the application.

Error Analysis & Fixes

Test No: 20

Failure Description: 500 error - page won't load

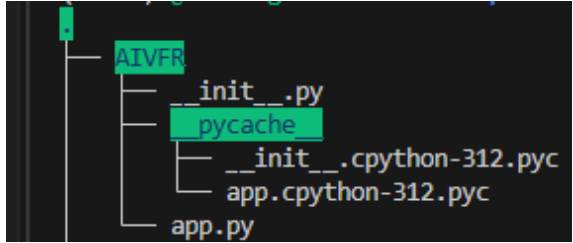
Cause: 'TemplateNotFound' error - it can't find menu.html

Fix: Currently unknown. (code was written as per tutorial) I will search online if the error is common amongst others, and a solution should be found.

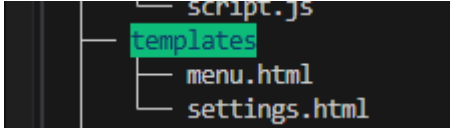
Justification: Better to search for a solution than to do things I don't know!

Iteration 9.5 - Fixing TemplateNotFound Error

A quick search revealed that my files locations may not have been placed correctly. To check this, I ran 'tree' in VSCode to give me the file positions of every file in the project relative to the root, and the error became evident.

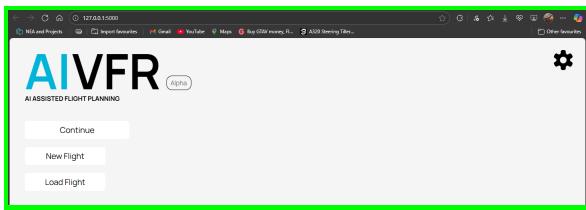


app.py is within the 'AIVFR' folder...



... but **templates** is attached to the root (.).

The evident fix was to move **templates** into the AIVFR folder. I ran another test and the page ran successfully.

Test No.	Test Description	Expected Outcome	Actual Outcome
21	Flask running test	Menu.html runs and shows up first on the screen.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Continuing the scripting process.

Iteration 10 - Scripting the buttons: Main Menu -> Disclaimer

menu.html

```
<div><button class="mainMenuButton button" id="newFlight">New Flight</button></div>
<div><button class="mainMenuButton button" id="loadFlight">Load Flight</button></div>

<span><button class="disclaimerButton button TARGET" id="disclaimerACCEPT">CONTINUE</button></span>
<span><button class="disclaimerButton button" id="disclaimerEXIT">GO BACK</button></span>
```

main-menu.js

```
//ensuring the HTML has loaded properly before running the script
document.addEventListener("DOMContentLoaded", function() {
  console.log("JS LOADED") //confirming the script has now loaded

  //linking the buttons and disclaimer to the script
  var disclaimer = document.getElementById("disclaimer");
  var Continue = document.getElementById("continue");
  var loadFlight = document.getElementById("loadFlight");
  var newFlight = document.getElementById("newFlight");
  var disclaimerContinue = document.getElementById("disclaimerACCEPT");
  var disclaimerGoBack = document.getElementById("disclaimerEXIT");

  //determines what happens if the user presses continue on disclaimer
  var EntryState = 'none'

  //main menu
  Continue.onclick = function() {
    disclaimer.style.display = "block";
    EntryState = 'Continue'
  }

  loadFlight.onclick = function() {
    disclaimer.style.display = "block";
    EntryState = 'Load'
  }

  newFlight.onclick = function() {
    disclaimer.style.display = "block";
    EntryState = 'New'
  }

  //disclaimer
  disclaimerGoBack.onclick = function() {
    disclaimer.style.display = "none";
  }
});
```

What I did

When I was about to start scripting, I realised something that may have an effect on the rest of my code, and it had something to do with the `.modal` class, which was in **components.css** at the time. The script was to change the value of `display:none` to `display:block`- this attribute is located in the styling for `.modal`, which is located in **components.css**, which will be used for other popups. This forced me to move it into **main-menu.css** to ensure it doesn't have an effect on other popups.

Another quick change I made was to change the class name `disclaimerModal` to `disclaimerContent`, for clarity purposes.

When coding the JavaScript for the buttons, I initially coded the single few lines below to see if I was on the right track and I hadn't deviated from what the modal tutorial was telling me:

```
var disclaimer = document.getElementById("disclaimer");
var Continue = document.getElementById("continue");

Continue.onclick = function() {
    disclaimer.style.display = "block";
}
```

This was enough to test if just the continue button was going to open the disclaimer, and nothing worked. There were no visible errors, but nothing worked.

It took me quite a long time to figure out why this code didn't work, but eventually it came down to the fact that the code was not running at all, and this was because the HTML had not fully loaded and the javascript was loading **before** the HTML had the chance. This led to the variables becoming *null* variables because javascript couldn't find them in the HTML.

The line `document.addEventListener("DOMContentLoaded"...` fixed this by checking that the HTML had fully loaded, and then running the script. This did mean that everything else must be written inside a function but this was not much of an issue.

The variable `EntryState` will be the way I can tell the code what to do depending on the button pressed in the main menu. The **Entry state** is simply the state in which the program will be entering into the planning module. Either continuing a flight plan, having made a new one, or loaded a saved one. When the user presses the 'continue' button, the entry state will tell the button what to do, e.g. if `EntryState = 'Continue'` then it would just open the dashboard, utilizing data already stored in the cache, but if it was `'Load'` it would need to pull up the user's files to access the .json file to load up first.

Why I did it this way

If I kept the styling for `.modal` in `components.css`, and used it for other popups, when triggering this modal, it will trigger all the other popups, too, which use the same class. This was not ideal at all, and so I made the decision to have it specific to the main menu, and moved it to `main-menu.css`.

Changing `disclaimerModal` to `disclaimerContent` directly illustrates what the class is responsible for. Using 'Modal' is ambiguous because it could mean the background (the actual modal) or the whole popup as a whole, when really it referred to just the content in the box, and so the name change shows exactly that. This evidently improved the readability of my code.

Because the disclaimer is what leads to the flight planning module in all 3 situations (continue/new/load), I needed a way for a single button to be able to still have 3 different outputs. Using a variable that changes depending on what button is clicked means I can have if statements that denote what happens if the state of entry is any of the 3. The other extremely inefficient alternative would see me creating 3 separate popups for 3 separate buttons, just so each button can be linked to continuing/loading/starting a new flight plan. This simple variable eliminates the need for this and is arguably easier to understand from a developer perspective.

Program output

 Disclaimer popup test.mp4

Known Limitations / Missing features

- Continue button doesn't open dashboard when 'Continue' selected
- Settings button not scripted

Test No.	Test Description	Expected Outcome	Actual Outcome
22	Disclaimer popup test	All 3 buttons can trigger the disclaimer popup	PASS (See Program Output)
23	Disclaimer popup closing	The 'Go back' button closes the disclaimer but does NOT do anything else	PASS (See Program Output)

Iteration Review

Met requirements: N/A

Changes: disclaimerModal -> disclaimerContent, .modal moved from components.css to main-menu.css

Next steps: Page to page scripting (settings and continue button) and menu completion.

Iteration 11 - Scripting the buttons: Settings button, Continue button

app.py

```
#dashboard
@main.route('/dashboard')
def dashboard():
    return render_template('dashboard.html')
```

menu.html

```
<!--settings button-->
<a href="{{ url_for('app.settingsMenu') }}"></a>
```


main-menu.js

```
//display the disclaimer
function displayDisclaimer() {
  disclaimer.style.display = "block";
}

//main menu
Continue.onclick = displayDisclaimer
EntryState = 'Continue'

loadFlight.onclick = displayDisclaimer
EntryState = 'Load'

newFlight.onclick = displayDisclaimer
EntryState = 'New'

//disclaimer
disclaimerGoBack.onclick = function() {
  disclaimer.style.display = "none";
}

//continue button
if (EntryState = 'Continue') {
  disclaimerContinue.onclick = ContinueFlight
}

//-----Continue, load, make new flight

//Continue
function ContinueFlight() {
  window.open("{ url_for('app.dashboard') }", '_self')
}
```

What I did

First, I linked and quickly tested the dashboard to its html page in app.py, as the continue button needed to lead to it.

Then, to link the settings button to the settings menu, I used an `<a>` element to make clicking the picture link to open the settings html page. I then used the `'url_for'` flask function to open the specified page when the button is clicked.

Before moving on to coding the main menu buttons in JS, I made a change so that a single function `displayDisclaimer()` displays the disclaimer, and when the buttons are clicked it just runs that function instead.

To code the buttons I used the same logic. I made an if statement that runs a specific 'button-running function' that runs a function for that specific button when pressed, this should ensure that only the correct function runs when the correct button is pressed.

I then used the same **url_for** function within the JS **windows.open** function to finally open the dashboard, only when the continue button was pressed (as EntryState would be 'Continue').

Why I did it this way

An alternative to using the anchor element when coding the settings button was to use the **<form>** element instead, which I quickly rejected after testing, because it meant turning the settings icon into a HTML button, which requires styling to make it invisible. Although that's a very quick thing to do, less code was present in the former option and so I took it for the sake of efficiency. This never changed the result, as you will see.

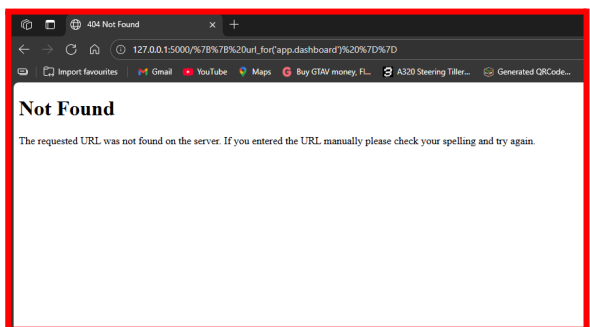
The **displayDisclaimer** choice was another choice for the sake of efficiency, as the same code was being used multiple times and so making it into a function made it more concise and easier to interpret.

Program output

 Page-to-page main menu loading test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
24	Main-menu -> settings page	Pressing the settings button takes you to the settings page	PASS (See Program Output)
25	Continue (via dashboard) -> dashboard page	Pressing the continue button on the dashboard page takes you to the dashboard ONLY if you've pressed 'Continue Flight'.	 ERROR (Other buttons take you to dashboard too)

Error Analysis & Fixes

Test No: 25

Failure Description: 404 Error appears when loading dashboard, 'New flight' and 'Load Flight' Both load the dashboard which shouldn't happen.

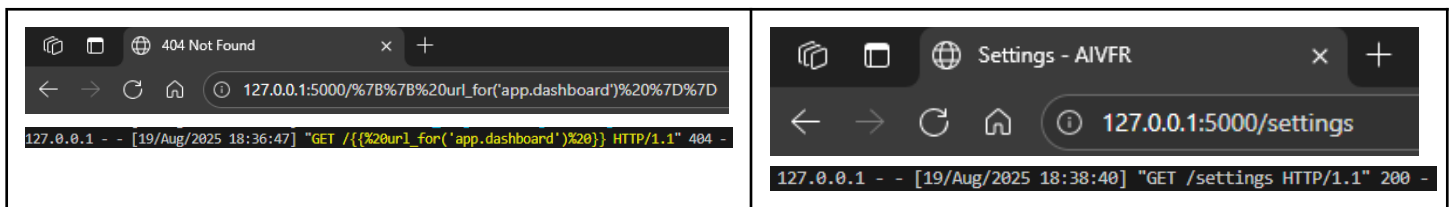
Cause: Probably due to the wrong URL in the search bar, caused by the wrong URL in the `window.open` function. The button issue is most likely an issue with the Entry state - perhaps it never changed.

Fix: Change the URL in JS to open disclaimer, and add code to check the entry state to find out the problem with the buttons.

Justification: A 404 error is caused by an incorrect URL in the search bar and the server can't find the page being requested, therefore, a suitable fix should be to change the URL that appears in the search bar. The `EntryState` variable is the only thing that could affect what function runs so it must be an issue related to the variable that is making the function apply to all three buttons.

Iteration 11.5 - Fixing the 404 Error

What stood out to me during testing was what was in the URL page when running the settings menu and what was there when running the dashboard page, as well as the output in the terminal.



From this I discovered that the `url_for` function only works when used alongside a `href` value, in a button or anchor element, for example. The `window.open` function didn't read this and just put the whole `"{{ url_for('app.dashboard') }}"` into the searchbar, hence the error. So, I assumed it just opened what you wrote in it, so I wrote the name of the page function in `app.py` ('dashboard') into the `window.open` function instead, which gave positive results.

For the issue regarding the entry state, I did exactly what I said I would and made sure I log the entry state whenever the button was clicked. This looked like this:

```
//main menu
Continue.onclick = displayDisclaimer
EntryState = 'Continue'
console.log(EntryState)

loadFlight.onclick = displayDisclaimer
EntryState = 'Load'
console.log(EntryState)

newFlight.onclick = displayDisclaimer
EntryState = 'New'
console.log(EntryState)
```

And the output was this:

JS LOADED	main-menu.js:3
Continue	main-menu.js:24
Load	main-menu.js:28
New	main-menu.js:32

The issue became apparent here. What I didn't notice was that the entry state changes without any condition control. I never specified for the entry state to change when a specific button is clicked, and so it changes when it reads the line to change it, which happens in this order. The obvious fix was to make sure the entry state only changes when their respective button is pressed, and the best way to do it was with the event listener.

The removal of the EntryState variable

In doing the above, I never succeeded to make the screen change only if the entry state is correct. For some reason it would either work on all buttons, or none at all. The way to fix this was to keep the `ContinueFlight()` function there, but have it not rely on the Entry state, but in the case that the Continue button was clicked (which was what I solved before). This meant that the entry state variable was essentially useless, and removing it was the suitable option as *keeping it in the code would not impact whether the dashboard page would open or not*. Effectively, the EventListener became the EntryState as whatever was within it would only run if the correct button was pressed.

Fixed solution

```
//display the disclaimer
function displayDisclaimer() {
  disclaimer.style.display = "block";
}

Continue.addEventListener("click", function() {
  displayDisclaimer();
  ContinueFlight(); //determines what happens if the user presses continue on disclaimer
});

loadFlight.addEventListener("click", function() {
  displayDisclaimer();
});

newFlight.addEventListener("click", function() {
  displayDisclaimer();
});

//disclaimer
disclaimerGoBack.onclick = function() {
  disclaimer.style.display = "none";
}

//-----Continue, load, make new flight

function ContinueFlight() {
  disclaimerContinue.onclick = function() {
    window.open("dashboard", '_self')
  }
}
```

Test No.	Test Description	Expected Outcome	Actual Outcome
26	Continue (via dashboard) -> dashboard page	Pressing the continue button on the dashboard page takes you to the dashboard ONLY if you've pressed 'Continue Flight'.	Debugged main menu test.mp4 PASS

Iteration Review

Met requirements: (Aim complete) Test the buttons such that the settings and continue/new/load flight buttons do what they're supposed to.

Changes: Removal of the Entry State Variable

Next steps: Section 3 - Settings menu

Section Review

Aims completed in this section:

- Implement the logo and appropriate colouring & styling of the page
- Make the three main menu buttons with appropriate hover effects
- Implement the first essential feature: **disclaimer**
- Test the buttons such that the settings and continue/new/load flight buttons do what they're supposed to.

Aims not completed in this section:

N/A

Iterative tests complete:

- The first page that loads up after the main menu is the dashboard.
- Disclaimer pops up before opening the dashboard, and there is no available way to bypass it.
- Buttons are operative. If colour is to change they do, and when clicked they trigger the correct output.

Remarks: Nothing to address.

Section 3: Settings Menu

Aims of this section

- Complete a success criteria item: Settings Menu
- Create the full main design for the settings menu, including buttons and headers
- Create the ability to navigate between the settings and main menu
- Code the ability to switch between light/dark mode
- Test light and dark settings for main menu and settings menu
- Allow the buttons to change relevant items in flight_data.json

This section will see me creating and implementing basic changes in flight_data.json. Pressing the relevant settings buttons should change the relevant variables. I am aware that this data may or may not reset when re-running the program, and the ability for these settings to save in the local storage will be implemented in section 5, as this section is mainly dedicated to the settings menu and its functioning, and the implementation of the aforementioned feature might require several iterations that would make this section unnecessarily long. Instead it shall be its own section.

Iteration 1 - Header

Settings.html

```
<link rel="stylesheet" href="../../static/settings.css">
</head>
<body>
  <header>
    <!--Go back arrow icon-->
    <span></span>
    <span>
      <!--title ('settings')-->
      <div id="settings">Settings</div>
    </span>
  </header>
</body>
```

Settings.css

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
/*header to contain the title and the back button*/
header {
  background-color: var(--highlight);
  display: flex;
  align-items: center;
  text-align: center;
}

/*Back arrow*/
#goBack {
  width: 40px;
  margin-left: 10px;
  margin-top: 6px;
  color: var(--text);
}

/*'settings'*/
#settings {
  position: relative;
  font-size: 24px;
  font-weight: 300;
  margin-left: 42vw /*not centered definitely but is estimate- should fix*/
}
```

What I did

This header section was relatively straightforward. The two elements are span elements to ensure they were aligned, and I used flexbox to make centering all the items a much easier process.

Whilst coding this I tested the idea of having the 'settings' title on the side or in the middle of the header, which is different from the design (which is on the side). I liked it better in the middle, and as well as avoiding a rare but potential misconception that I'll explain below, I decided to change it to be centered as shown in the program output.

When it came to centering however, an issue occurred in which I couldn't find the right way to center the 'Settings' title to the centre of the viewport. I instead used the viewport percentage (**42vw**) that gave the closest look to the centre of the screen, and tested it using a website that allows you to view different screens for different devices. It came alright, although I am still aware it's not perfectly centered and thus have left a comment saying this- perhaps I'll later research the right way to do this when I get the time (as all conventional methods seemed not to work).

Why I did it this way

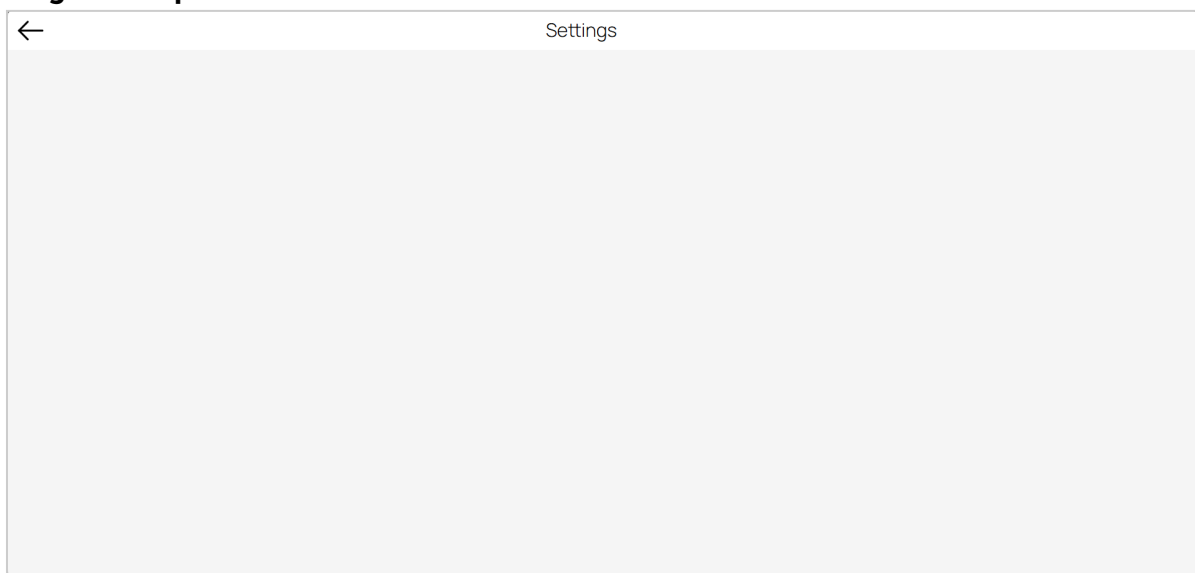
Initially, I had attempted to have the items placed without using flexbox and I had a rather difficult time trying to center the 'Settings' title vertically. All the things you would try

normally were to no avail and so after some research, flexbox was the most optimal solution.

I decided to center the settings title in the middle rather than on the side because there was a small chance of people thinking that the arrow that the title was next to lead to another settings page. Usually a word with an arrow next to it indicates that the arrow leads to that place, and so I wanted to avoid users from accidentally going back to the main menu. It also looked like a better design choice to have it in the centre, from a visual perspective.

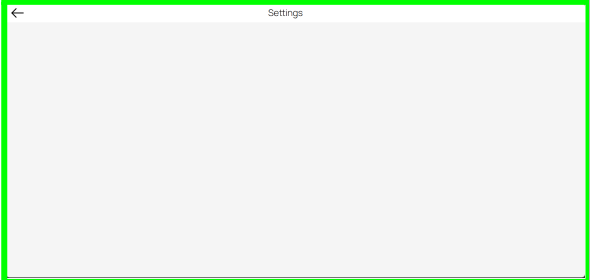
Leaving the Settings title at 42vw instead of trying to center it properly was a decision governed by the fact that I just didn't have the time to start looking for the proper way to do this, especially as all the conventional methods were not working properly. Although it was a quick solution it was a good fix as on other screens it still appears as though it is centred.

Program output



Known Limitations / Missing features

- Back button is inoperative

Test No.	Test Description	Expected Outcome	Actual Outcome
27	Header appearance testing	Header appears at the top of the screen with no gaps and all elements within are placed correctly.	

Iteration Review

Met requirements: N/A

Changes: 'Settings' Title is centered instead of on the side

Next steps: main content and buttons

Iteration 2 - Main content: Elements dump and organisation

As we did before for the main menu, I will dump everything we will need without any formatting in their expected positions. I will also be adding all the other options that were omitted initially from the design.

settings.html

```
<!--Light/dark toggle-->
<h2>THEME</h2>
<button>Light</button>
<span><button>Dark</button></span>

<!--Units-->
<h2>UNITS</h2>

<div style="display: flex;">
  <div> <!--AIRSPEED-->
    <button style="display: block;">Airspeed</button>
    <button style="display: block;">Knots</button>
    <button style="display: block;">MPH</button>
    <button style="display: block;">KM/H</button>
  </div>

  <div> <!--ALTITUDE-->
    <button style="display: block;">Altitude</button>
    <button style="display: block;">Feet (ft)</button>
    <button style="display: block;">Metres (m)</button>
  </div>

  <div> <!--MASS-->
    <button style="display: block;">Mass</button>
    <button style="display: block;">Kilograms (kg)</button>
    <button style="display: block;">Pounds (lbs)</button>
  </div>

  <div> <!--FUEL-->
    <button style="display: block;">Fuel</button>
    <button style="display: block;">Litres (l)</button>
    <button style="display: block;">US Gallons (gal)</button>
  </div>

  <div> <!--Distance-->
    <button style="display: block;">Distance</button>
    <button style="display: block;">Nautical Miles (NM)</button>
    <button style="display: block;">Kilometers (km)</button>
    <button style="display: block;">Statute Miles</button>
  </div>
</div>
```

What I did

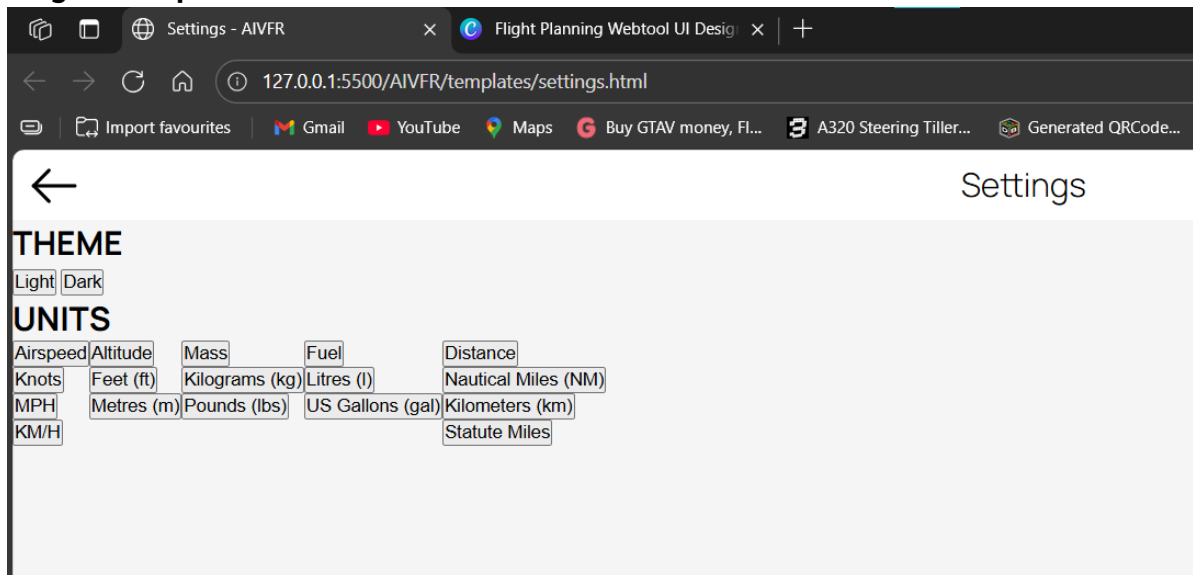
The result is shown below, as you can see the buttons are in the correct positions but lack the formatting. To make them like this, the best way was again to use flexbox here, to simply align them side by side from each other as shown below.

2 changes were made here. The first being that instead of it starting with 'Dark' on the left and 'Light' on the right, I swapped them around so it says light, then dark. This just seemed like convention and looked right to me. It is also what is common in other programmes too- light goes before dark. The 2nd thing I changed was the 'Time' column of options was changed to 'Distance', as *there were no useful units of time to switch between, and I had just placed it there to fill space without even releasing*. Distance on the other hand, has many more options as you can see below.

Why I did it this way

When I tried to use div and span elements, they never appeared in the correct format and I concluded that it would never work in my specific scenario. Flexbox allows each div to be positioned at equal intervals from each other as well, which will come in handy when I begin styling the buttons and the rest of the page.

Program output



Known Limitations / Missing features

- No styling applied
- All buttons are inoperative

Test No.	Test Description	Expected Outcome	Actual Outcome
28	Elements positioning	All elements positioned correctly as per design	

Iteration Review

Met requirements: N/A

Changes: Light button now comes before dark button | Replaced distance column with time column.

Next steps: Styling

Iteration 3 - Settings Menu Styling

Settings.css

```
/*All buttons default styling*/
button {
  margin-bottom: 20px;
  min-width: 165px;
  max-width: min-content;
  min-height: 45px;
  position: relative;
  font-family: 'Manrope';
  font-size: 20px;
  background-color: var(--highlight);
  border: none;
  padding: 8px 26px;
  border-radius: 5px;
}

/*All except the header*/
#options {
  padding: 40px;
}

/*dark button spacing from light button*/
#dark {
  position: relative;
  margin-left: 7px;
}

h2 {
  font-weight: 400;
}

/*ONLY the buttons under UNITS*/
#UnitsButtons {
  display: flex;
  gap: 10px;
}
```

What I did

To apply the default button styling to all the other buttons required me to use the same initial code I used for the main menu styling. I did this and applied it to all buttons, and then adjusted the minimum size so that they don't all have different sizes, and adjusted the maximum height so that they could all maintain the same width which was important for the spacing aspect of these buttons.

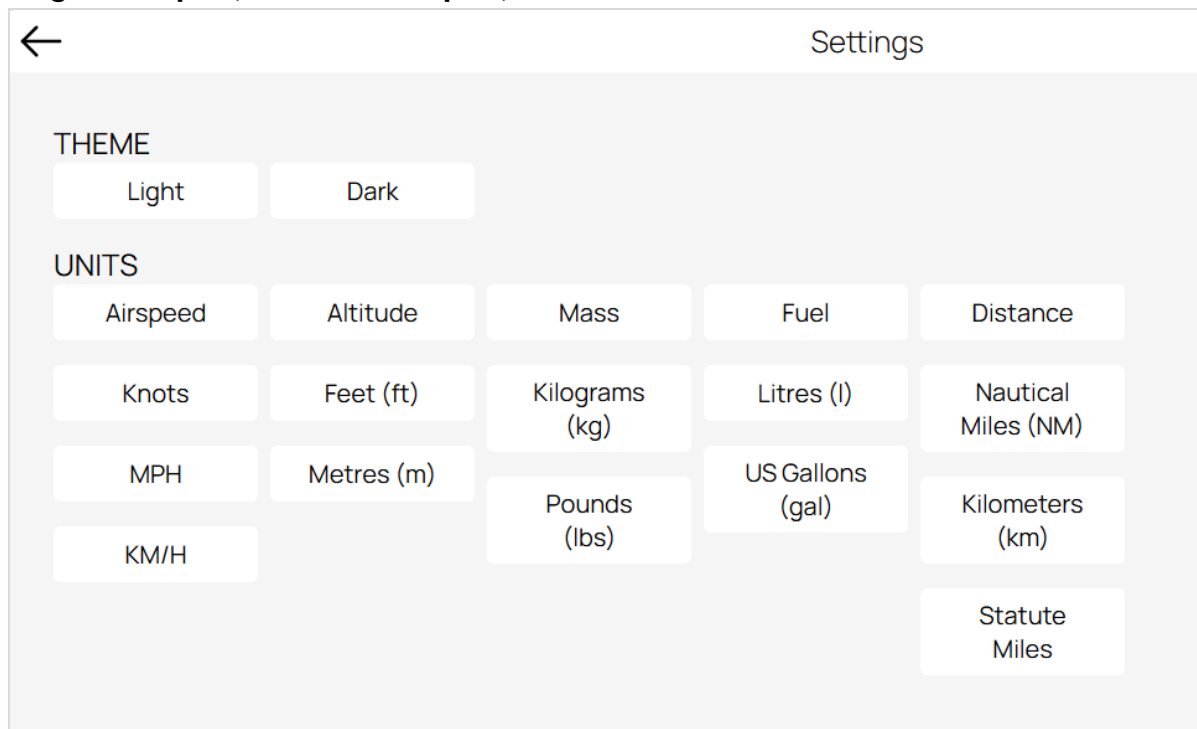
Flexbox allowed me to equally space the boxes by 10px, which produced the neat result as shown below. In order to achieve this whilst considering all the margins of all the buttons, and then just the units buttons, I had to split the page into 3 sections, the main **body**, the **options**, and the **unitsButtons**. The options, as shown above, are all the setting's options- which is everything but the header, and the unitsButtons are simply just the units buttons.

Why I did it this way

Using the mainMenuButton css to style these buttons seemed inefficient, but for this case I believe it's the best option. This is because this and the main menu are the only two pages that will have these types of buttons. The other buttons (I.e those in the planning module) will look smaller, and will act differently. This meant that there was no point in making one style for 'all' the buttons to inherit off of (which would be in base.css) because it would only apply for two pages. This technique will however be used when making the base template in the next section.

Splitting the page into 3 sections was the best way to edit large groups of things at once. Of course, this was made easy by putting these groups in their own divs which allowed me to edit them as a whole. Not doing it this way would mean I would've had to move them all separately which is outright ridiculous- this was the only feasible option in this case.

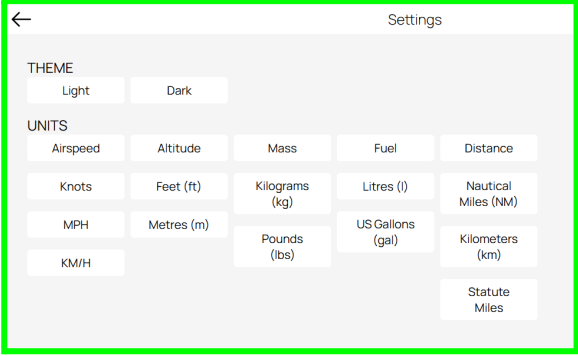
Program output (not entire viewport)



Known Limitations / Missing features

- All buttons are inoperative
- Specific button styling incomplete

- The spacing between the subheadings and buttons is not adjusted (will do this subtly in the next iteration, just adjusting the margin)

Test No.	Test Description	Expected Outcome	Actual Outcome
28	Initial styling and positioning test	-Positioning and styling is accurate as per design -no buttons cross over each other. -Each must have its own clear column.	

Iteration Review

Met requirements: (Aim complete) Create the full main design for the settings menu, including buttons and headers

Changes: N/A

Next steps: scripting light/dark toggle to switch between light and dark modes

Iteration 4 - The light/dark toggle

settings.html

```
<button id="lightToggle">Light</button>
<span><button id="darkToggle">Dark</button></span>
```

settings.css

```
#darkToggle {
  position: relative;
  margin-left: 7px; /*dark button spacing from light button*/
  background-color: black;
  color: white;
  border: 0.1px solid hsl(0 0% 96.1%);
}

#lightToggle {
  background-color: white;
  color: black;
  border: 0.1px solid hsl(0 0% 12.5%);
}
```

settings.js

```
//-----light/dark toggle-----
const root = document.documentElement //this is the <html> element in settings.html
const lightMode = document.getElementById("lightToggle")
const darkMode = document.getElementById("darkToggle")

darkMode.addEventListener("click", () => {
  root.classList.add("dark-mode");
}); //adding the class of the darkmode colours to the html page, so it applies those rules instead

lightMode.addEventListener("click", () => {
  root.classList.remove("dark-mode");
}); //reversing it so that it just applies whatever is in the :root
```

What I did

The first thing to be done was to style the light and dark buttons. In the design, these buttons have fixed colours that do **not** adjust to the theme, and so it's replicated here as well. The border for each mode was made so that it blends with the background of its opposite mode, i.e. the light mode button will have its border colour as the dark mode's background colour, and vice versa.

When it came to coding the toggle, initially, I thought of using a media query, but this proved to be not what I want for reasons I will explain below. The best option instead was to play with the classes currently in the <html> element of settings.html. Turns out that adding the "dark-mode" class from base.css here allows the <html> element to read it and apply it, which has the effect of applying dark mode. This also means that a single button toggles dark mode, and won't untoggle it unless the light mode button is pressed, which simply removes the class from <html> so it just applies :root instead.

Why I did it this way

The border was a key part in the design as it gave the user a confirmation on which option is currently active (although it may seem obvious). This is ultimately more intuitive for the user as there is feedback from the buttons when they're pressed. This method of using the border also relies on just styling with no code involved to change the border colour. Both borders are active but one is always blended in with the other's background. This made the process much quicker and efficient for me as the developer.

The issue with media queries is that they're usually triggered without any direct input without the user- essentially the programme waits till a parameter is reached and then applies what's in the query. The query I was going to use was

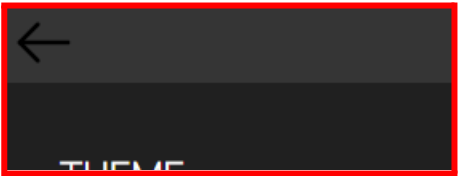
`@media (prefers-color-scheme: dark)`, which only applies when the user's system choice of theme is set to dark. Of course at the discovery of this I did think of having a 'system' option to apply this as it's pretty cool, but I decided not to add it, as per design, and to save time. I may add this feature in the future as it's not too hard to do.

Program output

 Light_Dark toggle test.mp4

Known Limitations / Missing features

- Units and back buttons are inoperative
- Target button styling incomplete

Test No.	Test Description	Expected Outcome	Actual Outcome
29	Darkmode test: Settings	All colours are displayed correctly in darkmode	 
30	Darkmode toggle test	Pressing 'Light' Toggles light mode and 'Dark' Toggles dark mode. One button should not be able to toggle both settings.	PASS (See program output)

Error Analysis & Fixes

Test No: 29

Failure Description: The back arrow does not switch to white when dark mode is toggled.

Cause: Initially I thought I made it a default colour instead of the variable (so it changes) but this wasn't the case. After some research it seemed like .svg files, whilst they can have their colour changed, they can't be stored within the element.

Fix: I'm going to follow a tutorial on how to add colour to .svg image files. This was beyond my scope as I thought you could just simply change the colour using the background or color values.

Justification: Again, given I did not know the solution to this issue a tutorial would suit better than trial and error.

Iteration 4.5 - Fixing the .svg color issue

As per the [tutorial](#), I needed to copy the source code for the svg file and put that embed code into my html. I then had to give it a 'fill' value of "currentColor" which allowed the color to be interchangeable. It was my common sense here to also add the id of the file to match the id of the styling of the image. It's a lot of code on one line as that was simply what the file's source code was, but the result does work and the important details are shown below.

settings.html

```
<!--Go back arrow icon-->|  
<svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 32 32" fill="currentColor" id="goBack"><def
```

Test No.	Test Description	Expected Outcome	Actual Outcome
31	Darkmode test: Settings	All colours are displayed correctly in darkmode	

I also realise that this issue may also be present in the main menu and I will fix these before testing.

Iteration Review

Met requirements: (Aim complete) Code the ability to switch between light/dark mode

Changes: used <svg> instead of to allow icons to change colour

Next steps: Choosing units and scripting appearance as per design

Iteration 5 - Choosing units: Hiding/Showing options

settings.html

```
<button style="display: block;" id="airspeed">Airspeed</button>  
<div id="airspeed_options" style="display: none;">  
  <button style="display: block;">Knots</button>  
  <button style="display: block;">MPH</button>  
  <button style="display: block;">KM/H</button>  
</div>  
</div>
```

settings.js

```
//-----Choosing units-----  
const airspeed = document.getElementById("airspeed")  
const airspeed_options = document.getElementById("airspeed_options")  
  
//airspeed  
airspeed.addEventListener("click", () => {  
  //close all other open options (should just be one column), then:  
  airspeed_options.style.display = "block";  
  airspeed.onclick = function() {  
    airspeed_options.style.display = "none";  
  }  
});
```


What I did

To do this I used the same methods as I did to do the disclaimer in the last section. Just as a recap, you are supposed to be able to press either of the 5 buttons to bring up options for that specific category. So, I needed a way to hide the buttons and show them when their category is clicked. I decided to test it with just the Airspeed options for this iteration.

To do this I added the buttons to be hidden in their own div with an id of 'airspeed_options' for easy identification, and hid it by default by setting the display value to none. Then in JS I coded it so that when the 'Airspeed' button was clicked, the buttons appeared.

I commented that before this happens it should close all other buttons currently open but since I haven't made those other divs yet It's just a comment to remind me to code it later.

Why I did it this way

I only did the airspeed options because I wanted to make sure the code I proposed worked exactly how I wanted it to before sharing it to all the other buttons. *If I decided to just do all of them at once and there is a bug, it'll affect all of them and so I'd have to change values on all button triggers which is unnecessary and time consuming.*

Program output

 Options display testing.mp4

Known Limitations / Missing features

- Not all buttons operative
- Target button styling incomplete

Test No.	Test Description	Expected Outcome	Actual Outcome
32	Dropdown test	The options for airspeed should appear when 'Airspeed' Category pressed	ERROR (Will not re-appear)
33	Dropdown test 2	The options for airspeed should disappear if button pressed again	PASS

Error Analysis & Fixes

Test No: 32

Failure Description: The boxes won't re-appear once they have been opened already.

Cause: It seems like the way I have done it hardcodes what I want it to do and it ends up doing exactly that. There is no way for it to loop and happen again so it doesn't work.

Fix: Although a loop may not be necessary, I need to find a way to use a button as a toggle rather than just hardcode what I want to happen. I will do research to find this out **to avoid wasting time solving the problem myself.**

Iteration Review

Met requirements: Initial scripting complete with some bugs

Changes: N/A

Next steps: Finding a new solution and applying to other dropdowns.

Iteration 5.5 - Adding a toggle for the options display

Much to my surprise, the fix was simple, and it was to use an if-else statement (I've expressed it as a ternary operator for simplicity) to do basically the following:

If the button is clicked and the display value is set to none, change it to block. If not (meaning it was showing already), hide it by setting the display value to none.

```
//airspeed
airspeed.addEventListener("click", () => {
  //close all other open options (should just be one column), then:
  airspeed_options.style.display = airspeed_options.style.display === "none" ? "block" : "none"; //toggles between the styles none and block
});
```

Toggle test 2.mp4

Test No.	Test Description	Expected Outcome	Actual Outcome
34	Dropdown test	The options for airspeed should appear when 'Airspeed' Category pressed	PASS

I can now apply these same settings to the other buttons, and the results are shown below:

toggle test final.mp4

Test No.	Test Description	Expected Outcome	Actual Outcome
35	Dropdown test	The options for All buttons should appear when the respective category is test pressed	PASS

Iteration 6 - Choosing units: Closing other options and adding colours

When a user presses one of the categories, that particular category should light up as selected and should stay that color. This is what I plan to code in this iteration. I will also make sure only one category is shown at a time.

settings.js (other buttons are similar so are omitted from this screenshot)

```
//mass
mass.addEventListener("click", () => {
  altitude_options.style.display = "none";
  airspeed_options.style.display = "none";
  fuel_options.style.display = "none";
  distance_options.style.display = "none";
  mass_options.style.display = mass_options.style.display === "none" ? "block" : "none"; //toggles between the styles none and block
});

//fuel
fuel.addEventListener("click", () => {
  altitude_options.style.display = "none";
  mass_options.style.display = "none";
  airspeed_options.style.display = "none";
  distance_options.style.display = "none";
  fuel_options.style.display = fuel_options.style.display === "none" ? "block" : "none"; //toggles between the styles none and block
});
```

```
//changing the color of the button to primary unless another button is clicked
const dropdown = document.querySelectorAll(".dropdown") //all elements in this class stored in array
dropdown.forEach((btn, index) => { //iterates through the array and gives each button its on event listener
  btn.addEventListener("click", () => {
    const isActive = btn.classList.contains("active"); //checks if button is selected already

    dropdown.forEach(drp => drp.classList.remove("active")); //reset anything else that's active first

    if(!isActive) {
      btn.classList.add("active"); //if not active already then make it active
    }
  });
});
```

settings.css

```
/*buttons retain TARGET colour unless clicked again. active must be in classList*/
.dropdown.active {
  background-color: var(--primary);
}
```

What I did

To make sure only one category appeared at a time, when a particular button was clicked I set all the other option's display values to none. This effectively made sure just that single column of options were showing at a time.

Then it came to doing the same thing for the colours, which was actually very difficult as I was working with a class called 'dropdown' that was assigned to all those main buttons (i.e. Airspeed, Altitude, Mass, etc). This would of course be easier than giving them all IDs and repeating this entire process as I did before.

So after some research I found that I can use the fact that when a button is active the 'active' value is added to its classList. This meant that I could use this as a basis to check what buttons are active and add/remove this from the classList to activate/deactivate the coloring. This definitely proved useful and I believe I will be using methods involving the manipulation of the classList to solve other problems in the future.

Why I did it this way

The method used to switch around the colours helped significantly in making sure I didn't have to write so many lines of code for a small function like this one. Once I discovered this method I realised that I could probably do the same with changing the display values of the options to 'none' and 'block', which meant I didn't have to write long lines of code like I did in this case, but I decided not to make the change because I had already written it out and it proved to work relatively quickly, and so I deemed it unnecessary to change. Should slowness be a problem, I might improve this like I did above to hopefully make the code run a little bit faster.

Program output

 Toggle color test.mp4

Known Limitations / Missing features

- Units cannot yet be chosen
- Back button is inoperative

Test No.	Test Description	Expected Outcome	Actual Outcome
36	Units toggle test with colours	All options should appear one at a time and the main button should highlight and stay highlighted with the primary colour	PASS

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Back button

Iteration 7 - Back button

What I did:

This one is very simple and just involves adding the 'onclick="history.back();"' values in the <svg> element:

settings.html

```
onclick="history.back();" style="cursor: pointer;"><defs><style>.cls-1{fill:nor
```

Now, when the button is clicked, the programme loads the latest page in the browser's history. This is the most efficient way, as it doesn't matter where the settings menu was opened, it will always lead back to the previously opened page and so it works anywhere. If I decided to make it so that it opens a specific page, I would have to program cases for every time that it is possible for the settings menu to open, which would mean creating

some other sort of entrystate mechanism which, from what we have seen before, isn't so efficient.

Program output:

 settings-main menu test.mp4

Test No.	Test Description	Expected Outcome	Actual Outcome
37	Settings → Main menu navigation test	Pressing the back arrow in the setting menu should take you to the main menu	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: The base template

Section Review

Aims completed in this section:

- Create the full main design for the settings menu, including buttons and headers
- Create the ability to navigate between the settings and main menu
- Code the ability to switch between light/dark mode

Aims not completed in this section:

- Complete a success criteria item: Settings Menu
- Test light and dark settings for main menu and settings menu
- Allow the buttons to change relevant items in flight_data.json

Iterative tests complete:

- Buttons are operative. If colour is to change they do, and when clicked they trigger the correct output.

Remarks: Throughout the process of making this settings menu I did some research on how I could implement picking the units for the settings menu. This meant doing things with the so-called 'flight data file' that holds all the information- or so I thought.

To keep things brief, the research showed that having a .json file that changes with different inputs from the user just doesn't work. But the file isn't fully scrapped. It will still be stored in the user's device, but the 'live' file that I want to be constantly updating will have to be different. What's good however is that it can be done with a dictionary, and flask can read and write data from this, and also parse this into a .json file which can then be stored by the

user, and even the other way round, when it comes to importing data. More of this will be elaborated later in section 5.

It was at this point that I realised I didn't want to tamper with all the settings and do this all without testing it first and finding the suitable methods, and so I will wait until section 5 (where I do all the testing and find out how to work the local storage and updating systems) and then script the rest of the settings menu with ease. If I decided to keep on going with coding the settings menu it would most likely be heavily tailored to solve just the problem of picking units and thus it may not be efficient when it comes to using the code again for other purposes.

It is also for this reason that the dark mode testing for the main menu has not yet been completed because it involves checking what mode the user has selected which is saved in the local Storage too (as it needs to be read across all screens).

Section 4: The Base Template

Aims of this section

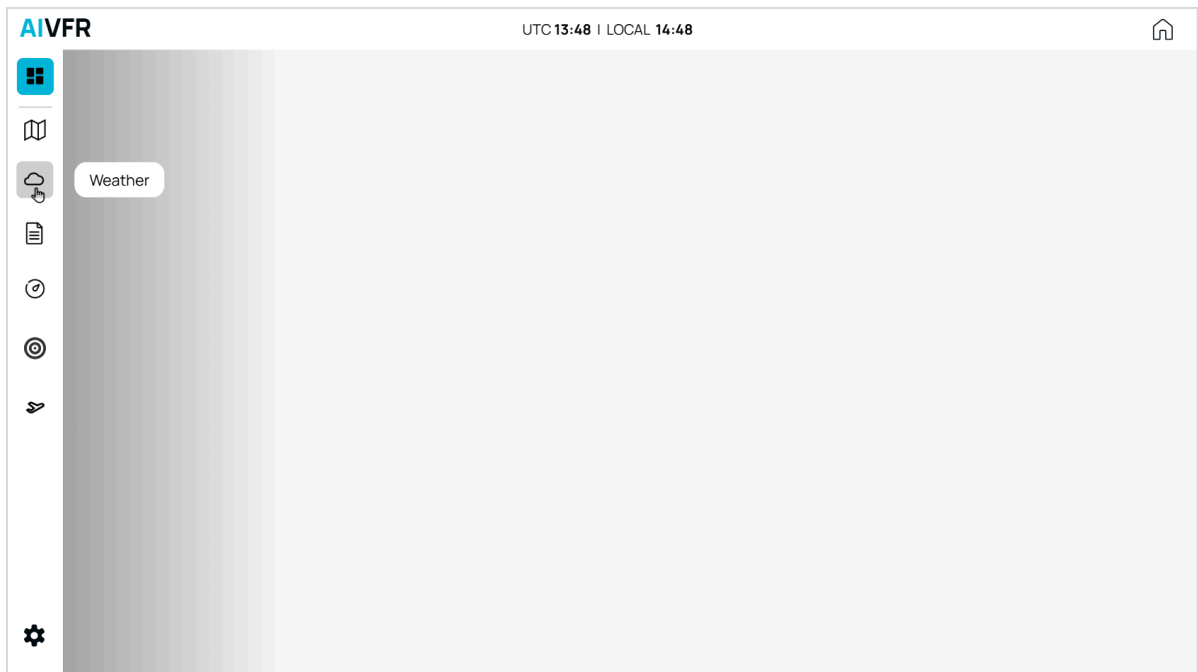
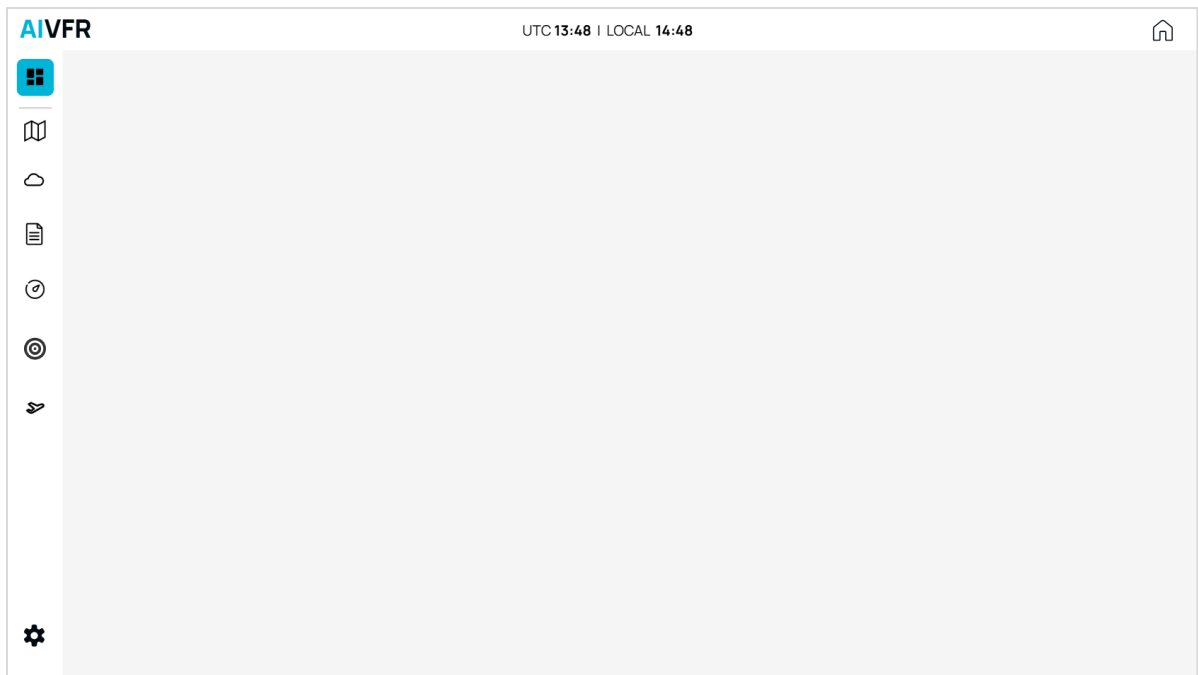
- To create the base elements that are present in all pages of the planning module
- To code the styling for selecting different pages
- Creating all pages and using HTML inheritance to have all pages share the same base template
- Code the ability to navigate through different tabs of the planning module using the navigation bar, including the main and settings menus

The base template is what will be present in all pages of the planning module, which essentially includes the navigation bar and the header at the top, with the remaining space being page-specific, I will also take this chance to create the styling involved with the navigation menu. This will be easier because there is no writing overlay that I need to worry about, and it should apply on all pages once finished.

The best way to do this is through HTML inheritance. This would essentially allow me to create one html page with the base template, and share it across all pages within the planning module. Then I can still add my own features to certain pages if I wanted to, such as the actual page content and the selected tab display on the navigation bar.

I could also code the html for the base template and copy it to every single page. This just unnecessarily increases the amount of code in my project and is very inefficient. It also makes the html code more complex to work inside due to all the base elements present within it already, but with inheritance, the coding space will appear less cluttered which makes the coding process more easier. Also, a major issue inheritance solves is that one edit made to the base template will change on all pages who have inherited the template, this means that if I made a mistake, I can make a small edit and it will apply to all pages, whereas, if I didn't use inheritance, I would have to make the same edits (which could potentially be complex) on every page which, with even more html written per page, could be very lengthy.

The base template should look something like what you see below, which is an edit of my design for one of the pages in the planning module, as you can see, just the navbar and the header is there, and this remains the same for all pages.



Iteration 1 - base.html setup and inheritance test

dashboard.html

```
{% extends "base.html" %} <!--grab all the html from base.html-->

{% block title %} <!--edit whats in the title block-->
    Dashboard - AIVFR
{% endblock %}

{% block content %} <!--edit what's in between the content block-->
    <h1>This is dashboard</h1>
{% endblock %}
```

base.html

```
<!DOCTYPE html>
<html lang='en'>
  <head>
    <!--Getting the Manrope font from google fonts-->
    <link rel="preconnect" href="https://fonts.googleapis.com">
    <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
    <link href="https://fonts.googleapis.com/css2?family=Manrope:wght@200..800&display=swap" rel="stylesheet">

    <meta charset='UTF-8' name="viewport" content="width=device-width, initial-scale=1.0" />

    <title>{% block title %}base{% endblock %}</title> <!--title can be changed in inherited pages-->
    <link rel="stylesheet" href="../static/base.css"/>
    <link rel="stylesheet" href="../static/components.css"/>
    <link rel="stylesheet" href="../static/basePage.css"/>
  </head>

  <body>
    <header>
      <p>header</p>
    </header>

    <nav>
      <p>nav</p>
    </nav>

    <main>
      {% block content %} <!--content in this element can be changed in inherited pages-->
      <p>content</p>
      {% endblock %}
    </main>
  </body>
</html>
```

What I did

I first begin by testing the inheritance process to see if it actually works as expected. First, I created the **base.html** base file and copied the boilerplate I used for the main menu to get the background and fonts loaded in.

Flask uses the **Jinja2** template inheritance system to allow HTML inheritance, and the majority of its just curly brackets and percentage symbols. When written in the base template that files must inherit from, it defines what places within the html file can be changed. For example, anything **in between** the `{% block content %}` and `{% endblock`

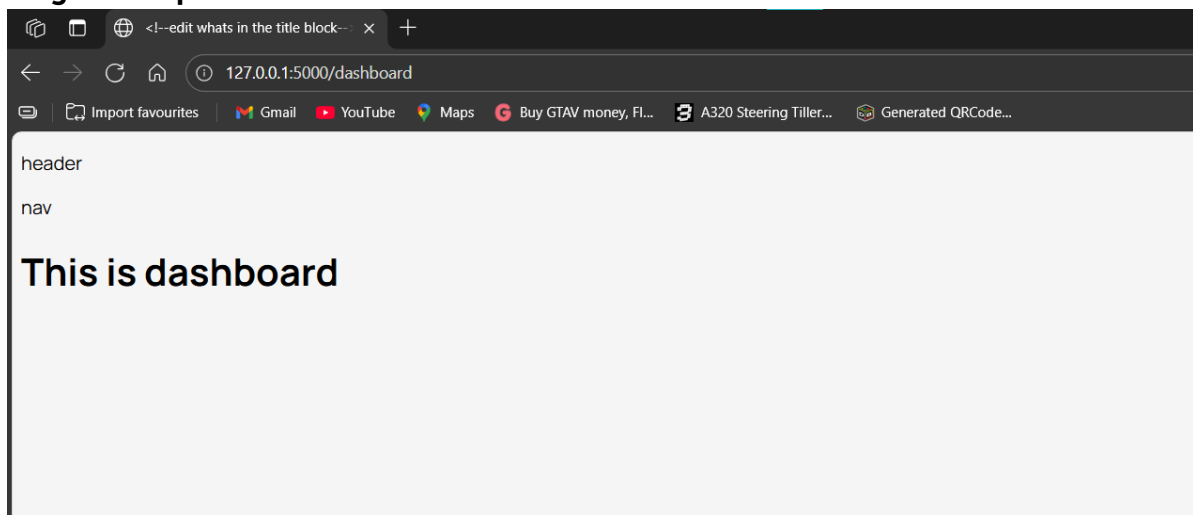
`%}` can be edited in the inherited page to be anything else. This is where the actual page content will be.

Then, I used the dashboard as the test page here. Since it should have inherited all the styling from base.html, I should only have to put what is necessary. I put the h1 'this is dashboard' as the test and I will see whether this shows up with the correct styling when tested.

Why I did it this way

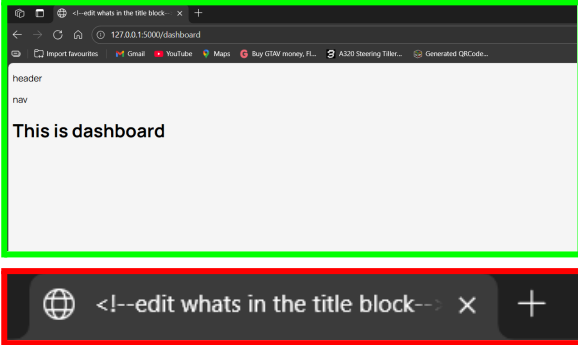
Testing to see if the system works correctly is obviously the best way to proceed when doing something new. I could've just coded the whole template without worrying about if it works or not, but if an error came up when tested at the end, it may be much harder to fix then than it will be now. A really good example of this is the logic error that has appeared in this iteration.

Program output



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
38	Jinja 2 inheritance test	Styling should apply Should see elements from both base template and the 'this is dashboard' heading	

Error Analysis & Fixes

Test No: 38

Failure Description: HTML comment appears in the title although it's a comment???

Cause: Quick research shows that Jinja2 html is its own language and turns out it has its own comment style as well, so it doesn't read simple html comments.

Fix (immediate): Replace the comments with jinja2 commenting syntax

<i>(Also installed jinja2 extension for better code clarity)</i> dashboard.html <pre>{% extends "base.html" %} {# grab all the html from base.html #} {% block title %} {# edit whats in the title block #} Dashboard - AIVFR {% endblock %} {% block content %} {# edit what's in between the content block #} <h1>This is dashboard</h1> {% endblock %}</pre>	
--	--

Justification: N/A (no alternative)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Base template creation: header

Iteration 2 - Base template header

```
<body>
  <header>
    <div>
      <!--Logo-->
      <span id="logoAI">AI</span>
      <span id="logoVFR">VFR</span>
    </div>

    <!--time-->
    <div>
      <p>UTC 13:48 | LOCAL 14:48</p> <!--Placeholder time to see visuals-->
    </div>

    <!--home button-->
    <div>
      <svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/xlink"
    </div>
  </header>

  <nav>
    <p>nav</p>
  </nav>

  <main>
    <!--{% block content %} content in this element can be changed in inherited pages-->
    <p>content</p>
    <!--{% endblock %}-->
  </main>
</body>
```

```

* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
  color: var(--text)
}

/*header to contain the Logo, the time and the home button*/
header {
  background-color: var(--highlight);
  display: flex;
  align-items: center;
  text-align: center;
  height: 50px;
  justify-content: space-between;
  padding-left: 20px;
  padding-right: 20px;
}

#homeButton {
  width: 30px;
  color: var(--text);
}

/*AIVFR logo*/
#logoAI {
  color: var(--primary);
}

#logoVFR {
  position: relative;
  right: 3px;
  color: var(--text);
}

#logoAI, #logoVFR {
  font-size: 25px;
  font-weight: 700;
}

```

What I did

To start off the header part, I used the same method as I did when it did the settings header, except this time flexbox is involved more evidently here. As you can see in the HTML, there are 3 divs within the header, and the use of flexbox allows me to space them equidistant from each other.

To make the logo, I just copied over the logo styling from the main menu and applied them here. Since the main-menu stylesheet is not linked to base.css, I used the same id names as it would not have an impact on the styling at all. This also just made it easier to read.

The home button was easy as expected, although since the main menu has no specific name (it's just '/' unlike '/dashboard') I'm not sure of the correct thing to put in the brackets to successfully open the main menu. I'll run some trial and error when I get to testing it and if that fails I'll search for the solution, however it's fine as it is now.

Finally, there's the time. There's obviously some extra stuff involved in this, and so I put a placeholder (from my design) in its place to simulate how it would look if it was already there, before I code it in later.

Why I did it this way

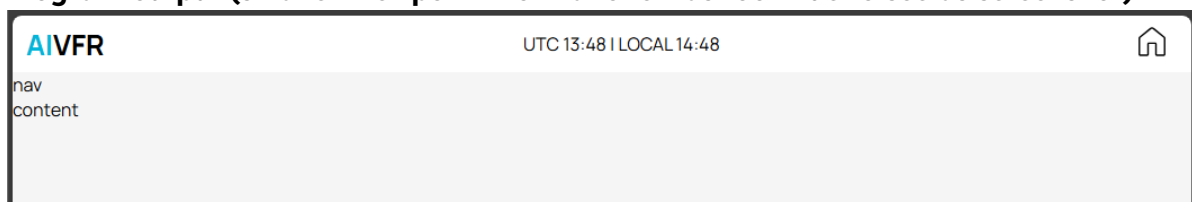
Without the use of flexbox, and say, I made position relative/absolute, I would most definitely struggle to make the div elements equidistant from each other, and the situation would be even worse when it comes to varying viewport sizes, flexbox was built to solve this problem exactly so In this case it is obviously the smarter and more efficient option.

I could have used different ID names for the logo, but since the code from the main menu still used the 'logoAI' and 'logoVFR' ids, changing the id would have me changing all its occurrences within the CSS file which is really unnecessary considering that it had no effect on the styling at all with just the same names.

The reason why I'm not testing the home button now is because I want to have a structure to the way I develop things. I first develop all the elements, then style them, then only after styling I move onto scripting, essentially its HTML → CSS → JS. This makes the coding process much more efficient because coding part of the JS when not all the css or html is done can make the coding process take longer than it should.

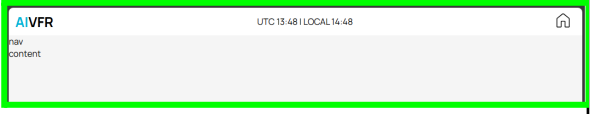
Note that this applies for certain sections, for example I've done the HTML and CSS for the header, now I can do the javascript for the header which includes the time and home button. I'll repeat the process for the navigation bar.

Program output (smaller viewport - normal one was too wide to see as screenshot)



Known Limitations / Missing features

- Time displayed is just a hardcoded placeholder
- Home button is inoperative

Test No.	Test Description	Expected Outcome	Actual Outcome
39	Header test	The header should appear similar to that shown in the design.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: The time

Iteration 3 - The Time and home

base.html

```
<div>
  <p>UTC <strong><span id="timeUTC"></span></strong> | LOCAL <strong><span id="timeLOC"></span></strong></p>
</div>
```

```
onclick="window.open('/', '_self')" style="cursor: pointer;"
```

basePlate.js

```
//-----Date and time-----
function updateClock() { //function allows just the time to update smoothly
  const now = new Date(); //gets current date and time

  //local
  const timeLOC = now.toLocaleString([], { hour: '2-digit', minute: '2-digit' }); //displays as hh:mm
  document.getElementById('timeLOC').textContent = timeLOC; //content of this (empty) element set to timeLOC

  //utc
  const timeUTC = now.toLocaleTimeString('en-GB', { hour: '2-digit', minute: '2-digit', timeZone: 'UTC' });
  document.getElementById('timeUTC').textContent = timeUTC;
}

updateClock() //when page loads for first time, run it immediately...
setInterval(updateClock, 1000); //...then update every second.
```

What I did

To get the time I followed a simple tutorial that explained the process. First, you'd change the time elements to span to ensure they're inline with the respective titles (UTC/LOCAL), and give them separate IDs. Then over in Javascript, you create a function and write the code to display the time within it.

To display the time, the `Date()` function gets the current date and time. Then all I had to do is edit it for my specific needs. For both times, setting the hour and minute values to '2-digit' made it only display the time as HH:MM, as per design. For the UTC time the timezone had to be set to UTC to adjust the fetched time to match UTC instead of local.

Finally, I selected the IDs from the html document of the elements whose content we wanted to replace with the time (now stored in `timeLOC` and `timeUTC`), and replaced them with those variables, which displays the time.

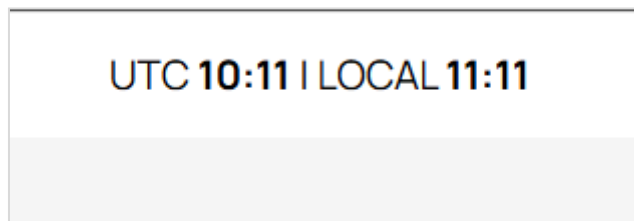
During this iteration I did the trial and error I mentioned last iteration to see what had to go in the brackets to display the main menu when the home button was clicked. Empty quotes didn't work, a blank space didn't work, then I tried just the forward slash as seen above, which proved successful.

Why I did it this way

Having the entire code in a function means that you can call the function only (which will update the time) without having to refresh the entire page over and over again which could potentially even clear data inputs which is not desirable. This is most likely what I will use for items such as weather re-fetching and such, as it avoids the same problems.

The choice of trial and error actually worked better in this situation, and was a quicker option than just searching for the answer online, because it was less than a line of code and the options available in this case were quite obvious compared to the previous times I've considered using this method of solving an issue. If I had no clue what I was doing or a lot of code was involved then this would be the call to do research instead of wasting time.

Program output



 Time update test.mp4

 base to Main menu test.mp4

Note that in the time video the title still includes the jinja2 syntax. This is because I was live viewing the html page only, without the use of flask. So any flask mechanics don't work hence the program displaying all that was in the title element as text.

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
40	Header test: Time	The UTC time and Local time both display correctly AND update accurately every minute.	PASS (See program output)
41	Base → Main menu test	Clicking the home icon should open the main menu.	PASS (See program output)

Iteration Review

Met requirements: /A

Changes: N/A

Next steps: Navigation bar

Iteration 4 - Navigation bar

base.html

```
<!--Navigation bar-->
<nav>
  <ul>
    <!--dashboard--><li><button id="dashboard"><svg xmlns="http://www.w3.org
    <hr size="3.5" color="#cdcdcd" noshade>
    <!--Route--><li><button><svg xmlns="http://www.w3.org/2000/svg" xmlns:xl
    <!--weather--><li><button><svg xmlns="http://www.w3.org/2000/svg" fill="
    <!--navlog--><li><button><svg xmlns="http://www.w3.org/2000/svg" fill="c
    <!--fuel--><li><button><svg xmlns="http://www.w3.org/2000/svg" fill="cur
    <!--M&B--><li><button><svg xmlns="http://www.w3.org/2000/svg" fill="curr
    <!--performance--><li><button><svg xmlns="http://www.w3.org/2000/svg" fi
    <!--settings--><li><button id="settings"><svg xmlns="http://www.w3.org/2
  </ul>
</nav>
```

basePage.css

```
ul {
  list-style-type: none;
  margin: 0;
  padding: 10px;
  width: 60px;
  height: 100vh;
  background-color: var(--highlight);
}

button {
  margin-bottom: 20px;
}

.icon {
  color: var(--text);
  width: 35px;
}

#settings {
  position: absolute;
  bottom: 10px;
}

hr, #dashboard {
  margin-bottom: 10px;
}
```


What I did

From the research I carried out on how to code a vertical navigation bar, it's essentially an unordered list, with all elements stacked onto each other. Since all elements for my navigation bar have button-like properties, all the icons .svg files have been put inside buttons as shown. This way I can easily apply button styling.

Each icon has been set to about the same size but some icons are a bit smaller by default (such as the cloud icon) and so they'll be adjusted manually.

Some changes I made here were to the actual icons displayed. You might have noticed that the icons for the fuel tab (previously represented by a gauge) and the mass and balance tab (previously represented as a target icon) have been changed to be a fuel dispenser and a centre of gravity icon/ symbol respectively.

Why I did it this way

For my case there was no other suitable option that allowed me to apply button styling to something that wasn't a button. The background of the actual .svg was a consideration but it didn't seem practical off the tongue nor does it seem feasible in practice. Encasing them in a button just makes styling much less of an issue and saves me time as I already have worked with them several times before in previous screens.

Regarding the icon changes, the former options were extremely vague, and so changing them better made it evident what page it was leading to. Without these alternatives, the design becomes counter-intuitive to the user which degrades user experience.

The gauge icon looked like any gauge and could've been mistaken for a speedometer which isn't related to fuel at all, and the target icon had absolutely no connotations to mass and balance, and the centre of gravity icon is a brilliant replacement given that we study a lot about moments and weight distribution, where we see this symbol a lot.

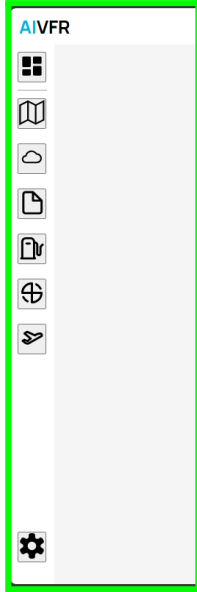
Program output

AI VFR



Known Limitations / Missing features

- Buttons are inoperative (except settings)

Test No.	Test Description	Expected Outcome	Actual Outcome
42	Navigation bar - positioning	Each icon button is positioned correctly with respect to the design	

Iteration Review

Met requirements: N/A

Changes: Changed fuel icon, CG icon

Next steps: button styling

Iteration 5 - Navbar: Buttons styling

basePlate.css (changes indicated by blue/green)

```
54  button {
55      margin-bottom: 25px;
56      background-color: var(--highlight);
57      border: none;
58      padding: 5px;
59      border-radius: 5px;
60
61  }
62
63  .icon {
64      color: var(--text);
65      width: 30px;
66  }
67
68  #settings {
69      position: absolute;
70      bottom: 90px;
71      width: 30px;
72      left: 14px;
73  }
74
75  hr, #dashboard {
76      margin-bottom: 9px;
77  }
78
79  hr {
80      position: relative;
81      bottom: 3px;
82  }
```

What I did

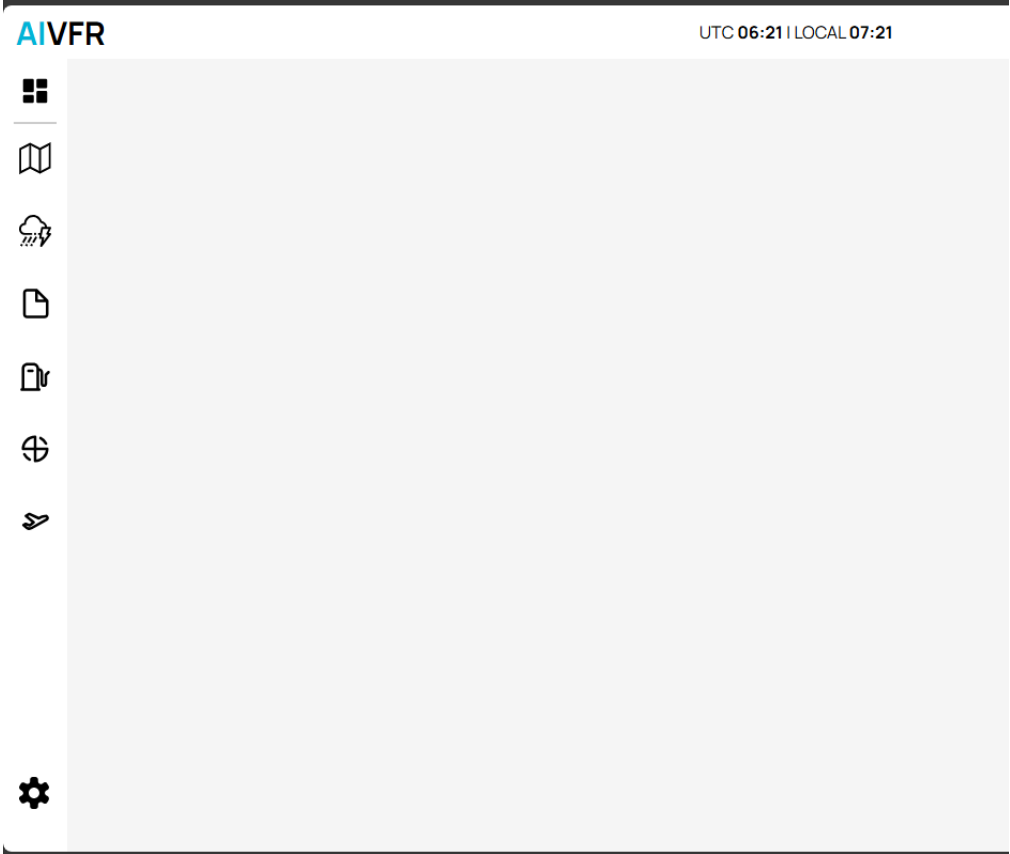
As I predicted before, styling the buttons was relatively simple. The background colour was changed to the same color as the navigation bar itself, and the borders were removed. As the html is also linked to components.css, the button hover effects were already applied so I didn't need to worry about that. I then added some padding and rounded the corners to match the design.

One thing I did change was the cloud icon. It was too small and when I tried to make it bigger it got big with the button, which was undesirable. I changed it for a much larger icon, and also took the chance to pick another icon in its place that perhaps depicted the tab a little better.

Why I did it this way

As explained before having the icons as buttons makes styling them a lot easier as you can see here. The change with the cloud icon was simply a design choice and **picking the exact same icon that was just bigger wouldn't have made a difference.**

Program output (not entire viewport)



Known Limitations / Missing features

- Buttons are inoperative (except settings)

Test No.	Test Description	Expected Outcome	Actual Outcome
43	Button appearance and styling test	buttons should look as they should as per design, and must have proper surrounding dimensions	A screenshot of the same AIVFR interface as above, but with a thick green rectangular border surrounding the entire content area, including the sidebar and the main gray rectangle.

Iteration Review

Met requirements: N/A

Changes: Weather: cloud icon changed to bigger cloud icon w/ thunderstorm.

Next steps: Selecting page overlay

Iteration 6: Selecting page overlay

Base.html

```
<!--Selection overlay-->
<nav id="NAVoverlay">
  <div class="label">Dashboard</div>
  <div class="label">Route</div>
  <div class="label">Weather</div>
  <div class="label">Navigation Log</div>
  <div class="label">Fuel</div>
  <div class="label">Mass & Balance</div>
  <div class="label">Performance</div>
</nav>
```

basePage.css

```
#NAVoverlay {
  position: fixed;
  padding-left: 15px;
  margin-left: 60px;
  width: 350px;
  height: 100vh;
  padding-top: 8px;
  border: 0.1px solid black;
}

.label {
  background-color: var(--highlight);
  border-radius: 14px;
  max-width: fit-content;
  padding: 10px;
  padding-left: 17px;
  padding-right: 17px;
  margin-bottom: 25px;
}
```

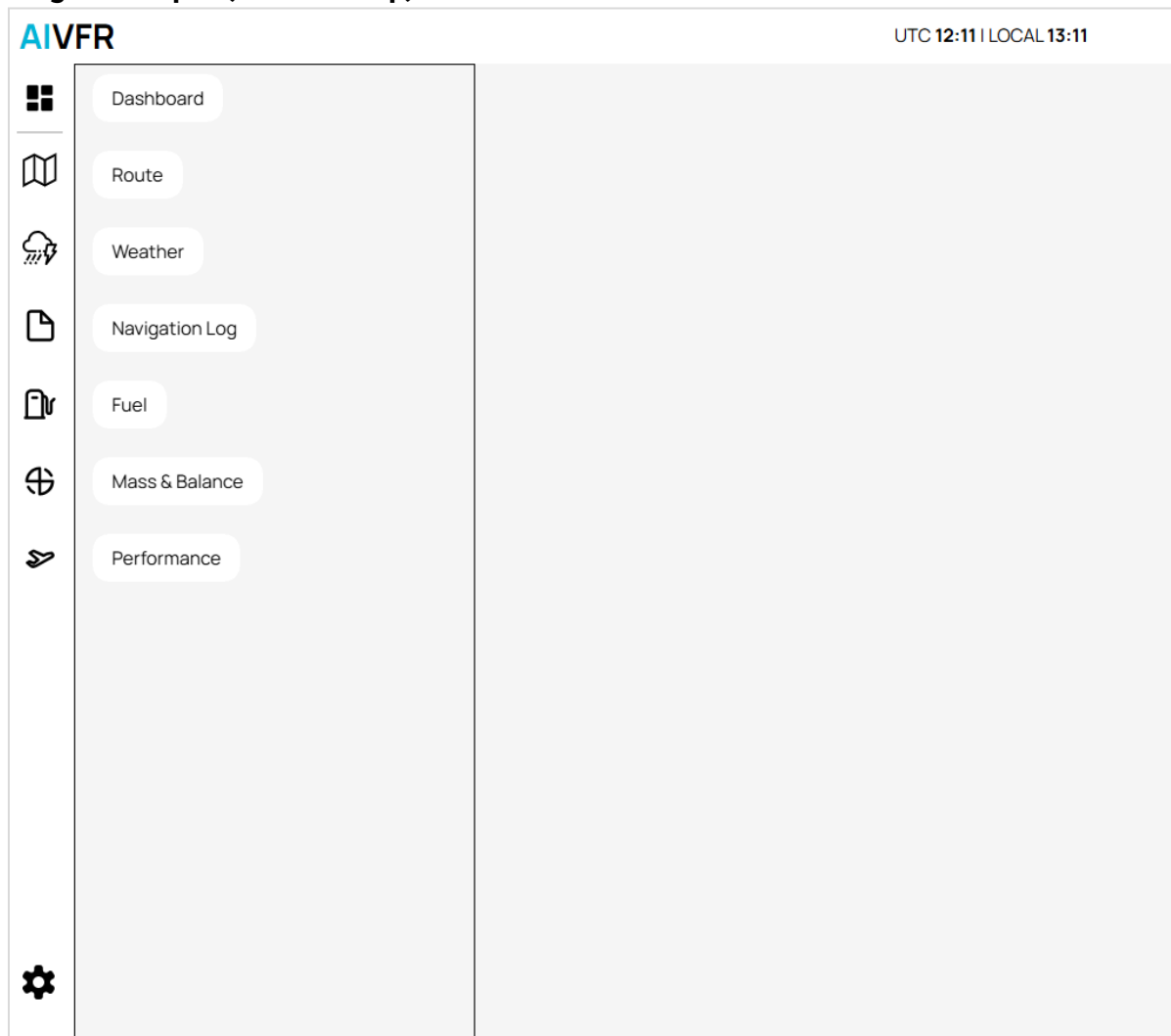
What I did

The method I'm taking to make the selection overlay is essentially to have a square box located next to the navigation bar that has a gradient from black to transparent, with buttons then appearing on top of it to make them visible. This is exactly what I've done here, with all the buttons contained in a separate nav box that's the size of the gradient block. The border helps to see what I'm working with prior to the actual gradient styling.

Why I did it this way

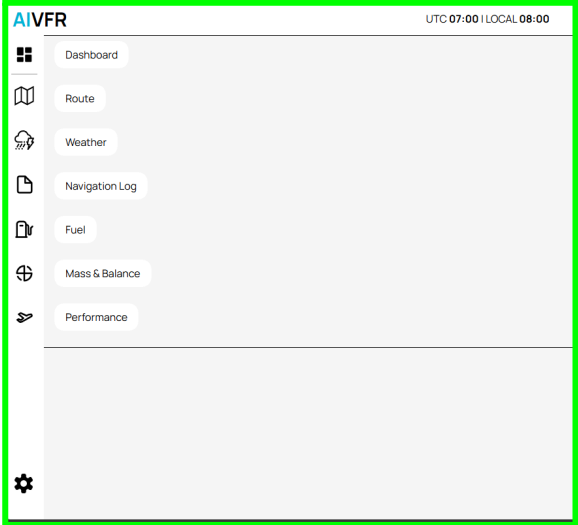
There were most likely many ways I could have done this, but I believe this is the most feasible way (perhaps the only way) as it provides the most flexibility with the gradient to be applied. The box outline is extremely important in showing the boundaries before I get to the styling part. Without it, I would get an incorrect measurement which would make me apply the wrong gradient as I would think it needs to stretch across more of the screen when in reality it's supposed to cover just a small section.

Program output (not entire vp)



Known Limitations / Missing features

- Tab buttons are inoperative

Test No.	Test Description	Expected Outcome	Actual Outcome
44	Label and gradient box positioning test	Labels are aligned with their respective icons, the gradient box is 350px wide as per design and covers a suitable space across the left hand side of the screen.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: gradient and toggling

Iteration 7 - Gradient and toggling

basePage.css

```
#NAVoverlay {
  position: fixed;
  padding-left: 15px;
  margin-left: 60px;
  width: 350px;
  height: 100vh;
  padding-top: 8px;
  background-image: linear-gradient(to right, rgba(0,0,0,50%), rgba(0,0,0,0%));
}

.label {
  background-color: var(--highlight);
  border-radius: 14px;
  max-width: fit-content;
  padding: 10px;
  padding-left: 17px;
  padding-right: 17px;
  display: none;
}
```


basePage.js

```
//Hovering over a tab when selecting-----
//pair each button to their label (as I didn't code them inline with eachother)
const pairs = {
  dashboard: "dashboardLABEL",
  route: "routeLABEL",
  weather: "weatherLABEL",
  navlog: "navLABEL",
  fuel: "fuelLABEL",
  "m&b": "mass_and_balanceLABEL",
  performance: "performanceLABEL"
};

for (let buttonId in pairs) {
  const button = document.getElementById(buttonId); //assign each button with button variable
  const LABEL = document.getElementById(pairs[buttonId]); //assign each label with LABEL variable

  if (button && LABEL) { //For each matching pair run an event listener for it
    button.addEventListener("mouseenter", () => {
      LABEL.style.display = "block";
    });
    button.addEventListener("mouseleave", () => {
      LABEL.style.display = "none";
    });
  }
}
```

What I did

The gradient, much to my surprise, was just a single line of code to create, and I managed to do it exactly how I wanted to. The 'to right' value changes its direction from left to right, and I used rgba values to adjust the opacity of the other side of the gradient to zero, so it fades out nicely into the background.

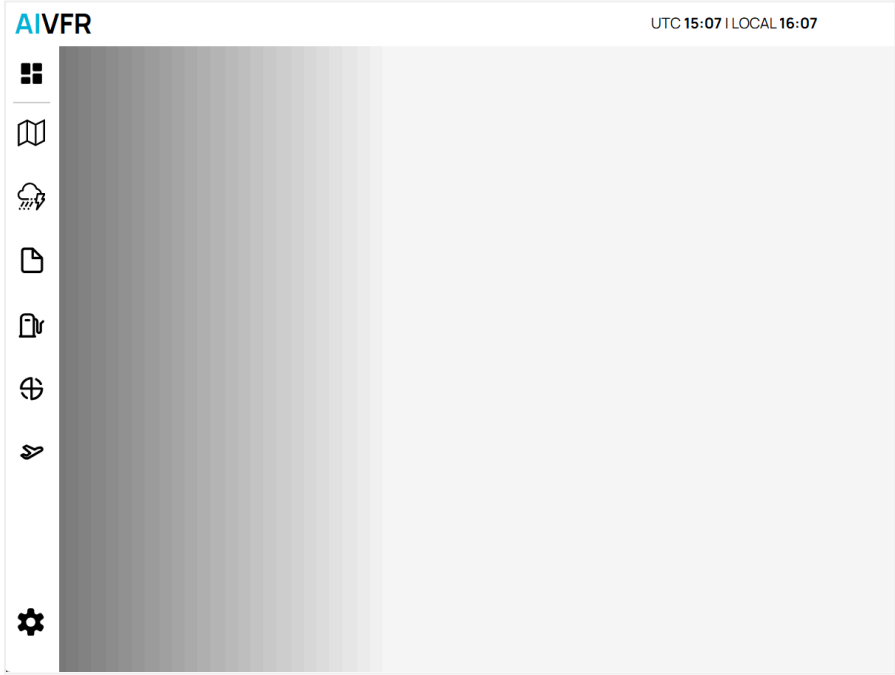
The harder part was trying to make the labels appear properly. What the tutorials were saying only applied button-label pairs that were created inline of each other, which these weren't. The actual buttons are inside the nav bar and to make their labels inline I would have to move them inside the navbar which immediately brought up formatting errors as I expected. So instead I found another solution that allowed me to still utilise elements separated from one another.

By pairing each button with their label as shown above, everything became easier, and I was surprised I never had thought of finding a way to simply link the two elements together. For the toggling code, a for loop assigns each button and pairs the variables 'button' and 'LABEL' respectively, then attaches an eventListener to each button-label pair. If it detects the mouse hovering over the button, the display is set to block, and then back to none when no hovering is detected.

Why I did it this way

It perhaps may have been more efficient to re-write my code (and therefore the styling due to the aforementioned formatting errors) to make the elements inline with each other, and I did think of this. However, I did abstain from doing it because it was simply too time consuming, and I was convinced that there was still a workaround for my specific situation, and it turned out to be exactly that. This is one of the things I may change in future updates for better code quality, but for now, I think the current layout is perfect and works exactly how I wanted it to, bar some minor positioning error that is shown below.

Program output (not whole vp)



📺 Labelling appearance test.mp4

Known Limitations / Missing features

- Tab buttons are inoperative

Test No.	Test Description	Expected Outcome	Actual Outcome
45	Gradient test	The gradient should start at a dim black and fade out into the background as per design. There should be no visible sign of a box outline on the faded end of the gradient.	A screenshot of the AIVFR application interface, identical to the one above, but enclosed within a green rectangular border. The top bar shows the logo 'AIVFR' on the left and the time 'UTC 15:07 LOCAL 16:07' on the right. Below the top bar is a sidebar with several icons: a grid, a book, a cloud with rain, a document, a folder, a magnifying glass, a hand, and a gear. The main area of the interface displays a horizontal gradient bar that transitions from dark grey on the left to light grey on the right.

46	Label appearance test.	When the mouse hovers over a specific button, its label should appear adjacent to the button and should disappear when the mouse moves away.	
----	------------------------	---	---

Error Analysis & Fixes

Test No: 46

Failure Description: All labels appear at the top as though all squashed in one place.

Cause: 100% has to be a positioning error of some sort. It looks like the labels when hidden don't stay fixed because their position is relative to themselves and when they disappear they can just move upwards freely whilst still maintaining the relative properties.

Fix: Try making the position absolute to see if they remain fixed. If so then position them again as necessary.

Justification: Absolute position is the best shot at the moment because it uses positioning relative to the parent element, which means they should in theory move and stay where they are as their position is relative to the actual content box not themselves where they can still move freely - which is what would be the case for any other position value.

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Debug and finish selection overlay

Iteration 8 - Selection overlay completion

basePage.css

```
/*adjusting the label's absolute positions*/
#dashboardLABEL { top: 10px; left: 10px; }
#routeLABEL      { top: 75px; left: 10px; }
#weatherLABEL    { top: 140px; left: 10px; }
#navLABEL         { top: 210px; left: 10px; }
#fuelLABEL       { top: 278px; left: 10px; }
#mass_and_balanceLABEL { top: 346px; left: 10px; }
#performanceLABEL { top: 414px; left: 10px; }
```

basePage.js (edited to include Nav overlay)

```
const gradient = document.getElementById("NAVoverlay");

if (button && LABEL) { //For each matching pair run an event listener for it
  button.addEventListener("mouseenter", () => {
    LABEL.style.display = "block";
    gradient.style.display = "block";
  });
  button.addEventListener("mouseleave", () => {
    LABEL.style.display = "none";
    gradient.style.display = "none";
  });
}
```

What I did

As I predicted, the issue regarding the appearance of the labels was caused by a positioning error. When I switched the positioning to absolute, the labels stayed where they were. However, this did mean that they'd still all start from the corner and so I had to physically move them so they aligned with their respective buttons.

Then, to complete the selection overlay I needed to give the gradient background also toggle with the labels. This was as simple as adding the style.display = 'block/none' command to toggle the gradient on and off at the same time a label is toggled on or off.

Why I did it this way

For this iteration, the debugging was justified prior and there were no feasible alternatives to the toggling options for the background, so there isn't much to say here.

Program output

 Selection hover test.mp4

Known Limitations / Missing features

- Tab buttons inoperative

Test No.	Test Description	Expected Outcome	Actual Outcome
47	Label and gradient appearance test	When the mouse hovers over a specific button, its label should appear adjacent to the button and should disappear when the mouse moves away. The gradient box should also behave the same.	PASS (See program output)

Iteration Review

Met requirements: (Aim complete) To code the styling for selecting different pages

Changes: N/A

Next steps: Creating pages and coding the buttons (and coloring)

Iteration 9 - Coding buttons, page creation and button coloring

dashboard.html

```
{% extends "base.html" %} {% grab all the html from base.html, #}

{% block head %} {{ super() }} {% inherit everything from the head but add this too #}
<link rel="stylesheet" href="{{ url_for('static', filename='dashboard.css') }}">
<script src="{{ url_for('static', filename='dashboard.js') }}" defer></script>
{% endblock %}

{% block title %} {% edit whats in the title block #}
    Dashboard - AIVFR
{% endblock %}

{% block content %} {% edit what's in between the content block #}
    <div id="title">DASHBOARD</div>
{% endblock %}
```

dashboard.css

```
#title {
    font-size: 29.8px;
    font-weight: 300;
}
```

What I did

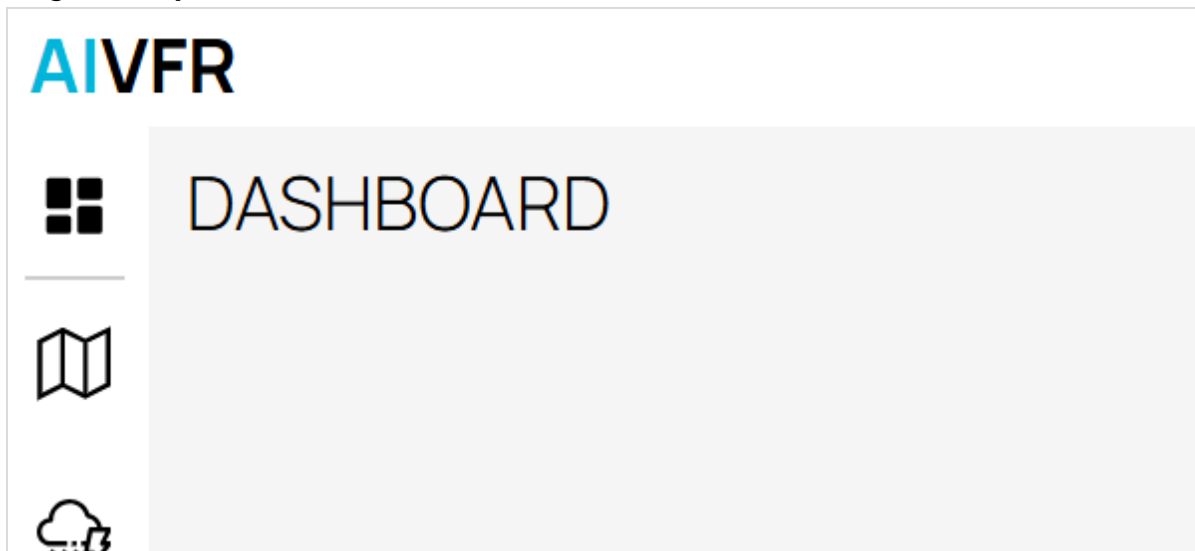
Before I started coding the buttons and creating the pages that link to them, I needed to do some tests with the single dashboard.html page that I had. This again was to avoid any potential problems I may encounter with other pages, that would mean I need to change the same thing several times. The tests I wanted to do tested whether the css file had been linked correctly to dashboard.html. Similar to python OOP, the super() function allows me to inherit everything from a particular block but it also allows me to add something onto it as well, without clearing the entire block.

After linking the css sheet I tested it by using the new **dashboard.css** file to style the title as per the design, and adjust the margins to fit in the free space. That way I'm still making progress of some sort.

Why I did it this way

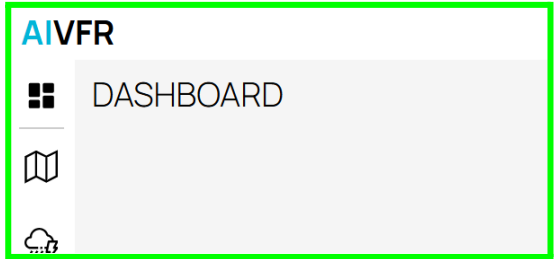
Deciding to test one page was useful in this iteration, as I initially made a syntax error by linking the CSS stylesheet using normal html syntax. *If I had copied this template to all the other pages I would have had to change them one by one which is a waste of time.*

Program output



Known Limitations / Missing features

- Tab buttons are inoperative

Test No.	Test Description	Expected Outcome	Actual Outcome
48	CSS Linking test	Styling that was applied to dashboard.css should apply to the text, and the title should appear as it would in the design.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Creating and linking pages and coding the buttons

Iteration 10 - Linking pages, coding buttons and color styling

basePage.js

```
//-----selecting another page-----  
button.addEventListener("click" , () => {  
  if (buttonId == "mass_and_balance") {  
    window.open("mass-and-balance", '_self') //if statement avoids syntax URL error  
  } else {  
    window.open(`${buttonId}`, '_self') //uses the button ID which matches the webpage name  
  }  
});
```

Fuel.css (similar for all tab pages)

```
#fuel {  
  background-color: var(--primary);  
  pointer-events: none;  
}
```

What I did

Firstly, I created all the css and JS files for all the other tabs. They all have similar names (eg. fuel.css, fuel.js). And copied the title code done in the last iteration into each css file, and changed the id name accordingly. After testing this worked as expected, I then created the rest of the html files and copied that inheritance code from the last iteration into each HTML file. I then went over to **app.py** and linked those html files to the necessary path for that webpage, just like I did for the dashboard. It's a lot of files and code with not a lot of impact and so I have decided not to show them here.

The first important thing I did afterwards was to find a way to have a button retain the primary colour when their page is selected. Initially I was worried this may be difficult because the colours are specific to the tab and I thought styling would just be either long (coding js for each tab) or just not possible.

However, once I realised that since the basePage.css is also linked to each other html file as a consequence of inheritance, I tested changing the color of the button using each page's css file, and it worked, which was relieving. Then, to stop the label from still showing up I made the button unclickable by setting the `pointer-events` attribute to none.

Finally, to actually allow me to actually switch to the page on a button click I utilised the fact that the button ids are used to show the button labels. So, by attaching an event listener to the buttons to listen for a click, and then making it open the page that's the

current buttonID (as the html pages match the spelling of the button ids) I successfully made switching pages possible without a lot of repeated JS.

One error that did show up was that 'mass_and_balance' was not a correct url (it was 'mass-and-balance' with hyphens instead of underscores), so used an if statement to make sure that if its the mass and balance tab is to be loaded, the correct spelling is used.

Why I did it this way

At some points during this iteration, alternative options always resulted in erroneous results. Examples include the coloring system, which, if I coded in JS like I did in the settings tab, the colours would literally reset when a new page was selected, which was not desirable.

When coding the JS, the main issue was to avoid having inefficient code. I could have just assigned each button with a variable and attached an event listener to it, but when I noticed this had already happened in the previous function I used it to my advantage, which not only made the process quicker but made for higher quality, more efficient code.

Program output

 Base plate testing final.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
49	Colouring and page switching pages test	Pressing a tab button should switch to that page. Also, the colour of the button should be the primary color, and it should stay like that and the button should be unclickable.	PASS (See Program output)
50	Full Base template functionality test	All elements of the base template should operate correctly with no errors of any sort.	PASS (See program output)

Iteration Review

Met requirements:

- (Aim complete) To create the base elements that are present in all pages of the planning module
- (Aim complete) Creating all pages and using HTML inheritance to have all pages share the same base template
- (Aim complete) Code the ability to navigate through different tabs of the planning module using the navigation bar, including the main and settings menus

Changes: N/A

Next steps: Section 5: Flight data and the cache

Section Review

Aims completed in this section:

- To create the base elements that are present in all pages of the planning module
- To code the styling for selecting different pages
- Creating all pages and using HTML inheritance to have all pages share the same base template
- Code the ability to navigate through different tabs of the planning module using the navigation bar, including the main and settings menus

Aims not completed in this section:

N/A

Iterative tests complete:

- Base template inheritance testing: Once the base template is made, other pages that must inherit from it should do so with no discrepancies between pages.
- User should be able to switch between tabs seamlessly and correct hover effect to be shown

Remarks: Nothing to address.

Section 5: The flight data and the Session

Aims of this section

- Complete 2 success criteria items: **Ability to save, edit and start new flight plans, Settings menu**
- Find out how to store data in local storage/Cache
- Create a method of writing and reading data from a session
- Perform tests on storing and reading input data via the session and outputting the read data on the screen (text input testing)
- Perform practical testing of this system with light dark toggling being saved and maintaining the theme throughout each screen
- Perform practical testing of this system by implementing units selection in the settings menu and coding one function to test changing the units of an inputted number value.
- Perform practical testing of this system by implementing the checkSave algorithm and clearing the session when the user selects 'New Flight'
- Testing making the session file into a .json file and saving it to the user's device, then using a saved file to load in a flight from the device (load flight button functionality)

This section is arguably going to be one of the most challenging ones (the route tab being the other because of the map, AI, etc), because of the limited information I have about

local storage and the 'session' in which data is stored whilst the program is running. But I do have a gist and that should be enough to make a start.

The aim of this section, in short words, is to create the **memory system** for the webtool. This will be used in all tabs to store the flight data and later turn it into a .json file for the user to store. There are many situations where the memory might be used and so I must account for and test for them.

The purpose of all of this is so I don't need to think about this when coding things like the route section, I can just do the HTML, CSS and JS and not worry much about the backend processing because I already know what it will be like. As you can imagine, if I waited to do this whilst doing the other sections it would prove to be extremely difficult and will dramatically slow down the progress of this project.

To make the backend for this project work, I would need 3 key features to be working properly, and these form part of the success criteria item which is the aim to complete in this section:

"Ability to save, edit and start new flight plans"

1. "Save" - Ability to Export data
2. "Edit" - Ability to Import **and** Update existing data, and display said updates
3. "Start New" - Ability to clear existing data and create a new file.

If I know how to do each of these, then I will be happy to move onto the planning module. This is because these are the key aspects that make this webtool usable, which is why it is a success criteria item. Without proper knowledge of how these work then time will be wasted trying to find solutions in the planning module, as stated before.

Iteration 1 - Initialising flight data and the session

A bunch of research was done to get to this point, so the code below is after I have a good understanding of Flask session and how to set it up.

__init__.py

```
app.secret_key = os.getenv("SESSION_KEY")
```

.env

```
SESSION_KEY=641b8bec2a99832a234c130b90...HEX code continues
```

app.py

```
#main menu
@main.route('/') #this will be the first to load up
def index():
    if "flight_data" not in session: #initialise the session if not created already
        session["flight_data"] = data_template.copy()
    return render_template('menu.html', data=session["flight_data"])
```

```
#debug route that allows me to see the flight data at any time
@main.route('/debug')
def debug_session():
    return jsonify(session.get("data", {}))
```

```
#-----flight data and session management-----

#making the template to store flight data during a session
#also allows setting default values

data_template = {
    "settings" : {"theme": "light",
                  "units_airspeed" : "knots",
                  "units_altitude" : "feet",
                  "units_mass": "kilograms",
                  "units_fuel" : "litres",
                  "units_distance" : "nautical_miles"},
    "flight" : {
        "departure" : "",
        "departure_name" : ""
    }
}
```

What I did

Flask session allows me to store current Flask session allows me to store user-specific data across requests using cookies, and is the tool that I will use to store the user's current flight plan and all its data. I've decided to use a client-side session for this project- or at least in development, as I don't expect the data file to get too large.

To initialise the session, a session key is required, which is a random hex code that you can choose. I used a hex generator to do this and stored it in the .env file so it's not picked up as a secret element by git (so it wouldn't commit). When the session is initialized, it copies the data in the data template onto the session, and renders it along with the menu.html file. Now this data can be accessed by flask and many things can be done to it.

The data template is a dictionary which, as I planned, contains every single detail about the user's flight, and even the settings they've used. This also allows me to set defaults for the units, which are all set to imperial measurements as the target user base is for pilots in the UK.

Now I also needed a way to see this data in the background, because, whilst it is supposed to be abstracted from the user, it needs to be seen by me so that I can view its contents during development. The solution was simple- make a separate 'debug' page that I can manually go to which displays all data within the flight_data dictionary.

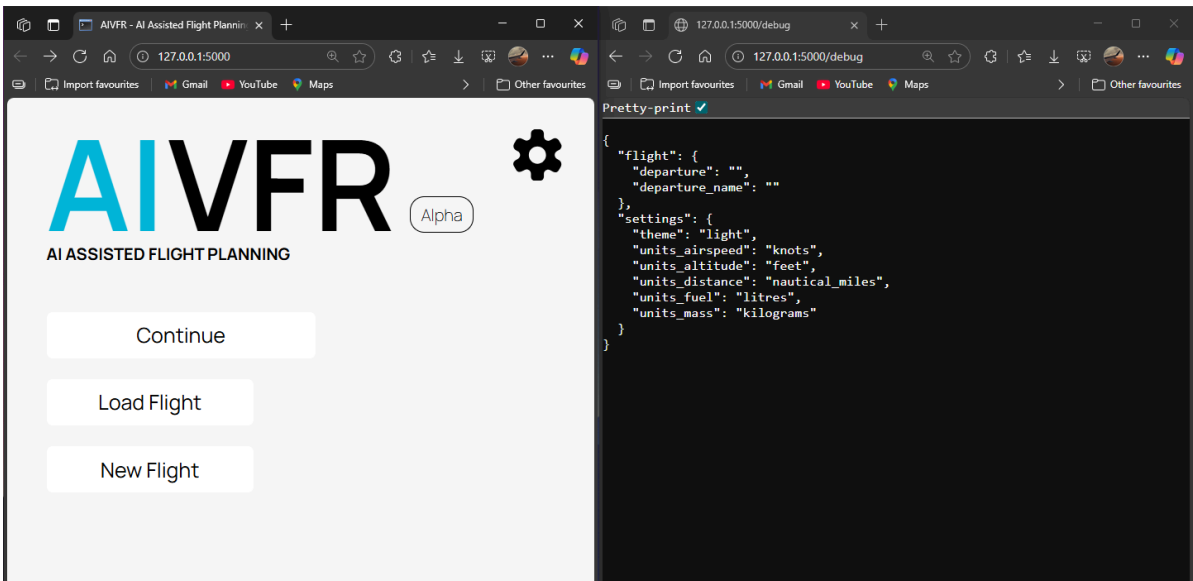
Why I did it this way

I elaborated in section 3 why I was going to use a tool like this instead, rather than just play around with .json files- it's because you simply cannot, especially in flask. You can only change data (such like that in a dict) into a json using **.jsonify**. This actually suited my needs perfectly, much more than the alternative I planned before and so this method must be the most feasible to use.

Using a server-side session instead of a client side session comes with a main advantage of allowing you to store large data files, but that comes at the cost of more complex coding that would take a while to learn as it is all beyond my scope. Also as I mentioned before these files will only take up perhaps a few KB when completely full, so client-side sessions

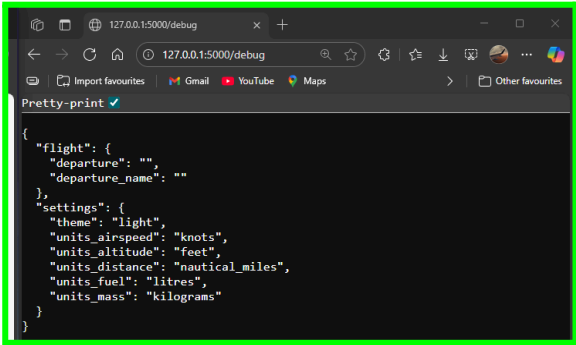
are the better pick for the current state of the project. If the whole system begins to slow down overtime then this is obviously a sign to switch to a different type.

Program output



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
50	Flask session initialisation test	A new session is created and the template is copied onto the session and can be viewed by viewing the /debug page	

Iteration Review

Met requirements: (Aim complete) Find out how to store data in local storage/Cache

Changes: N/A

Next steps: Updating data

Iteration 2 - Updating data

app.py

```
#-----UPDATING/MODIFYING DATA-----

#Updating Settings data
@main.route("/update-settings", methods=["POST"])
def update_session():
    key = request.json.get("key")
    value = request.json.get("value")
    session["flight_data"]["settings"][key] = value
    session.modified = True # To check if change happened successfully
    return jsonify(session["flight_data"])

#Updating flight data
@main.route("/update", methods=["POST"])
def update_session():
    key = request.json.get("key")
    value = request.json.get("value")
    session["flight_data"]["flight"][key] = value
    session.modified = True # To check if change happened successfully
    return jsonify(session["flight_data"])
```

settings.js

```
//--DARK MODE--
darkMode.addEventListener("click", () => {
    root.classList.add("dark-mode");
    var r = new XMLHttpRequest(); //allows sending HTTP requests (eg POST)
    r.open("POST", "http://127.0.0.1:5000/update-settings", true);
    r.onreadystatechange = function () {
        if (r.readyState != 4 || r.status != 200) return;
        console.log("sent"); //confirm that message was successfully sent
    };
    r.send(JSON.stringify({ "key": "theme", "value": "dark" }));
});
```

What I did

Now, moving onto the 'edit' part, I wanted to perform a test on something already present in the webtool, namely the settings menu. This makes it easier as, if I can do it successfully, then I have found a solution to updating the session but have also completed part of what is necessary to complete the settings menu, which makes developing a lot more efficient.

Starting small, I decided to test updating the "theme" key in the flight data to say 'dark' instead of 'light' when the dark mode button was pressed. This would allow me to find out how to send messages to flask when a button is pressed. The solution involved using the flask HTTP method 'POST', which allows you to send data to flask. Over in JS an XMLHttpRequest function is made to allow this. In app.py, I made both the routes for

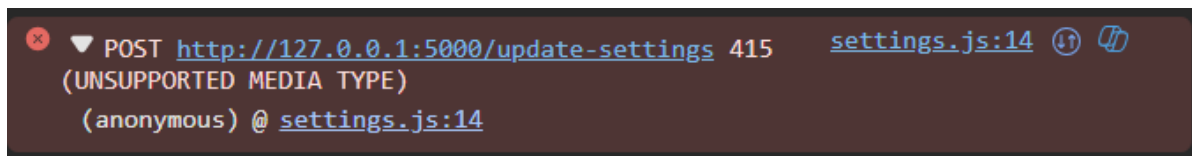
updating settings and the normal flight data. Again I had to specify that the 'POST' method is what was being used here, and request the sent keys and values, and then use those keys and values to update that which was in the session.

Why I did it this way

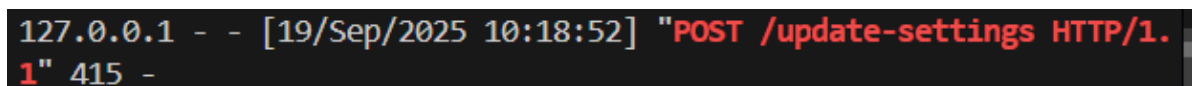
Arguably, a more rational approach to coding these items was to create a separate HTML page and test it from there. The main reason I didn't want to do this was because I wanted to avoid writing more HTML and instead I just wanted to focus on the JS aspect as that's where there is a link to flask and its components. Moreover, the solution I may find in the separate test may not be feasible to fit in the already existing HTML pages and code structures, which could prove to be a waste of time if I had to re-write the code to tailor my existing code.

Program output

Console



Terminal



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
51	Updating session test	When pressing 'dark' button in settings the "theme" value in the session should switch from 'light' to 'dark'	

Error Analysis & Fixes

Test No: 51

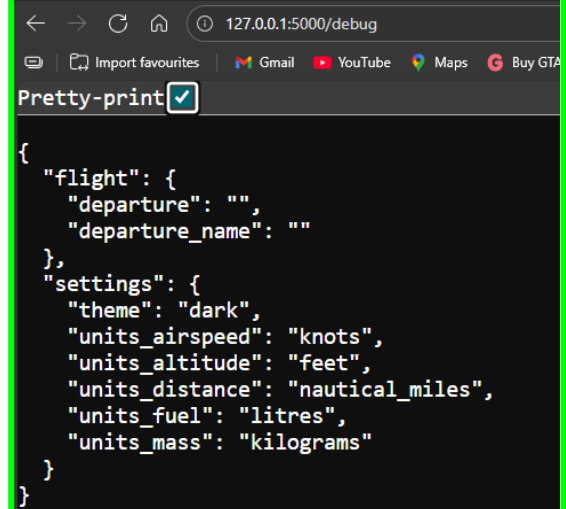
Failure Description: Error 415 - Unsupported media type, which is defined as: 'The server refuses to accept the request because the payload format is in an unsupported format.'

Cause: In regards to flask, the primary reason for this error is that there is an incorrect format of the Flask request. Upon searching the correct format, it looks like I was missing a key line of code, that tells flask that It's JSON that I'm sending.

Fix (immediate): Add missing line of code to fix request formatting.

settings.js

```
r.setRequestHeader("Content-Type",  
"application/json");
```



```
{  
  "flight": {  
    "departure": "",  
    "departure_name": ""  
  },  
  "settings": {  
    "theme": "dark",  
    "units_airspeed": "knots",  
    "units_altitude": "feet",  
    "units_distance": "nautical_miles",  
    "units_fuel": "litres",  
    "units_mass": "kilograms"  
  }  
}
```

Iteration Review

Met requirements N/A

Changes: N/A

Next steps: Light/Dark mode integration

Iteration 3 - Light/Dark mode integration

app.py

```
#-----READING DATA-----  
  
@main.route("/get-settings", methods=["GET"])  
def get_settings():  
    return jsonify(session["flight_data"]["settings"])  
  
@main.route("/get-flight", methods=["GET"])  
def get_flight():  
    return jsonify(session["flight_data"]["flight"])
```

settings.js, main_menu.js

```
//-----Check for theme-----  
const root = document.documentElement  
fetch("http://127.0.0.1:5000/get-settings") //fetches the current session data  
.then(response => response.json()) //turns it into a .json file so it can be read  
.then(settings => {  
    if (settings.theme === "dark") {  
        root.classList.add("dark-mode");  
    } else {  
        root.classList.remove("dark-mode");  
    }  
});
```

What I did

Now that I've managed to change data, we can move on to reading this data. The aim here is to allow other screens to see that the setting is set to dark and so load up in dark mode. To do this I used the GET method, whose functions in app.py are meant to return either the flight data or the settings data. For light/dark functionality I used the get_settings function which gets the settings data, and then over in settings.js I followed the syntax from fetching information from a flask GET request, which requires you to call the route.

Once fetched, a simple if statement can check whether the theme is set to dark or light and act accordingly. I then pasted this same code onto main menu.js to test if it shares across pages. Of course I didn't know if this would have worked as expected so I only used the main-menu and never applied this code to all the other pages.

Why I did it this way

This method is effective because it allows not just the settings page to be affected by light/dark mode. Instead, any page can read the data over in the dictionary to see what the current theme is and then load according to that. This in reality is the only feasible way to change the theme and stick with it across all pages.

Program output

📺 light-dark test 2.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
52	Light/dark sharing test	The main menu should be in dark mode when loaded up after switching to dark mode via settings. Settings should also stay in dark without resetting on reload.	ERROR - Screen still takes a second to switch to dark

Error Analysis & Fixes

Test No: 52

Failure Description: Flash of unstyled content error when reloading any page in dark mode.

Cause: Most likely because the entire page (with all its colouring) is rendered first before the javascript is read. So afterwards the JS is read then the style changes.

Fix: Check and apply dark mode before the page is rendered in, via the actual render template code in app.py.

Justification: This is the only logical solution because, when considering JS, all JS code will run after the page has been rendered in with its styling. This automatically means that there can be no alternative solution that involves the use of javascript. Moving onto app.py to find a solution, it became obvious that if I could render the theme in before the page is even rendered, then this should make the dark theme apply immediately.

Iteration 3.5 - Fixing unstyled content flash

To allow the theme to render first the settings data in the session had to be loaded in at the exact moment the page was being rendered in. So this is what was put into the render_template function.

app.py

```
#main menu
@main.route('/') #this will be the first to load up
def index():
    if "flight_data" not in session: #initialise the session if not created already
        session["flight_data"] = data_template.copy()
    return render_template('menu.html', data=session["flight_data"], settings=session["flight_data"]["settings"])

#settings menu
@main.route('/settings')
def settingsMenu():
    return render_template('settings.html', settings=session["flight_data"]["settings"])
```

Then I changed the root to be the body instead of the Document element (which is <html> not <body>) and had the body check the already rendered settings data to check if darkmode was enabled and to add it to the classList there and then if it was the case.

settings.js

```
//-----light/dark toggle-----
const root = document.body
```

Main-menu.html, settings.html

```
<body class="{ 'dark-mode' if settings.theme == 'dark' else '' }"> <!--check and set correct theme-->
```

Test No.	Test Description	Expected Outcome	Actual Outcome
53	Light/dark sharing test	The main menu should be in dark mode when loaded up after switching to dark mode via settings. Settings should also stay in dark without resetting on reload.	PASS (See full test below)

After a successful test, I then went on to applying all these settings to all the other pages, and testing out their darkmode, and trying to see if the CSS had any problems, for example if I set a default colour instead of the variable that will actually allow a change (this happened quite a few times).

After fixing any CSS styling errors I performed a full dark mode functionality test for all screens, which is shown below.

Test No.	Test Description	Expected Outcome	Actual Outcome
54	Darkmode Functionality: Planning module and Main menu	All screens in the test description should display dark mode colors when dark mode is selected in settings.	

Iteration Review

Met requirements: (Aim complete) Perform practical testing of this system with light dark toggling being saved and maintaining the theme throughout each screen

(Aim complete) Create a method of writing and reading data from a session

Changes: N/A

Next steps: Settings units selection

Iteration 4 - Settings Units Selection

settings.html

```
id="feet" class="{{ 'active' if settings.units_altitude == 'feet' else '' }}">Feet  
id="metres" class="{{ 'active' if settings.units_altitude == 'metres' else '' }}">Metres
```

settings.css

```
#knots.active,  
#mph.active,  
#kmph.active,  
#feet.active,  
#metres.active,  
#kg.active,  
#pounds.active,  
#litres.active,  
#gallons.active,  
#nautical_miles.active,  
#kilometres.active,  
#statute_miles.active {  
  
    background-color: var(--primary);  
}
```

settings.js

```
function updateSettings (key, value) {
  var r = new XMLHttpRequest(); //allows sending HTTP requests (eg POST)
  r.open("POST", "http://127.0.0.1:5000/update-settings", true);
  r.setRequestHeader("Content-Type", "application/json");
  r.onreadystatechange = function () {
    if (r.readyState !== 4 || r.status !== 200) return;
  };
  r.send(JSON.stringify({ "key": key, "value": value }));
}

//--DARK MODE--
darkMode.addEventListener("click", () => {
  root.classList.add("dark-mode");
  updateSettings("theme", "dark");
});

//--LIGHT MODE--
lightMode.addEventListener("click", () => {
  root.classList.remove("dark-mode");
  updateSettings("theme", "light");
});
```

```
//-----UNIT SELECTION-----
function selected (button, key, value) { //to update session value and color
  updateSettings(key, value);
  button.classList.add('active'); //when it is active its color changes to primary
};

//-----AIRSPEED-----
const knots = document.getElementById('knots');
const mph = document.getElementById('mph');
const kmph = document.getElementById('kmph');

knots.addEventListener("click", () => {
  mph.classList.remove('active')
  kmph.classList.remove('active') //active removed so returns to normal color
  knots.addEventListener("click", selected(knots, "units_airspeed", "knots"))
});
```

(Continues for all the other units for mass, fuel, distance, etc)

What I did

Now, given I had the basics of reading data completed, I decided to finish off the units section of the settings menu. All this meant doing was making the respective item in the session file change value when a button was clicked, also, its color must change to show it's selected.

Going back to settings.js, I realised I will be updating a lot more data and so I needed to turn that update function that I used for the dark/light toggle into a single function that takes in the key and value, and then runs that through the settings update function. This was quick and already made the code at dark/light mode look a bit cleaner.

The way the colours had to change works the same way as how I did light and dark mode. First, on initial load of the page the HTML reads what the current unit for that category is. Then for each button, the code `class="{{ 'active' if settings.units_[CATEGORY] == '[UNIT]' else '' }}"` will check if their unit is selected. If so, 'active' is added to their classList, which turns their colour to primary, as coded in settings.css.

Of course, I needed a way to immediately show that there is a change in units without having to reload the page. This is where I simply played around with each of the buttons 'active' attributes depending on the button pressed. For example, in the airspeed example in the snippet, when 'knots' is selected, the other two units have their active values removed so any selected will immediately return to its normal colour. Then, the `selected (button, key, value)` function will update the session using the keys and values for that category and button, as well as adding 'active' to that button's classList.

Why I did it this way

The obvious advantage of using all these functions is that code is written efficiently and thus is more readable and less time consuming, it is also very useful in the long term as I can refine the update function and have it apply to all buttons rather than update it once and have issues with repeating it for all occurrences (which would be even worse when it comes to the general flight details).

The alternative to using classList was to change the colours manually. This is simply inefficient because it has to happen too many times at once. The active class list value is almost like a function, because I can apply any styling the moment 'active' is added to its classList. Without this I'd have to write the applying and removal styling for all buttons whenever I press just one which is time consuming and a waste of space.

Program output

 Units test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
55	Settings units update test	When pressing a category, the selected unit should be in the primary color. When selecting a new unit the colour should change accordingly and the backend session should also update.	PASS (See program output)

Iteration Review

Met requirements: (Aim part-complete) Perform practical testing of this system by implementing units selection in the settings menu and coding one function to test changing the units of an inputted number value.

Changes: N/A

Next steps: 'Continue' button

Iteration 5 - Main menu New Flight button

app.py

```
#-- Starting a new flight --

#resets all flight data but NOT settings
@main.route("/new-flight")
def NewFlightRun():
    #resetting settings data for test (no data in flight to reset)
    session["flight_data"]["settings"] = data_template["settings"]
    session.modified = True
```

main-menu.js

```
function NewFlight() {
  fetch('/get-flight')
  .then(response => response.json())
  .then(data => {
    if (data.saved == "False") { //checks if current flight data is saved
      alert("Your Flight Plan has changes that have not yet been saved onto your device")
    }
  })
};

function clearSession() {
  fetch('/new-flight')
  console.log('here')
}

newFlight.addEventListener("click", function() {
  displayDisclaimer();
  NewFlight();
  disclaimerContinue.onclick = function() {
    clearSession();
    window.open("dashboard", '_self')
  };
});
```

What I did

Now that I had a good grip with working with the session and functions, I decided to code in more functionality for the main menu. The continue button was the easiest as it just meant clearing the session, and also implementing the CheckSave() algorithm. To check if the session was saved to the user's device, a simple 'saved' key was made in the session dict and is set to 'False' by default, then, at one point when the user saves this data into the dashboard then saved will be switched to true.

The function `NewFlight()` in main-menu.js checks whether the 'saved' key is set to False. If it is, then an alert will popup to tell the user that continuing via the disclaimer will clear all their unsaved data. This way the user can press 'Go Back' and save their data first through the dashboard.

`clearSession()` then runs the `/new-flight` function over in flask, which grabs the empty template for the flight data and makes the entire flight section equal to the empty template, clearing the data. For this one though, since there's no data currently available in the flight section I used the settings menu to test, which should still produce the same results (all settings should reset to defaults).

Why I did it this way

I used an alert box as it was the quickest way to code it in at the time and most browsers support it. If I decided to create a whole other popup the design would take some time that

I don't really have at the moment. I am more than happy to postpone such a feature for a simpler and fully working solution that still achieves the same goal.

By testing with the settings menu I can be sure that my system works 100%, because clearing the flight data would have the exact same functionality. In fact, I would only have to change the words 'settings' to 'flight' and it would apply. If I decided to try and test the flight data directly I would have to write separate code to update the values specifically which is actually possible but just unnecessary in my view. It's something I would test in the background, however.

Program output

 Clear session test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
56	New Flight button functionality	When the New Flight button is pressed an alert shows up warning the user. Then, if the user persists then the session should clear- in this case the settings menu should be reset and all defaults restored.	PASS (See program output)

Iteration Review

Met requirements: implemented **checkSave() algorithm**,
(Aim Complete) Perform practical testing of this system by implementing the checkSave algorithm and clearing the session when the user selects 'New Flight'

Changes: N/A

Next steps: Text reading and output testing

Iteration 6 - Inputting, reading and outputting text data

route.js

```
AIVFR > static > JS route.js > ...
1  function update (key, value) {
2      var r = new XMLHttpRequest(); //allows sending HTTP requests (eg POST)
3      r.open("POST", "http://127.0.0.1:5000/update-flight", true);
4      r.setRequestHeader("Content-Type", "application/json");
5      r.onreadystatechange = function () {
6          if (r.readyState !=4 || r.status != 200) return;
7      };
8      r.send(JSON.stringify({ "key": key, "value": value }));
9      r.send(JSON.stringify({ "key" : "saved", "value" : "False"}))
10 }
```

route.html

```
<div>From:<input id="departure_code" type="text" name="departure_code" placeholder="ICAO Code"></div>
<p id="departure_name"></p>

<script>
    const departure_code = document.getElementById("departure_code");
    const departure_name = document.getElementById("departure_name");

    departure_code.addEventListener("keydown", (event) => {
        if (event.key === "Enter") {
            update("departure_code", departure_code.value);
            //at this point we search for the aerodrome
            const output = departure_code.value;
            departure_name.innerText = "airport name for code" + output;
        }
    });
</script>
```

What I did

The next aim I want to complete is inputting, storing and reading text data. The test the above code aims to perform is to simply have a text input that changes a particular item in the session to the text inputted by the user, then that exact text is read from the session and outputted onto the screen.

An input tag takes the input with the tooltip as the place holder to tell the user that it's the ICAO code that is required and **not** the departure aerodrome's name. This will then be verified via the validation algorithm and then searched by the airport search algorithms that were written in the algorithms section. This all happens when the enter key is pressed, which is listened for by an EventListener linked to the text element. At the same time, the `update()` function (same as used in the settings menu) updates the session to change the departure code to the inputted text.

The `departure_name` variable is supposed to hold the name of the departure aerodrome after being fetched by the OpenAIP API, but for now, I've used placeholder text to demonstrate an output that still reads data from the session, as ultimately the `departure_name` will be read from the 'departure_name' key in the session.

Why I did it this way

Again, I decided to perform the test with relevance to a particular function that will later be fully implemented in the webtool, to avoid making new test environments for code that may not even work once integrated into the main program. I decided to use the 'departure_code' input that would be necessary in the route tab when the user inputs the departure aerodrome's airport ICAO code, which is designated to every registered airport globally.

Without the use of placeholder text, I would need to create another temporary variable to store the output that won't be linked to the session, which doesn't meet the required test data for this iteration. Also, trying to implement the API here now is unnecessary as, whilst it could work as test data, is not a requirement and also makes the coding process longer as I try to tackle any issues the API may bring up. That's all to be handled in the route tab section itself.

Program output

 Text input test.mp4

Known Limitations / Missing features

- Session is updated before input is validated (to be done in Sec 6)
- Input data disappears from textbox when page is reloaded

Test No.	Test Description	Expected Outcome	Actual Outcome
57	Text input, read and output test	The user enters text into the text box and the 'departure_code' item in the session updates to that text when the user presses enter. The departure name placeholder text should appear somewhere nearby the text to represent read data from the session.	ERROR (No outputted data)

Error Analysis & Fixes

Test No: 57

Failure Description: The session updates but there is no output feedback

Cause: Looking back at the code, It seems like there was a lot that wasn't coded properly. The data is not read from the session at all but it's just updated directly- although I'm not sure why that doesn't update either.

Fix: I'll code to what I actually want to be outputted (i.e. data being read from the session). This would require `departure_name` to be updated in the session too.

Justification: I could just use the normal text output and make it work, but this doesn't read from the session and thus is not the correct result I'm looking for. For this test to be successful I need to make sure that the output is directly from the session.

Iteration Review

Met requirements: Session is updated with text input

Changes: N/A

Next steps: Output via session

Iteration 7 - Outputting text data from the session

route.js

```
async function update(key, value) { //async function allows code to work in quick succession
  await fetch("/update-flight", { //waits for any running fetch to complete
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ key, value, saved: "False" })
  });
}

departure_code.addEventListener("keydown", async (event) => {
  if (event.key === "Enter") {
    await update("departure_code", departure_code.value);
    //at this point we validate data and then search for the aerodrome
    await update("departure_name", `airport name for ${departure_code.value}`);
  }
});
```

What I did

This iteration had me make a major change to the update function as you can see above. After researching how to solve the previous issue, it turns out that there was no output because the second update command wasn't running at all, and just the first one was. I did confirm this by logging two strings after each update in the console and only the first logged. This meant that there was most likely a problem with the update function.

Further research exposed what was wrong. Because of the proximity of the two lines of code that runs the function, the second update command didn't get the chance to run completely. This (weirdly) happened the other way around when I made an edit to update the output on the screen, so for a brief moment only the second one was updating at one point.

The solution to this was to use asynchronous functions. This makes it so that when the function is run, a 'promise' is created, and the entire function must run in its entirety before the promise is completed. The 'await' keyword waits for this to happen before running the function it is next to.

The major change I made in this one was the actual structure of the function itself. I discovered a new cleaner way to write the update function that doesn't tamper manually with the HTTP requests as before. Instead, it just fetches the function from app.py and updates the values straight after, which is much, much more cleaner and also good for the purpose of the async function.

Why I did it this way

Without the async function, either one of the update commands could run first unintentionally and block out the other. There were other ways that also weren't as useful as they still brought up errors related to the old update function, so I figured that the better solution would involve a much more efficient way to update the session. The current method is a great improvement because JS actually does less to perform the same function.

I've also moved the script away from the HTML and all into the JS file. It initially helped as I had reference to the html input but now that the code works this reference is not required and I'd like everything to be in their respective files for better code quality.

Program output

 Text read and output test.mp4

Known Limitations / Missing features

- Session is updated before input is validated (to be done in Sec 6)
- Input data disappears from textbox when page is reloaded

Test No.	Test Description	Expected Outcome	Actual Outcome
58	Text output test	The user enters text into the text box and the 'departure_code' item in the session updates to that text when the user presses enter. The departure name placeholder text should appear somewhere nearby the text to represent read data from the session.	PASS (See session output)

Iteration Review

Met requirements: (Aim complete) Perform tests on storing and reading input data via the session and outputting the read data on the screen (text input testing)

Changes: Used async function to allow many update functions to run in sequence.

Next steps: Changing the units of an input number value

Iteration 8 - Changing the units of an input number value

app.py

```
#-----UNITS CHANGING-----
import pint I actually imported this earlier just putting it here for reference.
ureg = pint.UnitRegistry()
ureg.formatter.default_format = "~" #use the shortened version by default

UNIT_MAP = { #maps all units to a base unit and a unit and the corresponding settings
key
    #for the unit that will actually be displayed
    "distance": {"base": ureg.meter, "settings_key": "units_distance"},
    "altitude": {"base": ureg.meter, "settings_key": "units_altitude"},
    "airspeed": {"base": ureg.meter / ureg.second, "settings_key": "units_airspeed"},
    "mass": {"base": ureg.kilogram, "settings_key": "units_mass"},
    "fuel": {"base": ureg.litre, "settings_key": "units_fuel"},
}

def update_units(session):
    for key, item in session["flight_data"]["flight"].items():
        if isinstance(item, dict) and "class" in item:
            #each unit gets a category to differentiate between unit groups
            category = item["class"]

            if category in UNIT_MAP:
                base_unit = UNIT_MAP[category]["base"]
                settings_key = UNIT_MAP[category]["settings_key"]

                #get the base value, which is where the conversion starts from
                canonical = item["value"] * base_unit

                #get the unit to be converted to (and to be displayed) from settings
                display_unit = session["flight_data"]["settings"][settings_key]
                converted = canonical.to(getattr(ureg, display_unit))

                #format the output so it has the number plus its units
                item["output"] = f"{int(converted.magnitude)} {converted.units}"

@main.route('/units-update')
def update_unitsRUN():
    update_units(session)
    session.modified = True
    return 'updated', 200
```

route.js

```
//-----TESTING-----  
distance.addEventListener("keydown", async (event) => {  
  if (event.key === "Enter") {  
    await update("distance", {class: "distance", value: Number(distance.value)});  
    fetch('/units-update')  
  }  
});
```

What I did

This iteration was pretty difficult, and the above code has quite a bit of logic errors currently that you can see in the program output. But anyways, here is how this works at the moment:

Now that I can update text in the session via a text box, the next thing I decided to do was to implement full functionality of changing the units of a number value. In route.js I copied the input for the departure aerodrome test and made some adjustments to fit the needs of the unit's change test. Since the user will hardly ever enter number values, I removed the output div and just made what was typed display directly into the box after the page reloaded. Immediate updating of values and the like will come afterwards.

The program I used to do the unit conversions was the python library '**pint**'. The idea was to have each unit be part of its own unit class (i.e. the unit group like airspeed for example). Then, the numbers are stored in SI units, but later converted to the units specified in `units_[unit group]`.

To do this I used a dictionary to map every unit to its base unit and the actual unit as specified in settings. To actually change these units there needs to be some sort of trigger action that reads the units now in session and does the conversion. This is where the `update_units()` function comes into play. To keep it simple, the function assigns each of the currently stored units a class, which differentiates them from each other in terms of unit groups. Then it gets the stored base value and the value to be converted to, and uses them in the `converted` variable which converts from the current (base) value to the unit in settings. Then this is formatted and stored in the 'output' key in the respective unit group. This is what's then displayed.

Why I did it this way

Storing the units in SI first made a foundation for which I can change the units from. Although now as you will see this logic is erroneous, the idea was that if everything was in its base units it would be easier to convert from because you never have to look for what unit the value was in before. From what I've seen now however, this might just be necessary.

There was an alternative library that I could use for unit conversion, and that was **unitpy**. After some consideration I decided that pint was the better option, as it's just a bit more beginner friendly and can be used for various purposes. Unitpy on the other hand is built to work with numpy which is not a library I'm familiar with nor that I plan to utilize. Therefore, if I did use unitpy, there is a higher chance that I wouldn't understand all the logic and code involved and it would therefore take me longer to develop a viable solution.

Program output

 Units conversion test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
59	Unit conversion test	When entering an input, the correct number should appear in the input box on reload. Then, once the unit is changed, on reload the value should change to correctly show the value in the selected unit.	ERROR (See program output)

Error Analysis & Fixes

Test No: 59

Failure Description: Two main logic errors:

- Initial conversion isn't proper and doesn't use currently selected units, but the base unit instead.
- Something in the update function is causing the units not to change in session.

Cause: The first is relatively obvious: for the distance the base unit is meters so when I enter the initial value it takes it in meters. This means that it attempts a conversion from meters to (in this case) nautical miles, which is not what I wanted.

I currently haven't got a clue on why the second error is occurring, but I do know it's the update function that is causing this error, so perhaps something there doesn't have the new value saved and resets it to the old one? I may need to do some extra research to find out why.

Fix: For the first, the base value should basically be the previous value. In the sense that, when changing the units, the new units are selected but the previous value is stored in the base so that it converts from that value to the next instead of going from metres all the time. This may need a new value in the main dict or just a small change in the dict map, but I'll need to see what fits best.

For the second error I'll confirm my theory with research so I don't waste too much time. Although it's probably worth looking back at the function to see if I can spot where the proposed reset occurs and that should narrow down the search.

Justification: I know the update function is the cause for the second error because commenting it out made the session actually update successfully. It's highly unlikely that it was something else related to the function especially since the update function also interacts with the session too.

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Refining the update units algorithm

Iteration 9 - Refining the update units algorithm

app.py (Indented part is part of `update_units()` function)

```
#update the base unit to the current unit
base_unit = ureg[display_unit]

#format the output so it has the number plus its units
unit_name = str(converted.units)
unit_name = CUSTOM_UNITS.get(unit_name, unit_name)
item["output"] = f"{converted.magnitude:.1f} {unit_name}"

#using custom units cus the default ones are weird
CUSTOM_UNITS = {
    'nmi' : "NM",
    'knot' : "kts",
    'kg' : "kg",
    'lb' : "lbs",
    'litre' : "L",
    'gallon' : "gal",
    'feet' : "ft",
    'meter' : "m",
    'kilometer' : "km",
    'smi' : "miles",
}
```

setting.js (Go back button)

```
onclick="fetch('/units-update').then(() => { history.back(); });"
```

What I did

To fix the first error I got in the previous iteration, I changed the update function so that once the unit is converted, that converted unit becomes the base unit for the next conversion. So instead of this:

Input in Metres → Output in Nautical Miles

changes units to kilometres

Takes value in metres → Outputs in Kilometres

(this disregards the units it was in before, which is nautical miles)

I get this:

Input in Nautical miles → Output in Nautical miles

changes units to kilometres

Takes value in current unit (Nautical miles) → Outputs in Kilometres

Then, I also decided to include in custom units because 'nmi', 'smi' and such are just not what pilots are used to seeing. So I used a dict to store the correct units, and made sure that it uses the custom units instead of the default units when outputting the final converted value.

Finally, the not-updating-on-reload issue was fixed by simply making one thing happen before the other, (Units updating before going back) and making sure the reload doesn't happen whilst the update function was running, hence the `then` command.

Why I did it this way

This webtool is supposed to be easy to learn for new users. That's why changing the units to what they'd normally expect to see is more useful and is more natural for them rather than other unit values that they maybe would've had to spend time navigating settings to find out what the actual unit means.

Program output

 Units test final.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
60	Unit conversion test	When entering an input, the correct number should appear in the output div on reload. Then, once the unit is changed, on reload the value should change to correctly show the value in the selected unit.	ERROR - SEE BELOW

Error Analysis & Fixes

Test No: 60

Failure Description: I actually did think this worked but when I looked back at it, there was an error that kept occurring that threw the whole logic of this off. Whilst the unit conversion worked, when I changed the units and actually inputted the units back in, the units would be based off of the current unit instead of the previous, so it converts from, for example, NM to NM after I changed the units to NM, rather than doing (previous unit) to NM, after changing the units to NM. Yeah, it's complex.

Cause: My fix to make sure the conversion is based on the previous unit TO the new selected unit was faulty, and its converting from the new unit to the new unit again.

Fix: Quite a few edits may need to be made to fix this but I currently need to investigate because I'm unsure of the full proper solution.

Iteration 10 - (Actually) Finalising the Units conversion algorithm

app.py

```
def update_units():
    units_changed = session["flight_data"]["settings"].get("units_changed", "False") == "True"
    for key, item in session["flight_data"]["flight"].items():
        if isinstance(item, dict) and "class" in item:
            category = item["class"]
            if category in UNIT_MAP:
                settings_key = UNIT_MAP[category]["settings_key"]
                display_unit = session["flight_data"]["settings"][settings_key]
                current_unit = ureg[display_unit]

                #Store previous unit in session settings if not present
                prev_unit_key = f"prev_{settings_key}"
                if prev_unit_key not in session["flight_data"]["settings"]:
                    session["flight_data"]["settings"][prev_unit_key] = display_unit

                if units_changed:
                    #Convert value from previous unit to new unit
                    prev_unit = session["flight_data"]["settings"].get(prev_unit_key, display_unit)
                    try:
                        prev_quantity = item["value"] * ureg[prev_unit]
                        new_quantity = prev_quantity.to(current_unit)
                        item["value"] = float(new_quantity.magnitude)
                    except Exception:
                        pass
                    #Update previous unit to current
                    session["flight_data"]["settings"][prev_unit_key] = display_unit

                #Always update output
                canonical = item["value"] * current_unit
                converted = canonical.to(current_unit)
                unit_name = str(converted.units)
                unit_name = CUSTOM_UNITS.get(unit_name, unit_name)
                item["output"] = f"{converted.magnitude:.1f} {unit_name}"

    #Reset the flag to show units have been updated
    session["flight_data"]["settings"]["units_changed"] = "False"
    session.modified = True
```

```

@main.route('/units-update')
def update_unitsRUN():
    if session["flight_data"]["settings"]["units_changed"] == "True":
        update_units()
        session["flight_data"]["settings"]["units_changed"] = "False"
        session.modified = True
        return 'updated', 200

    else:
        update_units()
        return 'no change', 200

```

What I did

One of the major problems that caused this was the fact that the units were updating every time I pressed the back button. I only wanted the units to update when an item is selected, so I made another variable in the python dict called `"units_changed"` to indicate this, and used it in the function.

The re-written function works like this:

The `units_changed` variable checks if the user has changed any of the units yet. (by checking `units_changed` in the session). The function then only selects to edit items that have a class (i.e. unit categories, which was the same for the old update function).

It then determines which unit is currently being selected for this category and gets the corresponding pint unit. The base unit is now called the `current_unit`, because instead of acting as the previous unit it acts as the new unit to be converted to (the display unit). A new `prev_unit` acts as the previous unit.

Each unit will have its own `prev_unit_key`, (eg. `prev_distance`) which holds the previously selected unit for which the unit will convert **from**. If it is not in the session already, it is created.

At this point, the previous unit and the current unit are **both the same**. This means that running the update function will not cause any change. If, however, a unit is changed in the settings, `"units_changed"` becomes true, and now a unit conversion will follow. This works because the moment a unit is selected, the previous unit stays as it is, but the current unit updates to the selected one. Now, it's possible to convert from one unit to the other.

Afterwards, the output is updated to display the result of the conversion, and the `units_changed` flag is reset to False again. Also after the conversion, the previous and current units are set equal to each other again.

Why I did it this way

This was actually the alternative type of solution I came across when first developing the units algorithm but was very determined to avoid, because it involved making 'previous unit' keys for all the different categories and I didn't know how to even work with them. The good thing is that this way of doing it will add these values to the session if they're not there so they don't even need to be included in the data template.

Renaming the base unit to the current unit improves readability by showing the developer what the variable is used for. `Base_unit` became inappropriate because both the previous

unit and the current unit are dynamic, so they change from time to time. At the same time 'base' is a little ambiguous to match with the current unit selected, as it has none of its connotations to it.

Program output

 units test final (actually).mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
61	Unit conversion test	When entering an input, the correct number should appear in the output div on reload. Then, once the unit is changed, on reload the value should change to correctly show the value in the selected unit.	PASS (See program output)

Iteration Review

Met requirements: **Success criteria item complete:** Settings menu

(Aim complete) Perform practical testing of this system by implementing units selection in the settings menu and coding one function to test changing the units of an inputted number value.

Changes: Updated the update_units function

Next steps: Saving a flight plan to the device and loading in a saved flight plan

Iteration 11 - Saving and loading a flight plan: Save button

Dashboard.html

```
<div id="header" display="flex">
  <div id="title">DASHBOARD</div>
  <button id="save" class="{ 'savingOrSaved' if flight.saved == 'True'
else ' ' }">{{ 'SAVED' if flight.saved == 'True' else 'SAVE FLIGHT'
}}</button>
</div>
```

dashboard.css

```
#header {
  display: flex;
  justify-content: space-between;
  border: 1px none var(--text);
  height: 46px;
}

#save { /*save flight*/
  width: 165px;
  height: 45px;
  font-family: 'Manrope';
  font-size: 15px;
  font-weight: 600;
  padding: 8px 26px;
  border-radius: 5px;
  border: 1px solid var(--text);
  background-color: var(--bg);
}

#save:hover {
  cursor: pointer;
  background-color: var(--onhover);
}

#save.savingOrSaved { /*user feedback when saving/saved */
  border: 1px solid gray;
  color: gray;
  cursor: not-allowed;
  pointer-events: none;
}
```

dashboard.js

```
//-----SAVE BUTTON-----

const saveButton = document.getElementById("save");

saveButton.addEventListener("click", () => {
  saveButton.classList.add("savingOrSaved");
  saveButton.textContent = "SAVING...";

  /*save flight algorithm here*/
  //when done:
  update("saved", "True");

  setTimeout(() => {
    saveButton.textContent = "SAVED";
  }, 2000); //2 seconds
})
```

What I did

So far all aims have been achieved in this section, and the last one is the ability to save and load in a flight plan from the user's device via a saved .JSON file. Since this is a single feature, not to be present in multiple pages such as the text input testing and unit conversions, I decided to implement it in its entirety all at once. This means making the menu load button fully functional.

This first iteration focuses on the save button itself, which is in the dashboard. I've styled it to match the design, and coded it to have three states (by using classList again): SAVE FLIGHT, SAVING..., and SAVED. When the page is loaded, the class List 'savingOrSaved' changes the styling such that it goes grey and cannot be clicked. Since this styling is required when saving is occurring and when the flight is already saved, both SAVING... and SAVED states utilize this styling (hence 'Saving or saved').

If the data is not saved ("saved" = 'False' in the session) then the button will be in its default (SAVE FLIGHT) state with the savingOrSaved classList removed. Then pressing it will add the classlist and change its text to say 'saving...', and this is where the user would be saving the session to their device. Once this is done, the text changes from SAVING... to SAVED, and remains unclickable. The "saved" value in the session is then set to true. This achieves the type of feedback that I explained as part of the usability features section in the design.

Why I did it this way

I could've also implemented this feature in a test format just like I did for text input testing and unit conversions, but again, it is a single feature that requires little coding to be

completed and so not doing it in full would mean I would have to complete it in the dashboard, which, however small, is one more task to do.

Program output

 Save button test 1.mp4

Known Limitations / Missing features

- Save button not entirely functional

Test No.	Test Description	Expected Outcome	Actual Outcome
62	Save button initial styling test	Clicking the 'SAVE FLIGHT' button should change its styling and make it unclickable, and its text should say 'SAVING...'. Then the session should update and saved should be true, and then the buttons text should say 'SAVED', and should remain unclickable	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Saving a flight to the device

Iteration 12 - Saving the flight data to the user's device

dashboard.js

```
async function saveFlight() {
    //fetch session data from flask
    const response = await fetch("/save-flight");
    const data = await response.json();

    //save the flight data as a json file
    const blob = new Blob([JSON.stringify(data)], { type: "application/json" });
    const url = URL.createObjectURL(blob);
    const a = document.createElement("a");
    a.href = url;
    //set the file name to departure and destination airports for uniqueness
    a.download = `${data.flight_data.flight.departureAirport_code}-${data.flight_data.flight.destinationAirport_code}.json`;
    document.body.appendChild(a);
    a.click();
    document.body.removeChild(a);
    URL.revokeObjectURL(url);
}
```

app.py

```
#-----EXPORTING DATA-----

#saves the current flight data as a json file
@main.route("/save-flight")
def save_flight():
    return jsonify(dict(session))
```

app.py - data_template (new fields in bold)

```
"flight" : {
    "saved" : "False",
    "departureAirport_code" : "",
    "departureAirport_name" : "",
    "destinationAirport_code" : "",
    "destinationAirport_name" : "",
    "distance" : {"value" : 0, "class" : "distance"},
}
}
```

What I did

The next thing to do was to actually save the flight to the user's device when the save button was pressed. Luckily, there is a very specific way to save the flask session as a downloadable .json and so this is the method you see above. It involves using a Blob, which is essentially a file-like, immutable object that can be used in JS so that whatever is treated as a blob is treated like a file. Thus, you can use calls that are meant to process files, such as `createObjectURL()` (which is what is most used, and is used in this case), allowing me to treat this file as if it was on a web server, and provide functionalities such as downloading said files using the browser onto the user's device.

The file name is set to the departure and arrival aerodromes, so I also had to add the destination airport name and code fields to app.py so they can be read when the file can be made (yes, these would just be empty as I didn't test an input for the arrival aerodrome, just the departure one).

Finally, I made some styling changes to the SAVE FLIGHT button. The borders have been removed in the 'savingOrSaved' state and the whole button turns grey instead, making it more obvious that it can't be accessed anymore. It was a small edit so is not included in the snapshots.

Why I did it this way

I had to make sure that the name of the file was not something like 'flight_data.json', otherwise all the different saves would be the same (albeit windows would number duplicates as part of its own file management system). The solution was to make the name the departure and destination airport codes, which is actually something many flight planning software uses, and immediately provides uniqueness to each flight.

The changes to the SAVE FLIGHT button were made in an attempt to make it more visual to the user in the sense of when it can and can't be clicked. Leaving it how it was before could still work (especially since the words themselves are a very defined gray), but the new version just makes it immediately obvious and the user doesn't even need to think to know that the button is unclickable. It's this type of feedback that makes the user feel like they're in control and interacting smoothly with the webtool.

Program output

 save flight test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
63	Save flight test	Upon pressing 'save flight'. The entire session should download in the browser onto the user's device.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: SAVE FLIGHT button styling changed to look more obvious to user

Next steps: Load flight popup

Iteration 13 - Load Flight Popup

main-menu.html

```
<!--Upload Flight Modal-->
<div id="uploadFlight" class="modalUpload">
  <div class="disclaimerContent">
    <h1 id="disclaimerHeader">LOAD FLIGHT</h1>
    <p>Select a valid flight file (.json) to load your previously saved flight.</p>
    <form id="uploadFlightForm" enctype="multipart/form-data">
      <!--form to allow the user to pick a JSON file from their device-->
      <input type="file" id="flightFile" name="flightFile" accept=".json" style="display:none;" />
      <label for="flightFile" class="browse">BROWSE</label>
      <span id="fileName">No file chosen</span> <!--separate file name element to allow custom button styling-->
      <button type="submit" class="uploadFlightButton TARGET">LOAD FLIGHT</button>
    </form>
    <hr width="80%">
    <button class="uploadFlightButton" id="uploadFlightEXIT">CANCEL</button>
  </div>
</div>
```

main-menu.js

```
loadFlight.addEventListener("click", function() {
  displayDisclaimer();
  disclaimerContinue.onclick = function() {
    disclaimer.style.display = "none";
    displayUploadFlight();
  }
});
```

```
//-----LOAD FLIGHT

//saves the name of the uploaded file to display in the popup
document.getElementById('flightFile').addEventListener('change', function() {
  var fileName = this.files[0] ? this.files[0].name : '';
  document.getElementById('fileName').textContent = fileName;
});
```

main-menu.css

```
/* Upload Flight Modal Content/Box */
.uploadFlightContent {
  position: relative;
  background-color: var(--highlight);
  margin: 10% auto;
  padding: 35px;
  width: 720px;
  border-radius: 15px;
  text-align: center;
  bottom: 50px;
}

.uploadFlightButton {
```

```

    position: relative;
    font-family: 'Manrope';
    font-size: 20px;
    background-color: var(--bg);
    border: none;
    padding: 6px 20px;
    border-radius: 5px;
    margin-left: 100px;
    margin-right: 5px;
    margin-bottom: 10px;
}

hr {
    border: 1px solid;
    width: 90%;
    margin-top: 1px;
    margin-bottom: 20px;
    color: hsla(0, 0%, 0%, 0.1);
}

.browse {
    position: relative;
    font-family: 'Manrope';
    font-size: 20px;
    background-color: var(--bg);
    border: none;
    padding: 6px 20px;
    border-radius: 5px;
    margin-left: 5px;
    margin-right: 5px;
    margin-bottom: 10px;
}

.browse:hover {
    cursor: pointer;
    background-color: var(--primary);
}

#fileName {
    font-family: 'Manrope';
    font-size: 18px;
    color: var(--text);
    margin-left: 15px;
    font-weight: 600;
}

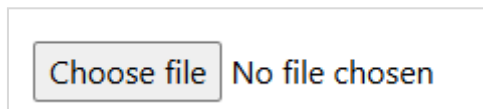
```

```
#uploadFlightEXIT {  
  position: relative;  
  right: 50px;  
}
```

What I did

Next was the actual popup where the user can upload their file. This part can be split into two: the popup itself and the actual file selection. For the popup, Most of the code was just directly copied over from the disclaimer popup, including the buttons, which uses the same styling as those used in the disclaimer. I coded the popup to appear immediately when the disclaimer was accepted, which holds consistently with the disclaimer being the first thing that shows up no matter what menu button you press.

Then it came over to the upload part. This was much easier than I anticipated all thanks to the ability to simply change the 'type' of input to 'file'. Then the rest was just styling. I got stuck at one point because no matter what styling I applied to the <input> element, the upload file format looked like this:



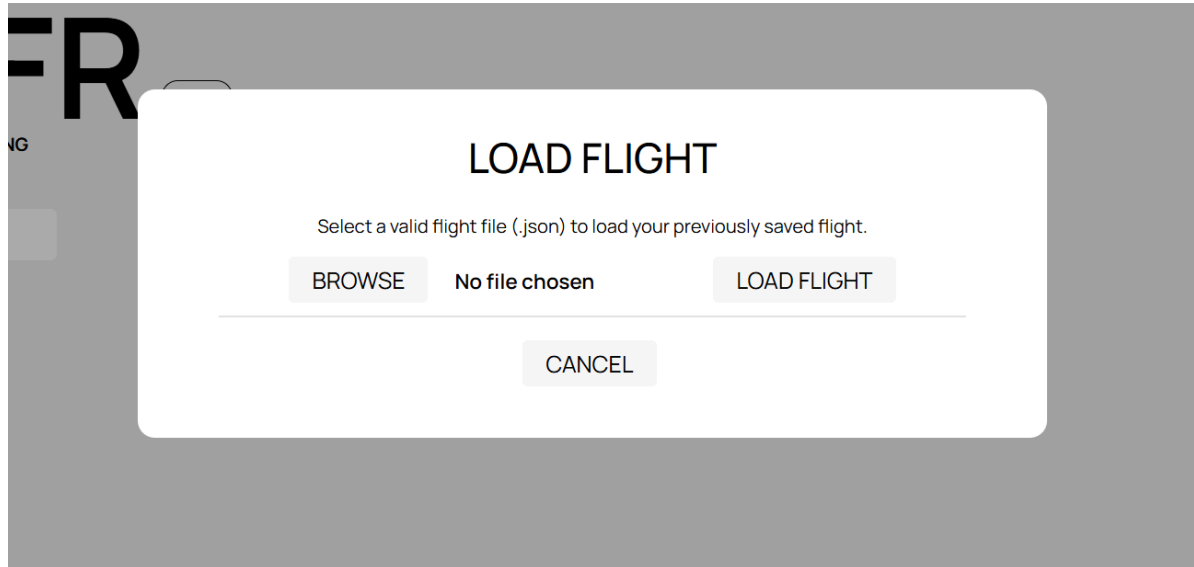
Which didn't match the design. After some research I found that you can use labels and attach them to other elements, so that accessing the label also accessed the linked element. In this case, I created a separate label, with words 'BROWSE' in it (I just preferred this over 'choose file'), styled it so it looks like the button, and added effects when it's hovered over. Then, when it's clicked, it activates the input element which opens the user's file system to upload the file.

The final issue was the fact that applying my own styling meant that I had to make the input element invisible, which took away the text that shows what file has been selected, to fix this, a separate inline element was made with id 'fileName', and over in **main-menu.js**, an eventListener listens for a change in the variable 'flightFile' which is attached to the button, and reads the name of any files uploaded, stores that in a variable fileName, and changes the textContent of the fileName inline element to say the name of the uploaded file.

Why I did it this way

I could've coded a different popup separately for this, and this would mean I could add extra functionalities if I wanted to since the styling used for each is different. But I decided not to do this simply because it was unnecessary at the time (and still is, really) and I didn't want to waste any time developing styles that already exist.

Program output



Load flight popup test.mp4

Note: I do actually go into my files to get the flight file it just didn't show as I screen recorded just this window not the whole screen

Known Limitations / Missing features

- 'Load flight' button has no functionality

Test No.	Test Description	Expected Outcome	Actual Outcome
64	Load flight popup test	Load flight popup should appear immediately when the disclaimer is accepted.	PASS (See program output)
65	File upload test	Users can upload a file by pressing the 'BROWSE' button. Upon uploading a file, the file name is replaced and can be seen by the user.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Replacing the session with the uploaded session

Iteration 14 - Replacing the session with the uploaded session

main-menu.html

```
<button type="submit" class="uploadFlightButton TARGET"
onclick="loadSession()">LOAD FLIGHT</button>
```

app.py

```
@main.route("/load-flight", methods=["POST"])
def load_flight():
    try: #try block allows me to catch any errors and display them on the page
        New_data = request.get_json()
        if not isinstance(New_data, dict):
            #raise an error if the data is not a dictionary
            raise ValueError("Invalid data format")

        #replace the current session data with the new data
        session.clear()
        session.update(New_data)
        update_units() #update units in case the loaded data has different units
        session.modified = True

    return jsonify({"status": "success", "message": "Flight data loaded successfully."}), 200
except Exception as error:
    return jsonify({"status": "error", "message": str(error)}), 400
```

main-menu.js

```
async function loadSession() {
    const input = document.getElementById('flightFile');

    //check if a file has been selected
    if (!input.files.length) return alert("Please select a file to upload.");

    //read the file as text
    const file = input.files[0];
    const text = await file.text();
    let flightData;
    try {
        flightData = JSON.parse(text); //parse the text as JSON
    } catch (error) { //return an error if the file is not valid JSON
        return alert("Invalid file format. Please upload a valid JSON file.");
    }

    //send the flight data to the server
    const response = await fetch('/load-flight', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json'
        },
        body: JSON.stringify(flightData)
    });
}
```

What I did

Finally, the last thing to do is to replace the session with the uploaded one. When you upload a file, they are appended into a **FileList**, which is essentially an array containing all files uploaded by an `<input>` Element. This is how I can access the uploaded file. The `loadSession` function first checks if there is anything even in the array, and if not an alert will be sent to let the user know that they must upload a file first.

Then, the file is turned into text before being parsed into JSON. This is in a try block because a parsing error can be fired if the text isn't valid, which the try block's exception will deal with (i.e the `catch(error)` part, which checks if any error, including a parsing error is fired). The uploaded and now validated session is then sent over to flask in **app.py**, where it can be replaced with the current session.

Over in `app.py`, the `/load-flight` function uses another try block to further check if the session is in fact a dictionary. This is important as the session still needs to be perfectly readable by the program. If it is a dictionary, then all criteria are met for this file to be valid, and so the current session is simply cleared, and replaced with the new session. The units are also updated in case of a change, for example if the user currently had different units selected.

Why I did it this way

Without converting the file into text first then parsing, there would be no way of confirming that it is truly a .json file that has been uploaded. This would give the user the ability to post files that could then potentially break the program, or something similar.

Without the try blocks, in both cases, there would be no way of detecting an error that happens in this entire process, and the user will just have a non-working webpage whilst the backend brings up the error. This is not useful and so these errors should be detected so alternate outputs that don't break the program can be made.

Program output

 upload flight final.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
66	Load flight test	Uploading a flight will successfully clear the session and replace it with the new uploaded session, and values from it should also be outputted.	PASS (See program output)

Some small additions made after this test:

main-menu.js

```
if (response.ok) { //check if the response is okay (it passed the server-side validation)
  uploadStatus.style.color = "green";
  uploadStatus.textContent = "Flight loaded successfully, redirecting to dashboard...";
  setTimeout(() => {
    window.open("dashboard", '_self'); //redirect to dashboard after a short delay
  }, 1000);
} else { //if the response is not okay, display the error message from the server
  const errorData = await response.json();
  uploadStatus.style.color = "red";
  uploadStatus.textContent = errorData.error || "An error occurred while loading the flight.";
}
```

menu.html

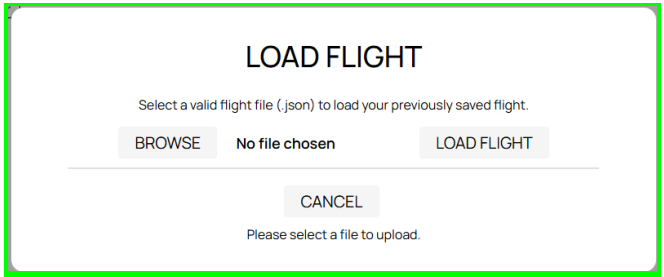
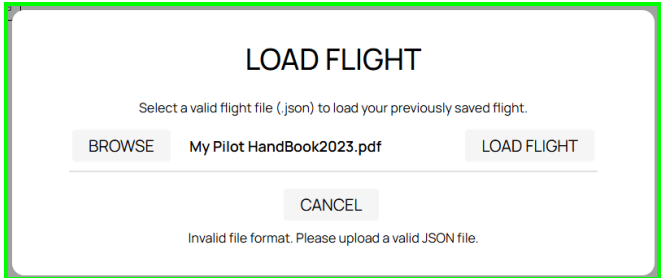
```
<div id="uploadStatus"></div>
```

New features:

- A new `uploadStatus` div displays the status of the upload instead of an alert. This is more visually appealing as it is embedded in the webtool instead of a browser alert.
- The user is redirected to the dashboard when the upload is complete. This came after fixing the URL changing issue I addressed in the first video. It meant the page wouldn't reload on the spot, but I found it would make sense if the user goes straight to the dashboard and so that's what I did.

The results of these tests were all functional and can be seen in the input validation tests below.

Input Validation Tests

Test Data	Expected Outcome	Actual Outcome
Nothing uploaded but load button pressed	Upload status text should appear reading: 'Please Select a file to upload'	
A pdf file is uploaded	Upload status text should appear reading: 'Invalid File format. Please upload a JSON file'	
A json file that is not a dictionary	Backend Value error should appear in the terminal/console,	Test.json looks like this (An AI generated Array of mixed data types):

	Upload status text should read 'an error occurred whilst loading the flight'	<pre>["Flight 123", 42, true, null, [1, 2, 3, ["nested", false]], { "aircraft": "C152", "hours": 12.5, "completed": false, "tags": ["training", "solo", "cross-country"] }, 3.14159, "End of data"]</pre> LOAD FLIGHT Select a valid flight file (.json) to load your previously saved flight. BROWSE test.json LOAD FLIGHT CANCEL An error occurred while loading the flight.
--	--	---

Full test video:

 Upload flight validation.mp4

Iteration Review

Met requirements: (Aim complete) Testing making the session file into a .json file and saving it to the user's device, then using a saved file to load in a flight from the device (load flight button functionality)

Success criteria item complete: "Ability to save, edit and start new flight plans"

Changes: N/A

Next steps: Section 6 - Route Tab

Section Review

Aims completed in this section:

- **Complete a success criteria item: “Ability to save, edit and start new flight plans”**
- Find out how to store data in local storage/Cache
- Create a method of writing and reading data from a session
- Perform tests on storing and reading input data via the session and outputting the read data on the screen (text input testing)
- Perform practical testing of this system with light dark toggling being saved and maintaining the theme throughout each screen
- Perform practical testing of this system by implementing units selection in the settings menu and coding one function to test changing the units of an inputted number value.
- Perform practical testing of this system by implementing the checkSave algorithm and clearing the session when the user selects ‘New Flight’
- Testing making the session file into a .json file and saving it to the user’s device, then using a saved file to load in a flight from the device (load flight button functionality)

Aims not completed in this section:

N/A

Iterative tests complete:

- Data stored in the .json flight data file should be able to be saved to the cache for future use.

Remarks: Really, really successful section! The planning module should become a much easier task now that all the complicated flask stuff has been sorted out.

THE PLANNING MODULE

Working Backwards

I've decided to work backwards when making the planning module, which means, instead of covering all the tabs in order, starting with the dashboard, I will end with the dashboard instead. This has very simple reasoning behind it, actually.

In the previous section, we managed to read data from the session and output it on the screen. The dashboard is a summary of the entire flight, so it wouldn't make sense to code it first- what data would there be to use? By the time I've completed the other tabs, the session should be complete with all the values, so I can put in a flight plan and code the dashboard to show these values as per design. *This is the more efficient way of doing it, rather than placing placeholder values instead to mimic the full design, because the use of real data allows me to detect any issues with my own data or even the session, unlike the hardcoded placeholder text that will only look like how I code it.*

It also means I can test how different data looks like in the dashboard by changing or completely filing a new flight plan.

Section 6: Route Tab

Aims of this section

- **Complete 2 success criteria items: Route generation, Map with route Visualisation**
- **Complete 3 Essential features:** Route generation, Ability to enter departure/Arrival aerodromes, Route map
- Give the user the ability to add departure and arrival aerodromes
- Validate this data by implementing 2 planned algorithms - `verifyICAO()` and `fetchAirportName()`
- Code the ability to add 'via' aerodromes and alternate aerodromes
- Code the route generator through implementation of `fetchRole()` and `prompt()` Algorithms
- Add tooltips for route setup
- Link the route to the session and allow it to be edited at any time by the user

Iteration 1 - HTML initial layout and styling

route.html

```
<main>
  <div id="title">ROUTE</div>
  <div id="content" display="grid">

    <!--ROUTE SETUP-->
    <div id="route_setup" class="routeItem">
      | ROUTE SETUP
    </div>

    <!--ROUTE IN ICAO FORMAT-->
    <div id="ICAO_route" class="routeItem">
      | ICAO ROUTE
    </div>

    <!--AI ROUTE GENERATOR-->
    <div id="route_generator" class="routeItem">
      | ROUTE GENERATOR
    </div>

    <!--MAP-->
    <div id="map" class="routeItem">
      | MAP
    </div>
  </div>
</main>
```

Route.css

```
/*Main content*/
#content {
  grid-template-columns: 5fr 5fr;
  display: grid; /* grid allows flexible layout */
  gap: 10px;
  border-radius: 10px;
}

.routeItem { /* all items in the route page */
  margin: 2px;
  background-color: var(--highlight);
  padding: 10px;
  border-radius: 10px;
}

/* all these three items are in the first column */
#route_setup, #ICAO_route, #route_generator {
  grid-column: 1 / span 1;
}

/*map remains in the second column but spans three rows*/
#map {
  grid-column: 2 / span 2;
  grid-row: 1 / span 3;
}

/*splitting the route items to take fractions of the viewport height*/
#route_setup {
  height: 30vh;
}

#ICAO_route {
  height: 20vh;
}

#route_generator {
  height: 50vh;
}
```

What I did

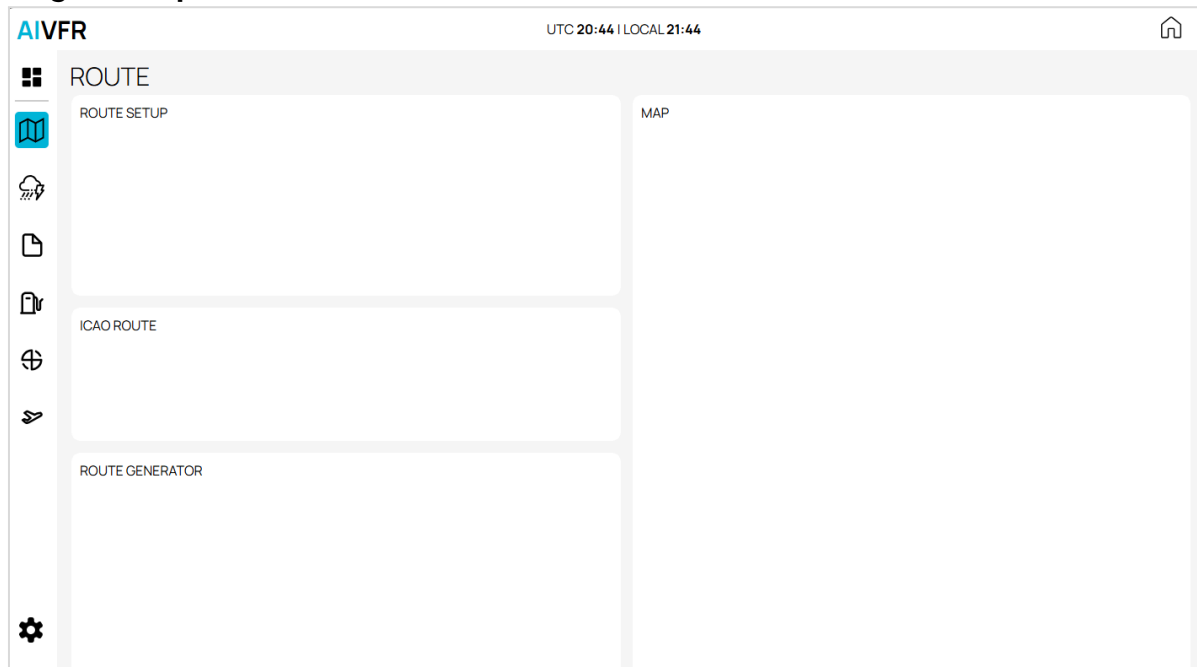
Most of the tabs here have quite a complex layout. So this layout is what I will be working on first when I start coding each tab. I've decided to use CSS grid, as this provides the best system that allows me to position elements in the necessary order and layout as per-design.

So the entire layout is placed in a div element 'content', whose display value is set to grid. Then, four more divs are encased within, for each of the four sections: route setup, ICAO route, Route generator and the route map. To lay it out as per design, I could simply adjust the grid positioning for certain elements so that they appear in certain rows and certain columns. I won't explain that here as the code already does that.

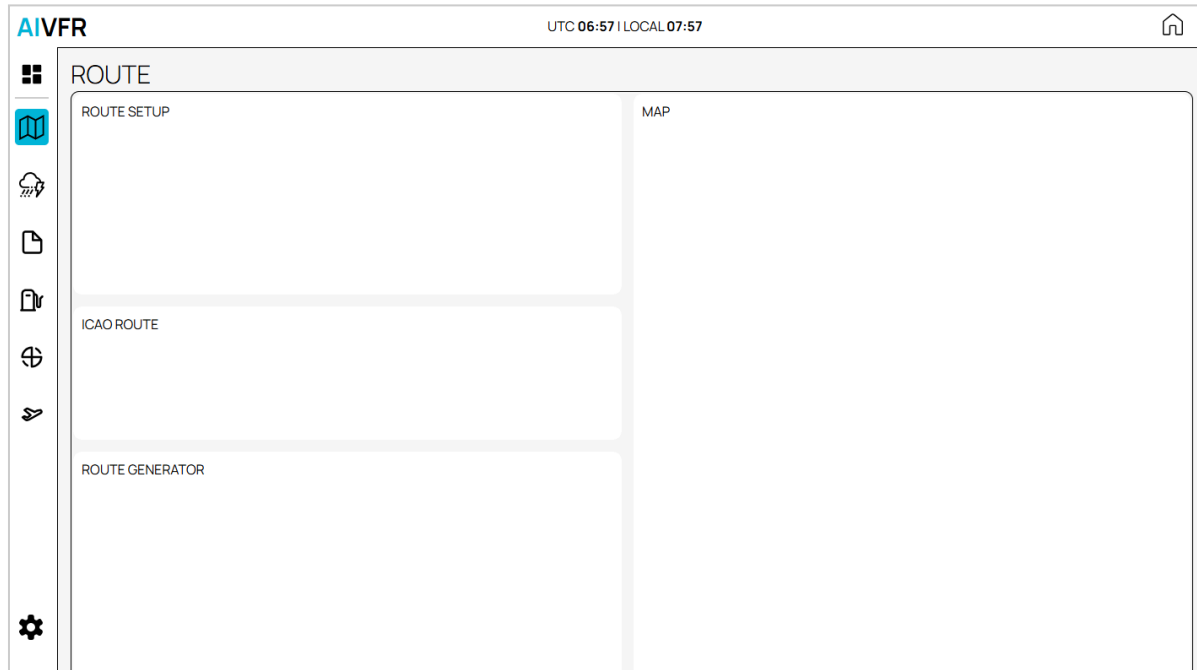
Why I did it this way

The clear alternative to using CSS grid was to use CSS Flexbox instead. The problem with this is that flexbox is used for one-dimensional layouts, with rows **or** columns, nothing in between. This is an issue because I need both. For this, and only this reason, CSS flexbox would not make a suitable option to achieve the type of layout you see below.

Program output

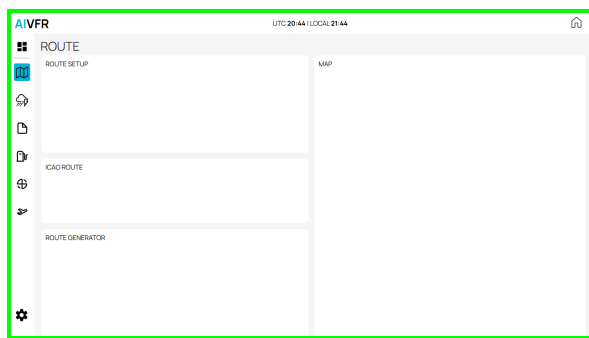


With borders to visualise edges



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
66	Page layout test	The page sections are laid out properly as per design.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Route setup

Iteration 2 - Route setup: Adding and positioning elements

Intended Design for reference

ROUTE

Route Setup ⓘ

50 minutes | 74 NM

From:

EGSX

North Weald

→

To:

EGKA

Shoreham

Via:

Alternate:

EGKB Biggin Hill

route.html

```
<!-------ROUTE SETUP-->
<div id="route_setup" class="routeItem">

  <!-------HEADER----->
  <div id="route_setup_header">
    <div> <!--Title with tooltip-->
    | <h3>Route Setup<svg xmlns="http://www.w3.org/2000/svg" id="route_setup_help" />
    </div>

    <div> <!--Route time & distance-->
    |   TIME | DISTANCE
    </div>
  </div>

  <!-------BODY----->
  <div id="route_setup_body">
    <div>
      <!--DEPARTURE--> From: <input id="departure_code" type="text" name="departure_code" />
      <!--DEPARTURE_NAME--> <span id="departure_name">North Weald</span>

      <!--ARROW--> <span><svg xmlns="http://www.w3.org/2000/svg" id="arrow" /></span>

      <!--ARRIVAL--> <span>To: <input id="arrival_code" type="text" name="arrival_code" />
      <!--ARRIVAL_NAME--> <span id="arrival_name">Shoreham</span>
    </div>

    <!--VIA--> Via: <input id="via_code" type="text" name="via_code" />
    <!--VIA_LIST--> <span id="via_list">NAME</span>
    <br>
    <!--ALTERNATE--> Alternate: <input id="alternate_code" type="text" name="alternate_code" />
    <!--ALTERNATE_NAME--> <span id="alternate_name">NAME</span>
  </div>
</div>
```

route.css

```
/*-----Route setup-----*/

#route_setup_header {
  display: flex;
  justify-content: space-between;
}

#routeSetupTooltip {
  margin-left: 5px;
  color: var(--text);
  cursor: pointer;
  vertical-align: middle;
}

#toArrow {
  vertical-align: middle;
  color: var(--text);
  transform: scaleX(-1); /* flip the arrow horizontally */
}
```

What I did

The aim of this iteration was to get all the elements for this subsection inside, in the positions they are supposed to be. This is why all items are just placeholders. It's a lot of elements so I've tried to make it as clear as possible in the code, and for that reason there is not much to explain here, It's just a step in the process.

Why I did it this way

The reason why there are only placeholder texts (and not the coded inputs as used in the last section) is because I'm coding this in a separate file that doesn't use any Jinja2 syntax. This way, I don't have to constantly go to the terminal and run flask every time I want to see the changes I made in the HTML, I can just use the live view extension I have on VS so it can update live, which saves a lot more time. Of course, when it comes to interacting with the session, I'll go back to the traditional 'flask run' commands in the terminal, which will be necessary anyway to navigate the webpage properly, but It's not necessary for styling purposes.

Program output

ROUTE

Route Setup ⓘ

From: North Weald → To:

Shoreham

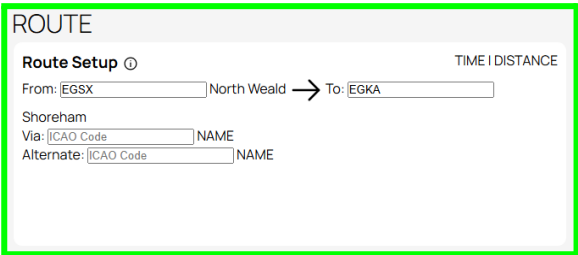
Via: NAME

Alternate: NAME

TIME | DISTANCE

Known Limitations / Missing features

- Inputs not functional

Test No.	Test Description	Expected Outcome	Actual Outcome
67	Initial positioning test	All elements should be added and in their correct positions	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Styling the elements

Iteration 3 - Route setup: Styling the elements

route.html

```

<!--BODY-->
<div id="route_setup_body">
  <div id="departureArrival">
    <!--DEPARTURE--> From: <input id="departure_c
    <!--DEPARTURE_NAME--> <span id="departure_nam

    <!--ARROW--> <span><svg xmlns="http://www.w3.

    <!--ARRIVAL--> <span>To: <input id="arrival_c
    <!--ARRIVAL_NAME--> <span id="arrival_name">S
  </div>

  <br>
  <!--VIA--> <span id="via">Via:</span> <input
  <!--VIA_LIST--> <span id="via_list">NAME</spa

  <br>
  <br>
  <br>

  <!--ALTERNATE--> <span id="alternate">Alternat
  <!--ALTERNATE_NAME--> <span id="alternate_nam

```

route.css

```
66  #toArrow {
67      vertical-align: middle;
68      color: var(--text);
69      transform: scaleX(-1); /* flip the arrow horizontally */
70      margin: 0 15px;
71  }
72  }
73
74  #departureArrival, #via, #alternate {
75      color: var(--text);
76      font-weight: 600;
77  }
78
79  #departure_name, #arrival_name, #via_list, #alternate_name {
80      font-weight: 400;
81      margin-left: 10px;
82      color: var(--text);
83      background-color: var(--bg);
84      padding: 5px 10px;
85  }
86
87  input {
88      border: 1px solid var(--text);
89      border-radius: 5px;
90      padding: 3px;
91      margin-left: 10px;
92      width: 100px;
93      text-align: center;
94      font-weight: 600;
95      font-family: 'Manrope';
96      font-size: 16px;
97  }
98
```

What I did

The styling was very simple as I knew exactly what needed to be done in order for the intended design to be met. A few adjustments were made in terms of spacing, for example the addition of
 elements, which helped space the 'via' and 'alternate' elements of this subsection.

Program output

ROUTE

Route Setup ⓘ

From:

→

To:

Via:

Alternate:

TIME | DISTANCE

Known Limitations / Missing features

- Inputs not functional

Test No.	Test Description	Expected Outcome	Actual Outcome
68	Styling comparison	The route set up section should be fully styled and should match the design	ROUTE Route Setup ⓘ From: EGSX North Weald → To: EGKA Shoreham Via: ICAO Code NAME Alternate: ICAO Code NAME TIME DISTANCE

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Validating input data

Iteration 4 - Route setup: Validating input data

basePage.js

```
//-----Alert Box-----  
const alertBox = document.getElementById("alertBox");  
  
export function showAlert(message) {  
  alertBox.textContent = message;  
  alertBox.style.display = "inline";  
  setTimeout(() => {  
    alertBox.style.display = "none";  
    alertBox.textContent = "";  
  }, 3000);  
}
```

route.js

```
//-----Verifying ICAO code-----  
  
function verifyICAO(code) {  
  if (code.length == 4) {  
    for (let letter of code) {  
      if (!isNaN(letter)) {  
        return false;  
      }  
    }  
  } else {  
    return false;  
  }  
  return true;  
}
```

basePage.css

```
/*alert box*/  
#alertBox {  
  display: none;  
  background-color: hsla(0 100% 50% / 0.6);  
  color: var(--highlight);  
  padding: 5px 10px;  
  border-radius: 5px;  
  font-weight: 600;  
  cursor: pointer;  
  position: absolute;  
  top: 30px;  
  left: 50%;  
  transform: translate(-50%, -50%); /*center the box*/  
}
```

route.js (same code for arrival and alternate aerodromes)

```
import { update, showAlert } from "../basePage.js";

//-----Departure Airport (ICAO)-----
const departureAirport_code = document.getElementById("departure_code");
const departureAirport_name = document.getElementById("departure_name");

departureAirport_code.addEventListener("keydown", async (event) => {
  if (event.key === "Enter") {
    //validating data
    if (verifyICAO(departureAirport_code.value)) {
      await update("departureAirport_code", departureAirport_code.value);
      await update("departureAirport_name", `airport name for ${departureAirport_code.value}`);
    } else {
      showAlert("Invalid ICAO code");
    }
  }
});
```

route.html (and the other html pages)

```
<div class="pageTitle" style="display: flex; align-items: center; gap: 10px;">
  <div id="title">ROUTE</div>
  <span id="alertBox"></span>
</div>
```

What I did

In order for this part to work effectively, I had to make quite a few changes and one small addition.

To start off, I took the code and put it into route.js (the one which utilises flask) in and replaced all the place holder code with Jinja2 syntax. Then in route.js, I added back the code I used for the test, to gain back that basic functionality I got in the last section. At this point there was nothing new.

Then it came to the first step of validating data, which started with verifying that a code with the correct syntax (no numbers or more/less than 4 letters) was put in, meaning I needed to implement the verifyICAO() algorithm. This was relatively simple, and was almost identical to the source pseudocode, unlike the checkSave() algorithm that changed quite a lot. When the algorithm was complete, I immediately got stuck at what to do if the algorithm detects an error, so I came up with making a little popup that will tell you that your input was invalid, which was again not a complex task.

At this point, I had this validation set for all the 3 current inputs I'm working on: departure, arrival and alternate airports ('via' can be more than once so it's a bit more complicated). I also had the algorithm that showed the alert for each input. To avoid this repetition, I turned it into a function `showAlert()` and replaced the code for each input as necessary. After this, I realized that I'd maybe want this box to be available in all of the pages, not just the route page, because errors always occur that need to be shown at one point. So, I added the necessary alert box code to the other html pages, and moved the css from **route.css** to **basePage.css**, which all the pages have access to.

Likewise, I moved the showAlert function to **basePage.js**, which again all the pages have access to- but I had to turn it into an export function so that I can import it into the respective js files in the different pages. It was after doing this that I realised I could have done the exact same thing with the **update()** function (as each page also utilises this

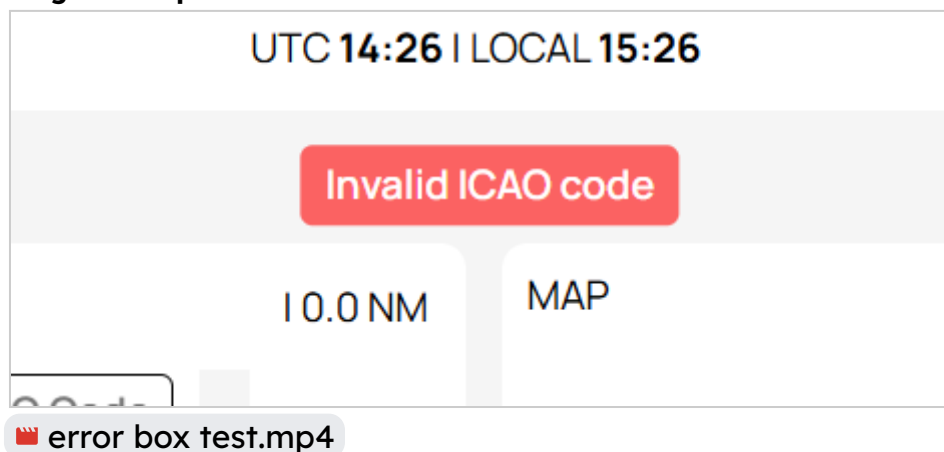
function), and so I did, which meant it didn't have to be this thing at the beginning of each file, and just the imports were necessary.

Why I did it this way

A less-time consuming alternative to the popup would have been to simply use an alert to show an invalid input. Whilst this is completely doable, It lacks authenticity because it's integrated in the browser and not the webtool itself. The more I utilise browser outputs like these, the more the user disconnects from the webtool as a fully self managing system and it, in my opinion, could make it seem like it's a flimsy or sloppy/low quality webtool altogether.

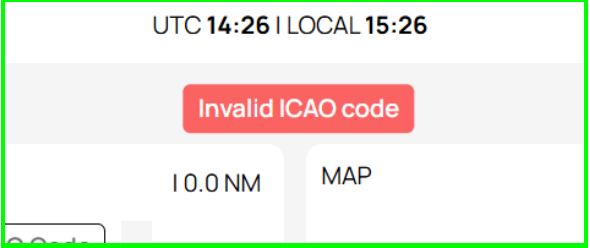
The decision behind the alert box being in all the pages was a result of me trying to find a way to show alerts to the user at any point in the planning process. If I didn't use a system like this I'd have to integrate alert systems in perhaps every subsection, which is extremely repetitive and not at all time efficient.

Program output



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
68	Input Validation test - syntax	If numbers or less/more than 4 letters are input into the box, an alert box pops up showing 'Incorrect ICAO code'	

Iteration Review

Met requirements: N/A

Changes: update() function is now in basePage.js, with imports being used instead, as well

as the new showAlert(function).

Next steps: Searching for the airport details and verifying the inputted codes

Iteration 5 - Route setup: Searching for the airport inputted

route.js

```
//-----Departure Airport (ICAO)-----
const departureAirport_code = document.getElementById("departure_code");
const departureAirport_name = document.getElementById("departure_name");

departureAirport_code.addEventListener("keydown", async (event) => {
  if (event.key === "Enter") {
    departureAirport_name.style.display = "inline"; //make the name div visible
    departureAirport_name.textContent = "..."; //simulate loading

    //validating data
    if (verifyICAO(departureAirport_code.value)) {
      const airportDetails = await fetchAirportDetails(departureAirport_code.value);
      if (airportDetails.country == "GB") { //restrict to UK airports only
        await update("departureAirport_code", departureAirport_code.value);
        await update("departureAirport_name", airportDetails.name);
        departureAirport_name.textContent = airportDetails.name;
      } else if (airportDetails.country != "GB") {
        showAlert("Only UK aerodromes are supported");
      }
    } else {
      showAlert("Invalid ICAO code");
    }
  }
});
```

```
//-----Fetching Airport Details from API-----

async function fetchAirportDetails(code) {
  try {
    const response = await fetch("/fetch-airport-details", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ code: code }), //sends off the code to the backend
    });

    if (!response.ok) {
      throw new Error("Network response was not ok"); //for non-200 errors
    }

    const data = await response.json();
    return data; //returns the airport details
  } catch (error) {
    showAlert("Error fetching airport details. Please make sure the ICAO code is correct.");
    return null;
  }
}
```

Route.html (same for arrival and alternate)

```
<!--DEPARTURE_NAME--> <span id="departure_name" class="{ 'active' if
flight.departureAirport_name != ' ' else ' }'" #this displays the airport
name div if it exists
```

app.py

```
#-----FETCHING AIRPORT DETAILS FROM API-----

load_dotenv()
api_key_openaip = os.getenv("OPENAIP_API_KEY")

@app.route("/fetch-airport-details", methods=["POST"])
def fetch_airport_details():
    code = request.json.get("code") #get the ICAO code from the request

    headers = {
        "x-openaip-api-key": api_key_openaip #correct header for OpenAIP API
    }
    params = {
        "search": str(code) #using the code from the request
    }

    url = "https://api.core.openaip.net/api/airports"

    response = requests.get(url, headers=headers, params=params) #make the request to the API
    data = response.json() #parse the response as JSON
    airport = data["items"][0]

    #taking just the relevant information
    icao_code = airport["icaoCode"]
    name = airport["name"]

    if icao_code != code:
        return jsonify({"error": "ICAO code does not match"}), 400

    return jsonify({"name": name}), 200
```

What I did

To quickly recap what this system is supposed to do:

When the user searches for an aerodrome, the code they input should get sent to the API and return the airport name for that code. Then this name can be sent back to the js to be outputted. It's through this that further validation can take place, namely if the airport doesn't exist, or the airport is outside of the UK.

Two functions work together to do this, and they are `fetch_airport_details()` in app.py which is in the **backend**, and `fetchAirportDetails()` in route.js which is in the **frontend**. The frontend function takes the airport code as a parameter, and uses method POST to send the code off to the backend function. The backend receives the code and stores it in variable `code`, which is used for in the API.

The [API](#) has very specific parameters and methods of collecting data depending on what requirements you need. All data requests requires you to have the HTTP header and query parameter that you see below (and thus in the code):

OpenAIP - Core API 1.1 OAS 3.0

<https://api.core.openaip.net/api/system/specs/v1/schema.json>

The Core API provides endpoints to handle and consume a variety of aeronautical content.

Authentication

To utilize the OpenAIP public API, you'll need a unique API key. Here's how to obtain and use your client API key:

1. Ensure you are logged into your OpenAIP account
2. Navigate to the user profile's **API Clients** page
3. On the API Clients page, request a new API client. Each client comes with its own unique API key
4. When making requests to the OpenAIP public API, include your API key using one of the following methods:
 - **HTTP Header:** Use the HTTP header **x-openaip-api-key** to include the API key in each request to the API.
 - **Query Parameter:** Use the query parameter **apiKey** to include the API key in each request to the API.

Keep your API keys confidential at all times! Treat them like passwords as they grant access to OpenAIP resources on your behalf.

But to get airport data specifically, I'd need to use the `/airports` endpoint, as also shown below on the same website.

Airports

CRUD endpoints for airport.

GET	<code>/airports</code>	Returns a list of airports.
POST	<code>/airports/rfc</code>	Endpoint to create an RFC for a new airport

The data is, as with most APIs, outputted in a way that allows it to be quickly parsed into a .json format. Then I can filter out only the airports primary details (name, location, elevation, etc) which are in the first index of the dictionary containing all the details. Others include frequencies and runways. We can then first check if the code found matches the code given, if so, it collects the name and country of the airport and sends them back to the frontend.

The frontend receives this data and begins further validation. First we need to perform one final check to ensure that a UK airport has been selected, and show an alert otherwise. If it passes this validation then it is fully validated as an airport and the session can then be updated, and the airport name can also be outputted as feedback to the user that it has been found with no errors.

Why I did it this way

N/A - No alternatives to accessing API.

Program output

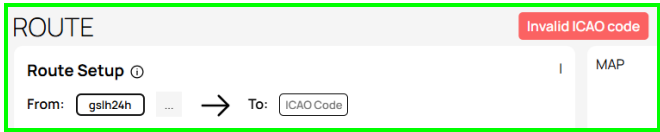
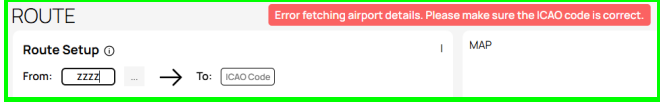
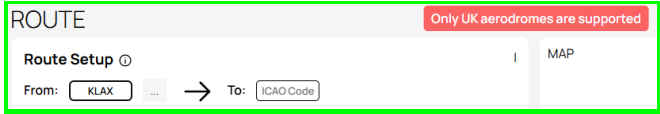
 OpenAIP API test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
69	Fetching airport name test	When the user presses enter with a valid code, the airport is found and outputted next to the input box.	PASS (See video output)

Input Validation Tests

Test Data	Expected Outcome	Actual Outcome
A code with invalid syntax (numbers and/or not 4 letters) is entered	An alert should pop up reading: "Invalid ICAO code"	
A code with correct syntax but isn't an actual airport code is entered (e.g 'ZZZZ')	An alert should popup reading: "Error fetching airport details. Please make sure the ICAO code is correct"	
A code for an airport outside of the UK is entered (e.g KLAX for Los Angeles Int.)	An alert should pop up reading: "Only UK aerodromes are supported"	

Full test video:

 Dep_arr input validation.mp4

Iteration Review

Met requirements: (Aim complete) Give the user the ability to add departure and arrival aerodromes

Validate this data by implementing 2 planned algorithms - verifyICAO() and fetchAirportName()

Essential feature complete: Ability to enter departure/Arrival aerodromes

Changes: N/A

Next steps: Route logistics

Removal of 'Via' route data

As of now, I've made the decision to scrap the idea of allowing the user to enter aerodromes to act as 'via' airports, meaning they plan to land there but it's not their final destination. The primary reason I decided to remove this is that if one route encompasses a journey from A to D, via B and C, I'd need to have separate routes for A-B, B-C and C-D, which all require their own navlog calculations, mass and balance, and fuel plan. This is obviously too complex, and so I've decided to take it down so I can only focus on one route from A-B. If the user wants to go via many airports, at this stage they would have to make more than one route. This is a feature that I may decide to add back later when the project is finished and time constraints are absent.

Iteration 6 - Route map: Initialising the map

Route.html

```
<link rel="stylesheet" href="https://unpkg.com/leaflet@1.9.4/dist/leaflet.css"
integrity="sha256-p4NxAoJBhIIN+hmNHzRCf9tD/miZyoHS5obTRR9BMY=" crossorigin="" />
<script src="https://unpkg.com/leaflet@1.9.4/dist/leaflet.js"
integrity="sha256-20nQCchB9co0qIjJZRGuk2/Z9VM+kNiyxNV1lvTlZBo="
crossorigin=""></script>
```

```
<!--MAP-->
<div id="map" class="routeItem">

<div id="routeMAP"></div> <!--setting up the actual map-->
<style>#routeMAP { height: 100%; border-radius: 10px; z-index: 90;}</style>
<script>
  var map = L.map('routeMAP').setView([51.505, -0.09], 13);
  L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
    maxZoom: 19,
    attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
  }).addTo(map);
</script>
```

What I did

Moving onto the route map, I decided to use leaflet, as it was compatible with OpenAIP's Tiles API (which provides the map layer containing all the airspace and airport data).

First, I wanted to initialise the map in the space that I provided for it, and make sure it loaded properly. Leaflet provides a brilliant [quick start guide](#), which is what I followed to get the successful output shown in the video. Mind that this layer is just a layer of the OpenStreetMap API but does not have any airspace/airport data from any sort of OpenAIP layer yet- this will be implemented next.

Why I did it this way

Leaflet was one of many third-party map clients I could have used. When working on the feasibility part of this project, I had chosen leaflet as the client I would use, because It works solely in JS, and seemed super easy to use. I also wanted to see if OpenAIP was compatible with it at the time, and by looking at this [catalog](#) I found that it was, and so I had made my final decision.

Some more prevalent alternatives were OpenLayers and MapBox. The latter required you to pay to use it which obviously wasn't useful for my case, and the former seemed much more rigorous to use as you had to run items in the console which I feared might interfere with

flask. Leaflet just requires you to put in the relevant links in the headers, and the HTML loads from that source making it extremely easy and user friendly to access.

Program output

 map test 1.mp4

Known Limitations / Missing features

- Missing the OpenAIP layer

Test No.	Test Description	Expected Outcome	Actual Outcome
70	Map test (initialisation)	The leaflet map showing a layer from OpenStreetMap should be displayed in full width/height in the designated div for the map.	PASS (See video output)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Adding the OpenAIP layer

Iteration 7 - Route map: Adding the OpenAIP layer

app.py

```
#-----ROUTE MAP-----  
  
#Send the API key to frontend  
@main.route("/get-api-key")  
def get_api_key():  
    return jsonify({"api_key": api_key_openaip}), 200
```

Route.html

```
<script>
//getting API key from backend
fetch('/get-api-key')
  .then(response => response.json())
  .then(data => {
    const apiKey = data.api_key;

    //Basemap layer from OpenStreetMap
    const Basemap = L.tileLayer('https://tile.openstreetmap.org/{z}/{x}/{y}.png', {
      maxZoom: 19,
      attribution: '&copy; <a href="http://www.openstreetmap.org/copyright">OpenStreetMap</a>'
    });

    //OpenAIP layer for airspace, airports, navaids, etc.
    const OpenAIP = L.tileLayer('https://api.tiles.openaip.net/api/data/openaip/{z}/{x}/{y}.png?apiKey=${apiKey}', {
      maxZoom: 18,
      attribution: '<a href="https://www.openaip.net">OpenAIP</a>'
    });

    //crafting the map
    const map = L.map('routeMAP', {
      center: [51.505, -0.09], //Initial center coords (set to London)
      zoom: 9,
      layers: [Basemap, OpenAIP]
    });
  })
  .catch(error => {
    console.error('Error fetching API key:', error);
    showAlert('Error fetching map data. Please try again later.');
```

What I did

As it was for OpenAIP's core API (with all the airport info), you require a key to use it, and this is the same for their [Tiles API](#). The problem here is that my key is stored in a dotenv file, which, from what I know, can't be accessed by javascript. So, the only way to get the key to use it was to get it from app.py. The function `get_api_key()` simply sends over the key that was already declared when I was accessing the core API. Once received in javascript it can then be used as an additional endpoint for the Layer's URL.

To make it work I had to rearrange the function so that the whole thing runs under the fetch-api-function and depends solely on that operation working. This way the map shouldn't generate at all if an error was to occur. Each layer has a name, with OpenStreetMap being the base map and OpenAIP's layer on top.

Why I did it this way

OpenAIP needs a base layer to be used as it just provides airspace/navigation data and does not provide any sort of map. So without OpenStreetMap it would look like this:

Test No.	Test Description	Expected Outcome	Actual Outcome
71	Complete map text (both layers)	The map should appear with the base map (Open street) on the bottom with airspace and airport data from OpenAIP on top.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Route logistics

Iteration 8 - Route map: Adding a satellite view

route.js

```
//satellite map from mapTiler
const mapTilerLogo = document.createElement('img');
mapTilerLogo.src = 'https://latestlogo.com/wp-content/uploads/2024/04/maptiler-logo.png';
mapTilerLogo.style.width = '70px';
mapTilerLogo.style.marginTop = '3px';

const Satellite = L.tileLayer(`https://api.maptiler.com/maps/satellite/{z}/{x}/{y}.jpg?key=${apiKeyMaptiler}`, {
  maxZoom: 19,
  attribution: `<a href="https://www.maptiler.com">${mapTilerLogo.outerHTML}</a>`
});

//finding the Basemap style from settings
fetch('/get-settings')
  .then(response => response.json())
  .then(settingsData => {
    const mapStyle = settingsData.map_style;
    if (mapStyle == 'normal') {
      //crafting the map
      const map = L.map('routeMAP', {
        center: [51.505, -0.09], //Initial center coords (set to London)
        zoom: 9,
        layers: [Basemap, OpenAIP]
      });
    } else if (mapStyle == 'satellite') {
      //crafting the map
      const map = L.map('routeMAP', {
        center: [51.505, -0.09], //Initial center coords (set to London)
        zoom: 9,
        layers: [Satellite, OpenAIP]
      });
    }
  })
}
```

app.py

```
146 data_template = {
147     "settings" : {"theme": "light",
148                 "map_style": "normal",
149                 "units_changed": "False",
150                 "units_airspeed" : "knot",
```

```
#Send the API keys to frontend
@main.route("/get-api-keys")
def get_api_keys():
    return jsonify({"openaip": api_key_openaip, "maptiler": api_key_maptiler}), 200
```

route.html

```
<!--Map style toggle-->
<h2>MAP STYLE</h2>
<button id="Normal" class="{{ 'active' if settings.map_style == 'normal' else '' }}">Normal</button>
<span><button id="Satellite" class="{{ 'active' if settings.map_style == 'satellite' else '' }}">Satellite</butt
```

What I did

This was not planned, but I was playing around with the layers I could use (by just replacing the OpenStreetMap link with other links) and found a nice satellite layer - then I realised that satellite layer would be brilliant for reasons I explain below.

Initially I found Google's satellite tile, but it was raw, meaning I did not have to go via their API to access it, which violates their ToS... so I used mapTiler instead, which also provides a satellite map, although not with that top-notch quality that Google offers. It was still good enough for me though. Regardless, here's how I did it:

MapTiler requires an API key too. So, after making my account and getting a key, I added it to my .env file with the other keys, and fetched this key in **app.py** along with the API key for OpenAIP. I changed the `/get-api-key` function to a `/get-api-keys` function, and returned both keys rather than just the OpenAIP one. The rest was the same in terms of plugging in the API key into the link, and creating the layer.

To toggle between the normal and satellite maps, I created another settings item, called `'map_style'`, and set it to normal by default. I then coded the HTML and CSS for the buttons, and used the `update()` function to update the `map_style` depending on the button pressed. Over in **route.js**, two different map creation methods were made to facilitate the type of `map_style` selected. As you can see, if the map style is set to normal, OpenStreetMap becomes the base layer, and if set to satellite then MapTile's satellite is set to the base map. Upon reloading, since the style of the map is determined by the session, it remains as it was left on, resulting in a fully functional toggle system.

Why I did it this way

I chose a satellite layer because they are particularly useful for viewing land features in real-life. As a pilot flying in Visual Flight Rules, it's important to know what the ground features we plan to fly over actually look like, hence the satellite view. SkyDemon has this, OpenAIP has this, and other software also have this feature, and it seemed doable and so I implemented it. The absence of this feature would mean the user would have to access google maps or other satellite imagery outside of the webtool, which is not good from a consistency standpoint.

An alternative to having the toggle in the settings menu was to have it on the route page itself, somewhere near the map. This is what I designed at first, but it meant I needed to

find out how to make a toggle switch (Which, despite the resources being available, I didn't want to waste any extra time on), and updating the session based on the toggle. To me I thought this would be harder, but I guess we'll never know. But this wasn't the primary reason.

The main reason why this was a bad idea was because I needed to update the map in real time once the switch was toggled. This brought all sorts of complications (mostly being I didn't know how to do it, and didn't want to waste time learning it) that I didn't want to deal with, and by using a toggle in settings instead, pressing the back button reloads the previous page anyway and so any changes are automatically applied, which seamlessly eliminated the issue.

Program output

 satellite switch test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
72	Satellite toggle test	Toggling between map styles via settings should apply the appropriate map style on the map.	PASS (see program output)

Iteration Review

Met requirements: N/A

Changes: Added satellite toggle

Next steps: The route

Iteration 9 - Styling ICAO route and route generation sections

route.css

```
#ICAOroute{
  padding: 5px;
  background-color: var(--bg);
  color: var(--text);
  height: 100%;
  width: 100%;
  border-radius: 10px;
  resize: none; /*do not allow resizing*/
  font-family: 'Manrope';
  font-size: 14px;
  border: 1px solid var(--onhover);
```

```

}

#AI_twinkle {
  margin-right: 6px;
  vertical-align: middle;
}

#routePrompt {
  padding: 5px;
  margin-top: 10px;
  background-color: var(--highlight);
  color: var(--text);
  border-radius: 10px;
  resize: none;
  font-family: 'Manrope';
  font-size: 14px;
  width: 100%;
  height: 76.5%;
}

#route_generator {
  display: grid;
  grid-template-columns: 1fr 1fr; /*display them side by side*/
  gap: 10px;
}

#routeOutput {
  background-color: var(--bg);
  color: var(--text);
  border-radius: 10px;
  resize: none;
  font-family: 'Manrope';
  font-size: 14px;
  width: 100%;
  border: 1px solid var(--onhover);
}

#generateRoute {
  color: var(--text);
  background-color: var(--onhover);
  border: none;
  border-radius: 10px;
  padding: 10px;
  font-size: 16px;
  cursor: pointer;
  width: 100%;
  font-family: 'Manrope';
  margin-top: 5px;
}

#generateRoute:hover {

```

```
background-color: var(--primary);
}
```

route.html

```
<!--ROUTE IN ICAO FORMAT-->
<div id="ICAO_route" class="routeItem">
  <textarea id="ICAOroute" name="ICAOroute" rows="4" cols="50" placeholder="Start typing a route in ICAO format,"
  </div>
<!--AI ROUTE GENERATOR-->
<div id="route_generator" class="routeItem">
  <div>
    <h3><svg xmlns="http://www.w3.org/2000/svg" id="AI_twinkle" height="20" viewBox="0 0 20 20" width="20"><path d=
    <textarea id="routePrompt" name="routePrompt" rows="4" cols="50" placeholder="Type a prompt here or just press
    <button id="generateRoute" class="button TARGET">Generate</button>
  </div>
  <div id="routeOutput">

  </div>
</div>
```

What I did

Before I moved onto the routing section of the route tab, I decided to finish off the styling for the page. Generally, there were not many issues in doing this and I managed to get it exactly matching the design, the only difference being that I added an outline around any text boxes that matches the color used for hovering over a button (i.e **--onhover**).

Other things worth mentioning included the use of the **<textarea>** element instead of the simple **<input>** element, which allowed for bigger boxes that could be styled freely as seen below.

Why I did it this way

The sole reason I've done this styling first before thinking about the route is because the ICAO route and the route generator work hand-in-hand, and both must be considered when finding out the optimal way of storing the user's route of choice. Given that this requires the use of available UI to input/output these routes, if I never finished the UI, the testing will be done on what won't be the final appearance, which is slightly inconvenient. Especially since in this case, the UI has no effect at all on the way the route will work, not doing it now all of a sudden becomes more of a missed opportunity to make a later coding experience much easier and smoother, as each time you're already experiencing the final product.

The outlines on the text boxes will become a consistent feature throughout each page. From my perspective I think it makes the UI a little cleaner and is more aesthetically pleasing to the user. It also just makes them easier for the user to see them as input boxes rather than something else potentially. It's key that I keep the design like this onwards to avoid mishaps where the user expects an input but instead gets a button, or something unusable, etc.

Program output

☰

ROUTE

🗺️

Route Setup ⓘ

From: EGSC CAMBRIDGE → To: EGML DAMYNS HALL

Alternate: EGKL DEANLAND

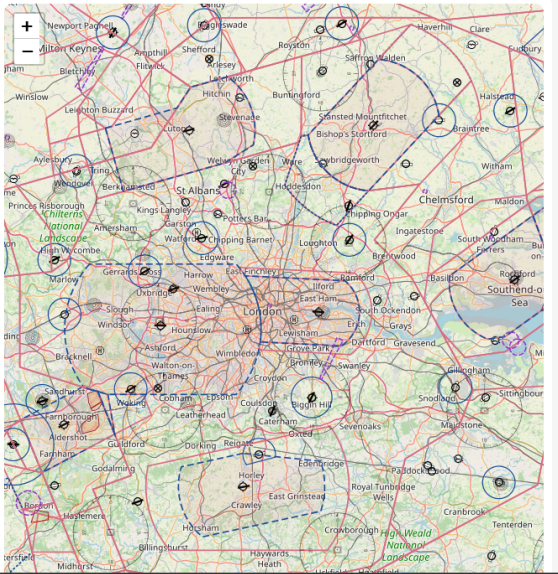
Start typing a route in ICAO format, or generate a route below.

✦ Route generator

Type a prompt here or just press 'Generate' to generate a route based on the current route setup. You can then edit it there and then, either manually or with another prompt.

UTC 22:37 | LOCAL 23:37

10.0 NM



☰

ROUTE

🗺️

Route Setup ⓘ

From: EGSC CAMBRIDGE → To: EGML DAMYNS HALL

Alternate: EGKL DEANLAND

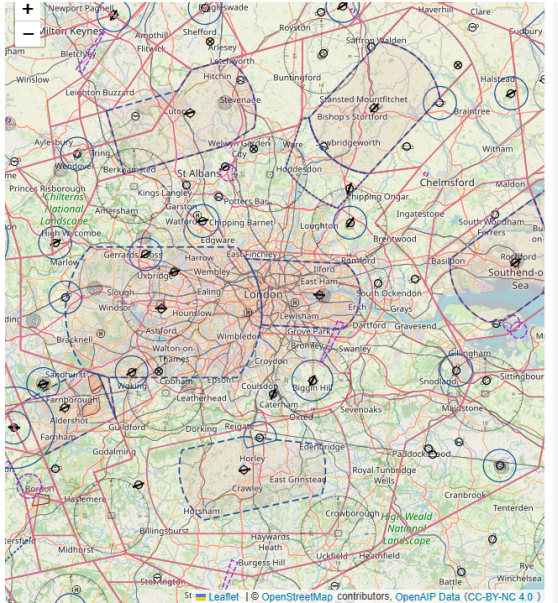
Start typing a route in ICAO format, or generate a route below.

✦ Route generator

Type a prompt here or just press 'Generate' to generate a route based on the current route setup. You can then edit it there and then, either manually or with another prompt.

UTC 22:37 | LOCAL 23:37

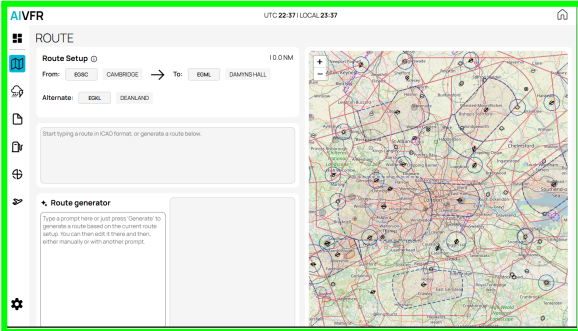
10.0 NM



Known Limitations / Missing features

- Route generator and route not functional

214

Test No.	Test Description	Expected Outcome	Actual Outcome
73	Route Tab design comparison	The final design of the route tab should match the layout of that planned in the design.	

Iteration Review

Met requirements: N/A

Changes: Added outline to input boxes

Next steps: The route plan

Iteration 10 - Route Plan: Drawing on the map

This iteration is primarily supposed to be dedicated to me finding the proper way to establish a full route that is compatible with the map. This is key, because the route needs to be able to be shown on this map, not just the route itself. Ideally it must be so that when the route is edited, the map updates with said edits.

route.js

```
function addLine(startCoords, endCoords) {
  var polyline = L.polyline([startCoords, endCoords]).addTo(map);
  return polyline;
}
```

```
if (mapStyle == 'normal' && theme == 'light') {
  mapTilerLogo.style.display = "none";
  //crafting the map
  const map = L.map('routeMAP', {
    center: [51.505, -0.09], //Initial center coords (set to London)
    zoom: 9,
    layers: [Basemap, OpenAIP]
  }); addLine([51.4700, -0.4543], [52.2053, 0.1218]); //Example line from EGLL to EGSS
```

What I did

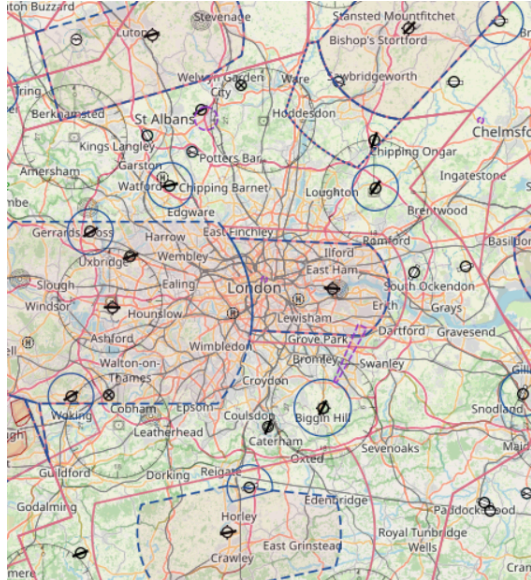
To start off things, I decided to find out how to draw a line onto the map. The leaflet quick start guide also has a tutorial for this, and so I decided to quickly paste it into my code and test to see if it works, which it did. From what I saw, I thought that since a line can take coordinates, a function `addLine()` would allow me to put in the coordinates of the line I would like to make on the map and have it drawn.

Why I did it this way

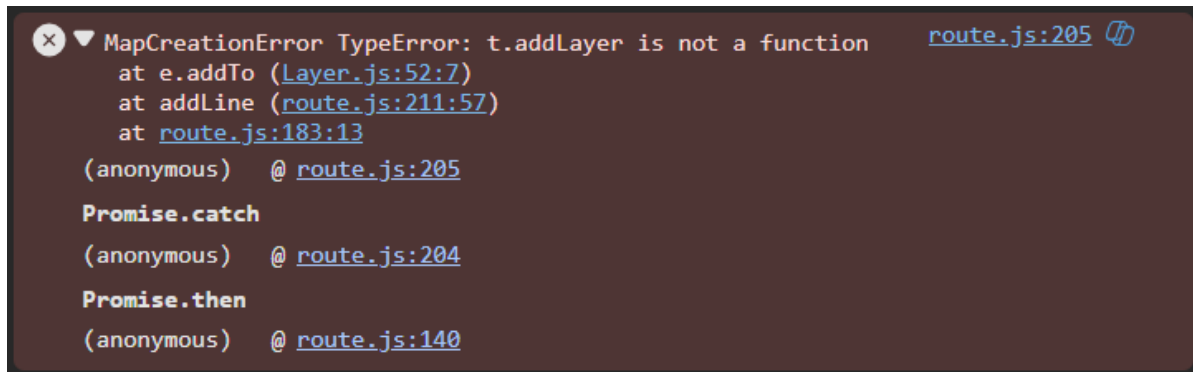
I started with finding out how to draw a line because the route revolves around how it can be represented on the map. If I started with thinking of the other parts of the route first without considering the map, I would have a hard time trying to figure out how my ready-made solution to the route would be able to be drawn onto the map, which would take a lot of time and effort.

Program output

Map (no line shown)

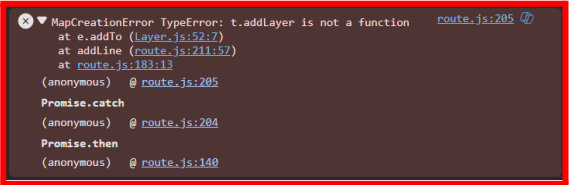


F12 DevTools



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
74	Line drawing test	The addLine function should draw a line on the map linking the two parameter coordinates.	

Error Analysis & Fixes

Test No: 74

Failure Description: TypeError appears and line not drawn on map.

Cause: Unknown. I have been searching everywhere for the solution but nothing seems to be helping - a lot requires me to re-write my code in an undesirable way.

Fix: Since the line worked without the function, I'll try keeping it like that and finding a way to still be able to draw a line that was pre-defined elsewhere (so that the route can be made elsewhere then put into this, so that the lines are then drawn for that route)

Justification: The aim is to have some sort of way to pre-define the route. Carrying on with making this addLine function work wouldn't be smart because the changes required to make this work may influence other parts of my code. Going off of what already works ensures that not much has to be changed, which makes it easier for me.

Iteration 10.5 - Eliminating the TypeError

route.js

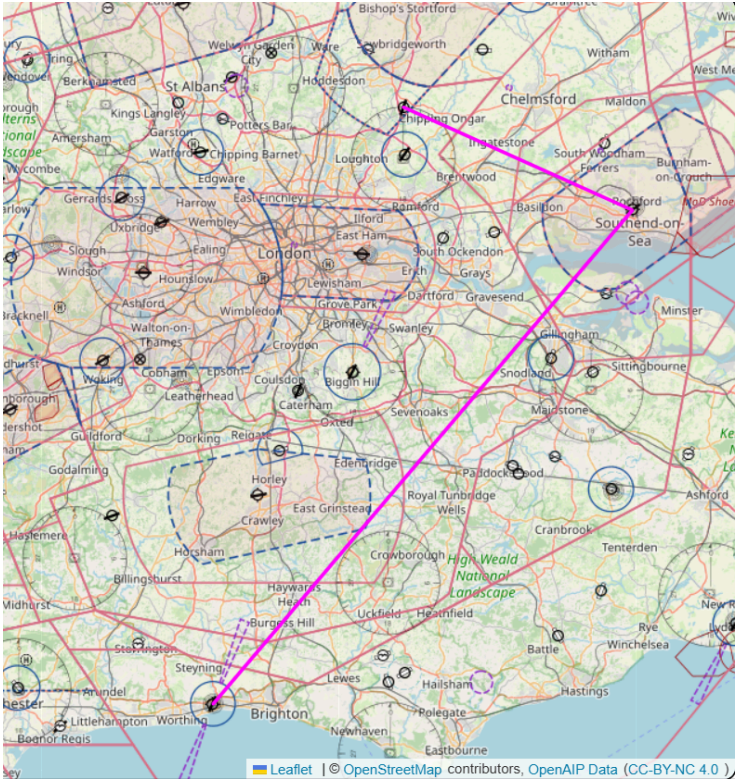
```
if (mapStyle == 'normal' && theme == 'light') {
  mapTilerLogo.style.display = "none";
  //crafting the map
  const map = L.map('routeMAP', {
    center: [51.505, -0.09], //initial center coords (set to London)
    zoom: 9,
    layers: [Basemap, OpenAIP]
  }); const line = L.polyline([[51.722000, 0.154000], [51.571000, 0.696000],
[50.835556, -0.297222]],
  { color: '#f0F' }).addTo(map); //f0F is magenta, usually used for nav
```

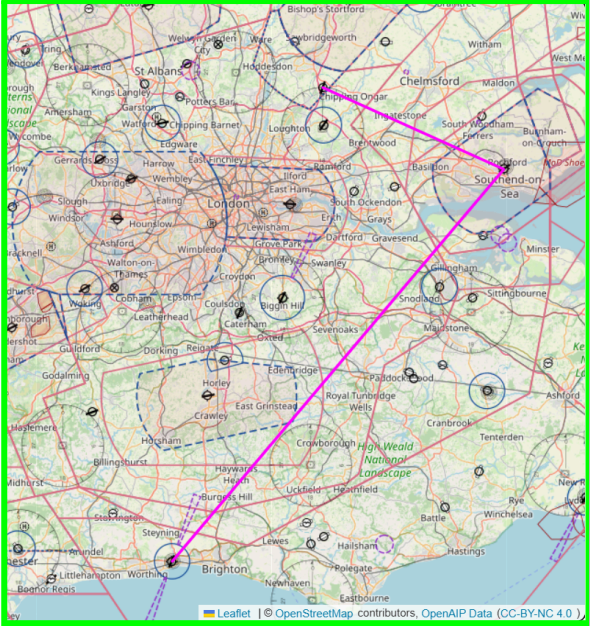
Here, I've just put the same code under the map generation code, which draws lines between three airports, whose coordinates I got from OpenAIP. This actually did draw on the map as you will see, which means, I would only need an array of coordinates for the route store it in a variable, and just replace the coordinates with that variable, to get something like this:

```
const line = L.polyline([route_coords],
  { color: '#f0F' }).addTo(map); //not yet coded just typed up here
```

This means that I can do anything to formulate a route so long as I can get the coordinates for each point and store them here.

Program output



Test No.	Test Description	Expected Outcome	Actual Outcome
75	Line drawing test	A line should be drawn on the map between North Weald, Southend and Shoreham airports.	

Iteration Review

Met requirements: N/A

Changes: Removed addLine() Function

Next steps: Making, storing and drawing a route.

Iteration 11 - Route Plan: Drawing the Departure-Arrival line

app.py

```
#-----FINDING COORDINATES OF AIRPORTS, NAVAIDS AND VRPs

@main.route("/get-avCoords", methods=["POST"])
def get_avCoords():
    data = request.json.get("data")

    headers = {
        "x-openaip-api-key": api_key_openaip #correct header for OpenAIP API
    }
    params = {
        "search": str(data)
    }

    #search for all 3 items
    urlAirports = "https://api.core.openaip.net/api/airports"
    urlNav aids = "https://api.core.openaip.net/api/nav aids"
    urlVrps = "https://api.core.openaip.net/api/reporting-points"

    response_airports = requests.get(urlAirports, headers=headers, params=params)
    response_nav aids = requests.get(urlNav aids, headers=headers, params=params)
    response_vrps = requests.get(urlVrps, headers=headers, params=params)

    data_airports = response_airports.json() #parse the response as JSON
    data_nav aids = response_nav aids.json()
    data_vrps = response_vrps.json()

    airport = data_airports["items"][0]
    nav aid = data_nav aids["items"][0]
    vrp = data_vrps["items"][0]

    #collecting relevant info
    icao_code = airport["icaoCode"]

    nav aid_name = nav aid["name"]
```

```

navaid_country = navaid["country"]

vrp_name = vrp["name"]
vrp_country = vrp["country"]

#first, check if the entered data is an airport
if icao_code == data:
    return jsonify({"coordinates" : airport["geometry"]["coordinates"]}), 200
#^^no need to verify country as already done in Verify ICAO

#if that doesnt work check if its a navaid
elif navaid_name == data:
    if navaid_country == 'GB':
        return jsonify({"coordinates" : navaid["geometry"]["coordinates"]}), 200
    else:
        return jsonify({"error": "Point not found in UK database"}), 400

#if that doesnt work check if its a reporting point
elif vrp_name == data:
    if vrp_country == 'GB':
        return jsonify({"coordinates" : vrp["geometry"]["coordinates"]}), 200
    else:
        return jsonify({"error": "Point not found in UK database"}), 400

#if that doesn't work, throw an error as point hasn't been found
else:
    return jsonify({"error": "Point not found in UK database"}), 400

```

```

#-----ROUTING-----
#-----ADDING/REMOVING WAYPOINTS

@main.route("/add-waypoint", methods=["POST"])
def add_waypoint():
    waypoint = request.json.get("waypoint") #expecting a list [latitude, longitude]
    session["flight_data"]["flight"]["route"].append(waypoint)
    session.modified = True
    return jsonify(session["flight_data"]["flight"]["route"]), 200

@main.route("/remove-waypoint", methods=["POST"])
def remove_waypoint():

```

route.js

```

arrivalAirport_name.textContent = airportDetails.name;

/*when arrival aerodrome is entered, fetch the coords for departure and
arrival for intial route drawing*/
const departureCoords = fetchAvCoords(departureAirport_code.value);
const arrivalCoords = fetchAvCoords(arrivalAirport_code.value);

```

```
//-----adding a waypoint

async function add_waypoint(waypoint) {
  try {
    const response = await fetch("/add-waypoint", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ waypoint: waypoint }), //sends off the corrdinates to the backend
    });

    if (!response.ok) {
      throw new Error("Network response was not ok"); //for non-200 errors
    }

  } catch (error) {
    showAlert("An error occured whilst adding the waypoint. Please try again.");
    return null;
  }
}
```

```
//-----Fetching the coords for airport related features-----

async function fetchAvCoords(point) {
  try {
    const response = await fetch("/get-avCoords", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ waypoint: point }), //sends off the code to the backend
    });

    if (!response.ok) {
      throw new Error("Network response was not ok"); //for non-200 errors
    }

    const data = await response.json();
    return data; //returns the coordinates

  } catch (error) {
    showAlert("Waypoint not found. Please check the name and location of your waypoint and try again.");
    return null;
  }
}

//-----Adding an aviation feature as a waypoint

async function add_av(avPoint) {
  AvCoords = await fetchAvCoords(avPoint);
  await add_waypoint(AvCoords);
}
```

What I did

The first thing I wanted to work on drawing was that line from the departure to the arrival aerodrome. This gives the user a good idea of where they're travelling before they begin to input any turning points.

For this to work, I needed a way to add a waypoint's coordinates to the route, which is an array now added as a value in the session data file. OpenAIP allows the user to get airports, navigation aids and reporting points, all which can be used to navigate. Since future waypoints may be of any of these types, I've decided to make one function that

accounts for all 3 types of aviation-related waypoints, called `get_avCoords()`, which, to be very brief, runs the API 3 times, each for airports, navigation aids, and reporting points, and checks which one of them matches the data inputted, and then returns the relevant coordinate for that point if the country they're situated in is indeed within the UK.

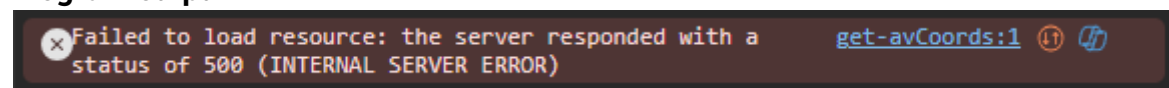
These coordinates are taken back to the fronted js, which can then be immediately added to the route via the `add_waypoint()` function, which sends the coordinates back to the backend to be added to the route array via the session.

Why I did it this way

The logic for adding the departure-arrival line was to have it done after the arrival aerodrome was inputted in. Most likely, the user will have put in a departure aerodrome first before entering, and so this makes it easier in the sense that I don't need to make a button to 'start drawing' the route or something.

I could have also waited to draw the whole route once the route was actually finished. This could in theory make some things easier, but I'd rather make the route as the user updates it to match what I planned on the design. Also, it can take a while to iterate through a long route list and search for all the waypoint's coordinates.

Program output



```
File "/workspaces/AIVFR/AIVFR/app.py", line 348, in get_avCoords
    navaid = data_navaids["items"][0]
            ~~~~~^
IndexError: list index out of range
127.0.0.1 - - [25/Oct/2025 09:36:34] "POST /get-avCoords HTTP/1.1" 500 -
```

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
76	Departure → Arrival route line drawing test	When the arrival aerodrome is entered, the departure and arrival airport coordinates should be added to the route	ERROR

Error Analysis & Fixes

Test No: 76

Failure Description: Index error occurs when trying to fetch the airport's coordinates

Cause: Currently unknown but most likely to do with what's been sourced out of the API, perhaps something erroneous happened that meant no airport was outputted?

Fix: Check the data imputed to make sure it's the airport code (so the input is clean), then check outputs being made by the API to determine if the API data has been outputted correctly or even fetched at all.

Justification: I could also revert to having the code go through the entire process one by one (instead of doing 3 things grouped at very short intervals) but this lengthens the code and makes it inefficient. There should be a way to do it in my method that allows the data to be searched for swiftly rather than running a whole function a bunch of times, only for it not to work, for example.

Iteration 11.5 - Eliminating the IndexError in the API fetch

Once I verified that the input was ok, when I checked the API output there was nothing. So I looked before the API was called and I noticed the error.

These lines of code:

```
airport = data_airports["items"][0]
navaid = data_navaid["items"][0]
vrp = data_vrps["items"][0]
```

Take the first group of items from the "items" array that contain the main info for each type of point. (e.g. name, location, etc). They however rely on these lines of code:

```
data_airports = response_airports.json()
data_navaid = response_navaid.json()
data_vrps = response_vrps.json()
```

Which parses the retrieved data into a json format, but these rely on these lines of code...

```
response_airports = requests.get(urlAirports, headers=headers, params=params)
response_navaid = requests.get(urlNavaid, headers=headers, params=params)
response_vrps = requests.get(urlVrps, headers=headers, params=params)
```

Which will only pass data if they find a matching name from the parameters (i.e, the data).

Essentially what was happening is that, the first time it ran the airports data request, it worked as the parameter matched the identifier for an airport. But when it moved on to get the request to a navaid, it returned nothing, (since there was nothing that matched), and so that got parsed into nothing (I know this because the output was just this: {}), which caused the index error when the line `navaid = data_navaid["items"][0]` runs which, because of the way I structured the code, **must** run.

The obvious fix for this was to use try blocks, and instead of if statements an `except IndexError` statement allowed me to catch those index errors as a failed attempt to find the waypoint in the database, and so instead of stopping the code right there, it moves on to try the next option. Now the refined code looks like this:

app.py

```
#first, check if the entered data is an airport
try:
    airport = data_airports["items"][0]
    icao_code = airport["icaoCode"]
    if icao_code == data:
        return jsonify({"coordinates" : airport["geometry"]["coordinates"]}), 200
    #^^no need to verify country as already done in Verify ICAO

#if that doesnt work check if its a navaid
except IndexError:
    try:
        navaid = data_navaids["items"][0]
        navaid_name = navaid["name"]
        navaid_country = navaid["country"]
        if navaid_name == data:
            if navaid_country == 'GB':
                return jsonify({"coordinates" : navaid["geometry"]["coordinates"]}), 200
            else:
                return jsonify({"error": "Point not found in UK database"}), 400

    #if that doesnt work check if its a reporting point
    except IndexError:
        try:
            vrp = data_vrps["items"][0]
            vrp_name = vrp["name"]
            vrp_country = vrp["country"]
            if vrp_name == data:
                if vrp_country == 'GB':
                    return jsonify({"coordinates" : vrp["geometry"]["coordinates"]}), 200
                else:
                    return jsonify({"error": "Point not found in UK database"}), 400

        #if that doesnt work, throw an error as point hasnt been found
        except IndexError:
            return jsonify({"error": "Point not found in UK database"}), 400
```

As you can see each item is a try block, with trying to find the waypoint in the index as the first instruction. If an index error is brought for finding an airport, then 'try' find a navigation aid. If that brings an error too, 'try' find a reporting point. If that fails too, then throw an error, because it was not found in all 3 cases meaning the input was invalid.

Program output

 map drawing test.mp4

Test No.	Test Description	Expected Outcome	Actual Outcome
77	Departure → Arrival route line drawing test	When the arrival aerodrome is entered, the departure and arrival airport coordinates should be added to the route	PASS (see program output)

Iteration Review

Met requirements: N/A

Changes: Used try blocks instead of if statements

Next steps: Adding cities and aviation features to the route

Iteration 12 - Route Plan: Adding cities and aviation features to the route

route.js

```
#-----FINDING CITY COORDS

@main.route("/get-cityCoords", methods=["POST"])
def get_cityCoords():
    city = request.json.get("city")
    api_url = 'https://api.api-ninjas.com/v1/city?name={}'.format(city)
    response = requests.get(api_url, headers={'X-API-Key': api_key_api_ninjas})

    if response.status_code == requests.codes.ok:
        data = response.json()
        if data:
            city_info = data[0] #Get the first matching city
            #get the necessary info
            name = city_info.get('name')
            country = city_info.get('country')
            latitude = city_info.get('latitude')
            longitude = city_info.get('longitude')

            #check for a matching city in the UK
            if name.lower() == city.lower() and country == "GB":
                return jsonify({
                    "name": name,
                    "coordinates" : [latitude, longitude]
                }), 200
            else:
                return jsonify({"error": "City not found"}), 400
        else:
            return jsonify({"error": "City not found"}), 400
```

```
//-----dealing with user-inputted waypoint-----

const waypoint = document.getElementById('searchWaypoint')
const waypointForm = document.getElementById('addWaypointForm')

//wait for a submission by the user
waypointForm.addEventListener('submit', function(event) {
  event.preventDefault();

  //get the current waypoint type from settings
  //(to know whether to add a city or an aviation feature)
  fetch('/get-settings')
  .then(response => response.json())
  .then(async data => {
    const addType = data.waypoint_addType;

    if (addType == 'City/Town') {
      await add_city(waypoint.value);
      waypoint.value = '';
      reload_map();
    } else if (addType == 'VRP/NAVAID/Airport') {
      await add_av(waypoint.value);
      waypoint.value = '';
      reload_map();
    }
  });
});
```

```
//-----Picking a waypoint-----

//switching between city/town or an aviation feature:

function updateSettings(key, value) {
  fetch("/update-settings", {
    method: "POST",
    headers: { "Content-Type": "application/json" },
    body: JSON.stringify({ key, value })
  });
}

const CityOrTown = document.getElementById('CityOrTown')
const avFeature = document.getElementById('avFeature')
const searchWaypoint = document.getElementById('searchWaypoint')

CityOrTown.addEventListener('click', () => {
  CityOrTown.classList.add('active')
  avFeature.classList.remove('active')
  searchWaypoint.placeholder = 'Enter a City or Town name'
  updateSettings("waypoint_addType", "City/Town")
})

avFeature.addEventListener('click', () => {
  CityOrTown.classList.remove('active')
  avFeature.classList.add('active')
  searchWaypoint.placeholder = 'Enter ICAO code, Navaid or VRP'
  updateSettings("waypoint_addType", "VRP/NAVAID/Airport")
})
```



```
//-----Updating map without reloading page

function reload_map() {
  //removes map instance without removing the whole container
  if (map) {
    try {
      map.off(); //detaches any event listeners (i dont have any here anyways)
      map.remove(); //removes the whole contaier
    } catch (error) {
      console.warn("Error removing map during reload:", error);
      showAlert('An error occured. Please reload the page')
    }
    map = null; //container then re innitalized but set to null
  }
  load_map();
}
}
```

```
//-----adding a city or town as a waypoint
async function add_city(city) {
  //first, check the city is in the cities database
  try {
    const response = await fetch("/get-cityCoords", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ city: city }), //sends off the city to the backend
    });

    if (!response.ok) {
      throw new Error("Network response was not ok"); //for non-200 errors
    }

    const data = await response.json();

    //then, add the city to the route
    await add_waypoint(data.coordinates);

  } catch (error) {
    showAlert("Error fetching city details. Please make sure the city name is correct.");
    return null;
  }
}
}
```

route.css

```
#ICAO_route {
  display: grid;
  grid-template-columns: 1.5fr 1fr;
  gap: 10px;
  height: 160px;
}

#addWaypointContainer {
  padding: 10px
```

```

}

.waypoint_choice, #addWaypoint {
  width: auto;
  font-family: 'Manrope';
  height: auto;
  font-size: 13px;
  padding: 7px;
  border-radius: 8px;
}

.waypoint_choice {
  background-color: var(--onhover);
  margin-bottom: 10px;
  padding-left: 11px;
  padding-right: 11px;
}

#addWaypoint{
  width: 100%;
  background-color: var(--bg);
}

#searchWaypoint {
  width: 100%;
  margin-bottom: 10px;
  height: 30px;
  border-radius: 5px;
  border: 1px solid var(--onhover);
  background-color: var(--bg);
  font-family: 'Manrope';
  padding-left: 5px;
}

#route_plan {
  margin-bottom: 10px;
}

#CityOrTown.active, #avFeature.active {
  background-color: var(--primary);
}

```

route.html

```

<!-------ROUTE----->
  <div id="ICAO_route" class="routeItem">
    <div>
      <h3 id="route_plan">Route Plan</h3>
      <textarea id="ICAOroute" name="ICAOroute" rows="4" cols="50" placeholder="Your
route will appear here."></textarea>
    </div>

    <div id="addWaypointContainer">

```

```

        <button class="button waypoint_choice {{ 'active' if settings.waypoint_addType ==
'City/Town' else '' }}" id="CityOrTown">City/Town</button>
        <span><button class="button waypoint_choice {{ 'active' if
settings.waypoint_addType == 'VRP/NAVAID/Airport' else '' }}"
id='avFeature'>VRP/NAVAID/Airport</button></span>

        <form id="addWaypointForm" enctype="multipart/form-data">
            <input id="searchWaypoint" type="text" name="searchWaypoint" placeholder="{{
'Enter a City or Town name' if settings.waypoint_addType == 'City/Town' else 'Enter ICAO code,
Navaid or VRP' }}" value="">
            <span><button type="submit" class="button TARGET" id="addWaypoint">Add to
Route</button></span>
        </form>

    </div>
</div>

```

What I did Map refreshing

The first thing I needed to be able to do before I continued was to have the map be able to update without me having to reload the page. The simple way to do this was to use the built-in `.remove()` command, which just removes the whole container, then, reload the map with the new settings. This worked initially, but it broke because when the container got removed there was no container to initialize the map on anymore, which made things a little more difficult. The final solution was to do the same methods of using `.remove`, but instead of just leaving it like that, I re-initialize the map by setting it to null, that way the map remains unloaded but initialized. I then encased all that map creation code from before into a single `load_map()` function, which I then ran again at the end of the `reload_map()` function to load back up the map with any existing changes.

Adding a city

As well as being able to navigate through airports and navigational aids, the user can also navigate using reference to major landmarks such as cities and towns, which is extremely common when flying in Visual Flight Rules.

To add a city was just about the same process as adding an aviation feature. I used a [cities API](#) to get the cities, and used their provided guide in `app.py` to ensure I called it properly. Then, with the city found to be matching the inputted value, I grab the name (to be added to the user-readable route), the country (to verify that it's in the UK), the latitude and the longitude. If the country is indeed in the UK then it sends all 4 of those data values back to the frontend, where `add_city` adds the respective uses of the sent in coordinates and adds them to the route via the add waypoint function.

The new route-picking design

In this process, I had to completely re-design the 'ICAO Route' section, which is now just the 'Route plan'.

The route plan section now looks like this:

Route Plan

Your route will appear here.

City/Town

VRP/NAVAID/Airport

Enter ICAO code, Navaid or VRP

Add to Route

In the Big box, the route plan in an English format is written as the route is created. To the right, The user has the option to manually adjust their flight plan. If they want to add a city or town they select the city or town button, which has an eventListener attached to it so that the relevant setting in the session changes and the styling also changes. The logic is identical to the satellite-normal map toggle I implemented earlier.

Depending on the type of waypoint the user wants to add, pressing the ‘Add to Route’ will do something different, as seen in the function attached to the button’s eventListener. This ensures the user can add city and aviation-related waypoints seamlessly.

Why I did it this way

Reasoning for abandoning the route in ICAO format

To put it simply, the International Civil Aviation Organisation (ICAO) has a standardised way of writing a flight route. It looks a little bit like this:

TS, CRUISING SPEED	LEVEL	ROUTE
M 0 8 2	F 3 5 0	BASURA BASUR G462 ALMA/N0490 F350 B416
SHJ A418	KASOL/N0481F390 A418	SYZ R 659 ISN R654 ZAJ G208 UMH
G781 BONAM VG81	VAN UL852	UDROS UZ117 KOMAN UL602 NERDI UT4
DEGET UL851	SUBES UL610	BATTY UL608 LOGAN LAM3A

This is something similar to what the **airliners** use, which have different names for defined waypoints (e.g the waypoint **DEGET**) or high-level airways (e.g airway **UL852**). These are just defined points on a map and don’t represent visual points on the ground, as the planes that use these routes operate using **Instrument Flight Rules** and not **Visual Flight Rules**.

For my purposes of flying visually, however, it’s very different. For example, a flight from North Weald Airfield to Brighton City Airport, I might want a route that goes over visual reference points, such as:

(Not in ICAO format)

North Weald Airfield (**EGSX**) -> Chipping Ongar -> Queen Elizabeth II Bridge -> Royal Tunbridge Wells -> Eastbourne -> Brighton Marina -> Brighton City Airport (**EGKA**)

However, in ICAO format, such a route looks like this...

EGSX DCT 5143N00015E DCT 5128N00016E DCT 5108N00016E DCT 5046N00017E DCT **EGKA**

...Which is not readable by the user as it just gives the coordinates. Turns out visual reference points don't count as waypoints when they're put in ICAO format, it's just a point on the map.

Program output

 Route crafting test.mp4

Known Limitations / Missing features

- Route does not appear in big box
- Cannot remove from route
- Start and end aerodromes not integrated in the normal route creation
- Cannot append from a chosen index

Test No.	Test Description	Expected Outcome	Actual Outcome
78	Route Crafting test	When 'City/Town' is selected, typing correctly and adding the waypoint should add the relevant city. The same should apply to the 'VRP/NAVAID/Airport' button.	PASS (See program output)

Input Validation Tests

Test Data	Expected Outcome	Actual Outcome
Putting the name of a city when an aviation feature is expected	An error should occur stating the waypoint is not found.	PASS (See full test video)
Putting the name of an aviation feature when a city name is expected	An error should occur stating the city is not found.	PASS (See full test video)
Putting in a town that does not exist (Any random text)	An error should occur stating the city is not found.	PASS (See full test video)
Putting in an aviation feature that does not exist (Any random text)	An error should occur stating the waypoint is not found.	PASS (See full test video)

Full test video:  Waypoint drawing validation tests.mp4

Iteration Review

Met requirements: (Aim partially complete) Link the route to the session and allow it to be edited at any time by the user

Changes: Route no-longer in ICAO format by default

Next steps: Indexing, editing and crafting an entire route

Iteration 13 - Route Plan: Crafting an entire route with indexing

route.html

```
<!--Submission buttons to either add/remove waypoints-->
<div id="addOrRemoveWaypoints">
  <button type="submit" class="button TARGET" id="addWaypoint" name="action" value="add">Add waypoint</button>
  <span><button type="submit" class="button TARGET" id="removeWaypoint" name="action" value="remove">Remove waypoint
```

app.py

```
@main.route("/add-waypoint", methods=["POST"])
def add_waypoint():
    waypoint = request.json.get("waypoint") #expecting a list [latitude, longitude]
    name = request.json.get("name") #expecting the name of the waypoint

    #waypoints are always added before the destination airport (so one before the end)
    add_index = len(session["flight_data"]["flight"]["route_names"]) - 1

    #allow the initial addition of departure and destination airports
    if add_index <= 0:
        session["flight_data"]["flight"]["route_names"].append(name)
        session["flight_data"]["flight"]["route"].append(waypoint)

    #for adding waypoints in between departure and destination
    else:
        session["flight_data"]["flight"]["route_names"].insert(add_index, name)
        session["flight_data"]["flight"]["route"].insert(add_index, waypoint)

    session.modified = True
    return jsonify(session["flight_data"]["flight"]["route"]), 200
```

```
@main.route("/remove-waypoint", methods=["POST"])
def remove_waypoint():
    index = request.json.get("waypointIndex") #expecting the index of the waypoint to remove
    try:
        #remove the waypoint at the specified index
        session["flight_data"]["flight"]["route_names"].pop(index)
        session["flight_data"]["flight"]["route"].pop(index)
        session.modified = True
        return jsonify(session["flight_data"]["flight"]["route"]), 200
    except IndexError:
        return jsonify({"error": "Invalid waypoint index"}), 400
```

route.css

```
#addOrRemoveWaypoints {
  display: flex;
  gap: 5px;
}
```

```
#removeWaypoint:hover {
  background-color: hsla(0 100% 50% / 0.6);
}

#addWaypoint, #removeWaypoint {
  background-color: var(--bg);
  font-family: 'Manrope';
}
```

route.js

```
//-----ADDING AND REMOVING WAYPOINTS MANUALLY-----
//adding event listener to the waypoint form
const waypoint = document.getElementById('searchWaypoint')

//wait for a submission by the user
document.getElementById('addWaypointForm').onsubmit = async function(event) {
  event.preventDefault();

  //determine whether to add or remove a waypoint (by reading what is stored in the
value)
  const actionValue = (event.submitter && event.submitter.value) || new
FormData(this).get('action');

  //*****REMOVING A WAYPOINT*****
  if (actionValue === 'remove') {
    //first, get the index of the waypoint to be removed
    fetch('/get-flight')
      .then(response => response.json())
      .then(async data => {
        const route_names = data.route_names;
        const WaypointIndex = route_names.indexOf(waypoint.value.toUpperCase());

        //if the waypoint is not found, alert the user
        if (WaypointIndex == -1) {
          showAlert("Waypoint not found in the current route. Please check the name
and try again.");
          waypoint.value = '';
          return;

          //prevent removing departure or arrival aerodrome
        } else if (waypoint.value.toUpperCase() == data.departureAirport_code ||
waypoint.value.toUpperCase() == data.destinationAirport_code) {
          showAlert("You cannot remove the departure or arrival aerodrome from
here.");
          waypoint.value = '';
          return;
        }
      })
  }
}
```

```

//then, remove the waypoint using the index
try {
  const response = await fetch("/remove-waypoint", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ waypointIndex: WaypointIndex }), //sends off the
index to the backend
  });

  if (!response.ok) {
    throw new Error("Network response was not ok"); //for non-200 errors
  }

} catch (error) {
  showAlert("An error occured whilst removing the waypoint. Please try
again.");
  return null;
}

waypoint.value = '';
reload_map();
update_route_names();
});

//*****ADDING A WAYPOINT*****

} else if (actionValue === 'add') {

  //get the current waypoint type from settings
  //(to know whether to add a city or an aviation feature)
  fetch('/get-all')
  .then(response => response.json())
  .then(async data => {
    const addType = data.settings.waypoint_addType;
    const departureAirport = data.flight.departureAirport_code;
    const destinationAirport = data.flight.destinationAirport_code;
    const route_names = data.flight.route_names;

    //prevent adding duplicate waypoints
    if (route_names.includes(waypoint.value.toUpperCase())) {
      showAlert("This waypoint is already in the current route. Please choose
another waypoint.");
      waypoint.value = '';
      return;
    }

    //prevent adding departure or arrival airport as a waypoint

```



```

        if (waypoint.value.toUpperCase() == departureAirport ||
waypoint.value.toUpperCase() == destinationAirport) {
            showAlert("You cannot add the departure or arrival aerodrome as a
waypoint.");
            waypoint.value = '';
            return;

        } else if (departureAirport == '' || destinationAirport == '') {
            showAlert("Please make sure both departure and arrival aerodromes are set
before adding waypoints.");
            waypoint.value = '';
            return;
        }

        //if all is valid a waypoint can be added
        if (addType == 'City/Town') {
            await add_city(waypoint.value);
            waypoint.value = '';

        } else if (addType == 'VRP/NAVAID/Airport') {
            await add_av(waypoint.value);
            waypoint.value = '';
        }

        reload_map();
        update_route_names();
    });
}
};

```

What I did

To finish off the route plan section, there were still a few things left to complete regarding crafting the route:

Route in readable format

To provide the route in a readable format to the user in the big box, the simplest solution was to utilize what was already available for me to use. In this case, since I already made both coordinate-searching algorithms return the name of the place along with its coordinates, I edited the add waypoint functions in **route.js** so that it takes two parameters, the coordinates *and* the name of the waypoint. Then, in the add_waypoint function in **app.py**, I changed it so that it also takes the name and adds it to a separate new array, **route_names**. In this sense, I can expect any edit made to **route** to be reflected in **route_names**, where route is abstracted from the user. The array is then shown in the big box, with (put join thing here) allowing it to be formatted nicely making it of use to the user.

Adding with proper indexing

This ‘indexing’ issue I keep mentioning refers to the need for the waypoints to be added in between the departure and arrival aerodromes, as these won’t change. This meant that a pointer that points to the waypoint index at which a new waypoint will be appended *to*, must be added.

This is the purpose of the `add_index`, which is set to always be one less than the total length of the array. This way, it always points to the waypoint *before the last*, which means adding a waypoint will append it to the second last waypoint which gives us what we need.

Removing a waypoint

To remove a waypoint, the user simply enters the waypoint in the route and clicks 'Remove waypoint'. This button is attached to the same form, and over in `route.js` whether to remove or add a waypoint is determined by what is in the 'value' attribute of the button.

The algorithm takes the index of the waypoint to be removed, checks that it exists (i.e the index is not -1), and then sends off the index to the backend in `app.py` which takes the index and removes the relevant item in both arrays (coords and names) since they have the same index.

Why I did it this way

The alternative to removing a waypoint was to use a toggle in the header to define whether pressing the button either adds or removes the waypoint. This however required more styling in that the tooltip had to change, the 'Add waypoint' button had to change to 'Remove waypoint', and the city/aviation feature had to be at least greyed or faded out to show that a selection in this case doesn't matter. This would have actually looked nicer, but it takes more time, and in this instance time matters most. Just using a normal button was nice and quick and I could simply attach that to the remove waypoint algorithm without changing anything extra in HTML than what is required.

I also could have made the remove waypoint algorithm as a function in itself, and this is how it was originally, but unfortunately, this caused errors when it came to reloading things in order. The map would refresh itself quickly before the route could even apply the changes, and using asynchronous functions didn't work in this case. So to keep it simple, I've decided to just put it here, the main reason being that it won't be required anywhere else but here.

Program output

 Full Route crafting test with removal.mp4

Known Limitations / Missing features

- Retained Limitations -

These will be touched upon in the Limitations section of the Evaluation.

The (manual) route plan section part is complete, however with two quite big limitations that I have decided to retain within the program due to time constraints.

- Inability to add a waypoint from any index/point in the route (only can add from last waypoint)
- Inability to change the departure/arrival aerodromes unless the entire route is reset (by making a new flight)

Test No.	Test Description	Expected Outcome	Actual Outcome
79	Full route crafting test	Users should be able to fully craft an entire route using a mixture of cities and aviation features as waypoints.	PASS (See program output)
80	Waypoint removal test	Upon pressing the 'remove waypoint' button with a waypoint typed in, the waypoint should be removed from the route.	PASS (See program output)

Input Validation Tests

Test Data	Expected Outcome	Actual Outcome
A duplicate waypoint	An error should occur stating that the waypoint is already in the current route.	PASS (See full test video)
The departure or arrival aerodrome	An error should occur stating that the departure or arrival aerodromes cannot be added as waypoints.	PASS (See full test video)
Removing the departure/arrival aerodrome	An error should occur stating that this is not possible and to start a new flight.	PASS (See full test video)
Removing a waypoint that doesn't exist in the route.	An error should occur stating that the waypoint was not found in the current route.	PASS (See full test video)
Adding the departure/Arrival aerodromes again (pressing enter twice upon inputting the arrival aerodrome)	An error should occur stating that the departure/arrival aerodromes already exist.	PASS (See full test video)

Full test video:  Route crafting validation

Iteration Review

Met requirements: Essential feature complete: Route map

Success criteria item complete: Map with route Visualisation

(Aim complete) Link the route to the session and allow it to be edited at any time by the user

Changes: Remove waypoint button added with functionality

Next steps: Route generation

Iteration 14 - Route generator: Initial Test

app.py

```
#-----AI-----

#-----Making the role for the AI
def fetchRole(type):
    #the role is dependent on the type of prompt being made
    if type == 'Route':
        role = 'user' #TEST

    elif type == 'Brief':
        role = 'ROLE FOR BRIEF'

    elif type == 'Navlog':
        role = 'ROLE FOR NAVLOG'

    return role

#-----Making a prompt
@main.route('/prompt', methods=["POST"])
def prompt():
    type = request.json.get("type") #expecting 'Route', 'Brief' or 'Navlog'
    text = request.json.get("text") #the actual prompt text by user

    #making the OpenAI client and making the request
    try:
        client = OpenAI()
        response = client.responses.create(
            model="gpt-4.1",
            input=[
                {
                    "role" : fetchRole(type),
                    "content" : text
                }
            ]
        )
        response = response.output[0].content[0].text #parsing the response to just the text

        return jsonify({"response": response}), 200 #sends back the AI response

    except Exception as error:
        return jsonify({"error": str(error)}), 400
```

basePage.js

```
//-----AI Prompting-----
export async function prompt(type, text) {
  try {
    const response = await fetch("/prompt", {
      method: "POST",
      headers: { "Content-Type": "application/json" },
      body: JSON.stringify({type: type, text: text}) //sends prompt to backend
    });

    if (!response.ok) {
      throw new Error("Network response was not ok"); //for non-200 errors
    }

    const data = await response.json();
    return data.response; //contains the full response
  } catch (error) {
    console.log(error);
    showAlert("An Error occured whilst generating the response. Check your connection and try again.");
    return null;
  }
}
```

route.js

```
//-----ROUTE GENERATOR-----

const generate = document.getElementById('generateRoute');
const output = document.getElementById('routeOutput');
const routePrompt = document.getElementById('routePrompt');

generate.addEventListener('click', async () => {
  generate.disabled = true; //preventing multiple clicks
  generate.textContent = "Generating...";

  const response = await prompt('Route', routePrompt.value)
  output.textContent = response; //displaying the result in the box

  generate.disabled = false; //allow clicking again
  generate.textContent = "Generate Route";
});
```

What I did

The first thing was to get the API up and running. Out of all the APIs, OpenAI provides the most extensive API documentation that makes making requests using their API very simple, so that wasn't so much of an issue. I implemented the `fetchRole()` algorithm, which required no adjustments, and put placeholders for now, specifically the 'user' role instead of the actual role of the route generator AI.

The purpose of this test was to get AI-generated content in its respective output box, prompted by whatever was typed in the input box when the 'Generate Route' button was pressed. In **app.py** the `prompt()` algorithm takes the sent in text and role type from `prompt(type, text)` in the **frontend**, and then uses those in the API request. Once a response is outputted it's sent back to the prompt function in the **frontend** and the frontend function returns the response as a string, which can then be put into the output box to be shown on screen.

Why I did it this way

I used 'user' as the role instead of typing up the actual role because this is a test of making sure the API works, not tailoring the AI to generate specific things. Thus, it is unnecessary to write up a role (which requires thought) as it would mean I would need to test that as well, and this also requires full integration of the AI with the route, all of which will be done later and not now.

I decided to write up the frontend prompt function in **BasePage.js** because it won't be the only time I'll need to access it. Having it here as an export function allows me to import it on any page and so whenever I need to use the API I can just use the function without having to re-write the whole thing again, which wastes time.

Program output

✦ Route generator

Hey! i'm testing your functionality. Say something!

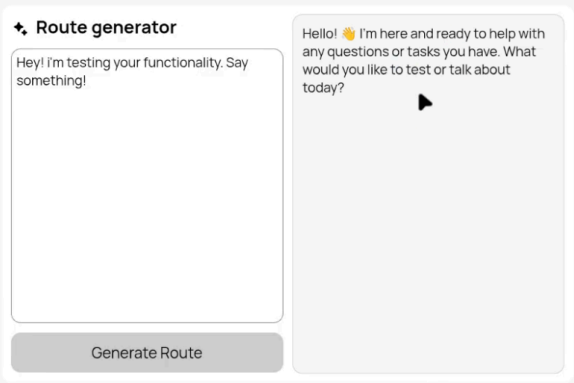
Generate Route

Hello! 🙌 I'm here and ready to help with any questions or tasks you have. What would you like to test or talk about today?

🚩 AI route initial test

Known Limitations / Missing features

- AI not connected to route
- No tailored responses

Test No.	Test Description	Expected Outcome	Actual Outcome
81	AI generation initial test	Upon typing in a prompt in the input box and pressing 'Generate Route', an AI generated response should appear in the output box.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Tailoring prompts and AI output for route integration

Iteration 15 - Route generator: Prompt and output making

app.py

```
"role" : 'system', "content" : 'You are a helpful flight planning assistant.',
"role" : 'user', "content" : (fetchRole(type) + "\n\n" + text) #user prompt with role instructions
```

ai_roles.py

```
#-----Making the role for the AI
def fetchRole(type):
    #the role is dependent on the type of prompt being made
    if type == 'Route':
        role = ""

GLOBAL INSTRUCTIONS:
You are an aviation routing assistant that generates VFR routes within the United Kingdom.
You must follow these rules strictly:
Flight rules:
You produce Visual Flight Rules (VFR) routes only.
Waypoints may only be UK airports (ICAO only), Nav aids (identifier only), VRPs (full official
VRP name), or cities/towns.
Airspace considerations:
You must consider controlled and restricted airspace.
Your route should avoid infringements or clearly justify routing through controlled airspace if
appropriate.
```

You do not consider: weather, fuel, performance, mass & balance, NOTAMs.

Format rules:

All waypoint names in route_names must be UPPERCASE ONLY.

Airports must use ICAO codes (e.g., "EGSU").

Nav aids must use their official identifier (e.g., "LAM").

VRPs must use their full published names (e.g., "LONDON GATEWAY PORT").

Coordinates must be in decimal degrees, latitude then longitude, e.g. 51.7212, 0.1543.

Output format:

You must respond only in the following JSON format:

```
{
  "route_names": ["WAYPOINT1", "WAYPOINT2", "..."],
  "route": [
    [LAT, LON],
    [LAT, LON]
  ],
  "justification": "text explaining the routing, airspace considerations, and any threats."
}
```

Threat awareness (TEM-style):

In the justification, always include any potential threats, such as:

Proximity to controlled airspace

VRPs close together

Terrain awareness (basic, no wx)

Intensively used training areas

Noise-sensitive zones

```
"""

elif type == 'Brief':
    role = 'ROLE FOR BRIEF'

elif type == 'Navlog':
    role = 'ROLE FOR NAVLOG'

return role
```


route.js

```
fetch('/get-flight')
.then(response => response.json())
.then(async FlightData => {
  const departureAirport = FlightData.departureAirport_code;
  const destinationAirport = FlightData.destinationAirport_code;
  const route_names = FlightData.route_names;

  //if there is a prompt provided by the user, use that to generate the route
  if (routePrompt.value) {
    var text = `Generate a VFR route within the UK based on the user's instructions and the provided departure, arrival, and any par
    You must strictly follow all rules from your global instructions.
    Interpret the user's intent precisely while maintaining VFR legality and airspace awareness.
    Departure Airport: ${departureAirport}
    Arrival Airport: ${destinationAirport}
    Existing Route Waypoints: ${route_names.join(', ')}
    User Instructions: ${routePrompt.value}`
  }
  //if there is no prompt, generate a route based on the current *route setup* so, no departure and arrival only
  } else if (!routePrompt.value) {
    var text = `Generate a sensible, safe, airspace-aware VFR route between the provided departure and arrival aerodromes.
    You must strictly follow all rules from your global instructions.
    Choose logical VFR waypoints such as VRPs, nav aids, cities/towns, or airports to shape a practical route that respects c
    Departure Airport: ${departureAirport}
    Arrival Airport: ${destinationAirport}`
  }

  const response = await prompt('Route', text);
  console.log(response); //TEST
  output.textContent = response; //displaying the result in the box
```

What I did

Making the prompts was a little challenging, as you never know what will give the output you require. The main thing was to ensure the generated routes consider the relevant factors, and that it is output in the correct format. I also needed the ability to pass in the relevant route info. This could be done from either the frontend or backend, but I have decided to use the frontend.

There are 3 main prompts being used here, the global prompt, the prompt that uses an existing route and the prompt that uses just the departure and arrival aerodromes. The **global prompt**, which itself was AI generated, provides the structure of the output and all that the AI should consider before providing a response. This is stored in the 'Route' category in the **fetchRoles()** function. Then, the 2 other prompts specifically ask the AI to generate a role based on either the user's input, or just the departure/arrival aerodromes if the user hasn't input anything.

Then in **app.py** these prompts are concatenated together to form one prompt that is then sent to the AI. I had the concept of 'role' a bit incorrect, as I was essentially giving it instructions in a place where I was supposed to explain its function. So this meant I needed to put the global prompt alongside the main prompts, so they can actually be read and acted upon instead of ignored.

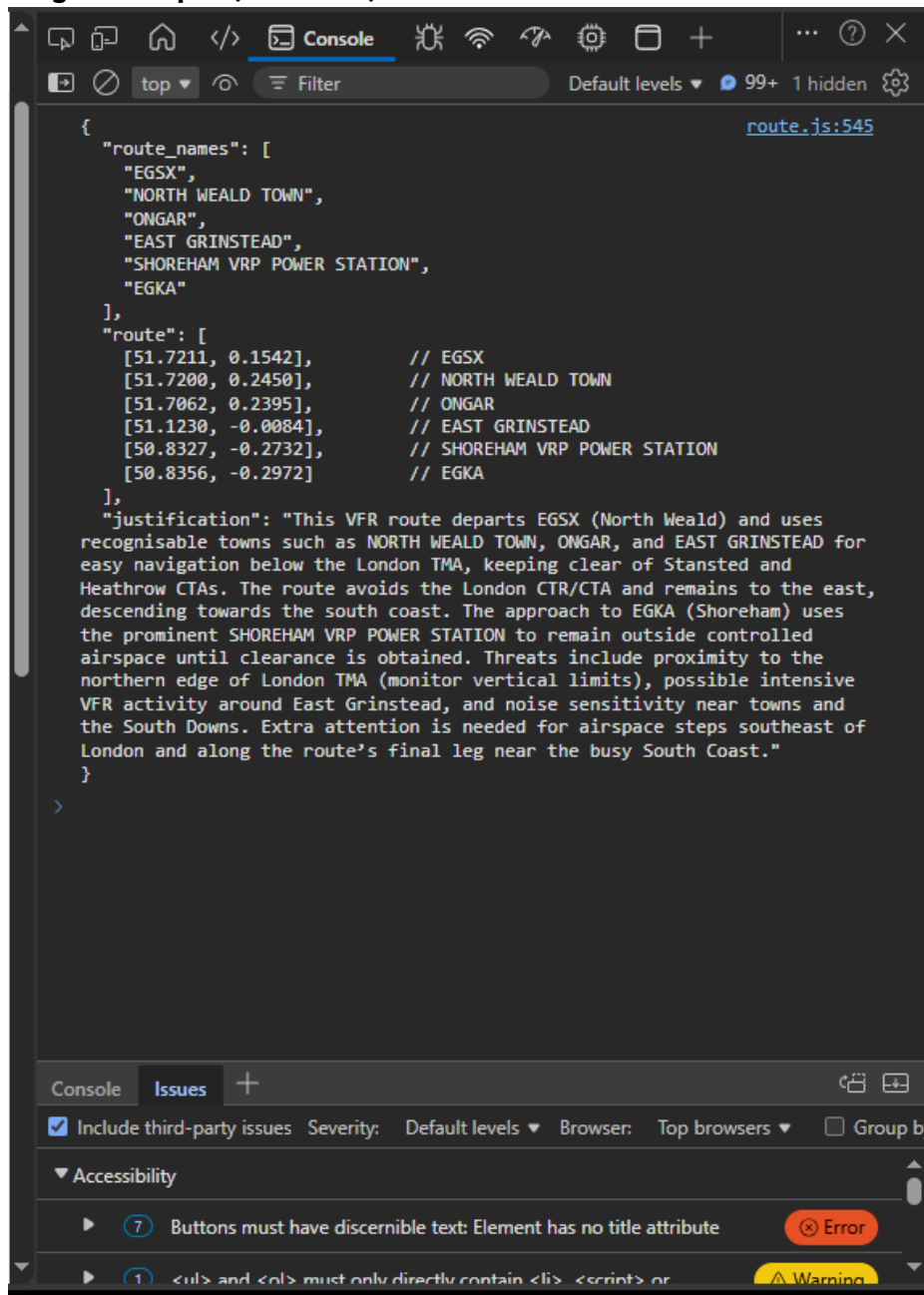
Why I did it this way

fetchRole() has been placed into a new file called **ai_roles.py**. This is simply because the global prompt is very large and I didn't want to place it in **app.py**, as it could potentially make things harder to read. The file will get even larger with other prompts so decided it was feasible to have it separate instead.

Using AI to generate the global prompt was the best option simply because I didn't want to miss out any instructions. When I asked ChatGPT to generate this, I specified that I wanted such and such accuracy with the exact format you see above (json with three values). It returned something with even more detail, proving that I would have had to constantly update a prompt written by me as I was more likely to miss out details that could affect the

accuracy of the output provided. Doing it this way, however, saved a lot more time and effort and greatly improved accuracy from the get-go.

Program output (F12 Menu)



```
{
  "route_names": [
    "EGSX",
    "NORTH WEALD TOWN",
    "ONGAR",
    "EAST GRINSTEAD",
    "SHOREHAM VRP POWER STATION",
    "EGKA"
  ],
  "route": [
    [51.7211, 0.1542], // EGSX
    [51.7200, 0.2450], // NORTH WEALD TOWN
    [51.7062, 0.2395], // ONGAR
    [51.1230, -0.0084], // EAST GRINSTEAD
    [50.8327, -0.2732], // SHOREHAM VRP POWER STATION
    [50.8356, -0.2972] // EGKA
  ],
  "justification": "This VFR route departs EGSX (North Weald) and uses
  recognisable towns such as NORTH WEALD TOWN, ONGAR, and EAST GRINSTEAD for
  easy navigation below the London TMA, keeping clear of Stansted and
  Heathrow CTAs. The route avoids the London CTR/CTA and remains to the east,
  descending towards the south coast. The approach to EGKA (Shoreham) uses
  the prominent SHOREHAM VRP POWER STATION to remain outside controlled
  airspace until clearance is obtained. Threats include proximity to the
  northern edge of London TMA (monitor vertical limits), possible intensive
  VFR activity around East Grinstead, and noise sensitivity near towns and
  the South Downs. Extra attention is needed for airspace steps southeast of
  London and along the route's final leg near the busy South Coast."
}
```

Console Issues +

☒ Include third-party issues Severity: Default levels Browser: Top browsers ☐ Group by

▼ Accessibility

- 7 Buttons must have discernible text: Element has no title attribute Error
- 1 and must only directly contain <script> or Warning

Known Limitations / Missing features

- session not updated with generated route info

Test No.	Test Description	Expected Outcome	Actual Outcome
82	AI route generation test	When generate is pressed, the AI generates a route, selecting suitable waypoints and considering as many factors as possible. It should appear in the aforementioned format.	ERROR

Error Analysis & Fixes

Test No: 82

Failure Description:

2 significant errors here. Whilst an output has been made, it's given a waypoint that doesn't exist: 'SHOREHAM VRP POWER STATION'. I feared this would happen and this represents a significant problem with the route generator. It's essential that any aviation features given are accurate, but it may not be able to find such accurate info.

Secondly, It has deviated from the format a little by adding what looks like comments to the .json, which is not required.

Cause: LLM, so unable to find exact names and locations of certain points.

Fix: To fix the commenting, I'll edit the prompt to not add any comments.

To fix the accuracy error, the only solution that makes sense is to simply give the AI the points I want it to use, with the correct names and coordinates, so that it becomes impossible to make a mistake.

Justification: Trusting the AI to make accurate outputs based on its own ability is dangerous, as I want to avoid providing incorrect data at all times during the use of this tool (despite the disclaimer, stating that this should at no point be used in the real world anyways). Thus, although it would require more work, I believe it is essential if it will mean more accurate data will be provided that the user can actually **relate** with, even though it may not be particularly practical.

Iteration 15.5 - Fixing AI output errors

app.py

```
#-----Getting select waypoints based on route (so i dont need to give all nav aids and vrps to the AI)

def get_waypoints(coords):
    header = {
        "x-openaip-api-key": api_key_openaip
    }

    minx = min(coords[0][1], coords[1][1]) - 0.5367 #approx 20 NM corridor
    maxx = max(coords[0][1], coords[1][1]) + 0.5367
    miny = min(coords[0][0], coords[1][0])
    maxy = max(coords[0][0], coords[1][0])

    # "bbox" defines a rectangular area of interest for the query
    params_vrp = {
        "country": "GB",
        "fields": "name,geometry.coordinates",
        "bbox": f"{minx},{miny},{maxx},{maxy}"
    }

    params_navaid = {
        "country": "GB",
        "fields": "identifier,geometry.coordinates",
        "bbox": f"{minx},{miny},{maxx},{maxy}"
    }

    vrp = "https://api.core.openaip.net/api/reporting-points"
    navaid = "https://api.core.openaip.net/api/navaids"

    response_vrp = requests.get(vrp, headers=header, params=params_vrp)
    response_navaid = requests.get(navaid, headers=header, params=params_navaid)

    data_vrp = response_vrp.json()
    data_vrp["items"]

    data_navaid = response_navaid.json()
    data_navaid["items"]

    #print only the name and coordinates of each reporting point
    waypoints = []
    for item in data_vrp["items"]:
        waypoints.append(f"{item['name']}, {item['geometry']['coordinates']}")
    for item in data_navaid["items"]:
        waypoints.append(f"{item['identifier']}, {item['geometry']['coordinates']}")

    return waypoints
```

```
{
    "role": "system", "content": "You are a helpful flight planning assistant.",
    "role": "user", "content": (
        fetchRole(type) + "\n\n"
        + text + "\n\n"
        + str(get_waypoints([departure_coords, destination_coords]))
    )
}
```

route.js

```
JS route.js X JS settings.js # route.css # main-menu.css app.py 1 # settings.css algorithms.dscode base.htm
AIVFR > static > JS route.js > onsubmit
515 >rate.addEventListener('click', async () => {
521 .then(async FlightData => {
522   const departureAirport = FlightData.departureAirport_code;
523   const destinationAirport = FlightData.destinationAirport_code;
524   const route_names = FlightData.route_names;
525
526   //if there is a prompt provided by the user, use that to generate the route
527   if (routePrompt.value && departureAirport && destinationAirport) {
528     var text = 'Generate a VFR route within the UK based on the user's instructions and the provided departure, arrival, and arrival airport. (If you choose to edit around an existing route, provide a full route including those existing waypoints) You must strictly follow all rules from your global instructions. Interpret the user's intent precisely while maintaining VFR legality and airspace awareness. Departure Airport: ${departureAirport} Arrival Airport: ${destinationAirport} Existing Route Waypoints: ${route_names.join(', ')} User Instructions: ${routePrompt.value}';
529     //If choosing a VRP or a Navaid, names AND coordinates MUST be chosen from the list provided below. If choosing a city or town, provide the name and ensure the coordinates are accurate.'
530
531     //if there is no prompt, generate a route based on the current *route setup* so, no departure and arrival only
532   } else if (!routePrompt.value && departureAirport && destinationAirport) {
533     var text = 'Generate a sensible, safe, airspace-aware VFR route between the provided departure and arrival aerodromes. You must strictly follow all rules from your global instructions. Choose logical VFR waypoints such as VRPs, navaids, cities/towns, or airports to shape a practical route that respects the rules of the air. Departure Airport: ${departureAirport} Arrival Airport: ${destinationAirport}';
534     //If choosing a VRP or a Navaid, names AND coordinates must be chosen from the list provided below. If choosing a city or town, provide the name and ensure the coordinates are accurate.'
535
536   } else {
537     showAlert("Please make sure both departure and arrival aerodromes are set before generating a route.");
538     generate.disabled = false; //allow clicking again
539     generate.textContent = "Generate";
540     return;
541   }
542
543   const response = await prompt('Route', text);
544   console.log(response); //TEST
545   output.textContent = response; //displaying the result in the box
546   generate.disabled = false; //allow clicking again
547   generate.textContent = "Generate";
548 }
549
550
```

prompting AI to look at listed waypoints

making sure dep and arrival are set first

ai_roles.py

YOU MUST STRICTLY OBEY THE ABOVE FORMAT, NO COMMENTING, BRACKETS, OR ANYTHING OUTSIDE OF THIS FORMAT OF ANY SORT SHOULD BE INCLUDED AT ANY POINT UNDER ANY CIRCUMSTANCE.

To code this solution, the first thing I needed to do was to find out how I can get select waypoints to give to the AI. My initial thought was to use the aviation features available in the OpenAIP database, as I knew how to retrieve them, parse them and output them so that I could ideally just add that output to the prompt string. The problem with this is that I wouldn't be able to just put all 400+ aviation features in the prompt because that would easily hit its prompt limit.

Luckily, OpenAIP has a parameter called '**bbox**', where you can request select waypoints within a specified box on the map. This enabled me to essentially create a corridor of waypoints surrounding the selected departure and arrival airports, whilst also giving enough freedom for the AI to select virtually any waypoint and still make a route solely by its discretion.

bbox string (query)	A comma separated list of lat/lon values that define a bound box of an area of interest (AOI). The bbox must be defined as 'minx,miny,maxx,maxy'. Example : 7.803258324701744,48.94584024337291,10.583009555132634,49.77589256829904 7.803258324701744,48.94584024337291,10.
---------------------------	--

With this, I created a new function **get_waypoints()** which requests the aviation features (so reporting points and nav aids - airports not added because they're less likely to be messed up) within a defined box which uses the departure and arrival aerodrome's coordinates. The response of this is then concatenated with the original prompt. I also thought it'd be nice to visualise it on a real map to show how I intended it to work and how the box is defined, and this can be seen below.

 bbox explanation.mp4

Test No.	Test Description	Expected Outcome	Actual Outcome
83	AI route generation test	When generate is pressed, the AI generates a route, selecting suitable waypoints and considering as many factors as possible. It should appear in the aforementioned format.	<pre> route.js:558 { "route_names": ["EGSX", "BILLERICAY LAKE", "EPPING", "GODSTONE RAILWAY STATION", "DORKING", "BILLINGSHURST", "WASHINGTON INTERSECTION", "BRIGHTON MARINA", "EGKA"], "route": [[51.7472, 0.2394], [51.63333333333333, 0.4166666666666667], [51.7, 0.11116666666666666], [51.218, -0.05116666666666666], [51.22694444444444, -0.335], [51.015, -0.45], [50.9095, -0.4078333333333333], [50.8101, -0.1034], [50.8366, -0.2974]], "justification": "This route avoids all controlled airspace, specifically keeping well clear of London City and Gatwick CTR/CTA. After departing EGX, the route heads southeast via BILLERICAY LAKE and EPPING, remaining north of the London TMA. By routing via GODSTONE RAILWAY STATION and DORKING, the track avoids the Gatwick airspace to the west, then descends through rural corridors south of DORKING and via BILLINGSHURST, bypassing busy Solent/Heathrow regions. From WASHINGTON INTERSECTION, the route remains clear of Shoreham's ATZ and enters circuit via the BRIGHTON MARINA VRP, in accordance with your approach request. Threats include potentially busy VRPs near greater London, terrain along the North and South Downs, and passing near noise-sensitive zones (villages south of Dorking and Billingshurst). Maintain good lookout for glider and parachute activity in rural stretches." } </pre>

Iteration Review

Met requirements: (Aim complete) Code the route generator through implementation of fetchRole() and prompt() Algorithms

Changes: Major prompt changes for more accurate output

Next steps: Updating the route using the response

Iteration 16 - Route generator: Updating the route using the response

route.js

```
var response = await prompt('Route', text);
try {
  response = JSON.parse(response); //parsing the response to JSON
  output.textContent = response.justification; //displaying the result in the box
} catch (error) {
  console.log(error);
  showAlert("An error occured whilst generating the route. Please try again.");
  generate.disabled = false; //allow clicking again
  generate.textContent = "Generate";
  return;
}

//adding the waypoints to the route
await update('route_names', response.route_names);
await update('route', response.route);
await update('route_gen_justification', response.justification);
reload_map();
update_route_names();

generate.disabled = false; //allow clicking again
generate.textContent = "Generate";
});
}
```

```
const line = L.polyline(route, { color: '■#f0f' , measurementOptions : { imperial:true }})
.addTo(map)
.showMeasurements(); //show the distances between each point on the map
```

app.py

```
route = [],
"route_names" : [],
"route_gen_justification" : "",
"distance" : {"value" : 0, "class" : "distance"},|
```

route.html

```
<!--AI OUTPUT-->
<div id="routeOutput">{{ flight.route_gen_justification if
flight.route_gen_justification else ' ' }}</div>
</div>
```

What I did

This last part was much simpler than I thought it would be, and I'm honestly relieved that it's all worked so well! Given that the route is outputted in json format already, all I needed to do is parse it and then start updating each value in the session to the newly generated routing. A new value called `route_gen_justification`, now contains the justification provided by the AI, so it can be stored in the session.

Additionally, I utilized the leaflet plug-in '[Leaflet Measure Path](#)' to show the distances between the points on the map. I wasn't able to find a way to change these units to the

user's units (as I don't know how to actually retrieve these distances for my use rather than just having them displayed), and for the sake of time, I've decided to move on.

Why I did it this way

I initially had the idea of just iterating through each item and updating the route, but this very (very) quickly became impractical as I realised that I have to run the add waypoint function several times to achieve this. Since the AI also includes the departure and arrival aerodromes, I realised the entire route could just be replaced without an issue.

The distance was a big issue, and I've made the decision to move on after finding no easy way of finding the distance between the points on the map. Initially I was seeking a coordinate API, but they all seemed like you had to pay quite a bit and I don't really want to waste time looking for one. This plugin will have to do for now, although it requires the user to convert the distances themselves.

Program output

 Full Route generation functionality test.mp4

Known Limitations / Missing features

- Retained Limitations -

These will be touched upon in the Limitations section of the Evaluation.

- Distances not displayed as planned
- AI-related limitations leading to unreliable and unpredictable route choices

Test No.	Test Description	Expected Outcome	Actual Outcome
84	Full route generation test	The AI should generate a suitable route that should generally follow the instructions it has been given. The route should be shown on the map, along with the names of the waypoints. All should happen without user input when the user presses the "Generate" button.	PASS (with major deficiencies- see limitations section)

Iteration Review

Met requirements: Essential Feature Complete | **Success criteria item complete:** Route Generation

Changes: Distances display

Next steps: Route setup Tooltip box

Iteration 17 - Tooltip Box

route.html

```
<!-------Tooltip Popup----->
<div id="tooltip" class="modal">
  <div class="tooltipContent">
    <!--header--><h1 id="tooltipHeader">ROUTE SETUP INSTRUCTIONS</h1>
    <ol>
      <li>To select an aerodrome, type in the ICAO code (not case sensitive) and press ENTER. Wait for the
      <li>After selecting the arrival aerodrome, the route will initialise. After this point, you must start
      <li>You can manually add and remove items from the route by using the box provided. Simply select the
      <li>VRPs, nav aids and airports are based upon the OpenAIP database, so use the map to access the corr
      <li>Please note that not all cities area available. If a city/town you have chosen is not in the API
    </ol>
    <!--close tooltip button--><span><button id="closeTooltip">CLOSE</button></span>
  </div>
</div>

<div id="content">
```

route.css

```
/*tooltip popup*/

.tooltipContent {
  position: relative;
  background-color: var(--highlight);
  margin: 10% auto;
  padding: 30px 30px 5px 35px;
  width: 720px;
  border-radius: 15px;
  text-align: center;
  bottom: 30px;
}

/*header*/
#tooltipHeader {
  font-weight: 500;
  font-size: 30px;
  margin-top: auto;
  margin-bottom: 10px;
}

/*close button*/
#closeTooltip {
  position: relative;
  font-family: 'Manrope';
  font-size: 20px;
  background-color: var(--bg);
  border: none;
  padding: 6px 20px;
  border-radius: 5px;
  margin-left: 5px;
  margin-right: 5px;
  margin-top: 20px;
}

#closeTooltip:hover {
  background-color: var(--primary);
}
```

```

}
.modal{
  display: none; /*makes it hidden by default */
  position: fixed;
  z-index: 3; /*this makes it appear ontop*/
  left: 0;
  top: 0;
  width: 100%;
  height: 100%;
  background-color: var(--popupbg);
}

```

route.js

```

//-----TOOLTIP-----
const tooltip = document.getElementById("tooltip");
const closeTooltip = document.getElementById("closeTooltip");
const showTooltip = document.getElementById("routeSetupTooltip")

//show when info icon clicked
showTooltip.addEventListener("click", function() {
  tooltip.style.display = "block"
})

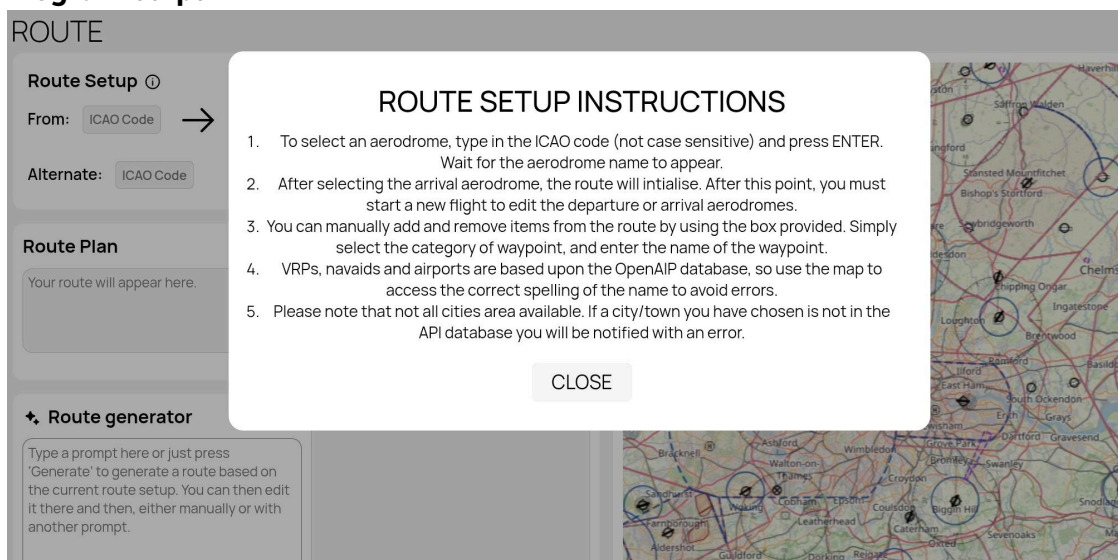
//close when 'close' button clicked
closeTooltip.addEventListener("click", function() {
  tooltip.style.display = "none"
})

```

What I did

The final part of this section was the tooltip that explains the functionality of the route setup, for ease of use. This was relatively straightforward as the implementation remains similar to the disclaimer and load flight popups, so there was nothing new.

Program output



Tooltip popup test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
85	Tooltip test	When the info icon is pressed the tooltip should appear. Upon pressing the 'close' button it should disappear.	PASS (See program output)

Iteration Review

Met requirements: (Aim Complete) Add tooltips for route setup

Changes: N/A

Next steps: Section 7: Weather Tab

Section Review

Aims completed in this section:

- **Complete 2 success criteria items:** Route generation, Map with route Visualisation
- **Complete 3 Essential features:** Route generation, Ability to enter departure/Arrival aerodromes, Route map
- Give the user the ability to add departure and arrival aerodromes
- Validate this data by implementing 2 planned algorithms - verifyICAO() and fetchAirportName()
- Code the ability to add alternate aerodromes
- Code the route generator through implementation of fetchRole() and prompt() Algorithms
- Add tooltips for route setup
- Link the route to the session and allow it to be edited at any time by the user

Aims not completed in this section:

- Code the ability to add 'via' aerodromes (explained after iteration 5)

Iterative tests complete:

- AI should output suitable responses in accordance to their given role and user prompt
- APIs are operative and display data in the correct way on the relevant page

Remarks: Forgot to test darkmode here. See end of Section 7 for test.

Section 7: Weather Tab

Aims of this section

- **Complete an Essential feature: Integration of aerodrome forecasts and reports**
- **Complete a Success criteria item:** Use of APIs to fetch weather data
- Give the user the ability to enter any aerodrome and access their weather data
- Give the user the ability to view the Meteorological Aerodrome Reports (METAR) and Terminal Aerodrome Forecasts (TAF) of the departure and arrival aerodromes, as well as any aerodrome of their choice
- Display whether or not the conditions are suitable for Visual flight rules via a known grading system
- Provide weather data in both coded and decoded formats

Note: In this section, the term 'WX' is an abbreviation for weather and will be used throughout this section.

Iteration 1 - HTML initial layout and styling

weather.html

```
<!--=====HEADER=====-->
<div class="pageTitle" id="header">
  <div id="title">WEATHER</div>
  <span id="alertBox"></span>
  <!--Update weather reports button-->
  <button id="update" style="display: flex; align-items: center;">
    <svg id='update_icon' xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/x
    <p>UPDATE</p>
  </button>
</div>

<div id="content">
<!-------AERODROME SEARCH-->
  <div id="WX_search" class="WX_item">
    <!-------HEADER----->
    <div id="WX_search_header" style="display: flex; align-items: center;">
      <svg height="30" viewBox="0 0 512 512" fill="currentColor" width="30" xmlns="http://www.w3.org
    </div>
  </div>

<!-------DEPARTURE AERODROME WEATHER-->
  <div id="WX_departure" class="WX_item">
    <!-------HEADER----->
    <div id="WX_departure_header" style="display: flex; align-items: center;">
      <svg style="enable-background:new 0 0 24 24; width: 30px; margin-right: 10px" version="1.1" vi
    </div>
  </div>

<!-------ARRIVAL AERODROME WEATHER-->
  <div id="WX_arrival" class="WX_item">
    <!-------HEADER----->
    <div id="WX_departure_header" style="display: flex; align-items: center;">
      <svg style="enable-background:new 0 0 24 24; width: 30px; margin-right: 10px" version="1.1" vi
    </div>
  </div>
</div>
```

weather.css

```
#header {
  display: flex;
  justify-content: space-between;
  border: 1px none var(--text);
  height: 46px;
}

#update { /*update weather report btn*/
  width: 165px;
  height: 40px;
  font-family: 'Manrope';
  font-size: 18px;
  font-weight: 600;
  padding: 8px 26px;
  border: 1px solid var(--text);
  border-radius: 5px;
  background-color: var(--highlight);
}

#update:hover {
  cursor: pointer;
  background-color: var(--primary);
}

#update_icon {
  width: 30px;
  margin-right: 10px;
}

/*Main content*/
#content {
  grid-template-columns: 5fr 5fr;
  display: grid;
  gap: 10px;
  border-radius: 10px;
  z-index: 1;
  height: 100vh;
}

.wx_item { /*all items on the weather page*/
  margin: 2px;
  background-color: var(--highlight);
  padding: 10px;
  border-radius: 10px;
}

/*departure and arrival in second column*/
#WX_departure, #WX_arrival {
  grid-column: 2 / span 1;
}

/*aerodrome search remains in the first column but spans 2 rows*/
#WX_search {
  grid-column: 1 / span 1;
  grid-row: 1 / span 2;
}
```

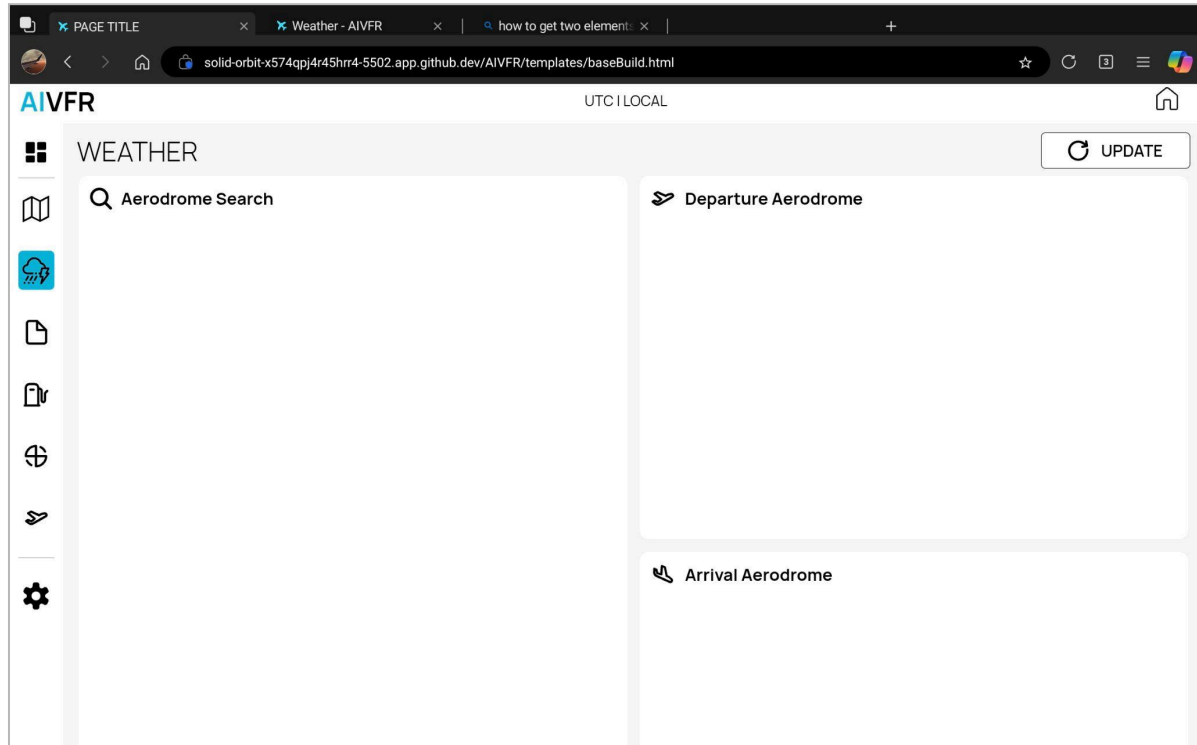
What I did

As always, the first thing to do is to get the general layout of the page done. This mainly consisted of making the correct grid layout as well as the 'update' button which should have the action of updating all weather reports.

Why I did it this way

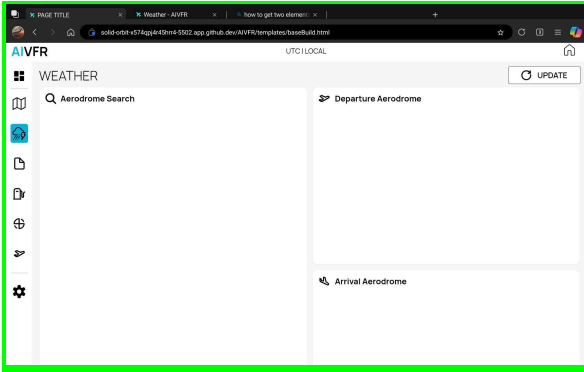
It's through doing it this way that I can organise the way I'd like text to be output in this section. It's very important here, as these weather reports can really differ in length and so getting the initial sizing right is vital. **Without this consideration, I will be risking facing major design flaws later in development.**

Program output



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
86	Initial layout test	Initial layout and design of the page should match the normal design.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Searching for an aerodrome

Iteration 2 - Aerodrome Search: Searching for an aerodrome weather.html

```

<!--Searching-->
<div id="searchAirport">
  <input id="airport_code" type="text" name="airport_code" placeholder="ICAO Code" value="">
  <span id="airport_name"></span>
</div>

<!-------WEATHER REPORTS----->
<div id="reports">
  <!--METAR-->
  <div id="METAR">
    <hr size="3.5" color="#cdcdcd" noshade style="margin-top: 25px;">
    <h3>Meteorological Aerodrome Report (METAR)</h3>
  </div>

  <!--TAF-->
  <div id="TAF">
    <hr size="3.5" color="#cdcdcd" noshade style="margin-top: 25px;">
    <h3>Terminal Aerodrome Forecast (TAF)</h3>
  </div>
</div>

```


weather.css

```
/*-----SEARCHING-----*/
#searchAirport {
  display: flex;
  justify-content: space-between;
}

#airport_code {
  margin-top: 10px;
  border: 1px solid var(--text);
  border-radius: 5px;
  padding: 5px;
  margin-left: 10px;
  width: 100px;
  text-align: center;
  font-weight: 600;
  font-family: 'Manrope';
  font-size: 13px;
  max-width: 80px;
  background-color: var(--bg);
  border: 1px solid var(--onhover);
}

#airport_name {
  font-weight: 400;
  margin-left: 10px;
  color: var(--text);
  background-color: var(--bg);
  padding: 5px 10px;
  font-size: 20px;
  font-weight: 600;
  display: none; /*hide the airport names until they are available*/
}

/*when the airport names are available, show them*/
#airport_name.active {
  display: block;
}

/*content box- this is where reports will be*/
#reports {
  display: grid;
  height: 100%;
  grid-template-rows: 1fr 1.5fr;
}
```


weather.js

(VerifyICAO() and fetchAirportDetails() also included but not shown as there are no changes)

```
//-----Airport search-----
const Airport_code = document.getElementById("airport_code");
const Airport_name = document.getElementById("airport_name");

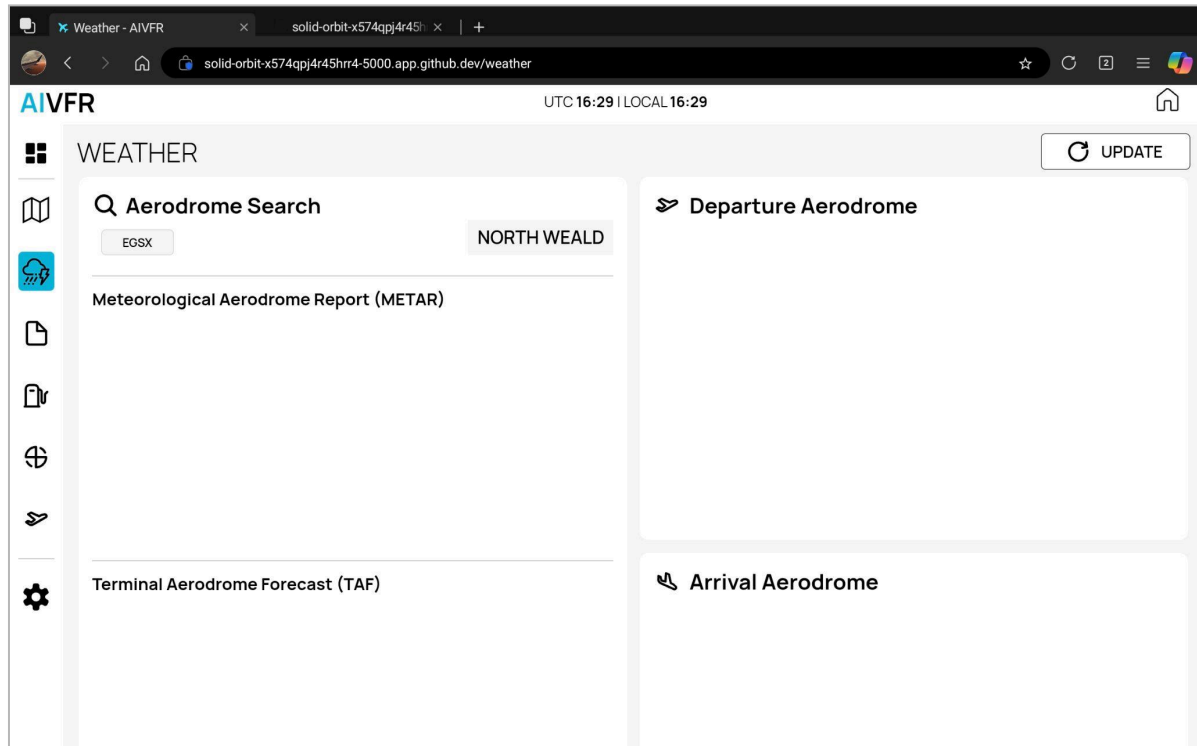
Airport_code.addEventListener("keydown", async (event) => {
  if (event.key === "Enter") {
    Airport_name.style.display = "inline";
    Airport_name.textContent = "...";

    //validating data
    if (verifyICAO(Airport_code.value)) {
      const airportDetails = await fetchAirportDetails(Airport_code.value);
      if (airportDetails.country == "GB") {
        await update("WX_airportCode", Airport_code.value.toUpperCase());
        await update("WX_airportName", airportDetails.name);
        Airport_code.style.textTransform = "uppercase";
        Airport_name.textContent = airportDetails.name;
      } else if (airportDetails.country != "GB") {
        showAlert("Only UK aerodromes are supported");
        Airport_name.style.display = "none";
      }
    } else {
      showAlert("Invalid ICAO code");
      Airport_name.style.display = "none";
    }
  }
});
```

What I did

Next, I implemented the ability to search for an aerodrome. This used the exact same logic as what was used for the route setup searches and so there was once again nothing new here, other than the locations of the search bar and the name of the aerodrome as you will see below (matching the design). I also coded the remaining sections of the UI, utilizing grid to separate them properly. Now that the foundation of it all is done, I can now move on to implementing the API and testing outputs.

Program output (With input validation)



 aerodrome search and validation test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
87	Search Aerodrome Test (Including input validation)	Upon searching an aerodrome and pressing enter, the typed in code should be validated and the aerodrome should appear on the right. If it is not a valid code, then the respective error should be shown.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Fetching and outputting reports

Iteration 3 - Aerodrome Search: Fetching and outputting reports

weather.js

```
//-----Update weather report-----
const update_wx = document.getElementById('update');
const code_searched = document.getElementById('airport_code');

async function getWeather(code) {
  if (code === "") {
    showAlert("Couldn't get data for searched aerodrome. Please enter an ICAO code.");
    return;
  }

  try {
    const response = await fetch("/get-weather", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ code: code.toUpperCase() }), //sends off the code to the backend
    });

    if (!response.ok) {
      throw new Error("Network response was not ok"); //for non-200 errors
    }

    const data = await response.json();
    return data; //returns the weather data
  } catch (error) {
    showAlert("Error fetching weather data");
    return null;
  }
};
```

```

//-----updating all weather data-----
const metar_searched = document.getElementById('METAR_searched');
const taf_searched = document.getElementById('TAF_searched');
const metar_searched_decoded = document.getElementById('METAR_searched_decoded');
const taf_searched_decoded = document.getElementById('TAF_searched_decoded');

update_wx.addEventListener('click', async () => {
  //Add loading feedback here
  const wx_searched = await getWeather(code_searched.value);

  if (wx_searched) {
    //update the session as well as the actual text for each report
    await update("METAR_searched", wx_searched.metar.raw_text);
    metar_searched.textContent = wx_searched.metar.raw_text;

    await update("TAF_searched", wx_searched.taf.raw_text);
    taf_searched.textContent = wx_searched.taf.raw_text;

    await update("METAR_searched_decoded", wx_searched.metar);
    metar_searched_decoded.textContent = wx_searched.metar;

    await update("TAF_searched_decoded", wx_searched.taf);
    taf_searched_decoded.textContent = wx_searched.taf;
  }
});

```

weather.css

```
/*-----Weather reports-----*/

h3 {
  margin-bottom: 20px;
}

.report {
  background-color: var(--bg);
  margin-top: 10px;
  margin-bottom: 10px;
  padding: 10px;
  border-radius: 10px;
}

.grading {
  margin-left: 15px;
  font-size: 20px;
  font-weight: 650;
}

#TAF_searched {
  resize: none;
  width: 100%;
  height: calc(100px + 2em);
  font-family: 'Manrope';
  font-size: 14px;
  border: 1px solid var(--onhover);
  pointer-events: none;
}

#METAR_searched_decoded, #TAF_searched_decoded {
  resize: none;
  margin-top: 20px;
  width: 100%;
  height: fit-content;
  font-family: 'Manrope';
}
```

weather.html (blue lines are new)

```
4 <input id="airport_code" type="text" name="airport_code" placeholder="ICAO code" value="" />
5 <span id="airport_name" class="{{ 'active' if flight.WX_airportName else '' }}">{{ flight.WX_airportName }}</span>
6 </div>
7 <!--Weather reports-->
8 <div id="reports">
9 <!--METAR-->
10 <div id="METAR">
11 <hr size="3.5" color="#cdcdcd" noshade="" style="margin-top: 25px;" />
12 <h3>Meteorological Aerodrome Report (METAR)</h3>
13 <span id="METAR_searched" class="report">{{ flight.METAR_searched }}</span><span id="METAR_searched_decoded" class="report">{{ flight.METAR_searched_decoded }}</span>
14 </div>
15 <!--TAF-->
16 <div id="TAF">
17 <hr size="3.5" color="#cdcdcd" noshade="" style="margin-top: 25px;" />
18 <h3>Terminal Aerodrome Forecast (TAF)</h3>
19 <textarea id="TAF_searched" class="report">{{ flight.TAF_searched }}</textarea><span id="TAF_searched_decoded" class="report">{{ flight.TAF_searched_decoded }}</span>
20 </div>
21 </div>
```

app.py

```
174 #-----WEATHER
175 "WX_airportCode" : "",
176 "WX_airportName" : "",
177 "METAR_searched" : "",
178 "TAF_searched" : "",
179 "METAR_searched_decoded" : "",
180 "TAF_searched_decoded" : "",
181
```

```
#-----WEATHER REPORTS-----
@main.route('/get-weather', methods=["POST"])
def get_weather():
    airport_code = request.json.get("code") #expecting ICAO code

    url_metar = f"https://api.checkwx.com/metar/{airport_code}/decoded"
    url_taf = f"https://api.checkwx.com/taf/{airport_code}/decoded"

    #making the requests to the API
    response_metar = requests.request("GET", url_metar, headers={'X-API-Key': api_key_wx})
    response_taf = requests.request("GET", url_taf, headers={'X-API-Key': api_key_wx})

    if response_metar.status_code == 200 and response_taf.status_code == 200:
        #parsing the responses to JSON
        data_metar = response_metar.json()
        metar = data_metar['data'][0]
        data_taf = response_taf.json()
        taf = data_taf['data'][0]
        return jsonify({"metar": metar, "taf": taf}), 200 #back to frontend
    else:
        return jsonify({"error": "Unable to fetch weather data"}), 400
```

What I did

To fetch the weather reports, I'll be using the CheckWX API, as mentioned in the design. This API in particular has the major advantage of providing decoded weather data along with the raw text, which makes my job much easier as I don't need to create an entire decoding algorithm (which can be very complex).

The logic is the same here as used in the last section: In **weather.js** the `getWeather()` function sends the inputted code to the backend through POST; when received in **app.py** the `get_weather()` function takes the code and puts it into the request for the API. Its output is then parsed into JSON format and sent back to the frontend where it is returned. If any errors occur within this process an alert is output stating there was an issue.

To ensure that the reports stay there on reload, they are all added to the session, and the default values are set to be empty (") if the actual key in the session is empty, or to show the actual weather data if data is present. Also, note that the **grading** (explained [here](#)) is a placeholder for now, as the data output is not yet parsed into a readable format.

Why I did it this way

I have not taken the time to parse the decoded data into a readable format in this iteration. This was simply because I was not testing it, I'm just testing the program's ability to fetch and output a response from the API, which has been successful. **There is a lot to consider when parsing and so deciding to do so here may just make this a lot longer than it should be, for both me and you, the reader.**

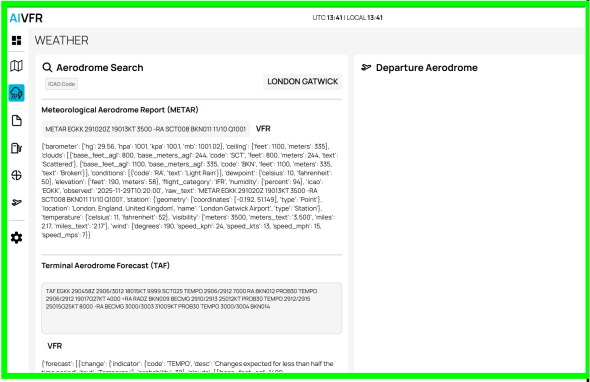
Program output

The screenshot displays the AI VFR application interface. At the top, it shows 'UTC 13:41 | LOCAL 13:41'. The main section is titled 'WEATHER' and contains a 'Q Aerodrome Search' form with 'LONDON GATWICK' entered. Below the search bar, the 'Meteorological Aerodrome Report (METAR)' is displayed, showing the text 'METAR EGKK 291020Z 19013KT 3500 -RA SCT008 BKN011 11/10 Q1001 VFR'. The report includes detailed JSON data for barometer, clouds, conditions, dewpoint, elevation, flight category, humidity, icao, observed, raw text, station, geometry, location, temperature, visibility, wind, speed_kph, speed_kts, speed_mph, and speed_mps. Below the METAR report, the 'Terminal Aerodrome Forecast (TAF)' is shown, displaying the text 'TAF EGKK 290458Z 2906/3012 18015KT 9999 SCT025 TEMPO 2906/2912 7000 RA BKN012 PROB30 TEMPO 2906/2912 19017G27KT 4000 +RA RADZ BKN009 BECMG 2910/2913 25012KT PROB30 TEMPO 2912/2915 25015G25KT 8000 -RA BECMG 3000/3003 31009KT PROB30 TEMPO 3000/3004 BKN014'. The forecast also includes JSON data for the forecast, change, indicator, code, desc, time period, text, temperature, probability, clouds, base, feet, and 100. The interface also includes a 'Departure Aerodrome' section on the right.

Weather report fetch & output test.mp4

Known Limitations / Missing features

- Major parsing issues (just the output was being tested here)

Test No.	Test Description	Expected Outcome	Actual Outcome
87	CheckWX api fetch and output test	Upon pressing the update button, the API should fetch the required weather data and it should be outputted on the screen fully raw.	

Iteration Review

Met requirements: **Success criteria item complete:** Use of APIs to fetch weather data (Aim complete) Give the user the ability to enter any aerodrome and access their weather data

Changes: N/A

Next steps: Parsing the data

Iteration 4 - Aerodrome Search: Parsing the METAR data

weather.js

```

130 |     await update("METAR_searched_decoded", await parseMETAR(wx_searched.metar));
131 |     metar_searched_decoded.textContent = await parseMETAR(wx_searched.metar);
132 |
133 |     await update("TAF_searched_decoded", await parseTAF(wx_searched.taf));
134 |     taf_searched_decoded.textContent = await parseTAF(wx_searched.taf);

```

```

Sync function parseMETAR(metar) {
    //fetching current units
    const response = await fetch("/get-settings");
    const settings = await response.json();

    const wind_unit = settings.units_airspeed;
    const altitude_unit = settings.units_altitude;

    //airspeed
    if (wind_unit == "knot") {
        var windspeed = `${metar.wind.speed_kts} knots`;
    } else if (wind_unit == "kilometer_per_hour") {
        var windspeed = `${metar.wind.speed_kph} km/h`;
    } else if (wind_unit == "mile_per_hour") {
        var windspeed = `${metar.wind.speed_mph} mph`;
    }
}

```



```

//altitude
if (altitude_unit == "feet") {
    try {
        var ceiling = `${metar.ceiling[0].feet} feet`;
    }
    catch (TypeError) {
        var ceiling = "N/A";
    }
    var cloud_base = `${metar.clouds[0].base_feet_agl} feet`;

} else if (altitude_unit == "meter") {
    try {
        var ceiling = `${metar.ceiling[0].meters} meters`;
    }
    catch (TypeError) {
        var ceiling = "N/A";
    }
    var cloud_base = `${metar.clouds[0].base_metres_agl} meters`;
}

//other non-unit related values
const time = metar.observed;
const visibility = metar.visibility.meters;
const visibility_text = metar.visibility.meters_text;
const wind_direction = metar.wind.degrees;
const temperature = metar.temperature.celsius;
const dewpoint = metar.dewpoint.celsius;
const pressure = metar.barometer.hpa;
const clouds_code = metar.clouds[0].code;
const clouds_text = metar.clouds[0].text;
const flight_category = metar.flight_category;
const humidity = metar.humidity.percent;

//constructing decoded METAR string
const decoded_METAR = `
Time of observation: ${time}\n
Wind: ${wind_direction}° at ${windspeed}\n
Visibility: ${visibility} meters (${visibility_text})\n
Temperature: ${temperature}°C\n
Dewpoint: ${dewpoint}°C\n
Pressure: ${pressure} hPa\n
Humidity: ${humidity}%\n
Ceiling: ${ceiling}\n
Clouds: ${clouds_code} (${clouds_text})\n
Cloud Base: ${cloud_base}
`;

//grading
METAR_searched_grading.textContent = flight_category;
if (flight_category == "VFR") {
    METAR_searched_grading.style.color = "green";
}

```

```

    } else if (flight_category == "MVFR") {
        METAR_searched_grading.style.color = "blue";
    } else if (flight_category == "IFR") {
        METAR_searched_grading.style.color = "red";
    } else if (flight_category == "LIFR") {
        METAR_searched_grading.style.color = "purple";
    }

    //returning decoded METAR
    return decoded_METAR;
}

```

What I did

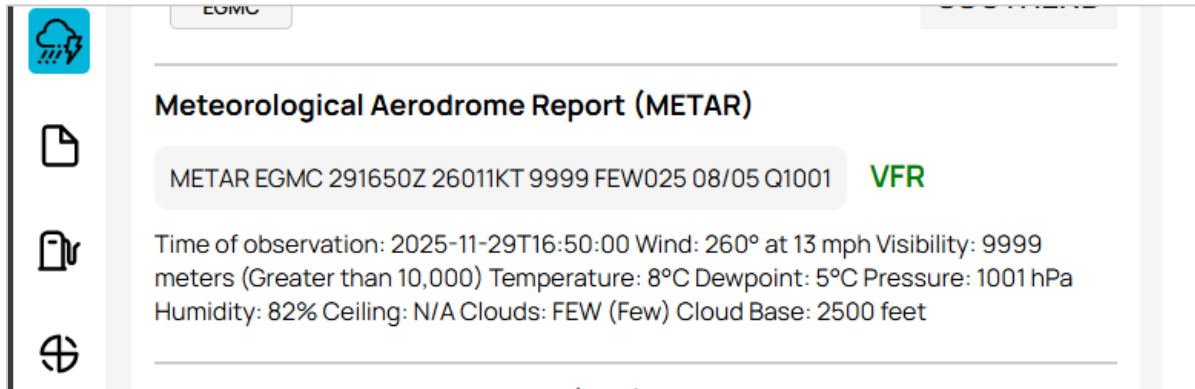
This process was rather grueling because of just how many options there were to use. The main concept of the parsing was to pick out the relevant data from that huge json and put it in a readable format. If you look at the several lines of constant variable declarations in , this is essentially all that `parseMETAR()` was supposed to do, and then they were just put into a multi-line string as shown. The same will apply for TAFs, but they're a little different so more consideration will need to be given on how to appropriately format them. Additionally, as part of the decoded items was the 'flight_category', which refers to the aforementioned grading system. I used this and changed the grading appropriately, along with the traditional colours used to represent the varying categories of flight rules.

When things started to get complicated was when it came to items that had specific units. I ended up going with the concept of checking the current units set, and changing the items units according to those settings, as the API even provided the same pieces of data in different units (which is very helpful!). I often received a `TypeError` when accessing data involving clouds due to the possibility of there not being some values such as a ceiling/base, which would in turn return undefined. This was fixed by incasing them in try-catch blocks so that if such an error occurs the value just defaults to 'N/A' without stopping the program. Finally, Visibility has been kept with its primary units (in this case, meters) as aviation laws regarding visibility utilise meters and so do reports, and so I saw no reason to change it.

Why I did it this way

There were two main ways I could've gone about implementing the unit-changing values. One was through using the previous unit conversion tool I made beforehand, and the other was to use the built in units and switch them according to the units set. I strived to go with the first choice because it meant that I could change the units in settings and immediately have the units change. This quickly became frustrating because the units given by the API were in knots, and just getting that number meant that the units may consider it in another unit that is not the same. This essentially means, that if I was to fix this it would mean fetching the units from the API that is the same as the unit to be converted to, and so on. (more complicated than it sounds), So out of this I realised that I should just rely on fetching the respective units from the API based on the current units set, so the units essentially can still convert although the 'update' button may need to be hit again for it to take effect, which for now I'm ok with.

Program output

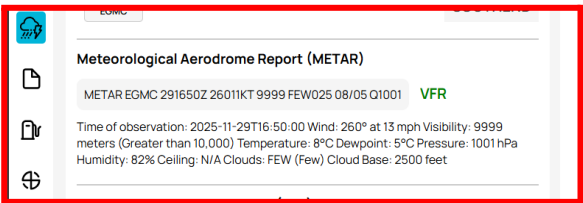


/debug

```
{
  "flight": {
    "METAR_searched": "METAR EGMC 291650Z 26011KT 9999 FEW025 08/05 Q1001",
    "METAR_searched_decoded": "\n  Time of observation: 2025-11-29T16:50:00\n\n  Wind: 260° at 13 mph\n\n  Visibility: 9999 meters (Greater than 10,000)\n\n  Temperature: 8°C\n\n  Dewpoint: 5°C\n\n  Pressure: 1001 hPa\n\n  Humidity: 82%\n\n  Ceiling: N/A\n\n  Clouds: FEW (Few)\n\n  Cloud Base: 2500 feet\n\n  ",
  }
}
```

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
88	Weather Parsing test - METAR	The METAR report for the searched aerodrome should be parsed to be readable by the user	

Error Analysis & Fixes

Test No: 88

Failure Description: Items aren't displayed line-by-line

Cause: Could be the way the items are saved in the session and it doesn't read the '\n' syntax as starting a new line, as they disappear but have no effect

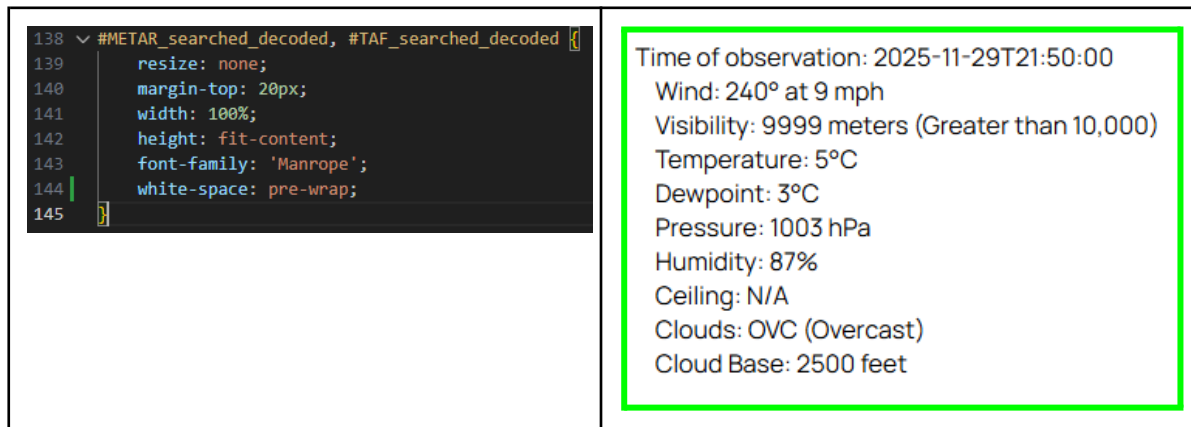
Fix: Research to find any specifics to outputting multi-line text, or use HTML formatting instead

Justification: The addition of the HTML formatting option as well is because I can use lists to hopefully create the same effect. But there may be a way to do this without this slightly lengthier process so it's added as an alternative if the first doesn't work.

Iteration 4.5 - Fixing METAR formatting error

Of the two options, it was actually option one that worked, with option 2 displaying the HTML tags as actual strings because that's how they were saved in the session.

The issue was that HTML doesn't read \n tags and so it just sees it as white space. However, changing the white-space value in css to 'pre-wrap' fixes it in that it just displays the items exactly as they were written in JS, so the \n tags weren't even needed.



Iteration Review

Met requirements: (Aim Complete) Display whether or not the conditions are suitable for Visual flight rules via a known grading system

Changes: N/A

Next steps: Parsing TAFs

Iteration 5 - Aerodrome Search: Parsing the TAF data

```
function parseTAF(taf) {  
  //parsing taf data depending on available fields  
  const parsed_TAF = taf.forecast.map(f => ({  
    start: f.timestamp?.from,  
    end: f.timestamp?.to,  
    wind: f.wind ?? null,  
    visibility: f.visibility ?? null,  
    clouds: (f.clouds ?? []).map(cloud => ({  
      type: cloud.text,  
      base_ft: cloud.base_feet_agl  
    })),  
    conditions: (f.conditions ?? []).map(cond => cond.text),  
    change: f.change?.indicator?.text ?? null,  
    probability: f.change?.probability ?? null  
  }));  
  
  let decoded_TAF = '';  
  
  //constructing decoded TAF string  
  parsed_TAF.forEach(period => { //iterate through each forecast period  
    decoded_TAF += `\\nFrom: ${period.start} To: ${period.end}\\n`;  
    if (period.change) { //if an item is undefined, it will be skipped  
      decoded_TAF += `  Change Indicator: ${period.change}\\n`;  
    }  
    if (period.probability) {  
      decoded_TAF += `  Probability: ${period.probability}%\\n`;  
    }  
    if (period.conditions.length > 0) {  
      decoded_TAF += `  Conditions: ${period.conditions.join(', ')}\\n`;  
    }  
  
    if (period.wind) {  
      decoded_TAF += `  Wind: ${period.wind.degrees}° at ${period.wind.speed_kts} knots\\n`;  
    }  
    if (period.visibility) {  
      decoded_TAF += `  Visibility: ${period.visibility.meters} meters (${period.visibility.meters_text})\\n`;  
    }  
    if (period.clouds.length > 0) {  
      period.clouds.forEach(cloud => {  
        decoded_TAF += `  Clouds: ${cloud.type} at ${cloud.base_ft} feet AGL\\n`;  
      });  
    }  
  });  
  
  return decoded_TAF;  
}
```

What I did

The TAF is significantly more difficult to parse due to the way it's laid out. A TAF doesn't just consist of a short single 1-line forecast, it gives forecasts for several time periods all in the same forecast. Some of these will include new cloud bases, new visibility, wind, or it may just include one new field- perhaps just a change in weather conditions. This means that the TAFs are **variable**, unlike the METAR that will always have those certain items as standard (except cloud base which I added using a try-catch statement).

Since any item has the ability to not be in the forecast, using try-catch blocks for all of them would be inefficient. The solution was to use the optional chaining operator (?.) which will by default return undefined if the string does not exist, which is significantly less code to do the same function.

Then, when constructing the string, before an item is added to the string, the `if (string)` statements check if the string exists (i.e. is not undefined) and will only add it to the final string if it indeed has content.

Why I did it this way

If I instead used try-catch blocks to do this, the code would become unnecessarily long and it would be more difficult to deal with an 'undefined' because it can only be dealt with in the catch part of the block. With this, there isn't an efficient way to code 'don't include it in the final string' within that block and so this contributed to my decision of just using the optional chaining operators, which work for this case almost completely in my favor.

Program output

Terminal Aerodrome Forecast (TAF)

```
TAF EGSS 300458Z 3006/0112 26009KT 9999 SCT025 BECMG 3020/3023 18011KT TEMPO
0100/0106 19018G28KT 6000 RA BKN010 PROB30 TEMPO 0103/0112 4000 +RA RADZ BKN007
BECMG 0106/0109 19016G26KT BKN010 TEMPO 0106/0112 6000 RA PROB30 TEMPO 0109/0112
20024G36KT
```

From: 2025-12-01T03:00:00 To: 2025-12-01T12:00:00

Change Indicator: Temporary

Probability: 30%

Conditions: Heavy Rain, Rain, Drizzle

Visibility: 4000 meters (4,000)

Clouds: Broken at 700 feet AGL

From: 2025-12-01T00:00:00 To: 2025-12-01T06:00:00

Change Indicator: Temporary

Conditions: Rain

Wind: 190° at 18 knots

Visibility: 6000 meters (6,000)

Clouds: Broken at 1000 feet AGL

From: 2025-11-30T20:00:00 To: 2025-12-01T06:00:00

Change Indicator: Becoming

Wind: 180° at 11 knots

Visibility: 9999 meters (Greater than 10,000)

Clouds: Scattered at 2500 feet AGL

From: 2025-11-30T06:00:00 To: 2025-11-30T20:00:00

Wind: 260° at 9 knots

Visibility: 9999 meters (Greater than 10,000)

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
88	Weather Parsing test - TAF	The TAF report for the searched aerodrome should be parsed to be readable by the user	PASS

Additional Changes

weather.js

```
//styling for item titles
//this takes temporary decoded weather data from the backend (if it exists) and inserts it into the correct divs
//along with proper HTML formatting
const resp = await fetch("/get-flight")
const weather = await resp.json();
var TEMPtaf_searched_decoded = weather.TAF_searched_decoded;
var TEMPmetar_searched_decoded = weather.METAR_searched_decoded;

if (TEMPtaf_searched_decoded && TEMPmetar_searched_decoded) { //more to be added for dep/arr wx
  var metar_searched_decodedTEMP = document.getElementById('METAR_searched_decoded');
  var taf_searched_decodedTEMP = document.getElementById('TAF_searched_decoded');
  taf_searched_decodedTEMP.innerHTML = TEMPtaf_searched_decoded
  metar_searched_decodedTEMP.innerHTML = TEMPmetar_searched_decoded
}
```

```
const decoded_METAR =
<b>Wind:</b> ${wind_dir}
<b>Visibility:</b> ${vis}
<b>Temperature:</b> ${temp}
<b>Dewpoint:</b> ${dew}
<b>Pressure:</b> ${pres}
```

What I did

Additionally, to make the item match the design I needed to make the titles of the categories in bold. This was a challenge for a number of reasons, specifically because, if I styled it with the `<b>` tags, on reloading the page, the `<b>` tags would be present in the text itself and no styling applied.

The solution I came up with utilises the fact that in the session, the `<b>` tags are stored as normal. First, it checks if an existing weather report is saved to session, which won't be the case when a new flight has been created. Then (if there is a report stored), it will load that report and update the innerHTML of the decoded output to match the session. And since `innerHTML` is used, the bold tags, which should be part of the string, should also apply too.

Program output



Meteorological Aerodrome Report (METAR)

METAR EGKK 300920Z 25005KT CAVOK 04/01 Q1014 **VFR**

Time of observation: 2025-11-30T09:20:00
Wind: 250° at 5 knots
Visibility: 9999 meters (Greater than 10,000)
Temperature: 4°C
Dewpoint: 1°C
Pressure: 1014 hPa
Humidity: 81%
Ceiling: N/A
Clouds: CAVOK (Clear skies)
Cloud Base: N/A

Terminal Aerodrome Forecast (TAF)

TAF EGKK 300459Z 3006/0112 26008KT 9999 SCT025 BECMG 3019/3022 19010KT PROB30 TEMPO 0100/0103 8000 -RA BECMG 0103/0106 19016G26KT BKN014 TEMPO 0103/0112 8000 -RA PROB30 TEMPO 0103/0112 4000 RA RADZ BKN009 PROB30 TEMPO 0110/0112 19024G36KT

From: 2025-12-01T03:00:00 **To:** 2025-12-01T12:00:00
Change Indicator: Becoming
Wind: 190° at 16 knots
Visibility: 9999 meters (Greater than 10,000)
Clouds: Broken at 1400 feet AGL

From: 2025-12-01T03:00:00 **To:** 2025-12-01T12:00:00
Change Indicator: Temporary
Conditions: Light Rain
Visibility: 8000 meters (8,000)

Iteration Review

Met requirements: (Aim complete) Provide weather data in both coded and decoded formats

Changes: HTML formatting changes to have bold titles

Next steps: Departure & Arrival aerodrome weather reports

Iteration 6 - Departure & Arrival Reports

Note:

- A lot of the work has already been done in terms of finding out how to fetch and format the METARs and TAF reports, which was the hardest part. Since I don't need to worry about this, I've gone full thrust and just coded the entire thing.
- The formatting of the departure and arrival reports are identical in every regard, the only difference being variable names (departure.something vs arrival.something for instance). **Thus, code snippets will be shown for the departure aerodrome only** because it won't benefit seeing the same thing twice. Most of the new stuff comes with the new html format at the top of each section as this differs from the searched aerodrome section.

app.py

```
183 #-----Departure airport
184 "WX_switch_dep" : "METAR",
185 "METAR_departure" : "",
186 "TAF_departure" : "",
187 "METAR_departure_decoded" : "",
188 "TAF_departure_decoded" : "",
189 "METAR_departure_grading" : "",
190 #-----Arrival airport
191 "WX_switch_arr" : "METAR",
192 "METAR_arrival" : "",
193 "TAF_arrival" : "",
194 "METAR_arrival_decoded" : "",
195 "TAF_arrival_decoded" : "",
196 "METAR_arrival_grading" : "",
197 }
```

weather.html

```
<!-------DEPARTURE AERODROME WEATHER-->
<div id="WX_departure" class="WX_item">
<!-------HEADERS----->
<div id="WX_departure_header" style="display: flex; align-items: center;">
<svg style="enable-background:new 0 0 24 24; width: 30px; margin-right: 10px" version="1.1" viewBox="0 0 24 24"
</div>
<!--header containing airport name and toggle buttons-->
<div id="WX_departure_header2">
<div id="WX_departure_name"><b>{{ flight.departureAirport_code if flight.departureAirport_code else 'N/A' }}</b>
<div><button class="WX_switch {{ 'active' if flight.WX_switch_dep == 'METAR' else '' }}" id="WX_departure_METAR_
</div>
<hr size="3.5" color="#cdcdcd" noshade="" style="margin-top: 25px;">
<!-------BODY----->
<!--METAR-->
<div id="METARDeparture">
<span id="METAR_departure" class="report">{{ flight.METAR_departure }}</span><span id="METAR_departure_grading"
<div id="METAR_departure_decoded"></div>
</div>
<!--TAF-->
<div id="TAFDeparture">
<div id="TAF_departure" class="report">{{ flight.TAF_departure }}</div>
<div id="TAF_departure_decoded"></div>
</div>
</div>
```

weather.css

```

/*-----Departure and Arrival Aerodrome Weather-----*/
#WX_departure_header2, #WX_arrival_header2 {
  display: flex;
  justify-content: space-between;
  margin-top: 12px;
  height: 36px;
}

#WX_departure_name, #WX_arrival_name {
  font-size: 17px;
  background-color: var(--bg);
  padding: 5px 10px;
}

.WX_switch {
  background-color: var(--bg);
  border-radius: 5px;
  padding: 5px 10px;
  font-family: 'Manrope';
  font-weight: 600;
  font-size: 17px;
  margin-left: 15px;
  width: 150px;
  max-height: 70px;
}

#WX_departure_METAR_switch.active, #WX_arrival_METAR_switch.active,
#WX_departure_TAF_switch.active, #WX_arrival_TAF_switch.active {
  background-color: var(--primary);
}

#METARDeparture, #METARArrival, #TAFDeparture, #TAFArrival {
  margin-top: 10px;
}

#TAF_departure, #TAF_arrival {
  resize: none;
  width: 100%;
  height: fit-content;
  font-family: 'Manrope';
  font-size: 14px;
  pointer-events: none;
}

#METAR_departure_decoded, #TAF_departure_decoded, #METAR_arrival_decoded, #TAF_arrival_decoded {
  resize: none;
  margin-top: 20px;
  width: 100%;
  height: fit-content;
  font-family: 'Manrope';
  white-space: pre-wrap;
}

```

weather.js

For HTML bold formatting

```
//departure weather
if (TEMPtaf_departure_decoded && TEMPmetar_departure_decoded) {
  var metar_departure_decodedTEMP = document.getElementById('METAR_departure_decoded');
  var taf_departure_decodedTEMP = document.getElementById('TAF_departure_decoded');
  taf_departure_decodedTEMP.innerHTML = TEMPtaf_departure_decoded
  metar_departure_decodedTEMP.innerHTML = TEMPmetar_departure_decoded
}
```

For the 'METAR' - 'TAF' toggle switches

```
//-----Departure and arrival weather switches-----
const WX_departure_METAR_switch = document.getElementById('WX_departure_METAR_switch');
const WX_departure_TAF_switch = document.getElementById('WX_departure_TAF_switch');
const WX_arrival_METAR_switch = document.getElementById('WX_arrival_METAR_switch');
const WX_arrival_TAF_switch = document.getElementById('WX_arrival_TAF_switch');
const METARDeparture = document.getElementById('METARDeparture');
const TAFArrival = document.getElementById('TAFArrival');
const TAFDeparture = document.getElementById('TAFDeparture');
const METARArrival = document.getElementById('METARArrival');

//initializing visibility of metar/taf based on current switch settings
//only the selected report is shown. The other is hidden.
const response = await fetch("/get-flight")
const flight = await response.json();

if (flight.WX_switch_dep == "METAR") {
  METARDeparture.style.display = "block";
  TAFDeparture.style.display = "none";
} else if (flight.WX_switch_dep == "TAF") {
  METARDeparture.style.display = "none";
  TAFDeparture.style.display = "block";
}

if (flight.WX_switch_arr == "METAR") {
  METARArrival.style.display = "block";
  TAFArrival.style.display = "none";
} else if (flight.WX_switch_arr == "TAF") {
  METARArrival.style.display = "none";
  TAFArrival.style.display = "block";
}

WX_departure_METAR_switch.addEventListener('click', async () => {
  WX_departure_METAR_switch.classList.add('active');
  WX_departure_TAF_switch.classList.remove('active');
  METARDeparture.style.display = "block";
  TAFDeparture.style.display = "none";
  update("WX_switch_dep", "METAR");
});
```

```

WX_departure_METAR_switch.addEventListener('click', async () => {
  WX_departure_METAR_switch.classList.add('active');
  WX_departure_TAF_switch.classList.remove('active');
  METARdeparture.style.display = "block";
  TAFdeparture.style.display = "none";
  update("WX_switch_dep", "METAR");
});

WX_departure_TAF_switch.addEventListener('click', async () => {
  WX_departure_TAF_switch.classList.add('active');
  WX_departure_METAR_switch.classList.remove('active');
  TAFdeparture.style.display = "block";
  METARdeparture.style.display = "none";
  update("WX_switch_dep", "TAF");
});

```

Grabbing all the HTML elements to be used when updating weather reports

```

const metar_departure = document.getElementById('METAR_departure');
const taf_departure = document.getElementById('TAF_departure');
const metar_departure_decoded = document.getElementById('METAR_departure_decoded');
const taf_departure_decoded = document.getElementById('TAF_departure_decoded');
const METAR_departure_grading = document.getElementById('METAR_departure_grading');

```

Updated `update_wx` function

```

255  update_wx.addEventListener('click', async () => {
256    //Add loading feedback here
257    const wx_searched = await getWeather(code_searched.value);
258    const wx_departure = await getWeather(departure_aerodrome);
259    const wx_arrival = await getWeather(arrival_aerodrome);

276    //-----DEPARTURE WX-----
277    await update("METAR_departure", wx_departure.metar.raw_text);
278    metar_departure.textContent = wx_departure.metar.raw_text;
279
280    await update("TAF_departure", wx_departure.taf.raw_text);
281    taf_departure.textContent = wx_departure.taf.raw_text;
282    await update("METAR_departure_decoded", await parseMETAR(wx_departure.metar, METAR_departure_grading));
283    metar_departure_decoded.innerHTML = await parseMETAR(wx_departure.metar, METAR_departure_grading);
284
285    await update("TAF_departure_decoded", await parseTAF(wx_departure.taf));
286    taf_departure_decoded.innerHTML = await parseTAF(wx_departure.taf);

```

What I did

Main changes

The main changes made regard the toggle switches between the METAR and TAF reports. The logic used here is the same as what was done for the satellite toggle in the settings menu, in that I update the session to show what switch was selected once it was pressed.

WX_departure/arrival_METAR/TAF_switch are the actual button switches, whilst METARdeparture/Arrival and TAFdeparture/Arrival are the actual div elements that will have their display property toggled depending on the current switch position. That way, when METAR is pressed, only the METAR is shown and the TAF is hidden, and vice versa.

However, upon pressing 'update' both types of reports still get updated. It's necessary that I initialise them first so that only the relevant report is shown on first load.

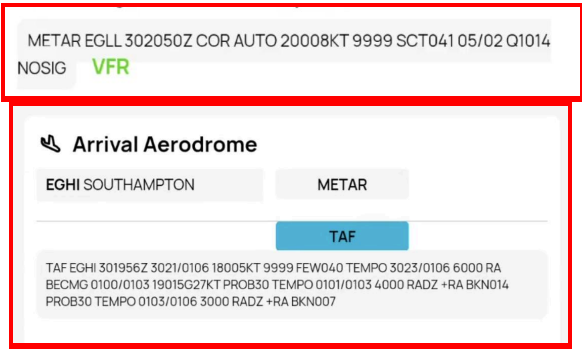
The rest of the changes is what was coded already, such as the added bold formatting for the departure and arrival reports, and the updated `update_wx` function to now include the variables for the departure and arrival aerodromes.

Program output

 Weather tab functionality test.mp4

Known Limitations / Missing features

- No proper formatting when first entering tab
- No 'uploading...' feedback added
- No validation for missing inputs

Test No.	Test Description	Expected Outcome	Actual Outcome
89	Weather tab functionality test	Upon pressing the update button weather reports should generate and be displayed as appropriate	PASS (see program output)
90	Weather tab formatting	No formatting errors (other than those at the beginning) should appear when codes are generated successfully	

Error Analysis & Fixes

Test No: 90

Failure Description: Minor formatting errors

Cause: The first is caused by a prolonged METAR (as they can vary in length), the other randomly occurs on reload, so I'll have to check the debug console to see what happened.

Fix: For the metar issue, making the div smaller should eliminate the issue.

Iteration 6.5 - Fixing minor formatting issues

The first error was due to height restrictions on the METAR_searched/departure/arrival elements, changing the height property to fit-content solved the issue and it now acts like a proper div:

WEATHER

Q Aerodrome Search

EGCC

MANCHESTER

Meteorological Aerodrome Report (METAR) **MVFR**

METAR EGCC 041520Z 28006KT 9999 -RA FEW011 BKN023 06/04 Q0996
TEMPO RA BKN011

Time of observation: 2025-12-04T15:20:00

Wind: 280° at 6 knots

The second error, however, was random and I couldn't pinpoint a reason why it kept happening or how to willingly make the error occur. I have made the decision to note this error and come back to it in the post-development section as I will be more able to try and reproduce the error when time constraints are a little more relaxed. *If I decided to continue debugging this error it could take hours that I could be utilizing for other sections and therefore wasting time on sections that have not yet been started.*

Iteration Review

Met requirements:

(Aim complete) Give the user the ability to view the Meteorological Aerodrome Reports (METAR) and Terminal Aerodrome Forecasts (TAF) of the departure and arrival aerodromes, as well as any aerodrome of their choice

Essential Feature Complete: Integration of aerodrome forecasts and reports

Changes: N/A

Next steps: Validation and styling for feedback and missing inputs

Iteration 7 - Validation and styling for feedback and missing inputs

weather.css

```
#working_icon {
  width: 100px;
  margin-right: 10px;
}

/*-----Covers when search not possible-----*/

#METAR_departure.active, #TAF_departure.active,
#METAR_arrival.active, #TAF_arrival.active,
#METAR_searched.active, #TAF_searched.active {
```

```

    display: block;
}

('inop' here meaning inoperative which is commonly used in aviation)
#WX_departure.inop, #WX_arrival.inop {
    pointer-events: none;
    opacity: 0.4;
}

```

weather.js

```

//--declarations for the boxes containing METAR/TAF reports--
const WX_departure = document.getElementById('WX_departure');
const WX_arrival = document.getElementById('WX_arrival');

update_wx.addEventListener('click', async () => {
    //'updating...' feedback
    update_wx.style.pointerEvents = 'none'; //disable button while updating
    update_wx.style.opacity = '0.6';
    update_wx.style.backgroundColor = 'var(--onhover)';
    update_wx.style.border = 'none';
    update_wx.style.alignItems = 'center';
    update_wx.innerHTML = `<svg id='working_icon' data-name="Layer 1" id="Layer_1" fill='curr
    <p style="font-size: 14px;">WORKING...</p>

    //fetching weather data for searched, departure and arrival aerodromes
    const wx_searched = await getWeather(code_searched.value);
    const wx_departure = await getWeather(departure_aerodrome);
    const wx_arrival = await getWeather(arrival_aerodrome);

    //restoring button state
    update_wx.style.pointerEvents = 'auto';
    update_wx.style.opacity = "";
    update_wx.style.border = "";
    update_wx.style.backgroundColor = "";
    update_wx.innerHTML = `<svg id='update_icon' xmlns="http://www.w3.org/2000/svg" xml
    <p>UPDATE</p>

} if (wx_departure) {
    WX_departure.classList.remove('inop');
    metar_departure.classList.add('active');
    taf_departure.classList.add('active');
    //-----DEPARTURE WX-----

```

What I did

To finish off this section, the final thing to work on was what happens when there is no data to deal with, in this case, missing airports.

First off, if the update button is pressed, only data that is present will be updated. This was done by simply adding an if statement before updating the elements, checking if the relevant airport had been added. For the case of the search aerodrome, there will always be the option to plug in a code and fetch the weather for that aerodrome, so I have not applied any styling that provides visual feedback to the user. For the departure and arrival

aerodromes however, I wanted to make it so that the user cannot interact with the relevant subsections until the relevant airport has been inputted. Thus, for both boxes, `pointer-events` is set to none, and `opacity` to 0.4 to give feedback to the user that it cannot be interacted with. Of course, when the aerodromes are inputted, the class `inop` is removed and the styling goes back to before, and the user can interact with the subsections.

Another styling change was to ensure that those boxes containing METAR reports (and the one TAF report for the searched aerodrome) are hidden when there is no report being shown for that subsection. This is handled by the `METAR/TAF_searched/departure/arrival.active` styles, whose name is added to the class list of the relevant subsection div before a report is generated.

Finally, there was the 'WORKING...' feedback as per the example mentioned in the [limitations](#) subsection of this document. This simply involved me changing the background of the button, and the innerHTML to show the new icon and wording, and then simply reverting back to the previous styling when the process is complete.

Why I did it this way

There were other ways I could have made it clear to the user that the departure/arrival reports aren't accessible when the departure/arrival airports aren't yet inputted. My initial attempt saw me placing a darkened box over both subsections, with a text in the middle stating that the relevant aerodrome needs to be inputted first. This was turned over when I discovered that changing the z-layer (so it can be on-top) wouldn't work because the display is already set to 'grid' which in short doesn't allow such z-index formatting within. The second option was to remove the boxes completely, which wouldn't be intuitive at all because it would make the user assume that they can only search an aerodrome. It would also introduce styling issues now that two grid items are gone. The third and current option works well because it still provides the user with that visual feedback, and the user can still tell that it's the lack of an aerodrome input that's the problem because of the 'N/A' that replaces the name of the aerodrome.

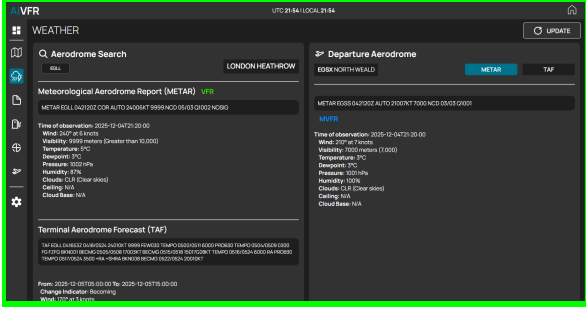
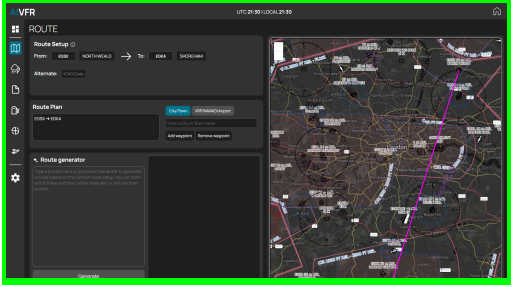
Program output

 Weather tab functionality test with validation.mp4

Known Limitations / Missing features

- Random formatting bug still unsolved

Test No.	Test Description	Expected Outcome	Actual Outcome
91	Weather tab functionality test with validation	Upon pressing the update button, the correct formatting should apply. If an aerodrome input is missing then the report for that aerodrome should not be generated, and vice versa.	PASS (see program output)

92	Dark mode test: Weather Tab	Darkmode should apply the correct coloring to all elements.	
93	Dark mode test: Route Tab (forgot to do it before)	Darkmode should apply the correct coloring to all elements.	

Iteration Review

Met requirements: Section 7 complete

Changes: Minor styling changes and inclusion of extra fields (conditions) in the METAR report. Also made the entire page have reduced opacity when loading a report (not seen above) for additional feedback.

Next steps: Section 8 - Navlog Tab

Section Review

Aims completed in this section:

- **Complete an Essential feature: Integration of aerodrome forecasts and reports**
- **Complete a Success criteria item:** Use of APIs to fetch weather data
- Give the user the ability to enter any aerodrome and access their weather data
- Give the user the ability to view the Meteorological Aerodrome Reports (METAR) and Terminal Aerodrome Forecasts (TAF) of the departure and arrival aerodromes, as well as any aerodrome of their choice
- Display whether or not the conditions are suitable for Visual flight rules via a known grading system
- Provide weather data in both coded and decoded formats

Aims not completed in this section:

N/A

Iterative tests complete:

- Light/dark Mode switches correctly with the correct colours

Remarks: Nothing to address.

Section 8: Navlog Tab

Aims of this section

- **Complete 2 Essential features: Pilot navigation log, Basic flight calculations**
- **Complete 2 success criteria items:** Flight computer, Printable pilot Navigation log
- Provide the ability of the user to choose altitudes, speeds and other logistical aspects of their route as per design
- Allow AI the ability to generate these aspects itself, providing a justification for its thought process.
- Allow the user to download the log in a .pdf format

Iteration 1 - Calculating Route distance

route.js

```
//-----DISTANCE FINDING-----

async function getDistance(coord_arr) {
  var len = coord_arr.length;
  let total = 0
  var separate_distances = []

  //iterate through the route coordinates
  //and find the distance between each point and the next
  for (let i = 0; i = len; i++) {
    var distance = map.distance(coord_arr[i], coord_arr[i+1]) / 1852; //for
nautical miles
    console.log(distance) //TEST
    total = total + distance
    separate_distances.push(distance)
  }

  //updating the session to include distance values
  await update("separate_distances", separate_distances)
  await update("distance[value]", total)

  return total
}

async function updateDistances() {
  fetch('/get-flight')
  .then(response => response.json())
  .then(async FlightData => {
    const route_coords = FlightData.route;

    //only update the distance when a waypoint exits
```

```

    if (route_coords.length >= 3) {
        var total = getDistance(route_coords)
    }

    //updating the distance to display to the user
    const response = await fetch('/time-distance');
    if (response.ok) {
        const text = await response.text();
        document.getElementById('timeDistance').textContent = text;
    }

    return total
  });
}

```

```

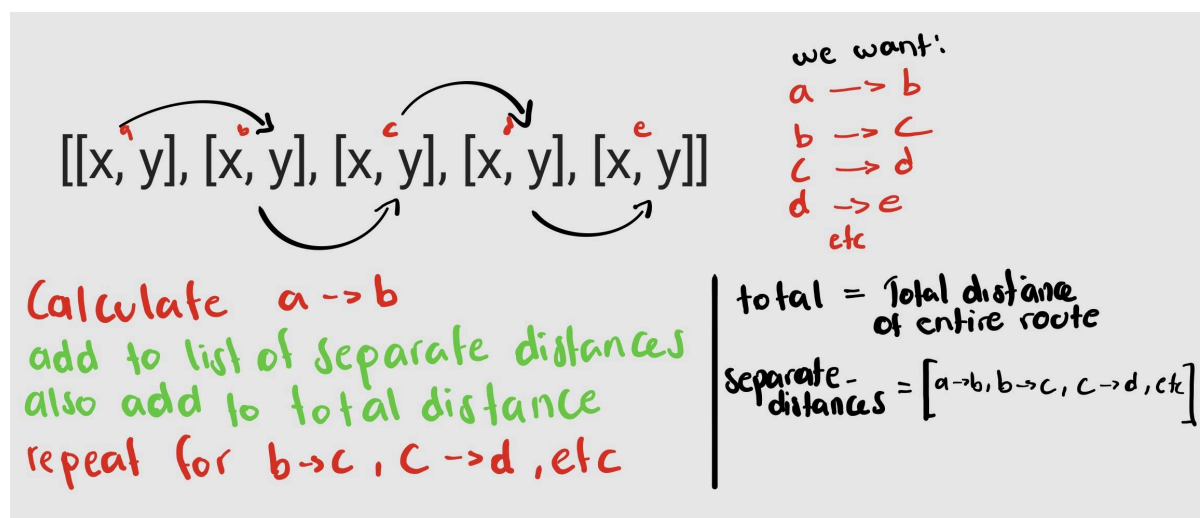
#-----Time & Distance-----
@main.route('/time-distance')
def get_time_distance():
    return f"{session['flight_data']['flight']['time']} | {session['flight_data']['flight']['distance']['output']}", 200

```

What I did

Looking back at the design for the navigation log, it seems like I would have had to come back and work on finding the distance between points and across the whole route. This was necessary because you can't calculate the time without the distance, and both these features are of large importance in a pilot log regardless, and so missing it out would render the essential feature incomplete.

So to find the distance, using the leaflet plugin I used to display distances on the map wasn't going to work as it never gave me access to the actual distance being shown. After some research, I discovered that leaflet has their own built-in method called `distance` which takes 2 coordinates and outputs their distance in metres. Using this method I came up with a way to calculate distances for each separate leg and thus the whole entire route, shown in the image I made below for simplicity.



The image above describes `getDistance`, and the function `updateDistances` runs this function so long as a waypoint exists in the route (i.e if the route has 3 or more points: Departure, a waypoint, and arrival).

After this, the time-distance text in the route tab is updated, by fetching the new values from the session, bringing them to the frontend and updating the `textContent` to display the new values. The `separate_distances` will then be used here in the navlog for each of the distances for each leg.

Program output

```
✖ ▶ Uncaught (in promise) TypeError: Cannot read properties of null (reading 'distance')
    at getDistance (route.js:630:28)
    at route.js:652:25
```

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
94	Distance calculation test	Distances should be calculated for each leg and output in the console, and the total distance should be displayed as text in the route tab.	<pre>✖ ▶ Uncaught (in promise) TypeError: Cannot read properties of null (reading 'distance') at getDistance (route.js:630:28) at route.js:652:25</pre>

Error Analysis & Fixes

Test No: 94

Failure Description: TypeError when calculating distance

Cause: Most likely due to the fact that the map class is set to none in this case, because I tested this within `load_map()` and a distance was indeed displayed.

Fix: Find a way to make sure that the map class is initialised before this line runs.

Justification: The alternative would be to incorporate this within the `load_map()` function, which would work but would decrease the flexibility I get with the distances function as I can't have the distances calculated by themselves without resetting the map, which in itself is unnecessary.

Iteration 1.5 - Fixing TypeError in distance calculation

```
638 //-----DISTANCE FINDING-----
639
640 async function getDistance(coord_arr) {
641     var len = coord_arr.length;
642     let total = 0
643     var separate_distances = []
644     let operative = false
645
646     //finding correct units
647     const response = await fetch("/get-settings");
648     const settings = await response.json();
649     const distance_unit = settings.units_distance;
650
651     if (distance_unit === "nautical_mile") {var divideBy = 1852}
652     else if (distance_unit === "kilometer") {var divideBy = 1000}
653     else if (distance_unit === "us_statute_mile") {var divideBy = 1609.344}
654
655     //iterate through the route coordinates
656     //and find the distance between each point and the next
657     do {
658         try {
659             for (let i = 0; i < len - 1; i++) {
660                 var distance = map.distance(coord_arr[i], coord_arr[i+1]) / divideBy;
661                 total = total + distance
662                 separate_distances.push(distance)
663                 operative = true
664             }
665         } catch (error) { //in case the map isnt loaded yet
666             operative = false
667             await reload_map();
668         }
669     } while (!operative);
670
671     //updating the session to include distance values
672     await update("separate_distances", separate_distances)
673     await update("distance.value", total)
674
675     return total
676
677 }
```

```
async function load_map() {
    return new Promise((resolve) => {
        if (map) {
            map.remove();
            map = null;
        }
    })
    //getting API keys from backend
}
```

What I did

The issue indeed was that map had been set to null before map.distance was called, so an error occurred. In order to make sure the map was still initialized, I made the function `load_map()` return a promise that only gets resolved when the map has been initialized. It's when the promise is resolved that the code has been 'awaited' and other stuff begins to run, from what I understand.

Now this method actually worked, and I got distances returned in the console and the session was being updated as required. However, sometimes `map.distance` gets called and I get the error of it being in that null state again. This happened randomly and I couldn't reproduce it on command, and it seemed like this race between the asynchronous functions and the method call can sometimes result in the class being set to null before its method is called, but this actually doesn't happen more often than not.

To make sure this didn't have an effect on producing distances, I used a do-while loop nested in a try-catch block. What happens is that the variable `operative` is set to false by default, and only on a successful run will it be set to true, in which case the do-while loop ends and the function continues. In the case where an error is caught, it means the map was found to be null and so `operative` remains false, and the process is retried again after reloading the map. This worked surprisingly well, and in the cases where an error occurred, it took just one other attempt for it to work again, and so I was happy with the result.

Test No.	Test Description	Expected Outcome	Actual Outcome
95	Distance calculation test	Distances should be calculated for each leg and output in the console, and the total distance should be displayed as text in the route tab.	PASS (See program output)

Program output

 Distance functionality test.mp4

Iteration Review

Met requirements: Distance issue resolved

Changes: Distances now adjusts for units as well (also in code snippet above)
Disclaimer wording has been changed to be more concise (see program output)

Next steps: Initial elements and styling for navlog

Iteration 2 - Navlog header elements and styling

navlog.css

```
#navlogHeader {
  margin-top: 10px;
  display: flex;
  justify-content: space-between;
  align-items: top;
}

#autofillButton {
  background-color: var(--primary);
```

```

}

#variationContainer {
  background-color: var(--highlight);
  padding: 10px;
  height: fit-content;
  border-radius: 10px;
}

#variation {
  border: 1px solid var(--text);
  border-radius: 5px;
  padding: 5px;
  margin-left: 10px;
  width: 100px;
  text-align: center;
  font-weight: 600;
  font-family: 'Manrope';
  font-size: 13px;
  max-width: 80px;
  background-color: var(--bg);
  border: 1px solid var(--onhover);
}

#autofillButton {
  text-align: center;
  font-size: 16px;
  font-weight: 500;
  font-family: 'Manrope';
  padding: 7px 10px 7px 10px;
  width: fit-content;
  border-radius: 8px;
  cursor: pointer;
  margin-right: 13px;
}

#AI_icon {
  margin-right: 5px;
}

```

navlog.html

```
<!--=====HEADER=====-->
<div class="pageTitle" style="display: flex; align-items: center; gap: 10px;">
  <div id="title">NAVIGATION LOG</div>
  <span id="alertBox"></span>
</div>
<div id="navlogHeader">
  <!--Variation-->
  <div id="variationContainer">
    Variation (+° <b>E</b>, -° <b>W</b>): <input type="number" id="variation" value="0" step="1"/>
  </div>
  <!--Autofill and download buttons-->
  <div>
    <span id="autofill">
      <button id="autofillButton" class="baseButton"><svg fill="currentColor" height="20" id="AI_icon" viewBox="0 0 20 20" w
    </span>
    <span id="download">
      <button id="downloadNavlogButton" class="baseButton"><svg height="24" viewBox="0 0 24 24" width="24" xmlns="http://www.
    </span>
  </div>
</div>
<hr size="3.5" color="#cdcdcd" noshade>
<!--=====BODY (TABLE)=====-->
</main>
```

What I did

The styling was pretty straight-forward with nothing significant to report. However, there are some major changes to some of the inputs that need to be explained and justified.

The top header part, as-per design used to look a little like this:

NAVIGATION LOG

TAS: KTS VARIATION: WIND: /

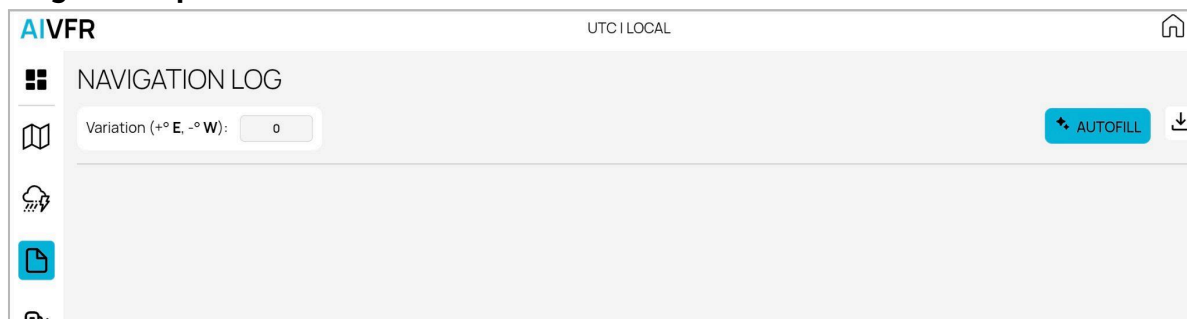
With True Airspeed (TAS), magnetic variation and wind speed and velocity as singular values. The new design, however, includes just the magnetic variation as a singular fixed value, with the rest to be included in the table as flexible values. The magnetic variation input now also has an aid, showing the user to input positive numbers representing degrees East, and negative numbers indicating degrees West.

Why I did it this way

The purpose of a value being flexible means that the values can differ for each leg. For example, the wind for **a** → **b** can be set to be something different for **b** → **c**, which can sometimes be the case if not relying on a spot-wind forecast, which can sometimes help for accuracy (albeit the spot-winds are pretty accurate). **Not making the wind flexible won't account for those users who would like to use separate winds and would force them to utilise spot-winds which, for them, may not be desirable.** The same applies to TAS, although it's less common in slower fixed-pitch propeller general aircraft who don't need to worry so much about managing their cruising speed as the speed is often determined by what RPM is most efficient for the engine. This could be different for more complex aircraft, however, so it's considerate to also allow this value to also be flexible. However, the magnetic variation used to calculate the magnetic heading would remain the same for all legs. The reason for not allowing *this* value to be flexible is because within the UK this value is not only so small to a degree that it is often (wrongfully) neglected, but it also does not change so significantly moving across the UK. **So, allowing it to be flexible would just increase the code workload as the values most likely will be the same for the entire route anyway so this would be completely unnecessary.**

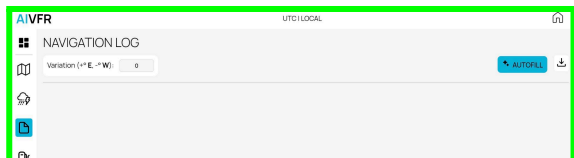
Other values, such as planned altitude, **must** be flexible because the altitude for one leg almost certainly changes and may be different for the other leg, often due to airspace restrictions.

Program output



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
96	Initial styling	Element positioning and styling should be as per design/as per intended changes.	

Iteration Review

Met requirements: N/A

Changes: TAS and wind are now flexible values.

Next steps: Navigation log table

Iteration 3 - Navigation log table styling

navlog.html

```
<!--=====BODY (TABLE)=====-->|
<table id="navlogTable">
  <thead> <!--header row-->
    <tr>
      <th style="width:20%">FROM / TO</th>
      <th>MSA</th>
      <th>TAS</th>
      <th>TRACK (°T)</th>
      <th>Wind DIR (°)</th>
      <th>Wind SPD (KT)</th>
      <th>HDG (°T)</th>
      <th>HDG (°M)</th>
      <th>GS (KT)</th>
      <th>DIST (NM)</th>
      <th>TIME (Min)</th>
    </tr>
  </thead>
  <tbody id="navlogTableBody">
    <!--Rows will be dynamically added here-->
    <!--TEST ROW-->
    <tr>
      <!--FROM/TO--><td>POINT A - POINT B</td>
      <!--MSA--><td><input type="number" class="tableInput" value=""></td>
      <!--TAS--><td><input type="number" class="tableInput" value=""></td>
      <!--TRACK (°T)--><td><input type="number" class="tableInput" value=""></td>
      <!--Wind DIR (°)--><td><input type="number" class="tableInput" value=""></td>
      <!--Wind SPD (KT)--><td><input type="number" class="tableInput" value=""></td>
      <!--HDG (°T)--><td><div class="tableCalculated"></div></td>
      <!--HDG (°M)--><td><div class="tableCalculated"></div></td>
      <!--GS (KT)--><td><input type="number" class="tableInput" value=""></td>
      <!--DIST (NM)--><td><input type="number" class="tableInput" value=""></td>
      <!--TIME (Min)--><td><div class="tableCalculated"></div></td>
    </tr>
  </tbody>
</table>
```

navlog.css

```
#navlogTable {
  margin-top: 15px;
  width: 100%;
  border-collapse: collapse;
  background-color: var(--highlight);
}

thead {
  background-color: black;
}

th {
  color: white;
  width: 120px;
}

#navlogTable, td, th {
  text-align: center;
```

```

padding: 5px;
font-family: 'Manrope';
font-size: 14px;
font-weight: 700;
border: 2px solid var(--text);
}

/*highlighting for automatically calculated values*/
td:nth-child(7), td:nth-child(8), td:nth-child(11) {
    background-color: var(--onhover);
}

.tableInput, .tableCalculated {
    width: 100%;
    height: 50px;
    box-sizing: border-box;
    border: 1px solid var(--onhover);
    background-color: transparent;
    text-align: center;
    font-family: 'Manrope';
    font-size: 14px;
    border-radius: 5px;
    font-weight: 700;
    color: var(--text);
}

.tableCalculated {
    pointer-events: none;
    display: flex;
    align-items: center;
    justify-content: center;
    border: none;
}

/*Remove spinners from number inputs*/
input[type=number]::-webkit-outer-spin-button,
input[type=number]::-webkit-inner-spin-button {
    -webkit-appearance: none;
    margin: 0;
}

```

What I did

The aim of this iteration was to achieve the correct styling and element formatting for the navlog table, and this was to be done by creating the headers (which will be unchanging) and then making a sample row and just playing with the styling until I get something that matches my design. In reality, the table rows will be added dynamically, as will be seen in the next few iterations.

The format for most of the rows consists of an input box (with `type='number'`). Inputs that are to be calculated automatically, as per design, are gray with `pointer-events: none;` to ensure they cannot be interacted with.

Why I did it this way

Styling the table this way allows me to make sure everything is set and matches the design before any sort of javascript gets involved. It also allows me to determine the HTML required to get the specific styling and element placement of each cell in a row. *Without this technique I would have to guess what will suit best and then have to constantly make adjustments to the innerHTML to try and make it match the design which is considerably harder as I can't access things like a live server.*

Program output

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
97	Table styling and positioning	Table styling and positioning should match the design.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Saving variation to session and adding rows dynamically

Iteration 4 - JS: Variation, adding rows dynamically

app.py

```
#-----NAVLOG TABLE-----

@main.route('/makeNavlog')
def makeNavlog():
    #clear existing rows
    session["flight_data"]["flight"]["NAVLOG"]["rows"] = []

    route_names = session["flight_data"]["flight"]["route_names"]
    separate_distances = session["flight_data"]["flight"]["separate_distances"]
    separate_bearings = session["flight_data"]["flight"]["separate_bearings"]
    len_route = len(route_names)

    #only make the navlog if there are at least 3 points
    #(departure, destination and at least 1 waypoint)
    if len_route >= 3:
        for i in range(len_route - 1):
            row = { #making each row as a dictionary.
                    #'calculated' indicates if its user input or calculated automatically
                    "FROM/TO": {"value": route_names[i] + " - " + route_names[i+1], "calculated": True},
                    "MSA": {"value": "", "calculated": False},
                    "TAS": {"value": "", "calculated": False},
                    "TRACK (°T)": {"value": separate_bearings[i], "calculated": True},
                    "Wind DIR (°)": {"value": "", "calculated": False},
                    "Wind SPD (KT)": {"value": "", "calculated": False},
                    "HDG (°T)": {"value": "", "calculated": True},
                    "HDG (°M)": {"value": "", "calculated": True},
                    "GS (KT)": {"value": "", "calculated": False},
                    "DIST (NM)": {"value": separate_distances[i], "calculated": True},
                    "TIME (Min)": {"value": "", "calculated": False}
                }
            #append the row to the navlog
            session["flight_data"]["flight"]["NAVLOG"]["rows"].append(row)

    session.modified = True
    return jsonify(session["flight_data"]["flight"]["NAVLOG"]), 200
```

```
separate_bearings = [], #bearings for each point
"variation" : 0,
"NAVLOG" : {
    "rows" : []
```

navlog.js

```
const table = document.getElementById('navlogTableBody');

await makePLOG();

//-----NAVLOG FUNCTIONS
//Update variation
const variation = document.getElementById('variation');

//wait for a change in variation input, then update
variation.addEventListener('change', async () => {
  await update('variation', variation.value);
});

//-----CREATING PLOG TABLE

async function makePLOG() {
  //clear the table's HTML first (so it dissapears)
  table.innerHTML = '';

  await fetch('/get-flight')
    .then(response => response.json())
    .then(async FlightData => {
      const route_names = FlightData.route_names;
      const len = route_names.length;

      //only make the navlog if there are at least 3 points
      if (len >= 3) {
        await fetch('/makeNavlog');
        const log = FlightData.NAVLOG;

        //create all rows, which will be dependent on route length
        log.rows.forEach(row => {
          const newrow = table.insertRow();
          //iterate through each row and create the table using the fetched data, and with specific HTML
          for (const key in row) {
            const cell = row[key];
            if (cell.calculated) {
              newrow.innerHTML += `<td><div class="tableCalculated">${cell.value}</div></td>`;
            } else {
              newrow.innerHTML += `<td><input type="number" class="tableInput" value="${cell.value}"></td>`;
            }
          }
        });
      }
    });
}
```

navlog.html

```
<!--No data message when no route created--><h2 id="noData" class="{ 'active' if
flight.route|length < 3 else '' }">NO ROUTE DATA</h2>
```

navlog.css

```
/*---inactive header state when no route available---*/
#navlogHeader.inactive, #navlogTable.inactive {
  opacity: 0.5;
  pointer-events: none;
}

#noData {
  position: absolute;
  top: 40%;
  left: 50%;
  transform: translate(-50%, -50%); /*centering*/
  font-family: 'Manrope';
}
```

```

    font-size: 20px;
    font-weight: 600;
    background-color: hsla(0 100% 50% / 0.6);
    padding: 10px 25px 10px 25px;
    border-radius: 10px;
    display: none;
}

#noData.active {
    display: block;
}

```

What I did

First, to update the magnetic variation, I made an event listener that listens for a change in the value within the input box. If a change is detected then the variation in the session is updated to the value in the box.

Next, it came to making the table items. From my research, the best way to store a table is not by just copying the HTML and throwing it into the session, but instead by using a JSON format to store each row. Hence, the `NAVLOG` key in the session stores an array called `rows` which stores an array which will contain all the rows in a json format. The `makeNavlog()` function works by first clearing the table, then storing all the pre-calculated data in the session, in their own variables (i.e. distances and bearings). The route names are also taken to be put in the table. Then, if a route with at least one waypoint exists, a for loop goes through each waypoint, with `i`, then index, pointing to the relevant waypoint. This is so that, when adding the track value for a specific waypoint, it takes the index of the current waypoint, but from the relevant array (so `separate_bearings`) and uses that in its output. Same applies with the to-from cell, where `route_names[i] + " - " + route_names[i+1]` takes the route name from the current index and then concatenates it with the route name from the next index, so we get an 'a-b'. After each row's data is created it is appended to the `rows` array.

Each cell has a `value` key, and a `'calculated'` key, which is a boolean. The purpose of this is so that if the calculated value is set to true, the HTML for that cell will change as per what we saw in the last iteration. This is applied in the `makePLOG()` function, where again the route length is checked before running the `makeNavlog` function, and taking the response in a JSON format. Then we first add the rows, by firing `insertRow()` for each row in the `rows` array. Then, we look at each separate row, and add a cell containing the relevant data (`cell.value`), but with specific HTML depending on whether `'calculated'` is true or false. This allows me to have those specific columns in that gray colour with the relevant styling as well.

Finally, I added some HTML so that when the route has insufficient waypoints for a navlog to be created, the screen has its opacity reduced and a NO ROUTE DATA warning appears to tell the user that no data is present for anything to be created yet.

Why I did it this way

For this section, I decided to update items in the session when a change is detected, rather than when 'enter' is pressed. This was principally because many edits are going to be made in this section, and in order for the user experience to be nice and fluid, I believe this is the best option. It would take much longer and thus be more frustrating to the user should they need to constantly press enter to update a value, rather than just simply clicking off the input box, completely abstracting this process from the user.

Program output

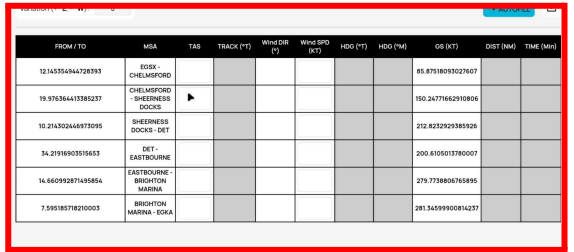
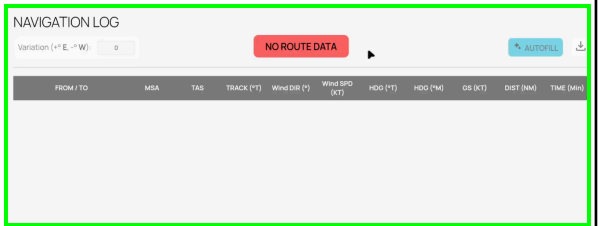
variation (+° E, -° W): 0 + AUTOFILL

FROM / TO	MSA	TAS	TRACK (°T)	Wind DIR (°)	Wind SPD (KT)	HDG (°T)	HDG (°M)	GS (KT)	DIST (NM)	TIME (Min)
12.145354944728393	EGSX - CHELMSFORD							85.87518093027607		
19.976364413385237	CHELMSFORD - SHEERNESS DOCKS							150.24771662910806		
10.214302446973095	SHEERNESS DOCKS - DET							212.8232929385926		
34.21916903515653	DET - EASTBOURNE							200.6105013780007		
14.660992871495854	EASTBOURNE - BRIGHTON MARINA							279.7738806765895		
7.595185718210003	BRIGHTON MARINA - EGKA							281.34599900814237		

 navlog creation test.mp4

Known Limitations / Missing features

- Decimal outputs not rounded
- Manual reload required before table creation

Test No.	Test Description	Expected Outcome	Actual Outcome
98	Navlog creation test	The navigation log should be created with the relevant rows, waypoints, distances and bearings already present, with styling applied to the respective columns.	
99	Variation update text	Upon updating the variation and clicking off, the session should update with the new variation.	PASS (See program output)
100	No data warning	If less than 3 waypoints exist in the route then a NO ROUTE DATA alert should appear, screen dimmed out and no table generated.	

Error Analysis & Fixes

Test No: 98

Failure Description:

Navigation Log - AIVFR

expert-space-potato-pjp: x | expert-space-potato-pjp: wx-5000.app.github.dev/navlog

UTC: 17:03 | LOCAL: 17:03

NAVIGATION LOG

Variation (+° E, -° W): -2

AUTOFILL

FROM / TO	MSA	TAS	TRACK (°T)	Wind DIR (°)	Wind SPD (KT)	HDG (°T)	HDG (°M)	GS (KT)	DIST (NM)	TIME (Min)
12.145354944728393	EGSX - CHELMSFORD							85.87518093027607		
19.976364413385237	CHELMSFORD - SHEERNESS DOCKS							150.24771662910806		
10.214302446973095	SHEERNESS DOCKS - DET							212.8232929385926		
34.21916903515653	DET - EASTBOURNE							200.6105013780007		
14.660992871495854	EASTBOURNE - BRIGHTON MARINA							279.7738806765895		
7.595185718210003	BRIGHTON MARINA - EGKA							281.34599900814237		

wrong positioning

Cause: Incorrect ordering of data as it is stored in the session. Apparently this isn't the issue though, but it's the source, the real problem is that I am relying on the order of data in the session to produce the table, which is in alphabetical order. This ultimately ruins the output.

Fix: Use fixed columns and match each cell to their respective column to get correct ordering regardless of how it is stored in session.

Justification: Another fix could have been to just find a way to ensure that the session does not store items in alphabetical order, and instead in the order that I made them in. While this would have worked, this would not be practical long term as the session order isn't actually the issue, and it could potentially mess with other future features throughout the webtool's development.

Iteration 4.5 - Fixing positioning errors in the navlog

app.py

```
NAVLOG : {  
  "headers" : ["FROM/TO", "MSA", "TAS", "TRACK (°T)", "Wind DIR (°)", "Wind SPD (KT)", "HDG (°T)", "HDG (°M)"],  
  "rows" : []  
},
```

Extra fix - rounding the numbers

```

    "TRACK (°T)": {"value": round(separate_bearings[i]), "calculated": True},
    "Wind DIR (°)": {"value": "", "calculated": False},
    "Wind SPD (KT)": {"value": "", "calculated": False},
    "HDG (°T)": {"value": "", "calculated": True},
    "HDG (°M)": {"value": "", "calculated": True},
    "GS (KT)": {"value": "", "calculated": False},
    "DIST (NM)": {"value": round(separate_distances[i]), "calculated": True},
    "TIME (Min)": {"value": "", "calculated": False}

```

navlog.js

```

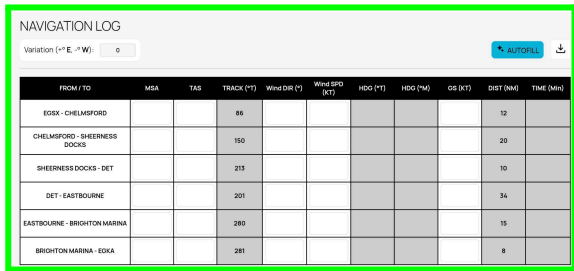
//iterate through each row and create the table using the fetched data, and with specific HTML
log.headers.forEach(column => {
  const cell = row[column];

```

Now, the columns key in the session describes the exact layout of the columns in the session, and makePLOG adds cells with reference to the respective column in the order specified in that **columns** array, **NOT** the ordering of items in each row in the session.

Program output

 navlog creation test 2.mp4

Test No.	Test Description	Expected Outcome	Actual Outcome
101	Navlog creation test	The navigation log should be created with the relevant rows, waypoints, distances and bearings already present, with styling applied to the respective columns.	

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Updating data in the table

Iteration 5 - Updating data in the table

navlog.js

```
//On page load, check if route has changed
await fetch('/get-flight')
  .then(response => response.json())
  .then(async FlightData => {
    const route_changed = FlightData.route_changed;
    if (route_changed === "True") {
      await makePLOG(); //allows for re-creation of PLOG, with calculations etc
      update("route_changed", "False");
    } else {
      await refreshPLOG();
    }
  });
```

```
async function makePLOG() {
  await fetch('/get-flight')
    .then(response => response.json())
    .then(async FlightData => {
      const route_names = FlightData.route_names;
      const len = route_names.length;

      //only make the navlog if there are at least 3 points
      if (len >= 3) {
        await fetch('/makeNavlog');
        await refreshPLOG();
      }
    });
}
```

makePLOG() only runs when its first time or change has been made.

Separated functions

```
async function refreshPLOG() {
  const newtable = document.createElement('tbody');
  newtable.id = 'navlogTableBody';
```

refreshPLOG()'s function is to load the HTML using data stored in Session.

```
  await fetch('/get-flight')
    .then(response => response.json())
    .then(async FlightData => {
      const log = FlightData.NAVLOG;
      const len = FlightData.route_names.length;

      if (len >= 3) {
        //create all rows, which will be dependent on route length
        log.rows.forEach(row => {
          const newrow = document.createElement('tr');
          newtable.appendChild(newrow);
          //iterate through each row and create the table using the fetched data, and with specific HTML
          log.headers.forEach(column => {
            const cell = row[column];
            if (cell.calculated) {
              newrow.innerHTML += `<td><div class="tableCalculated">${cell.value}</div></td>`;
            } else {
              newrow.innerHTML += `<td><input type="text" class="tableInput" value="${cell.value}"></td>`;
            }
          });
          table.innerHTML = newtable.innerHTML;
        });
      }
    });
}
```

New table made in background

Then table has innerHTML swapped. So switch is seamless and innerHTML of table doesn't need to be set to ""

```
//-----UPDATING PLOG TABLE

table.addEventListener('change', async (event) => {
  const target = event.target;
  if (target.classList.contains('tableInput')) {
    //get the row and column of the changed input
    const rowIndex = target.parentElement.parentElement.rowIndex - 1; //adjust for header row
    const cell = target.parentElement;
    const columnIndex = cell.cellIndex;
    const columnName = table.parentElement.querySelector('thead').rows[0].cells[columnIndex].innerText;

    //validating inputs
    if (target.value < 0) {
      target.value = 0; //all inputs cannot be negative
    } if (columnName == "Wind DIR (°)" && target.value > 360) {
      target.value = 360; //wind direction cannot be more than 360
    }
    const newValue = target.value;

    //send the updated value to the server
    await fetch('/update-cell', {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        row: rowIndex,
        column: columnName,
        value: newValue
      })
    });
    await refreshPLOG();
  });
});
```

```
//-----CLEARING PLOG TABLE

const clearNavlogButton = document.getElementById('clearNavlogButton');

clearNavlogButton.addEventListener('click', async () => {
  await fetch('/clearNavlog');
  await refreshPLOG();
});
```

app.py

```
#-----NAVLOG
"route_changed" : "False",
```

makeNavlog() closeup

```
"MSA": {"value": session["flight_data"]["flight"]["NAVLOG"]["rows"][i]["MSA"]["value"] if i <
len(session["flight_data"]["flight"]["NAVLOG"]["rows"]) else "", "calculated": False},
```

```
if len_route >= 3:
  for i in range(len_route - 1):
    #dealing with removed waypoints: if an a-b pair no longer exists, remove that row as its not present anymore
    try:
      if session["flight_data"]["flight"]["NAVLOG"]["rows"][i]["FROM/TO"]["value"] != route_names[i] + " - " + route_names[i+1]:
        session["flight_data"]["flight"]["NAVLOG"]["rows"].pop(i)
    except IndexError: #if the index is out of range, a new row should be made anyway
      pass
```

```

@main.route('/makeNavlog')
def makeNavlog():
    route_names = session["flight_data"]["flight"]["route_names"]
    separate_distances = session["flight_data"]["flight"]["separate_distances"]
    separate_bearings = session["flight_data"]["flight"]["separate_bearings"]
    len_route = len(route_names)

    #only make the navlog if there are at least 3 points
    #(departure, destination and at least 1 waypoint)
    if len_route >= 3:
        for i in range(len_route - 1):
            #dealing with removed waypoints: if an a-b pair no longer exists, remove that row as its not present
            try:
                if session["flight_data"]["flight"]["NAVLOG"]["rows"][i]["FROM/TO"]["value"] != route_names[i] + " - " + route_names[i+1]:
                    session["flight_data"]["flight"]["NAVLOG"]["rows"].pop(i)
            except IndexError: #if the index is out of range, a new row should be made anyway
                pass

            row = { #making each row as a dictionary.
                    #'calculated' indicates if its user input or calculated automatically
                    "FROM/TO": {"value": route_names[i] + " - " + route_names[i+1], "calculated": True},
                    "MSA": {"value": session["flight_data"]["flight"]["NAVLOG"]["rows"][i]["MSA"]["value"] if i < len(route_names) - 2 else "", "calculated": True},
                    "ALT PLAN (FT)": {"value": session["flight_data"]["flight"]["NAVLOG"]["rows"][i]["ALT PLAN (FT)"]["value"] if i < len(route_names) - 2 else "", "calculated": True},
                    "TAS": {"value": session["flight_data"]["flight"]["NAVLOG"]["rows"][i]["TAS"]["value"] if i < len(route_names) - 2 else "", "calculated": True},
                    "TRACK (°T)": {"value": round(separate_bearings[i]), "calculated": True},
                    "Wind DIR (°)": {"value": session["flight_data"]["flight"]["NAVLOG"]["rows"][i]["Wind DIR (°)"]["value"] if i < len(route_names) - 2 else "", "calculated": True},
                    "Wind SPD (KT)": {"value": session["flight_data"]["flight"]["NAVLOG"]["rows"][i]["Wind SPD (KT)"]["value"] if i < len(route_names) - 2 else "", "calculated": True},
                    "HDG (°T)": {"value": "", "calculated": True},
                    "HDG (°M)": {"value": "", "calculated": True},
                    "GS (KT)": {"value": "", "calculated": True},
                    "DIST (NM)": {"value": round(separate_distances[i]), "calculated": True},
                    "TIME (Min)": {"value": "", "calculated": True}
                }
            #append the row to the navlog
            try:
                session["flight_data"]["flight"]["NAVLOG"]["rows"][i] = row
            except IndexError:
                session["flight_data"]["flight"]["NAVLOG"]["rows"].append(row)

    session.modified = True
    return {"headers": session["flight_data"]["flight"]["NAVLOG"]["headers"],

```

```

#-----Updating data
@main.route('/update-cell', methods=["POST"])
def update_cell():
    row_index = request.json.get("row")
    column_name = request.json.get("column")
    new_value = request.json.get("value")

    navlog_rows = session["flight_data"]["flight"]["NAVLOG"]["rows"]

    #validate row index (shouldn't ever be triggered)
    if row_index < 0 or row_index >= len(navlog_rows):
        return jsonify({"error": "Invalid row index"}), 400

    #validate column name (also shouldn't ever be triggered)
    if column_name not in navlog_rows[row_index]:
        return jsonify({"error": "Invalid column name"}), 400

    #update the cell value
    navlog_rows[row_index][column_name]["value"] = new_value
    navlog_rows[row_index][column_name]["calculated"] = False #mark as user input

    #-----Calculations retrigged-----

    session.modified = True
    return jsonify({"status": "ok"}), 200

```

navlog.html

```
28 <!--Autofill, clear and download buttons-->
29
30 <div>
31   <span id="autofill">
32     <button id="autofillButton" class="baseButton"><svg fill="currentColor" height="
33   </span>
34   <span id="clear">
35     <button id="clearNavlogButton" class="baseButton"><svg width="25px" height="25px
36   </span>
37   <span id="download">
38     <button id="downloadNavlogButton" class="baseButton"><svg height="24" viewBox="6
39   </span>
```

What I did

Splitting the functions

Firstly, I made the decision to split the makePLOG function into two functions, `makePLOG()` And `refreshPLOG()`. This was due to the fact that the makeNavlog() function, which is called by the makePLOG function, reset the entire table, which was not ideal. The makeNavlog() function in the backend now works to account for any changes made to the route and make a new table based on these changes, whilst retaining data already inputted. Then instead I incorporated a new HTML button that links to a `clearNavlog()` function (which is just a copy of the old makeNavlog function which cleared the table - see previous iteration), so that clicking it clears all the data in the table seamlessly.

Route changed

To identify whether the route has changed (and to thus determine whether or not to run makePLOG or just refreshPLOG) a key `"route_changed"` which is a boolean (as a string just for consistency) is set to false whenever the `add_waypoint/remove_waypoint` or AI generation functions are run. Then upon opening the navlog tab, if `"route_changed"` is set to true, then makePLOG is run to ensure that the changes are applied. If not, the table is just refreshed and should look the same as how it did when the user left it. The refreshPLOG function works solely on what is currently in the session, so as long as the session stays the same then the output will remain the same too.

Updating data

To update the data, all I needed was a way to change the relevant value in the session, then run refreshPLOG, since that relies only on what's currently in the session. To do this I extracted the row index, column name and new value for the particular cell that is being updated, which will be detected by an event listener listening for a change in the value. Then, as usual It is sent to the frontend, where the values undergo validation (which it should always pass given that all data is retrieved from the session). Then the values are updated in the session, and the 'calculated' value is set to false in case of any error switching it (all editable values should have this set to false). It will be then where calculations will be performed and the session updated with the results of those calculations, before the function ends and frontend re-runs refreshPLOG with all the updated session data.

Other input validation involves ensuring all inputted values are not negative, and that the wind direction value is between 0 and 360. [E6BX](#), for example, hardcodes it so that if you input a larger number, it resets to the limit. This is also what I have implemented.

Program output

 Navlog update test and input validation.mp4

Known Limitations / Missing features

- The table refresh deselects the currently selected cell if one is selected immediately after

Test No.	Test Description	Expected Outcome	Actual Outcome
102	Updating navlog test	Upon updating a value, the log should update in the session and items should remain on reload. When a waypoint is added/removed, the table should update accordingly.	PASS (See program output)

Input Validation Tests

Test Data	Expected Outcome	Actual Outcome
A negative number in a cell	The number should reset to 0 once the cell is clicked off of.	PASS (See program output)
A number above 360 in the 'Wind DIR' cell	The number should reset to 360 once clicked off of.	PASS (See program output)

Iteration Review

Met requirements: (Aim complete) Provide the ability of the user to choose altitudes, speeds and other logistical aspects of their route as per design

Changes: Made the limit for the Wind DIR to be 359 instead of 360 (because that's 0) - Not documented above

Next steps: Calculations

Iteration 6 - Performing Calculations

```
def calculate_row(row_index):
    row = session["flight_data"]["flight"]["NAVLOG"]["rows"][row_index]
    variation = float(session["flight_data"]["flight"]["variation"])

    #getting necessary values
    tas = float(row["TAS"]["value"]) if row["TAS"]["value"] != "" else ""
    track = float(row["TRACK (°T)"]["value"])
    wind_dir = float(row["Wind DIR (°)"]["value"]) if row["Wind DIR (°)"]["value"] != "" else ""
    wind_spd = float(row["Wind SPD (KT)"]["value"]) if row["Wind SPD (KT)"]["value"] != "" else ""
    distance = float(row["DIST (NM)"]["value"])

    #only calculate if tas, wind_dir and wind_spd are provided
    if all(value != "" for value in [tas, wind_dir, wind_spd, track]):

        #heading
        row["HDG (°T)"]["value"] = calc_HDG(tas, track, wind_dir, wind_spd)
        row["HDG (°M)"]["value"] = (row["HDG (°T)"]["value"] + variation)

        #ground speed
        row["GS (KT)"]["value"] = calc_GS(tas, wind_dir, wind_spd, float(row["HDG (°T)"]["value"]))

    else:
        return

    gs = row["GS (KT)"]["value"]
    if gs not in [0, ""]:
        row["TIME (Min)"]["value"] = round((distance / gs) * 60, 1) #time in minutes
```

```
#-----Calculating calculated fields

#TRUE HEADING
def calc_HDG(tas, track, wind_dir, wind_spd):
    #convert inputs to radians
    track_rad = radians(track)
    wind_dir_rad = radians(wind_dir)
    wind_spd_ratio = wind_spd / tas

    #calculate wind correction angle
    wca = asin(wind_spd_ratio * sin(wind_dir_rad - track_rad))

    #calculate heading
    hdg_rad = track_rad + wca
    hdg = degrees(hdg_rad) % 360

    return round(hdg)

#GROUND SPEED
def calc_GS(tas, wind_dir, wind_spd, hdg):
    #convert inputs to radians
    wind_dir_rad = radians(wind_dir)
    hdg_rad = radians(hdg)

    #calculate ground speed
    gs = sqrt(tas**2 + wind_spd**2 - 2 * tas * wind_spd * cos(wind_dir_rad - hdg_rad))

    return round(gs)
```



```
#----recalculating magnetic heading for all rows (for when variation is changed)
@main.route('/recalculate-magnetic-HDG')
def recalculate_variation():
    variation = float(session["flight_data"]["flight"]["variation"])
    rows = session["flight_data"]["flight"]["NAVLOG"]["rows"]

    for i in range(len(rows)):
        row = rows[i]
        #only recalculate if HDG (°T) is calculated
        if row["HDG (°T)"]["value"] != "":
            row["HDG (°M)"]["value"] = (row["HDG (°T)"]["value"] + variation)

    session.modified = True
    return jsonify({"status": "ok"}), 200
```

```
//-----NAVLOG FUNCTIONS
//Update variation
const variation = document.getElementById('variation');

//wait for a change in variation input, then update
variation.addEventListener('change', async () => {
    await update('variation', variation.value);
    await fetch('recalculate-magnetic-HDG'); //recalculate magnetic headings
    await refreshPLOG();
});
```

What I did

To make calculations, for a particular row, I simply made the parameter of the `calculate_row()` function be the row index, so that I can use that index to get all the required information from that row. If even one of these items were empty, then no calculations will be performed. The `if all()` statement checks that none of the data is an empty string, which is what it would have been set to if no data was present. Once all values are confirmed to exist, true heading is calculated, and to get magnetic heading, we add on the variation. Then ground speed is also calculated, and from this we calculate the time by doing distance/ground speed X 60 (due to distance being in NM) to get the time in minutes. Often we receive the time to one decimal place so I've kept it like this also.

The formulas used to calculate the true heading and ground speed are simply standard equations, which I can show below. You will recognise that this is exactly what you can see in the respective python function.

True Heading

$$WCA = \arcsin\left(\frac{V_w \cdot \sin(\theta_w - TRK)}{TAS}\right)$$

Where:

V_w = wind speed

θ_w = wind direction from which the wind is blowing (in degrees true)

TRK = desired track (degrees true)

TAS = true airspeed

Then True Heading = TRK + WCA.

Groundspeed

$$GS = \sqrt{TAS^2 + V_w^2 - 2 \cdot TAS \cdot V_w \cdot \cos(\theta_w - TH)}$$

Finally, I needed a way to recalculate just the magnetic heading once the variation had been changed. This was done through the `recalculate_variation()` which iterates through each row and adjusts the true heading value depending on the current variation set.

Why I did it this way

Most of what I did regarding calculations is pretty standard and wouldn't be feasible to have been done another way such as in JS due to the lack of direct access to the session. If I did do this in JS, I would have to fetch data and then use the update function to update specific data, which gets really complicated especially when it comes to getting particular values in particular rows. This is the same reason as to why the recalculate variation function is not implemented in JS.

Program output (See below for explanation)

AIVFR											
UTC 09:21 LOCAL 09:21											
FROM / TO	MSA	ALT PLAN (FT)	TAS	TRACK (°T)	Wind DIR (°)	Wind SPD (KT)	Hdg (°T)	Hdg (°M)	GS (KT)	DIST (NM)	TIME (Min)
EGSX - CHIPPING ONGAR A414/B184			91	99	151	10	104	103	84	3.4	2.4
CHIPPING ONGAR A414/B184 - QEII			91	178	151	11	175	174	81	14.9	11
QEII - TUNBRIDGE WELLS			91	179	168	09	178	177	82	19.9	14.6
TUNBRIDGE WELLS - EASTBOURNE			91	179	168	08	178	177	83	21.8	15.8
EASTBOURNE - EGKA			91	280	156	11	274	273	97	22.3	13.8

North Weald - Shoreham											
SR 08:04 Z, MCT 07:23 Z											
EGSX North Weald 321 ft (12 hPa)	MSA	Level	Trk	Wind	Hdg	GS	Dist	Time	FuelR	ETA	ATA
Chipping Ongar A414/B184 Interchange		1,300	2,000	099	151/10	103	69	3.4	3	65.9	
Qeii		1,700	2,200	178	151/11	174	81	15	11	62.8	
Royal Tunbridge Wells		2,400	2,200	179	168/09	177	82	20	15	58.7	
Eastbourne		2,000	2,200	178	168/08	176	82	22	16	54.2	
EGKA Shoreham 7 ft (0 hPa)		1,600	2,200	280	156/11	273	96	23	14	50.2	
SS 15:58 Z, ECT 16:37 Z											
Speeds are 91 kt TAS.											
83 59 45.2											

The output above is a direct comparison of values in a pilot log of the same route on real navigation software ([Skydemon](#)) and my webapp. I was happy to see that for the most part, all values were in fact spot-on accurate, with some values just being one off of the value produced by Skydemon, which I can accept given that being one unit/degree off should not have an impact on reaching the desired point. Moreover, VFR is visual and so any discrepancies should not be detrimental to the pilot as they should be referring to visual references to check that they are going in the right direction. Regardless, I'll repeat, one degree/unit off is most definitely not an issue.

There is one value that is several units off of what was expected and that is the ground speed for the first leg. I believe this may be because Skydemon took a lower TAS for the climb especially because all the other ones are accurate meaning there is nothing wrong with the formulae.

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
103	Calculation accuracy test	The output numbers should be within ± 1 unit of off of the value output by Skydemon	PASS

Iteration Review

Met requirements: Essential feature complete: Basic flight calculations

Success criteria item complete: Flight computer

Changes: Added highlighting to magnetic heading column (common in most PLOGs), ensured bearings are 3 digit, and increased font size.

Wind DIR (°)	Wind SPD (KT)	HDG (°T)	HDG (°M)	GS (KT)	DIST (NM)	TIME (Min)
025	10	074	073	35	12.1	20.7

Next steps: The Alternate aerodrome

Iteration 7 - The Alternate Aerodrome

route.js

```
153 //-----Alternate Airport (ICAO)-----
154 const alternateAirport_code = document.getElementById("alternate_code");
155 const alternateAirport_name = document.getElementById("alternate_name");
156
157 alternateAirport_code.addEventListener("keydown", async (event) => {
158     if (event.key === "Enter") {
159         alternateAirport_name.style.display = "inline";
160         alternateAirport_name.textContent = "...";
161
162         //validating data
163         if (verifyICAO(alternateAirport_code.value)) {
164             const airportDetails = await fetchAirportDetails(alternateAirport_code.value);
165             if (airportDetails.country == "GB") {
166                 //swapping cords to lat-long
167                 const alternateCoords = airportDetails.coordinates;
168                 [alternateCoords[0], alternateCoords[1]] = [alternateCoords[1], alternateCoords[0]];
169                 //updating session data
170                 await update("alternateAirport_code", alternateAirport_code.value.toUpperCase());
171                 await update("alternateAirport_name", airportDetails.name);
172                 //updating the alternate coords
173                 await update("alternateAirport_coords", alternateCoords);
174                 await update("route_changed", "True"); //to trigger navlog reload
175
176                 alternateAirport_code.style.textTransform = "uppercase";
177                 alternateAirport_name.textContent = airportDetails.name;
178                 updateDistances();
179                 await reload_map(); //reload the map to show the alternate route
180
181             } else if (airportDetails.country != "GB") {
182                 showAlert("Only UK aerodromes are supported");
183                 alternateAirport_name.style.display = "none";
184             }
185         } else {
186             showAlert("Invalid ICAO code");
187             alternateAirport_name.style.display = "none";
188         }
189     }
190 }
```

```
//adding alternate
if (alternate_coords.length == 2) {
    const alt_line = L.polyline([route[route.length - 1], alternate_coords], { color: '■#ffa500' , weight
    .addTo(map)
    .showMeasurements();
}
```

```

696     do {
697         try {
698             for (let i = 0; i < len - 1; i++) {
699                 var distance = map.distance(coord_arr[i], coord_arr[i+1]) / divideBy;
700                 total = total + distance
701                 var bearing = getBearing(coord_arr[i], coord_arr[i+1]);
702
703                 if (alternate_coords.length == 2) { //only update if alternate is set
704                     var update_alternate = true
705                     var alt_distance = map.distance(coord_arr[i+1], alternate_coords) / divideBy;
706                     var alt_bearing = getBearing(coord_arr[i+1], alternate_coords);
707                 }
708
709                 separate_bearings.push(bearing)
710                 separate_distances.push(distance)
711                 operative = true
712             }
713
714             if (update_alternate) {
715                 await update("alternate_distance", alt_distance);
716                 await update("alternate_track", alt_bearing);
717             }
718
719         } catch (error) { //in case the map isnt loaded yet
720             operative = false
721             await reload_map();
722         }
723     } while (!operative);
724

```

navlog.css

```

01  /*highlighting for automatically calculated values, but not for alternate airport*/
02  tr:not(.alternate) td:nth-child(8), tr:not(.alternate) td:nth-child(10), tr:not(.alternate) td:nth-child(12) {
03      background-color: var(--onhover);

```

navlog.js (refreshPLOG())

```

74  //format alternate airport row differently
75  if (row["FROM/TO"].value === arrival_code + " - " + alternate_code) {
76      newrow.style.backgroundColor = "var(--alternate)";
77      newrow.classList.add('alternate')
78  }
79

```

app.py

```

631  #alternate airport details
632  alternate_track = session["flight_data"]["flight"]["alternate_track"]
633  alternate_distance = session["flight_data"]["flight"]["alternate_distance"]
634  alternateAirport_code = session["flight_data"]["flight"]["alternateAirport_code"]
635  arrivalAirport_code = session["flight_data"]["flight"]["destinationAirport_code"]
636
637  if alternateAirport_code != "":
638      #if a row already exists for the alternate airport, store it and remove it from the navlog temporarily
639      if session["flight_data"]["flight"]["NAVLOG"]["rows"][-1]["FROM/TO"]["value"] == route_names[-1] + " - " + alter
640          existing_alt_row = session["flight_data"]["flight"]["NAVLOG"]["rows"][-1]
641          session["flight_data"]["flight"]["NAVLOG"]["rows"].pop(-1)
642
643      #if the last row contains a waypoint beggining with the arrival airfield, remove it as the alternate airport has
644      #this prevents duplicate rows for the alternate airport, and only applies if above statement is false
645      elif f"{arrivalAirport_code} - " in session["flight_data"]["flight"]["NAVLOG"]["rows"][-1]["FROM/TO"]["value"]:
646          session["flight_data"]["flight"]["NAVLOG"]["rows"].pop(-1)
647          existing_alt_row = None
648
649      else:
650          print("no existing alt row")
651          existing_alt_row = None
652

```

```

203     "alternate_track" : 0,
204     "alternate_distance" : 0,

685     #adding the alternate airport row if an alternate airport has been specified
686     if alternateAirport_code != "":
687         #if a row already exists for the alternate airport, append it back
688         if existing_alt_row:
689             session["flight_data"]["flight"]["NAVLOG"]["rows"].append(existing_alt_row)
690
691         else: #if not make a fresh row
692             alt_row = {
693                 "FROM/TO": {"value": route_names[-1] + " - " + alternateAirport_code, "calculated": True},
694                 "MSA": {"value": "", "calculated": False},
695                 "ALT PLAN (FT)": {"value": "", "calculated": False},
696                 "TAS": {"value": "", "calculated": False},
697                 "TRACK (°T)": {"value": round(alternate_track), "calculated": True},
698                 "Wind DIR (°)": {"value": "", "calculated": False},
699                 "Wind SPD (KT)": {"value": "", "calculated": False},
700                 "HDG (°T)": {"value": "", "calculated": True},
701                 "HDG (°M)": {"value": "", "calculated": True},
702                 "GS (KT)": {"value": "", "calculated": True},
703                 "DIST (NM)": {"value": round(alternate_distance, 1), "calculated": True},
704                 "TIME (Min)": {"value": "", "calculated": True}
705             }
706             session["flight_data"]["flight"]["NAVLOG"]["rows"].append(alt_row)
707
708

```

clearNavlog() just contains a fresh row for the alternate.

What I did

The alternate aerodrome works by being an additional airport to the route, to which the pilot will divert to if for whatever reason they cannot make it to the arrival aerodrome. Since I don't want the alternate to be part of the route (because it's not), everything to do with the alternate airport is done sort of 'separately'. The alternate aerodrome has its own track value, coordinates, (all calculated from the same functions as seen above) etc that simply update when I enter an ICAO code.

The first change is actually in **route.js**, in which the line for the alternate route is shown from the arrival aerodrome, which is similar to what you'd see in professional software too. The addition here (for each type of map) is simply an orange dashed line, running from the last item in `route_names` (the arrival aerodrome) and ending at the coordinates of the arrival aerodrome. This was an addition to the original design, because I simply saw that it was possible to do.

The main change was to do with the navigation log. Given that the details of the navlog will never change (i.e the cell values- it will always be 'arrival-alternate'), if the navigation log has been loaded before (and so the alternate aerodrome row already exists) then the row is removed from the table and stored in the variable `existing_alt_row`. Then, the route can edit as it normally would, and then (because the variable exists) the alternate row stored in `d` will be appended back into the navlog. This way, the unchanging alternate aerodrome row doesn't interfere with any position changing/clearing that takes place whilst the route is being adjusted. If it sees there is a row for an alternate aerodrome but the alternate aerodrome doesn't match the currently set one, `existing_alt_row` is set to none so a fresh row can be made with the newly set alternate aerodrome.

Why I did it this way

Making the alternate aerodrome code separate from the main code was very important in that it does not mess up other features of the webtool so much, and it provides more freedom. For instance, because the alternate route is not added to "route" in the session, I

can freely change the alternate route without clearing the session, and simply reload the map and changes are made. If I added it to the route array it would be similar to the departure/arrival aerodromes, which need more work if I want to give the ability to change such values on the spot.

Program output

 alternate aerodrome test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
104	Alternate aerodrome line drawing	An orange dashed line should be drawn on the map once an alternate aerodrome is added, running from the arrival aerodrome to the alternate aerodrome	
105	Alternate aerodrome row: navlog	The navlog row for the alternate aerodrome should be styled as per design, and its values should remain the same when waypoints are added/removed from the route. The value should clear and change when the alternate aerodrome is changed.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: Added alt aerodrome line to map

Next steps: Implementing live inputs for navlog (see below)

Removal of AI autofill

After some consideration I've decided to remove the AI autofill feature from this tab. The reason being that It doesn't serve that much of a useful function to the user. It can only fill cells in the MSA, planned altitude, and (potentially) TAS if I gave it knowledge of the aircraft type to be flown, because all other values are either calculated or too complex to let the AI decide these values. With the existing spaces it can fill, I don't think the outputs will be as of high quality due to the lack of pre-knowledge of airspace data and such, and I believe the AI wouldn't be able to produce suitable altitudes that avoid controlled airspace, etc. Allowing AI to generate this data would be more 'dangerous' because with no map to visualise, if for whatever reason a pilot uses this data (highly unlikely nevertheless) they could penetrate class D airspace (Major airports like LHR, LGW, SEN), or, even worse, class A (Not permitted at all for VFR traffic). Therefore, in an effort to improve accuracy I may move this feature to another page such as fuel, where calculations are pretty straightforward and a decision by AI might actually be useful.

Iteration 8 - Live input refreshing on navlog

app.py

```
return jsonify({"hdgT": row["HDG (°T)"]["value"], "hdgM": row["HDG (°M)"]["value"], "gs": row["GS (KT)"]["value"], "time": row["TIME (Min)"]["value"]})

#heading
try:
    row["HDG (°T)"]["value"] = calc_HDG(tas, track, wind_dir, wind_spd)
    row["HDG (°M)"]["value"] = (row["HDG (°T)"]["value"] + variation)
except ValueError: #Return 'ERROR' for all fields if a calculation cannot be done
    return jsonify({"hdgT": "ERROR", "hdgM": "ERROR", "gs": "ERROR", "time": "ERROR"})
```

navlog.js

```
table.addEventListener('input', async (event) => {
    const target = event.target;
```

```
const data = await response.json();

if (data.data == 'none') {
    return
}

//update the specific row with returned values
const tableRow = table.rows[rowIndex];
for (let i = 6; i < tableRow.cells.length; i++) {
    const columnName = table.parentElement.querySelector('thead').rows[0].cells[i].innerText;
    //iterate through each column and replace the relevant cells with the fetched data
    if (columnName == "HDG (°T)") {
        tableRow.cells[7].querySelector('div').innerText = formatBearing(data.hdgT)
    } else if (columnName == "HDG (°M)") {
        tableRow.cells[8].querySelector('div').innerText = formatBearing(data.hdgM)
    } else if (columnName == "GS (KT)") {
        tableRow.cells[9].querySelector('div').innerText = data.gs
    } else if (columnName == "TIME (Min)") {
        tableRow.cells[11].querySelector('div').innerText = data.time
    }
}
});
```


What I did

This edit became necessary because refreshing the entire navlog brought some major issues, which I will explain. But regarding how this works:

The live updating of values in the table is something that [E6BX](#) also implemented in their navlog. This simply looks for an input (instead of a change) and updates the row instead of refreshing the entire table. The way I implemented this was by making the calculations return the necessary values for that row (given that calculate_row just calculates values for that specific row) and in the frontend, we iterate through each cell in the row (I've started from the 6th row because we only care about the calculated rows) and depending on the column, the respective value from the fetched calculations gets placed into the cell there and then. This happens every single time a user enters a character on the keyboard, and doesn't involve refreshing any data. This way, the table can be updated live and is more intuitive for the user.

Why I did it this way

Refreshing the table brought many problems- the most significant one being that it caused the selected cell to deselect as the table refreshed, which means the user could not move from one cell directly to another without double clicking. This is obviously 'buggy' and is not an intuitive feature. Moreover, after deploying the webapp to a website for production testing, it gets really, really slow having to refresh all the table data and often I could switch directly to another cell, and even start typing info before it just clears and deselects because it took that long for the table to refresh. Such a problem no longer exists with live updating.

Program output

 live update test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
106	Live update test	Calculations should update live as the user types and sufficient values are present. The user should also be able to switch directly from row to row.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: Navlog now updates live

Next steps: Downloading the navlog as PDF

Iteration 9 - Downloading the navlog as PDF

navlog.html

```
<script  
src="https://cdnjs.cloudflare.com/ajax/libs/html2pdf.js/0.10.1/html2pdf.bundle.min.js  
></script>
```

navlog.js

```
//-----DOWNLOADING NAVLOG  
const download = document.getElementById('downloadNavlogButton')  
  
download.addEventListener('click', () => {  
  downloadPDF();  
});  
  
function downloadPDF() {  
  //getting the table  
  const element = document.getElementById('navlogTable');  
  
  //using html2pdf to convert the html table into a image, then save to PDF  
  const options = {  
    margin: 0.5,  
    filename: 'navlog.pdf',  
    image: { type: 'jpeg', quality: 0.98 },  
    html2canvas: { scale: 1, useCORS: true },  
    jsPDF: { unit: 'in', orientation: 'landscape' }  
  };  
  
  html2pdf().set(options).from(element).save();  
}
```

What I did

The final thing to do was to add the ability to download the log as a PDF. To do this, I used the [html2pdf](#) plugin which uses html2canvas and jsPDF, to, as the names suggest, convert any HTML element into an image and then put said image in a PDF. Its full documentation provided the layout I needed to produce the desired result. The aim was to just have the <table> element be placed in the PDF which should contain everything the user needs. I then linked this code to the download button in HTML.

Why I did it this way

The alternative to using html2pdf was to use jsPDF alone, which takes the HTML and converts it into a PDF without changing it into an image first; the advantage of this being that the items in the list are selectable. I tried this, but ended up with a result that looked something like [this](#), which is not at all the size nor the font that is of the real thing. The ones generated by html2pdf however, were good enough and of a decent size to be used as an intuitive reference by the user.

Program output

FROM / TO	MSA	ALT PLAN (FT)	TAS	TRACK (°T)	Wind DIR (°)	Wind SPD (KT)	HDG (°T)	HDG (°M)	GS (KT)	DIST (NM)	TIME (Min)
EGSX - CHELMSFORD	1200	2200	90	086	097	17	088	088	73	12.1	9.9
CHELMSFORD - QEII				207						18.2	
QEII - BRIGHTON MARINA				199						41.6	
BRIGHTON MARINA - EGKA				281						7.6	

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
107	PDF generation test	Upon pressing the download button a PDF should be generated and should look like the original navlog.	

Iteration Review

Met requirements: (Aim complete) Allow the user to download the log in a .pdf format

Essential Feature Complete: Pilot navigation log

Success criteria item complete: Printable pilot Navigation log

Changes: N/A

Next steps: Adding extra details

Iteration 10 - Adding details to the PDF

base.html

```
<!--Navlog PDF template-->
<div id="navlogPDF_template">
  <div id="pdf_header">
    <!--Logo-->
    <div>
      <span id="logoAI">AI</span>
      <span id="logoVFR">VFR</span>
    </div>
    <h2>Navigation Log</h2>
  </div>

  <div id="pdf_content" style="padding: 20px 0;"></div>

  <div id="pdf_footer">
    <p><em>FOR EDUCATIONAL PURPOSES ONLY. NOT TO BE USED IN REAL WORLD FLIGHT OPERATIONS</em></p>
    <p><span id="pdf_dateAndTime"></span> | <b>www.aivfr.co.uk</b></p>
  </div>
</div>
```

basePage.css

```
/*-----Navlog pdf template-----*/

#navlogPDF_template {
  position: absolute;
  left: -9999px;
  width: 800px;
}

#pdf_header {
  display: flex;
  justify-content: space-between;
  border-bottom: 2px solid #333;
  padding-bottom: 10px;
}

#pdf_footer {
  text-align: center;
  font-size: 14px;
  margin-top: 50px;
  border-top: 1px solid #ccc;
}
```

navlog.js

```
async function downloadPDF() {  
  await fetch('/get-flight')  
  .then(response => response.json())  
  .then(async FlightData => {  
    //get the departure and arrival names to put in the file name  
    const departure_code = FlightData.departureAirport_code  
    const arrival_code = FlightData.destinationAirport_code  
  
    download.style.pointerEvents = 'none'  
    download.style.opacity = 0.4  
    await refreshPLOG();  
    //getting the table  
    const navlog = document.getElementById('navlogTable');  
  
    //getting the template  
    const template_container = document.getElementById('navlogPDF_template');  
    const navlog_content = document.getElementById('pdf_content');  
    const dateAndTime = document.getElementById('pdf_dateAndTime');  
  
    //inserting the data into the template  
    navlog_content.innerHTML = navlog.outerHTML;  
    dateAndTime.innerHTML = `Generated on ${new Date().toLocaleDateString()} at ${new Date().toLocaleTimeString()}`;  
    //using html2pdf to convert the html table into a image, then save to PDF  
    const options = {  
      margin: 0.2,  
      filename: `navlog ${departure_code}-${arrival_code}.pdf`,  
      image: { type: 'jpeg', quality: 0.98 },  
      html2canvas: { scale: 2, useCORS: true },  
      logging: false,  
      jsPDF: { unit: 'in', format: 'letter', orientation: 'landscape' }  
    };  
  
    html2pdf().set(options).from(template_container.innerHTML).save();  
  })  
}
```

What I did

For this final part I wanted to add some details to the PDF so that it looked a little more official and a lot less plain I wanted to add the AIVFR logo to the side and perhaps a title for the page, but the most important part was to include a warning that the details provided on the PDF were not to be used in real world flight.

The most traditional way to do this according to the internet, was the 'off screen clone' method. Since this involves having html elements (header and footer) with the content in the centre, All these things had to go somewhere without being visible on the screen of the user. So, after adding the HTML, the css lines `position: absolute; left: -9999px;` takes this div thousands of pixels from the left (basically out of the screen) whilst not interfering with the main viewport content at all.

Then in JS, the new elements are added, the table is refreshed to ensure that all data is saved and will be displayed, and then the table is slotted into the `pdf_content` div. Only then is the PDF generated, which looks like as you can see below.

Other changes include using the departure and arrival aerodrome's codes to form the name of the PDF file for easy identification, and also adding the date and time of when the pdf was generated.

Why I did it this way

Regarding the addition of these items, it was simply a design choice that I thought would make the document a little more appealing to the user. It was only whilst I was making it did I realise that I actually needed a footer to warn the user about the use of the data, but other than that, **not adding anything else would not have an effect on the data quality nor user experience.**

The alternative to using the off screen clone was to build the entire thing in javascript, which, honestly wouldn't be much different but I found the first method to be the cleanest as it allows me to have edits directly in HTML. This also allowed me to fix dark mode issues as dark mode caused styling issues for the PDF, so I needed to add classes that linked to files other than **navlog.html** to ensure the table always downloads using light mode colors.

Program output

 navlog EGSX-EGKA.pdf

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
108	New pdf style testing	The generated PDF should contain a header with a logo and title, with the appropriate footer containing the date, and a warning to the user.	 navlog EGSX-EGKA.pdf
109	Navlog tab dark mode test	Darkmode should apply the correct coloring to all elements.	 PASS*

*The main change was adding a new root color **--alternate: hsl(48.54, 83.18%, 29.02%)** and **--alternate: hsl(48.54, 83.18%, 79.02%)** to allow a suitably coloured alternate row in dark mode.

Iteration Review

Met requirements: Section 8 Complete

Changes: Styling changes, warning added for data usage

Next steps: Fuel Tab

Section Review

Aims completed in this section:

- **Complete 2 Essential features: Pilot navigation log, Basic flight calculations**
- **Complete a success criteria item:** Flight computer
- Provide the ability of the user to choose altitudes, speeds and other logistical aspects of their route as per design
- Allow the user to download the log in a .pdf format

Aims not completed in this section:

- Allow AI the ability to generate these aspects itself, providing a justification for its thought process.

Iterative tests complete:

- Light/dark Mode switches correctly with the correct colours

Remarks: Nothing to address.

Section 9: Fuel Tab

Aims of this section

- **Complete an essential feature: Fuel Calculations**
- **Partly complete a success criteria item:** [Fuel](#), Mass and balance, and Expense calculators
- Allow users to choose their fuel policy and manually enter specific times for each stage of flight.
- Allow these times to be filled by AI automatically based on the current fuel policy for that element of flight.
- Display the users total fuel required for each stage as well as for the whole flight, including the weight of said fuel using data provided by the user
- Determine whether a stopover along the route is required and notify the user of this.

Iteration 1 - HTML initial layout and styling

fuel.html

```
<!--Fuel Policy-->
<div id="fuel_policy" class="with_highlight">
  <div style="display: flex; align-items: center; justify-content: space-between;">
    <h2 style="margin-bottom: 15px;">Fuel Policy</h2>
    <span id="autofill">
      <button id="autofillButton" class="baseButton"><svg fill="currentColor" height="20" id="AI_icon" viewBox="0
    </span>
  </div>
  <p style="color: #737373">You can use the provided general policy or (preferably) edit it to your operator's p
  <p style="color: #737373">You can use Autofill to calculate time values based on the currently entered policy.
  <table>
    TABLE
  </table>
</div>

<!--Fuel burn, taxi time, specific gravity-->
<div id="fuel_info" class="with_highlight" style="display: flex; justify-content: center; gap: 20px; flex-wrap: wr
  <span class="input"><b>Fuel Burn: </b> <input id="fuelBurn" type="number"> <b>[unit]/hr</b></span>
  <span class="input"><b>Taxi Time: </b> <input id="taxiTime" type="number"> <b>mins</b></span>
  <span class="input"><b>Specific Gravity (SG): </b> <input id="specificGravity" type="number"></span>
</div>

<!--Fuel Required-->
<div id="fuel_required" class="with_highlight">
  <h2 style="margin-bottom: 20px;">Fuel Required</h2>
  <p style="margin-bottom: 20px;">Taxi Fuel: </p>
  <p style="margin-bottom: 20px;">Trip Fuel: </p>
  <p style="margin-bottom: 20px;">Contingency: </p>
  <p style="margin-bottom: 20px;">Alternate: </p>
  <p style="margin-bottom: 20px;">Final Reserve: </p>
  <p style="margin-bottom: 20px;">Additional: </p>
  <div class="input">Extra: <input id="extraFuel" type="number"> <b>[unit]</b></div>

  <hr size="3.5" color="#cdcdcd" noshade="" style="margin-top: 25px;">
  <!--Totals-->
  <span class="totals" style="margin-top: 20px;">Total: </span>
  <span class="totals" style="margin-top: 10px;">Weight: </span>
</div>

<!--Determining stopover requirement-->
<div id="stopovers">
  <div class="input"><b>Max Tank Capacity: </b> <input id="maxTankCapacity" type="number"> <b>[unit]</b></div>
```

fuel.css

```
#content {
  display: grid;
  grid-template-columns: 1fr 1fr;
}

#fuel_policy {
  grid-column: 1;
  grid-row: 1 / span 3;
}

#stopovers {
  grid-column: 2;
  grid-row: 3;
  border-radius: 10px;
  padding: 15px;
}
```



```

#fuel_required {
  grid-column: 2;
  grid-row: 2;
  padding: 20px;
}

#fuelBurn_and_taxiTime {
  grid-column: 2;
  grid-row: 1;
  min-height: fit-content;
}

.with_highlight {
  background-color: var(--highlight);
  border-radius: 10px;
  padding: 15px;
  margin: 5px;
}

input {
  border: 1px solid var(--text);
  border-radius: 5px;
  padding: 5px;
  margin-left: 10px;
  width: 100px;
  text-align: center;
  font-weight: 600;
  font-family: 'Manrope';
  font-size: 13px;
  background-color: var(--bg);
  border: 1px solid var(--onhover);
}

#stopoverRequired {
  display: flex;
  font-size: 17px;
  margin-top: 10px;
  padding: 3%;
  background-color: hsla(0 100% 60% / 0.7);
  border-radius: 15px;
  justify-content: center;
}

#stopoverNotRequired {
  display: flex;
  font-size: 17px;
  margin-top: 10px;
  padding: 3%;
  background-color: var(--onhover);
  border-radius: 15px;
}

```

```

        justify-content: center;
    }

    .totals {
        font-size: 18px;
        font-weight: 700;
        margin-right: 40%;
    }

    #autofillButton {
        text-align: center;
        font-size: 16px;
        font-weight: 500;
        font-family: 'Manrope';
        padding: 7px 10px 7px 10px;
        width: fit-content;
        border-radius: 8px;
        cursor: pointer;
        margin-right: 13px;
        background-color: var(--bg);
    }

    #autofillButton:hover {
        background-color: var(--primary);
    }

    #AI_icon {
        margin-right: 5px;
    }

```

What I did

Starting off with the general layout, I used CSS grid to structure each section so that they appear with dimensions similar to those in the design. All the divs except the last one with the 'Max tank capacity' value and stopover alert have the class `with_highlight` which allows all boxes to have a background except the last one as per design. You can see both stopover alert boxes present in the preview as there is no JS written, and they will appear/dissapear accordingly depending on the total fuel value compared with the max tank capacity. For the units, a placeholder [unit] is put in its place for now because I am yet to code in the actual variable units.

I've made a few changes: Weight and thus specific gravity (used to calculate weight) have been added. Also, the AUTOFILL button has been moved to the route policy table. As the new tooltip suggests, each description for each part of the fuel policy can be editable, then, to integrate AI into this, the AI will read these descriptions and choose a suitable time period based on this. This time will then be used to calculate the fuel required. The AI will have some route data as well, such as the route time to aid in calculating the trip time. Also, extra fuel is not going to be added on the table but instead in the fuel resources, as this is *more intuitive to the user as all values would be calculated and so they would have things to base their decisions off of, rather than having to make their own inferences.*

Why I did it this way

Initially, I included the weight so that the user can see it before they get into the mass and balance page- this is useful as often pilots may have predefined weight values that they're

trying to meet whilst planning. However, I quickly realised that it was a necessity, because it stores the current weight in the session early, so that I can easily access it in the mass and balance tab. Without this I would have to calculate the weight in the mass and balance tab which may not be intuitive as it would essentially always have to be done on load, which is longer than just reading what is already loaded into the session. The application of AI here is better as less mistakes are likely to occur here, as it doesn't need to use map data or route data, just some considerate reasoning skills and common sense. This also allows the user to have quick suggestions which then lay a foundation for the final values they choose to pick.

Program output

FUEL

Fuel Policy

AUTOFILL

You can use the provided general policy or (preferably) edit it to your operator's policy.

You can use Autofill to calculate time values based on the currently entered policy.

TABLE

Fuel Burn: [unit]/hr

Taxi Time: [mins]

Specific Gravity (SG): [unit]

Fuel Required

Taxi Fuel:

Trip Fuel:

Contingency:

Alternate:

Final Reserve:

Additional:

Extra: [unit]

Total:

Weight:

Max Tank Capacity: [unit]

Excess Fuel: Stop-overs are required for this route.

No stop-overs are required for this route.

Known Limitations / Missing features

- No table

Test No.	Test Description	Expected Outcome	Actual Outcome
110	HTML and CSS general layout test	general layout should appear as per design, along with the included changes	FUEL Fuel Policy AUTOFILL You can use the provided general policy or (preferably) edit it to your operator's policy. You can use Autofill to calculate time values based on the currently entered policy. TABLE Fuel Burn: [unit]/hr Taxi Time: [mins] Specific Gravity (SG): [unit] Fuel Required Taxi Fuel: Trip Fuel: Contingency: Alternate: Final Reserve: Additional: Extra: [unit] Total: Weight: Max Tank Capacity: [unit] Excess Fuel: Stop-overs are required for this route. No stop-overs are required for this route.

Iteration Review

Met requirements: N/A

Changes: Added AI button, weight and SG

Next steps: Adding fuel policy table

Iteration 2 - Adding fuel policy table

fuel.html

```
<!--Fuel Policy Table-->
<table id="fuel_policy_table" style="width: 100%;">
  <tr>
    <th>Element</th>
    <th>Policy</th>
    <th>Time Allowed</th>
  </tr>
  <tr>
    <td>Trip</td>
    <td><textarea name="tripPolicy" id="tripPolicy" value="">Take-off to Landing. Should consider steep cl
    <td><input id="TA_trip" type="number"></td>
  </tr>
  <tr>
    <td>Contingency</td>
    <td><textarea name="contingencyPolicy" id="contingencyPolicy" value="">Either 5% of Trip, 3% if Enrout
    <td><input id="TA_contingency" type="number"></td>
  </tr>
  <tr>
    <td>Alternate</td>
    <td><textarea name="alternatePolicy" id="alternatePolicy" value="">For Missed approaches / landing at
    <td><input id="TA_alterate" type="number"></td>
  </tr>
  <tr>
    <td>Final Reserve</td>
    <td><textarea name="finalReservePolicy" id="finalReservePolicy" value="">Enough for 45 min Flight time
    <td><input id="TA_finalReserve" type="number"></td>
  </tr>
  <tr>
    <td>Additional</td>
    <td><textarea name="additionalPolicy" id="additionalPolicy" value="">Enough for 15 mins holding time</
    <td><input id="TA_additional" type="number"></td>
  </tr>
</table>
```

fuel.css

```
/*Policy table*/
table {
  width: 100%;
  border-collapse: collapse;
  margin-top: 10px;
}

th, td {
  border: 1px solid var(--text);
  padding: 8px;
  text-align: center;
  font-family: 'Manrope';
  font-size: 14px;
}
```

```

textarea {
  width: 100%;
  height: 100px;
  border: none;
  border-radius: 5px;
  padding: 10px;
  font-family: 'Manrope';
  font-size: 14px;
  color: var(--text);
  resize: none;
  background-color: var(--highlight);
}

#policyTooltipText {
  display: none;
  position: absolute;
  background-color: var(--highlight);
  color: var(--text);
  padding: 10px;
  border-radius: 5px;
  width: 300px;
  font-size: 13px;
  box-shadow: 0 0 10px rgba(0, 0, 0, 0.2);
  z-index: 1000; /*stays on top of everything*/
  top: 110px;
  left: 80px;
}

```

fuel.js

```

import { update, updateSettings, showAlert } from "../basePage.js";
updateSettings("current_page", "/fuel");

//fuel policy tooltip
const policyTooltip = document.getElementById("policyTooltipText");
const policyInfoIcon = document.getElementById("FuelPolicyTooltip");

policyInfoIcon.addEventListener("mouseover", () => {
  policyTooltip.style.display = "block";
});

policyInfoIcon.addEventListener("mouseout", () => {
  policyTooltip.style.display = "none";
});

```

What I did

The table consists of three columns, the element of flight, the actual policy and the time allowed for that period, which determines the fuel to be used for that time. All policy cells have been placed in a `<textarea>` tag so that they are editable, but for the time being I've hardcoded the general fuel policy as a placeholder text. In actual practice, this policy will

be retrieved from the session. To give some structure the textareas have a defined height, and will roll into a scroll if it gets too large, which it shouldn't. Then the time allowed section is simply a number input.

The fuel policy tooltip has also moved into an info icon like the route tab, however i've done it so that the tip shows up like an actual onhover tooltip, as the aim was to remove the tooltip text from the main page.

Why I did it this way

I hid the tooltip principally because it took too much space and was making the table-length in the grid unequal, which wasn't visually appealing. This alternative allowed the message to still be quickly accessible to the user and perhaps looks more professional and intuitive too, as this is a common thing in many applications as well.

Program output

 Fuel policy tooltip test.mp4

Fuel Policy ⓘ

✦ AUTOFILL

Element	Policy	Time Allowed
Trip	Take-off to Landing. Should consider steep climbs, descents and arrival procedures.	
Contingency	Either 5% of Trip, 3% if Enroute alternative aerodrome available, or Whatever is sufficient for 20min of flight. Whichever higher.	
Alternate	For Missed approaches / landing at alternative aerodromes. Should include time for a circuit at destination.	
Final Reserve	Enough for 45 min Flight time	
Additional	Enough for 15 mins holding time	

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome																		
111	Table layout test	Tables should appear as per design, with place holder descriptions with correct formatting.	Fuel Policy ⓘ AUTOFILL <table><thead><tr><th>Element</th><th>Policy</th><th>Time Allowed</th></tr></thead><tbody><tr><td>Trip</td><td>Take-off to Landing. Should consider steep climbs, descents and arrival procedures.</td><td> </td></tr><tr><td>Contingency</td><td>Either 5% of Trip, 3% if Enroute alternative aerodrome available, or Whatever is sufficient for 20min of flight. Whichever higher.</td><td> </td></tr><tr><td>Alternate</td><td>For Missed approaches / landing at alternative aerodromes. Should include time for a circuit at destination.</td><td> </td></tr><tr><td>Final Reserve</td><td>Enough for 45 min Flight time</td><td> </td></tr><tr><td>Additional</td><td>Enough for 15 mins holding time</td><td> </td></tr></tbody></table>	Element	Policy	Time Allowed	Trip	Take-off to Landing. Should consider steep climbs, descents and arrival procedures.		Contingency	Either 5% of Trip, 3% if Enroute alternative aerodrome available, or Whatever is sufficient for 20min of flight. Whichever higher.		Alternate	For Missed approaches / landing at alternative aerodromes. Should include time for a circuit at destination.		Final Reserve	Enough for 45 min Flight time		Additional	Enough for 15 mins holding time	
Element	Policy	Time Allowed																			
Trip	Take-off to Landing. Should consider steep climbs, descents and arrival procedures.																				
Contingency	Either 5% of Trip, 3% if Enroute alternative aerodrome available, or Whatever is sufficient for 20min of flight. Whichever higher.																				
Alternate	For Missed approaches / landing at alternative aerodromes. Should include time for a circuit at destination.																				
Final Reserve	Enough for 45 min Flight time																				
Additional	Enough for 15 mins holding time																				
112	Tooltip hover test	Upon hovering over the info icon, a tooltip should appear until the mouse is no longer hovering over the icon	Fuel policy tooltip test.mp4																		

Iteration Review

Met requirements: N/A

Changes: Fuel policy tooltip now an hover tooltip attached to an info icon near subheading

Next steps: Adding new keys for values and units

Iteration 3 - Adding keys for values & Units

app.py

```
#-----FUEL
#FUEL POLICY - general policy as default
"fuelPolicy_trip" : {
    "policy" : "Take-off to Landing. Should consider steep climbs, descents and arrival procedure",
    "time_allowed" : "",
},
"fuelPolicy_contingency" : {
    "policy" : "Either 5% of Trip, 3% if Enroute alternative aerodrome available, or Whatever is left",
    "time_allowed" : "",
},
"fuelPolicy_alterate" : {
    "policy" : "For Missed approaches / landing at alternative aerodromes. Should include time to land",
    "time_allowed" : "",
},
"fuelPolicy_finalReserve" : {
    "policy" : "Enough for 45 min Flight time",
    "time_allowed" : "",
},
"fuelPolicy_additional" : {
    "policy" : "Enough for 15 mins holding time",
    "time_allowed" : "",
},
#FUEL VALUES - NON EDITABLE#
"fuel_total" : {"value" : 0, "class" : "fuel"},
"fuel_taxi" : {"value" : 0, "class" : "fuel"},
"fuel_trip" : {"value" : 0, "class" : "fuel"},
"fuel_contingency" : {"value" : 0, "class" : "fuel"},
"fuel_alterate" : {"value" : 0, "class" : "fuel"},
"fuel_finalReserve" : {"value" : 0, "class" : "fuel"},
"fuel_additional" : {"value" : 0, "class" : "fuel"},
"fuel_endurance" : "",
"fuel_mass" : {"value" : 0, "class" : "mass"},
#EDITABLE PARAMETERS#
"fuel_extra" : {"value" : 0, "class" : "fuel"},
"fuel_burn" : {"value" : 0, "class" : "fuel"},
"specific_gravity" : 0,
"max_tank_capacity" : {"value" : 0, "class" : "fuel"},
"taxi_time" : "",
```

fuel.html

```
<!--Fuel Policy Table-->
<table id="fuel_policy_table" style="width: 100%;">
  <tr>
    <th>Element</th>
    <th>Policy</th>
    <th>Time Allowed (mins)</th>
  </tr>
  <tr>
    <td>Trip</td>
    <td><textarea name="tripPolicy" id="tripPolicy" value="">{{ flight.fuelPolicy_trip.policy }}</textarea>
    <td><input id="TA_trip" type="number" value="{{ flight.fuelPolicy_trip.time_allowed }}"></td>
  </tr>
  <tr>
    <td>Contingency</td>
    <td><textarea name="contingencyPolicy" id="contingencyPolicy" value="">{{ flight.fuelPolicy_contingency.policy }}</textarea>
    <td><input id="TA_contingency" type="number" value="{{ flight.fuelPolicy_contingency.time_allowed }}"></td>
  </tr>
  <tr>
    <td>Alternate</td>
    <td><textarea name="alternatePolicy" id="alternatePolicy" value="">{{ flight.fuelPolicy_alterate.policy }}</textarea>
    <td><input id="TA_alterate" type="number" value="{{ flight.fuelPolicy_alterate.time_allowed }}"></td>
  </tr>
  <tr>
    <td>Final Reserve</td>
    <td><textarea name="finalReservePolicy" id="finalReservePolicy" value="">{{ flight.fuelPolicy_finalReserve.policy }}</textarea>
    <td><input id="TA_finalReserve" type="number" value="{{ flight.fuelPolicy_finalReserve.time_allowed }}"></td>
  </tr>
  <tr>
    <td>Additional</td>
    <td><textarea name="additionalPolicy" id="additionalPolicy" value="">{{ flight.fuelPolicy_additional.policy }}</textarea>
    <td><input id="TA_additional" type="number" value="{{ flight.fuelPolicy_additional.time_allowed }}"></td>
  </tr>
</table>
```



```

<!--Fuel Required-->
<div id="fuel_required" class="with_highlight">
  <h2 style="margin-bottom: 20px;">Fuel Required</h2>
  <p style="margin-bottom: 20px;">Taxi Fuel: <b>{{ flight.fuel_taxi.output }} </b></p>
  <p style="margin-bottom: 20px;">Trip Fuel: <b>{{ flight.fuel_trip.output }}</b></p>
  <p style="margin-bottom: 20px;">Contingency: <b>{{ flight.fuel_contingency.output }}</b></p>
  <p style="margin-bottom: 20px;">Alternate: <b>{{ flight.fuel_alternate.output }}</b></p>
  <p style="margin-bottom: 20px;">Final Reserve: <b>{{ flight.fuel_finalReserve.output }}</b></p>
  <p style="margin-bottom: 20px;">Additional: <b>{{ flight.fuel_additional.output }}</b></p>
  <div class="input">Extra (PIC discretion): <input id="extraFuel" type="number" value="{{ flight.fuel_extra.val

  <hr size="3.5" color="#cdcdcd" noshade="" style="margin-top: 25px;">
<!--Totals-->
<span class="totals" style="margin-top: 20px;">Total: {{ flight.fuel_total.output }}</span>
<span class="totals" style="margin-top: 10px;">Endurance: {{ flight.fuel_endurance.output }}</span>
<span class="totals" style="margin-top: 10px;">Mass: {{ flight.fuel_mass.output }}</span>
</div>

```

What I did

First I added the new keys to the session template. For the fuel policy values, given that each row has two storable values attached to it (Policy and Time Allowed), the values are stored within their respective objects, each with their two values. For most of the fuel values, the same format for unit-changing values is used (i.e. with 'value' and 'class'), except taxi time and fuel endurance, whose units don't need to be changed.

For values that are unit-changing but are also in an input box, the **.value** is targeted instead of **.output**. Remember, when the units are converted initially the 'output' value is appended to each object, which contains the string to display e.g. '100 Gal'. The **value** just contains the number itself: 100 in this case, and is what is used to make the string with the unit in the end. The value/output differences are explained in more detail in the iterations that implemented the unit conversions. Nevertheless, by targeting the value only, only the digits appear without the units, which would be necessary as the input box's automatic validation takes numbers only.

The rest of the work is over in **fuel.html**, where I replaced all the placeholder text/inputs with the relevant Jinja2 references to allow them to display data stored in the session, as per usual.

Why I did it this way

Using separate objects for each table row makes organising the table values much easier because I can have keys with the same name ('policy', 'time allowed'), and thus it makes referencing them an easier job too. It also makes the data easier to see, interpret and understand from a developer standpoint. If I just incorporated it as several keys, they would be hard to distinguish between and key names would be unnecessarily long, which inevitably makes the developing experience much more difficult.

Program output

 Fuel units changing test 1.mp4

Fuel Policy ⓘ

AUTOFILL

Element	Policy	Time Allowed (mins)
Trip	Take-off to Landing. Should consider steep climbs, descents and arrival procedures.	
Contingency	Either 5% of Trip, 3% if Enroute alternative aerodrome available, or Whatever is sufficient for 20min of flight. Whichever higher.	
Alternate	For Missed approaches / landing at alternative aerodromes. Should include time for a circuit at destination.	
Final Reserve	Enough for 45 min Flight time	
Additional	Enough for 15 mins holding time	

Fuel Burn: 0.0 gph
Specific Gravity (SG): 0

Taxi Time: mins
Max Tank Capacity: 0.0 Gal

Fuel Required

Taxi Fuel: 0.0 Gal

Trip Fuel: 0.0 Gal

Contingency: 0.0 Gal

Alternate: 0.0 Gal

Final Reserve: 0.0 Gal

Additional: 0.0 Gal

Extra (PIC discretion): 0.0 Gal

Total: 0.0 Gal
Endurance:

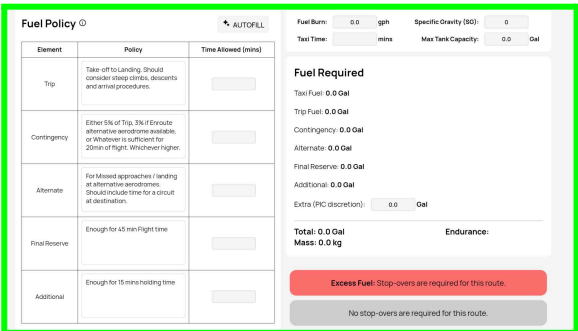
Mass: 0.0 kg

Excess Fuel: Stop-overs are required for this route.

No stop-overs are required for this route.

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
113	Session load and display testing	The values that can change have their data fetched from the session and can be displayed properly.	
114	Units changing test	The units of any unit changing items change when the units are switched from the settings menu.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: handling all inputs and calculating fuel values.

Iteration 4 - Input handling and Calculations

```
//-----CALCULATING TOTALS
async function calculateTotals() {
  const response = await fetch("/calculate-totals", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    }, //send extra fuel value to backend to avoid desync
    body: JSON.stringify({ extra: extraFuel.value }),
  });
  if (!response.ok) {
    showAlert("Error calculating totals. Please try again.", "error");
    return;
  }
  //fetch updated totals from backend
  const data = await response.json();

  //updating totals
  totalFuel.innerText = `Total: ${data.total_fuel}`;
  fuelMass.innerText = `Mass: ${data.fuel_mass}`;
  fuelEndurance.innerText = `Endurance: ${data.fuel_endurance}`;
}
```

app.py

```
#-----FUEL-----#

#-----calculating totals
@main.route('/calculate-totals', methods=["POST"])
def calculate_totals():
    flight_data = session["flight_data"]["flight"]

    #---TOTAL FUEL---#
    #getting all fuel components
    taxi = flight_data["fuel_taxi"]["value"]
    trip = flight_data["fuel_trip"]["value"]
    contingency = flight_data["fuel_contingency"]["value"]
    alternate = flight_data["fuel_alternate"]["value"]
    finalReserve = flight_data["fuel_finalReserve"]["value"]
    additional = flight_data["fuel_additional"]["value"]
    extra = request.json.get("extra")

    #calculating total fuel
    total_fuel = sum([
        float(taxi),
        float(trip),
        float(contingency),
        float(alternate),
        float(finalReserve),
        float(additional),
        float(extra)
    ])
```

```

])

#updating session
flight_data["fuel_total"]["value"] = round(total_fuel, 1)

#---TOTAL MASS---#
specific_gravity = flight_data["specific_gravity"]
if specific_gravity != "" and specific_gravity != 0:
    #if fuel is in litres, mass is fuel * SG (kg/litre)
    if session["flight_data"]["settings"]["units_fuel"] == "litre":
        fuel_mass = total_fuel * float(specific_gravity)

    #if fuel is in gallons, mass is fuel * SG * 8.345 (lbs/gallon)
    elif session["flight_data"]["settings"]["units_fuel"] == "US_liquid_gallon":
        fuel_mass = total_fuel * float(specific_gravity) * 8.345
    flight_data["fuel_mass"]["value"] = round(fuel_mass, 1)
else:
    flight_data["fuel_mass"]["value"] = 0

#---ENDURANCE--- to calculate how long the fuel will last based on burn rate
fuel_burn = flight_data["fuel_burn"]["value"] #in units of fuel per hour
if fuel_burn != "" and fuel_burn != 0:
    endurance_hours = total_fuel / float(fuel_burn)
    hours = int(endurance_hours)
    minutes = int((endurance_hours - hours) * 60)
    endurance = f"{hours}hrs {minutes}mins"
    flight_data["fuel_endurance"] = endurance

update_unitsRUN() #reapply units after totals have been calculated
session.modified = True
return jsonify({"total_fuel": flight_data["fuel_total"]["output"],
               "fuel_mass": flight_data["fuel_mass"]["output"],
               "fuel_endurance": flight_data["fuel_endurance"]
               }), 200

```

fuel.js

```
//-----FUEL INPUT HANDLING
//user-editable fuel info
const fuelBurn = document.getElementById("fuelBurn");
const taxiTime = document.getElementById("taxiTime");
const maxTankCapacity = document.getElementById("maxTankCapacity");
const specificGravity = document.getElementById("specificGravity");
const extraFuel = document.getElementById("extraFuel");

//fuel required elements
const elements = document.querySelectorAll("#fuel_required .elements");

//totals
const totalFuel = document.getElementById("totalFuel");
const fuelMass = document.getElementById("fuelMass");
const fuelEndurance = document.getElementById("fuelEndurance");

//wait for a change in fuel inputs, then update
fuelBurn.addEventListener("change", async () => {
  await update("fuel_burn.value", Number(fuelBurn.value));
  calculateTotals();
});

taxiTime.addEventListener("change", async () => {
  await update("taxi_time", Number(taxiTime.value));
  calculateTotals();
});

maxTankCapacity.addEventListener("change", async () => {
  await update("max_tank_capacity.value", Number(maxTankCapacity.value));
}); //no calc needed - max capacity doesnt affect totals

specificGravity.addEventListener("change", async () => {
  await update("specific_gravity", Number(specificGravity.value));
  calculateTotals();
});

extraFuel.addEventListener("change", async () => {
  await update("fuel_extra.value", Number(extraFuel.value));
});

//waits for a change in any fuel required element, then calculates all totals
//as one necessitates the other
elements.forEach(element => {
  element.addEventListener("change", async () => {
    calculateTotals();
  });
});
```

What I did

Calculating totals

To calculate all the necessary total values, we need to get all the fuel required values, and sum them to get the total fuel, then use the user's SG input plus the calculated fuel to calculate mass, and endurance can be calculated using the fuel burn input. In **app.py**, to calculate the sum, the necessary data is obtained from the session.

Then, to calculate the total fuel they're all added together, and the session updated. To calculate the mass, I take the specific gravity input by the user, verify its not empty, then use $mass(m) = Volume(V) \times density(\rho)$ to calculate the mass. Note that the density is numerically equivalent to the specific gravity. If the fuel volume is in gallons, it's necessary to multiply by 8.34, as the reference is **water = 8.34 lb/gal** rather than **1kg/l** for a volume in litres. To determine the correct calculation I simply take the current fuel unit from the session and run the correct calculation depending on what unit is currently selected. The session is then updated again with the calculated mass.

Finally, to calculate the endurance, it's simply utilising the fuel burn to find out how long the fuel will actually last approximately in the air. The hours are calculated by dividing the total fuel by the fuel burn, and then the minutes by subtracting this by the full (decimal) hours and multiplying by 60 to get the minutes. They're then put into a string and stored in the session. Note that in the end both hours and minutes get set to integers (and not floats) to ensure there aren't any decimals in the time display.

All this data is then set straight back to the frontend, where the `calculateTotals()` function continues to get the response from the backend and replaces the text in the DOM to show the calculated values.

Input handling

For each fuel information input, after the user clicks off the input box (i.e. a change) then the session updates with the entered values. `calculateTotals()` then runs to apply any new changes to the totals.

Why I did it this way

The 'extra' fuel is passed in the `calculateTotals` function to `app.py` instead of it simply being fetched from the session. This was because of a desync that occurred meaning that the extra fuel value read by python was always one value behind the actual, which offset the calculations. Instead, using the value that was just inputted directly into the DOM serves better as it will always match the current value, and any desync would be stopped.

Program output

 Totals calculation and unit conv test.mp4

Known Limitations / Missing features

- Time allowed values for fuel required not yet usable

Test No.	Test Description	Expected Outcome	Actual Outcome
115	Totals calculation test	Upon adjusting the extra fuel value, the total should be added to the total fuel, and upon entering necessary fuel information weight and endurance should be calculated too.	PASS (See program output)

116	Unit conversion test	Upon changing the units the units of all values should change to produce accurate values.	ERROR (See program output)
-----	----------------------	---	-----------------------------------

Error Analysis & Fixes

Test No: 116

Failure Description: Unit conversions are inaccurate - fuel volume doesn't convert at all

Cause: Not sure for certain, most likely a problem with the unit conversion algorithm in app.py

Fix: Run an isolated unit conversion test for fuel and other values to see if the error exists solely with the fuel values or others too. From this I should be able to determine if it's an issue with the conversion function or not.

Justification: If I simply targeted the fuel values only, I wouldn't be able to tell if the problem affects more units and may implement a change that may break other unit conversions, or even the entire algorithm. Running an isolated test should allow me to see what is going on exactly and should allow me to see exactly where the problem stems from and what values it is affecting to better find a suitable solution.

Iteration 4.5 - Fixing unit conversion inaccuracies

app.py

```
#-----MASS AND BALANCE

#ISOLATED UNIT CONVERSION TEST
'unit_test_fuel' : {"value" : 124, "class" : "fuel"},
'unit_test_mass' : {"value" : 1000, "class" : "mass"},
'unit_test_distance' : {"value" : 80, "class" : "distance"},
'unit_test_altitude' : {"value" : 2500, "class" : "altitude"},
'unit_test_airspeed' : {"value" : 90, "class" : "airspeed"}
```

mass-and-balance.html

```
<div class="pageTitle" style="display: flex; align-items: center; gap: 10px;">
  <div id="title">MASS & BALANCE</div>
  <span id="alertBox"></span>
</div>
<h3>Isolated unit conversion test</h3>
<p>Fuel value: {{ flight.unit_test_fuel.output }}</p>
<p>Mass value: {{ flight.unit_test_mass.output }}</p>
<p>Distance value: {{ flight.unit_test_distance.output }}</p>
<p>Altitude value: {{ flight.unit_test_altitude.output }}</p>
<p>Airspeed value: {{ flight.unit_test_airspeed.output }}</p>
<% endblock %>
```


The test is set in a free tab - mass and balance. I set pre-defined values for each of them, and intend to change them each in the test, and see which ones do and which ones don't.

Program output (live test)

 isolated unit conversion test.mp4

After looking back over the process of the unit's conversion section, the error became obvious.

This line:

app.py

```
#Update previous unit to current
session["flight_data"]["settings"][prev_unit_key] = display_unit
```

Updates the previous unit to the current unit when the conversion is done. This is so that once a conversion happens again, the previous unit should not equal the new unit that was selected before the newER conversion was made. This meant that, for the first loop, the conversion would work, but then the previous unit gets set to the current unit, so in the next iteration instead of converting prev -> current it's converting current -> current: i.e the same thing.

To fix this, I took it out of the for loop, and iterated through all the keys again once the initial for loop was done, this time, just to change the previous units to the current units. This works because this process will only happen once all conversions have been made. The issue was resolved!

```
#Update previous unit in session settings
for key, item in session["flight_data"]["flight"].items():
    #only process items that are dictionaries with a "class" key (i.e. unit-based items)
    if isinstance(item, dict) and "class" in item:
        category = item["class"]
        if category in UNIT_MAP:
            settings_key = UNIT_MAP[category]["settings_key"]
            display_unit = session["flight_data"]["settings"][settings_key]
            current_unit = ureg[display_unit]

            #Store previous unit in session settings if not present
            prev_unit_key = f"prev_{settings_key}"
            session["flight_data"]["settings"][prev_unit_key] = display_unit
```

Test No.	Test Description	Expected Outcome	Actual Outcome
117	Unit conversion test	Upon changing the units the units of all values should change to produce accurate values.	PASS

Iteration Review

Met requirements: (Aim complete) Display the users total fuel required for each stage as well as for the whole flight, including the weight of said fuel using data provided by the user

Changes: Fixed unit conversion bug

Next steps: Fuel policy calculations and stopovers

Iteration 5 - Fuel policy calculations & stopovers

fuel.js (just showing first 2 categories but its the same for all the others)

```
//-----FUEL POLICY INPUT HANDLING
const tripPolicy = document.getElementById("tripPolicy");
const TA_trip = document.getElementById("TA_trip");

const contingencyPolicy = document.getElementById("contingencyPolicy");
const TA_contingency = document.getElementById("TA_contingency");

//-----updating fuel policy selections

tripPolicy.addEventListener("change", async () => {
  await update("fuelPolicy_trip.policy", tripPolicy.value);
});

contingencyPolicy.addEventListener("change", async () => {
  await update("fuelPolicy_contingency.policy", contingencyPolicy.value);
});

//-----updating time allowed inputs and calculating fuel required
TA_trip.addEventListener("change", async () => {
  await update("fuelPolicy_trip.time_allowed", Number(TA_trip.value));
  await calculateFuelRequired(TA_trip.value, "trip");
  await calculateTotals();
});

TA_contingency.addEventListener("change", async () => {
  await update("fuelPolicy_contingency.time_allowed", Number(TA_contingency.value));
  await calculateFuelRequired(TA_contingency.value, "contingency");
  await calculateTotals();
});
```

```
//-----FUEL REQUIRED CALCULATIONS
async function calculateFuelRequired(timeAllowed='None', elementType='None') {
  //if timeallowed and element type arent provided, its a call to calculate all fuel required
  if (timeAllowed == 'None' && elementType == 'None') {
    await calculateFuelRequired(taxiTime.value, "taxi");
    await calculateFuelRequired(TA_trip.value, "trip");
    await calculateFuelRequired(TA_contingency.value, "contingency");
    await calculateFuelRequired(TA_alternate.value, "alternate");
    await calculateFuelRequired(TA_finalReserve.value, "finalReserve");
    await calculateFuelRequired(TA_additional.value, "additional");
    await calculateTotals();
    return;
  }

  //get values from inputs
  const burn = Number(fuelBurn.value);
  const element = document.getElementById(`fuel_${elementType}`);

  //calculate fuel required
  if (fuelBurn.value != "" && timeAllowed != "") {
    const fuel_required = (Number(timeAllowed) / 60) * burn

    //update session data and units
    await update(`fuel_${elementType}.value`, fuel_required);
    await fetch('/units-update')

    //get updated value from backend
    fetch('/get-flight')
      .then(response => response.json())
      .then(async data => {
        //update the element with the new value from backend
        const updatedValue = data[`fuel_${elementType}`].output;
        element.innerText = updatedValue;
      });
  } else {return}
}
}
```

app.py

```
#----STOPOVER CHECK----#
max_tank_capacity = flight_data["max_tank_capacity"]["value"]
if max_tank_capacity != "" and max_tank_capacity != 0:
    #check if total fuel exceeds max tank capacity
    if total_fuel > float(max_tank_capacity):
        stopover_status = True #needed
    else:
        stopover_status = False #not needed
else:
    stopover_status = False #cannot determine
```

calculateTotals() - fuel.js

```
//handling stopover alert
const stopoverRequired = document.getElementById("stopoverRequired");
const stopoverNotRequired = document.getElementById("stopoverNotRequired");

if (data.stopover_status == true) {
  stopoverRequired.style.display = "block";
  stopoverNotRequired.style.display = "none";
} else {
  stopoverRequired.style.display = "none";
  stopoverNotRequired.style.display = "block";
}
```

What I did

First, I made the fuel policy descriptions editable, by adding their variables and updating the session once a change is detected as per usual. The same also applied with the time allowed value. Now the session should save any edits made to the session from those fields. Now that we can utilise the time allowed values, I can calculate the fuel required values for each section. The `calculateFuelRequired()` function works by taking the time entered into the time allowed box, converting it to hours and then multiplying it by the fuel burn. Then the second parameter, the `elementType`, contains a string of the particular fuel element being edited- say it be 'contingency' or 'finalReserve'. Once the fuel calculation has been completed, `update(`fuel ${elementType}.value`, fuel_required);` will update that particular element's fuel required value, and update the units afterwards to account for the change. Then, once the update value has changed it's fetched again from the session and the text is replaced to show the fuel required value on the screen.

When the fuel burn value changes, all the fuel required calculations need to be redone. To deal with this, I made the function recursive so that if no parameters are given, it runs all the normal fuel calculations one after the other.

The stopover requirement is decided in the `calculateTotals()` function. It compares the total fuel with the max tank capacity and returns true or false depending on whether the fuel exceeds the capacity or not. This result is also sent to the frontend. Then at the end of the `calculateTotals` function the styling of the relevant box is edited according to the boolean provided by `app.py`.

Why I did it this way

Making the function recursive was the better alternative to simply just having the function default to calculating all fuel required values every single time just one is changed. This puts less stress on the server, as only all edits are made if all edits need to be made. Where this is not the case then just the relevant fuel required value is calculated. This also makes it more suited for the AI implementation too, because once all the time values are updated I can then simply just use this method to update all values at once and not one by one.

Program output

 Fuel policy functionality test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
118	Fuel policy description editing test	When a fuel policy is edited, the edit should be saved in session.	PASS (See program output)
119	Fuel required calculation tests	When the time allowed value is entered and fuel burn is present, the relevant fuel required value is calculated and the totals redone. If the fuel burn changes, all fuel required values should update too.	PASS (See program output)
120	Stopover appearance test	If the max capacity is less than the total fuel required then a box should replace the one present stating a stopover will be required for the route.	PASS (See program output)

Iteration Review

Met requirements: Essential feature complete: Fuel Calculations

Success criteria item complete: Fuel, Mass and balance, and Expense calculators

(Aim complete) Allow users to choose their fuel policy and manually enter specific times for each stage of flight.

(Aim complete) Determine whether a stopover along the route is required and notify the user of this.

Changes: N/A

Next steps: Avoiding any conflicts in units

Iteration 6 - Fixing unit conflicts

settings.html

```

<!--Units-->
<h2>UNITS</h2>
<div id="baseUnits">
  <button id="imperial" style="margin-right: 7px;" class="{ 'active' if settings.base_uni
  <span><button id="metric" class="{ 'active' if settings.base_units == 'metric' else ''
</div>

```

settings.js

```
h3 {
  font-weight: 400;
  margin-bottom: 10px;
  color: var(--text);
  margin-top: 20px;
}

#advancedDropdown {
  position: relative;
  right: 4px;
  top: 5px;
  cursor: pointer;
  color: var(--text);
  transition: transform 0.3s ease; /*for a smooth rotation*/
}
```

settings.js

```
//-----BASE UNITS TOGGLE-----
const imperial = document.getElementById("imperial")
const metric = document.getElementById("metric")
const advancedDropdown = document.getElementById("advancedDropdown");
const unitsSection = document.getElementById("UnitsButtons");

//toggle advanced settings (user can choose each unit individually)
advancedDropdown.addEventListener("click", () => {
  if (unitsSection.style.display === "none" || unitsSection.style.display === "") {
    updateSettings("base_units", "advanced"); //for identification in fuel tab
    imperial.classList.remove('active');
    metric.classList.remove('active');
    unitsSection.style.display = "flex";
    advancedDropdown.style.transform = "rotate(180deg)"; //dropdown arrow points up
  } else {
    unitsSection.style.display = "none";
    advancedDropdown.style.transform = "rotate(0deg)"; //dropdown arrow points down
  }
});

//imperial units
imperial.addEventListener("click", async () => {
  imperial.classList.add('active');
  metric.classList.remove('active');
  await updateSettings("base_units", "imperial");
  await massToPounds();
  await fuelToGallons();
  await altitudeToFeet();
  await airspeedToKnots();
  await distanceToNauticalMiles();
});

//metric units
metric.addEventListener("click", async () => {
  metric.classList.add('active');
  imperial.classList.remove('active');
  await updateSettings("base_units", "metric");
  await massToKilograms();
  await fuelToLitres();
  await altitudeToMetres();
  await airspeedToKnots(); //airspeed remains the same due to standard units
  await distanceToNauticalMiles(); //distance remains the same due to standad units
});
```

fuel.js

```
//-----UNITS CHECK-----  
//units here must match. if the vol is l, mass must be kg.  
//if vol is US liquid gallon, mass must be lbs  
fetch('/get-all')  
  .then(response => response.json())  
  .then(async data => {  
    const base_units = data.settings.base_units;  
  
    if (base_units === "custom") {  
      showAlert("Please ensure that mass and volume units match when using customised units.");  
    }  
  });
```

What I did

Whilst finishing off the unit conversion error, I noticed another significant error that I decided to combat in its own iteration. The user has the ability to change the units to units that conflict with each other. This essentially means having fuel volume set in Gallons but mass set in Litres (instead of pounds) and vice versa. This leads to incorrect fuel calculations due to complications involving specific gravity.

After many attempts to find an efficient solution, I came up with this: To include an imperial/metric switch system that automatically changes the necessary units as required. If the user wants to fine tune this the normal unit selection will be available, but as 'custom' units. To do this, I first implemented the basic switch toggle mechanism for imperial/metric (similar to satellite/normal map style toggle switches). I then added the 'Custom' text with a dropdown icon, and hid the manual unit's selection by default.

To deal with the imperial/metric switch, I had to make each unit changing function into an actual function, then add the eventListeners back to run that function. This was so I could actually call each of them easily when converting units. Each unit uses a simple format: '*categoryToUnit*', to make it absolutely clear what unit is becoming what.

A new key `"base_units"` tracks the current unit types selected. When the dropdown is selected, we can assume the user wants to pick custom units and so the base units are set to custom. Then, the arrow is flipped the other way (much easier than just going to find a new svg, placing it there, editing display etc) and the custom units become visible and can be edited. Finally, over in the fuel tab, if the units *are* set to custom, a warning message is shown to check that there are no mismatches in units. This will prompt them to just pick imperial/metric units and at most conflicts should be avoided this way.

Why I did it this way

There were several ways I could have done this. My first attempt was to have the units change automatically when entering the fuel tab whilst units were set to conflicting ones. This didn't work out well, leading to me having to reload the page a few times to apply changes, but also meaning that if I switched the units to a more metric system from imperial, the detection just switches back to the imperial setting (and vice versa) and I couldn't find an efficient way to keep track of the user's selections. My most notable attempt was to have it so that if the units are changed to conflicting units in settings, and the fuel page was the current page selected, the units would switch automatically to remove any conflicts. What caused this to fail was mostly javascript issues, and the fact that page reloading still needed to happen if the user had changed settings before entering the fuel tab.

Instead, simply having a way for the user to quickly resolve any conflicts just makes the whole process a whole lot more intuitive and easy, especially for the pilot that may already know that they're comfortable with a certain units system and just wants to have that selected right away. And from a usability standpoint, this for sure does make things better for the user!

Program output

 Imperial/metric tests.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
121	Imperial/metric switch, dropdown test	The relevant units should be changed when imperial/metric is selected. If the dropdown is selected, the base units is set to custom	PASS (See program output)
122	Mismatch alert test	If the units are set to custom and the fuel tab is opened, an alert should show to indicate to the user that they should avoid conflicting units.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: Revamped unit selection in settings to avoid conflicts in units

Next steps: AI autofill.

Iteration 7 - AI Autofill

ai_roles.py

```
elif type == 'Fuel':
    role = '''
Calculate appropriate time allowances (in minutes) for the following fuel policy elements according to the policy descriptions provided.

Your calculations should consider the relevant flight data provided, including departure and arrival ICAO codes, distance, and estimate.

Your response should be in the following JSON format:

{
  "times": [trip_time_in_minutes, contingency_time_in_minutes, alternate_time_in_minutes, final_reserve_time_in_minutes, additional_time_in_minutes]
}

All values should be *integers* representing time in minutes.

YOU MUST STRICTLY OBEY THE ABOVE FORMAT, NO COMMENTING, BRACKETS, OR ANYTHING OUTSIDE OF THIS FORMAT OF ANY SORT SHOULD BE INCLUDED.
Do not ask any questions when outputting.
'''
```

fuel.js

```
//-----AI PROMPTING FOR TIME ALLOWED CALCS
const autofillButton = document.getElementById("autofillButton");

autofillButton.addEventListener("click", async () => {
  autofillButton.disabled = true; //disable button to prevent multiple clicks
  autofillButton.innerHTML = `<svg fill="currentColor" height="20" id="AI_icon" viewBox="0 0 24 24"><path d="M12 2L2 12 12 22 22 12 12 2Z" /></svg>`;

  const response = await fetch('/get-flight');
  const flightData = await response.json();

  //constructing prompt with relevant flight data
  const promptText = `
POLICY DESCRIPTIONS:
Trip: ${tripPolicy.value}
Contingency: ${contingencyPolicy.value}
Alternate: ${alternatePolicy.value}
Final Reserve: ${finalReservePolicy.value}
Additional: ${additionalPolicy.value}

RELEVANT FLIGHT DATA:
Departure ICAO: ${flightData.departureAirport_code}
Arrival ICAO: ${flightData.destinationAirport_code}
Alternate ICAO: ${flightData.alternateAirport_code}
Distance: ${flightData.distance.output}
Time Enroute: ${flightData.time}
`;

  const aiResponse = await prompt("Fuel", promptText);
  try {
    const resp = JSON.parse(aiResponse); //parsing the response to JSON
    var times = resp.times; //extracting times object
  } catch (error) {
    console.log(error);
    showAlert("An error occurred whilst generating times. Please try again.");
    autofillButton.disabled = false;
    autofillButton.innerHTML = `<svg fill="currentColor" height="20" id="AI_icon" viewBox="0 0 24 24"><path d="M12 2L2 12 12 22 22 12 12 2Z" /></svg>`;
    return;
  }

  //updating inputs with generated times
  TA_trip.value = times[0]
  TA_contingency.value = times[1]
  TA_alternate.value = times[2]
  TA_finalReserve.value = times[3]
  TA_additional.value = times[4]
});
```



```
//update session data and recalculate fuel requireds
await update("fuelPolicy_trip.time_allowed", Number(TA_trip.value));
await update("fuelPolicy_contingency.time_allowed", Number(TA_contingency.value));
await update("fuelPolicy_alterate.time_allowed", Number(TA_alterate.value));
await update("fuelPolicy_finalReserve.time_allowed", Number(TA_finalReserve.value));
await update("fuelPolicy_additional.time_allowed", Number(TA_additional.value));
await calculateFuelRequired();

autofillButton.disabled = false;
autofillButton.innerHTML = `<svg fill="currentColor" height="20" id="AI_icon" viewBox="0
`);
```

What I did

As stated prior, the AI autofill function is supposed to suggest the time items for each element. The main prompt asks the AI to find times based on the given descriptions, and output them in an array of values. Then, the response is coupled with the actual descriptions, as well as relevant flight data such as trip time, departure/destination to help the AI inform its decisions. Once it returns the array, the response is parsed and to set the times on the screen, I iterate through each value in order, changing the value text. Then I update the session for each of the values, and then finally, update all the fuel required values. At the start and end of this process, the autofill button styling changes to show 'generating...' and switches back to normal once the process is finished.

Why I did it this way

Using an array as output was a much better alternative than using json keys and values, as targeting each time value would be much more complex due to all the names involved. To deal with that I would have to alter the initial table structure in app.py which means I need to change other times where these specific values were called- it just becomes a lengthy mess. Using an array has no effect on the way it's targeted due to the fact that each item is in order, so I can simply iterate through and replace the values as required, which works perfectly.

Program output

 AI autofill test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
123	AI autofill test	Once 'AUTOFILL' is pressed, the time values are filled for the entire table, saved to the session, and the fuel required + total values are calculated afterwards.	PASS (See program output)

124	Dark mode test: fuel tab	Darkmode should apply the correct coloring to all elements.	
-----	--------------------------	---	--

Iteration Review

Met requirements: (Aim complete) Allow these times to be filled by AI automatically based on the current fuel policy for that stage of flight.

Fuel tab complete

Changes: N/A

Next steps: Mass and balance tab

Section Review

Aims completed in this section:

- **Complete an essential feature: Fuel Calculations**
- **Complete a success criteria item: Fuel**, Mass and balance, and Expense calculators
- Allow users to choose their fuel policy and manually enter specific times for each stage of flight.
- Allow these times to be filled by AI automatically based on the current fuel policy for that element of flight.
- Display the users total fuel required for each stage as well as for the whole flight, including the weight of said fuel using data provided by the user
- Determine whether a stopover along the route is required and notify the user of this.

Aims not completed in this section:

N/A

Iterative tests complete:

- Light/dark Mode switches correctly with the correct colours
- AI should output suitable responses in accordance to their given role and user prompt

Remarks: Nothing to address.

Section 10: Animations

Iteration 1 - Animations

basePage.css

```
86 button {
87   margin-bottom: 10px;
88   background-color: var(--highlight);
89   border: none;
90   padding: 5px;
91   border-radius: 5px;
92   transition: background-color 0.2s, transform 0.2s ease-out;
93 }
94
95 button:hover {
96   transform: scale(1.05);
97 }
```

basePage.js

```
36 if (button && LABEL) { //For each matching pair run an event listener for it
37   button.addEventListener("mouseenter", () => {
38     LABEL.style.visibility = "visible";
39     LABEL.style.opacity = "1";
40     gradient.style.visibility = "visible";
41     gradient.style.opacity = "1";
42   });
43   button.addEventListener("mouseleave", () => {
44     LABEL.style.visibility = "hidden";
45     LABEL.style.opacity = "0";
46     gradient.style.visibility = "hidden";
47     gradient.style.opacity = "0";
48   });
49 }
```

settings.css

```
341 button {
342   transition: background-color 0.2s, transform 0.2s ease-out;
343 }
```

```
177 button {
178   transition: background-color 0.2s;
179 }
180
181 #imperial, #metric {
182   color: var(--text);
183 }
184
185 body {
186   transition: background-color 0.2s, color 0.2s;
187 }
```

main-menu.css

```
110 #settings {
111   position: relative;
112   margin-top: 15px;
113   width: 30px;
114   left: 5px;
115   transition: transform 0.3s ease-in-out;
116 }
117
118 #settings:hover {
119   transform: rotate(45deg);
120 }
```

```
80 transition: translate 0.2s;
81 }
82
83 .mainMenuButton:hover {
84   translate: 10px;
85 }
```

```
131 /*gradient overlay*/
132 #NAVOverlay {
133   position: fixed;
134   padding-left: 15px;
135   margin-left: 60px;
136   width: 350px;
137   height: 100vh;
138   padding-top: 8px;
139   background-image: linear-gradient(to right, transparent 49%, #000 49%, #000 51%, transparent 51%);
140   z-index: 50;
141   transition: opacity 0.5s, visibility 0.5s;
142   opacity: 0;
143   visibility: hidden;
144 }
145
146 /*all labels*/
147 .label {
148   position: absolute;
149   background-color: var(--highlight);
150   border-radius: 14px;
151   max-width: fit-content;
152   padding: 10px;
153   padding-left: 17px;
154   padding-right: 17px;
155   opacity: 0;
156   visibility: hidden;
157   transition: opacity 0.2s, visibility 0.2s;
158 }
159
```

What I did

This is more like an unplanned mini section. After the previous tab, I began playing around with the `transition` attribute in CSS. It turns out making a fade in/out transition or a moving transition is not at all hard, and so I took this opportunity to make some QoL improvements to the webapp, the purpose of which are explained below.

Specific changes include:

- Menu buttons are translated to the right when the user hovers over them making a cool selection effect
- Normal buttons now fade into their selection color and are also enlarged with a scale of 0.10x the original button size
- Navigation overlay now has a fade effect which prevents a strobe effect when hovering over different tabs quickly
- Settings icon rotates 45 degrees when hovered over as a little animation effect

Why I did it this way

The purpose of adding animations to my webapp was to improve user experience by showing responsiveness and dedicated feedback to their inputs. Without a doubt, the addition of these animations have greatly improved the smoothness of the application, one could argue to a more professional standard. Without this the webapp would of course be usable, but it would seem less polished, and often this determines the way the user perceives and thinks about the app, and I would like the user to feel like their experience has been thought about and that the webapp is indeed polished in their view.

Program output

 button animations showcase.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
125	Button animation test	All buttons should have the correct animations applied when the cursor interacts with them	PASS (See program output)

Section Review

Met requirements: Added animations!

Changes: N/A

Next steps: Mass and balance tab

Section 11: Mass & Balance Tab

Aims of this section

- **Complete 2 essential features:** **Mass and balance** and **flight performance calculations, Integration of AI for analysis & suggestions**
- **Complete a success criteria item:** **Fuel, Mass and balance**, and Expense calculators
- To allow the user to enter the weight and arm of each measurement item and have the moment calculated based on that information
- Calculate the total total weight, moment and centre of gravity at different stages of flight and display them to the user in a table
- Allow the user to enter details related to manoeuvring speeds to produce manoeuvring speed values for takeoff and landing
- Allow manoeuvring speed to be calculated from a custom weight
- Have weights that go over the entered limits highlighted.
- To have AI suggest a change the user could make should their CG value be out of limits

Iteration 1 - HTML initial layout and styling

mass_and-balance_html

```
<!--=====BODY=====-->
<div id="content">
  <div id="half1">
    <!--Weights and moments-->
    <div id="weights_and_moments" class="highlight">
      <h3>Weights & Moments</h3>
    </div>
    <!--Footer & change suggest button-->
    <div id="footer" style="display: flex; flex-direction: column; align-items: center;
      <div style="margin-bottom: 20px;"><p style="color: #737373">Use your aircrafts w
      <div style="display: flex; align-items: center; justify-content: center;">
        <p style="position: relative; bottom: 4px;"><b style="margin-right: 15px;">Not i
        <button style="display: flex;" id="suggestChange"><svg fill="currentColor" heigh
      </div>
    </div>
  </div>
  <div id="half2">
    <!--CG calculator-->
    <div id="cg_calculator">
      <h3>Center of Gravity Calculator</h3>
      <span style="display: flex; justify-content: center; gap: 50px; flex-wrap: wrap;">
        <!--Maximum takeoff weight--><div><b>MTOW:</b> <input type="number" name="Max_Ta
        <!--Maximum landing weight--><div><b>MLW :</b> <input type="number" name="Max_La
      </span>
    </div>
    <!--Manoeuvring speeds-->
    <div id="manoeuvring_speeds" class="highlight">
      <h3>Manoeuvring Speeds</h3>
      <div style="display: flex; justify-content: center; gap: 50px; flex-wrap: wrap; ma
        <!--Va @ MGW--><div><b>V<sub>A</sub></b> @ MGW:</b> <input type="number" name="Va_MG
        <!--Max Gross Weight --><div><b>MGW:</b> <input type="number" name="Max_Gross_We
      </div>
      <div style="display: flex; flex-direction: column; gap: 10px; flex-wrap: wrap; m
        <p>V<sub>A</sub> Takeoff: <span id="va_takeoff"></span></p>
        <p>V<sub>A</sub> Landing: <span id="va_landing"></span></p>
      </div>
    <!--Any weight calculation-->
    <div style="margin-bottom: 15px;"><b>Any Weight:</b> <input type="number" name="
```

Mass_and_balance.css

```
.highlight {
  background-color: var(--highlight);
  padding: 20px;
  border-radius: 8px;
  margin-bottom: 15px;
}

#content {
  display: grid;
  grid-template-columns: 1.5fr 1fr;
}

#weights_and_moments {
  margin: 10px;
  min-height: 500px;
}

#cg_calculator {
  margin: 10px;
  min-height: 350px;
}

#manouvering_speeds{
  margin: 10px;
  height: fit-content;
}

#manouvering_speeds input {
  border: 1px solid var(--onhover);
}

input {
  border: 1px solid var(--text);
  border-radius: 5px;
  padding: 5px;
  margin-left: 10px;
  width: 100px;
  text-align: center;
  font-weight: 600;
  font-family: 'Manrope';
  font-size: 13px;
  background-color: var(--bg);
}

#suggestChange {
  background-color: var(--highlight);
  border: none;
  border-radius: 5px;
  padding: 8px 12px;
  font-size: 14px;
  font-weight: 400;
  cursor: pointer;
  margin-left: 10px;
  font-family: 'Manrope';
}
```

```
#suggestChange:hover {
  background-color: var(--primary);
}

#AI_icon {
  margin-right: 10px;
  position: relative;
}

h3 {
  margin-bottom: 15px;
}
```

What I did

Like always, we start with the html and styling. I've used css grid to make 2 halves, one containing the weights and moments and the footer, and the other with the centre of gravity and manoeuvring speed calculations. The tables are a bit more complex and take a few lines of code so they'll be done in the next iterations. I decided to centre the footer too instead of having it to the left- this just looked a bit cleaner.

Why I did it this way

Not using a CSS grid of 4 items (one for each section) was important as I could not get them to overlap each other this way. The overlapping was necessary as part of the design due to space constraints- I wanted to have it so the user has to scroll less when accessing any page. Of course, this can often depend on the screen size the user has. Using a tablet for example will almost always mean scrolling is required, but at least the amount will be decreased significantly.

Program output

MASS & BALANCE

Weights & Moments

Use your aircrafts weight & balance schedule to check if your CG is in limits.

Not in limits?

Suggest a Change

MTOW:

MLW:

Manoeuvring Speeds

V_A @ MGW:

MGW:

V_A Takeoff:

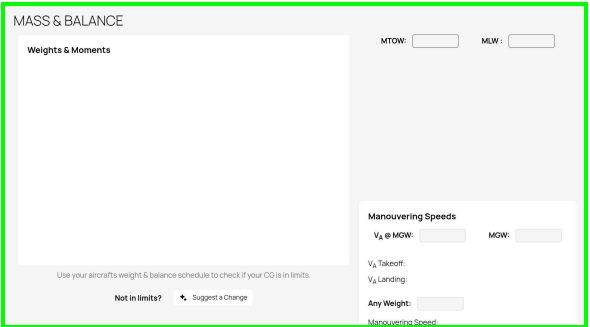
V_A Landing:

Any Weight:

Manoeuvring Speed:

Known Limitations / Missing features

- No tables
- No JS

Test No.	Test Description	Expected Outcome	Actual Outcome
126	Initial HTML & CSS layout test	All the HTML and styling should match the design, except for the tables which are incomplete.	

Iteration Review

Met requirements: N/A

Changes: Centered the footer

Next steps: Table HTML and styling

Iteration 2 - Table HTML and styling

mass_and_balance.css

```
/*table styling*/

table {
  width: 100%;
  border-collapse: collapse;
  margin-top: 10px;;
}

#CG_table th, #CG_table td {
  border: 1px solid var(--text);
  text-align: center;
  font-family: 'Manrope';
  font-size: 14px;
  padding: 15px;
}

th {
  background-color: var(--tableHeader);
}

#table_left, #table_right {
  width: 35px;
  height: 35px;
  pointer-events: none;
  opacity: 0.3;
}
```



```

        transition: transform 0.2s ease;
    }

    #table_right {
        transform: rotate(180deg);
    }

    #moments_table th, #moments_table td {
        border: 1px solid var(--text);
        text-align: center;
        font-family: 'Manrope';
        font-size: 14px;
        padding: 10px;
        width: 180px;
    }

    #moments_table input {
        width: 160px;
        border: 1px solid var(--onhover);
        background-color: var(--highlight);
    }

    #moments_table1, #moments_table2 {
        display: none;
    }

    #moments_table1.active, #moments_table2.active {
        display: table-row-group;
    }

    /*table arrows for user feedback*/
    #table_left.active, #table_right.active {
        opacity: 1;
        pointer-events: auto;
        cursor: pointer;
    }

    #table_left:hover, #table_right:hover {
        transform: scale(1.1);
    }

    #table_right:hover {
        transform: rotate(180deg) scale(1.1);
    }

```

mass_and_balance.html

```

22 <!--HEADER-->
23 <div id="weights_and_moments" class="highlight">
24   <div style="display: flex; justify-content: space-between">
25     <h3>Weights & Moments</h3>
26     <span>
27       <svg id="table_left" fill="currentColor" viewBox="0 0 32 32" xmlns="http://www.w3
28       <svg id="table_right" class="active" fill="currentColor" viewBox="0 0 32 32" xmlns
29     </span>
30   </div>
31 <!--BODY-->
32 <table id="moments_table">
33   <thead>
34     <tr>
35       <th>Item</th>
36       <th>Weight</th>
37       <th>Arm</th>
38       <th>Moment</th>
39     </tr>
40   </thead>
41   <!--TABLE DISPLAY 1-->
42   <tbody id="moments_table1" class="active">
43     <tr>
44       <td>Basic Empty Weight</td>
45       <td><input type="number" name="BE_weight" id="weight_basic_empty"></td>
46       <td><input type="number" name="BE_arm" id="arm_basic_empty"></td>
47       <td><input type="number" name="BE_moment" id="moment_basic_empty"></td>
48     </tr>
49     <tr>
50       <td>Oil</td>
51       <td><input type="number" name="oil_weight" id="weight_oil"></td>
52       <td><input type="number" name="oil_arm" id="arm_oil"></td>
53       <td id="moment_oil"></td>
54     </tr>
55     <tr>
56       <td>Pilot 1</td>
57       <td><input type="number" name="pilot1_weight" id="weight_pilot1"></td>
58       <td><input type="number" name="pilot1 arm" id="arm_pilot1"></td>

```

```

<!--TABLE DISPLAY 2-->
<tbody id="moments_table2">
  <tr>
    <td>Fuel load: Tank 1</td>
    <td><input type="number" name="fuel1_weight" id="weight_fuel1"></td>
    <td><input type="number" name="fuel1_arm" id="arm_fuel1"></td>
    <td id="moment_fuel1"></td>
  </tr>
  <tr>
    <td>Fuel load: Tank 2</td>
    <td><input type="number" name="fuel2_weight" id="weight_fuel2"></td>
    <td><input type="number" name="fuel2_arm" id="arm_fuel2"></td>
    <td id="moment_fuel2"></td>
  </tr>
  <tr>
    <td>Fuel burned on ground: Tank 1</td>
    <td><input type="number" name="fuel1_burned_weight" id="weight_fuel1_ground_burned"></td>
    <td><input type="number" name="fuel1_burned_arm" id="arm_fuel1_ground_burned"></td>

```

```

<!--CG table-->
<table id="CG_table">
  <thead>
    <tr>
      <th>Condition</th>
      <th>Weight</th>
      <th>Moment</th>
      <th>C/G</th>
    </tr>
  </thead>
  <tbody>
    <tr>
      <td>Basic</td>
      <td><span id="basic_weight"></span></td>
      <td><span id="basic_moment"></span></td>
      <td><span id="basic_CG"></span></td>
    </tr>
    <tr>
      <td>Zero-Fuel</td>
      <td><span id="zero_fuel_weight"></span></td>
      <td><span id="zero_fuel_moment"></span></td>
      <td><span id="zero_fuel_CG"></span></td>
    </tr>
    <tr>
      <td>Ramp</td>
      <td><span id="ramp_weight"></span></td>
      <td><span id="ramp_moment"></span></td>
    </tr>
  </tbody>
</table>

```

mass_and_balance.js

```

const table_left = document.getElementById('table_left');
const table_right = document.getElementById('table_right');
const moments_table1 = document.getElementById('moments_table1');
const moments_table2 = document.getElementById('moments_table2');

//Initial state (no session involvement - will always be 1 on page load)
let current_table = 1;

//Event listeners for table arrows
table_right.addEventListener('click', () => {
  if (current_table === 1) {
    moments_table2.classList.add('active');
    moments_table1.classList.remove('active');
    table_left.classList.add('active');
    table_right.classList.remove('active');
    current_table = 2;
  }
});

table_left.addEventListener('click', () => {
  if (current_table === 2) {
    moments_table1.classList.add('active');
    moments_table2.classList.remove('active');
    table_right.classList.add('active');
    table_left.classList.remove('active');
    current_table = 1;
  }
});

```

What I did

To incorporate the table 'split' that you see in the design, I've used two <tbody> elements instead of one. Each table contains a certain number of rows, with the necessary details. In this fashion, I can alter the display properties for each table upon clicking an arrow, which allows me to switch between table contents without changing the table headers.

To incorporate the switch, each table can either have 'active' in their classList or not. When a particular table needs to be shown, 'active' is added to their classlist and the relevant table is shown (display set to `table-row-group`, not block as it wouldn't format properly.) The same applies to the arrows, in which, the active one is colored in and the unavailable one is dimmed out. By default, the first table shows on page load.

Why I did it this way

I didn't consider saving the currently selected table in the session because it's simply not something that needs to be saved. There is no benefit in saving the table position as data inputted is always saved and it only takes one click of a button to move to the next table. Saving the table position to session would be a waste of time and session space.

Program output

Table switch test.mp4

MASS & BALANCE

Weights & Moments

←

→

Item	Weight	Arm	Moment
Basic Empty Weight			
Oil			
Pilot 1			
Pilot 2			
PAX 1			
PAX 2			
Baggage section 1			
Baggage section 2			

Use your aircrafts weight & balance schedule to check if your CG is in limits.

Not in limits?

 Suggest a Change

MTOW: MLW:

Condition	Weight	Moment	C/G
Basic			
Zero-Fuel			
Ramp			
Takeoff			
Landing			

Manouvering Speeds

V_A @ MGW:

MGW:

V_A Takeoff:

V_A Landing:

Any Weight:

Manouvering Speed:

Known Limitations / Missing features

- No js

Test No.	Test Description	Expected Outcome	Actual Outcome
127	Table design testing	Table layout and designs should match the original design	
128	Table switch test	Upon clicking the relevant arrow, the relevant table should display and so should the styling of both arrows as appropriate.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: User input handling

Iteration 3 - User Input handling

app.py

```

267 #-----MASS AND BALANCE
268 #BASIC INPUTS#
269 "MTOW" : {"value" : 0, "class" : "mass"},
270 "MLW" : {"value" : 0, "class" : "mass"},
271 "MGW" : {"value" : 0, "class" : "mass"},
272 "va_MGW" : {"value" : 0, "class" : "airspeed"},
273 "Any_Weight" : {"value" : 0, "class" : "mass"},
274 "va_any" : {"value" : 0, "class" : "airspeed"},
275 "va_takeoff" : {"value" : 0, "class" : "airspeed"},
276 "va_landing" : {"value" : 0, "class" : "airspeed"},
277 #WEIGHT/BALANCE INPUTS#
278 "weight_items" : {
279     "basic_empty" : {
280         "weight" : {"value" : 0, "class" : "mass"},
281         "arm" : 0,
282         "moment" : 0,
283     },
284     "oil" : {
285         "weight" : {"value" : 0, "class" : "mass"},
286         "arm" : 0,
287         "moment" : 0,
288     },
289     "pilot1" : {
290         "weight" : {"value" : 0, "class" : "mass"},
291         "arm" : 0,
292         "moment" : 0,
293     },
294     "pilot2" : {

```

```

2      #CG OUTPUTS#
3      "CG_calulations" : {
4          "basic_condition" : {
5              "weight" : {"value" : 0, "class" : "mass"},
6              "moment" : 0,
7              "CG" : 0
8          },
9          "zero_fuel_condition" : {
10             "weight" : {"value" : 0, "class" : "mass"},
11             "moment" : 0,
12             "CG" : 0
13         },
14         "ramp_condition" : {
15             "weight" : {"value" : 0, "class" : "mass"},
16             "moment" : 0,
17             "CG" : 0
18         },
19         "takeoff_condition" : {
20             "weight" : {"value" : 0, "class" : "mass"},
21             "moment" : 0,
22             "CG" : 0
23         },
24         "landing_condition" : {
25             "weight" : {"value" : 0, "class" : "mass"},
26             "moment" : 0,
27             "CG" : 0
28         }
29     },

```

mass_and_balance.html

```

-----TABLE DISPLAY 2----->
id="moments_table2">

d>Fuel load: Tank 1</td>
d><input type="number" name="fuel_load1_weight" id="weight_fuel_load1" value="{{ flight.weight_items.fuel_load1.weight.v
d><input type="number" name="fuel_load1_arm" id="arm_fuel_load1" value="{{ flight.weight_items.fuel_load1.arm if flight.
d id="moment_fuel_load1">{{ flight.weight_items.fuel_load1.moment if flight.weight_items.fuel_load1.moment != 0 else ''
>

d>Fuel load: Tank 2</td>
d><input type="number" name="fuel_load2_weight" id="weight_fuel_load2" value="{{ flight.weight_items.fuel_load2.weight.v
d><input type="number" name="fuel_load2_arm" id="arm_fuel_load2" value="{{ flight.weight_items.fuel_load2.arm if flight.
d id="moment_fuel_load2">{{ flight.weight_items.fuel_load2.moment if flight.weight_items.fuel_load2.moment != 0 else ''
>

d>Fuel burned on ground: Tank 1</td>
d><input type="number" name="fuel1_burned_weight" id="weight_fuel_ground_burned1" value="{{ flight.weight_items.fuel_gro
d><input type="number" name="fuel1_burned_arm" id="arm_fuel_ground_burned1" value="{{ flight.weight_items.fuel_ground_bu
d id="moment_fuel_ground_burned1">{{ flight.weight_items.fuel_ground_burned1.moment if flight.weight_items.fuel_ground_b
>

d>Fuel burned on ground: Tank 2</td>

<th>Condition</th>
<th>Weight</th>
<th>Moment</th>
<th>C/G</th>
<tr>
<td>Basic</td>
<td><span id="basic_weight"></span>{{ flight.CG_calulations.basic_condition.weight.value if flight.CG_calu
<td><span id="basic_moment"></span>{{ flight.CG_calulations.basic_condition.moment if flight.CG_calulation
<td><span id="basic_CG"></span>{{ flight.CG_calulations.basic_condition.CG if flight.CG_calulations.basic
<tr>
<td>Zero-Fuel</td>
<td><span id="zero_fuel_weight"></span>{{ flight.CG_calulations.zero_fuel_condition.weight.value if flight
<td><span id="zero_fuel_moment"></span>{{ flight.CG_calulations.zero_fuel_condition.moment if flight.CG_ca
<td><span id="zero_fuel_CG"></span>{{ flight.CG_calulations.zero_fuel_condition.CG if flight.CG_calulation
<tr>
<td>Ramp</td>
<td><span id="ramp_weight"></span>{{ flight.CG_calulations.ramp_condition.weight.value if flight.CG_calula
<td><span id="ramp_moment"></span>{{ flight.CG_calulations.ramp_condition.moment if flight.CG_calulations

```


mass_and_balance.js

```
//-----WEIGHT/BALANCE INPUTS

//BASIC EMPTY
const weight_basic_empty = document.getElementById('weight_basic_empty');
const arm_basic_empty = document.getElementById('arm_basic_empty');
const moment_basic_empty = document.getElementById('moment_basic_empty');

//Event listeners for basic empty
weight_basic_empty.addEventListener('input', () => {
  update('weight_items.basic_empty.weight.value', Number(weight_basic_empty.value));
});

arm_basic_empty.addEventListener('input', () => {
  update('weight_items.basic_empty.arm', Number(arm_basic_empty.value));
});

moment_basic_empty.addEventListener('input', () => {
  update('weight_items.basic_empty.moment', Number(moment_basic_empty.value));
});

//OIL
const weight_oil = document.getElementById('weight_oil');
const arm_oil = document.getElementById('arm_oil');
const moment_oil = document.getElementById('moment_oil');

//Event listeners for oil
weight_oil.addEventListener('input', () => {
  update('weight_items.oil.weight.value', Number(weight_oil.value));
});

arm_oil.addEventListener('input', () => {
  update('weight_items.oil.arm', Number(arm_oil.value));
});

//PILOT1
```

```
//-----CG OUTPUTS

//basic condition
const basic_weight = document.getElementById('basic_weight');
const basic_moment = document.getElementById('basic_moment');
const basic_CG = document.getElementById('basic_CG');

//zero fuel condition
const zero_fuel_weight = document.getElementById('zero_fuel_weight');
const zero_fuel_moment = document.getElementById('zero_fuel_moment');
const zero_fuel_CG = document.getElementById('zero_fuel_CG');

//ramp condition
const ramp_weight = document.getElementById('ramp_weight');
```

```
const MTOW = document.getElementById('MTOW'); //Maximum Takeoff Weight
const MLW = document.getElementById('MLW'); //Maximum Landing Weight
const MGW = document.getElementById('MGW'); //Maximum Gross Weight
const va_MGW = document.getElementById('va_MGW'); //Manouvering Speed at Maximum Gross Weight
const Any_Weight = document.getElementById('Any_Weight'); //Any Weight
const va_any = document.getElementById('va_any'); //Manouvering Speed at Any Weight
const va_takeoff = document.getElementById('va_takeoff'); //Manouvering Speed at Takeoff Weight
const va_landing = document.getElementById('va_landing'); //Manouvering Speed at Landing Weight

//Event listeners for basic inputs
MTOW.addEventListener('input', () => {
  update('MTOW.value', Number(MTOW.value));
});

MLW.addEventListener('input', () => {
  update('MLW.value', Number(MLW.value));
});

MGW.addEventListener('input', () => {
  update('MGW.value', Number(MGW.value));
});

va_MGW.addEventListener('input', () => {
  update('va_MGW.value', Number(va_MGW.value));
});
```

What I did

To handle user inputs, as always I need to add the relevant data to the session template. The majority of the inputs here are indeed unit-changing, so they have the relevant format. Table items have been added in the same way as in the fuel section, as the values need not be updated dynamically.

In **mass_and_balance.js** each cell in the table has its own id with its own event listener listening for an input to then save the currently entered value in the session. Similarly, the normal inputs have event listeners too and listen for inputs that will mostly just affect the manoeuvring speed values.

Why I did it this way

I had to make a decision on whether or not to store the table in a similar way to what was in the navlog, or to do it like I did in the fuel section. The issue with the latter was that there's simply just so many cells! I thought that this may have an effect on how the code ran but after testing it seems to not have any effect. I decided not to use arrays to form the table because I did not want to formulate the table from the session, rather I wanted only the data in the table cells to be pulled from the session. Doing it the way I did allows this to happen easily.

Program output

 Initial input handling test M&B.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
129	Input handling test	Inputs should be able to be saved to the session upon entry. (saved data tested by reloading the page)	ERROR (See program output)

Error Analysis & Fixes

Test No: 129

Failure Description: PAX 1 & 2 weight and arm values, as well as all manoeuvring speed inputs do not save in the session.

Cause: Most probable cause is a wrong id call somewhere, either in the HTML or in app.py. If this isn't the case, it has to be an event listener issue or something similar.

Fix: Search across mass_and_balance.js and app.py and check for id discrepancies

Justification: When an item doesn't save to the session, it can only be because it did not find said attribute in the session to save to. Usually when this is the case a new attribute is created, but even then it would be outside the necessary object and so will still not be

visible on the screen. This is how I know that it is most likely an id issue and not an alternative error.

Iteration 3.5 - Fixing session saving error

I quickly found the error with the PAX values, it was indeed an id discrepancy. In the session it is called 'PAX1/2' whilst **mass_and_balance.js** has the same thing but in lower case.

app.py

```
},
"PAX1" : {
  "weight" : {"value" : 0, "class" : "mass"},
  "arm" : 0,
  "moment" : 0,
},
"PAX2" : {
  "weight" : {"value" : 0, "class" : "mass"},
  "arm" : 0,
  "moment" : 0,
}
```

mass_and_balance.js

```
//Event listeners for pax 1
weight_pax1.addEventListener('input', () => {
  update('weight_items.pax1.weight.value', Number(weight_pax1.value));
});

arm_pax1.addEventListener('input', () => {
  update('weight_items.pax1.arm', Number(arm_pax1.value));
});

//PAX2
const weight_pax2 = document.getElementById('weight_pax2');
const arm_pax2 = document.getElementById('arm_pax2');
const moment_pax2 = document.getElementById('moment_pax2');

//Event listeners for pax 2
weight_pax2.addEventListener('input', () => {
  update('weight_items.pax2.weight.value', Number(weight_pax2.value));
});

arm_pax2.addEventListener('input', () => {
  update('weight_items.pax2.arm', Number(arm_pax2.value));
});
```

For the second error, I forgot to add the path .value to the items (as they are unit changing so have that config):

```
px;">
value="{{ flight.Va_MGW }}"></div>
value="{{ flight.MGW }}"></div>

5px;">

"Any_Weight" value="{{ flight.Any_Weight }}"><
```

Program output

 user input handling M&B 2.mp4

Test No.	Test Description	Expected Outcome	Actual Outcome
129	Input handling test	Inputs should be able to be saved to the session upon entry. (saved data tested by reloading the page)	PASS (See program output)

Iteration Review

Met requirements: N/A (inputs complete but not outputs)

Changes: N/A

Next steps: Calculating moments

Iteration 4 - Calculating moments and CG

mass_and_balance.js

```
//-----CALCULATING MOMENTS-----

async function calculateMoment(target) {
  //target acts as the id to identify which moment cell is being calculated
  const weight = document.getElementById(`weight_${target}`); //used here as suffix
  const arm = document.getElementById(`arm_${target}`);
  const moment = document.getElementById(`moment_${target}`);

  //moment is 0 if either weight or arm is empty
  if (weight.value === '' || arm.value === '') {
    moment.textContent = '';
    await update(`weight_items.${target}.moment`, 0);
    await calculateCG();
    return;
  }

  //moment = weight * arm
  const calculated_moment = (Number(weight.value) * Number(arm.value));

  if (target === 'basic_empty') { //exception for basic empty as moment is an input field
    moment.value = calculated_moment;
  } else {
    moment.textContent = calculated_moment;
  }

  //updating the session with new moment
  await update(`weight_items.${target}.moment`, calculated_moment);
  await calculateCG();
}
```

```
//-----CALCULATING CG-----

async function calculateCG() {
  const items = [['basic_empty', 'oil', 'pilot1', 'pilot2'], //basic condition
    ['PAX1', 'PAX2', 'baggage1', 'baggage2'], //zero fuel condition
    ['fuel_load1', 'fuel_load2'], //ramp condition
    ['fuel_ground_burned1', 'fuel_ground_burned2'], //takeoff condition
    ['fuel_flight_burned1', 'fuel_flight_burned2']]; //landing condition
  const conditions_output = ["basic_", "zero_fuel_", "ramp_", "takeoff_", "landing_"];

  var total_weight = 0;
  var total_moment = 0;

  //iterating through each condition
  items.forEach( async (condition) => {
    //iterating through each item in that condition
    condition.forEach( async (item) => {
      //taken the weight and moment from that row
      const weight = document.getElementById(`weight_${item}`);
      const moment = document.getElementById(`moment_${item}`);

      if (item !== 'basic_empty') {
        if (weight.value !== '' && moment.textContent !== '') {
          //for takeoff and landing we subtract the weight and moment (fuel is being burned b
          if (conditions_output[items.indexOf(condition)] === 'takeoff_' || conditions_output
            total_weight -= Number(weight.value);
            total_moment -= Number(moment.textContent);

          //otherwise we add them (so basic, zero fuel and ramp conditions)
        } else {
          total_weight += Number(weight.value);
          total_moment += Number(moment.textContent);
        }
      }
    })
  })
}
```

```
//getting the output fields for that condition
const weight_output = document.getElementById(`${conditions_output[items.indexOf(condition)]}weight`);
const moment_output = document.getElementById(`${conditions_output[items.indexOf(condition)]}moment`);
const CG_output = document.getElementById(`${conditions_output[items.indexOf(condition)]}CG`);

//updating the output fields
weight_output.textContent = roundifDecimal(total_weight) !== 0 ? roundifDecimal(total_weight) : '';
moment_output.textContent = roundifDecimal(total_moment) !== 0 ? roundifDecimal(total_moment) : '';

//calculating CG = moment / weight
if (total_weight === 0 || total_moment === 0) {
  CG_output.textContent = '';
} else {
  var calculated.CG = total_moment / total_weight;
  CG_output.textContent = roundifDecimal(calculated.CG) !== 0 ? roundifDecimal(calculated.CG) : '';
}
});
//finally update session with new values in the CG table
const response = await fetch("/update-cg-table", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify({ table: {
    //send the entire table as one into backend
    basic_condition: {
      weight: {"value" : Number(basic_weight.textContent), class : 'mass'},
      moment: Number(basic_moment.textContent),
      CG: Number(basic.CG.textContent)
    },
    zero_fuel_condition: {
      weight: {"value" : Number(zero_fuel_weight.textContent), class : 'mass'},
      moment: Number(zero_fuel_moment.textContent),
      CG: Number(zero_fuel.CG.textContent)
    }
  } })
});
```

What I did

The moments are calculated in a similar way to how the totals in the fuel section were calculated. I used a suffix so that the target passed into the **calculateMoment()** function can fit any of the cells simply by using the suffix. In this fashion I can access any cell in the table. Then, after calculating the moment, the `textContent` in the cell is updated and the relevant value is also updated in the session.

The CG calculations are where things get a little more complex. An aircraft condition in this case refers to the state of the aircraft at a specific stage in its journey. For instance, the basic empty weight here considers the aircraft, its oil and 2 pilots, but no passengers or luggage. Then for each condition, there will be a total weight, moment, and this a centre of gravity (CG) calculated by performing moment/weight.

The conditions are stored in a 2D array, and each condition contains the suffixes of all the relevant fuel items in that condition. The function iterates through each condition, then inside that, iterates through each item in that condition. The weight and moment values are all added up to produce the total weights and moments for that condition, which is what is displayed in the table. For takeoff and landing conditions, fuel will have decreased before these conditions so an if statements checks and ensures such conditions have their items subtracted and not added. The weights and moments are added cumulatively, so the totals are defined outside of the loops.

Then, the CG is calculated using those totals. At this point we now have all 3 values for a condition so we move to the next one, until all conditions have their values calculated. Then first the values on the screen are replaced to show the new table immediately, then all values of the table are passed in the format laid out in the session file and sent to the backend, where the current table is replaced with the newly filled one.

Why I did it this way

The alternative to updating the table in the session was to use the normal **update()** function and simply update each cell one after the other. This worked but was really slow- I could update another cell whilst the other one I just entered is still having the data processed to calculate new CG values. Also the server needs to be accessed multiple times, which is most likely the reason why it was so slow. The current method is much simpler as I can just access the table once, with the entire table ready and the save takes a significantly shorter time to complete.

Program output

 M&B moments and CG calc test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
130	Moments and centre of gravity calculation test	Upon editing an item, the moment, if it can, should be calculated and displayed in the correct column. Then	PASS (See program output)

		the CG table should be filled using current data inputted in the weights and balance table.	
--	--	---	--

Iteration Review

Met requirements: (Aim complete) To allow the user to enter the weight and arm of each measurement item and have the moment calculated based on that information
(Aim complete) Calculate the total total weight, moment and centre of gravity at different stages of flight and display them to the user in a table

Changes: N/A

Next steps: Unit conversions

Iteration 5 - Unit conversion fixing

app.py

```
def update_units():
    units_changed = session["flight_data"]["settings"].get("units_changed", "False") == "True"

def process_item(item):
    #only process items that are dictionaries with a "class" key (i.e. unit-based items)
    if isinstance(item, dict) and "class" in item:
        category = item["class"]
        if category in UNIT_MAP:
            settings_key = UNIT_MAP[category]["settings_key"]
            display_unit = session["flight_data"]["settings"][settings_key]
            current_unit = ureg[display_unit]

            #store previous unit in session settings if not present
            prev_unit_key = f"prev_{settings_key}"
            if prev_unit_key not in session["flight_data"]["settings"]:
                session["flight_data"]["settings"][prev_unit_key] = display_unit

            if units_changed:
                #convert value from previous unit to new unit
                prev_unit = session["flight_data"]["settings"][prev_unit_key]
                try:
                    prev_quantity = item["value"] * ureg[prev_unit]
                    new_quantity = prev_quantity.to(current_unit)
                    item["value"] = float(new_quantity.magnitude)
                except Exception:
                    pass

            #always update output
            canonical = item["value"] * current_unit
            converted = canonical.to(current_unit)
            unit_name = str(converted.units)
            unit_name = CUSTOM_UNITS.get(unit_name, unit_name)
            item["output"] = f"{converted.magnitude:.1f} {unit_name}"

        elif isinstance(item, dict):
            for subkey, subitem in item.items():
                process_item(subitem)
```

update_units() continued

```
#process all items in flight data
for key, item in session["flight_data"]["flight"].items():
    process_item(item)

#update previous unit in session settings
def process_prev_unit(item):
    if isinstance(item, dict) and "class" in item:
        category = item["class"]
        if category in UNIT_MAP:
            settings_key = UNIT_MAP[category]["settings_key"]
            display_unit = session["flight_data"]["settings"][settings_key]
            #store previous unit in session settings if not present
            prev_unit_key = f"prev_{settings_key}"
            session["flight_data"]["settings"][prev_unit_key] = display_unit

    elif isinstance(item, dict):
        for subkey, subitem in item.items():
            process_prev_unit(subitem)

for key, item in session["flight_data"]["flight"].items():
    process_prev_unit(item)

#reset the flag to show units have been updated
session["flight_data"]["settings"]["units_changed"] = "False"
session.modified = True
```

What I did

This iteration occurred as I noticed an error whilst testing out unit conversions that meant I needed to rework the `update_units()` function to compensate.

A unit-converting value looks like this:

```
"item" : {"value" : 0, "class" : "[class]"}
```

and the 'output' value is added once `update_units()` runs on that value. This allows me to know that the unit can be converted/has been converted. This output value (which contains the whole value plus its units) was missing for all values inside of `"weight_items"` and `"CG_calculations"`. This was because these two values don't themselves have the value and class items, which is what the function looks for before converting. To fix this I needed to find a way of being able to access dict items inside of a parent item.

To do this I created a separate function `process_item()` which contains the code to convert the units. Then within this function, the if statement checks if a dict with 'class' is inside as normal, and processes the unit-converting value if it does. If it doesn't it will then begin to check the items within this primary item and recursively run `process_item()` for each of those. This will eventually keep going until either a value has nothing inside that fits the required format, or a unit-changing value is actually found. And the exact same thing happens with the previous unit function.

Why I did it this way

This recursive approach allows me to go as deep as I needed into any dict object. If I didn't do this recursively, and instead just targeted one 'step' inside, then I would definitely be limited by the way I must store my values in the data template in order for such a system to still work (meaning I would only be able to have a parent item with a unit-converting item immediately after).

Program output

 M&B unit conversion test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
131	Unit conversion test	Upon changing the units, all weight and mass values should convert. A successful conversion will see the CG remain the same for both imperial and metric unit types.	PASS (See program output)

Iteration Review

Met requirements: Essential feature complete: Mass and balance, and flight performance calculations

Success criteria item complete: Fuel, Mass and balance, and Expense calculators

Changes: Re-worked the units function to act recursively to enable it to access nested values.

Next steps: Pre-inputting table values, symmetric arm value autofill

Iteration 6 - Pre-inputted values & symmetric arm autofill

app.py - calculateTotals()

```
###MASSES###
specific_gravity = flight_data["specific_gravity"]["value"]
if specific_gravity != "" and specific_gravity != 0:
    #mass = volume * density (specific gravity)
    fuel_mass = total_fuel * float(specific_gravity)
    flight_data["fuel_mass"]["value"] = round(fuel_mass, 1)
    #distributing fuel mass equally between 2 tanks for weight and balance
    flight_data["weight_items"]["fuel_load1"]["weight"]["value"] = fuel_mass / 2
    flight_data["weight_items"]["fuel_load2"]["weight"]["value"] = fuel_mass / 2

    #fuel burned on ground
    flight_data["weight_items"]["fuel_ground_burned1"]["weight"]["value"] = (float(taxi) * float(specific_gravity))
    flight_data["weight_items"]["fuel_ground_burned2"]["weight"]["value"] = (float(taxi) * float(specific_gravity))

    #fuel burned in flight
    flight_data["weight_items"]["fuel_flight_burned1"]["weight"]["value"] = (float(trip) * float(specific_gravity))
    flight_data["weight_items"]["fuel_flight_burned2"]["weight"]["value"] = (float(trip) * float(specific_gravity))
else:
    flight_data["fuel_mass"]["value"] = 0
    flight_data["weight_items"]["fuel_load1"]["weight"]["value"] = 0
    flight_data["weight_items"]["fuel_load2"]["weight"]["value"] = 0
    flight_data["weight_items"]["fuel_ground_burned1"]["weight"]["value"] = 0
    flight_data["weight_items"]["fuel_ground_burned2"]["weight"]["value"] = 0
    flight_data["weight_items"]["fuel_flight_burned1"]["weight"]["value"] = 0
    flight_data["weight_items"]["fuel_flight_burned2"]["weight"]["value"] = 0
```


mass_and_balance.js

```
3
4 //-----symmetric arm checkbox handling
5 const symmetricArmCheckbox = document.getElementById('symmetricArmCheckbox');
6
7 //Event listener for checkbox changes
8 symmetricArmCheckbox.addEventListener('change', async () => {
9     if (symmetricArmCheckbox.checked) {
10         await updateSettings('fill_symmetric_arms', true);
11     } else {
12         await updateSettings('fill_symmetric_arms', false);
13     }
14 });
15
```

```
89 //-----WEIGHT/BALANCE INPUTS
90
91 //Arm setting check
92 async function checkSymmetricArms() {
93     const response = await fetch('/get-settings');
94     const data = await response.json();
95     return data.fill_symmetric_arms;
96 }
```

```
//PAX1 (PAX = Passenger)
const weight_PAX1 = document.getElementById('weight_PAX1');
const arm_PAX1 = document.getElementById('arm_PAX1');
const moment_PAX1 = document.getElementById('moment_PAX1');

//Event listeners for pax 1
weight_PAX1.addEventListener('input', async () => {
    await update('weight_items.PAX1.weight.value', Number(weight_PAX1.value));
    await calculateMoment('PAX1');
    await calculateCG();
});

arm_PAX1.addEventListener('input', async () => {
    await update('weight_items.PAX1.arm', Number(arm_PAX1.value));

    if (await checkSymmetricArms()) {
        //if symmetric arms is checked, update PAX2 arm as well
        arm_PAX2.value = arm_PAX1.value;
        await update('weight_items.PAX2.arm', Number(arm_PAX2.value));
        await calculateMoment('PAX2');
    } else {
        await calculateMoment('PAX1');
    }
    await calculateCG();
});
```

```
arm_fuel_load1.addEventListener('input', async () => {
    await update('weight_items.fuel_load1.arm', Number(arm_fuel_load1.value));

    if (await checkSymmetricArms()) {
        //if symmetric arms is checked, update all other fuel values as well
        await setFuelArms(arm_fuel_load1.value);
    } else {
        await calculateMoment('fuel_load1');
    }
    await calculateCG();
});
```



```
//setting all fuel arms to same value (for symmetric arms)
async function setFuelArms(arm_value) {
  let fuel_items = ['fuel_load1', 'fuel_load2', 'fuel_ground_burned1', 'fuel_ground_burned2', 'fuel_fli

  fuel_items.forEach( async (item) => {
    const arm_element = document.getElementById(`arm_${item}`);
    arm_element.value = arm_value;
    await update(`weight_items.${item}.arm`, Number(arm_element.value));
    await calculateMoment(item);
  });
}
```

mass_and_balance.html

```
31 | <!--BODY-->
32 | <div style="display: flex; margin-bottom: 10px; align-items: center; gap: 10px;">
33 |   <p>Fill Symmetric Arm values</p>
34 |   <label class="switch">
35 |     <input type="checkbox" id="symmetricArmCheckbox" {% if settings.fill_symmetric_arms%}checked{% endif %}>
36 |     <span class="slider round"></span>
37 |   </label>
38 | </div>
```

mass_and_balance.css

```
/*slider for symmetric arm toggle*/
.switch {
  position: relative;
  display: inline-block;
  width: 40px;
  height: 24px;
}

/*hiding default checkbox*/
.switch input {
  opacity: 0;
  width: 0;
  height: 0;
}

/*custom slider*/
.slider {
  position: absolute;
  cursor: pointer;
  top: 0;
  left: 0;
  right: 0;
  bottom: 0;
  background-color: var(--onhover);
  transition: .4s ease-out;
}

.slider:before {
  position: absolute;
  content: "";
  height: 16px;
  width: 16px;
  left: 4px;
  bottom: 4px;
  background-color: var(--highlight);
  transition: .4s ease-out;
}
```

```

input:checked + .slider {
  background-color: var(--primary);
}

input:checked + .slider:before {
  transform: translateX(16px);
}

/*rounding the slider*/
.slider.round {
  border-radius: 34px;
}

.slider.round:before {
  border-radius: 50%;
}

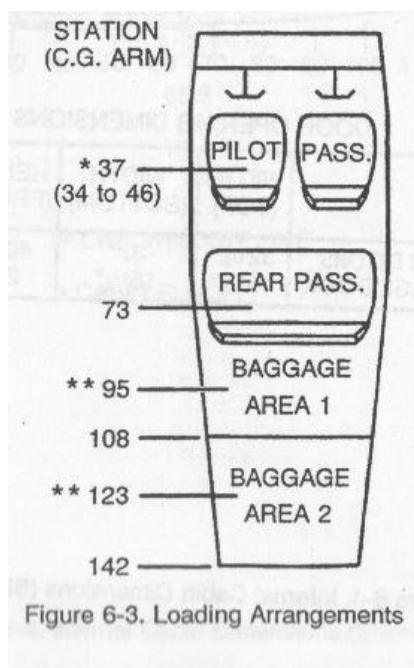
```

What I did

Pre-inputting values refers to the ability for the user to have pre-calculated values (i.e. fuel mass) put in the table automatically. I implemented this by simply adding to the `calculateTotals()` function, updating those specific cells to contain the relevant fuel data, divided by 2 and so they can be split equally across each fuel tank. For instance, to calculate the fuel burned in flight for each tank, calculate the fuel mass of the trip fuel and divide this by 2. It's then saved to the session so that when I load in the fuel section the values are already inputted into the table, and can even be edited if needed.

Symmetric arm values

This is a feature I decided to add to improve the usability of the moments table. Before I explain how I did it I'll briefly touch upon what I mean by symmetric arm values.



Most general aviation aircraft will have some things at an equal distance from the datum (arm) just due to their positioning.

Looking at the Cessna 172 loading arrangement diagram to the left, You can see that both the pilot and the passenger at the front have the same arm distance of 37 (Average given the front seats can move back and forth - hence a range given below)

The same occurs with the rear passenger seats, and often the fuel tanks which are often symmetrically placed on either side of the aircraft wing, and so are often considered to have the same arm value.

In this fashion I thought it would be useful to allow the user to have symmetric arm values filled automatically to save time typing in each cell. They can also turn off this feature and edit each cell independently, as not all positions are symmetric at all times at all stages of flight.

The first thing I did was to make a toggle switch to toggle the feature on/off. Following a W3Schools tutorial, the switch has 2 states of styling. Each is a small div with an even smaller one inside of it that imitates a circle due to the lack of content within. One style (.slider) has the slider to the left and background set to the onhover color. The other has the primary background color. Clicking the slider changes the styling, and also translates the slider div to the other side in a smooth fashion along with the different background color styling. The toggle is also accompanied with a new value 'fill_symmetric_arms' in **app.py**, which is true or false. The button toggles between the two.

With the setting off, editing cells work as normal. With the setting on, for the pilot and passenger positions, the other arm value is edited and updated in the session as well. As for the fuel, if one is edited I need all of them to edit, so I put all the suffixes of the items in an array, iterated through them and updated their values on the screen and in the session.

Why I did it this way

The addition of both these features have the aim of improving the user experience. The first aims to prevent the user from having to move back and forth from the fuel tab to the mass and balance tab to get/calculate fuel values to input into the table. Doing such a thing would be easily daunting and would discourage the user from using the webapp. The second may look small but saves the user about 30 seconds or so of inputting duplicate arm values in cells, which, from a user perspective, can be quite significant. Both of these features make using the webtool intuitive as features flow together, and the user can utilise these features efficiently and quickly. For instance, after filling in the fuel tab, the entire second table (with fuel values) in the mass and balance tab gets filled with the appropriate weight values, and all the user must do is enter one arm value to fill the rest and the entire 2nd table is filled with just one input!

Program output

 Pre-input and symmetric arm test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
132	Pre-inputted value test	When fuel values are entered, upon entering the mass and balance tab these values should be automatically added to the fuel values on the second table page.	PASS (See program output)
133	Symmetric arm autofill test	When symmetric arm values are turned on, then upon editing the arm of the pilot/PAX or fuel conditions, the arm value is duplicated to the other relevant cells.	PASS (See program output)

Iteration Review

Met requirements: N/A

Changes: Added pre-inputted values from fuel section and to optionally allow symmetrical arm values to be automatically filled out.

Next steps: Manoeuvring speeds and table highlighting

Iteration 7 - Manoeuvring speeds and table highlighting

mass_and_balance.js

```
//-----MANOEUVRING SPEEDS-----//

//manoeuvring speeds for takeoff and landing
async function calculateVa_takeoffAndLanding() {

  //ensure no calculation occurs if any input is missing
  if (takeoff_weight.textContent === '' || landing_weight.textContent === '' || MGW.value === '') {
    va_takeoff.textContent = '';
    va_landing.textContent = '';
    return;
  }

  //manoeuvring speed = Va @ max gross weight * sqrt(weight / max gross weight)
  const Va_takeoff = Number(va_MGW.value) * Math.sqrt(Number(takeoff_weight.textContent) / Number(MGW.value));
  const Va_landing = Number(va_MGW.value) * Math.sqrt(Number(landing_weight.textContent) / Number(MGW.value));

  await update('va_takeoff.value', Number(Va_takeoff));
  await update('va_landing.value', Number(Va_landing));

  //fetch output string from session to display
  const resp = await fetch('get-flight');
  const data = await resp.json();
  va_takeoff.textContent = data.va_takeoff.output;
  va_landing.textContent = data.va_landing.output;
}

//manoeuvring speed for any weight
async function calculateVa_any() {
  const Va_any = va_MGW.value * Math.sqrt(Any_Weight.value / MGW.value);
  await update('va_any.value', Number(Va_any));

  //fetch output string from session to display
  const resp = await fetch('get-flight');
  const data = await resp.json();
  const va_any_output = data.va_any.output;
  va_any.textContent = va_any_output;
}
```

```
//-----Checking weights limits
async function checkWeights() {
  //basic, zero-fuel and ramp weights should not exceed MGW
  const ground_weights = [basic_weight, zero_fuel_weight, ramp_weight];

  ground_weights.forEach((weight) => {
    if (weight.textContent !== '' && MGW.value !== '') {
      if (Number(weight.textContent) > Number(MGW.value)) {
        weight.parentElement.classList.add('active');
      } else {
        weight.parentElement.classList.remove('active');
      }
    } else {
      weight.parentElement.classList.remove('active');
    }
  });

  //takeoff weight should not exceed MTOW
  if (takeoff_weight.textContent !== '' && MTOW.value !== '') {
    if (Number(takeoff_weight.textContent) > Number(MTOW.value)) {
      takeoff_weight.parentElement.classList.add('active');
    } else {
      takeoff_weight.parentElement.classList.remove('active');
    }
  } else {
    takeoff_weight.parentElement.classList.remove('active');
  }

  //landing weight should not exceed MLW
  if (landing_weight.textContent !== '' && MLW.value !== '') {
    if (Number(landing_weight.textContent) > Number(MLW.value)) {
      landing_weight.parentElement.classList.add('active');
    } else {
      landing_weight.parentElement.classList.remove('active');
    }
  } else {
    landing_weight.parentElement.classList.remove('active');
  }
}
}
```

mass_and_balance.css

```
#CG_table td.active {
  background-color: hsla(0 100% 50% / 0.6);
  color: var(--bg);
}

#CG_table td {
  transition: 0.5s ease;
}
```

What I did

To calculate the manoeuvring speeds, I simply took the currently set takeoff and landing weights, as well as the user-entered max gross weights and manoeuvring speed at said max gross weight. With this information, I calculated the manoeuvring speed as follows:

$$V_A = V_{A(max)} \times \sqrt{\frac{W_{current}}{W_{max}}}$$

Then the result is updated in the session as required, and the output (i.e. with units) is fetched from the session to display on the screen.

The same applies for the 'any weight' manoeuvring speeds, except the input weight is that of the user and not that that was calculated in the CG_table.

For table highlighting, I tested to find that I can change the background colour of the <td> element of each cell to change the colors. In `checkWeights()`, an array containing all the weights of the ground is iterated through and each value in the cell is compared with the max gross weight, which it should not exceed. If this is the case, the 'active' class is added to the parent element (the <td> - the actual value is a inside this) which turns it red, with smooth fade transition. Takeoff and landing weights use max takeoff weight and max landing weight as comparisons and their cells will go red if they exceed them.

Why I did it this way

There is a very good reason why these manoeuvring speeds calculations, as well as the Centre of Gravity and moments calculations, are all done in javascript frontend instead of the flask backend. Simply put, items are processed client-side extremely quickly; much quicker than that of the server-side. What this means for the user is that if I decide to make calculations in the server then bring it back to the frontend to display, the solutions will appear much slower than they would if I did it in the frontend which is almost immediate. The approach I have taken is to perform calculations, have them displayed immediately, then update the session afterwards, which will be in the background, so the user won't even notice. This makes the process more fluid for the client as they don't need to wait for what may be a slow server to calculate values before they can be displayed on screen.

Program output

 Manoeuvring speed and table highlighting test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
134	Manoeuvring speed and table highlighting test	With the relevant values inputted, the manoeuvring speeds should be calculated and displayed on the screen. If weights exceeding a maximum weight are present in the CG table the relevant weight is highlighted in red.	PASS (See program output)

Iteration Review

Met requirements: (Aim complete) Allow the user to enter details related to manoeuvring speeds to produce manoeuvring speed values for takeoff and landing

(Aim complete) Allow manoeuvring speed to be calculated from a custom weight

(Aim complete) Have weights that go over the entered limits highlighted.

Changes: N/A

Next steps: AI change suggestions

Iteration 8 - AI change suggestion

mass_and_balance.html

```
18 |         <!--Suggest a change modal-->
19 |         <div id="suggestion" class="modal">
20 |             <div class="suggestionContent">
21 |                 <div style="display: flex; justify-content: center; align-items: center">
22 |                     <h3>Suggest a change</h3>
23 |                     <svg id="closepopup" fill="currentColor" width="20" height="20" stroke="#000000" v
24 |                 </div>
25 |                 <div id="output"></div>
26 |                 <p style="color: ■ #737373; margin-top: 15px;">This assessment is qualitative and is
27 |             </div>
28 |         </div>
29 |     <!--=====BODY=====-->
```

ai_roles.py

```
elif type == 'Mass_and_Balance':
    role = ''

INSTRUCTIONS:
The user's current Centre of Gravity (CG) is outside the allowable limits for safe flight. Your task is to suggest adjustments to bring the CG back within limits.
You must consider the effectiveness of changes alongside the operational penalty it could cause.

The hierarchy of adjustments is as follows. Each adjustment will have its associated CG effectiveness score and an operational penalty score.

low
medium
high
very high

Scores will be given as an array of two values, the first being the CG effectiveness score, the second being the operational penalty score.
Aim to suggest adjustments that have a high CG effectiveness score and a low operational penalty score.

A range of possible adjustments are given below. You are not limited to suggesting only one adjustment.

Moving people between rows/seats [high,low]
Moving baggage compartments [high,low]
Removing baggage [very high,high]
Removing rear passengers [very high,very high]
Removing front passengers [medium,very high]
Changing seat occupancy (solo vs dual) [medium,medium]
Fuel quantity adjustments [low,high]
Fuel distribution adjustments (if applicable) [low,medium]
You will be provided with lots of data about the current flight to help you make your suggestion. U
```



```

Are there multiple seating positions?
Is there baggage currently onboard?
Is fuel above minimum legal reserve?
Are there optional legs or tech stops?
Does a lever even exist to be suggested?
Many more of such considerations may be relevant.

OUTPUTTING METHOD

The suggestion should be output as ranked strategies, not commands. So if multiple adjustments are suggested
Each suggestion should include what changes, why it helps the CG, the tradeoffs and what the pilot should do
You MUST be explicit about any uncertainties or assumptions you are making in your suggestions.

Your output will be a JSON object with the following format:

{
  "suggestion": "string explaining your suggested adjustments. This is a single string and do not create
}

YOU MUST STRICTLY OBEY THE ABOVE FORMAT, NO COMMENTING, BRACKETS, OR ANYTHING OUTSIDE OF THIS FORMAT OF AN
Do not ask any questions when outputting as the user has no ability to respond.

'''

```

mass_and_balance.js

```

//-----AI suggestion

const output = document.getElementById("output");

async function suggestChange() {
  output.style.color = "■#737373";
  output.textContent = "Generating suggestion...";

  const response = await fetch('/get-flight');
  const flightData = await response.json();

  const promptText = `
  FLIGHT DATA:

  ROUTE:
  Departure Airport: ${flightData.departureAirport_name}
  Arrival Airport: ${flightData.destinationAirport_name}
  Alternate Airport: ${flightData.alternateAirport_name}
  Distance: ${flightData.distance.output}
  Time Enroute: ${flightData.time}
  Route: ${flightData.route_names}

  FUEL POLICIES:
  Trip: ${flightData.fuelPolicy_trip}
  Contingency: ${flightData.fuelPolicy_contingency}
  Alternate: ${flightData.fuelPolicy_alternate}
  Final Reserve: ${flightData.fuelPolicy_finalReserve}
  Additional: ${flightData.fuelPolicy_additional}

  FUEL VALUES:
  Taxi Fuel: ${flightData.fuel_taxi.output}
  Trip Fuel: ${flightData.fuel_trip.output}

```



```

MASS AND BALANCE:
MTOW: ${flightData.MTOW.output}
MLW: ${flightData.MLW.output}
MGW: ${flightData.MGW.output}

Weight and Balance Table:
${JSON.stringify(flightData.weight_items)}

CG Table:
${JSON.stringify(flightData.CG_calculations)}
`

const aiResponse = await prompt("Mass_and_Balance", promptText);
try {
  const resp = JSON.parse(aiResponse); //parsing the response to JSON
  output.style.color = "□#000000";
  console.log(resp);
  output.textContent = resp.suggestion
} catch (error) {
  console.log(error);
  output.style.color = "■#FF0000";
  output.textContent = "An error occurred while generating the suggestion. Please en

```

What I did

Before explaining the prompt, I'll briefly touch on how it will be laid out: upon pressing the button a popup will appear with a div containing the response. Immediately the response will begin to be generated and once it's done it'll be displayed for the user to read. It won't be saved to the session as the session is getting too large to store it (this may be an issue in the future).

For the prompt, each option is graded with a score for its effectiveness in changing the centre of gravity, and also the operational penalty it may have. A good example of this in practice is removing a passenger from the aircraft, which has a high CG effectiveness but also a high operational penalty. The AI is given a range of possible solutions with their associated scores, but it can also use its own suggestions too and derive its own scores so long as it can justify it. Decisions are made based on what is available, so within the prompt I asked the AI to consider whether they're passengers even available, or whether the fuel is below legal reserve- these must also be considered before even thinking about changing things to influence the CG. Finally, as always, the output is given as a JSON so I can simply parse, extract and display it on the screen. The prompt is paired with virtually all flight data except from the navlog which shouldn't be necessary to implement changes.

Why I did it this way

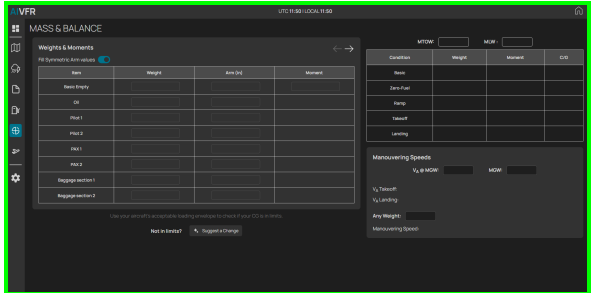
When it comes to suggestions, they must strictly be what they are: suggestions. Meaning that they shouldn't tell the pilot what to do, but rather what they **could do**. This is what I needed to consider carefully when creating the prompt for this feature. I could have made set outcomes based on a ton of conditions and research, but these would be hardcoded, and assuming that the user will fall into one of those categories, which is obviously not true, and it really depends on several factors that only the pilot knows best. So taking what the pilot has already entered and suggesting changes with that data is a much better approach, as it doesn't guide them into making incorrect decisions, but rather gives them a risk assessment of a range of possible options so that the correct solution can be developed solely by the user themselves.

Program output

AI_suggest a change test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
135	Suggesting a change generation test	Upon pressing the 'suggest a change' button, a popup should appear and moments after a generated response containing solutions to move the CG to a valid position should appear in the div element.	PASS (See program output)
136	Dark mode test: Mass and balance tab	Darkmode should apply the correct coloring to all elements.	

Iteration Review

Met requirements: Essential Feature Complete: Integration of AI for analysis & suggestions

(Aim complete) To have AI suggest a change the user could make should their CG value be out of limits

Section complete: Mass and balance tab

Changes: N/A

Next steps: Performance tab

Section Review

Aims completed in this section:

- **Complete an essential feature: Mass and balance, and flight performance calculations**
- **Complete a success criteria item: Fuel, Mass and balance, and Expense calculators**
- To allow the user to enter the weight and arm of each measurement item and have the moment calculated based on that information
- Calculate the total total weight, moment and centre of gravity at different stages of flight and display them to the user in a table
- Allow the user to enter details related to manoeuvring speeds to produce manoeuvring speed values for takeoff and landing
- Allow manoeuvring speed to be calculated from a custom weight
- Have weights that go over the entered limits highlighted.
- To have AI suggest a change the user could make should their CG value be out of limits

Aims not completed in this section:

N/A

Iterative tests complete:

- AI should output suitable responses in accordance to their given role and user prompt
- Light/dark Mode switches correctly with the correct colours

Remarks: Nothing to address.

Section 12: Performance Tab

Aims of this section

- Complete an **Essential feature: Mass and balance, and flight performance calculations**
- To allow the user to input their own performance figures, and have them saved into the session.

Iteration 1 - Performance Tab

app.py

```
#-----PERFORMANCE
"TODR" : 0,
"LDR" : 0,
"TODA" : 0,
"LDA" : 0,
```

performance.html

```
<!--=====HEADER=====-->
<div class="pageTitle" style="display: flex; align-items: center; gap: 10px;">
  <div id="title">PERFORMANCE</div>
  <span id="alertBox"></span>
</div>
<!--=====BODY=====-->
<div id="content">
  <div id="perf_inputs">
    <div class="perf_input"><b>Take-off Distance Required:</b> <input type="number" id="TODR" value="{{ flight.TODR if flight.TO
    <div class="perf_input"><b>Landing Distance Required: </b> <input type="number" id="LDR" value="{{ flight.LDR if flight.LDR
    <div class="perf_input"><b>Take-off Distance Available: </b> <input type="number" id="TODA" value="{{ flight.TODA if flight.TO
    <div class="perf_input"><b>Landing Distance Available:</b> <input type="number" id="LDA" value="{{ flight.LDA if flight.LDA
  </div>
  <div id="tooltip">
    <p style="color: #737373; margin-bottom: 5px;">TODA and LDA should be factored. All values in metres (m).</p>
    <p style="color: #737373">Due to the critical nature of performance values in flight, the ability to calculate performance
  </div>
```

performance.css

```
/*entire div*/
#perf_inputs {
  padding: 2%;
  background-color: var(--highlight);
  width: fit-content;
  border-radius: 15px;
}

#tooltip {
  position: relative;
  margin-top: 15px;
  width: 500px;
  left: 20px;
}

/*centering all content*/
#content {
  display: flex;
  flex-direction: column;
  align-items: center;
}

/*input div (text and box)*/
.perf_input {
  margin-bottom: 15px;
}

/*all input boxes*/
input {
  border: 1px solid var(--text);
  border-radius: 5px;
  padding: 5px;
  margin-left: 100px;
  width: 100px;
```

```

text-align: center;
font-weight: 600;
font-family: 'Manrope';
font-size: 13px;
background-color: var(--bg);
border: 1px solid var(--onhover);
}

```

performance.js

```

import { update, updateSettings } from "../basePage.js";
updateSettings("current_page", "/performance");

const todr = document.getElementById('TODR')
const ldr = document.getElementById('LDR')
const toda = document.getElementById('TODA')
const lda = document.getElementById('LDA')

//each input gets an event listener, and updates
//with the element's value when a change is detected

todr.addEventListener('change', async () => {
  update("TODR", todr.value)
})

ldr.addEventListener('change', async () => {
  update("LDR", ldr.value)
})

toda.addEventListener('change', async () => {
  update("TODA", toda.value)
})

lda.addEventListener('change', async () => {
  update("LDA", lda.value)
})

```

What I did

The performance tab is really simple, as it's just 4 inputs that need to be updated once changed, and it really doesn't go beyond that. I've mostly managed to style it as per design, and I added a single line to the tooltip for some clarity. Also, it's standard that performance distances are given in meters and so that's why this is the unit I have defaulted to. The event listeners look for a change in any of these boxes and take the current value, and update the session.

Why I did it this way

Here, the event listener does not listen for inputs but just simply a change, because this involves less code execution and there are no live updates necessary for this tab which may imply the requirement of listening out for any input rather than changes.

Program output

 Performance tab test.mp4

Take-off Distance Required:

1235

Landing Distance Required:

5678

Take-off Distance Available:

91011

Landing Distance Available:

1213

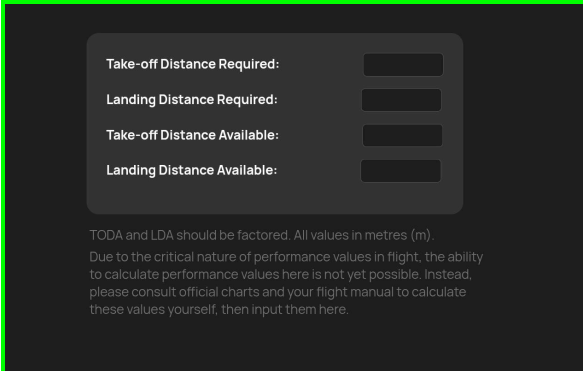
TODR and LDR should be factored. All values in metres (m).

Due to the critical nature of performance values in flight, the ability to calculate performance values here is not yet possible. Instead, please consult official charts and your flight manual to calculate these values yourself, then input them here.

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
137	Performance tab test	Entering values should update the performance values in the session.	PASS (See program output)

138	Dark mode test	Darkmode should apply the correct coloring to all elements.	 The screenshot shows a dark-themed user interface for flight performance calculations. It features four input fields: 'Take-off Distance Required:', 'Landing Distance Required:', 'Take-off Distance Available:', and 'Landing Distance Available:'. Below these fields, a disclaimer states: 'TODA and LDA should be factored. All values in metres (m). Due to the critical nature of performance values in flight, the ability to calculate performance values here is not yet possible. Instead, please consult official charts and your flight manual to calculate these values yourself, then input them here.'
-----	----------------	---	--

Iteration Review

Met requirements: Essential feature complete: Mass and balance, and flight performance calculations

(Aim complete) To allow the user to input their own performance figures, and have them saved into the session.

Changes: N/A

Next steps: Changelog

Section Review

Aims completed in this section:

- **Complete an Essential feature: Mass and balance, and flight performance calculations**
- To allow the user to input their own performance figures, and have them saved into the session.

Aims not completed in this section:

N/A

Iterative tests complete:

- Light/dark Mode switches correctly with the correct colours

Remarks: Nothing to address.

Adding a Changelog

CHANGELOG.md

```
1  ▾ ## Changelog
2
3      AIVFR uses [Semantic Versioning](https://semver.org/spec/v2.0.0.html).
4      Format is based on [Keep a Changelog](https://keepachangelog.com/en/1.1.0/).
5
6      ---
7  ▾ ### [0.7.1-alpha](https://github.com/TheEasyPilot/AIVFR/releases/tag/v0.7.1-alpha) - 2026-02-03 [YANKED]
8  ▾ #### Fixed
9
10     - Fixed changelog display bug caused by directory fetching issues
11     - updated log to include brackets where necessary
12
13  ▾ ### [0.7.0-alpha](https://github.com/TheEasyPilot/AIVFR/releases/tag/v0.7.0-alpha) - 2026-02-03
14
15  ▾ #### Added
16
17     - Added changelog
18     - Added plane silhouette backdrop to main menu
19     - Current functionality: Menu, settings, route setup, WX, navlog, fuel, mass and balance and performance tabs
20
21  ▾ ### [0.1.0-alpha](https://github.com/TheEasyPilot/AIVFR/releases/tag/v0.1.0-alpha) - 2025-12-29
22
23  ▾ #### Changed
24
```

menu.html

```
<!--changelog-->
<div id="changelog">
  Loading changelog... <!--when changelog isn't loaded yet this is shown-->
  <script>
    fetch('/CHANGELOG.md')
      .then(response => response.text())
      .then(text => {
        //converting markdown to HTML using marked.js
        document.getElementById('changelog').innerHTML = marked.parse(text);
      })
      .catch(error => {
        document.getElementById('changelog').innerHTML = 'Failed to load changelog.';
        console.error('Error loading changelog:', error);
      });
  </script>
</div>

<!--plane silhouette-->

</body>
```

main-menu.css

```
#bluePlaneLogo{
  position: fixed;
  bottom: -320px;
  left: 330px;
  width: 85%;
  opacity: 0.15;
  z-index: -1;
```



```

}
/*-----Changelog-----*/

#changelog {
  position: absolute;
  top: 100px;
  right: 250px;
  width: 500px;
  max-height: 400px;
  overflow-y: auto;
  background-color: var(--highlight);
  padding: 15px;
  border-radius: 10px;
  box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
  font-size: 14px;
  font-family: 'Manrope';
}

/*adding media query to ensure changelog doesn't cover title when screen is small*/
@media (max-width: 1200px) {
  #changelog {
    position: static;
  }
}

```

What I did

The trigger to develop this was because the main menu looked really empty, and I knew I wanted this feature already, but I didn't know where I'd implement it until now. The blue plane backdrop was also a design choice and the CSS is above, but that is not the focus.

The changelog uses a changelog format from [keep a changelog](#), which was very helpful in explaining how a changelog should look and how it can be consistent with different versions of the software. Both github and keep a changelog recommend me use semantic versioning, which, from my understanding, is an attempt to standardise, or at least improve the way developers release versions of their work. And that's it really, the changelog follows this format and can be seen above! Each commit will include the changes and a new release containing the changelog extract for that specific version.

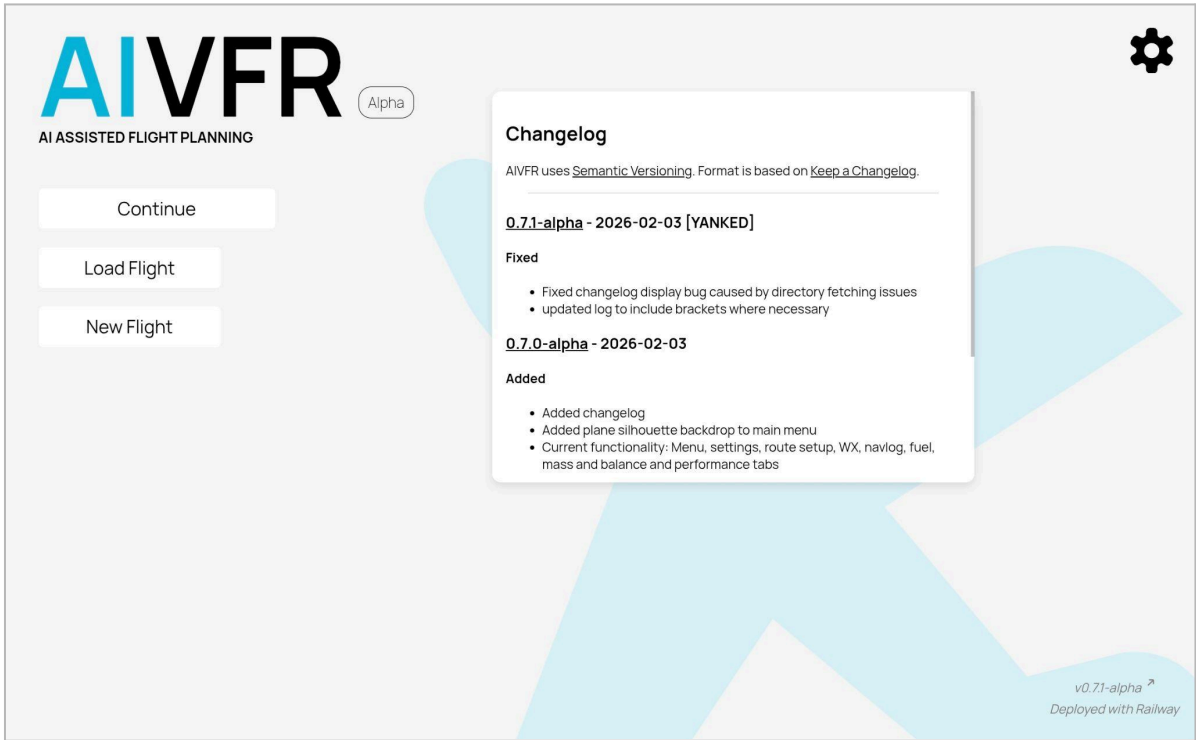
The changelog is written in markdown, so that it can be displayed nicely on github, but I needed to display it on the main menu as well. To do this, I used [marked.js](#) to turn the fetched markdown file into html similar to what is seen on github (although github does this much better). A try-catch block will show an error in the div if the changelog can't be fetched, which will most likely be due to a network error, or the corruption of the markdown file.

Why I did it this way

Obviously, this software doesn't really need a changelog at this stage, but from a development standpoint, it's really essential, especially because I may develop this further considering how invested I am into it already. Not having this changelog and adding features (and still progressing version numbers) makes it harder to deal with when I actually will start needing a changelog, so implementing it now not only makes it more

professional but will actually help me in the long run when releasing updates to this software should I choose to do so.

Program output



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
139	Changelog display	Changelog should be displayed and be legible in a div to the left of the main menu	

Iteration Review

Met requirements: N/A

Changes: Added a changelog and backdrop design to the main menu

Next steps: Dashboard

Section 13: Dashboard

Aims of this section

- **Complete 3 success criteria items: Dashboard, Full flight brief, Fuel, Mass and balance, and [Expense calculators](#)**
- Display Route information:
 - Departure → Arrival
 - Time and distance
 - Alternate aerodrome
- Display aircraft weight and performance:
 - Persons on board
 - Cargo weight
 - Fuel
 - Total (ramp)
 - MTOW, MLW
 - TOD, LD
- Display route information:
 - Route
 - Cruise and groundspeed
 - Alternate details
- Display weather reports
- Allow the user to add expenses to their flight
- Make the AI generate a briefing for the entire flight
- Allow all flight details to be downloaded as a pdf

Iteration 1 - HTML initial layout and styling

dashboard.css

```
/*content*/

#content {
  display: grid;
  grid-template-columns: 1fr 1.2fr 1.3fr;
  gap: 15px;
  margin-top: 20px;
}

#route_information {
  grid-column: 1;
  grid-row: 1;
  min-height: 200px;
}

#weight_and_performance {
```

```

    grid-column: 1;
    grid-row: 2;
    min-height: 200px;
}

#route_details {
    grid-column: 1;
    grid-row: 3;
    min-height: 300px;
}

#weather_info {
    grid-column: 2;
    grid-row: 1 / span 2;
}

#expenses {
    grid-column: 2;
    grid-row: 3;
}

#brief {
    grid-column: 3;
    grid-row: 1 / span 3;
}

.dashboard_content {
    background-color: var(--highlight);
    border-radius: 10px;
    padding: 15px;
}

```

dashboard.html

```

<!--=====HEADER=====-->
<div id="header" display="flex">
  <div id="title">DASHBOARD</div>
  <span id="alertBox"></span>
  <button id="save">SAVE FLIGHT</button>
  <!--<button id="save" class="{ 'savingOrSaved' if flight.saved == 'True' else '' }">{{ 'SAVED'
</div>
<!--=====BODY=====-->
<div id="content">
  <!--route information-->
  <div id="route_information" class="dashboard_content">
    <!--header-->
    <div id="route_info_header" style="display: flex; align-items: center; gap: 10px;">
      <svg style="width: 25px;" viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg"><g><path d="M
      <h3>Route information</h3>
    </div>
  </div>

  <!--weight and performance-->
  <div id="weight_and_performance" class="dashboard_content">
    <!--header-->
    <div id="weight_and_performance_header" style="display: flex; align-items: center; gap: 10px;">
      <svg style="width: 25px;" viewBox="0 0 512 512" xmlns="http://www.w3.org/2000/svg"><path fill=
      <h3>Weight and performance</h3>
    </div>
  </div>

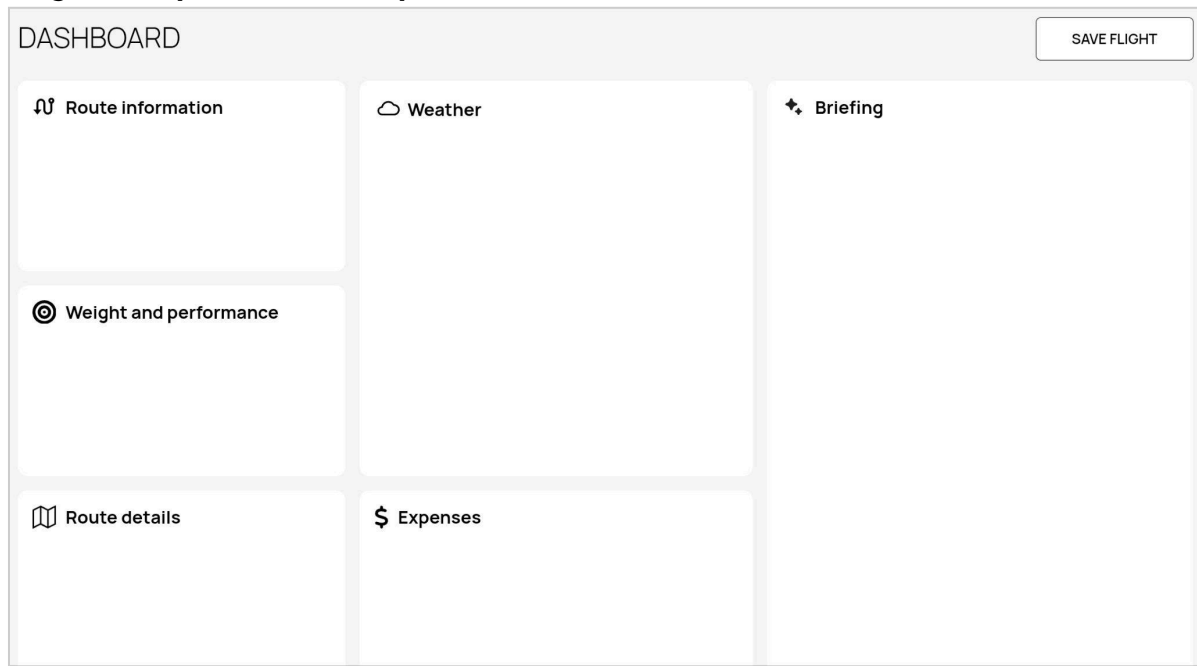
  <!--route details-->
  <div id="route_details" class="dashboard_content">
    <!--header-->

```

What I did

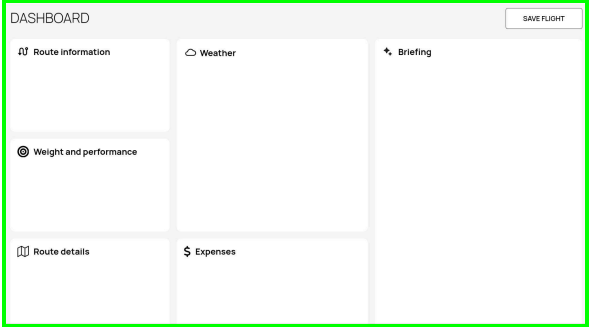
Once again, each section is arranged into a grid-like format as per design. This time I never needed to split into two grids because there are no overlaps, so I simply created the divs and organised them in the grid format as required. Each item has a header currently containing the title and the relevant icon. For some buttons the header will contain other items too.

Program output (not whole vp)



Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
140	Grid layout test	Grid layout should match the layout for the dashboard in the design.	 A screenshot of the dashboard grid layout test. The dashboard is shown with a green border around the main content area. The layout matches the design shown in the previous image, with sections for Route information, Weather, Briefing, Weight and performance, Route details, and Expenses.

Iteration Review

Met requirements: N/A

Changes: N/A

Next steps: Finishing page design

Iteration 2 - Finishing page design and styling

dashboard.html

```
<div id="content">
  <!--ROUTE INFORMATION-->
  <div id="route_information" class="dashboard_content">
    <!--header-->
    <div id="route_info_header" style="display: flex; align-items: center; gap: 10px;">
      <svg style="width: 25px;" viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg"><g><path
      <h3>Route information</h3>
    </div>
    <!--content-->
    <div id="route_info_content" style="font-size: 15px; margin-top: 15px;">
      <p><b>ABCD</b> Depart Airport → <b>EFGH</b> Arrival Airport</p>
      <p style="margin-top: 5px;"><b>X</b> minutes | <b>Y</b> NM</p>
      <p style="margin-top: 5px;"><b>Alternate:</b> <b>IJKL</b> Alternate Airport</p>
    </div>
  </div>

  <!--WEIGHT AND PERFORMANCE-->
  <div id="weight_and_performance" class="dashboard_content">
    <!--header-->
    <div id="weight_and_performance_header" style="display: flex; align-items: center; gap: 10px">
      <svg style="width: 25px;" viewBox="0 0 512 512" xmlns="http://www.w3.org/2000/svg"><path f
      <h3>Weight and performance</h3>
    </div>
    <!--content-->
    <p><b>Aircraft:</b><input style="margin-left: 10px; width: 150px; margin-top: 15px;" type="t
    <div id="weight_and_performance_content" style="font-size: 15px; margin-top: 5px;">
      <div class="WP_content" style="margin-top: 15px;">
        <p><b>POB: </b>X</p>
        <p><b>Cargo: </b>X</p>
        <p><b>Fuel: </b>X</p>
      </div>
      <div class="WP_content">
        <p><b>Ramp: </b>X</p>
        <p><b>TOW: </b>X</p>
        <p><b>LW: </b>X</p>
      </div>
      <div class="WP_content">
        <p><b>TODR: </b>X</p>
        <p><b>LDR: </b>X</p>
      </div>
    </div>
  </div>
</div>
```

```

<!--ROUTE DETAILS-->
  <div id="route_details" class="dashboard_content">
    <!--header-->
      <div id="route_details_header" style="display: flex; align-items: center;
      <svg style="width: 25px" id="Layer_1" style="enable-background:new 0 0
      <h3>Route details</h3>
      </div>
    <!--content-->
      <div id="route_details_content">
        <div id="route_names">
          ABCD → WAYPOINT1 → WAYPOINT2 → DEFG
        </div>
        <div style="display: flex; gap: 20px; margin-top: 15px;">
          <p style="margin-bottom: 10px;"><b>Cruise: </b>xxxxft</p>
          <p><b>GS: </b>X</p>
        </div>
        <hr>

        <p style="margin-bottom: 15px;"><b>Alternate</b> (Alternate)</p>
        <p style="margin-bottom: 5px;">Distance from [arrival]: <b>X</b></p>
        <p style="margin-bottom: 5px;">Fuel Required: <b>X</b></p>
        <p style="margin-bottom: 5px;">ETA to alternate: <b>X</b></p>
      </div>
    </div>
  </div>

```

```

<!--WEATHER INFORMATION-->
  <div id="weather_info" class="dashboard_content">
    <!--header-->
      <div id="weather_info_header" style="display: flex; justify-content: space-between;" >
        <div style="display: flex; align-items: center; gap: 5px;">
          <svg style="width: 30px" viewBox="0 0 512 512" xmlns="http://www.w3.org/2000/svg"><title></title></svg>
          <h3>Weather</h3>
        </div>
        <button class="button" id="refresh_WX"><svg style="width: 25px" xmlns="http://www.w3.org/2000/svg"><title></title></svg>
        </div>
    <!--content-->
      <!--departure weather-->
      <div id="departure_WX">
        <!--header-->
        <div id="departure_WX_header" style="display: flex; align-items: center; gap: 10px;">
          <b>ABCD</b> Depart Airport
          <p><b>GRAD</b></p>
        </div>
        <!--metar-->
        <div id="departure_WX_metar" class="WX_metar" style="margin-bottom: 20px;">
          <p><b>METAR ABCD 010000Z 00000KT 9999 FEW020 25/15 Q1013</b></p>
        </div>
      </div>

      <!--arrival weather-->
      <div id="arrival_WX">

```

```

<!--EXPENSES-->
<div id="expenses" class="dashboard_content">
  <!--header-->
  <div id="expenses_header" style="display: flex; align-items: center; gap: 10px; justify-content: space-between;">
    <div style="display: flex; align-items: center; gap: 15px;">
      <svg style="width: 15px" viewBox="0 0 320 512" xmlns="http://www.w3.org/2000/svg"><path fill="currentColor" d="M302 492c-16 0-29 13-29 29s13 29 29 29 29-13 29-29s-13-29-29-29z" />
      <h3>Expenses</h3>
    </div>
    <button class="button" id="add_expense"><svg style="width: 30px" viewBox="0 0 256 256" xmlns="http://www.w3.org/2000/
  </div>
  <!--content-->
  <div id="expenses_content">
    <ul>
      <li class="expense">Expense 1: <b>EX</b></li>
      <li class="expense">Expense 2: <b>EX</b></li>
      <li class="expense">Expense 3: <b>EX</b></li>
    </ul>
  </div>
  <div id="total_expense" style="display: flex; justify-content: center; margin-bottom: 10px;">
    <p>Total Flight Cost: <b>EX</b></p>
  </div>
</div>

<!--AI BRIEFING-->
<div id="brief" class="dashboard_content">
  <!--header-->
  <div id="brief_header" style="display: flex; align-items: center; gap: 10px; justify-content: space-between;">
    <div style="display: flex; align-items: center; gap: 15px;">
      <svg fill="currentColor" height="25px" viewBox="0 0 20 20" width="25px" xmlns="http://www.w3.org/2000/svg"><path d=
      <h3>Briefing</h3>
    </div>
    <div style="display: flex; gap: 5px;">
      <button class="button" id="refresh_brief"><svg height="24" viewBox="0 0 24 24" width="24" xmlns="http://www.w3.org/
      <button class="button" id="refresh_WX"><svg style="width: 25px" xmlns="http://www.w3.org/2000/svg" xmlns:xlink="htt
    </div>
  </div>
  <!--content-->
  <div id="AI_breif_output">
    <p>Lorem ipsum dolor, sit amet consectetur adipisicing elit. Libero error vitae, nostrum minima natus explicabo dolor
  </div>
</div>
</div>

```

dashboard.css

```

/*content styling*/

/*route information*/
#route_names {
  font-family: 'Manrope';
  font-size: 14px;
  background-color: var(--bg);
  padding: 10px;
  margin-top: 15px;
  border-radius: 10px;
  min-height: 70px;
}

#refresh_WX, #refresh_brief {
  background-color: var(--bg);
  width: 40px;
  height: 38px;
}

#refresh_WX:hover, #refresh_brief:hover {
  background-color: var(--onhover);
  cursor: pointer;
}

```



```

/*weather info*/
.wx_metar {
    background-color: var(--bg);
    padding: 10px;
    border-radius: 5px;
    margin-top: 5px;
}

/*expenses*/
#add_expense {
    background-color: var(--bg);
    width: 40px;
    height: 39px;
}

#add_expense:hover {
    background-color: var(--onhover);
    cursor: pointer;
}

#expenses_content {
    position: relative;
    left: 20px;
    margin-bottom: 10px;
    height: calc(100% - 100px);
}

.expense{
    margin-bottom: 10px;
}

#total_expense {
    position: relative;
    background-color: var(--bg);
    padding: 10px;
    border-radius: 10px;
}

/*brief*/
#AI_brief_output {
    background-color: var(--bg);
    padding: 20px;
    border-radius: 10px;
    height: calc(100% - 48px);
    overflow-y: auto;
}

```

What I did

The main aim of this part was to place all the CSS and HTML required to complete the page, so that I can simply move on to replacing the placeholder text with actual values from the session. Each little section of the dashboard is coded with the intention of matching the design, and for the most part, this worked well. An additional feature was to add an 'Aircraft' input where the user can enter the type of aircraft they'll be flying, although this doesn't yet influence any values at the moment.

Why I did it this way

An alternative way I could have approached this was to focus on each section one by one, filling out the HTML, CSS and JS before moving on to the next. I decided not to do this because it would require me to move back and forth between the HTML build file and the actual dashboard with Jinja2, which makes it easier to make annoying errors when copying and pasting values from one file to another. Also it is nicer from a developer perspective to see the entire tab as it would be final, as it makes it easier to make small changes and edits before moving onto the script. Now, since I've done it this way, I no longer need to worry about any styling issues as I've got it sorted out now, and only scripting is required, which means less movements between files and therefore a faster development process.

Program output

DASHBOARD

SAVE FLIGHT

📍 Route information

ABCD Depart Airport → EFGH Arrival Airport
X minutes | Y NM
Alternate: IJKL Alternate Airport

⚖️ Weight and performance

Aircraft:

POB: X Cargo: X Fuel: X
Ramp: X TOW: X LW: X
TODR: X LDR: X

📖 Route details

ABCD → WAYPOINT1 → WAYPOINT2 → DEFG

Cruise: xxxft GS: X

Alternate (Alternate)
Distance from [arrival]: X
Fuel Required: X
ETA to alternate: X

☁️ Weather

ABCD Depart Airport GRAD
METAR ABCD 010000Z 00000KT 9999 FEW020 25/15 Q1013

DEFG Arrival Airport GRAD
METAR DEFG 010000Z 00000KT 9999 FEW020 25/15 Q1013

IJKL Alternate Airport GRAD
METAR IJKL 010000Z 00000KT 9999 FEW020 25/15 Q1013

\$ Expenses

• Expense 1: £X
• Expense 2: £X
• Expense 3: £X

Total Flight Cost: £X

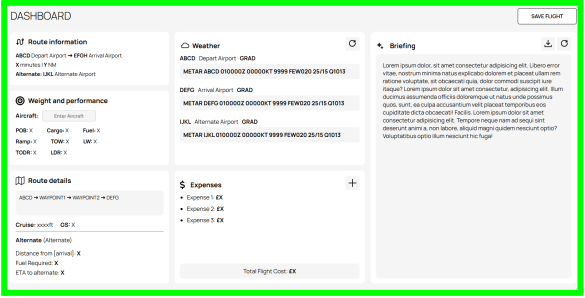
✈️ Briefing

Download Refresh

Lorem ipsum dolor, sit amet consectetur adipisicing elit. Libero error vitae, nostrum minima natus explicabo dolore et placeat ullam rem ratione voluptate, sit obcaecati quia, dolor commodi suscipit iure itaque? Lorem ipsum dolor sit amet consectetur, adipisicing elit. Illum ducimus assumenda officis doloreque ut natus unde possimus quos, sunt, ea culpa accusantium velit placeat temporibus eos cupiditate dicta obcaecati! Facilis. Lorem ipsum dolor sit amet consectetur adipisicing elit. Tempore neque nam ad sequi sint deserunt animi a, non labore, aliquid magni quidem nesciunt optio? Voluptatibus optio illum nesciunt hic fuga!

Known Limitations / Missing features

- No JS

Test No.	Test Description	Expected Outcome	Actual Outcome
141	Content styling test	Full HTML and CSS for the dashboard should be complete and matching the design.	

Iteration Review

Met requirements: N/A

Changes: Added aircraft input

Next steps: Route information & Weight and performance sections.

Iteration 3 - Filling data not requiring JS

dashboard.html

```
<!--=====BODY=====-->
<div id="content">
  <!--ROUTE INFORMATION-->
  <div id="route_information" class="dashboard_content">
    <!--header-->
    <div id="route_info_header" style="display: flex; align-items: center; gap: 10px;">
      <svg style="width: 25px;" viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg"><g><path d="M0 0h24v24H0z" fill="none"/><path d="M4 15
      <h3>Route information</h3>
    </div>
    <!--content-->
    <div id="route_info_content" style="font-size: 15px; margin-top: 15px;">
      <p><b>{{ flight.departureAirport_code }}</b> {{ flight.departureAirport_name }} → <b>{{ flight.destinationAirport_code }}</b> {{ flight
      <p style="margin-top: 5px;">{{ flight.time }} <b></b> {{ flight.distance.output }}</p>
      <p style="margin-top: 5px;"><b>Alternate:</b> <b>{{ flight.alternateAirport_code }}</b> {{ flight.alternateAirport_name }}</p>
    </div>
  </div>

  <!--WEIGHT AND PERFORMANCE-->
  <div id="weight_and_performance" class="dashboard_content">
    <!--header-->
    <div id="weight_and_performance_header" style="display: flex; align-items: center; gap: 10px;">
      <svg style="width: 25px;" viewBox="0 0 512 512" xmlns="http://www.w3.org/2000/svg"><path fill="currentCo
      <h3>Weight and performance</h3>
    </div>
    <!--content-->
    <p><b>Aircraft:</b><input style="margin-left: 10px; width: 150px; margin-top: 15px;" type="text" placeholder="Aircraft" /></p>
    <div id="weight_and_performance_content" style="font-size: 15px; margin-top: 5px;">
      <div class="WP_content" style="margin-top: 15px;">
        <p><b>POB: </b><b>X</b></p>
        <p><b>Cargo: </b><b>X</b></p>
        <p><b>Fuel: </b>{{ flight.fuel_mass.output }}</p>
      </div>
      <div class="WP_content">
        <p><b>Ramp: </b>{{ flight.CG_calculations.ramp_condition.weight.output }}</p>
        <p><b>TOW: </b>{{ flight.CG_calculations.takeoff_condition.weight.output }}</p>
        <p><b>LW: </b>{{ flight.CG_calculations.landing_condition.weight.output }}</p>
      </div>
      <div class="WP_content">
        <p><b>TODR: </b>{{ flight.TODR }}m</p>
        <p><b>LDR: </b>{{ flight.LDR }}m</p>
        <p><b>Endurance: </b>{{ flight.fuel_endurance }}</p>
      </div>
    </div>
  </div>
</div>
```

```

<!--ROUTE DETAILS-->
<div id="route_details" class="dashboard_content">
  <!--header-->
  <div id="route_details_header" style="display: flex; align-items: center; gap: 10px;">
    <svg style="width: 25px" id="Layer_1" style="enable-background:new 0 0 128 128;" version="1.1" viewBox="0 0 128 128" xml:space="preserve">
    <h3>Route details</h3>
  </div>
  <!--content-->
  <div id="route_details_content">
    <div id="route_names">{{ ' ' → '.join(flight.route_names) if flight.route_names else '' }}</div>
    <div style="display: flex; gap: 20px; margin-top: 15px;">
      <p style="margin-bottom: 10px;"><b>Cruise: </b>xxxxft</p>
      <p><b>GS: </b>X</p>
    </div>
    <hr>
    <p style="margin-bottom: 15px;"><b>{{ flight.alternateAirport_code }}</b> {{ flight.alternateAirport_name }} (Alternate)</p>
    <p style="margin-bottom: 5px;">Distance from {{ flight.destinationAirport_code }}: <b>{{ flight.NAVLOG.rows[-1]["DIST (NM)"].value }}NM</b>
    <p style="margin-bottom: 5px;">Fuel Required: <b>{{ flight.fuel_alternate.output }}</b></p>
  </div>
</div>

```

What I did

First, I decided to fill in all the flight data that just required me to access the data in the session via Jinja2 ({{ stuff in these brackets }}), as it is very easy to integrate those into the HTML. This worked out for all route information parts, most of the weight and performance data, and most of the route details. The rest of the data requires some level of JS to integrate, which is why I've saved it for later. I also added the endurance in the weight and performance section as I considered it important enough to be placed there.

Why I did it this way

If I went on to tackle each section one by one (i.e. including the javascript), It would be unlikely to make a huge difference in time, but it could possibly be more complex from a developer perspective, as I would have to work with easy less complex things and more complex algorithms at once. I find it much easier to work with increasing difficulty, in that I complete the easier actions first, and then move on to the harder ones, not mixing the two.

Program output


Route information

EGSX NORTH WEALD → EGKA SHOREHAM
 0hrs 56mins | 61.9 NM
 Alternate: EGHR CHICHESTER/GOODWOOD


Weight and performance

Aircraft:
 POB: X Cargo: X Fuel: 31.9 kg
 Ramp: 1357.0 kg TOW: 1355.6 kg LW: 1340.8 kg
 TODR: 1205m LDR: 1286m Endurance: 2hrs 0mins


Route details

EGSX → BANBURY RESERVOIR → A25 GODSTONE JUNCTION
 → HANDCROSS → WASHINGTON INTERSECTION → EGKA
 Cruise: xxxxft GS: X
 EGHR CHICHESTER/GOODWOOD (Alternate)
 Distance from EGKA: 17.6NM
 Fuel Required: 3.3 L


Weather

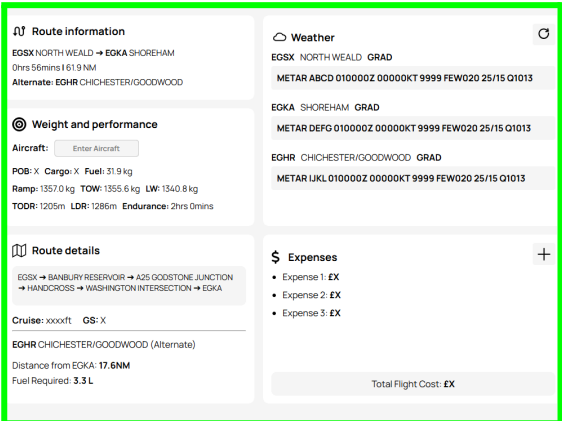

EGSX NORTH WEALD GRAD
 METAR ABCD 010000Z 00000KT 9999 FEW020 25/15 Q1013
 EGKA SHOREHAM GRAD
 METAR DEFG 010000Z 00000KT 9999 FEW020 25/15 Q1013
 EGHR CHICHESTER/GOODWOOD GRAD
 METAR IJKL 010000Z 00000KT 9999 FEW020 25/15 Q1013


Expenses


• Expense 1: £X
 • Expense 2: £X
 • Expense 3: £X
 Total Flight Cost: £X

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
142	Date presentation test (jinja2 only)	Upon inputting some flight data, and calculating fuel, perf etc, upon returning to the dashboard the necessary data should appear in the sections where jinja2 has been implemented.	 The screenshot shows a flight dashboard with the following sections: Route information: EGSX NORTH WEALD → EGKA SHOREHAM, 0hrs 56mins 161.9 NM, Alternate: EGHR CHICHESTER/GOODWOOD. Weight and performance: Aircraft: [Enter Aircraft], POB: X, Cargo: X, Fuel: 31.9 kg, Ramp: 1357.0 kg, TOW: 1355.6 kg, LW: 1340.8 kg, TODR: 1205m, LDR: 1286m, Endurance: 2hrs 0mins. Route details: EGSX → BANBURY RESERVOIR → A25 GOODSTONE JUNCTION → HANDCROSS → WASHINGTON INTERSECTION → EGKA, Cruise: xxxoft, GS: X, EGHR CHICHESTER/GOODWOOD (Alternate), Distance from EGKA: 17.6NM, Fuel Required: 3.3 L. Weather: EGSX NORTH WEALD GRAD, METAR ABCD 010000Z 00000KT 9999 FEW020 25/15 Q1013, EGKA SHOREHAM GRAD, METAR DEFG 010000Z 00000KT 9999 FEW020 25/15 Q1013, EGHR CHICHESTER/GOODWOOD GRAD, METAR IJKL 010000Z 00000KT 9999 FEW020 25/15 Q1013. Expenses: Expense 1: £X, Expense 2: £X, Expense 3: £X, Total Flight Cost: £X.

Iteration Review

Met requirements:

(Aim complete) Display Route information:

- Departure → Arrival
- Time and distance
- Alternate aerodrome

Display aircraft weight and performance:

- Persons on board
- Cargo weight
- Fuel
- Total (ramp)
- MTOW, MLW
- TOD, LD

Display route information:

- Route
- Cruise and groundspeed
- Alternate details

Changes: Added endurance data

Next steps: Forming values that need JS

Iteration 4 - JS required values: POB, Cargo, Cruise, Average GS

dashboard.js

```
//-----WEIGHT AND PERFORMANCE-----  
//---aircraft input  
const aircraft = document.getElementById("aircraft");  
  
aircraft.addEventListener("change", () => {  
  update("aircraft", aircraft.value);  
})  
  
//---determining and updating POB (persons on board)  
  
let pob = 1; //assume at least the pilot is on board...  
  
//if we have weight values for pilot2, PAX1 and PAX2, we can assume they are occupied  
if (data.flight.weight_items.pilot2.weight.value != 0) {  
  pob += 1;  
} if (data.flight.weight_items.PAX1.weight.value != 0) {  
  pob += 1;  
} if (data.flight.weight_items.PAX2.weight.value != 0) {  
  pob += 1;  
}  
  
const POB = document.getElementById("POB");  
POB.innerHTML = `<b>POB: </b>${pob}`;  
  
//-----ROUTE DETAILS-----  
//---determining the cruise altitude  
let cruiseAltitude = 0;  
  
//iterate through the NAVLOG rows to find the highest planned altitude, which we can assume is the cruise altitude  
for (const row_index in data.flight.NAVLOG.rows) {  
  const altitude = data.flight.NAVLOG.rows[row_index]["ALT PLAN (FT)"].value;  
  if (altitude > cruiseAltitude) {  
    cruiseAltitude = altitude;  
  }  
}  
  
if (data.settings.units_altitude == "metre") {  
  cruiseAltitude = Math.round(cruiseAltitude * 0.3048); //convert feet to meters  
}  
  
const cruise_alt = document.getElementById("cruise_alt");  
  
//using a tertiary when updating to put the relevant units for altitude based on the user's settings  
cruise_alt.innerHTML = `<b>Cruise: </b>${cruiseAltitude} ${data.settings.units_altitude == "feet" ? 'ft' : 'm'}`;
```

mass_and_balance.js

```
//-----Calculating cargo weight  
async function calculateCargo() {  
  //determining and updating cargo weight for dashboard  
  let cargoWeight = 0;  
  
  if (Number(weight_baggage1.value) != 0) {  
    cargoWeight += Number(weight_baggage1.value);  
  } if (Number(weight_baggage2.value) != 0) {  
    cargoWeight += Number(weight_baggage2.value);  
  }  
  await update("cargo.value", cargoWeight);  
}
```

navlog.js

```
//-----CALCULATING AVG GROUND SPEED
async function calculateAvgGS() {

  const response = await fetch('/get-flight')
  const data = await response.json();

  let totalGS = 0;
  let countGS = 0;

  //taking the groundspeed values from each row and calculating average
  for (const row_index in data.NAVLOG.rows) {
    const gs = data.NAVLOG.rows[row_index]["GS (KT)"].value;
    if (gs != 0) { //only include non-zero ground speeds in the average
      totalGS += gs;
      countGS += 1;
    }
  }
  const avgGS = Math.round(totalGS / countGS);
  await update("average_airspeed.value", avgGS);
}
```

app.py

```
#-----DASHBOARD
"aircraft" : "",
"cargo" : {"value" : 0, "class" : "mass"},
"average_airspeed" : {"value" : 0, "class" : "airspeed"},
```

dashboard.html

```
<p id="POB"><b>POB: </b>...</p>
<p id="cargo"><b>Cargo: </b>{{ flight.cargo.output }}</p>

"cruise_alt" style="margin-bottom: 10px;"><b>Cruise: </b>...ft</p>
"groundspeed" style="margin-bottom: 10px;"><b>GS (Average): </b>{{ flight.average_airspeed.output }}</p>
```

What I did

To determine the persons on board, I simply calculated the amount of weight items that are for occupants (so pilot1 and 2, PAX1 and 2) whose weight values are not empty. If this is the case I can assume that there is an occupant, and I increment the POB value. At the end of this, the final POB value is displayed. To determine the cargo weight, I needed the total weight of the item in both baggage sections. These two values are added together and updated in the session so that when the dashboard is opened, the total cargo weight is displayed.

To determine the cruise altitude, I have taken the highest planned altitude value in the navigation log, as this is often the cruise altitude. The algorithm iterates through each row, looking at the 'ALT PLAN (FT)' cell, and compares it with the value stored in `cruiseAltitude`. The value will be replaced if it is higher than what is stored. The final values then output on the screen, and will be converted to metres if the units are set to metres on page load.

Finally, to calculate the average groundspeed, I stored the count of all (non-zero) GS values as well as their sum to calculate the average. These values are found in the same iterative process as for calculating the cruise altitude.

Why I did it this way

An alternative I briefly considered to calculate the cruise altitude was to take the mode of the altitude as often that corresponds to the cruise altitude (As the aircraft stays at this altitude longest). This became not feasible because short trips may only have two legs at cruise and more at a lower altitude, which now makes it a problem as the mode will appear to be the lower altitude. Very rarely will the highest altitude chosen not be the cruise altitude and so I deemed it better to use the highest altitude rather than the most frequently appearing one.

Program output

 weight&perf and flight details JS test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
143	Weight and performance and flight details test	Upon entering necessary flight details, data for both these sections should be calculated and displayed accurately on the dashboard.	PASS (See program output)

Iteration Review

Met requirements:

(Aim complete)

Display aircraft weight and performance:

- Persons on board
- Cargo weight

Display route information:

- Cruise and groundspeed

Changes: NA

Next steps: Weather reports

Iteration 5 - Current weather reports

dashboard.js

```
//-----WEATHER-----
//-----initializing grading colours and styling
const grading = document.getElementsByClassName('grading');

async function initializeGradingColors() {
  for (let i = 0; i < grading.length; i++) {
    if (grading[i].textContent == "VFR") {
      grading[i].style.color = "■ #40d400";
    } else if (grading[i].textContent == "MVFR") {
      grading[i].style.color = "■ #0091ff";
    } else if (grading[i].textContent == "IFR") {
      grading[i].style.color = "red";
    } else if (grading[i].textContent == "LIFR") {
      grading[i].style.color = "■ #800080";
    }
  }
}

async function getWeather(code) {
  if (code === "") {
    return undefined;
  }

  try {
    const response = await fetch("/get-weather", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ code: code.toUpperCase() }), //sends off the c
    });

    if (!response.ok) {
      throw new Error("Network response was not ok"); //for non-200 errors
    }

    const data = await response.json();
    return data; //returns the weather data
  } catch (error) {
    showAlert("Error fetching weather data");
    return null;
  }
};
```

```

//---displaying weather data for departure, destination and alternate airports

const METAR_departure = document.getElementById("departure_WX_metar");
const METAR_arrival = document.getElementById("arrival_WX_metar");
const METAR_alternate = document.getElementById("alternate_WX_metar");
const departure_WX_grading = document.getElementById("departure_WX_grading");
const arrival_WX_grading = document.getElementById("arrival_WX_grading");
const alternate_WX_grading = document.getElementById("alternate_WX_grading");

const refreshWX = document.getElementById("refresh_WX");

async function displayWeather() {
  //feedback for refresh button
  refreshWX.classList.add("refreshing");

  //fetching the data
  if (data.flight.departureAirport_code) {
    var departureWeather = await getWeather(data.flight.departureAirport_code);
  } else {
    var departureWeather = undefined;
  }

  if (data.flight.destinationAirport_code) {
    var arrivalWeather = await getWeather(data.flight.destinationAirport_code);
  } else {
    var arrivalWeather = undefined;
  }

  if (data.flight.alternateAirport_code) {
    var alternateWeather = await getWeather(data.flight.alternateAirport_code);
  } else {
    var alternateWeather = undefined;
  }
}

```

```

//displaying the METARs should they exist
if (departureWeather) {
  METAR_departure.innerHTML = `<b>${departureWeather.metar.raw_text}</b>`;
  departure_WX_grading.innerHTML = `<b class='grading'>${departureWeather.metar.flight_category}</b>`;
} else {
  METAR_departure.innerHTML = "<b>METAR not available</b>";
  departure_WX_grading.innerHTML = "<b style='color: gray;'>NIL</b>";
}

if (arrivalWeather) {
  METAR_arrival.innerHTML = `<b>${arrivalWeather.metar.raw_text}</b>`;
  arrival_WX_grading.innerHTML = `<b class='grading'>${arrivalWeather.metar.flight_category}</b>`;
} else {
  METAR_arrival.innerHTML = "<b>METAR not available</b>";
  arrival_WX_grading.innerHTML = "<b style='color: gray;'>NIL</b>";
}

if (alternateWeather) {
  METAR_alternate.innerHTML = `<b>${alternateWeather.metar.raw_text}</b>`;
  alternate_WX_grading.innerHTML = `<b class='grading'>${alternateWeather.metar.flight_category}</b>`;
} else {
  METAR_alternate.innerHTML = "<b>METAR not available</b>";
  alternate_WX_grading.innerHTML = "<b style='color: gray;'>NIL</b>";
}

await initializeGradingColors();
//feedback for refresh button
refreshWX.classList.remove("refreshing");
}

//--on page load:
await displayWeather();

//refreshing the weather data

refreshWX.addEventListener("click", () => {
  displayWeather();
})

```

What I did

The weather report algorithm heavily works the same way as the algorithms used for the weather tab. The weather reports are obtained via the `getWeather()` function (extracted straight from `weather.js`) which returns the necessary weather data. When being displayed, I take the METAR's raw text as well as the grading for that METAR. They are then displayed on the screen by replacing the HTML content of the dom element. If a report is not available, or no airport for that category has been entered already, it will return undefined for that specific airport category, and alternative messages are displayed as shown above. Finally, `initializeGradingColors()` (also extracted from `weather.js`) corrects each grading to their relevant colours.

Why I did it this way

Another option I initially considered before resorting to this solution was that instead of fetching entire decoded reports (only for the raw text METAR to be shown) I instead write a separate algorithm in `app.py` to just give me the single METAR report, which takes less time to fetch. This was rejected because I needed the grading for the reports as well which unfortunately is only available in the whole decoded report. A shame indeed.

Program output

 DashboardWX test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
144	Dashboard weather display test	Upon loading the dashboard the relevant METAR for each airport should be displayed, and an alternative message 'NIL + METAR not found' displayed if there is no airport selected.	PASS (See program output)

Iteration Review

Met requirements: (Aim complete) Display weather reports

Changes: N/A

Next steps: Expenses

Iteration 6 - Adding Expenses

dashboard.html

```
151 | <button class="button" id="add_expense"><svg id="add_expense_icon" fill="currentColor" style="width: 30px" viewBo
152 | </div>
153 | <!--content-->
154 | <div id="expenses_content">
155 |   <p><b>Loading...</b></p>
156 | </div>
157 | <div id="total_expense" style="display: flex; justify-content: center; margin-bottom: 10px;">
158 |   <p>Total Flight Cost: <b>£ ...</b></p>
159 | </div>
160 | <div id="create_expense" class="hidden" style="display: flex; justify-content: center; margin-bottom: 10px;">
161 |   <p><input id="expense_name" type="text" placeholder="Enter expense" style="margin-right: 10px;"></p>
162 |   <p><b></b><input id="expense_cost" type="number" placeholder="Enter cost (£)" style="margin-left: 10px;"></p>
163 |   <button class="button" id="submit_expense" style="margin-left: 10px;">Add</button>
164 | </div>
```

dashboard.js

```
//-----EXPENSES-----
//-----Adding expenses
const add_expense_button = document.getElementById("add_expense");
const add_expense_icon = document.getElementById("add_expense_icon");
const create_expense = document.getElementById("create_expense"); //container for adding expenses
const total_expense = document.getElementById("total_expense"); //total expenses display

//managing add expense (+) button and containers using class toggles
add_expense_button.addEventListener("click", () => {
  if (add_expense_button.classList.contains("adding")) {
    add_expense_button.classList.remove("adding");
    add_expense_icon.classList.remove("adding");
    create_expense.classList.add("hidden");
    total_expense.classList.remove("hidden");
  } else {
    add_expense_button.classList.add("adding");
    add_expense_icon.classList.add("adding");
    create_expense.classList.remove("hidden");
    total_expense.classList.add("hidden");
  }
})

//adding an expense
async function addExpense(name, price) {
  //updating the session
  const response = await fetch("/add-expense", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
    },
    body: JSON.stringify({ expense : [name, price] }), //sends off the expense data to the backend
  });

  if (!response.ok) {
    showAlert("Error adding expense");
    return;
  }
}

const submit_expense = document.getElementById("submit_expense");
```

```

submit_expense.addEventListener("click", async () => {
  try {
    //feedback for add button pressing
    submit_expense.style.opacity = "0.6";
    submit_expense.textContent = "...";
    submit_expense.style.pointerEvents = "none"; //prevent multiple clicks

    const expense_name = document.getElementById("expense_name")
    const expense_cost = document.getElementById("expense_cost")
    //user cant submit empty fields
    if (expense_name.value === "" || expense_cost.value === "") {
      showAlert("Please enter both an expense name and price");
      submit_expense.style.opacity = "1";
      submit_expense.textContent = "Add";
      submit_expense.style.pointerEvents = "auto";
      return;
    } else {
      await addExpense(expense_name.value, expense_cost.value);
      await displayExpenses(); //update the expenses list and total
      //reset the input fields
      expense_name.value = "";
      expense_cost.value = "";

      //reset the button after a short delay to simulate processing time

      submit_expense.style.opacity = "1";
      submit_expense.textContent = "Add";
      submit_expense.style.pointerEvents = "auto";
    }
  } catch (error) {
    console.error(error);}
})

```

```

async function displayExpenses() {
  //---displaying expenses list
  const expenses_content = document.getElementById("expenses_content");

  //fetching the data from session as it may have been updated
  const response = await fetch("/get-flight");
  const data = await response.json();

  //iterate through the expenses and display them in list format
  let expensesHTML = `<div>`;
  for (let i = 0; i < data.expenses.length; i++) {
    expensesHTML += `<button class='expense_item Expense_item' style='margin-right: 10px;'>${data.expenses[i].name} </button>`;
  }
  expensesHTML += `</div>`;
  expenses_content.innerHTML = expensesHTML;

  //---calculating and displaying total expenses

  //iterate through the expenses price and total them
  const total_expense = document.getElementById("total_expense")
  let total = 0;
  for (let i = 0; i < data.expenses.length; i++) {
    total += parseFloat(data.expenses[i][1]);
  };
  total_expense.innerHTML = `<p>Total Flight Cost: <b>E${total.toFixed(2)}</b></p>`; //2dp for price
}

```

app.py

```

#-----DASHBOARD-----

#-----ADDING AN EXPENSE
@main.route('/add-expense', methods=["POST"])
def add_expense():
    expense = request.json.get("expense") #expecting an array [expense name, amount]
    session["flight_data"]["flight"]["expenses"].append(expense)
    session.modified = True
    return jsonify({"status": "success"}), 200

```

dashboard.css

```

#total_expense {
  position: relative;
  visibility: visible;
  opacity: 1;
  background-color: var(--bg);
  padding: 10px;
  border-radius: 10px;
  transition: all 0.3s ease-in-out;
  z-index: 11;
}

#total_expense.hidden {
  visibility: hidden;
  opacity: 0;
  translate: 0 +20px;
  transition: all 0.3s ease-in-out;
}

#add_expense.adding {
  background-color: var(--alert);
}

```

```

        transition: 0.3s ease-out;
    }

    #add_expense_icon {
        transition: 0.3s ease-out;
    }

    #add_expense_icon.adding {
        transform: rotate(45deg);
        transition: 0.3s ease-out;
    }

    #create_expense {
        opacity: 1;
        transition: all 0.3s ease-in-out;
        position: relative;
        background-color: var(--bg);
        padding: 20px 5px 10px 5px;
        border-radius: 10px;
        bottom: 77px;
        z-index: 10;
    }

    #create_expense.hidden {
        visibility: hidden;
        opacity: 0;
        translate: 0 -20px;
        transition: all 0.3s ease-in-out;
    }

    #submit_expense {
        background-color: var(--primary);
        padding: 5px 15px;
        border-radius: 5px;
        font-family: 'Manrope';
        font-weight: 600;
    }

    #submit_expense:hover {
        cursor: pointer;
    }

    .expense_item {
        font-size: medium;
        font-family: 'Manrope';
        transition: box-shadow 0.1s ease-in-out
    }

    .expense_item:hover {
        background-color: var(--highlight);
        box-shadow: 0 0 0 3px var(--alert);
        cursor: pointer;
    }

```

What I did

To add expenses I first needed to find a suitable interface for the user to add them. I got a cool idea, where the plus icon rotates to form an X, and in this position some other UI becomes visible and the user can add as many expenses as they want. They could then press the X, and it would rotate back into a plus and the UI resets to how it was before.

This was done with a couple of transitions and transformations, and it rotates as well as the background changing to a red to notify the user that they can click it again.

For the adding items UI, I decided to do the same thing in some sense. Essentially, there are two containers at the bottom of the expense subsection: Total price and the adding expenses container. By default, the total price container is what is visible, but upon pressing the plus icon, the UI for the total price is swapped with the UI to add expenses. This way it doesn't take a lot of space.

The expenses are stored in a 2D array `"expenses"` in `app.py`, where the first item is the expense name and the second is the price of the expense. To display these expenses they are fetched from the session, and each expense is formulated into a button with the name and price. They are then embedded in a `<div>`, and the entire expense div is HTML is replaced with the new expenses. The purpose of them being buttons was for the click-to-remove feature I planned to develop which you can see the explanation and justification for in the next iteration. To display the total I iterate through each item and add the float version of it (as they can be so) to a total, then replace the HTML to display the new total.

To add an expense, a button in the adding UI can be pressed. Then, the items stored in the input boxes are extracted and are sent into the `addExpense()` function, which sends both these values to the backend. The backend then appends the values to the expenses array. Once this is complete, `displayExpenses()` is run again, which should now show the added expense.

Why I did it this way

An arguably more simpler way of doing this was to use a popup modal like i've used quite a lot in this project. This was the initial idea but I wanted a method that would integrate with the already existing UI. This is another one of those things that can make the website feel more polished and professional. If all inputs like these were popups, this could potentially annoy the user if they plan to add a lot of expenses due to the constant popping up of items on the screen. However, a smooth transition to what was already there by all means is more appealing to the user, as all functionality is embedded within the specific subtab, also giving them access to other parts of the dashboard too.

Program output

 Adding expenses test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
145	Adding expenses test	When pressing the add button, the UI smoothly transitions	PASS (See program output)

		from normal to the adding UI as described. Items can be added by putting the details in the input boxes and pressing 'Add'; they should then be displayed on screen.	
--	--	--	--

Input Validation Tests

Test Data	Expected Outcome	Actual Outcome
No inputs in the expense name and price	An error message stating that both inputs must be filled should be displayed	PASS
At least one input filled but the other not	An error message stating that both inputs must be filled should be displayed	PASS

Full test video:  Add expense validation.mp4

Iteration Review

Met requirements: (Aim complete) Allow the user to add expenses to their flight

Changes: N/A

Next steps: Removing expenses

Iteration 7 - Removing expenses

dashboard.html

```

<!--header-->
<div id="expenses_header" style="display: flex; align-items: center; gap: 10px; justify-content: center;">
  <div style="display: flex; align-items: center; gap: 15px;">
    <svg fill="currentColor" style="width: 15px; viewBox="0 0 320 512" xmlns="http://www.w3.org/2000/svg">
      <h3>Expenses</h3>
    </div>
    <p id="click_to_remove" style="color: #737373; font-size: 15px;">Click to remove</p>
  </div>

```

dashboard.css

```

#click_to_remove {
  transition: opacity 0.3s ease-in-out;
  opacity: 0;
}

```

dashboard.js

```
//-----Removing expenses
const click_to_remove = document.getElementById("click_to_remove");
document.getElementById("expenses_content").addEventListener("mouseover", async (event) => {
  if (event.target.classList.contains("Expense_item")) {
    click_to_remove.style.opacity = "1";
  })

document.getElementById("expenses_content").addEventListener("mouseout", async (event) => {
  if (event.target.classList.contains("Expense_item")) {
    click_to_remove.style.opacity = "0";
  })

//on click, if the target is an expense item the name is extracted
document.getElementById("expenses_content").addEventListener("click", async (event) => {
  if (event.target.classList.contains("Expense_item")) {

    //sometimes the click registers on the bold price text within the button,
    //so we need to check for that too and extract the name from the parent element instead
    if (event.target.classList.contains("expense_item")) {
      var expense_text = event.target.textContent;
    } else {
      var expense_text = event.target.parentElement.textContent;
    }

    //get the name part of the expense (value before the colon)
    const expense_name = expense_text.split(":")[0];

    //send the name to the backend to remove the expense from session
    const response = await fetch("/remove-expense", {
      method: "POST",
      headers: {
        "Content-Type": "application/json",
      },
      body: JSON.stringify({ name: expense_name }), //sends off the expense name to the backend
    });

    if (!response.ok) {
      showAlert("Error removing expense");
      return;
    }

    click_to_remove.style.opacity = "0";
    await displayExpenses(); //update the expenses list and total
  }
})
```

app.py

```
#-----REMOVING AN EXPENSE
@main.route('/remove-expense', methods=["POST"])
def remove_expense():
    name = request.json.get("name") #expecting the name of the expense to remove
    try:
        #find the expense index with the specific name, then remove it
        for index, expense in enumerate(session["flight_data"]["flight"]["expenses"]):
            if expense[0] == name:
                session["flight_data"]["flight"]["expenses"].pop(index)
                break
        session.modified = True
        return jsonify({"status": "success"}), 200
    except IndexError: #shouldn't be possible as a click is required on frontend
        return jsonify({"error": "Invalid expense index"}), 400
```

What I did

To remove an expense, I decided to go with a 'click-to-remove' aspect, where the items are pressable buttons that can be clicked to be removed from the expenses list. To implement this, the first step was already complete, and that was to have each price modeled as a <button> element instead of a <p> element.

Next, I just needed to code the removing algorithm. When something is clicked inside the expenses_content div, the algorithm checks if the target item is an 'expense-item' which is the ID given to all expenses once created. If that is the case, the name part is extracted via a split at the colon of the text (name: £price) and is sent to the backend. `remove_expense()` takes the name, iterates through the expenses array (with the index this time) and removes the item index whose name matches the target name to remove. After this, `displayExpenses()` is run again to display the updated expenses and totals.

An annoying bug that occurred during this was that often I'd end up clicking the price part of the button instead of the actual button, and because it didn't have the class 'expense-item', the click did nothing. I fixed this by giving both classes an alternative class of 'Expense-item' (difference being the capital E) and by default the remove algorithm checks for this class. If that element also consists of the normal 'expense-item' class, then the text to extract is just the text content of that element. If not, it means the element clicked is the price and the parentElement must be used instead to ensure the right data is used to extract the name (so 'name: £price', not '£price').

Why I did it this way

There were many ways I could have approached deleting expenses, for instance, I could have a separate UI where I typed in the name to remove, or I could have a delete label/button appear next to the item when the user hovers over it, so they can press it to delete. Once again, these both require extra elements, which I'm trying my best to avoid. My approach uses the already existing expenses as buttons, which keeps everything minimal and also quick and intuitive for the user, especially in comparison to having to type the whole name and pressing a remove button to do the same action.

Program output

 Removing expenses test.mp4

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
146	Removing expenses test	When an expense is pressed, it gets removed from the expenses list and this is saved in the session (so on reload there is no difference)	PASS (See program output)

Iteration Review

Met requirements: Success criteria item complete: Fuel, Mass and balance, and Expense calculators

Changes: N/A

Next steps: AI brief

Iteration 8 - AI Brief

ai_roles.py

```
elif type == 'Brief':
    role = '''
Generate a comprehensive VFR pre-flight briefing for the following flight details. Your briefing should cover all relevant aspects of the flight.

(N) - NOTAMS: Not to be covered in the briefing as there is currently no access to NOTAM data.
W - Weather: Provide a detailed weather briefing for the departure, enroute, and arrival locations
K - Known ATC Delays: Identify any known ATC delays or restrictions that could affect the flight, such as airspace congestion or staffing issues.
R - Runway Lengths: Provide information on the runway lengths at both the departure and arrival airports, including any relevant considerations.
A - Alternatives: Revise the suitable alternatives for the flight, including alternate airports or routes.
F - Fuel: Revise the calculated fuel given and consider any additional fuel requirements based on the current flight data and conditions.
T - Takeoff and Landing Performance: Assess the takeoff and landing performance requirements for the flight.

Where the user has not provided sufficient information for you to cover all aspects of the mnemonic, you should inform the user of the missing information.

Your response should be in the following JSON format:

{
  "briefing": "string containing the briefing. Use HTML tags to format your response (bold, emphasis, headings, colors, underlines etc.)"
}

YOU MUST STRICTLY OBEY THE ABOVE FORMAT, NO COMMENTING, BRACKETS, OR ANYTHING OUTSIDE OF THIS FORMAT OF ANY SORT SHOULD BE INCLUDED AT ANY POINT.
Do not ask any questions when outputting as the user has no ability to respond.
'''
```

dashboard.js

```
//-----AI BRIEF-----

const refresh_brief= document.getElementById("refresh_brief");
const AI_brief_output = document.getElementById("AI_brief_output");

refresh_brief.addEventListener("click", async () => {
  refresh_brief.style.opacity = "0.6";
  refresh_brief.style.pointerEvents = "none"; //prevent multiple clicks while generating

  const response = await fetch('/get-flight');
  const flightData = await response.json();

  const promptText = `
  FLIGHT DATA:

  ROUTE:
  Departure Airport: ${flightData.departureAirport_name}
  Arrival Airport: ${flightData.destinationAirport_name}
  Alternate Airport: ${flightData.alternateAirport_name}
  Distance: ${flightData.distance.output}
  Time Enroute: ${flightData.time}
  Route: ${flightData.route_names}

  WEATHER:
  Departure Airport METAR: ${METAR_departure.textContent} (${departure_WX_grading.textContent})
  Arrival Airport METAR: ${METAR_arrival.textContent} (${arrival_WX_grading.textContent})`
```

```

    MGW: ${flightData.MGW.output}

    Navigation log:
    ${JSON.stringify(flightData.NAVLOG)}

    Weight and Balance Table:
    ${JSON.stringify(flightData.weight_items)}

    CG Table:
    ${JSON.stringify(flightData.CG_calculations)}
    `;

    AI_brief_output.innerHTML = "<p style='color: █#737373;'>Generating brief...</p>";

    const aiResponse = await prompt("Brief", promptText);
    try {
        const resp = JSON.parse(aiResponse); //parsing the response to JSON
        console.log(resp);
        AI_brief_output.style.color = "█#000000";
        AI_brief_output.innerHTML = `<p>${resp.briefing}</p>`;
        update("brief", resp.briefing);
    } catch (error) {
        console.error(error);
        AI_brief_output.style.color = "█#FF0000";
        AI_brief_output.innerHTML = "<p> An error occurred while generating the brief. Please ensure your r
    }
    refresh_brief.style.opacity = "1";
    refresh_brief.style.pointerEvents = "auto";
}
)

```

What I did

For the brief, the AI follows an acronym quite popular in general aviation, called NWCRAFT. Above are the meanings of each letter. This way, I can ensure all the aspects of the brief are covered in a somewhat acceptable manner. It was also important to ensure that any items not used for any reason (mostly due to them being unavailable) are told to the user, so that it is clear on any assumptions made by the AI. The data it is given is the same as the data given in mass and balance, but it includes weather data and the navlog too. I've also given the AI ability to use HTML elements. Output is the same, a json which is parsed and the string extracted from it, and then the innerHTML of the output div is replaced by the AI's output.

Why I did it this way

I could've asked the AI to give a brief based on its own interpretation and its own way of outputting, but this is high risk as there is no guarantee that the AI will cover everything that is actually required in the brief. Giving it an acronym to follow still leaves tons of autonomy, but the output is structured in a way that ensures the AI will cover all aspects of the flight plan. Giving the AI ability to use HTML was really important, especially for things that need to be clear- like the AI establishing that the flight is not safe to execute, as the user can have their focus drawn to what is most important in the brief, and avoid the risk of missing this information should there be no such formatting.

Program output

terminal

```

/workspaces/AIVFR/.venv/lib/python3.12/site-packages/werkzeug/sansio/response.py:232: UserWarning:
The 'session' cookie is too large: the value was 4126 bytes but the header required 26 extra
bytes. The final size was 4152 bytes but the limit is 4093 bytes. Browsers may silently ignore
cookies larger than this. dump_cookie(

```

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
147	AI briefing test	Upon pressing the refresh button the AI should generate a comprehensive flight brief based on current flight data, with proper HTML formatting.	<pre>/workspaces/AIVFR/.venv/lib/python3.12/site-packages/werkzeug/sansio/response.py:232: UserWarning: The 'session' cookie is too large: the value was 4126 bytes but the header required 26 extra bytes. The final size was 4152 bytes but the limit is 4093 bytes. Browsers may silently ignore cookies larger than this. dump_cookie(</pre>

Error Analysis & Fixes

Test No: 147

Failure Description: The session exceeded the 4KB limit that is available for the cookie and so data could not be added to the session anymore.

Cause: This was triggered as I was filling out sample flight data so that the AI can generate a flight with all the necessary data, but the session got too big as I was doing this.

Fix: Being honest- I saw this coming from a mile away but chose to ignore it as I didn't think the session would get too large. Now that it has become a barrier in development, I have no choice but to make a database that can store however much data is needed within it.

Justification: A database is the most standard way to store user data, and this is one of those times where the alternative gets implemented. I can't imagine storing the data in a variable on the server side as this can get very big and also will not be user specific, in the sense that in the future I can't have the user log in and retrieve their data. The same thing applies to using a variable in the frontend, which is considerably worse for the same reasons.

Switching from Session to a Database

models.py (new)

```
from datetime import datetime
from .extensions import db

class Flight(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    created_at = db.Column(db.DateTime, default=datetime.now)
    data = db.Column(db.Text)
```

extensions.py (new)

```
from flask_sqlalchemy import SQLAlchemy

db = SQLAlchemy()
```

__init__.py

```
8 def create_app():
9     app = Flask(__name__)
10    app.secret_key = os.getenv("SESSION_KEY")
11    app.config["SQLALCHEMY_DATABASE_URI"] = "sqlite:///flights.db"
12
13    app.config["SQLALCHEMY_TRACK_MODIFICATIONS"] = False
14
15    db.init_app(app)
16
17    from .app import main as app_blueprint
18    app.register_blueprint(app_blueprint)
19
20    with app.app_context():
21        db.create_all()
22
23    return app
24
```

app.py (The rest of the changes are here.)

```
#-----SERVER-SIDE FUNCTIONS
def get_flight_data():
    flight = Flight.query.get(session["flight_id"])
    data = json.loads(flight.data)
    return flight, data

def save(flight, data):
    flight.data = json.dumps(data)
    db.session.commit()

def update(path, value):
    flight, data = get_flight_data()
    target = data

    #using paths to update nested values (as implemented in frontend)
    #splitting at the dots for each level
    keys= path.split(".")

    #iterating through the keys to get to the target value
    for k in keys[:-1]:
        if k not in target or not isinstance(target[k], dict):
            return jsonify({"error": "Invalid key path"}), 400
        target = target[k]
    target[keys[-1]] = value
    save(flight, data)
    return data, 200

#main menu
@main.route('/') #this will be the first to load up
def index():
    if "flight_id" not in session: #initialise the session if not created already
        new_flight = Flight(data=json.dumps(data_template))
        db.session.add(new_flight)
        db.session.commit()
        session["flight_id"] = new_flight.id
        update_units()

    return render_template('menu.html', APP_VERSION="0.7.1-alpha", data=get_flight_data()[1], settings
```

Directory routes

Old

```
return render_template('settings.html', settings=session["flight_data"]["settings"],
flight=session["flight_data"]["flight"])
```

New

```
return render_template('settings.html', settings=get_flight_data()[1]["settings"],
flight=get_flight_data()[1]["flight"])
```

Updating the flight data

Old

```
Session["flight_data"]["flight"]["path"] = value
```

New

```
update('flight.path', "value")
```

All instances of the values above throughout app.py had to be replaced with the new values for the database to work properly. A more detailed explanation of changes is provided in the video test for this iteration.

Program output

 Session to database conversion test and explanation.mp4 [21mins]

Known Limitations / Missing features

- N/A

Test No.	Test Description	Expected Outcome	Actual Outcome
148	Session → Database conversion test	On creating a sample flight, all variables should be stored in the database without any issues, and all algorithms should still be fully functional as they were before the conversion.	PASS (See program output)
147	AI briefing test	Upon pressing the refresh button the AI should generate a comprehensive flight brief based on current flight data, with proper HTML formatting.	 AI brief test.mp4

Iteration Review

Met requirements: **Success criteria item complete:** Full flight brief
(Aim Complete) Make the AI generate a briefing for the entire flight

Changes: Converted the data storage method from a session cookie to a database.

Next steps: Downloading the flight plan as a PDF

Iteration 9 - Downloading the flight plan as a PDF

base.html

```
<!--Base PDF Template-->
<div id="BasePDF_template">
  <div id="base_pdf_header">
    <!--Logo-->
    <div>
      <span id="logoAI_pdf" style="font-size: 70px;">AI</span>
      <span id="logoVFR_pdf" style="font-size: 70px;">VFR</span>
      <p style="margin-top: -16px; font-size: 12px;"><b>AI ASSISTED FLIGHT PLANNING</b></p>
    </div>
    <h2 style="color: #000000;">Flight Plan</h2>
  </div>

  <!--Content-->
  <div id="base_pdf_content" class="light_mode" style="padding: 20px 0;">
<!-------ROUTE----->
    <div class="pdf_title">ROUTE</div>
    <div id="pdf_route_content" class="pdf_content">
      <div> <!--FROM-->
      <b>FROM</b>
      <div class="container"><b>{{ flight.departureAirport_code }}</b> {{ flight.departureAirport_name }}
      </div>

      <!--TIME & DISTANCE-->
      <p>{{ flight.distance.output }} <b>|</b> {{ flight.time }}</p>
      

      <div> <!--TO-->
      <b>TO</b>
      <div class="container"><b>{{ flight.destinationAirport_code }}</b> {{ flight.destinationAirport_name }}
      </div>
    </div>

    <div style="display: flex; justify-content: center; margin-top: 30px; text-align: center;">
    <div>
      <b>ALTERNATE</b>
      <div class="container" style="background-color: var(--alternate);"><b>{{ flight.alternateAirport_code }}</b>
      </div>
    </div>

<!--WAYPOINTS-->
    <div style="display: flex; justify-content: center;">
    <div class="container" style="margin-top: 30px; width: 75%; min-height: 120px;">
      {{ ' ' → '.join(flight.route_names) if flight.route_names else 'No Route Data' }}
    </div>
    </div>

<!--AIRCRAFT DETAILS----->
    <div class="pdf_title" style="margin-top: 30px;">AIRCRAFT DETAILS</div>
    <div id="pdf_aircraft_details" style="display: flex; justify-content: center; gap: 70px; margin-top: 20px;">
    <div>
      <p>Type: <b>{{ flight.aircraft }}</b></p>
      <p>MTOW: <b>{{ flight.MTOW.output }}</b></p>
      <p>MLW: <b>{{ flight.MLW.output }}</b></p>
    </div>
    <div>
      <p>Max Tank Capacity: <b>{{ flight.max_tank_capacity.output }}</b></p>
      <p>Fuel Burn: <b>{{ flight.fuel_burn.output }}</b></p>
    </div>
    </div>

<!--FUEL----->
    <div class="no-split">
    <div class="pdf_title" style="margin-top: 30px;">FUEL</div>
    <!--FUEL POLICIES-->
    <table id="fuel_policy_table">
      <thead>
        <tr>
          <th>Element</th>
          <th>Policy</th>
          <th>Time Allowed (mins)</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <td>Trip</td>
          <td>{{ flight.fuelPolicy_trip.policy }}</td>
          <td>{{ flight.fuelPolicy_trip.time_allowed }}</td>
        </tr>
        <tr>
          <td>Contingency</td>
          <td>{{ flight.fuelPolicy_contingency.policy }}</td>
          <td>{{ flight.fuelPolicy_contingency.time_allowed }}</td>
        </tr>
      </tbody>
    </table>
  </div>
</div>
```

```

<!--FUEL REQUIRED-->
<h2 style="text-align: center; margin-top: 30px;">Fuel Required</h2>
<div id="pdf_fuel_required" style="display: flex; justify-content: center; gap: 70px;">
  <div>
    <p>Taxi: <b>{{ flight.fuel_taxi.output }}</b></p>
    <p>Trip: <b>{{ flight.fuel_trip.output }}</b></p>
    <p>Contingency: <b>{{ flight.fuel_contingency.output }}</b></p>
  </div>
  <div>
    <p>Alternate: <b>{{ flight.fuel_alternate.output }}</b></p>
    <p>Final Reserve: <b>{{ flight.fuel_finalReserve.output }}</b></p>
    <p>Additional: <b>{{ flight.fuel_additional.output }}</b></p>
    <p>Extra: <b>{{ flight.fuel_extra.output }}</b></p>
  </div>
</div>
<!--Fuel totals-->
<div id="pdf_fuel_totals" style="display: flex; justify-content: center; gap: 70px; margin-top: 20px;">
  <div class="container"><b>Total: {{ flight.fuel_total.output }}</b></div>
  <div class="container"><b>Mass: {{ flight.fuel_mass.output }}</b></div>
  <div class="container"><b>Endurance: {{ flight.fuel_endurance }}</b></div>
</div>
</div>
<!--MASS & BALANCE----->
<div class="no-split">
  <div class="pdf_title" style="margin-top: 30px;">MASS & BALANCE</div>
  <!--LOADING TABLE-->
  <h2 style="text-align: center; margin-top: 30px;">Loading Table</h2>
  <table id="loading_table">
    <thead>
      <tr>
        <th>Item</th>
        <th>Weight</th>
        <th>Arm (in)</th>
        <th>Moment</th>
      </tr>
    </thead>
    <tbody id="moments_table1" class="active">
      <tr>
        <td>Basic Empty</td>
        <td><p>{{ flight.weight_items.basic_empty.weight.value|round(1) if flight.weight_items.basic_empty
        <td><p>{{ flight.weight_items.basic_empty.arm if flight.weight_items.basic_empty.arm != 0 else ''

```

```

<!--PERFORMANCE----->
<div class="pdf_title" style="margin-top: 30px;">PERFORMANCE</div>
<div id="performance_content" style="display: flex; justify-content: center; gap: 70px; margin-top: 20px;">
  <div class="container">
    <p>Takeoff distance required: <b>{{ flight.TODR }}</b></p>
    <p>Landing distance required: <b>{{ flight.LDR }}</b></p>
  </div>
  <div class="container">
    <p>Takeoff distance available: <b>{{ flight.TODA }}</b></p>
    <p>Landing distance available: <b>{{ flight.LDA }}</b></p>
  </div>
</div>
<div class="page-break"></div>
<!--WEATHER----->
<div class="no-split">
  <div class="pdf_title" style="margin-top: 30px;">WEATHER</div>
  <div id="weather_content" style="margin-top: 20px;">
    <!--DEPARTURE WEATHER-->
    <h2 style="text-align: center; margin-top: 20px;">{{ flight.departureAirport_name }}</h2>
    <h3>METAR</h3>
    <p style="margin-top: 10px; margin-bottom: 10px;" class="container">{{ flight.METAR_departure }}</p>
    <p class="report">{{ flight.METAR_departure_decoded|safe }}</p>
    <h3 style="margin-top: 20px;">TAF</h3>
    <p style="margin-top: 10px; margin-bottom: 10px;" class="container">{{ flight.TAF_departure }}</p>
    <p class="report">{{ flight.TAF_departure_decoded|safe }}</p>
    <!--ARRIVAL WEATHER-->
    <h2 style="text-align: center; margin-top: 20px;">{{ flight.destinationAirport_name }}</h2>
    <h3>METAR</h3>
    <p style="margin-top: 10px; margin-bottom: 10px;" class="container">{{ flight.METAR_arrival }}</p>
    <p class="report">{{ flight.METAR_arrival_decoded|safe }}</p>
    <h3 style="margin-top: 20px;">TAF</h3>
    <p style="margin-top: 10px; margin-bottom: 10px;" class="container">{{ flight.TAF_arrival }}</p>
    <p class="report">{{ flight.TAF_arrival_decoded|safe }}</p>
  </div>
</div>
<div class="page-break"></div>

```

```

</div class="page-break">/div>
<!--EXPENSES----->
<div class="pdf_title" style="margin-top: 30px;">EXPENSES</div>
<div id="expenses_content" style="margin-top: 20px;">
  <ul>
    {% for expense in flight.expenses %}
    <li><b>{{ expense[0] }}: </b> £{{ expense[1] }}</li>
    {% endfor %}
  </ul>
</div>
<div class="page-break"></div>
<!--BREIF----->
<div class="pdf_title" style="margin-top: 30px;">BRIEFING</div>
<div class="no-split" id="briefing_content" style="margin-top: 20px;">
  <p class="report no-split">{{ flight.brief|safe }}</p>
</div>
</div>
<div id="base_pdf_footer" class="light_mode">
  <p style="font-weight: 400;"><i>FOR EDUCATIONAL PURPOSES ONLY. MUST NOT BE USED IN REAL WORLD FLIGHT</i>
  <p><span style="font-weight: 400;" id="base_pdf_dateAndTime"></span> | <b>www.aivfr.co.uk</b></p>
</div>
</div>
</html>

```

basePage.css

```

/*-----Navlog pdf template-----*/

#navlogPDF_template {
  position: absolute;
  left: -9999px;
  width: 800px;
}

#BasePDF_template {
  position: absolute;
  left: -9999px;
  width: 800px;
}

#navlog_pdf_header, #base_pdf_header {
  display: flex;
  justify-content: space-between;
  border-bottom: 1px solid #ccc;
  padding-bottom: 10px;
  align-items: center;
}

#navlog_pdf_footer, #base_pdf_footer {
  text-align: center;
  font-size: 14px;
  margin-top: 50px;
  border-top: 1px solid #ccc;
}

/*-----Base pdf template-----*/

```

```

.pdf_title {
  font-size: 29.8px;
  font-weight: 300;
}

/*route*/
#pdf_route_content {
  display: flex;
  justify-content: space-between;
  gap: 20px;
  align-items: center;
  text-align: center;
}

.pdf_content {
  padding: 10px;
}

.container {
  background-color: var(--bg);
  padding: 10px;
  border-radius: 10px;
  margin-top: 5px;
}

#plane_icon {
  position: absolute;
  width: 30px;
  color: var(--primary);
  opacity: 0.4;
  transform: rotate(135deg);
  left: 350px;
  top: 206.5px;
  scale: 3.5;
  z-index: -1;
}

/*fuel*/

/*Policy table*/
table {
  width: 99%;
  border-collapse: collapse;
  margin-top: 10px;
}

th, td {
  border: 1px solid var(--text);
  padding: 8px;
  text-align: center;
  font-family: 'Manrope';
  font-size: 14px;
}

#fuel_policy_table td:nth-child(2) {

```

```

        width: 40%;
    }

    th {
        background-color: var(--tableHeader);
    }

    /*weather reports formatting*/
    .report {
        white-space: pre-line;
    }

    /*dealing with page breaks*/

    .page-break {
        page-break-before: always;
        break-before: page;
    }

    .no-split {
        page-break-inside: avoid;
        break-inside: avoid;
    }

```

dashboard.js

```

//-----downloading as PDF

const save_pdf = document.getElementById("save_pdf");

save_pdf.addEventListener("click", () => {
    save_pdf.classList.add("savingOrSaved"); //button feedback for saving
    save_pdf.innerHTML = "...";

    saveFlight_pdf();
    update("saved_pdf", "True");

    //simulate saving time
    setTimeout(() => {
        save_pdf.innerHTML = `<div><svg id="save_pdf_icon" fill="currentColor" viewBox="0 0 24 24" xmlns="http://www.w3.org/2000/svg"><path d="M19 19H5V5H19V19Z" /></div>`;
    }, 2000); //2 seconds to simulate saving time
});

async function saveFlight_pdf() {
    await fetch('/get-flight')
    .then(response => response.json())
    .then(async FlightData => {
        //get the departure and arrival names to put in the file name
        const departure_code = FlightData.departureAirport_code
        const arrival_code = FlightData.destinationAirport_code

        //getting the template
        const template_container = document.getElementById('BasePDF_template');
        const dateAndTime = document.getElementById('base_pdf_dateAndTime');

        dateAndTime.innerHTML = `Generated on ${new Date().toLocaleDateString()} at ${new Date().toLocaleTimeString()}`
        //using html2pdf to convert the html table into a image, then save to PDF
        const options = {
            margin: 10,
            filename: `${departure_code}-${arrival_code}.pdf`,
            image: { type: 'jpeg', quality: 0.98 },
            html2canvas: { scale: 2, useCORS: true, scrollX: 0, scrollY: 0 },
            logging: false,
            jsPDF: { unit: 'mm', format: 'letter', orientation: 'portrait' },
            pagebreak: { mode: ['avoid-all', 'css', 'legacy'] } //to prevent unwanted page breaks in the PDF
        };

        html2pdf().set(options).from(template_container.innerHTML).save();
    })
}

```

What I did

To generate the pdf, the same tactics (i.e. html2pdf) were used. However in this case, I had to design the entire document in HTML, whilst before I only needed a header, footer and then I'd just insert the navigation log into the DOM. The design aims to be simple and legible to the user, and contains all flight data, all of which is sourced from the database via Jinja2 syntax.

Then to have the pdf downloaded, the exact same format of the function used in the navlog is used here, with a few adjustments. First, there is no longer a need to fetch the navlog as that will not be included in the plan. Then, the pdf is set up with the additional option of `pagebreak`, which when set to the settings above will put breaks in the document wherever an element with the `.pagebreak` class is found. Additionally, to minimise splitting of elements across pages, the `.no-split` element prompts the algorithm to break onto a new page if it finds that it will need to split an element across two pages.

Why I did it this way

The navlog was not included in the plan as I had no way of accessing it without crafting an entirely new one, which is complex as it's rows are dynamic so I would have to make the algorithm that adds them again, which I deemed as unnecessary as I had already implemented the ability to download the navlog from the navlog tab itself. Similar to the navlog, I could have used the normal jsPDF to generate this, but this would mean I would lose styling which is important in the formatting of the document. Later on I may use tools like puppeteer which provides perfect pdfs that have selectable text as well and is not just an image, but I cannot implement it now due to time constraints.

Program output

 Flight plan pdf generation test.mp4

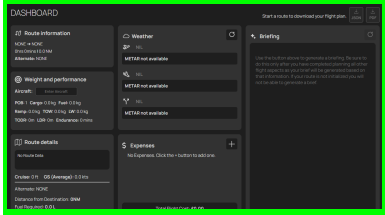
Known Limitations / Missing features

- Retained Limitations -

These will be touched upon in the Limitations section of the Evaluation.

- Some text is still split across pages despite them having a `.no-split` class.

Test No.	Test Description	Expected Outcome	Actual Outcome
149	Flight plan pdf generation test	Upon pressing the pdf download button on the dashboard, the flight plan should be downloaded in a pdf format with all the necessary flight data as appropriate.	PASS (See program output)

150	Dark mode test	Darkmode should apply the correct coloring to all elements.	
-----	----------------	---	---

Iteration Review

Met requirements: Success criteria item complete: Dashboard
(Aim complete) Allow all flight details to be downloaded as a pdf

Changes: N/A

Section Review

Aims completed in this section:

- **Complete 3 success criteria items: Dashboard, Full flight brief, Fuel, Mass and balance, and Expense calculators**
- Display Route information:
 - Departure → Arrival
 - Time and distance
 - Alternate aerodrome
- Display aircraft weight and performance:
 - Persons on board
 - Cargo weight
 - Fuel
 - Total (ramp)
 - MTOW, MLW
 - TOD, LD
- Display route information:
 - Route
 - Cruise and groundspeed
 - Alternate details
- Display weather reports
- Allow the user to add expenses to their flight
- Make the AI generate a briefing for the entire flight
- Allow all flight details to be downloaded as a pdf

Aims not completed in this section:
N/A

Iterative tests complete:

- AI should output suitable responses in accordance to their given role and user prompt
- APIs are operative and display data in the correct way on the relevant page
- Light/dark Mode switches correctly with the correct colours
- Dashboard displays correct data and updates accordingly

Remarks: Development section complete!

EVALUATION

Post Development testing

Structure of the tests

As mentioned in the design section:

‘A successful post-development test in the case of this project would ideally be a test of all these features **3 different times**, using the 3 different routes outlined in the route tab tests. If all 3 plans can be made without any significant errors or problems, then I as a developer can happily take my webtool as a reliable and robust system, and deem the project as successful.’

In light of this, the post development tests will be conducted as follows. There will be 3 tests, each associated with a test route as described in the design section.

1. The First test (EGBM - EGSC) will be a **Functionality test**, testing all AI functionality and ensuring every aspect of the webapp works as appropriate (full demo), and will be reflective of the general success of the webapp as a whole. This test will be conducted in light mode.
2. The Second test (EGSX-EGMC) will be a **Usability test**. I will try to act as a pilot who has never used the app before, whilst making sure that all the planned usability features have been implemented and are working properly. I will also be watching out for when I assume user understanding of complex features, which will reveal a usability flaw. This test will be conducted in dark mode.
3. The Third test (EGSX - EGSX) will be a **Robustness test**. While making this route I will try to ‘break’ the webapp in several ways to see the way it responds. This test will be conducted in light mode.

A second iteration of this test will be conducted with manual inputs only (this will fulfil the manual input tests and will also determine if any non-AI related errors are clean errors that can be reproduced), and will be conducted in dark mode.

Functionality test: Tatenhill (EGBM) to Cambridge (EGSC) [DEMO]

 AIVFR Functionality test [FULL DEMO].mp4

NOTE: *I realized after this test that I had a VPN still connected which was why things were slightly slower than they should have been. Other tests will demonstrate much faster processing times!*

Post-development test data 

Test description (12)	Test Data	Successful test?
Webpage loads	Running Flask	SUCCESSFUL 0:26
Settings changes work, appropriate variable in data file changes	Settings buttons	SUCCESSFUL 1:34
Disclaimer operates as expected	Continue/New/Load flight buttons, disclaimer buttons	SUCCESSFUL 3:34
Dashboard is empty on first look	New flight button	SUCCESSFUL 4:03
User can navigate between pages	Cursor input	SUCCESSFUL 6:02
User can plan a route using the Route Tab with AI	Test route for functionality test (EGBM - EGSC)	SUCCESSFUL 8:00
Weather displays and updates correctly	Test route inputs, and 3 random airports of my choice (that don't form any routes from the above). At least one must not provide either a METAR or TAF report.	SUCCESSFUL 16:04
Navigation log can be edited either by AI or Manually	User input 'AUTOFILL' button	SUCCESSFUL 19:30

Fuel can be added with correct calculations for all fuel policies	User Input (int values only)	SUCCESSFUL 25:05
Correct mass and balance calculations	User input (int values only)	SUCCESSFUL 28:33
Dashboard shows all the flight details in a clear format and a brief generated on command	Flight_data.json (reading only) Generate brief button	SUCCESSFUL 36:25
Saving a flight to the device is possible	'Save flight' button input flight_data.json	PARTIALLY SUCCESSFUL 38:29

Usability Test: North Weald (EGSX) to London Southend (EGMC)

 AIVFR usability test.mp4

Evaluation of Usability Features

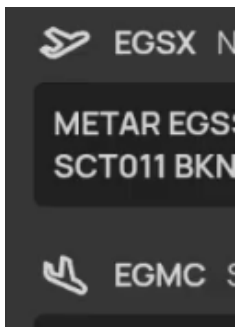
Usability features[↗]

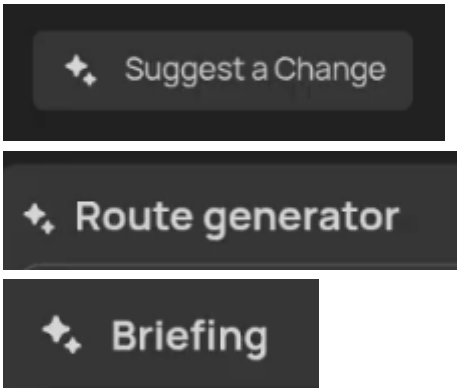
Learnability

Icons - **PARTIALLY MET**

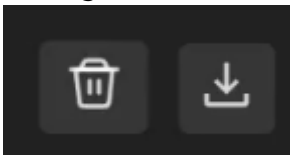
Icons were something that were very persistent throughout the test, and **almost all of them were clear** to what feature they were addressing, however this was not the case for all of them.

Some examples of the **successful** use of icons:

	This is an example of an icon without an attached label, but the definition remains the same (departure, arrival alternate). This works not only because you can see the actual departure and arrival airports placed next to it, but because these icons are already defined in the weather tab with their associated labels so common sense can be applied here and the user would understand what these icons address. The absence of these icons would make this
---	--

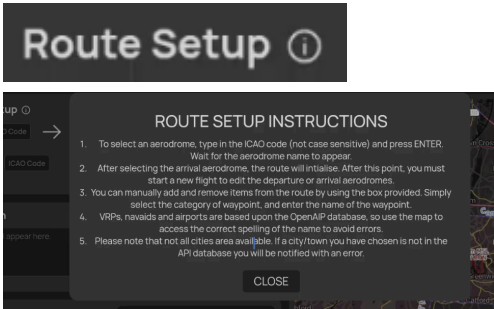
	‘telling’ process a bit harder because the user now has to examine the airport name and codes to determine what reports are being shown to avoid any mixups.
	Another good example was the AI icons, as they were key in showing the user that the feature they are going to access is an AI feature. The user can almost certainly get confused if these icons didn’t exist near their labels, and to whether manual input is required, until they actually interact with them which in some cases may not be what the user wants to do at that particular time.

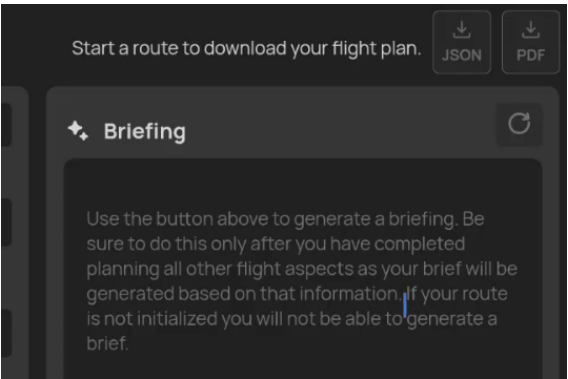
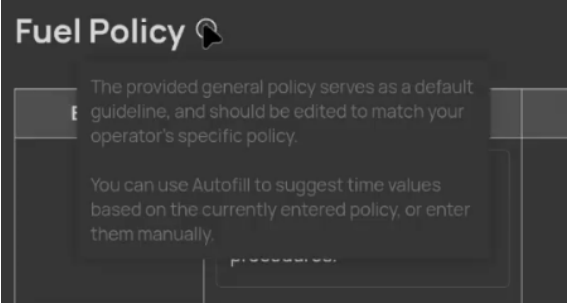
An example of an **unsuccessful** use of icons:

Navlog tab 	Here, these icons remain slightly ambiguous to the user on what they do, because of the lack of proper labelling. This is especially the case with the bin icon, which, as I mentioned in the video, could be mistaken to clear the route instead of clearing the navigation log. If a label was present for this button, or at least a tooltip, it would be clear to the user that the button clears the navlog only and they would be less hesitant to use it (which would hinder them from using the button even if they needed to)
---	--

Tooltips - **FULLY MET**

Tooltips have been used successfully throughout the webtool, all of them allowing the user to know what to do at certain points and also how some functionalities work. Some notable examples are shown below.

	The route setup tooltip was crucial, as It lays the foundation to how inputs work in quite a lot of places in the webtool. The user may be surprised on how to operate this tab at first, but by seeing the little (i) icon and clicking it, they are provided with all they need to make a route without any issues. If this tooltip failed to display then the user would struggle a lot in trying to plan the route, add waypoints etc, the most probable issue being how to actually add the airports (by pressing enter).
---	--

	These tooltips are of high importance for the user, as this is what they see when they enter the dashboard for the first time. This is what will prompt them to immediately move onto the route section, so that they can start the planning process. This tooltip is also useful, because, after all the planning is complete, the user remembers that these previously disabled features are now enabled, prompting them to utilize them. The absence of these tooltips would see the user enter the dashboard for the first time and not know how to progress straight away, as they may even try to click around on different aspects of the route to try and make a flight plan but nothing will work.
	The fuel tooltip is also a good example of letting the user know crucial details about the functionality of the webtool. Without it, the user may still be able to deduce what to do regarding time allowed, but there may definitely be some ambiguity on the fuel policies and what to do with them, whether they're fixed or can be edited (which they can).

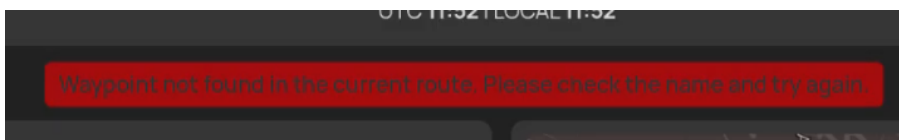
Error Tolerance

Confirmation message - **NOT MET**

The confirmation message that was supposed to appear when the flight is not saved and the user wants to start a new flight did not appear. This error came down to an error in the code, where I did not change the condition to look at what is stored in the key 'saved_json' and instead it checked what was in 'saved' which was the old key I used when the download pdf button did not exist yet. This would be a big problem for the user if they are not reminded, as they could accidentally clear all their flight data and have it lost forever without any way of retrieving it back.

Error alerts - **FULLY MET**

This was demonstrated at **8:20** where I was explaining what a natural error could look like due to the misinterpretation of what had to be typed in to remove a waypoint. Luckily, the error shows that the waypoint isn't present in the current route, helping the user know exactly what the issue is, being that they did not type what was exactly displayed in the route plan.

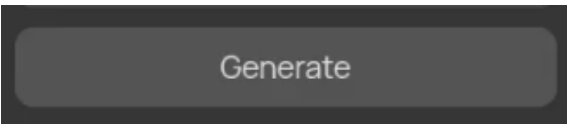
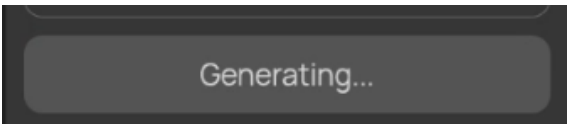
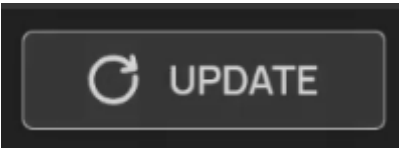
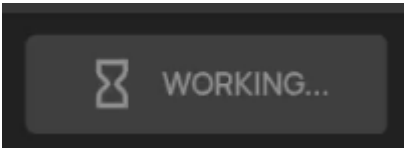
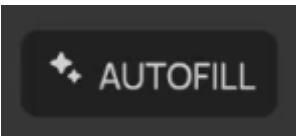
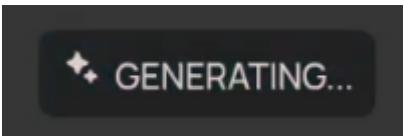
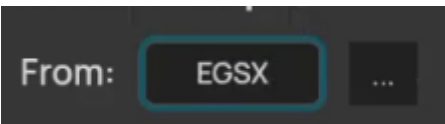


Without this knowledge from the error, the user will assume that it is an issue with the system and that it cannot detect the name of the navigational aid they were trying to remove.

Feedback

Button state changes - FULLY MET

Changing the state of the buttons to show that a process was happening in the background was crucial in the webapp, and has been implemented successfully for all of such processes. This feedback is often accompanied with button interaction being disabled, which prevents the user from activating the process again whilst one is already taking place. If either of these fail, the user will not know at any point what is happening in the background and whether their inputs have done anything, except when the output finally appears. They could also break the webapp, as, due to their uncertainty, may repeatedly click the buttons to try and get a process to work. Some good side-to-side comparisons are shown below:

Consistency

Consistency has successfully been maintained throughout the webapp, with no minor differences in appearance that could draw the attention of the user. A non-consistent user interface could make it harder for the user to understand certain functionality and therefore degrade the way they interact with the webtool features and the general user experience.

- Navigation menu is always on the left hand side of the screen FULLY MET
- Button colours and sizes are consistent across pages FULLY MET
- The methods for searching for an aerodrome is the same for both the route and weather tab FULLY MET
- The method to input a value is the same throughout FULLY MET
- Colour scheme is consistent throughout FULLY MET
- Same general style throughout FULLY MET

Robustness tests: North Weald (EGSX) Local flight

Robustness Test 1

 AIVFR Robustness test 1.mp4

Robustness Test 2 (Manual)

 AIVFR Robustness test 2.mp4

As a result of the AI error, both tests use manual route creation.

Robustness test description (10)	Test Data	Expected Outcome	Successful test? (Test 1 Test 2)
How the webapp (particularly the AI) responds to having the same departure and destination airports	Same departure as arrival (EGSX-EGSX)	The AI should be able to recognise that it will be a local flight, and should be able to create the route as appropriate to ensure it starts and ends at the same place whilst still having waypoints in between.	UNSUCCESSFUL 1:20 UNSUCCESSFUL 1:32 Clean error
Real world route test and comparison	Real flight plan data from a flight I created using the SkyDemon software	The data, especially calculations, should be near or exactly matching the real data for the flight I planned to show the webapp could provide accurate enough data for supposed real-world application.	SUCCESSFUL 8:40 SUCCESSFUL 6:00
Response to quick inputs and/or edits	Any suitable input for relevant tab	The webapp should be able to continuously update the database without losing track of data and upon reloading all entered data should exist even after a quick change.	PARTIALLY SUCCESSFUL 20:10 PARTIALLY SUCCESSFUL 15:13 Clean error

Input validation tests	Erroneous/boundary inputs for the relevant input	Data inputs at the boundary should still be processed accordingly and the webapp should not break, and the correct error should be shown for erroneous data entered into any input.	SUCCESSFUL 14:54	SUCCESSFUL 17:12
Response to random clicks	Clicking on areas that are not buttons	All screens should do nothing when clicks made to non-button elements are made unless coded otherwise.	SUCCESSFUL 22:10	SUCCESSFUL 20:17
Navlog response to large route changes	Re-writing the entire route	The navlog should still contain all relevant waypoints for the route after making many edits to the waypoints.	SUCCESSFUL 23:57	SUCCESSFUL 25:39
AI hallucination test with weather	Not generating weather at the weather tab	The AI should recognise that no TAF's (at least, given METARs will always be available) are available and thus should not hallucinate the weather forecast . It should mention the absence to the user and warn them appropriately on how to ensure the weather will be appropriate in the future.	SUCCESSFUL 30:00	SUCCESSFUL 23:35
Response to a failed API request although input data is correct	Removing API keys to trigger errors/adjusting code to fail the fetch by force	The relevant error should appear telling the user an issue occurred and to try again/adjust their input	SUCCESSFUL 27:38	SUCCESSFUL 21:00
Responses to page refreshes during processing	Page refresh button	If a process is being executed and the page is refreshed it should stop execution, and data should not be lost/the app should not crash	SUCCESSFUL 28:44	SUCCESSFUL 21:20

Prompt injection attempt in route generation	Custom prompt	In the route tab the user should not be able to make their own prompts simply by writing in the box what they like. If anything the AI should stay on topic and, given the user prompt and global prompt don't match, it should ask for clarification via the justification box.	SUCCESSFUL N/A 28:00
User can plan a route using the Route Tab manually	Test route for robustness test 2 (EGSX - EG SX)	I should have the ability to use the add/remove waypoint interface to add waypoints to the route.	SUCCESSFUL 3:40 2:23

Comments on Robustness

Generally, the webapp remains mostly robust, managing to withstand many of the different robustness tests conducted. I will call it a general success, and I'm happy that there were even some moments where I did expect a failure but there was no failure!

Addressing the two failed robustness tests:

How the webapp (particularly the AI) responds to having the same departure and destination airports

This failed for reasons I still don't know as I am yet to look into it, but I did check what the console was outputting and is said that the value stored in the ai_justification key was 'null', which means that the API fetch did not happen and nothing was returned as a result, and the error occurred because the JS was trying to fetch something that was non-existent. I'll have to find out what is happening server side for this to happen, as there is something that skips processes due to the departure and arrival airports being equal to one another- my best guess is a comparison/OR condition written somewhere that doesn't account for what if the two parameters are the same.

Response to quick inputs and/or edits

The main reason this failed in the mass and balance section was due to the poor coding of having everything update when one input is made. This 'heavy' updating that occurs frequently gets stacked up and execution will not stop until all of the inputs are dealt with, and this caused an error with the database in SQLAlchemy as seen in the first robustness video.

As mentioned in both videos, the best way to deal with this is to use javascript completely and only update tables at once once everything has been calculated and the user has not been making any changes for an extended period of time. This will ensure that updates to the database are only made when absolutely necessary.

Evaluation of Success Criteria

Success criteria - Page 28

The specific iteration in development at which a success criteria was (or wasn't) completed is referenced, as well as the specific timestamp in the **Functionality test** that shows the relevant success criteria being met.

Criteria (13)	Explanation	Has it been met?	If not, why?
Dashboard	This will show all relevant flight details, such as the route, the expenses, the fuel, the planned mass and balance, weather, and full flight brief, All in one place.	FULLY MET See iteration 36:25	
Route generation	Using AI and the departing and arrival aerodromes inputted by the user, a route should be generated considering factors such as avoiding controlled airspace, fuel quantities, danger areas , TFRs , NOTAMs and more.	FULLY MET See iteration 8:00	
Use of APIs to fetch weather data	The use of APIs to fetch live weather data from any aerodrome. These will be presented to the user, and will also be used to inform how the route will be or how the full flight briefing is at the end. For example, the AI may suggest the flight is at high risk if thunderstorms are in the vicinity and the pilot has a chance of getting caught in them.	FULLY MET See iteration 16:04	
Use of API's to fetch NOTAM and TFR data	The same applies with warnings and temporary flight restrictions. These will be analysed by the AI and could influence the outcome of the final route/flight briefing.	NOT MET See iteration	See The issue with fetching NOTAM reports.

Use of AI (LLM)	Use of a large language model such as GPT-3.5 turbo to analyse route, weather, NOTAM data, etc. This will also be responsible for providing a full flight brief to the user at the end of a flight.	FULLY MET Route, Fuel, M&B, Brief 8:00, 23:40, 29:45, 35:35	
Decoding algorithm	A set algorithm that looks at certain keywords of a report and decodes them into readable user-friendly english language, that can then be presented to the user or sent to an LLM to analyse and inform the flight brief.	FULLY MET (Target still achieved despite no direct implementation) See iteration	Since the CheckWX API also provided a decoded version of the weather reports, an algorithm to decode the raw reports from scratch was no longer required. This was even better for my case as this ensures all aspects of the report are covered and the outputs are as accurate as possible.
Map with route Visualisation	Map showing the route lines, including surrounding airspace, Visual Reference points, etc.	FULLY MET See iteration 6:12	
Flight computer	A flight computer algorithm covering basic calculations such as heading, airspeed and groundspeed	FULLY MET See iteration 19:31	
Fuel, Mass and balance, and Expense calculators	These calculators should use data selected by the user to provide reliable values that can then be used to inform the route suggested by AI.	FULLY MET See iteration 25:05, 28:33, 34:05	
Full flight brief	The AI should output a full flight brief containing details such as weather, the departure/arrival airports, things/threats/warnings to look out for en-route, airspace and more.	FULLY MET See iteration 36:25	

Printable pilot Navigation log	A log containing details about the route, Minimum Safety Altitudes (MSA), airspeed, groundspeed, headings, ETA, etc, in a printable format such as png or pdf.	FULLY MET See iteration 21:25	
Settings menu	For changes such as units used for measurement, and other minor tweaks that are dependent on the user	FULLY MET See iteration 1:28	
Ability to save, edit and start new flight plans	The user can save the current flight plan in the cache, externally save flights in the computer and log new ones, allowing them to log several flights and edit older ones, etc.	PARTIALLY MET See iteration 36:35 -> end	Whilst I could save the flight data as both json and pdf, the pdf file was missing the expenses and brief on the first download and it had to be re-downloaded again in order for them to appear, whilst everything else appeared nicely immediately.

Addressing Unmet Features in Further Development

Unmet Success Criteria

Use of API's to fetch NOTAM and TFR data

To deal with this issue in the future, I'll have to consult official sources, as no public APIs exist for this type of data. This may or may not require a subscription, and even after this, I'll need to either find an open source parsing algorithm or develop my own to parse the data into more user-friendly readable formats. NATS UK provides NOTAM and TFR data for the UK so I will start with them and see what's involved, and depending on the complexity I may or may not decide to fully incorporate this in the future. If, for instance, there are a lot of legal and licensing implications, as a sole developer I may restrain myself from going that far to get this data, and instead choose not to include this feature into my webtool, focusing on just the weather, route and performance values.

Ability to save, edit and start new flight plans

The error involved here may be fixable by looking at how the items are saved when they are generated, that being whether they are saved on time (before pressing download pdf) or not. I can test this by having a separate route leading to a HTML page with all my data for the pdf, and checking its state at all times (this is the method I was using when developing this pdf styling in the first place). Then from this it will be quite simple to recognise at what point the data reaches the pdf format and the cause of the error should be able to be pinpointed without much issue.

Unmet Usability Features

Icons

To fix that misinterpretation risk with the buttons in the navigation log, the quickest way would be to utilize the built-in tooltip options in HTML, so that after hovering over those buttons for an extended period of time, the relevant function will show in text. Being honest, I may not need anything more than this, but if I wanted to perhaps standardise the way tooltips look like in my webtool I may code an artificial tooltip that appears in the same conditions, but will be tailored more to the styling and colours of AIVFR, which does maintain consistency throughout the webtool.

Confirmation message

The solution to this issue is simply to change the variable reference from saved to saved_json. It was so quick and such an important feature that it made sense to quickly fix this bug now, as it was not a major issue like the ones we've seen that will require a little bit more time and effort to fix!

main-menu.js	main-menu.js
Erroneous version in usability test	Fixed version
<pre>//-----NEW FLIGHT async function NewFlight() { await fetch('/get-flight') .then(response => response.json()) .then(data => { if (data.saved == "False") { //checks if d alert("Your Flight Plan has changes th } }) };</pre>	<pre>//-----NEW FLIGHT async function NewFlight() { await fetch('/get-flight') .then(response => response.json()) .then(data => { if (data.saved_json == "False") { //checks alert("Your Flight Plan has changes tha } }) };</pre>

Evaluation of Limitations

These limitations were found in the development section, but have been **retained** in the software for now due to their complexity in solving them. They are the main limitations of the finished product, and are all explained below:

Inability to add a waypoint from any index/point in the route

Why is this a limitation?

This was a problem as it is a key feature that should be available to the user when editing the route. Essentially, this means that for a route **A→B→C→D**, whilst I can remove any of those points at any index, I can't add on point **F** to be after point **B**, I can only do it after the last added point, which is point **D**. The immediate repercussion of this is that I must remove points **D** and **C** completely before then adding **F**, and then adding **C** and **D** again. This makes planning routes quite difficult as often the user should be able to have uncertainty on where to go, quickly adjusting the route at any point and seeing if this fits their own requirements. But if I have a route with 7 waypoints and want to quickly add a point after point 2, it all of a sudden becomes a long process and the user now has to take extra care to avoid having to make such efforts just to add new waypoints.

What improvements/changes can be made to deal with this?

The solution I have been considering involves the use of mechanics used in that little expenses section in the dashboard. Since I can click each of the items, I could perhaps have a selecting mechanism where I can click a waypoint to add from that waypoint, otherwise the next added waypoint will be added from the last added waypoint. This would require a remodelling of the waypoint selection UI but it offers more selectivity and more fluid route management than just typing things in. For instance, even later down the line I could adopt a system that allows the user to just drag the little button containing the waypoint and place it anywhere they like in the route, allowing for extremely quick rearranging and adjusting of the route!

Entire route must be reset to change departure/arrival airports

Why is this a limitation?

This is another thing where the user has to lose progress to make a small change, but this is more significant. The source of all of these issues is that once something like a departure/arrival waypoint changes, so many other things need to change too: such as the route, weather, and more annoyingly, the navigation log. This would take a long time to process if rushed, which is why they were left in the first place, but they really affect user experience as they must go through the whole planning process extremely slowly ensuring they don't fall back making edits.

What improvements/changes can be made to deal with this?

For this issue specifically, I would have to completely re-write the route initialisation algorithms (so the route no-longer initialises after adding the arrival aerodrome) and perhaps re-define how a route is 'initialized' and what processes take place when this happens. Before all of this, I would need to also re-work the app for efficiency in updating values (covered in a separate limitation), as, if all of this aims to improve fluidity for the user, the absolute minimum should be that when the user does something, things happen there and then.

AI-related limitations

Why is this a limitation?

This is currently the biggest limitation in my opinion, as it affects the reliability of the webtool as a whole. Whilst the initial limitation of what the AI does and doesn't do has been resolved, the issue of what the AI output still exists, and this is particularly evident in the route generation, as the AI constantly fails to generate a route that avoids controlled airspace. This makes generating routes near controlled airspace pretty difficult, and, on top of the route editing limitation, totally makes this feature unusable for this purpose, unless the route in general is not near any controlled airspace to begin with.

What improvements/changes can be made to deal with this?

The solution I imagine would require the AI to have some sort of access to where the airspace boundaries are actually geographically located. If this is the case, then even a lower model could very easily avoid controlled airspace without an issue. How to do this... I don't know. The waypoints stored in the OpenAIP database don't have airspace data attached to them at the time of writing, just the airspace itself which can get pretty large... maybe an advanced AI that has access to the airspace map and locations only could be developed to then help deduce whether the route is in controlled airspace or not. There will definitely be a lot of digging to do.

Slowness with calculations

Why is this a limitation?

This is another limitation I want to get rid of as quickly as possible. This involves having things process much faster than they do now, because currently, even with good internet, some things update way too slowly for my liking, especially in the fuel section and mass and balance sections, which have a lot of calculations. The post development tests show you how the fuel values take a while and update one by one, or how the CG table does not update immediately after changing a value in the loading table. This (once again) affects the fluidity of the system, and, accompanied with a poor internet connection could result in a pretty bad user experience as you could imagine.

What improvements/changes can be made to deal with this?

The ideal outcome would be what is shown in this example of the **E6BX** app:

 **E6BX speed calcs example.mp4**

With things just being calculated extremely quickly with no delay. One of the reasons E6BX can achieve this is because it isn't storing any of the data, so everything is simply javascript which executes immediately, and there is no storing of values. I, on the other hand, have to do so, but I have a good idea which could solve the need to worry about this issue.

The solution would be to update values after the user leaves the page, or after a specific interval. This means that the javascript will run freely with no updating values at all (this is where the slowness comes in due to the need to contact the server) and only afterwards will the system update everything in the background at once. This makes it much quicker because the user no longer has to wait for the database to update before the next element is updated on the screen, which should give the user the ability to get that 'instant' feel that is demonstrated in the video.

Split text in the brief pdf

Why is this a limitation?

This refers to this happening in the brief pdf:

time.
R - Runway Lengths
Blackpool (EGNH)
Blackpool (EGNH): Runway 10/28: 2446 m / 8039 ft (max) Runway 13/31: 1267 m / 4160 ft

Consideration: Per regulation holding/diverting, especially <u>plan for direct-to-alternate</u> T - Takeoff and Landing Per
Weights: MTOW/MLW: 1600 lbs Takeoff Weight (Planned): Landing Weight (Planned):

Which can become a legibility issue if it occurs with other data, and also could make important data completely illegible to the user.

What improvements/changes can be made to deal with this?

The best way to deal with this issue is to not use html2pdf at all, and use a tool like **puppeteer** instead, which allows very clean html to pdf output. It does require some setup (npm for instance) but it's nothing I can't do with a bit of research. This is a limitation that I am not quite worried about compared to the others.

How were initial limitations handled?

Limitations[↗]

The use of Artificial Intelligence

The AI integration limitation in this project has been successfully overcome, as it ensures that no critical values are deduced by the AI. Since NOTAMs were deprecated, the best example of this is the weather tab, where the embedded decoding algorithm that comes with the API takes precedence over AI, as AI hallucinations in this regard are simply unacceptable. Where they are acceptable, however, are in other applications such as suggesting a route, where the AI can generate the route and have it displayed on the map, along with its own justification for picking that route. With this the user can then adjust this data to their own needs, which is key in making sure that the AI does not have the final say. The same applies to the time allowed inputs in the fuel tab- they are filled into <input> fields that can be adjusted immediately by the user should they not be happy with what the AI has provided.

The second limitation I mentioned regarding AI was the cost. This hasn't affected me as much as I thought it would, as I am the sole user at the moment and so no one else would be utilising the API and so rates remain in what is freely available.

Technical Limitations

Here I talked about the requirements of an internet connection as a limitation to some users as they won't be able to access it without one. This is not at all a problem, given that it is an expectation as it is accessed via a web browser, which requires an internet connection in the first place, so by necessity, this limitation can be disregarded.

Time constraints

Here I explained the issues with time and the possibility of me not being able to complete this by the deadline, and that a minimum viable solution would have to be crafted in order to ensure the success of this software, which is defined by the success criteria and essential features. This limitation can also be disregarded, as I have exactly a week to submit this at the time of writing this subsection, which is plenty of time! I worked quite a lot on this throughout the approx. 1yr I had to develop this, and I am extremely grateful that I have managed to complete the development of this ambitious project before the deadline!

Resources and experience

This limitation was overcome mainly through the inclusion of what was covered in section 1, and section 5 of development, with more emphasis on the latter section. In both these sections, I used tutorials to help me learn how to use different aspects of flask and Jinja2, as well as doing sample tests for everything I needed to include in my project regarding setting up the session, changing data, etc. Once this was done, all the remaining tabs were significantly easier (albeit some points of development were still pretty challenging) to do as the core foundation of what needed to be done was already learnt, greatly decreasing the time it took to develop this software.

Maintenance issues

Deployment issues

Since I decided to deploy this software (aivfr.co.uk) If I want to keep it up there I will need to be able to maintain it and keep it accessible to the public for use. To do this, I'll need to consider things like:

API rates and limits, which will have to be implemented to ensure users cannot repeatedly abuse the APIs present within the software, costing money and also possibly (on the worse case) having my key terminated due to too many constant requests without any remedial actions taken, which would be reflective across the entire system for all users... not good.

Deployment costs. As I deployed my software using [Railway](#), which requires a monthly subscription. Whilst relatively cheap at the moment, It obviously builds up to a commitment cost over time and funding would eventually be required to keep this software running if I choose to do so. Some ways I may do this would be to make this project closed source and ask users to pay to access certain features, perhaps more AI features as this would be more necessary to the user. Or, more easily, I could have advertisements on my product from which I receive funding from to pay for deployment and website improvements/maintenance.

Server maintenance

If I decide to make this a user based project with all users having access to their own array of flight plans I may need some sort of server which will be used to run the server-side requests and store user data. This comes with its own issues relating to setup, maintenance and cost of storage, but could prove to make the usability of the product much faster if it has a positive effect on the speed of the software. Perhaps remote storage is another option that may put some weight on me resource-wise, but it poses another reliability issue as that service must stay operative in order for my website to continue working properly.

API reliability and cost

The APIs I use should be running nominally at all times, but sometimes, they won't be. There was one point through my development that the CheckWX API actually failed and I couldn't perform the weather tab tests I was doing at the time. The best way to deal with this issue is to provide proper user communication and have a system that alerts the users live that a certain feature is now no longer available temporarily. An alternative could be to have 2 APIs running for the same feature, such as Gemini and ChatGPT for AI generated outputs, so that when one fails, the other will take place until the previous one is up and running again.

Another API issue is related to costs, and with repeated use I will need some sort of funding. The methods I could use for this are the same as the aforementioned methods explained in the deployment issues.

User feedback

User feedback is an essential part of maintaining any project, as sometimes as the developer it is hard to spot bugs until an update is published, and only then will the users find them and report them for me to work on. There are a number of ways I can implement this. The quickest way that could be done in like 5 minutes is to implement a google form allowing users to submit any bugs/feedback, and a button/anchor in the webtool that the user can press to go to said form and fill it out. More sophisticated forms could be GitHub

discussions, where for each version I can open up a discussion so that users can comment on the changes and bugs they found, or, if the community is really large, things like a server on Discord could even be implemented, which are greatly advantageous for user-developer communication. This will ensure I can ensure the software is always meeting the users minimum requirements whilst also providing the opportunity for users to request potential features that could be implemented in the software in later versions as well.

~ ~

RESOURCES & CREDITS

Thanks for reading!

Full Code

The full code for v1.0.0-beta (i.e what this NEA covers) is available in GitHub via the link below:

<https://github.com/TheEasyPilot/AIVFR/tree/v1.0.0-beta>

Otherwise, the project is available on Github by following the link below. There may be updates.

<https://github.com/TheEasyPilot/AIVFR>

Testing Videos

All testing videos for this project can be found in the shared drive below.

[AIVFR development tests v1.0.0-beta](#)

APIs



OpenAIP - Core API
(Documentation)



CheckWX API



API Ninjas

Mapping



[Leaflet Measure Path](#)



[OpenStreetMap](#)



[OpenAIP - Tiles API](#)



Useful Tools

[Geodesy](#) (For bearings)

[Html2pdf](#)

[Marked.js](#) (.md to html)

[Keep-a-changelog](#)

[SemVer](#) (semantic versioning)

libraries

[openai](#)

[pint](#)

[flask](#)

Tutorials and aids

[Flask](#)

[Python Flask Tutorial - FULL COURSE](#)

[Flask Full Course: Build Stunning Web Apps Fast | Python Flask Tut...](#)

[build a meme Python website \(Flask Tutorial for Beginners\)](#) (used specifically for web directory setup)

[How to use Flask-Session in Python Flask - GeeksforGeeks](#)

[Flask SQLAlchemy Tutorial - Geeks for Geeks](#)

HTML/CSS/JS

[W3 Schools](#)
[MDN](#)
[Stack Overflow](#)

AI use within development

Explicit aid from AI has been used to:

- Create the AI prompt for the route generator (justified in the relevant [section](#)). ([ChatGPT](#))
- Help adjust the disclaimer after the initial design. This was simply to ensure it met legal standards. I ended up adjusting it again after this as is seen in the post-development, and below. ([ChatGPT](#))
- Assist in making repetitive code corrections/writing repetitive code to save time. ([Github Copilot](#))

Please see below all the references to ChatGPT conversations:

1. Generating the AI prompt for the route:

<https://chatgpt.com/share/69725d7f-71bc-800d-a2d5-4e8890c3ed46>

2. Disclaimer modelling

<https://chatgpt.com/share/697279df-0424-800d-8c6b-ff5a14cda67d>

Finally, Please see below a live demo I recorded mid-development of using github's copilot AI when correcting an error that meant I needed to change several values.

 Copilot_AI use.mp4

Disclaimer (As seen in webapp)

Final for v1.0.0-beta and below. Consult the software or Github for the latest version.

This web application is an experimental, non-certified project that uses AI-generated content. The information provided may be incomplete, outdated, inaccurate, or misleading.

This tool is provided strictly for learning, demonstration, and educational purposes only. No output from this system is suitable for, or may be relied upon for, real-world aviation use.

This application must **not** be used for:

- Operational flight planning or navigation
- Weather assessment or decision-making related to real flight operations
- Airspace or regulatory compliance

By using this tool, you acknowledge and agree that:

- The application is in the **beta** stage of development and may contain bugs, errors, or incorrect data.
- No information provided by this system is authoritative, certified, or approved for aviation use.
- The pilot in command remains solely responsible for all planning, decisions, and actions related to any flight.
- I, the developer, accept no responsibility or liability for any decisions, actions, losses, or outcomes arising from the use of this tool.
- Any use of information generated by this application, whether accurate or not, is entirely at your own risk.

This tool must NOT be used for real-world flight operations under any circumstance!

