Time and Mileage Tracker

sp06-red's Final Report

CS 4850/02 Spring 2024, Professor Perry

Justin Barker, Reese Gassner, and Jason Zhou

2024-04-27

1155 lines of code

Components: Flutter, Dart, Geolocator, Git, GitHub, path_provider, dart io, MS Teams

Project GitHub Repository

Project Website

Table of Contents

Introduction

1.1 Purpose

A mobile application that tracks the distance and duration of a vehicular trip. Users can use it to help with monitoring their vehicle use, business reimbursements, and even taxes. A user can either add a trip manually or start and keep the track of the specific trip in the app.

1.2 Project Goals

The app should facilitate efficient and accurate tracking of mileage and time for users' travel activities, providing a user-friendly interface and seamless integration with GPS to streamline data capture and minimize manual input.

Design Constraints and Considerations

2.1 User Characteristics

The app is designed with simplicity and intuitiveness in mind, catering to both novice users who may be less familiar with technology and experienced users who may be interested in more advanced features. Users may value efficiency and time-saving features, particularly when tracking mileage and time for business or personal purposes. The app offers streamlined data entry methods, automated tracking options (e.g., GPS integration), and customizable reporting features to help users save time and effort.

2.2 System

The tracking system is a sandboxed app on the user’s phone. The app relies on GPS tracking for accurate mileage calculations, imposing constraints on location accuracy, battery consumption, and device compatibility. Because the app is written using the Flutter framework, it should be compatible with both iOS and Android while maintaining consistent functionality and user experience. The system efficiently stores and manages large volumes of mileage and time data for individual users and organizations.

2.3 Assumptions and Dependencies

The application's functionality is influenced by the availability and reliability of network connections. Given the mobility of the application, a stable internet connection cannot always be ensured. Moreover, the application's dependence on GPS data for mileage tracking is affected by various factors that impact location accuracy, including signal strength, satellite visibility, and environmental obstacles. It's important to consider these factors when assessing the overall performance of the application.

2.4 General Constraints

The application's functionality is influenced by the availability and reliability of network connections. Given the mobility of the application, a stable internet connection cannot always be ensured. Moreover, the application's dependence on GPS data for mileage tracking will be affected by various factors that impact location accuracy, including signal strength, satellite visibility, and environmental obstacles. It's important to consider these factors when assessing the overall performance of the application.

2.4 Goals and Guidelines

The mileage/time tracking app is designed with several key goals and guidelines in mind to ensure its effectiveness and usability. Its primary goal is to provide users with a seamless and efficient way to track their mileage and time while maximizing productivity. To achieve this, the app focuses on accuracy, implementing robust algorithms and validation mechanisms to ensure precise tracking of data. It also prioritizes flexibility, offering customizable features and settings to cater to diverse user preferences.

Requirements

3.1 Trip Logging

3.1.1 Description

Enables users to accurately record and track details of their trips within the app. Whether for business or personal use, this feature simplifies the process of capturing essential information about each journey, including start and end locations, distance traveled, duration, purpose, and additional notes.

3.1.2 Functional Requirements

3.1.2.1 User should be able to create new trip entries by providing the total trip mileage and time.

3.1.2.2 User should be given the option to start a GPS tracked trip

3.2 Trip Organization

3.2.1 Description

Efficiently organize user trip entries within the app by assigning tags. This functionality enhances usability, searchability, and customization, and enables users to manage their data in a more structured and personalized manner

3.2.2 Functional Requirements

3.2.2.1 User should be able to assign one or more tags to a logged trip

3.2.2.2 User should be given the option to add an annotation to a logged trip

3.2.2.3 Logged trips should be optionally organized by the user by tag and other entry-related properties such as time, distance, and duration.

3.3 User Data Storage

3.3.1 Description

Store data inside user local storage ensuring the user can access without a internet connection. Ensure that data stored in the system retains its accuracy and consistency every time the user makes a change and if the application is open/closed. Complies with the standard data privacy laws and regulations.

3.3.2 Functional Requirements

3.3.2.1 Trip data created is stored into CSV file

3.3.2.2 Data remains persistent after user closes application

3.3.2.3 Read from file and rebuild entry list once application is loaded back up

3.4 User Interface

3.4.1 Description

This feature encompasses the visual and interactive elements that users interact with to input, view, and manage their mileage and time data.

3.4.2 Functional Requirements

3.4.2.1 The app should include a navigation menu or toolbar for easy access to different sections and functionalities, such as mileage entry, and time tracking.

3.4.2.2 Input forms should be included to allow users to enter mileage and time data, including details such as trip start/end times, distances, and tags.

3.4.2.3 Should include a scene for summary views of users' entries in a concise and organized format, providing an overview of past activities and allowing users to review, edit, and delete entries.

Analysis

4.1 Flutter

Flutter is a critical architectural component in developing the time mileage tracker. Flutter provides a comprehensive set of tools, libraries, and features that enables build responsive and performant user interfaces for both iOS and Android platforms.

4.1.1 Application

Flutter's UI programming model enables clear explanation of the UI hierarchy using an extensive set of customizable widgets. This helps facilitate the creation of an appealing and responsive user interface.

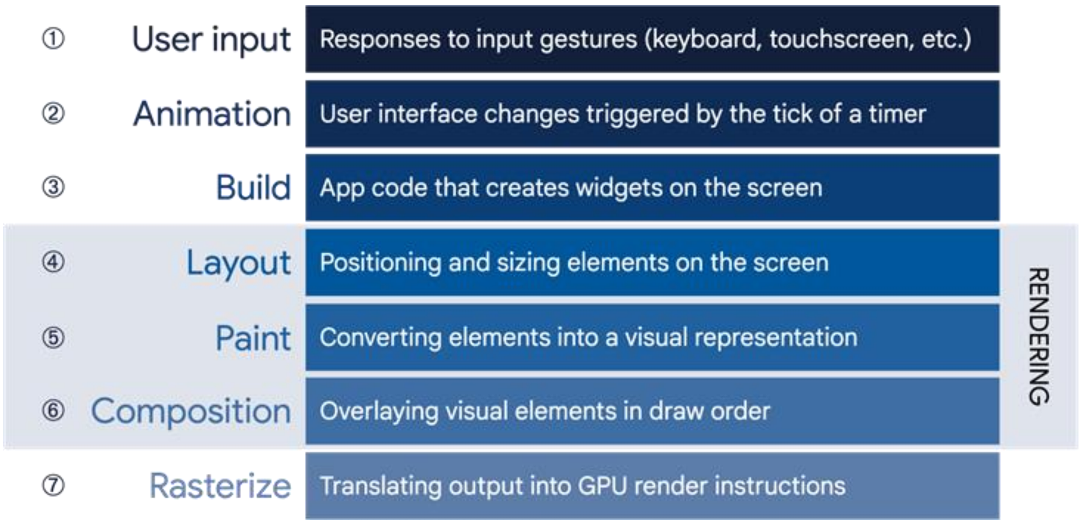
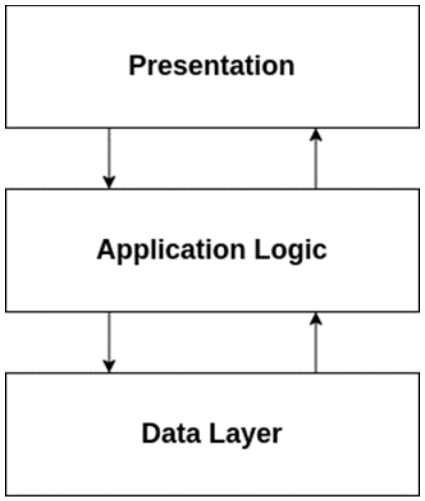

① User input
Responses to input gestures (keyboard, touchscreen, etc.)
② Animation
User interface changes triggered by the tick of a timer
③ Build
App code that creates widgets on the screen
④ Layout
Positioning and sizing elements on the screen
⑤ Paint
Converting elements into a visual representation
⑥ Composition
Overlaying visual elements in draw order
⑦ Rasterize
Translating output into GPU render instructions
RENDERING

Figure 1: Flutter Architectural Overview [1]

Flutter's architecture is designed for efficient UI handling, such as rendering, animation, and user input, offering a smooth experience for cross-platform apps. User input is handled by widgets triggering callbacks. The layout phase calculates each widget's size based on parent constraints, followed by the painting phase, drawing widgets based on layout calculations. The composition and rasterization stages optimize rendering, generating a layer tree, which converts into pixels on the screen, ensuring high-quality and fast-rendering apps across platforms.

Flutter also provides extensive support for integrating functionality through packages. Packages such as location and geolocator will be utilized to integrate GPS tracking and map functionality into the app ensuring consistent behavior across platforms.

Architecture Design



5.1 High-Level Overview

The time milage tracking app is designed with the idea of clear partitioning of responsibilities across several components and their respective subsystems. These components and subsystems are organized into layers each responsible for distinct aspects of the application's functionality. These layers are the presentation layer, application logic layer, and data layer. This separation ensures that each component can be developed, tested, and maintained independently leading to a more modular and maintainable architecture.

5.2 System Breakdown and Responsibilities

5.2.1 Presentation Layer

The presentation layer is broken down into UI Views and User Input subsystems. UI Views has the responsibilities of the user interface elements required for the application such as the map, trip logs, and other information required for the user to interact. The User Input subsystem is responsible for capturing and processing user actions like starting and stopping trips so that it can convert interactions into commands and events that can be understood and processed by the other layers.

5.2.2 Application Layer

The application logic layer is the driver for the application. It is responsible for orchestrating the core logic and operations of the application. The layer is broken into several subsystems, each handling specific aspects of trip tracking, reporting, and analysis. The trip tracking subsystem manages not only the tracking of user trips using GPS data, but also data that is inputted by the user that GPS is not required for like trips that were done before that a user wants to log by inputting distance and time. Trip tracking also processes trip data, calculates mileage based on GPS location. The reporting and analysis subsystem analyzes user travel patterns and trip data to generate and provide users with valuable insights to their travel behaviors.

5.2.3 Data Logic Layer

The data layer within the architecture of the time mileage tracking app serves as the foundation for managing trip data storage and organization. The layer is composed of local data storage and tags/category organization. The local data subsystem handles storage of trip records locally on the device. This was chosen over other methods of storage so that the application can provide fast and guaranteed access to data without an internet connection. Since this application is for trips over a distance, making the accusation of data reliable was important, so it was decided to use local storage. Furthermore, the tag and category subsystem allow users to organize past trips by assigning tags and categories, enhancing organization and filtering capabilities.

Detailed System Design

6.1: UI Views

6.1.1 Classification

The UI Dialog Map is a sub system.

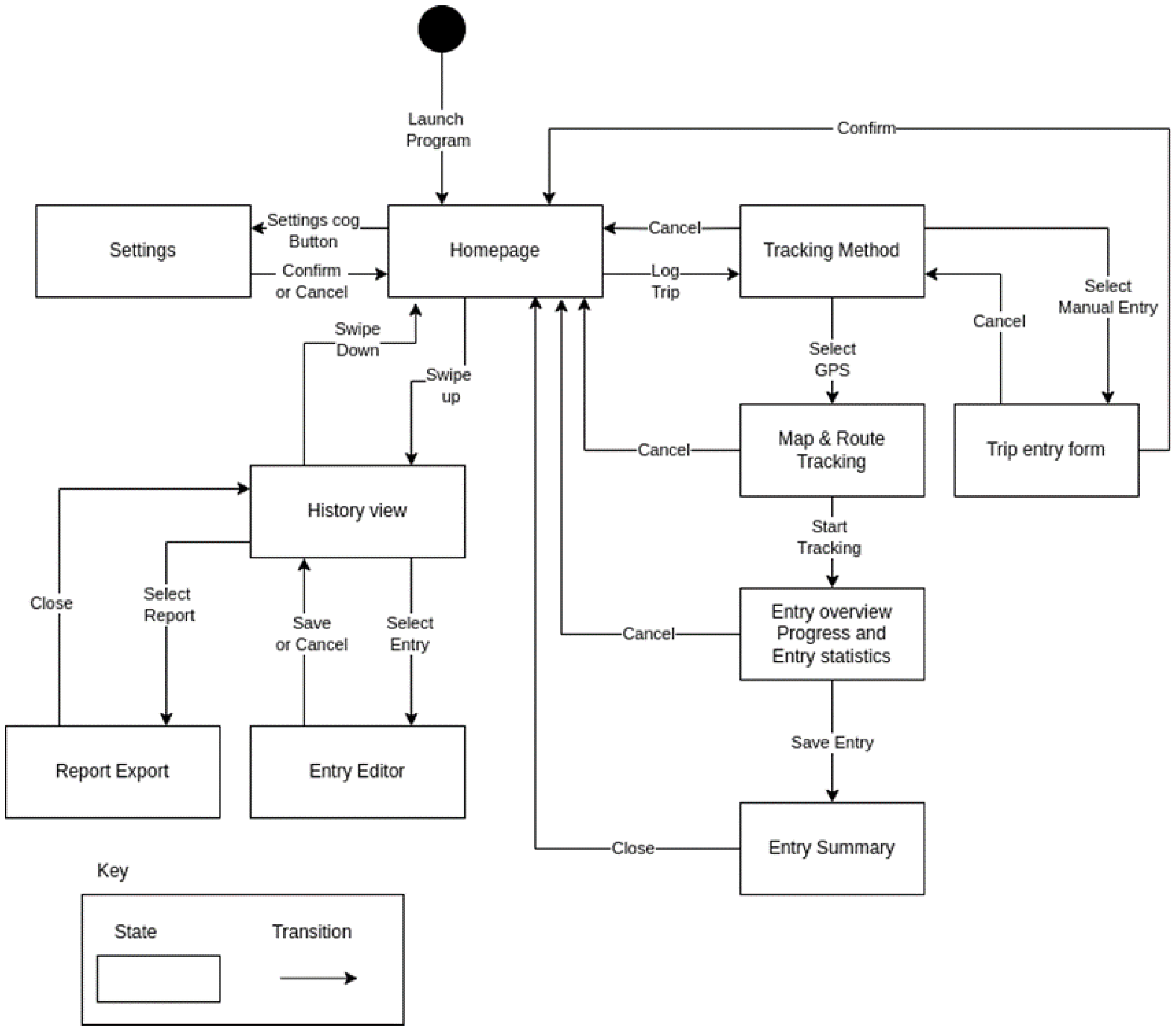
6.1.2 Definition

The reason for the sub system is to show both the developer and the client what happens at each transition and state of the tracking app.

6.1.3 Uses and Interactions

The application starts on the home page. Then the user can either go to settings, tracking method, or history view. The settings are used to adjust color and specific symbols for the navigation system. The history view allows users to either enter data or you can view the reports. The tracking method can be either selected or entered manually, which will lead a user to the trip entry form. In this state the user enters the trip mileage and time that was calculated from the tracking mileage state. There is also the select GPS after tracking method which starts tracking the route, which eventually leads to the summary of the trip. This includes both the time and the mileage of the whole trip.

6.1.4 User Interface Dialog Map



This diagram exists to visualize the overall user interface architecture of the application abstracted from the visual appearance. It will allow development to better understand what processes a user would follow to accomplish their goal(s). Some of the transitions allow the user to back out of sequences to increase usability in the situation a user changes their mind.

6.2: Trip Tracking Logic

6.2.1 Classification

The trip tracking logic will be a package.

6.2.2 Definition

The trip track logic allows the user to log a trip by entering their start locations, end locations, starting date, ending date, and distance. The user does not need to input their duration as it is automatically calculated from the start date and end dates the user gives which includes the user having to specify the starting and ending times. The user can also log a trip using GPS. If the user chooses to log a trip using GPS, the application will ask permission to use the user's location data. When the user allows permission, their location is tracked from a stream of longitude and latitude coordinates that are being compared to a previous longitude and latitude coordinate to calculate the distance you are traveling.

6.3: Data Storage

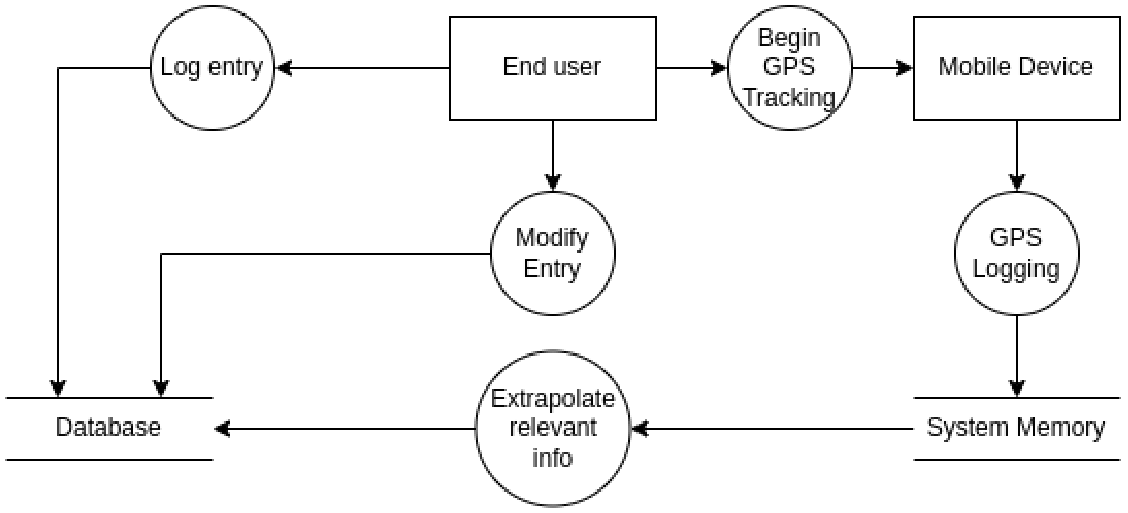
6.3.1 Classification

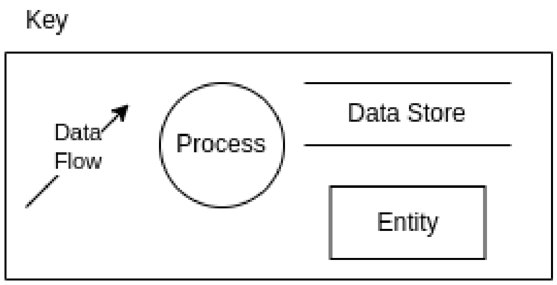
Data storage is best classified as a package working as a proxy to a group of files.

6.3.2 Definition

Responsible for storing and managing the mileage and time tracking data generated by users. This component encompasses the database or file system used to persistently store the data, as well as the software modules or classes responsible for interacting with the storage medium. The data storage component ensures the integrity, accessibility, and security of the stored data, allowing users to reliably track, view, and manage their mileage and time entries within the app.

6.4.3 Data Flow Diagram





This diagram visualizes the overall flow of data through the program and shows which events generate data. Because storing a stream of GPS information is relatively intensive on storage, the system's memory will be used and processed into a more space conscious format.

Development

7.1 Frontend

7.1.2 Flutter UI

We wanted to keep the UI of our application minimal and straightforward. This was accomplished by not using many complicated routes, rather opting to use dialog windows overlaid on top of the main list view. All functionality of the app can be accessed from this main view. Our main screen is a Stateful widget which primarily tracks the activity of the GPS module and a ListManager object.

7.1.3 Entry list

On app initialization we wait until the ListManager has loaded the list from the device's disk via the StorageManger. The body of the scaffold object is a ListView builder, which is given a count of how many items are contained in the list so it can return accurate index values which ensures any editing corresponds to the correct entry tracked by the ListManger object.

The bottom navigation bar consists of multiple IconButtons encapsulated in Card widgets. These buttons allow for various functionalities, such as starting/ ending GPS tracked trips, adding manual entries, filtering the list, and flushing the list.

7.1.3.1 GPS Tracking Toggle

GPS Tracking can be toggled by the user with the leftmost button on the user-interface. This button's symbol depends on the isTracking state of the root window and is changed from a record icon to a stop icon if a trip is actively being tracked. The button calls the _toggleGPSTracking function and sets the appstate every time it is pressed by the user. This function also includes the logic to interface with the GPSTrip module to enable and disable tracking and to append the resulting trip(s) from the GPS module to the ListManager's entry list. The root state variable, isTracking, is inverted every time the button is pressed for future reference by this function.

7.1.3.2 Adding Entries

When the user presses the button containing a plus icon, a new dialog is overlaid above the list view which contains many buttons and fields prompting the user for various properties of the new Entry. The first two are date pickers to define the start and end time of the trip. Local DateTime objects, start and end, are updated on confirmation of the respective date picker. Following the date time pickers are two text fields which prompt for the mileage and a list of tags each separated by spaces. On confirmation of this dialog, prior to the generation of any new Entry objects to add to the ListManager, each field is verified by a sequence of checks to be legal and valid.

If a trip is currently being tracked by the GPS module, the user cannot manually enter a list. This is done intentionally as if a trip is being tracked, the user is expected to be operating a motor vehicle and should not be attempting to manually enter trips.

7.1.3.3 Editing Entries

Any entry in the list can be tapped on by the user, which causes the app to respond with a Dialog widget which is similar to the Add entry dialog, but each user input is pre-populated with the selected Entry's original values using TextEditingControllers for the Entry's taglist and mileage and dates via the DatePicker's start parameter.

Filtering Entries

Filtering the global list is accomplished by setting the filter ranges and constructing a new list from the global list. The list created using the filter values is then set as the active list which the end-user can see and manipulate. Any manipulation such as editing, deletion, or adding new entries to this sub-list is reflected in the global list of entries. Users can return to the global list view by opening the filter dialog once again, and pressing the reset button.

7.1.3.4 Adding Saved Locations

The user can select the 'location add' button to add a new named location to the auto-tag list. This function usually takes a moment as the device needs to get a GPS lock and position value prior to prompting the user to assign a name to the location. Once the location's name is submitted and auto-tagging is enabled in the settings then GPS tracked trips will be tagged with this location's name provided they either start or end within a 5m radius.

7.2 Backend

7.2.1 GPS Logic

We wanted to stay with the idea of modularity, so the GPS logic is implemented in its own file. GPS is being implemented in the application through a library called Geolocator. Geolocator is a library in Flutter that allows you to receive the longitude and latitude of a device from a position stream. In the GPS file, there is a class called GPSTrip. This class holds all the core logic for the GPS. The class starts with instance variables to retain essential information for GPS trips to be logged correctly. There are two DateTime variables that have the start time and end time of the trip, a position variable to hold the starting position of the trip, a double variable to hold the calculated distance, and a subscription to the stream of positions.

7.2.2 Checking Permissions

The first method in the class is called permissionCheck. The permissionCheck method is defined to check if location services are enabled and if location permissions are granted. Two variables are declared, serviceEnabled and permission. serviceEnabled is a boolean that will store whether location services are enabled on the device. permission is of type LocationPermission and will store the current location permission status. We then call the isLocationServiceEnabled method from the Geolocator library to check if location services are enabled for the device. If the location services are disabled, serviceEnabled is set to false, and the method throws an error. If true, the method can continue on to where the checkPermission method is called from the Geolocator library, which checks to see if the user has already given us permission before from a different session. If not, we use the requestPermission method from the Geolocator library to request user permission. If the permission status is still denied after the request, the method throws an error with the message 'Location permissions are denied'. If the permission status is denied forever, the method throws an error with the message 'Location permissions are permanently denied, we cannot request permissions.' If the user has chosen to deny location permissions forever for the app, the app cannot request these permissions again. If none of the errors were thrown from being denied, then whatever method calls the permissionCheck method will be able to continue with its work.

7.2.3 Starting Trips

The second method in the class is the startTrip method, which is how every GPS trip is started when the user presses the button to start a GPS trip. The method starts by logging the start date and time using

DateTime.now representing the trip's start. Then the permissionCheck method is called to check if location services are enabled and if the necessary location permissions are granted. If the permissionCheck method throws an error, the startTrip method will not proceed with the location tracking. Then the instance variable from earlier that is supposed to hold the starting position for the trip, is set by using the getCurrentPosition method from Geolocator that is also set to have a location accuracy of high to get the most accurate position. This same location is used in tandem with a saved locations list, if this location is within a 5-meter radius of any location in the saved locations list, the location’s name from the list will be added to the new entry’s taglist. After the starting position is found the trackLocation method is called.

7.2.4 Tracking Using GPS

The third method is _tracklocation which is responsible for tracking the location during a trip. First we call the getPositionStream method from Geolocator to get a stream of position updates. This method returns a Stream<Position>, so listen is used to subscribe to the stream and get the position updates. A callback function is passed to the listen method. This function is called every time a new position update is received. The function takes the current position of the user as a parameter. Now, inside that call back, a try catch block is implemented to handle any errors that could occur during the calculation of the distance. That distance variable from earlier is set by the calling of the distanceBetween method inside the Geolocator library, which takes in the starting position and the current positions of longitudes and latitudes. The distance is found from the two sets of given coordinates in meters and is converted to miles by multiplying it by .000621371 and is added to a new variable called totalDistance which keeps track of the total distance of the trip. Once the distance is found the starting position variable is overwritten by the current position and the method waits for the next current position from the position stream.

7.2.5 Ending the Trip

The last method is the endTrip method, which is responsible for ending the GPS tracking for a trip. The ending time variable from earlier, which was initialized at the beginning of the class, is set to the current date and time using DateTime.now. Then, the positionStreamSubscription is canceled using the cancel method from Geolocator. The subscription stream is then set to null to free up resources and stop getting user location data. A new Entry object is made with the start date and time, end date and time, and the totalDistance of the trip. Using the final location data, if it is within a 5-meter radius of any location in the saved locations list, the location’s name from the list will be added to the new entry’s taglist. Once the entry is made the total distance is then set back to zero in preparation for the next trip.

7.2.2 List Manager

The ListManager object is a controller that manages a list of Entry objects which operates as an interface between the StorageManager, filtering, and the user interface. It tracks two lists, the global list, and the active list. By default the global list is the active list and includes every entry created from the CSV file and by the user. In the case the user applies a filter, the listmanager generates a new list based on the filter parameters and sets the active list to this newly created list. It then notifies the UI listeners, which prompts the listbuilder to update. This class also includes the basic create, read, update, and delete operations required for managing each data element tracked by the list. On any CRUD change to the global list, the list is saved to the device's disk to ensure data persistence in the event the app is unexpectedly closed.

7.2.3 Storage Manager

The StorageManager object is a controller that interfaces with the ListManager and the device's local storage (functionality provided by Flutter's path_provider package). The StorageManger's writeEntries function iterates through each line in the locally stored CSV file. For each line in the file a new entry is parsed and added to a temporary list of Entry objects. This list is later used to construct a ListManager object. On changes to the ListManager's list, the entries in the list are rewritten to the file using the writeEntries function by exporting each entry in the list to a CSV formatted string and writing that string to a new line in the file on the device. We stray from a 100% CSV format when writing tags to the file where instead of being separated by commas, each tag is separated by a period.

Testing

8.1 Test Plan

To test our app, we generally applied integration testing as modules became available. As each module sufficiently passed integration testing, we applied in-house acceptance testing. Our acceptance criteria for a test to pass is straightforward: a module/feature passes if the associated requirement is well implemented and is generally bug free in most use cases. In the case a bug is discovered, a new issue is raised on GitHub describing the expected output, the real output, estimated severity, the commit hash the bug was discovered in, and steps to recreate it.

8.2 Test Result

ID	Functional Requirements	Pass	Fail
1	Create new trip entries on a user given mileage and time	pass	
2	Edit current entries and have changes saved	pass	
3	Correct duration entered when date of entry gets changed	pass	
4	Start a GPS tracked trip	pass	
5	GPS trip ends when user selects end trip	pass	
6	Assign a tag to a already created trip	pass	
7	Add an annotation to a logged trip		fail
8	Filter trips by tag and other properties	pass	
9	Generate report that summarizes every trip in the log		fail
10	Report exports to CSV		fail
11	All buttons in navigation tool bar activate on press	pass	

Version Control

Git was used as our version control system for the project. GitHub hosted our repository to allow for straightforward collaboration between group members. For each change or feature to be added, a new working branch would be created, once sufficiently implemented, the author/owner of the working branch should make a pull request on GitHub to allow other members to review and audit the changes. If others are satisfied with the changes and no merge conflicts exist, then the pull request would be accepted and merged into our main branch to finalize the changes and/or additions.

Conclusion

We believe we've achieved our core objective of creating a time and mileage tracking app. Our resulting app is functional, generally bug-free, and is well documented. We found agile methodologies incredibly effective in constructing an app over short bursts of high effort for features. We admit, not quite everything we planned and came up throughout the semester was implemented, including features such as exporting reports, opt-in cloud synchronization, and many more issues (labelled as enhancements) on our GitHub issue tracker. In the future, most of these enhancements will be further considered for implementation. We think this project further led us to become comfortable in our ability to cooperate in a group setting and problem solving with new technology stacks.

Source Code