An App to Automate the Creation of Load Manifests for AK Fire Crews

Dawson Nash, Ben Brown

CSCE – 470

December 12th, 2024

Table of Contents:

An App to Automate the Creation of Load Manifests for AK Fire Crews

Abstract

In Alaska, fire crews create detailed manifests listing all crewmembers and gear scheduled to be flown to remote locations to combat wildfires across the state. To assist with this high-stress and critical task, we developed an app to automate the manifest creation process. The app enables users to input key information, including crew details, helicopter weight and seat limits, and preferences for how the loads should be organized. Based on this information, the app generates a manifest that users can review and export for sharing.

1. Introduction

In the world of wildland firefighting, having sufficient time to prepare for operations is a rare luxury. From responding quickly to emerging incidents to deploying via helicopter to isolated locations, it is a fast-paced job with many stress-inducing challenges. In Alaska, fire crews frequently travel to remote fires far from any road system1, relying on aircraft to transport personnel and cargo safely and efficiently. A critical aspect of aircraft travel involves creating detailed manifests. These documents list the weight of each crew member and piece of equipment to ensure compliance with the helicopter's specific weight limits.

Currently, this manifesting process is done entirely with paper and pencil. It can take over an hour to complete due to the complexities of transporting a crew to and from a fire. Standard operating procedures must be followed, and manifests must be tactically organized to ensure both crew safety and effective firefighting. The process becomes even more complicated when helicopters or operational circumstances change unexpectedly, requiring crews to restart the manifesting process from scratch. These disruptions can cause delays and add unnecessary stress to time-sensitive operations.

2. Project Overview

To address this challenge, we have developed a digital manifesting app that automates and streamlines the weight calculation and manifesting process for fire crews. The app allows users to input the weights of crewmembers, tools, and equipment, along with specific preferences or requirements, such as ensuring water is included on each load, the crew boss is on the first load, or leadership is evenly distributed. By automating this process, the app saves valuable time, reduces human error in weight calculations, and provides the flexibility to quickly adapt to last-minute changes in helicopter assignments or operational needs. Most importantly, it reduces stress for firefighters operating in high-intensity environments. Rapid and accurate mobilization is critical, as crews play a pivotal role in suppressing wildfires before they grow larger, more dynamic, and costly[2].

The most important challenge of the manifesting app was finding the right balance between automating the task of manifesting and allowing fire crews to maintain full control over the algorithm's functionality. This balance makes the use of load preferences essential to the app's success. For the app to be valuable to Alaska fire crews, it must create manifests based on their specific, self-determined needs. To address this further, we incorporated a manual manifesting tool. This feature allows fire crews to build a manifest themselves within the app using input crew and gear data, while the app provides guardrails to ensure a valid manifest. This dual functionality offers crews the flexibility to create manifests based on their need for fine-tuned control or to prioritize the algorithm’s automation for efficiency.

3. Project Client and User Base

The primary client who was consulted throughout this project and provided consistent feedback on the app’s core functionality and requirements was Bryan Quimby. He has extensive experience in wildland fire, ranging from being a simple crew member to being the commander of large, complex fire incidents, having to interface with many state and federal resources. Quimby was the ideal point of contact for a project of this nature, as aviation operations in fire, to include manifesting personnel and cargo loads, has been a large part of his career in wildfire. The intended user base for the Fire Manifest App includes users similar to Quimby, a niche set of individuals, fire personnel, who will be conducting air operations on a daily basis. For this reason, while the intent is to make the app intuitive to any user, there is still a degree of workplace-specific knowledge that will allow the greatest use of the app’s functionality.

4. Requirements

The project requirements are loosely defined but center on creating an app that supports Alaska fire crews in the manifest creation process. The goal is to automate as much of the process as possible while ensuring that the fire crews retain a strong sense of control over the sorting algorithms.

4.1 Functional Requirements

1. Allow users to input crewmember and gear information tailored to their specific crew requirements.
2. Enable users to define preferences for how manifests should be structured, including the allocation of crewmembers, teams, and gear across different loads.
3. Include an algorithm to process user-defined preferences and generate a load manifest schedule (trip) based on those preferences and crew specifications.
4. Provide users with the ability to manually create trips, offering them full control over the manifesting process.
5. Support the exporting of manifests for sharing with team members and helicopter staff, ensuring the validity of the manifests and adherence to the planned schedule.
6. The app should be built for mobile devices first and foremost, as manifest creation is often done in the field or on the move
7. The app should be multi-platform, on both IOS and Android, to meet the requirements of all potential users based on their needs.

4.2 User Requirements

1. The software must include training instructions to ensure fire crews can effectively learn how to use the app for creating manifests.
2. Automated manifests generated by the app must be reviewed by a crewmember. Given the significant impact of potential mistakes in the manifest, it is essential that users monitor the outputs, even if the creation process is largely automated.
3. While designed for a niche user base, the app should still be intuitive and easy to navigate.

5. System Design

The design of the app followed its intended core functionalities: the input of crew member and gear data, the creation of manifests, and the user's ability to customize how those manifests are created.

5.1 Architecture

The framework for the app is built on Flutter, a software development kit created by Google and written in Dart that allows for deployment on many different platforms that are critical for the app's user base like Android and iOS. As a Flutter Project, the app was developed through Android Studio, which allowed easy testing on Android devices and could be ported to Xcode for iOS testing. All core functionality of the app to include data management, user-interface retrieval, or algorithms are run exclusively within in-device storage, as being able to use the app in its full extent is an absolute necessity for crews without internet service in remote locations.

The specific architecture of the app, which arose mostly organically from the design process of meeting core functionalities, can be described as layered, modular, and event driven. The app is tailored heavily to mobile platforms and has a strong focus on data-driven decision-making and user-centric features, i.e., the app is designed for a specific task, sorting, and the user is in command of everything else in-between. This architecture can be split between three distinct layers. The first of these, the presentation layer, manages the user-interface and interactions by using stateful Flutter widgets. It is responsible for providing real-time viewing of both input and created data. The second of these layers is the logic layer, which handles all core app logic to include the load-sorting algorithm, weight calculations, user preferences, and the interface between real-time data updates on the user's screen to ensure clean data is being passed to app functions. The third and last layer is the data layer, which manages data persistence, retrieval, and updates. This consisted of both temporary data storage on each app launch, as well as long-term storage through Hive, Flutter's native NoSQL database. In addition to being a layered architecture, the app was also modular in design. Each phase of the development process focused on a specific aspect of the app such as the Crew/CrewMember/Gear, Trip/Load, TripPreference, and Algorithm modules. Each of these are divided into well-defined sections that allow for better maintainability and scalability if need be. The architecture is also event driven. It relies on events like user interactions for adding or removing data to trigger user interface updates or logic execution. If data is changed, the Hive database also needs to automatically update and provide that information back to the user.

5.2 Data Design

The data management structure for the app follows multiple class-based, hierarchical structures. The three types of data managed fall into the following categories: Crew, Trip, and Trip Preference. In the Crew structure, as can be viewed in Figure 1, the user's crew is organized into both crew members and gear. When crew members are created, the user must define their name, flightweight (body, backpack, and personal protection equipment weight), and whether they have any personal tools (tools that never separate from the crew member). When gear is created, the user must define its name, weight, and whether there is more than one.

Crew	CrewMember[] crewMembers
	Gear[] gear
	int totalCrewWeight
CrewMember	String name
	int flightWeight
	int position
	Gear[] personalTools
Gear	String name
	int weight
	int quantity

Figure 1

In the Trip class structure, as can be seen in Figure 2, the data for each manifest is stored. A Load object can be thought of as one single manifest, while the Trip can be thought of as the set of manifests that are required to get a whole crew from one place to the next. These objects are created in the manifesting process, which happens following user-triggered responses in the *Quick Manifest* and *Build Your Own Manifest* pages (Figure 4). The user must define a Trip Name, the allowable weight of the helicopter, and the number of available seats. Then, depending on user preference, Load objects are created via algorithm or by the user themselves.

Trip	String tripName
	int allowable
	int availableSeats
	Loads[] loads
Load	int loadNumber
	int weight
	CrewMember[] loadPersonnel
	Gear[] loadGear

Figure 2

The last class structure, as seen in Figure 3, is the Trip Preferences. This data structure is created under the *Trip Preferences* branch under *Edit Crew* (Figure 4), where the user first creates a Trip Preference object. This object contains both PositionalPreference and GearPreference objects that reflect how the user wants their crew members and gear sorted during the manifesting process. Each preference object contains a load preference which can be either first, last, or balanced. This means that each crew member or gear item can be defined to go on the first or last load, and if grouped can be distributed evenly amongst all loads in a balanced fashion. This structure is by far the most complex and difficult to understand as it is the backbone input for the load calculating algorithm and must provide the user with any considerations they might think of while creating a manifest. To name a complexity, the CrewMembers array in PositionalPreferences had to be made dynamic as the user had to have the ability to define whether they wanted a single individual to follow a load preference (first, last, or balanced), or an array of individuals to be constrained by the same preference. This is especially important in cases where you have “Saw Teams”, two crew members responsible for chainsaw operation, who can never be separated and must be on the same load. With this Dynamic List data structure, the algorithm can recognize whether an object passing through is a team of individuals or not. There were many considerations like this which went into the design of the class structure. An example of a Trip Preference can be seen in the *Preference Loadout Screen*, found in *Section 11, User Manual*.

SavedPreferences	`TripPreference[] tripPreferences`
TripPreference	`String tripPreferenceName`
	`PositionalPreference[] positionalPreferences`
	`GearPreference[] gearPreferences`
PositionalPreference	`int priority`
	`int loadPreference`
	`DynamicList[] crewMembers`
Gear	`int priority`
	`int loadPreference`
	`Gear[] gear`

Figure 3

5.3 User Interface Design

The user-interface design, as can be seen in Figure 4, follows a top-down approach where all screens stem from the main page, *Main Menu*. All data creation and editing within the Crew/CrewMember/Gear and TripPreferences modules are done through the second layer page, *Edit Crew*. Here, the user creates their crew and defines how they want it sorted when traveling to and from a fire. This is where they can create and edit the CrewMember and Gear objects that are added to the global Crew Object. This is also where the user can create a TripPreference that is used during the manifesting process. Another second layer screen, *Manifest Page*, is the route the user takes for creating crew manifests. They have the option of performing a Quick Manifest, which uses the user-customized load calculating algorithm, or a Build Your Own Manifest where they can build loads one at a time without aid. Both of these manifesting options operate within the Trip/Load module where Trip objects are created and manipulated, however, this route utilizes all modules when passed to the load calculating algorithm that performs operations on all types of data. The last branch of the user-interface design is the *View Trips* page. Here, the user can view Trips created from the *Manifest Page* route, and export both individual loads and entire trips to official state and federally used manifest form PDFs.

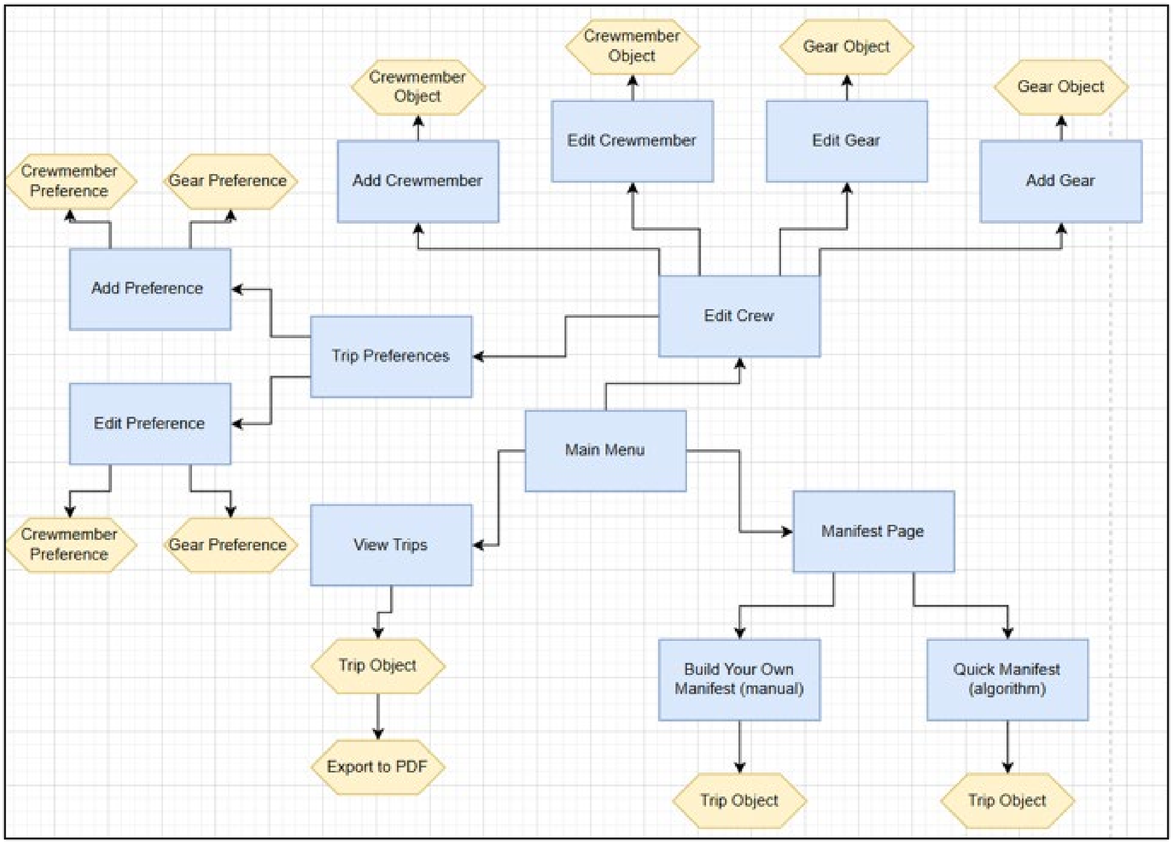


Figure 4

5.4 Algorithm

The backbone of the Fire Manifest App is the load calculating algorithm. This 408-lined function takes in TripPreference object and generates the necessary number of sorted loads to get a user's crew to their destination, may it be to the fire line or back home. From a high-level overview, the algorithm simply takes in the user's preferences, places those items and people on their respective loads, and then smartly sorts the remaining crew. For a more thorough breakdown, the algorithm works in the following fashion.

1. Inputs, Calculations, and Data Initialization

The algorithm begins by taking in both a Trip and a Trip Preference object (Figures 2 & 3). It then calculates and initializes the total number of loads that are required based on the total crew weight, the allowable weight of the helicopter, and its available seats. The more seats that are available, the closer the algorithm can get to the allowable for each load weight. The algorithm then creates copies of the crew data to avoid direct manipulation of the source data, and because once Trip Objects are created, they need to be isolated from any future changes users make to crew members and gear. Once a manifest is made, its data is meant to persist.

2. Trip Preferences

The algorithm then considers the user's preferences. It loops through each Positional and Gear Preference and places the crew member and gear objects into their respective loads. There are several considerations during this process. While placing Positional Preferences, it identifies whether there is an individual crew member or a team of crew members. If it's a team, it ensures that team stays together. For example, if the Positional Preference is "*Saw Team 1, Saw Team 2, Saw Team 3, Saw Team 4; Load Preference: Balanced*", it cyclically loops through each load placing entire saw teams on each load. For Gear Preferences, another crucial consideration is the quantity of items in a preference. If a quantity for an item is greater than one, the algorithm will create additional gear objects of the same item until the preference is fulfilled. This way a gear item can be stored as a single object and have a single *quantity* attribute that defines how many exist, as opposed to consuming memory with multiple objects of the same type. For example, if a Gear Preference is "*5-Gal Water (x2); Load Preference: Balanced*" the algorithm will start with a single *5-Gal Water* gear object, create additional objects until it reaches the designated quantity (x2) for that load, and then cyclically repeat for each successive load until there is no more *5-Gal Water* gear objects left in the crew inventory.

3. **Smart Loading and Shuffling**

Once the algorithm has considered the user' s preferences, it then sorts the remaining crew members and gear. It does this by first shuffling the crew members by position to ensure distributed crew member skillsets among loads and avoid grouping identical positions on the same load. A user wouldn't want all of their medics, for example, to be on the same load; they would want them distributed in case the crew gets separated. There is also an element of randomness introduced into this process. Once positions have been shuffled, the algorithm then cyclically places the remaining crew members and gear onto loads. The cyclic approach guarantees that load weights will be as close as possible given the user's predefined constraints. It also guarantees that, even if the user didn't define it in their Trip Preference, gear items will be evenly distributed, which balances out critical items with larger quantities like water and food.

6. Software Development Process

The development process for the Fire Manifest App was loosely defined at the beginning of the academic semester and followed an Agile approach that contained five total sprints. The end goal for the app revolved around three concrete objectives: be able to input crew and gear data, be able to input preferences for how the user wants to sort loads, sort crew and gear into loads based on preferences and be able share those loads. These goals, while concrete, seemed very ambiguous in the precise method for how they would be accomplished at the beginning of the semester. This was especially true with the user-defined preferences, as there were so many considerations to include in this potential algorithm. By breaking down the project into several sprints, it was able to come to fruition. The goals of each sprint are as follows:

Sprint 1

During this sprint, completed on October 1st, the focus was on setting up the project and ensuring all required packages, dependencies, and software were downloaded to begin creating the UI and data design. This is the time when the initial app pages were designed and implemented on Flutter, which required research into the Flutter development kit and the Dart language. These initial app pages included the crew member and gear input pages, albeit the backend data structures were not yet implemented.

Sprint 2

During this sprint, completed on October 14th, the focus was on building the initial Crew data structure to support the front-end UI. Given that the initial pages were built already in

Flutter from sprint 1, there was good boiler plate code to start generating the UI framework for a lot more pages. Once the UI coding became easier to understand, which took up a large part of the initial sprint, time could be spent on developing the logic to connect data between pages and provide functions to front-end UI widgets. This is also the sprint that the *Quick Manifest* page was built.

Sprint 3

During this sprint, completed on October 28th, a basic template for data would be created and stored by the user was already implemented, so the focus became on allowing them to edit their created data. Additionally, this is the sprint that pseudo code for the load calculating algorithm and potential design for the Trip Preference data structure was developed. Both of these tasks were very abstract during the initial design phase of the project so most of the time spent during this spring was less on coding and more back on the white board and notebook trying to make this concept make sense.

Sprint 4

During this sprint, completed on November 11th, a basic sorting algorithm was created and generated the first manifest, and shortly after the Trip Preferences feature was actualized. In addition, the *Build Your Own Manifest* page was created which allowed the user to create their own manifests from scratch. A lot of the time spent during this sprint was on turning the pseudo code of the Trip Preferences and load calculating algorithm into actual code.

Sprint 5

In the final sprint, completed on November 25th, all focus was spent on applying final touches to the app to prepare it for the final presentation. This included refining the algorithm and bug testing to make sure the core functionality was where it was intended to be.

7. Results

7.1 Results, Analysis, and Strengths

Given the timeframe for the project, a little over 3 months, and the resultant functioning app, the final product turned out better than could have been expected. The app still needs a significant amount of bug fixes, user-interface scalability restructuring, and algorithm testing, but all the initial goals for this project were met. The user is able to input their crew data, define how they want them sorted, and generate instantaneous manifests that they can download and share with their crew. The app has very clear definitive strengths that are the result of meticulous data management design. One of these is how well the algorithm sorts based on user preferences. Yes, there are still clear bugs and the occasional crash with certain edge-case inputs, but when it does sort, it sorts just how it was intended: like the thought process of a wildland firefighter. The Trip Preferences are simple, intuitive and highly effective. Their incorporation into the algorithm, as well as the modular design of the algorithm, also makes understanding how sorted items are placed into loads easy to understand from a developer point of view. This allows easy debugging if something goes wrong in the manifesting process.

7.2 Improvement and Future Work

There are several items that do not work as intended quite yet and require further work. Given the complexity of the algorithm and the required data structures, managing data within the app came with a lot of backend work to ensure the user could not break the app or, more importantly, the app did not generate wrong calculations for a task as high stake as aviation operations in wildland fire. The nature of the user base's line of work is a testament to how much more rigorous testing still needs to be done for releasing it for official use. On the minor improvements to the app, less focused on functionality, there are a lot of quality-of-life features that need to be implemented. This includes items like dynamic user-interface scaling for mobile devices of all screen sizes, adding a home button, re-imagining the allowable slider to allow for keyboard input, etc. On the larger improvements of app function, some core features need a lot of work. The foremost is the ability to edit created trips. This would be a highly beneficial feature to allow more freedom in manifest creation. Currently, once a manifest is made, it is set in stone unless it's deleted and reattempted. Another big improvement and challenge that was not overcome was the ability to store Trip Preferences permanently within the Hive database. Hive does not allow for arrays of a dynamic type, which is used in PositionalPreferences, and changing this would require a lot of app restructuring.

8. Conclusion

The Fire Manifest App project this semester represents a potentially large improvement in streamlining the manifesting process for Alaska fire crews. By automating weight calculations and load assignments while maintaining flexibility for manual manipulation of loads, the app addresses critical challenges in fire aviation operations. It reduces the time, stress, and human error associated with creating load manifests, enabling crews to focus on their primary mission – fighting wildfires effectively and safely.

Throughout development, key lessons were learned, particularly around balancing automation with user control. The inclusion of both the algorithm-driven and manual manifesting options ensures the app remains intuitive and adaptable to unique operational needs. Additionally, the app's modular architecture and well-structured data hierarchy lay a strong foundation for any future enhancements, scalability, and debugging.

However, challenges still remain, such as refining trip editing capabilities, dynamic UI, and addressing edge-case bugs that break the limits of the load calculating algorithm. Rigorous testing and feedback from wildland firefighters in the field are going to be critical for guaranteeing the app's reliability in high stake scenarios.

Ultimately, the Fire Manifest App came into fruition after several months of planning and design and is a practical tool that has the ability to support fire crews in Alaska. With continued work, the app has the potential to become a very essential resource for assisting air operations in wildland fire and ensuring the crews can focus on their primary mission.

9. References

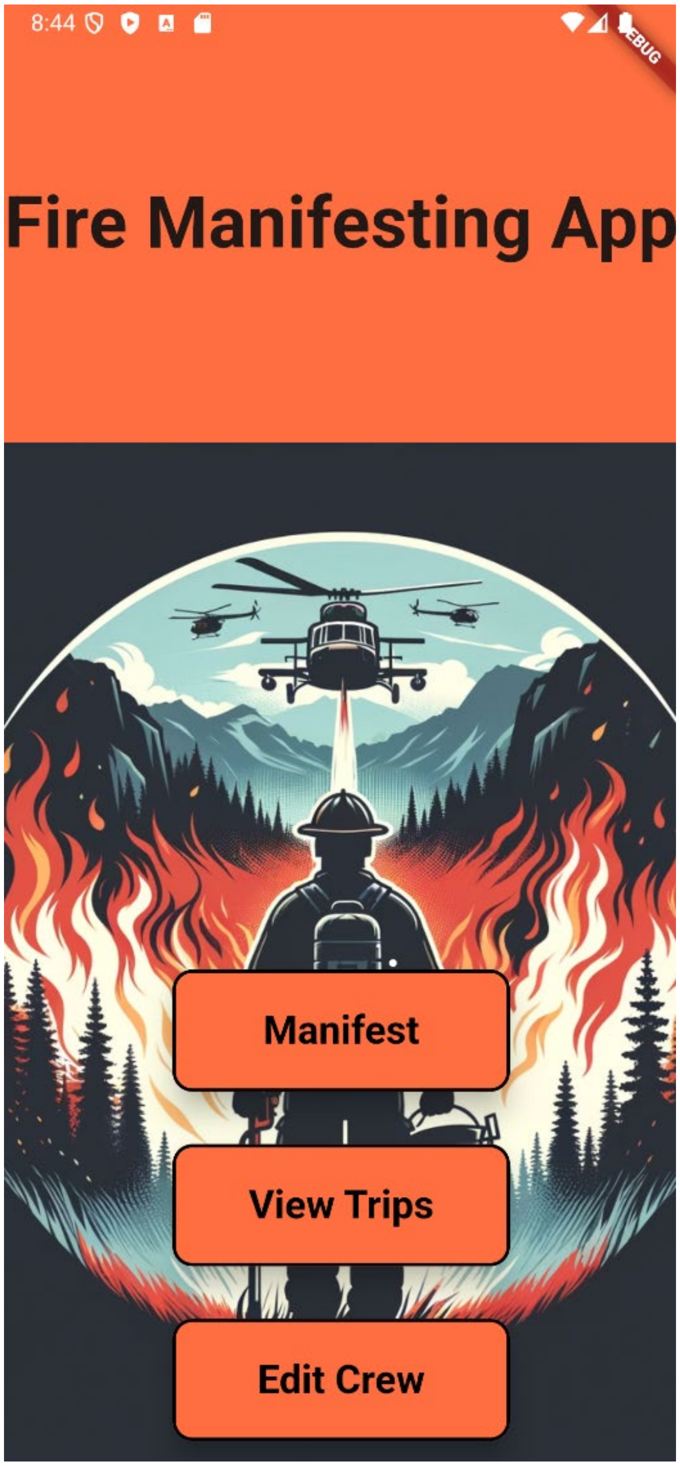
1. Alaska Department of Natural Resources Division of Forestry & Fire Protection, *Fire Assignments,* 15 September 2017, https://forestry.alaska.gov/fire/assignments#map
2. Alaska Department of Natural Resources Division of Forestry & Fire Protection, *Aviation Program,* 15 September 2017, https://forestry.alaska.gov/fire/index

10. Actions Taken from Code Review

Severity	Defect	Resolution
L	increase from 2 to 4 spaces for indentation	Not implemented, all code was indented per Flutter development kit and Android Studio
Q	Is ceiling what you want?	Yes, as you need as many or more loads than the sum of the weight needed to be manifested
L	error check if currentLoadWeight > maxLoadweight	Already implemented in load calculating algorithm. This check determines when to move to the next load.
L	reduce similar code in case statements	Not implemented because we did not have the time. A lot of the case statements were similar, but each unique enough to cause a good headache to try and restructure.
L	magic numbers for cases -- replace with named constants	Same as previous defect. This would likely be a easy fix, it just was not placed as high priority.
Q	crew is global variable?	Yes, one crew object that acts as the user's crew.
Q	consider a class for load preference?	Did not implement, it works better as an attribute of Positional and Gear Preferences for our purposes.
M-H	nested for loops; complexity concern	We decided that due to the small scale of the sorted data, complexity was not a big enough concern to justify rewriting the algorithm to be faster. Especially due to the limited deadline and that the algorithm was sorting 'instantly' from the user perspective.
M	deep nesting	Same answer as above
L	rewrite as regular if statement	Did not implement. Did not see it as a priority, and code made sense to us.
L	shorten conditionals using a function or boolean	Similar to the reasoning for restructuring code for the case statements, this is a good idea and would help with readability, but it was placed on the backburner to get other features complete.
L-M	use of break statements within loops (switch breaks are OK)	The break statements were never taken out, but it is very likely that they are not necessary. The load calculator works with them, and a simple test could be done to determine whether they're needed.

11. User Manual

Home Screen:



The Fire Manifesting App has three main functions:

1. Input and Edit Crew, Gear, and Preferences

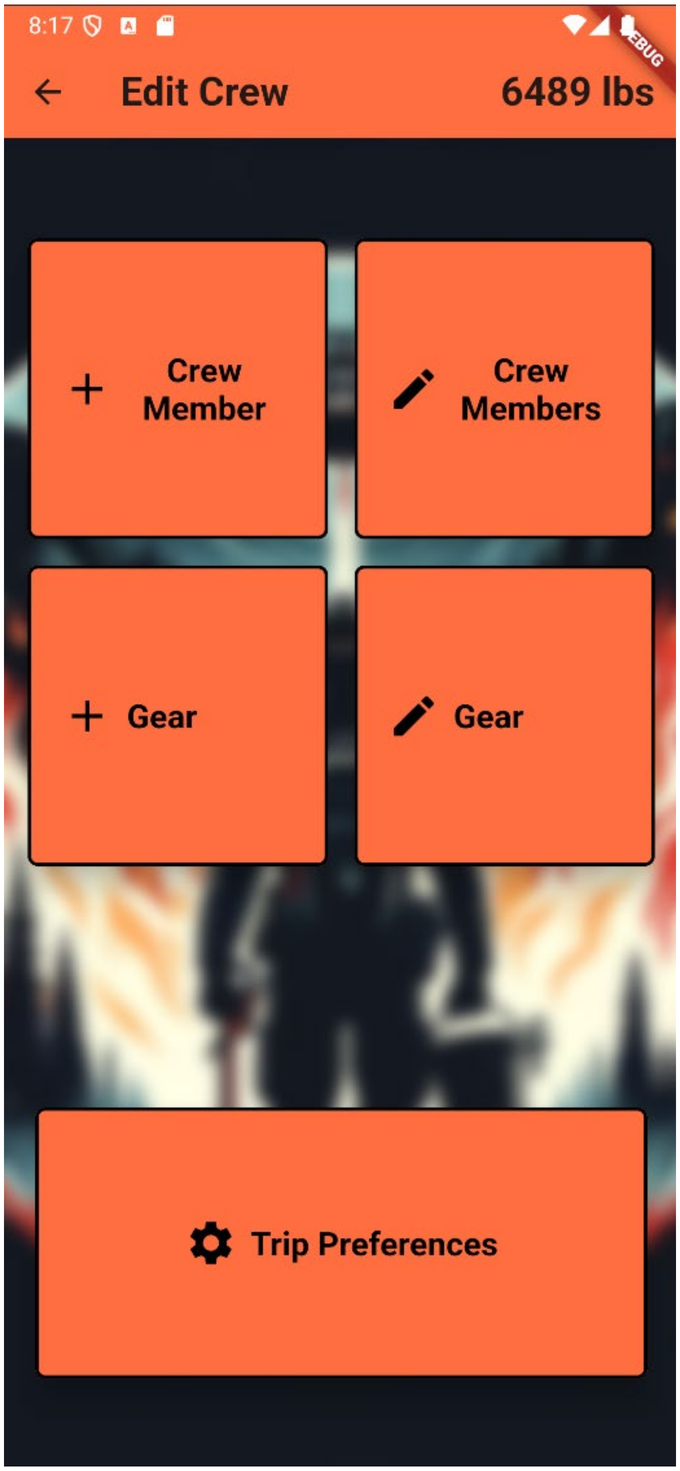
2. Create Manifests Based on aircraft Requirements

3. View and Export Trips

These functions are accessible through the three options on the home screen.

To navigate between sections, use the back arrow to return to the home screen before selecting a different section of the app.

Edit Crew Screen:



The edit crew screen includes five option menus:

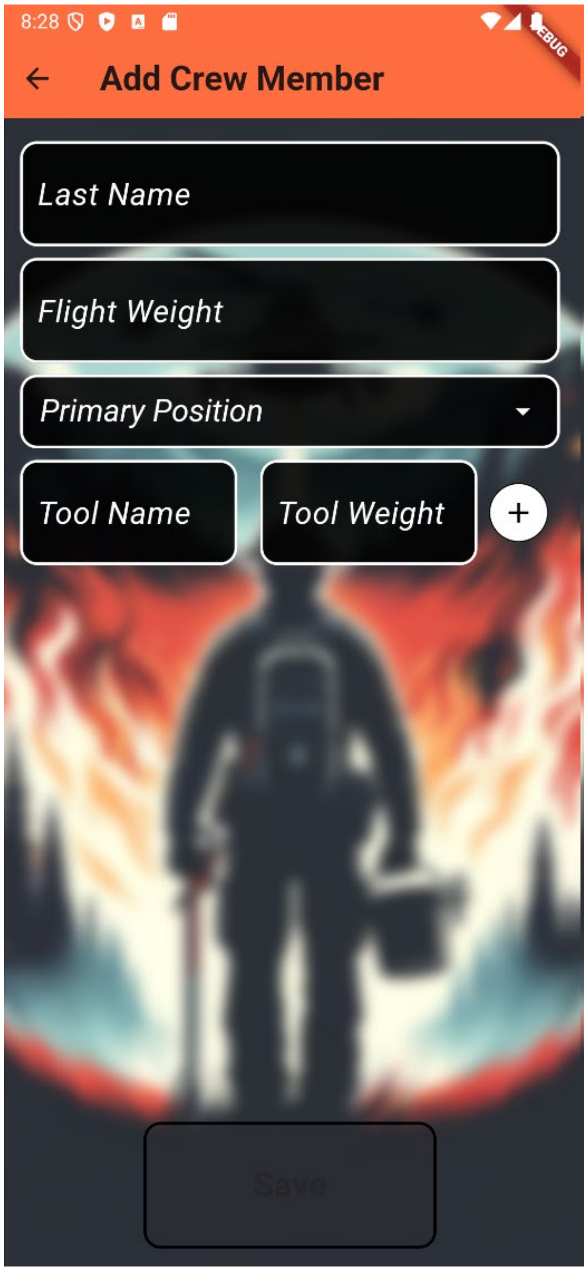
1. Add Crewmember
2. Edit Crewmember
3. Add Gear
4. Edit Gear
5. Trip Preference Settings

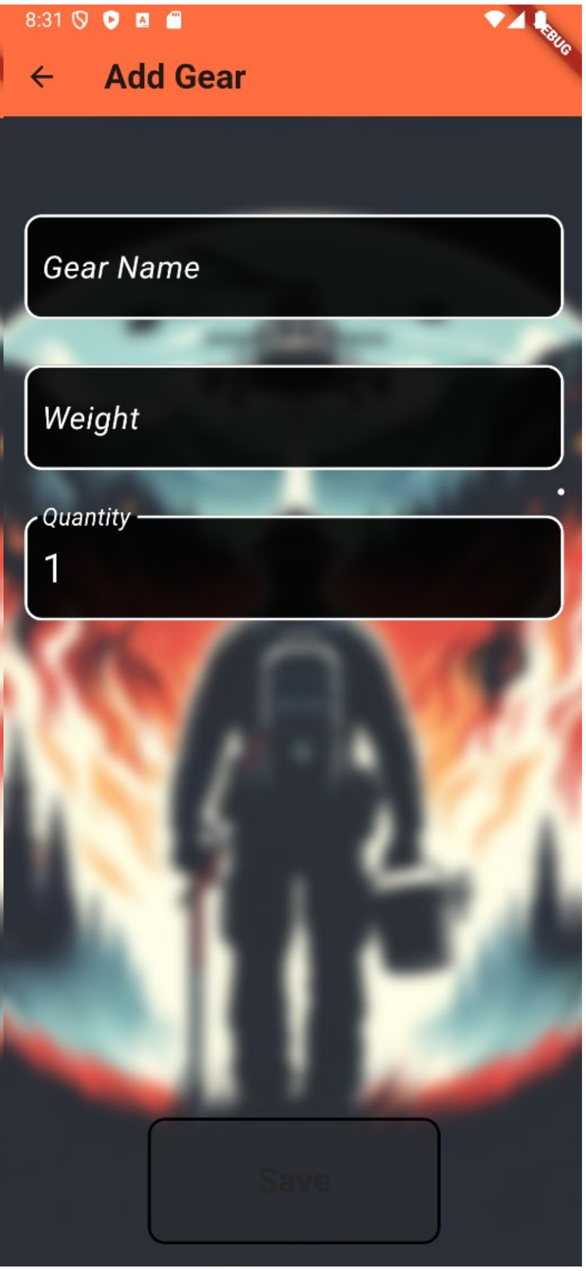
The Add and Edit Crewmember and Gear screens allow you to input and save new crewmembers or gear or edit existing entries.

The Trip Preference Settings option lets you customize and adjust trip preferences to tailor the manifest sorting algorithm to your needs.

Add Crewmember and Gear Screens:

Enter crewmember and gear information, then select Save to store the new crewmember or gear within the app.

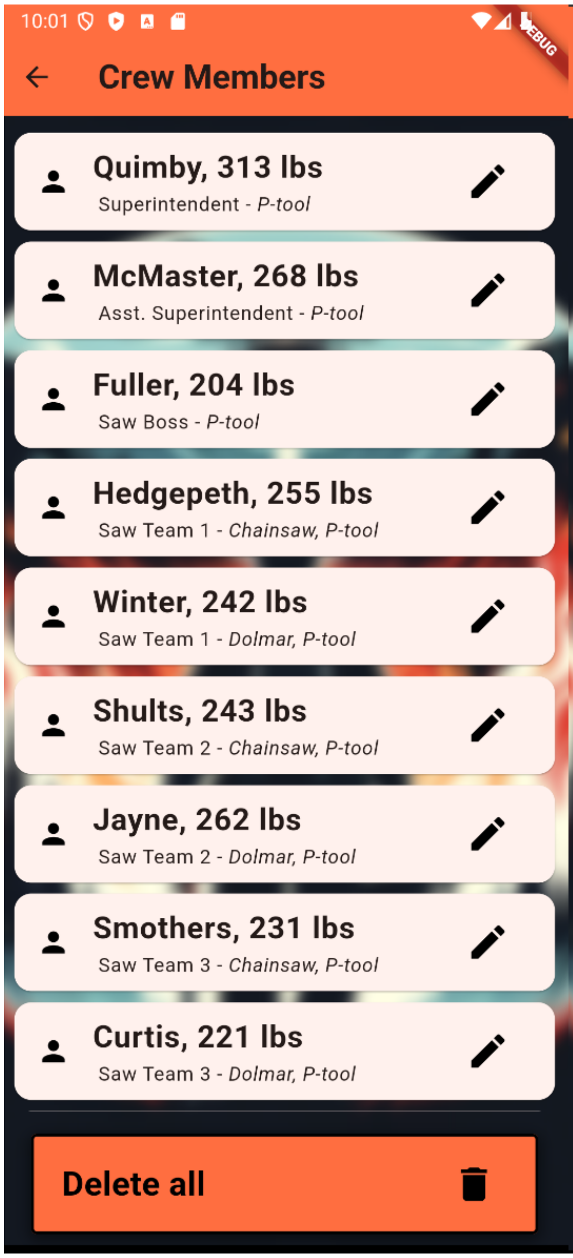




Quantity
1

Edit Crewmember and Gear Screens:

Select the Edit icon for the crewmember or gear item you wish to modify. Update the necessary attributes, then select Save to apply the changes.



10:01

Crew Members

Quimby, 313 lbs
Superintendent - *P-tool*

McMaster, 268 lbs
Asst. Superintendent - *P-tool*

Fuller, 204 lbs
Saw Boss - *P-tool*

Hedgepeth, 255 lbs
Saw Team 1 - *Chainsaw, P-tool*

Winter, 242 lbs
Saw Team 1 - *Dolmar, P-tool*

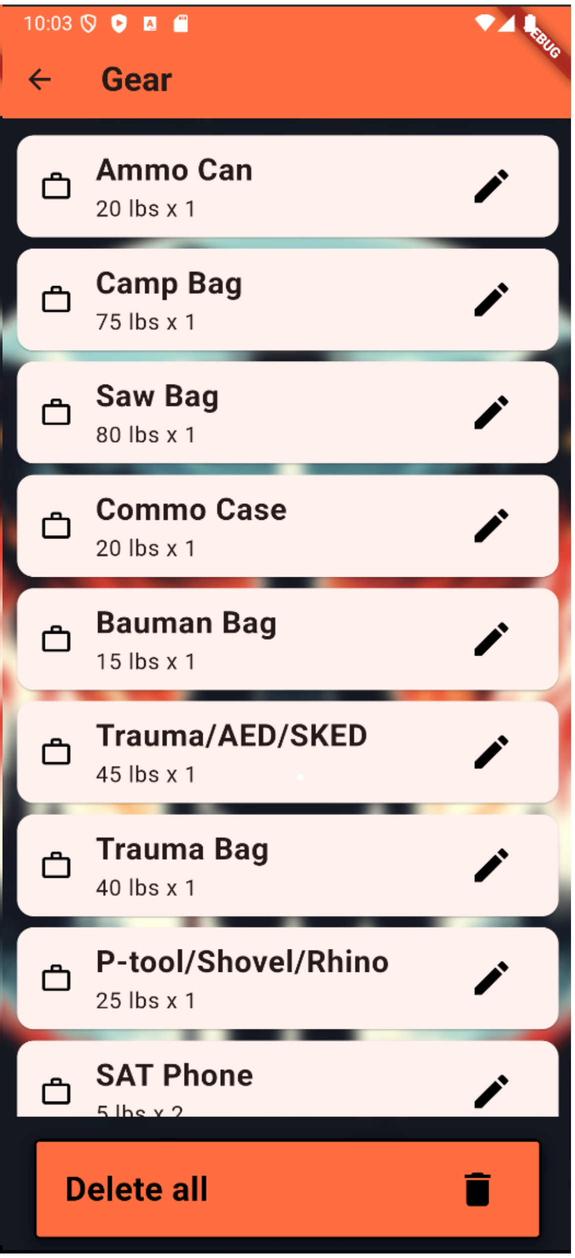
Shults, 243 lbs
Saw Team 2 - *Chainsaw, P-tool*

Jayne, 262 lbs
Saw Team 2 - *Dolmar, P-tool*

Smothers, 231 lbs
Saw Team 3 - *Chainsaw, P-tool*

Curtis, 221 lbs
Saw Team 3 - *Dolmar, P-tool*

Delete all



10:03

Gear

Ammo Can
20 lbs x 1

Camp Bag
75 lbs x 1

Saw Bag
80 lbs x 1

Commo Case
20 lbs x 1

Bauman Bag
15 lbs x 1

Trauma/AED/SKED
45 lbs x 1

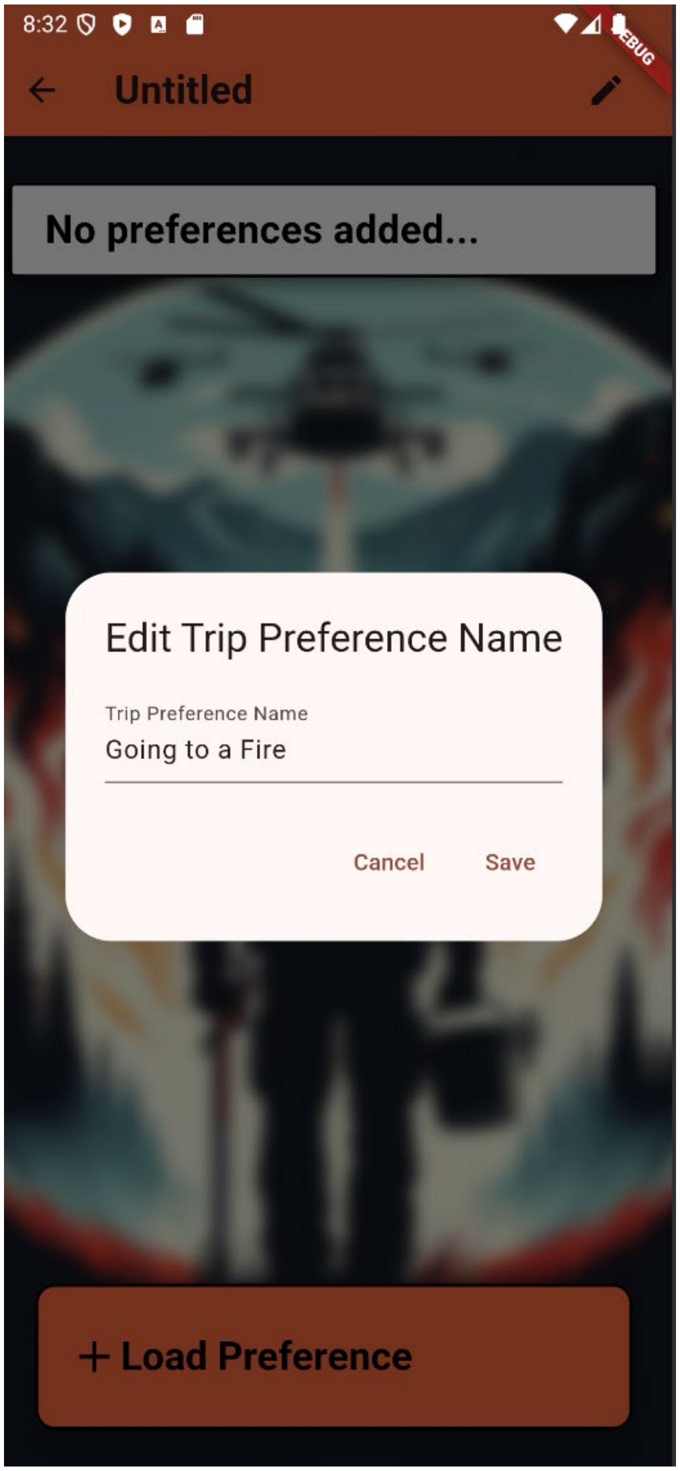
Trauma Bag
40 lbs x 1

P-tool/Shovel/Rhino
25 lbs x 1

SAT Phone
5 lbs x 2

Delete all

Create Trip Preference Loadout Screen:

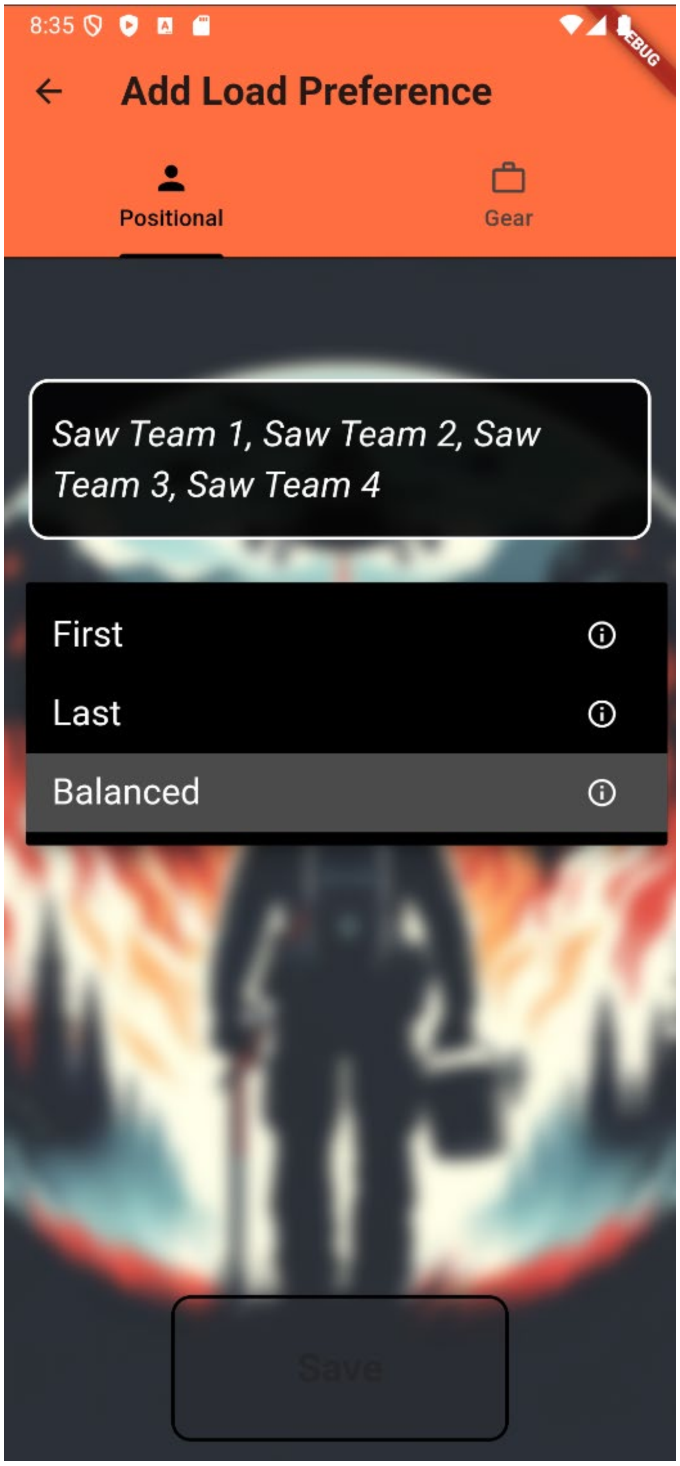


Preferences allow you to **customize** how the app creates manifests based on your specific needs and requirements.

To do this, you create a **Preference Loadout**, which includes all your desired preferences. You can then select this loadout when prompted during the manifest creation process.

The first step in creating a loadout is to assign it a name. Once named, click **Add Load Preference** to define specific preferences.

Add Load Preference Screen:



Saw Team 1, Saw Team 2, Saw Team 3, Saw Team 4

On the Add Load Preference screen, you can choose between Positional Preferences and Gear Preferences.

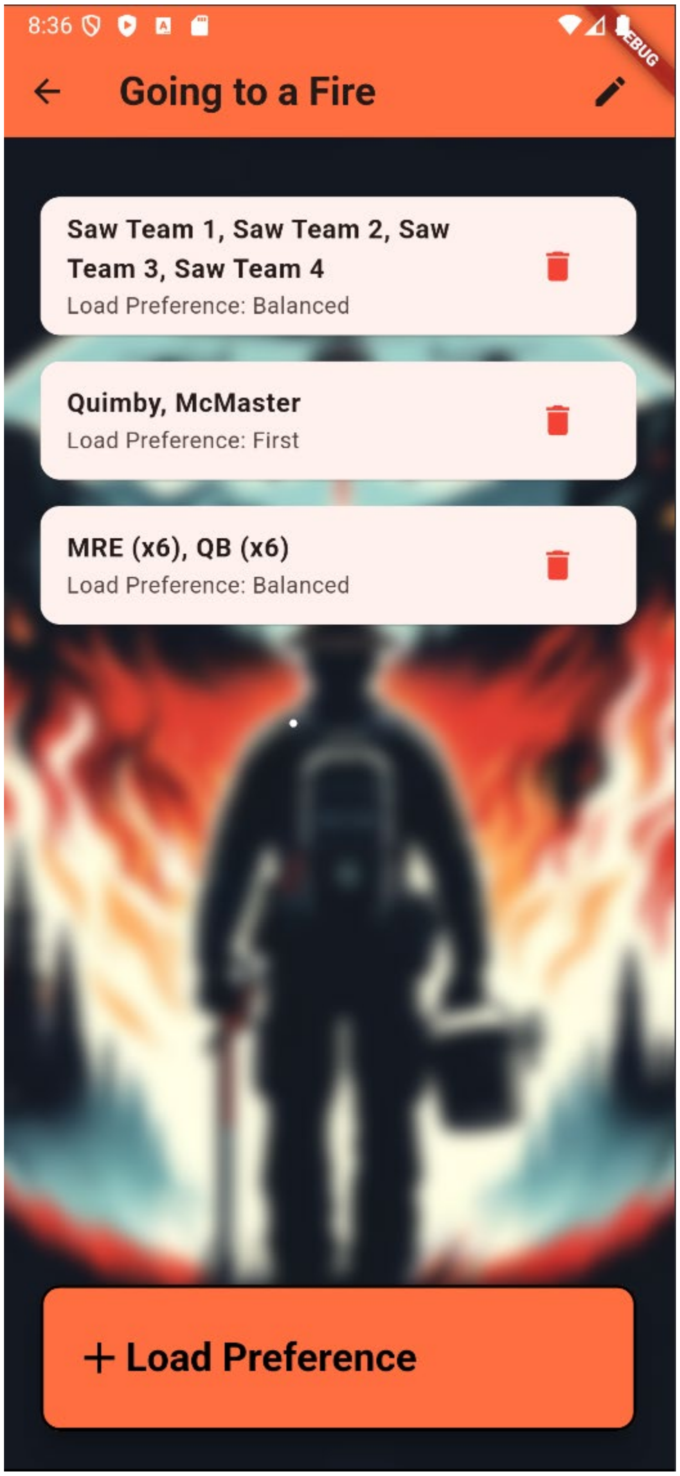
For each type of preference, you can select any combination of crewmembers or gear. To determine how the selected items are allocated, you can choose from the following distribution options:

First: All selected items are placed on the first load.

Last: All selected items are placed on the last load.

Balanced: All selected items are evenly distributed across all loads.

Preference Loadout Screen:


8:36

Going to a Fire

Saw Team 1, Saw Team 2, Saw Team 3, Saw Team 4
Load Preference: Balanced

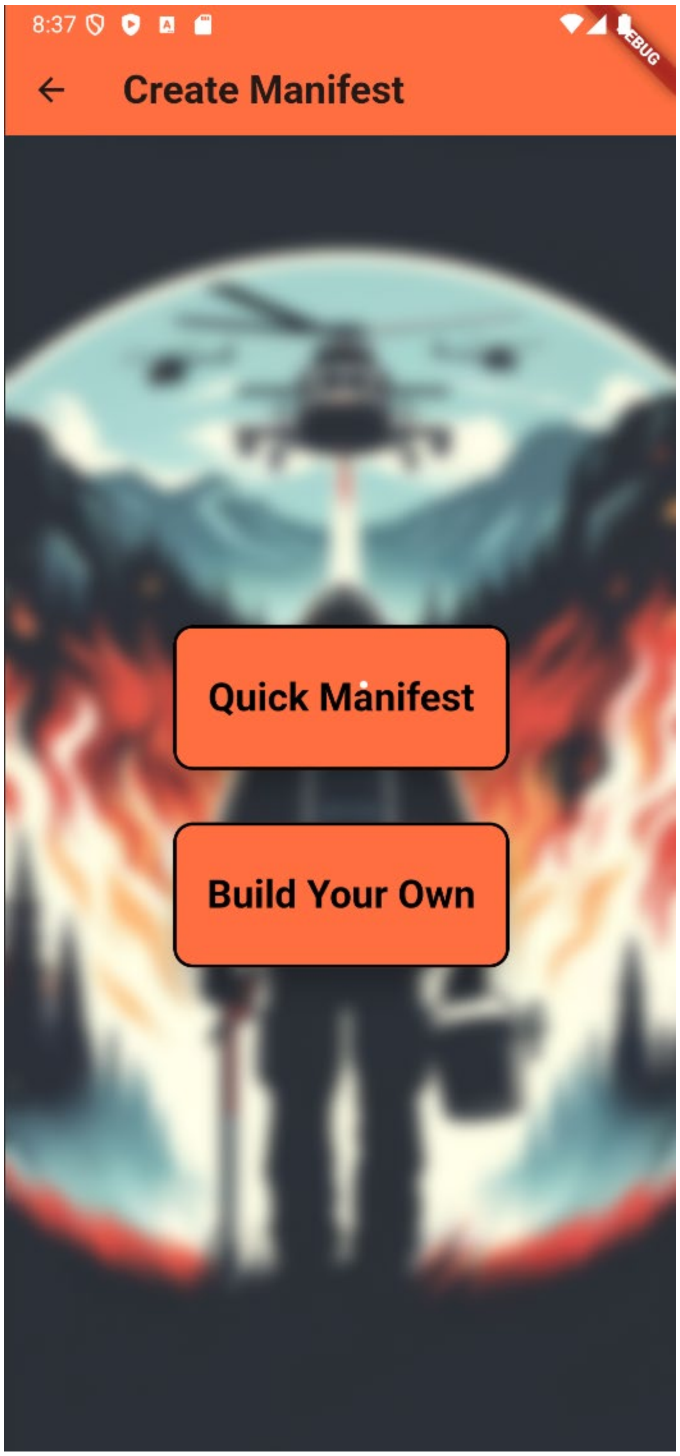
Quimby, McMaster
Load Preference: First

MRE (x6), QB (x6)
Load Preference: Balanced

After adding the desired positional and gear preferences, you can exit the Load Preference screen.

If you need to make additions or edits in the future, simply return to the Loadout screen to update your preferences.

Create Manifest Screen:

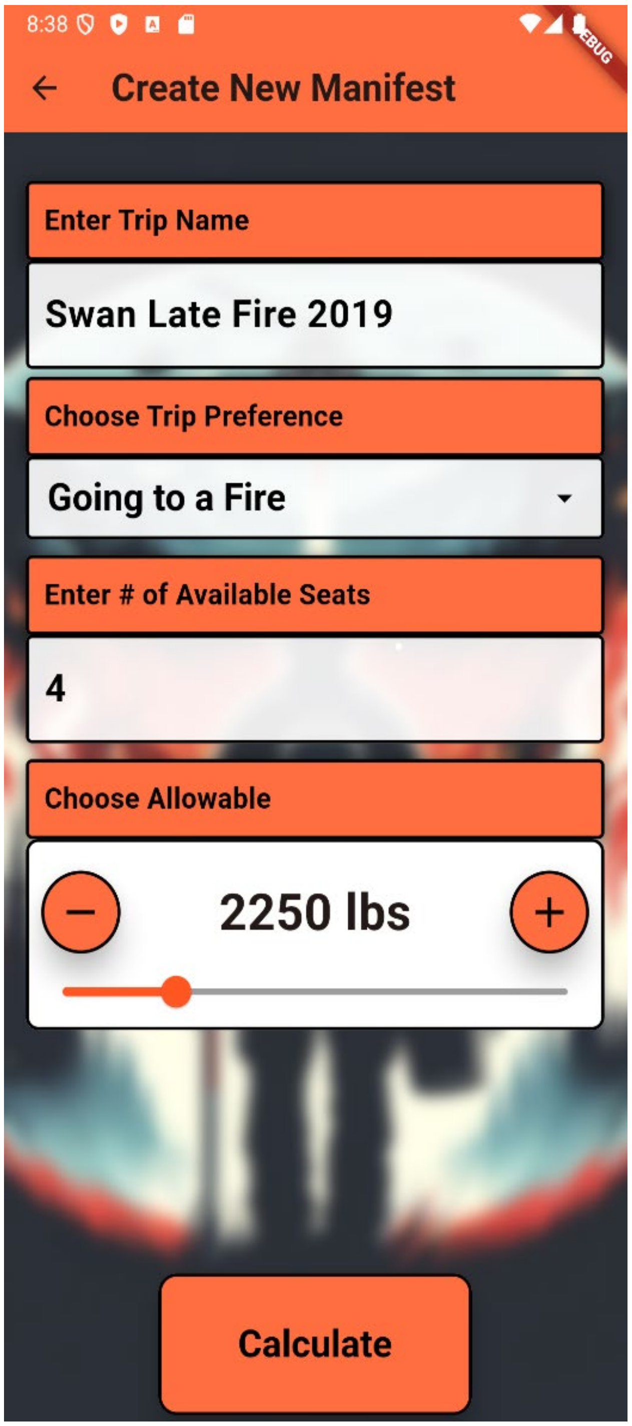


The Create Manifest screen offers two options:

1. Quick Manifest: This option uses the app's sorting algorithm and the user's preferences to automate the manifest creation process.

2. Build Your Own: This option provides a drag-and-drop interface, allowing the user to manually create a manifest by placing items into different loads. The app ensures that all created loads comply with the helicopter's constraints.

Quick Manifest Screen:


8:38

Create New Manifest

Enter Trip Name

Swan Late Fire 2019

Choose Trip Preference

Going to a Fire

Enter # of Available Seats

4

Choose Allowable

2250 lbs

To use the Quick Manifest Option you need to provide the following four required details:

1. **Trip Name:** which needs to be unique.

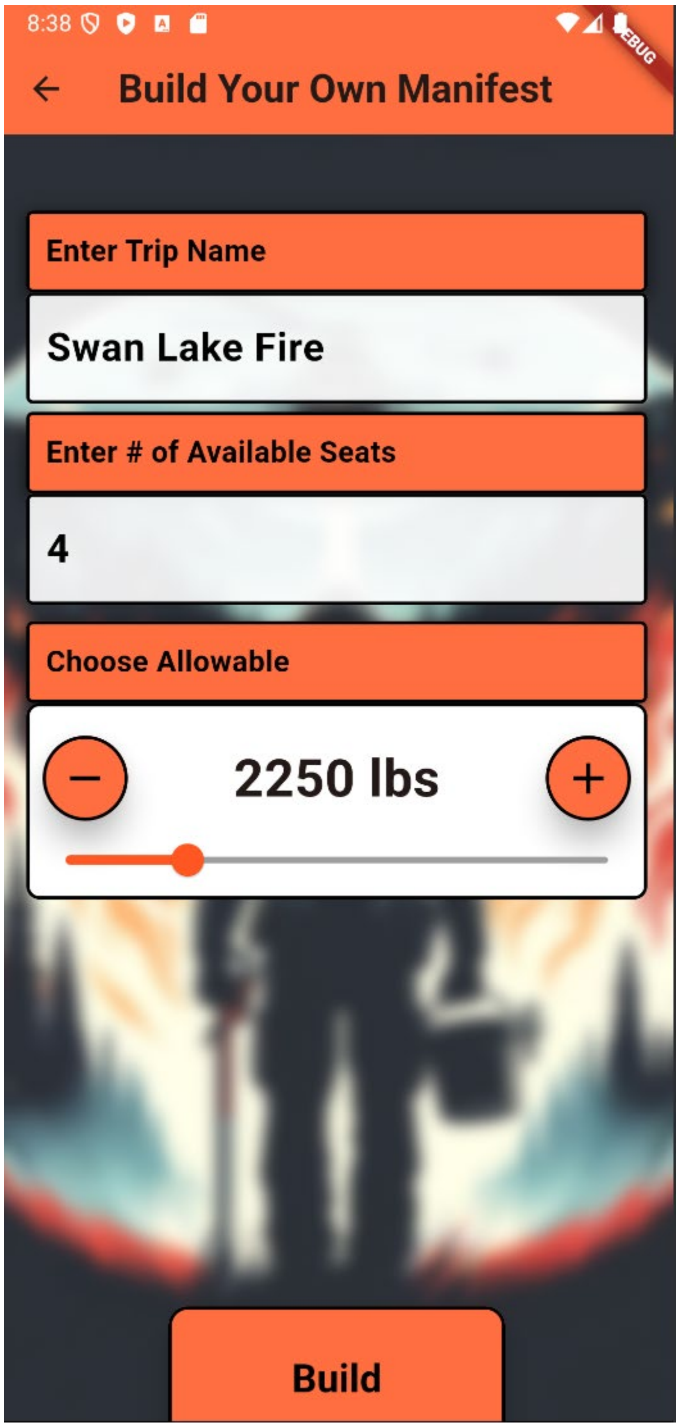
2. **Trip preference loadout:** which was defined earlier in the preference settings window.

3. **Number of available seats:** Specify the total number of seats available.

4. **Allowable weight:** Enter the weight limit for the aircraft

Once these details are entered click 'Calculate' to prompt the algorithm to generate a trip based on the selected trip preference loadout.
The trip can be viewed in the saved trips screen.

Build Your Own Manifest (settings) Screen:



Enter Trip Name

Swan Lake Fire

Enter # of Available Seats

4

Choose Allowable

2250 lbs

To use the Build Your Own Manifest screen, start by entering the following details:

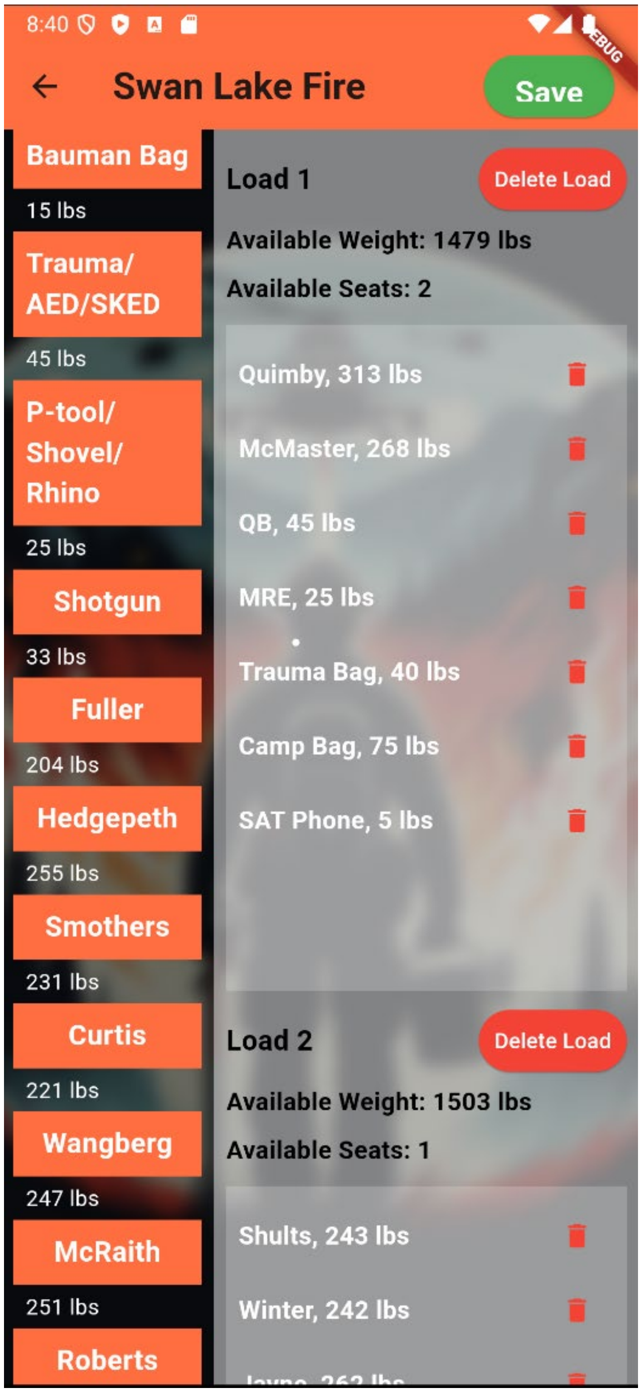
1. **Trip Name:** Must be unique.

2. Number of Available Seats: Specify the total number of seats available.

3. Allowable Weight: Enter the helicopter's weight limit.

After entering these details, select Build to proceed to the drag-and-drop screen.

Build Your Own Manifest (drag-drop) Screen:



8:40

Swan Lake Fire

Save

Bauman Bag
15 lbs

Trauma/ AED/SKED
45 lbs

P-tool/ Shovel/ Rhino
25 lbs

Shotgun
33 lbs

Fuller
204 lbs

Hedgepeth
255 lbs

Smothers
231 lbs

Curtis
221 lbs

Wangberg
247 lbs

McRaith
251 lbs

Roberts

Load 1

Delete Load

Available Weight: 1479 lbs

Available Seats: 2

Quimby, 313 lbs

McMaster, 268 lbs

QB, 45 lbs

MRE, 25 lbs

Trauma Bag, 40 lbs

Camp Bag, 75 lbs

SAT Phone, 5 lbs

Load 2

Delete Load

Available Weight: 1503 lbs

Available Seats: 1

Shults, 243 lbs

Winter, 242 lbs

Drag items from your saved gear and crewmember list on the left side into the load sections on the right side of the screen.

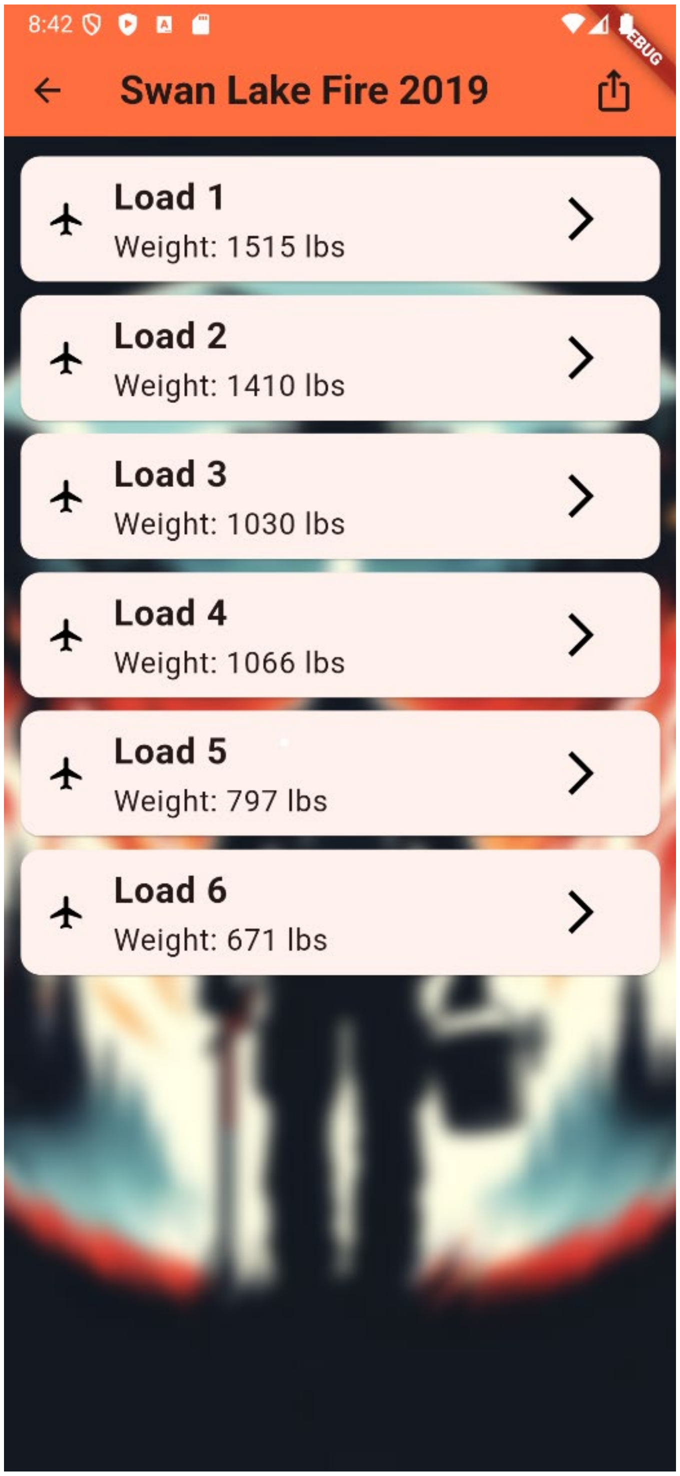
Add new loads using the **Add Load** button or remove existing ones with the **Delete Load** button.

The **Available Weight** and **Available Seats** indicators display the remaining capacity for each load.

If a load exceeds the weight limit, these indicators will turn bright red to signal that the load is invalid.

Once the manifest is complete, click the **Save** button to store this trip in your **Saved Trips**.

Saved Trips Screen:


8:42

Swan Lake Fire 2019

Load 1
Weight: 1515 lbs

Load 2
Weight: 1410 lbs

Load 3
Weight: 1030 lbs

Load 4
Weight: 1066 lbs

Load 5
Weight: 797 lbs

Load 6
Weight: 671 lbs

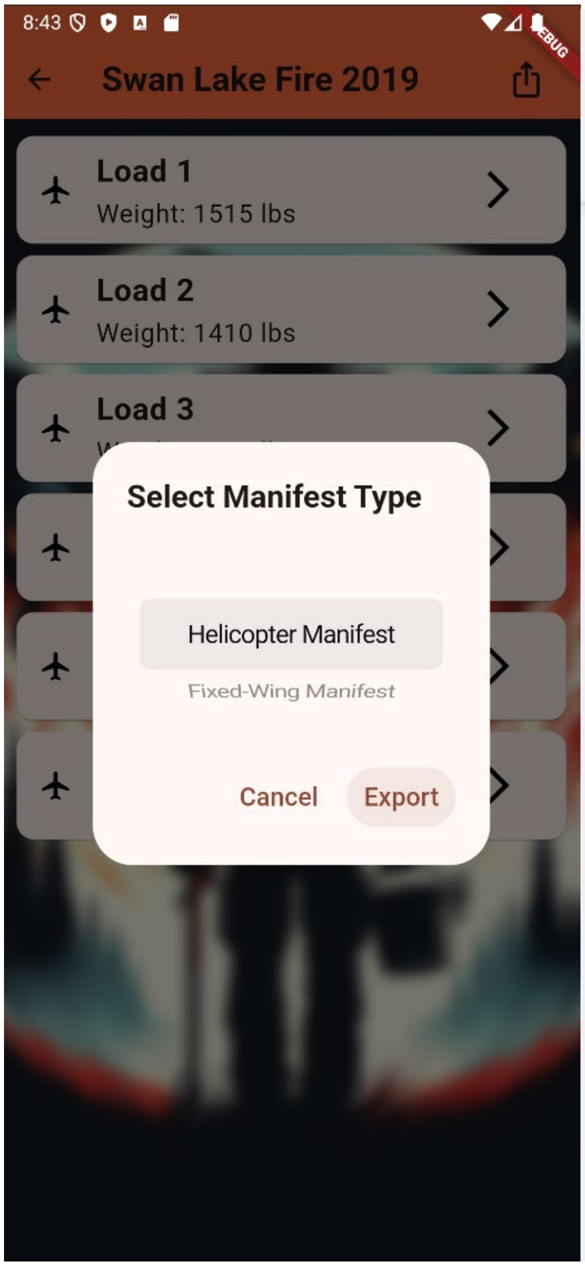
View Load Contents: To see the contents of a specific load, select the load you want to view. A detailed screen will appear, displaying each crew member and gear item assigned to that load.

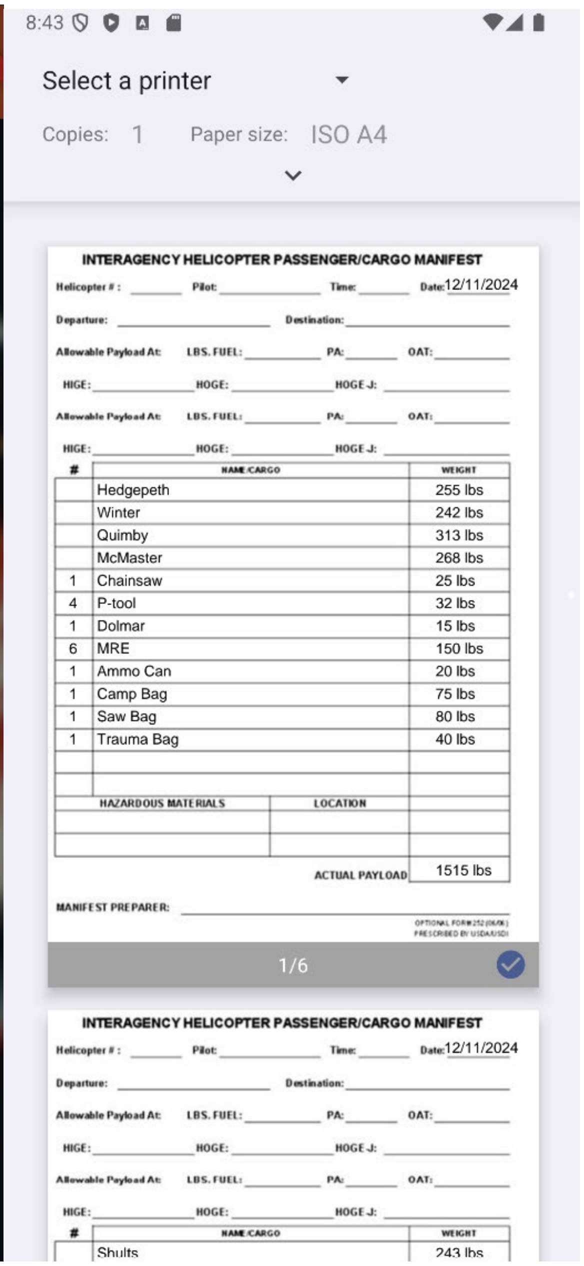
Export to PDF: To export the trip as a PDF document, click the **Export** button located in the top-right corner of the screen.

Export Trip Screen:

After clicking the **Export** button, you will be prompted to select either a **Fixed-Wing** or a **Helicopter** manifest.

Choose your preferred option, then click **Export** to generate the manifest as a PDF, which can be shared outside of the app.




8:43

Select a printer

Copies: 1 Paper size: ISO A4

INTERAGENCY HELICOPTER PASSENGER/CARGO MANIFEST

Helicopter # : ____ Pilot: ____ Time: ____ Date: 12/11/2024

Departure: ____ Destination: ____

Allowable Payload At: LBS. FUEL: ____ PA: ____ OAT: ____

HIGE: ____ HOGE: ____ HOGE J: ____

Allowable Payload At: LBS. FUEL: ____ PA: ____ OAT: ____

HIGE: ____ HOGE: ____ HOGE J: ____

#	NAME/CARGO	WEIGHT
	Hedgepeth	255 lbs
	Winter	242 lbs
	Quimby	313 lbs
	McMaster	268 lbs
1	Chainsaw	25 lbs
4	P-tool	32 lbs
1	Dolmar	15 lbs
6	MRE	150 lbs
1	Ammo Can	20 lbs
1	Camp Bag	75 lbs
1	Saw Bag	80 lbs
1	Trauma Bag	40 lbs

HAZARDOUS MATERIALS	LOCATION	
	ACTUAL PAYLOAD	1515 lbs

MANIFEST PREPARER: ____

OPTIONAL FORM 252 (06/06)
PRESCRIBED BY USDA/USDI

1/6

INTERAGENCY HELICOPTER PASSENGER/CARGO MANIFEST

Helicopter # : ____ Pilot: ____ Time: ____ Date: 12/11/2024

Departure: ____ Destination: ____

Allowable Payload At: LBS. FUEL: ____ PA: ____ OAT: ____

HIGE: ____ HOGE: ____ HOGE J: ____

Allowable Payload At: LBS. FUEL: ____ PA: ____ OAT: ____

HIGE: ____ HOGE: ____ HOGE J: ____

#	NAME/CARGO	WEIGHT
	Shults	243 lbs

12. Source Code

The source files for the Fire Manifest App can be found on GitHub at the following link: https://github.com/dawsonnash/fire_manifest_app. All User Interface and Data management files can be found under the *lib* folder. The rest, besides the *asset* folder and *pubspec.yaml* file, were altered either very little or not at all, as they come generated with a Flutter Project and are responsible for tasks like converting Dart to the native languages of iOS and Android.

12.1 Data Files

The *lib* -> *Data* folder contains all files relevant for managing the app data. Each data file is accompanied by its *.g* file, which is auto-generated as a result of classifying the data class files as Hive objects. Inside the Crew class structure are all the functions relevant for manipulating crew member and gear data as well as the calculations necessary for updating attributes of these objects. Inside the Trip class file contains all the functions relevant to manipulating Trip and Load objects. Inside the SavedPreferences file contains the entire Trip Preference class structure for creating Positional and Gear Preferences. Lastly, in this folder is the load calculator file.

12.2 User Interface Files

Placed directly in the *lib* folder is the entirety of the app's user interface files. These files align greatly with the UI layout in Figure 4. The naming convention of all files follows directly what their designed purpose is.

13. Bug Tracking

UI Text Overflow Issue

Description:

In user input text fields across various app pages—including the Add and Edit Crewmember and Gear screens, Preference Settings, and Trip Manifesting screens—text entered by the user can overflow the available text window if it is too long. Flutter responds to this issue by displaying grey and yellow error bars that obstruct the view of the overflowed section.

Severity:

Medium

App Crashing When Manifesting Without Preference Loadout

Description:

If the user attempts to use the algorithm to create a manifest in the Quick Manifest screen without selecting a preference loadout, the app crashes. This issue manifests as the app freezing and becoming unresponsive, requiring a restart to regain functionality.

Severity:

High