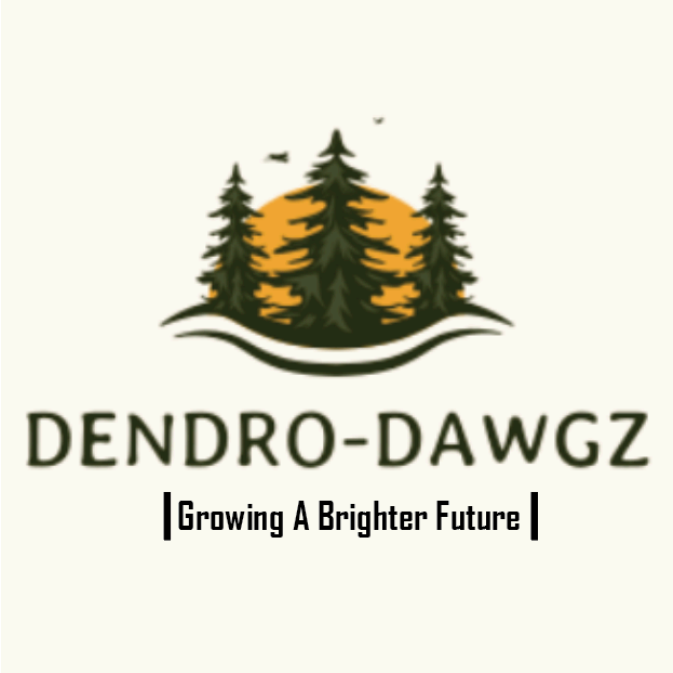
# - Final Report -

Version 1

May 3, 2024




- Team Members -
Nile Roth
Niklas Kariniemi
Zachariah Derrick
Asa Henry

- Sponsored by -
Prof. Andrew Richardson, SICCS/ECOSS
Prof. Mariah Carbone, ECOSS
Prof. George Koch, ECOSS
Austin Simonpietri, ECOSS

- Team Mentor -
Tayyaba Shaheen

# Table of Contents

# 1 Introduction

*Dendrology* is the study of trees, and it continues to be one of the most overlooked sciences. With the current rates of deforestation causing concern around the world, dendrology studies are more essential now than ever. Millions of acres of forest land are torn down every year for the establishment of businesses, farms, housing, and sometimes just for their natural resources. Tree physiology and ecology play a big role in providing the population with the understanding of tree's significant involvement in the balance of life. Our project focuses on easing and supporting research projects like these. More specifically, we are alleviating the processes of installing and obtaining data from dendrometers.

A dendrometer is a data-logging instrument that measures the changes in diameter growth over time. They are generally used on the trunks of trees, but can be used on many different types of plants and expanding objects. The data retrieval process from these devices in trees is very laborious due to the fact that the data is transferred through a short cable that connects to USB. This implies that a laptop must be transported up the tree in order to successfully obtain the data. This perilous process is not only physically demanding, but also puts costly equipment in jeopardy. Portability is the essential feature that this system lacks. To assess the issue we plan to implement a mobile application with an easy to navigate user interface. Users of this application can use their android phones or tablets to easily obtain and observe the dendrometer data. This application also allows for the analysis of multiple dendrometers' output simultaneously.

The simplification of this process will not only ease the lives of those who retrieve the dendrometer data, but will also help improve and increase the flow of data needed for research projects. Many research projects at Northern Arizona University and around the globe depend on frequent data retrievals from these dendrometers, meaning the process should be simple and efficient. When we make this task less of a hassle, there will be an increase in visits to collect dendrometer data, overall leading to a larger existence of research relevant statistics.

The completion of this project has been very satisfactory for us and our clients. We are very happy with the final version of the DendroDoggie application which includes the core functionalities of downloading, visualizing, merging, exporting, and storing data. We also include useful features such as selective downloading, line visibility toggling, and many more. The DendroDawgz team ensures reliability, accuracy, and security in all of these functions. This document will cover all processes completed for the planning and development of this application. We thank you for your interest and support in our application.d

# 2 Implementation Overview

Our intended solution to solve our clients' problems is an Android mobile application that utilizes FTDI chip libraries to communicate with pieces of hardware - a TMD adapter and a TOMST point dendrometer - and display the graphically retrieved data to the user of the application. Our application will additionally utilize Google Firebase in order to export data to the cloud and share it with other users.

The overall approach we are taking with this application resembles multiple software design patterns, namely the Hardware Device design pattern and the Producer-Consumer design pattern. Our application will be communicating heavily with an external piece of hardware, and afterwards it will be consuming that data and converting it to a user-readable format.

The frameworks and packages that will be key for our application include the Android SDK and NDK, the FTDI chip library, Google Firebase, and MPAndroidChart. The Android framework will be a critical component of our application, because the application itself will be built upon this framework, utilizing the SDK for user interface and backend components, and the NDK for native device communication - such as communicating with the dendrometer. We will be using both the official C FTDI chip library and the unofficial Java FTDI chip library in order to directly communicate with the hardware. Google Firebase will be used to export and store user data, authenticate users, and share data with other designated users. Finally, MPAndroidChart will be used to graphically display the data to users of the application in conjunction with the Android SDK. In total, our mobile application will be written predominantly in Java and Kotlin, with a moderate amount of C for communicating with the hardware.

# 3 Project Requirements

In order for our team to implement an application capable of reading in data from a dendrometer and displaying it to the user in a manner acceptable to our clients, we must follow the following outlined domain-level requirements. These key requirements are split up into functional requirements, non-functional requirements, and environmental requirements, and will be evaluated in the following sections.

**Key Requirements**

- The application will be implemented on a mobile platform, specifically Android.
- The application will contain a backend written in a low-level language, and be capable of reading in data from a TOMST point dendrometer.
- The application will implement a frontend that initiates data-reading from the dendrometer, and downloads the data to the appropriate location.
- The frontend will contain a graphical user interface capable of displaying the dendrometer data.
- The application will support metadata creation and a master file grouping system, so that several dendrometers can be tracked in the same file and by the same graphs. Additional information will also be provided with each set of data.
- The application will implement a cloud export and sharing system, to allow the users to upload data to the cloud and share with other users.

## 3.1 Functional Requirements

The basic product the team is aiming to produce is an Android application which can interact with a TOMST dendrometer to pull data and save the data to the device. The application should provide an interface to select 1 or more datasets and another one to create a graph (visualization) of the selected data. The user should be able to run a variety of analyses on the graph, and should also have the capability to save the resulting graph to the device. The users should additionally be able to select a window of time that they would like to view or run statistical analyses on. Lastly, the application should provide a way to upload the data to a cloud service for easy sharing.

### 3.1.1 | Data Visualization

Creating a visual representation of the data collected requires there to be what can be understood as a canvas onto which data signifiers (i.e. points, lines, etc.) can be drawn. Knowing what data is desired can be handled with the "data selection" function – and will be covered later – for now we can assume there is some dataset. Depending on if the format of dendrometer data is unique or standardized, the application will first need to pull the data from the file into memory in a parsable structure. Then this structure can be queried for x- and y-axis names and values to add information to the visualized graph.

- Use Case: User uses the application to run linear regression on data from a single dendrometer
- Actors: User, Application
- Process Flow:
    - User chooses "Visualize"
    - Application loads *Visualization* screen
    - User loads dataset from dendrometer
    - Application updates *Visualized Data* canvas with selected dataset
    - User chooses "Analyze"
    - Application presents *Analyze* menu
    - User selects "Linear Regression" algorithm
    - User chooses "Select"
    - Application run *Linear Regression* algorithm on new dataset
    - Application updates *Visualized Data* canvas with analyzed dataset
    - User chooses "Export"
    - Application presents *Export* menu
    - User enters a filename for visualization
    - User chooses "Export"
- Preconditions:
    - There is at least one dataset from a dendrometer present
    - Application has a designated directory for reading/writing data and visualization
- Postconditions:
    - Application now holds user's visualization at <path>/<filename>

### 3.1.2 | Data Reading

The most critical requirement our project should satisfy is a successful transfer of data. Our software must have the ability to accurately obtain and store the data file created by the dendrometer. This process is perilous due to the necessity of precise translation. When uploading data from the dendrometer, csv (comma separated values) files are produced containing raw data with low readability. A third csv file must be created by the application to present the data in an organized structure. The goal of this translation is to support the process of reading from the file

and inputting to the chart. With this goal achieved, the chart creation should be seamless and require only simple file reading expressions.

- Use case: User transfers data from dendrometer onto their mobile device.
- Actors: User, Application, dendrometer, computer
- Process Flow:
    - User opens application on their mobile device
    - User chooses ‘Create New Set’
    - Application halts and waits for connection to dendrometer
    - User connects dendrometer to mobile device via USB to TMD adapter cable
        - This step may require the addition of a USB to USB-C adapter depending on the user’s mobile device.
    - Application identifies connection and begins uploading process
    - Computer generates CSV files based on dendrometer commands
    - Application compiles these files into a single, more user readable CSV file
- Preconditions:
    - Dendrometer has measured data to upload
    - Adapters are fully functional with no ware that may cause faulty connection
- Postconditions:
    - Application now holds user accessible data that was downloaded from the dendrometer

### 3.1.3 | Data Selection

Obviously, data visualization does not work if the user cannot choose what dataset(s) they would like to visualize and run analyses on. On the other hand, the selection interface needs to be relatively effortless to navigate. Therefore, a canvas should appear over the chart when a button is clicked. On the canvas, the application will show a hierarchy of the contents of its designated directory where graphs are grouped and stored (the root of the hierarchy being the application’s designated directory). As the user clicks through directories, the contents will be right justified to signify they are children of the selected item, and the most recently selected item will be highlighted.

- Use case: User loads one or more datasets for analysis and visualization
- Actors: User, Application
- Process Flow:
    - User chooses “Add Layer”
    - Application presents *Load Data* menu
    - If desired dataset is in a directory
        - User selects directory
        - User repeats until desired dataset is found
    - User selects dataset
    - User chooses “Add”

    - Application closes *Load Data* menu
    - Application updates *Visualized Data* canvas with dataset
- Preconditions:
    - There is at least one dataset from a dendrometer present
    - User has navigated to the *Visualization* screen
- Postconditions:
    - *Visualization Data* canvas is populated with the chosen dataset(s)4.1.4 | Data Storage

A big part of the program will be the ability to store data taken from a dendrometer, and store it on the cloud. More specifically, the application will take any inputted data and store it on the cloud service AWS. The data will be stored within AWS on a database called DynamoDB, which is a very fast and scalable NoSQL database. This gives the application the best possible performance when it comes to moving large amounts of data. It will also give users the ability to access their uploaded data from any device, not just the device used for uploading the data.

- Use case: User wants to store data on the cloud and not on a physical device
- Actors: User, Application, Cloud
- Process Flow:
    - User clicks the share button
    - User will need to input username and password
    - User clicks on login button
    - User selects where in the database they want to save the data
    - User clicks on share button
    - Application displays a progress bar of the upload
    - After data is uploaded user can click on the back button to return to the visualization screen
- Preconditions:
    - Data from the dendrometer, an account to link data to
- Postconditions:
    - Data will be stored on the cloud database

### 3.1.5 | User Authentication

To support the process of storing data, the users will have to sign in. Users will be given a couple different options to sign in. They can either use a google account or they can create an account. If they create an account that info will be stored securely on the cloud. When users sign in they will be able to see all the data that is linked to their account. If a user uploads data from a dendrometer, they will be the manager/admin of that data. These administrators can add any other users to be viewers of that dataset. This will make sharing the large amounts of data very easy and efficient.

- Use case: User links data to their account and shares data with other users
- Actors: Users, Application, Cloud Authentication

- Process Flow:
    - Application prompts user with sign in screen
    - If user already has log in or chooses to log in with google
        - User will input username/email and password
        - User will click login button
        - Application will verify if information is correct
    - If user does not have an account or does not want to use google
        - User will click create account
        - User will then input an email, username, and password
        - User will click finish creating account button
        - Application will run a verification on the email
- Preconditions: None
- Postconditions:
    - User will have access to all data linked to their account.
    - User will have an account they can link data to.

### 3.1.6 | Multiple Layered Graphs

An important feature of our application is the ability to layer multiple different data sets on a single graph. This allows for the comparison of multiple dendrometer readings simultaneously. This is especially useful when multiple dendrometers are installed on the same tree or in the same vicinity. Many research projects focus on studying these circumstances and therefore require these overlapping capabilities. Merging several dendrometer data files together is also an important function that should be implemented in addition to layering several files together, allowing the end user to save and combine several dendrometers permanently.

- Use case: User adds a new data layer to their graph
- Actors: User, Application,
- Process Flow:
    - User selects “Add Layer” button
    - Application accesses database
    - Application displays list of directories/files
    - User selects desired data set to add
    - User selects “Add To Graph” button
    - Application adds new data set to existing graph
    - Application displays multi-layered graph
- Preconditions:
    - User is logged in
    - User has an existing graph opened in the application
    - Data sets are already uploaded from the dendrometer to the Application’s database
- Postconditions:

    - Multiple data sets visible on the same charting plane.
    - Remove layer action becomes available

### 3.1.7 | Saving Created Graphs

Saving a graph in the application entails uploading a raw data file onto the database. This is a short and easy process that only involves the application and the database. This feature is obviously crucial to the productivity of our software, and therefore must be consistently functional. If a user is unable to save his/her work, they are unable to reference any previous findings which will be a hindrance to research. Saving capabilities will not only help to support progress, but it also creates feasibility for a graph sharing feature. There should additionally be a way to merge and save a new file with the dendrometer information from two or more dendrometer data files.

- Use case: User Saves a graph to the database
- Actors: User, Application, database
- Process Flow:
    - User selects “Save” button
        - In the case graph is new (not yet saved) button will display “Save as”
        - Application displays new dialogue box with current working directory and a text box for filename
        - User clicks through directories until at desired file path
        - User inputs desired filename
        - User selects “Save” button
    - Application uploads graph file into database
- Preconditions:
    - User is logged in
    - User has an existing graph (with data) opened in the application
- Postconditions:
    - Graph file is stored into the database

### 3.1.8 | Sharing Created Graphs

Once the user has one or more datasets and visualizations they wish to share, there needs to be some way to upload them to the remote database so others can view and download that information. This would be done through a two screen interface. The application needs to know which user is uploading data, what data they wish to upload, and the destination database. The first screen will ask the user to provide a username and password. When login is successful, the user will then be presented with a list of directories and files at the application’s path, and a list of databases to which they can upload.

- Use case: User wants to share a visualization from two dendrometers with their colleagues

- Actors: User, Application, Database
- Process Flow:
    - User chooses “Share”
    - Application loads *Share* screen
    - If desired dataset is in a directory
        - User selects directory
        - User repeats until desired dataset is found
        - User selects dataset
    - Otherwise, user selects dataset
    - User selects “Database 1”
    - User chooses “Share”
    - Application presents *Upload Progress* menu
    - Application starts upload process
        - User waits for upload process to complete
    - Application finishes upload process
    - User selects “Back”
- Preconditions:
    - One or more datasets and or one or more visualizations are present
    - User is logged in
- Postconditions:
    - A copy of the given data is stored in the cloud

### 3.1.9 | Deleting Created Graphs

Deleting a graph from the database is also a simple process. A user may want to delete a file in order to free up space, erase irrelevant data, and/or maintain a more organized work space. This process only requires a quick access and modification of the database. Users should have the option to delete a graph anytime their stored data is displayed to them. This includes when attempting to open and share a graph. The deletion process from the user’s perspective is quick and easy, only requiring clicking a couple buttons.

- Use case: User Deletes graph from database
- Actors: User, Application, Database
- Process Flow:
    - Application displays file system
    - User selects desired file to delete
    - Application provides user with actions (one being delete)
    - User selects trash can icon (delete)
    - Application sends query to database to remove file
- Preconditions:
    - User is logged in

    - Database is displayed from an attempt to open or share a file
    - Database contains at least one file
- Postconditions:
    - Deleted file no longer exists in database

## 3.2 Non-Functional Requirements

### 3.2.1 | Swift Rendering Times

Users desire a program with minimal stand-by time. Little to no time should be spent waiting for a response from our application. Charts should be rendered quickly despite a large data set. For massive sets, we may require the use of parallel programming. We can use multiple threads to input data into our chart simultaneously. Any wait time on any front of our application should be preceded with some sort of message informing the user of what they are waiting for.

### 3.2.2 | Data Stored Securely

One of the most important things to consider when implementing user-specific data storage is the security of the data in every step of the process. Users should be confident that their data is secure and protected both in the cloud and in transit, and our application must ensure this safety. Thankfully, Google Firebase - the cloud provider we are using with our application - implements several contemporary approaches to data security. All of Firebase's services are certified by ISO and SOC security compliance standards, which shows they are considered secure by official sources. In transit from the user's device to the cloud database, data is encrypted using HTTPS. When data is at rest in the cloud database, the data is also encrypted using the AES-256 encryption algorithm.

### 3.2.3 | Simple and Aesthetic User Interface

Our application will hold an architecture that is easily navigable and understandably functional. Users should know the functionality of all of our features without having to learn through testing. To accomplish this, all buttons and other means of I/O in our application should be descriptively labeled. We plan on creating a 'FAQ' or 'help' page in order to further improve user clarity. Along with a simple user interface, we also commit to providing a clean and appealing aesthetic to our application. This will require strict attention to color theory and CSS during implementation. Our webpage will hold an organized format that is pleasing to the eye, and interactively logical.

### 3.2.4 | Interactive Graphs

Being able to visualize data is very important to understand relationships between factors such as temperature and tree growth. To enhance the capabilities of data analysis, the application

will offer the user ability to adjust the view and to zero in on points of interest. This will require some understanding of gestures so that screen input has an effect on the graph. In addition, the team would need to convene with the client to decide on a simple and easy gesture for zooming in and out of an area on the graph.

### 3.2.5 | System Reliability

Having an application that is consistently functional and available is an important factor. A user should be able to enter the application without the occurrence of any issues or crashes. A user should also be capable of using any of the features the app offers without fail. If a user wants to load data from the database, they should be able to without any issues arising. Success should also be assured when a user wants to share any data. A user should also be able to trust the analysis done on any of the data. The algorithms should be fully functional and always provide correct results. All of these features included in the application should be working 99.9% of the time to help create an enjoyable and useful experience for the user.

## 3.3 Environmental Requirements

### 3.3.1 | Restriction to Android Devices

Our mobile application will be limited to the Android platform due to the Apple ecosystem and its general requirements surrounding connected accessories. In order for a connected device to be able to be used, it must fall into a strict program laid out by the company, known as the MFI (Made for iPhone) program. The company making the accessory must apply for this program and follow their additional regulations in order for any iOS device to work with the accessory. These regulations generally involve changing the circuitry, accessory software, and/or production process. TOMST has made it clear to us that they do not plan on joining the MFI program in the foreseeable future, leaving us with just Android.

### 3.3.2 | Restriction to C and Java FTDI Libraries

The TMD-USB, Temperature Measure Device, adapter that our application will be communicating with utilizes Future Technology Devices International Limited (FTDI) chips to communicate with the dendrometer. In order to read in data, we need a library to work with these chips - the only official library existing in C. An unofficial library exists in Java, and while it is certainly usable, the C library is generally the preferred option. Without using either of these libraries we would be forced to implement our own FTDI communication functions using standard USB libraries in the chosen language, which would be too long and unnecessary. Therefore we are restricted to using either Java or C to implement our FTDI chip communication.

### 3.3.3 | Restriction to USB TMD Adapter

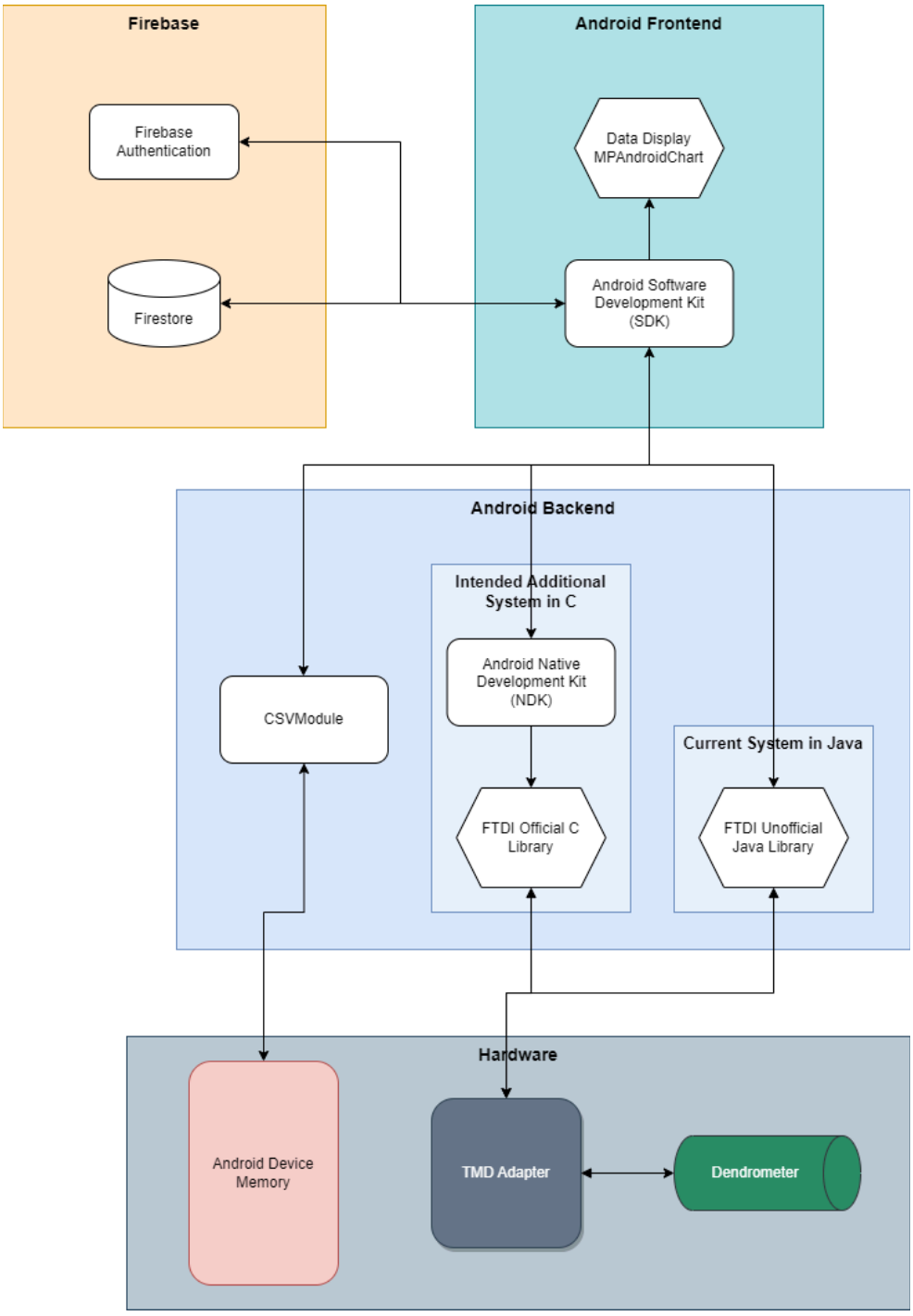
Communication with the dendrometer is only possible through TOMST's TMD adapter. This is because there is a specific software flow that initiates and performs the communication that is set up through their adapter. Trying to use a separate two-pin connector would not work, because the translation of data and reading of commands is done through the TMD adapter. This means that we are restricted to using just TOMST hardware for all of our dendrometer communication, and would not be able to use any other accessory (like the Apple iButton).

### 3.3.4 | Restriction to Wired Connection

Communicating with the dendrometer can only happen through a wired connection, through the TMD adapter. Bluetooth, radio, or another wireless communication method would make the functionality of the data-logging instrument much better, due to the fact that users could access the device and data from the ground. Unfortunately wireless functionality is not supported by the dendrometer. TOMST has explained to us that bluetooth and other wireless communications are not being considered in the foreseeable future, so we will have to disregard this path and stick to wired connections.

# 4 Architectural Overview

## 4.1 | Architecture Diagram



## 4.2 | Architecture Discussion

The architectural pattern we are modeling our Android application on is a layered model-view-controller architecture (MVC). In our system we have several layers that help separate distinct domains within the software and contain specific responsibilities, including the UI/Frontend layer, The Domain Layer/Backend, the Hardware layer, and the Cloud layer.

The frontend is responsible for providing the entire user interface for the mobile application, which is done through the Android SDK and MPAndroidChart. The key responsibilities of this layer include providing the interface that begins the dendrometer data download, the graph display of dendrometer data, selection of dendrometer data to be displayed, and other typical interface functions like settings and app-specific options. This layer communicates with the backend frequently, through the Android SDK and NDK, typically to initiate dendrometer reading, or CSV file manipulation. This layer also communicates with Firebase at the cloud layer through the Firebase API to log users in, to store and retrieve user specific dendrometer data, and to share data amongst users.

The backend is responsible for most of the heavy lifting within the application, communicating with the hardware layer frequently through the FTDI chip library both in C and Java. We plan on moving as much Java code over to C as possible. The backend sends commands through the FTDI API to the TMD adapter, which retrieves data from the dendrometer and sends it back to the application. The backend translates this data into human/graph readable CSV files and stores them locally on the device. Manipulating the CSV files and loading them into the application using the CSVModule, and reading data in from hardware are the key responsibilities of the backend.

The cloud layer consists entirely of Google Firebase services, including Firebase authentication and Firestore. The key responsibilities of this layer are to communicate with the frontend in order to log users in, store user specific data, and share data between users. Firebase handles the specifics of authentication and the database, and our application communicates with it through the Firebase API.

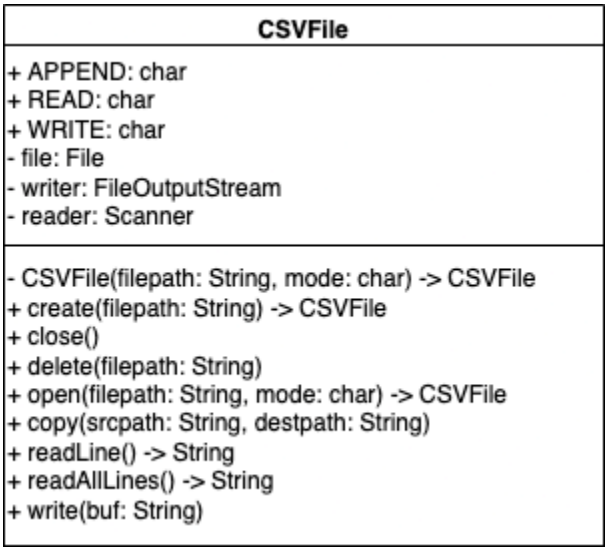
# 5 Module Interface Descriptions

## 5.1 | CSV Module

### 5.1.1 | Description

Because our application will allow the visualization of one or multiple datasets from TOMST dendrometers, a module to handle reading and writing CSV files is required. The module will be responsible for reading a CSV file into memory to be displayed, writing mutated data to a new file or overwriting a file, as well as copying a file. In the event the data is mutated or copied, the module will create a new CSV file and write the data.

### 5.1.2 | UML Diagram




### 5.1.3 | Public Interface

Making up most of the CSV Module is the “CSVFile” class which provides other modules a concise way to manipulate datasets without having to interact with Java’s file IO interface directly.

The CSVFile provides two ways of opening a file for data. The first way to open a file is via the public, static “create” method which wraps a call to the *createNewFile* method on Java’s File class. The *create* method accepts a string specifying the path at which to create the file, as well as the name of the file to create. Upon successful creation of a file, the *create* method will return a CSVFile object through which a module can interact with the new file. By default, the *create* method opens a file in “write” mode, since no content will exist in the new file.

The second way to open a file is via the public, static “open” method which handles opening a reader and writer to interact with a file on the system. The *open* method accepts a string specifying the file path, as well as a mode to indicate whether the module intends to write or add to, or reader content from the opened file. Upon successful opening, the *open* method will return a CSVFile object through which the module can interact with the opened file.

Once a file is created and or opened, the public “write”, “readLine”, and “readAllLines” methods become available to a module. The *write* method allows the writing or appending of content to an opened file. The *write* method accepts a buffer string, of any length, and writes the contents to the opened file. Whether the content is written or appended, depends on the mode specified when opened; otherwise, the content is written if the file was created.

The *readLine* method accepts no parameters, and returns the content from the last newline character (\n) read to the next newline character. Similarly, the *readAllLines* method reads all the content in a file, and returns contents as a string. The file must be opened in “read” mode to use these methods.

For the event a module needs to make a copy of a file, the CSVFile provides a public, static “copy” method. The *copy* method takes the path to the file to be copied, and the path to which the file will be copied. The *copy* method takes care of opening the source file, creating the destination file, and moving content from the source to the destination.

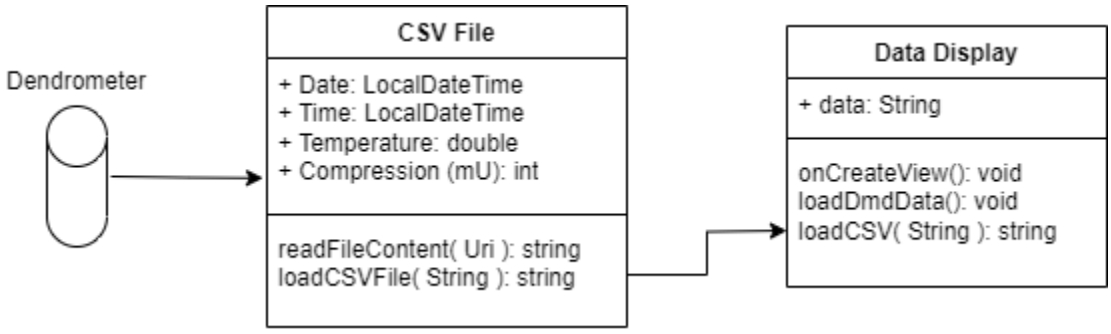
When a module is done working with a file, a public “close” method is provided. The *close* method takes no parameters and returns nothing; this method is solely used to close the reader and writer when a module is done working with a file.

## 5.2 | Display Data Module

### 5.2.1 | Description

Creating a visual representation of the data collected requires there to be what can be understood as a canvas onto which data signifiers (i.e. points, lines, etc.) can be drawn. The data collected from the dendrometer comes in the form of large csv files, and must be translated into a useful line graph. This module is the most essential part of our application because it makes up the biggest and most used UI component of our software. Users must be provided with consistent and precise visualization of their data sets. In order to accomplish this transition of raw data into useful statistics, we utilized the charting library MPAndroidChart.

### 5.2.2 | UML Diagram




### 5.2.3 | Public Interface

The majority of work done for this module is handled by MPAndroidChart. This library is widely used for the organized representation of data, and has the ability to implement interactive, appealing charts and graphs. As stated in the previous module, the data collected from the dendrometers are brought into our application as CSV files. More specifically, each line in the data file corresponds to a single measurement taken by the dendrometer. A single measurement consists of a reading of the date, time, temperature, and compression (tree growth). After this data is obtained from the file, methods in the GraphFragment.java file utilize many of MPAndroidChart's functions to transform the data into a precise graph.

A method named "onCreateView" is executed on the click of "view data". This creates an empty X-Y plane for the line data to be displayed on. The "LoadCSVFile" method uses a for loop to scan through the file entries and create an input set that is recognized by the charting library. This method utilizes the sub-functions processLine and readLine. readLine simply reads a line of the CSV and returns it in a string format. We can input this string as the parameter of processLine to parse the string and extract its data.

At this point we can simply use MPAndroidChart's setData function which outputs the set of data onto the already created empty graph. This is done by the private void "LoadDmdData" method which also incorporates an animation and initial setting of view for the user.

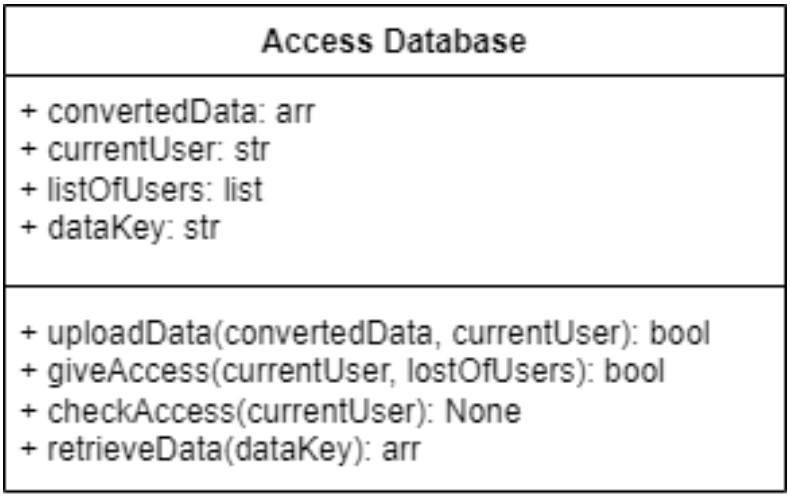
In some instances, users may desire to view multiple data sets on a single graph. This allows for an efficient comparison between two or more dendrometer readings. We are currently finishing up final touches on a new method "mergeCSVFiles". This will be called upon to combine the components of multiple CSV files. LoadCSVFile handles the input of a merged file, and is able to process the lines of more than one file into multiple data structs. The reading from MPAndroidChart is the same process, just called once for each dataset.

## 5.3 | Database Module

### 5.3.1 | Description

Our mobile application will need to have the ability to upload data to a cloud database. This is the reason the application will include a module for the database, which in our case in Cloud Firestore. The module will be able to handle any oncoming data. The module will upload the converted data to the cloud, where it will give access to that data to whoever uploaded it. If anyone else needs access to the data, the module will be able to give other users access to it. The module will also handle retrieving any previously uploaded data.

### 5.3.2 | UML Diagram




### 5.3.3 | Public Interface

All of the methods that are required to communicate with the database will be under the AccessDatabase class.

The uploadData method is responsible for taking the converted data and uploading it to the database. It will take in the converted data and the current user as the parameters of the method. Since the application will be connected to the database at all times, there is no need to pass in any connection parameters.

Once inside of the uploadData method, the module will access the upload function that is provided by Firebase. Before uploading any data, the module will create a collection for the data file, this will be used later on in the giveAccess method and checkAccess method. The data that has been passed in is split into two columns. The module will access the first column in the database and take the first column of the data and upload it. After that is done it will do the same with the second column of data. After all the data is uploaded, the module will access the giveAccess method to set the permissions for the data. After that is all done, the class will

determine if the upload was successful or not. This boolean value is what the method will return so it can be displayed to the user.

The giveAccess method is responsible for giving the proper access to the data that is uploaded to the database. For the parameters it will take in the current user and a list of users that the current user can pass in. The current user and the list of users are who will be given access to the data in the database.

In the Cloud Firestore there will be a collection, or a list of users, for each data file uploaded. The module will access this collection and add the current user and list of users that was passed in. This will give those users access to the data if they are signed in with that account. The method will return a boolean value that says if giving access was successful or not.

The checkAccess method is responsible for checking if the current user that is signed in, has permission to access a specific data file that is on the database. The method will take in the current user as the parameter.

After getting the current user that is signed in, the method will check all of the collections, or list of users, that are in the database and find the ones that the user is in. The method will then take the list of data files that the user has access to make that data accessible for the user. The method does not have to return anything.

The retrieveData method is responsible for retrieving data that the user is requesting. Since the checkAccess method will have already ran, there is no need to check if the user has access to that data. The parameters for this method will be the id, or key, that is linked to the data file the user wants to retrieve.

Once in the method, the module will access the retrieve function that is provided by Firebase. For this function, it will need to pass in the id that was given to the method. After finding the data, the method will return the data retrieved so it can be used elsewhere in the application.

## 5.4 | Plan vs Deliverable

There were few parts of our deliverable that strayed from our initial plan. One of the main parts that strayed from our initial plan was the database. After doing our feasibility report, we decided on using AWS for our cloud infrastructure. However, after some time we got in contact with TOMST and started a collaboration with them to complete the app. TOMST was using Firebase for their cloud infrastructure so we decided to change from AWS to Firebase. Then after we got deeper into the implementation of the app, we came across an issue. In order to use a database we would have to manage every row and column of our csv files. This would take a lot of work, so we decided on switching over to cloud storage. This allowed us to simply upload and read in our csv files as we needed.

There were also a couple stretch goals that were never addressed due to a higher priority in polishing and debugging:

- Statistical Analyses

    - Identifying useful components of the graph such as most growth in a day, linear regression, ect…
- Parallelized Downloading
    - Using multiple threads to speed up the graphing process.

These would both be great additions to the app if development does continue.

Finally, in the concluding paragraphs of the section, you should discuss how what you built differs from what you intended to build: in other words, the differences between your prescriptive and descriptive architectures...between "as-planned" and "as-built". Example dimensions that could be applicable: What functions did you end up not building and why? What functions did you implement in a different way than what you planned and why? What architectural decisions did you have to change during development, and what drove these changes?

# 6 Testing

---

The DendroDawgz team is devoted to easing the process of obtaining and analyzing dendrometer data in order to support research projects around the world. Our long-term goal is to increase the overall flow of dendrometer data, leading to an improvement of the contributions of tree research. We have successfully created an alpha version of our application DendroDoggie with the ability to accurately perform all of our core modules. These include reading in data from the dendrometer, merging csv files, visualizing single and merged data files, and exporting data to the cloud. Before a final version release, we must run these processes through a series of tests to confirm their full functionality. There are three different types of tests to be performed on our application: Unit, Integration, and Usability testing. Unit testing relates to the actual testing of code and its outputs. We plan to run unit tests with JUnit on the most critical java files in our system. Integration testing involves ensuring that data is seamlessly transported throughout the application. This type of testing is important for our application because of its prevalence when switching between our application’s multiple views. Lastly we have usability testing, which will be very useful for improving our user interface and user-friendliness. This process will involve having our clients complete a series of basic tasks inside the application. We are mainly looking to obtain feedback about the intuitiveness of our front-end, so that we can improve upon features of our app that may be difficult for new users to understand or reenact. The structured approaches described will ultimately lead to a more robust and user-friendly application. In the following sections, we delve into each testing area in detail, outlining specific test cases, methodologies, and expected outcomes.

## 6.1 | Unit Testing

Unit testing is a strategy which picks crucial functionality in a software system to test. The tests can be run any time, but are generally run before code makes its way to end users. Unit tests are written to provide a benchmark for the system; this is useful when fixing bugs and adding new features because pieces of the software can be changed which could affect other pieces. By having a collection of tests to verify the software is producing expected results and to verify the software can handle edge cases, the integrity of a system can be confirmed during and after the development process. Additionally, unit testing can be a critical part of regression testing, allowing developers to quickly see if new code changes cause bugs or failed unit tests inside other components. This allows these developers to easily see if their changes alter previous fixes or versions of the code, and update the changes accordingly.

### 6.1.1 | Unit Testing Plan

To ensure the integrity of the DendroDoggie application, the team has decided to employ the standard and widely used “JUnit” testing framework to test our software. Junit is the most commonly used Java testing framework on GitHub, and is consistently worked on and updated to provide the most testing tools possible. JUnit also can be used extremely easily within the Android Studio build process, by integrating the library with the gradle file. You are able to customize the test implementation runner within the build.gradle file, and simply adding JUnit there ensures that testing is properly integrated with your development process.

JUnit contains all of the expected test cases, including an initial setup of variables and environment, assertions, mocking and stubbing, test runners for suites and classes, and teardown - to name a few important functions. With these functionalities we will be able to thoroughly test the most important pieces of our software to ensure that everything works as expected, and continues to work as expected as future developers maintain it.

Over the following section, we will describe what classes we are testing and why, and go into specifics of testing inputs for the functions we want to have coverage of, setup requirements, and potential edge cases.

- pars.java
    - Because this class is used to parse the information that we receive from the dendrometer and other TOMST devices, this class is a critical piece of the core software that needs testing.
    - copyInt: inputs should be a string with an integer the length of two characters inside it at a given start point. Example: “12345678” starting at 0 with a count of 2 should return 12.
    - copyIntGTM: should be capable of handling the same input as copyInt, but additionally should handle hex values, and edge cases for negative UTC offset values. Anything over 0x80 should be subtracted from 0x80 and negated. For

example, “A8” should return -28, because 0xA8 - 0x80 = 40, and 40 to decimal is 28.
    - copyHex: Similar to the above functions, if the string is “A8” this one should return 168 as expected in hex.
    - dissasembleDate: This function uses the previous functions inside of it, and needs a class to be set up and initialized before testing. This should be tested with edge cases for the UTC offset, and normally expected values.
    - disassembleData: A class also needs to be set up and initialized for this function, and the inputs should contain both “ADC” and no “ADC” inside the input string for full coverage of the function. An assertion of all individual class values populated from the return will be run.
- CSVFile.java
    - The CSVFile class resides in the application’s backend, and provides a simplified interface for changing the contents of files, as well as adding, removing, copying, and transforming files for the programmer
    - toParallel: should accept a merged file which is in the “serial” format - where dendrometer data is listed one after the other - and restructure the data to be in “parallel” - where data for each dendrometer is separated into columns (i.e. there are N points of data on one line - where N refers to the number of dendrometers in the file for which data exists).
    - toSerial: should accept a merged file which is in the aforementioned “parallel” format and restructure the data to be in the aforementioned “serial” format
- GraphFragment.java
    - The GraphFragment class is responsible for implementing the functionality the users need to visualize and manipulate data. This class is a critical component of the user interface, and is the main piece with which the user will interact.
    - DisplayData: this function does not accept any parameter; however, the function works with data previously constructed by the *loadCSVFile* function. This function must be able to parse N collections of data from N dendrometers - where N communicates the number of data sets from a dendrometer contained within a file. More specifically, this function needs to be able to identify and extract each piece of data - differentiating between data from temperature sensors, humidity sensor, growth sensors, etc. - and organize the data into a structure which the chart can use to visualize data for the user. Testing this would entail processing some mock data, and then asserting the pieces of the charting structure are what is expected.
    - loadCSVFile: should accept a file path, and be able to load the contained data into memory for further processing and/or visualization. The function should be able to handle files with a single data set and merged files. The function should be able to keep track of which data belongs to which dendrometer. Testing this function

would require asserting the filled data structures have the expected data and the expected organization.

- processLine: should be able to parse a line read from a data set file. The function should be able to identify if the line contains a serial number; otherwise, the function should be able to extract temperature, humidity, and growth data the device collected. The function should be device agnostic which would allow for our clients, and other users by extension, to visualize data from any of TOMST's devices.
- mergeCSVFile: should accept an array of file names and compile the data set(s) from each file into one file - known as a merged file. The function must apply a header which contains the number of data sets, the serial numbers of the dendrometers' data contained within, as well as each dendrometer's latitude and longitudinal position. The function should also be able to merge data from a merged file with other, single data set files, and other, merged files. An example: if the user already has a merged file with 3 data sets in it, and wants to merge it with a file with a single data set, then the resulting file would count "4" data sets, list the 4 dendrometers, as well as contain the data for each dendrometer present in the source files. If the user has two merged files - one has 3 data sets, and the other has 4 data sets - the resulting file would count "7" data sets, and contain the data present in the files which were merged.
- LoadDmdData: does not accept any parameters; however, the functionality through which the user can visualize data as they collect data from the dendrometer. Therefore, it is imperative the function can parse data - coming from the dendrometer - and put the data somewhere from which it can be changed and saved. So, the function needs to be able to access and read data from the dendrometer - passed through the DmdViewModel by the HomeFragment - and parse the data, then collect the data into a construct which can be visualized, and finally convert the construct into an object which the chart can use to visualize.

- ListFragment.java:
  - This class is responsible for updating the list that resides in the File Viewer page of the application. Some of the functionality within this class is responsible for reading and writing the csv files to the cloud. The class also handles the initialization of merging csv files into one csv file. In order to initially populate the list the class will pull files from the device and the cloud.
  - loadAllFiles: This function will take all the files that are stored on the device and populate the list with them. The function should be able to take the list of files on the device, fFriends, and create an adapter with it. Then it will take that adapter and set it for the list view. In order to test this we would need to check if fFriends has the correct csv files. We would test this against the known list of csv files on the device.

- loadFromStorage: This function should be able to take all the necessary csv files from the cloud and load them into the list view. It will do this by going through each csv file on the cloud. Then for each file it will check if the current user is in the list of users that has access to the file. The list of users with access is stored in the metadata of the file. If the current user has access to the csv file, it will download the file by calling the downloadCSVFile function. This will add the file to fFriends and the function will then update the list view with this new fFriends by setting a new adapter for the list view. In order to test this functionality we will check if the current user is actually part of the user access list. The test will also include checking if the pulled file from the cloud is in the fFriends list.
- downloadCSVFile: This is a fairly simple function that will take in the file name and file path. It will then pull the file from the cloud based on the file name input. Then for this file it will download it to the path that got inputted. To test this function all we have to do is check if the file is in fFriends.
- uploadDataToStorage: This function is responsible for uploading csv files to the cloud. The function will take all the files that are selected and go through each one. To get the selected files it will check the selected flag for each file in fFriends. For each file, the function will upload it to the Files folder in the cloud. After a successful upload, the function will update the metadata to include the user who uploaded the file in the file access list. There are two things to test here, if the file got uploaded and if the metadata got updated. To test if the file got uploaded, we would go through the files in the cloud, and check if the file is in there. Then if we do find the file, check if the metadata includes the user who uploaded the file.
- updateMetadata: This function gets called from the shareData function. If a user clicks share and inputs email addresses, the shareData function will then call updateMetadata. The updateMetadata function is responsible for adding users to the file access list. The function will take in an array of emails and go through each selected csv file. Then for each file, the function will go through each inputted email and that email to the user list for the file. This allows users to share files with each other. In order to test this, we would take the file that got updated and check if all the emails got properly added to the access list.

### 6.1.2 | Unit Test Results

Because the clients provided a few points of improvement on the current software being used – Lolly – the team focused heavily on ensuring the features outlined worked reliably. This is what guided us when deciding the list of classes and methods above to test. There were minimal problems with unit testing except the challenge of verifying methods which used “Toast” to display text to the screen to update the user on what the application is doing.

However, usability testing revealed a few improvements and changes which influenced what unit tests for the CSVFile class look like upon release.

## 6.2 | Integration Testing

Integration testing is another crucial part of ensuring software works as expected and even fulfills initially given software requirements. Unlike unit testing, this version of testing works by comparing values that are passed between interfaces instead of focusing on the specific assertions of function return values.

For example, if within one view of an Android application, you input your user credentials and it takes you to a separate view with that user's data, you want to ensure that the credentials properly passed from one view to the other, and that the Android application went to the intended view after checking the user's input.

Integration testing in our Android application will be fairly straightforward, as we only have a few interfaces that we need to test proper flow and data passing. For our tests, we are going to use an emulated environment present on Android Studio, and use the robust logging and debug system to ensure that all components are thoroughly examined and meet our defined requirements.

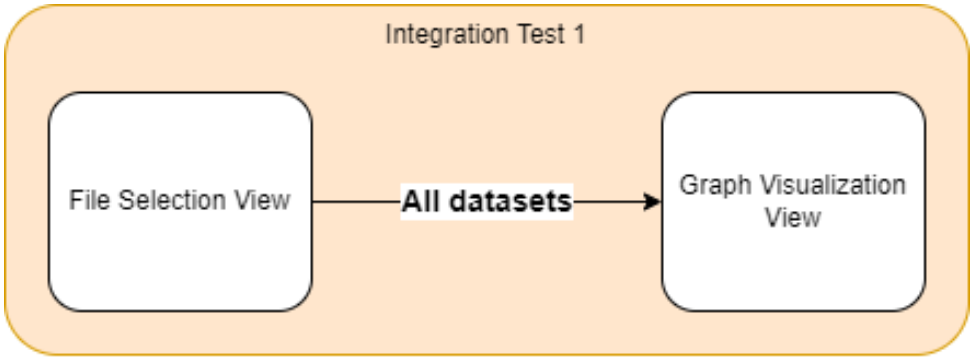
### 6.3.1 | Integration Testing Plan

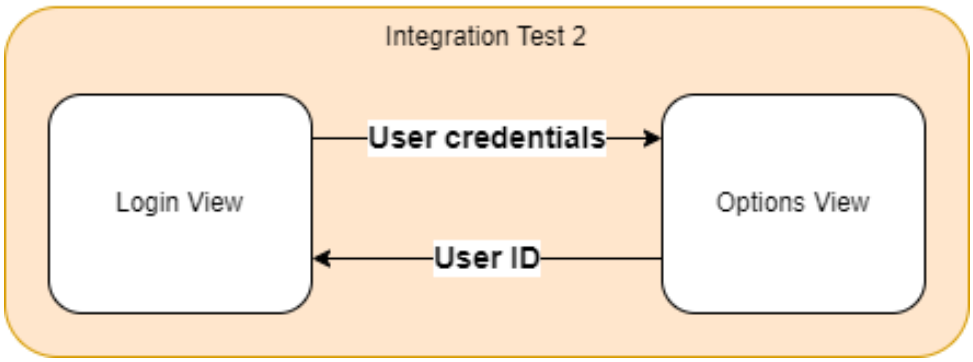
- For our first integration test, we will be analyzing the relationship between the file selection view and the graph visualization view, and the process of passing large amounts of data between these views. It is important that the data is the same between views, and that the application is capable of handling several use cases and functional requirement metrics.
    - Preparing the test data: First our emulator needs to have the required test datasets present on the device. There will be four datasets that are examined: one large dataset with 50,000 entries and a header, one small dataset with 1,000 entries and a header, one dataset with 1,000 entries with no header/metadata information, and two datasets that will be merged during the selection/viewing process: the 50,000 and 1,000 datasets will be merged together with the appropriate header information to distinguish them. The first three datasets need to be created and placed in the emulator's internal storage. The fourth will be created during runtime.
    - Setting up metrics: For this test we will be using the logging system and appropriate tags inside Android Studio to ensure metrics are met manually. For this integration test we need to be certain that serial numbers given by the datasets are properly passed, the actual data is properly passed, and the time taken for loading/passing each dataset is measured. To do this, serial numbers will be

logged in the graph visualization view, the 100th (40,000th for the large sets) data points will be logged in the graph visualization view, and the time of execution for both views will be logged. Comparisons will be made afterwards.
    - Executing the tests and monitoring the results: Inside the emulator - after setting up the environment - the tester will select the datasets and execute them in the order provided above. First the user will test the 1,000 entry dataset and monitor the logs for the given metrics, then the 50,000 entry dataset, then the 1,000 entry dataset with no header, and finally the user will select the 1,000 and 50,000 datasets, which will be merged during runtime.
    - Reporting and analyzing results: The serial numbers in the graph visualization view will be compared to the CSV file's serial number to see that they are equal, the selected data points will be compared in the graph visualization view against the CSV file's data points to see that they are equal, and the difference between times between views will be compared against a given value. If these values are all correct, then this integration test is good to be signed off.

- For this integration test, we will analyze the flow between log in view and the options view. We will look at if the user data is getting properly passed between the two views. It is important that the correct user information is getting passed as we don't want to distribute the incorrect information. Also important to make sure that data is actually being passed between the two views.
    - Preparing the test data: For the data we will use three different users. Each user will have a different email and password. There is no data that needs to be added to the emulator itself.
    - Setting up metrics: For the actual test, we will use the logging functionality inside of Android Studio. For this test, we need to make sure that the user id that logs in is the same user id that gets passed to the options view. When user information is inputted into the login form, we will log the user id associated with that user. Then when the user logs in, we will then log the user id that was logged in. The final log will happen when the user enters the options view. Here we will log the user id that is being read.
    - Executing the tests and monitoring the results: After everything is set up, inside of the emulator the tester will be able to start the test. First the tester will take login info for user 1, and input them into the login form. At this point, the tester should be able to see the user id displayed in the log. After clicking login, once again the tester will look at the log and see the user id displayed. Then the tester will move to the options page and look at the log and see the user id displayed. The tester will repeat these steps for the other two users.
    - Reporting and analyzing results: The three different user id's displayed during the test will be compared to the known user id for each of the three users. If the user

id's match, then we can confirm that the integration between the two views is working correctly. If there is any discrepancy, then we can further test to find the problem.




### 6.3.2 | Integration Test Results

1. For the first integration test, all values and tests passed:
    a. 1000 entry dataset:
        i. Serial Number: 95146101
        ii. 100th growth/soil moisture value: 22.0625
    b. 50000 entry dataset
        i. Serial Number: 92224514
        ii. 100th growth/soil moisture value: 19.0000
        iii. 40000th growth/soil moisture value: 18.3125
    c. 1000 entry dataset - no header:
        i. Serial Number: 92224514
        ii. 100th growth/soil moisture value: 17.3750
    d. Merged datasets

i. Serial Numbers: 92224514 and 95146101
ii. 100th growth/soil moisture value: 23.1875
iii. 40000th growth/soil moisture value: 18.2500

For the second integration test, we received expected results for all three emails and passwords. This indicates that information is properly persisted across the application and signals that communication between the fragments – particularly the options and home fragments – are working as expected.

## 6.3 | Usability Testing

Usability testing is the final piece of testing that a good piece of software should implement. This testing ensures that the actual end product meets the usability requirements set out before it during the requirements acquisition phase, and that the software is actually usable. If the application is extremely non-intuitive, then no matter how functional the application actually is, users will have no idea how to access that functionality. This is why usability testing is so helpful: it helps the development team see exactly what where users struggle with the interface of the application.

For our Android application, the usability testing that we will implement is user testing. We luckily have access to the four main users of our application, and so we are able to watch them perform several steps of the applications workflow, and precisely see where they falter or where the interface is unclear. We can then use this information to make our UI/UX better and more tuned to our clients' thought processes. The four users of our application are members of the NAU ECOSS, and prominent researchers in the environmental science field, and will be the main users of this application. They are Andrew Richardson, Mariah Carbone, George Koch, and Austin Simonpietri.

### 6.3.1 | Usability Testing Plan

- The workflows that we want to test with the users are the most critical pieces of our software. We only have a few key workflows and processes in our application, and so it is critical to us that these are as intuitive as possible. The following workflows are the workflows we will be testing with our users:
    - Data reading: downloading data is a key workflow for our Android application, and therefore needs to be thoroughly vetted before releasing to production. This workflow from a top-level point of view goes as follows: the user may select options to dial in the downloading, then the user plugs in a TMD adapter, approves permissions, and plugs the device in until it is downloaded. For each of these pieces of the workflow, the application must be very usable and intuitive, or else users are not capable of comfortably using the most important function of the

application. Therefore, our application will ensure that following tests and requirements are followed and met during the usability test:

- Step one: user selects download options:
  - The user should be able to clearly see where to navigate within the application to find the download options. The options themselves must be short, clear, and self-explanatory for a typical user of the application. The following are selections from worst to best for finding the options menu:
    - "I could not find where the button was located on the screen"
    - "I know to use the navigational menu, but the button was hard or indistinguishable from other buttons"
    - "I found where the button was, but the icon could better communicate"
    - "I was able to navigate to the options screen"
  - The following are selections from worst to best for selecting the options:
    - "I did not know what any of the options were for"
    - "I was able to intuit what a few of the options were for"
    - "Most, if not all, options were clear and well-described"
- Step two: downloading data
  - The user should be able to navigate back to the home view from the options menu, and that will be judged by the same navigation test above. The application should clearly state next steps for downloading, and accurately describe the permissions required for the app. The application should additionally present useful information when downloading data. The following selections from worst to best will determine results of this section's usability:
    - "I was unable to determine what to do next to download data"
    - "It was clear to me what the next step would be to download data"
  - The following selections from worst to best will determine results of this section's usability:
    - "No useful data was presented to me during the download"
    - "There was some data presented during the download, but not all of it was useful, or some was missing"
    - "All of the data presented during the download was useful, and none was missing"

- Viewing data: The visualization of data is the most critical module of our application. This process must be completed with extreme precision to ensure that users are obtaining data that is completely consistent with the original dendrometer readings. The graphs created from our charting library should be navigable, simple, and efficient. Efficiency of a graph relates to the productivity and convenience it supplies to a user. Users should easily obtain valuable data quickly proceeding the rendering of the graph. This process is very simple and only requires 3 ‘clicks’. The following steps will be performed for usability testing of the visualization of data in our application:
    - Precondition: User has CSV file stored on device in documents folder
    - Step one: user navigates to ‘file viewer’ tab
        - The user should be able to intuit which button in the navigation menu will take them to the aforementioned page; it is crucial the icon communicate this; therefore, from worst to best, the following statements outline the experience for finding the button:
            - “I could not find where the button was located on the screen”
            - “I know to use the navigational menu, but the button was hard or indistinguishable from other buttons”
            - “I found where the button was, but the icon could better communicate”
            - “I was able to navigate to the file selection screen”
    - Step two: User selects a csv file to visualize
        - The user should have the ability to easily select/deselect any of the files. The following statements list possible experiences for selecting/deselecting datasets:
            - “I could not find my csv files in the application”
            - “I am stuck on how to select a file”
            - “I am able to select a file, but cannot deselect it”
            - “Files are easily selected and deselected”
    - Step three: user finalizes selection, and device visualizes the data.
        - The user should be able to go straight to the button to finalize their selection and display the csv file; it is important the button be obvious and communicate its function for new and returning users; therefore, from worst to best, the following statements outline the experience for finalizing file selection:
            - “I could not find the button to finalize selection”

        - "I found the button to finalize my selection, but it took some time to figure out which button performed this function"
        - "I knew pretty much instantly which button to press to perform the visualization"
- Merging data: merging data sets should be a straightforward process which only requires the user to follow two or three steps with each step only taking a few seconds to a minute. By definition, this process is intuitive for the user, and allows them to get to visualizing and reviewing data as quickly as possible. Therefore, when testing, the user should be able to complete this step in 2 minutes - allowing for time to choose data sets; there user should be able to intuit how to select data sets to merge, and be able to find the button to visualize with ease. Therefore, the following steps will outline the tests which need to be met, as well as the degrees for success to communicate which aspects of the process must be improved.
    - Step one: user moves to the file selection page
        - The user should be able to intuit which button in the navigation menu will take them to the aforementioned page; it is crucial the icon communicate this; therefore, from worst to best, the following statements outline the experience for finding the button:
            - "I could not find where the button was located on the screen"
            - "I know to use the navigational menu, but the button was hard or indistinguishable from other buttons"
            - "I found where the button was, but the icon could better communicate"
            - "I was able to navigate to the file selection screen"
    - Step two: user selected a number of files to merge
        - The user should be able to begin the process of selecting files; it is very important the user understand which files will be merged and which files will not be merged; therefore, from worst to best, the following statements outline the experience for selecting files:
            - "I got stuck trying to select files to merge"
            - "I could select two or more files, but I need an indicator of which files will be merged"
            - "I was able to select and deselect the files I wanted and did not want to merge"
    - Step three: user finalizes selection, and the phone is able to visualize the data
        - The user should be able to go straight to the button to finalize their selection and merge the files; it is important the button be obvious

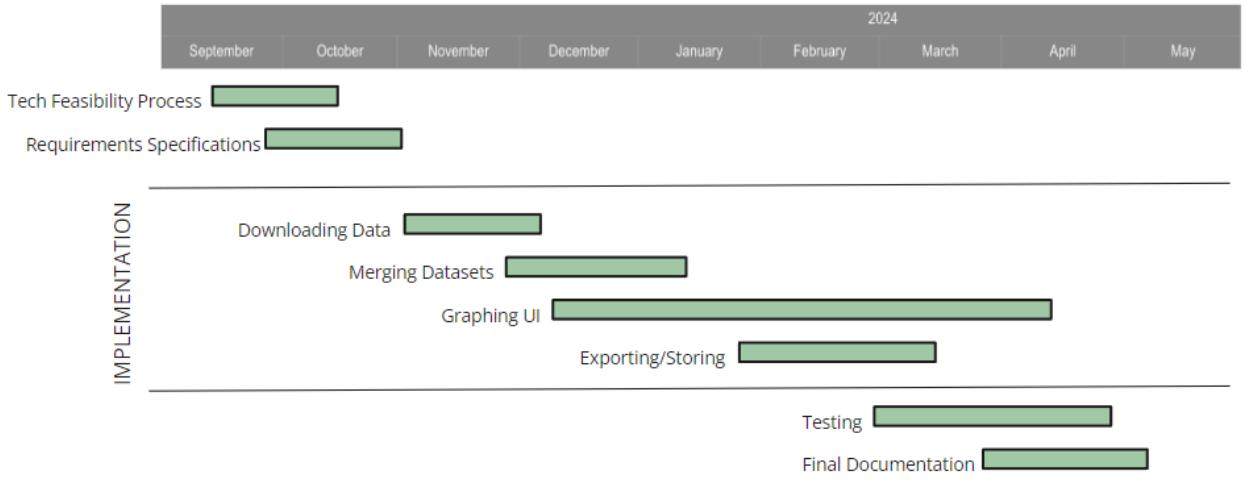
and communicate its function for new and returning users; therefore, from worst to best, the following statements outline the experience for finalizing file selection:
    - "I could not find the button to finalize and merge the selected files"
    - "I found the button to finalize and merge, but it took some time to figure out which button performed this function"
    - "I was able to find the button to finalize and merge the selected files, and visualize the data"
- Exporting data to the cloud: Being able to export data to the cloud is a key piece of the workflow. This allows users to share data easily across any device. This should be a very simple and quick process. The high-level overview of this would first be the user navigating the file viewer page. Then the user should be able to select any amount of files that they want to upload to the cloud. Once selected, the user should be able to press the upload button, allowing the files to be uploaded to the cloud. The following tests will be performed to complete the usability testing for exporting data to the cloud.
    - Step one: user navigates to file viewer page
        - The user should be able to tell what button on the bottom navigation will lead to the file viewer page. The following statements outline the experience in navigation to the file viewer page:
            - "I could not find where the button was located on the screen"
            - "I know to use the navigational menu, but the button was hard or indistinguishable from other buttons"
            - "I found where the button was, but the icon could better communicate"
            - "I was able to navigate to the file viewer page"
    - Step two: user selects file(s)
        - The user should be able to tell how to select files. They should also be aware that they are choosing files to be uploaded to the cloud. The following statements outline the experience in selecting files to upload:
            - "I got stuck trying to select files to upload"
            - "I could select one or more files, but I need an indicator of which files will be uploaded"
            - "I was able to select and deselect the files I wanted in order to upload them"
    - Step three: user presses upload button and files get uploaded

- The user should easily be able to recognize how to upload the selected files. The following statements outline the experience in selecting files to upload:
    - “I was not able to figure out how to upload selected files”
    - “I was able to find the upload to cloud button but could not upload the selected files”
    - “I was able to find the upload to cloud button and upload the selected files”

### 6.3.2 | Usability Test Results

All four usability tests were done at the same time, during one of our weekly meetings near the end of April with the four main users of our application: Andrew Richardson, Mariah Carbone, George Koch, and Austin Simonpietri. Because of the productive and extensive feedback from our wonderful clients, these tests were the most beneficial to our application’s growth. We received very positive feedback regarding the planned workflows above. Our clients tested our app further and provided us with suggestions and new features they deemed necessary. These included adding a delay after a successful download so it does not start redownloading right away. Along with this was adding a notification sound to indicate when the download was complete. Also part of the feedback were some bugs they were encountering while using the app. We were able to implement these and include them in our final version release.

# 7 Project Timeline



# 8 Future Work

As we release our final version of DendroDoggie, we are also passing on the ownership of the code to the TOMST company. Even with our application being as satisfactory as it is, our application still has lots of potential for optimization. If future work was to be pursued, we would recommend starting with the following additions.

## 8.1 | Dendrometer & TMS-4 Differential

The dendrometer captures growth in microMeters (uM) through a compression pin. It also has a single gauge to read in temperature. The TMS-4 on the other hand, is used for reading in soil moisture and includes 3 different temperature readings (above ground, at ground level, and around 4 in. below surface). Even with these significant differences in readings, both of these devices have the same CSV file output. This results in useless dummy values of temperature 2 & 3 values for data read from the dendrometer. Our application toggles these lines off by default to avoid any confusion when viewing dendrometer graphs.

Because the output from both of these devices are identical, our application handles and graphs data from both alike. Data is read in and graphed without any acknowledgment of which device is being read. With some cooperation from the TOMST company, some sort of identifier can be placed in the outputted CSV file to efficiently identify which device the data is coming from. This would open up the ability to have completely separate graph views each made specifically for a device. The dendrometer graph view would only contain T1 and Growth lines, while the TMS-4 contains soil moisture, temperature 1, 2, & 3 values. This will improve upon our UI so that there are never any unnecessary variables, overall leading to a cleaner legend and graphing view.

## 8.2 | Linear Regression

An addition of statistical analyses to the graphs would make the graphing interface even more useful to our users. A linear regression analysis is one statistical addition we think would serve great use. Users can use this to predict how any of the lines will continue in the near future, as well as analyze how a line strays from its calculated rate of change. The line of best fit can be generated using a simple algorithm that scans the arrays of growth/humidity and temperature values stored in each reading object. Because this feature applies to each and every line in the

graph, we recommend only allowing the linear regression line to be shown with its respective line. Adding another line to the already complex graph might make data harder to differentiate. This would entail having a small button next to each lines' checkbox, which when pressed toggles all lines off except for the line that is being calculated, and toggles on the visibility of that line's line of best fit.

### 8.3 | Metadata Addition

With our current software, a serial number and date of reading is what is used to identify each CSV file. We feel as though a reading could be determined easier with the addition of more information. This could include things such as tree description, location, height, device type, or anything that could improve upon differentiating devices. On the application side this prompt for user input would occur after downloading the data from the device. This metadata would be added to the header of the CSV file, and can be viewed from within our application.

# 9 Conclusion

---

Our clients are currently using an application to collect data from dendrometers. However, when using this application they experience a lot of problems. The current application is only available on a windows laptop, meaning they have to carry a big heavy laptop up the tree with them. This can pose a danger to the equipment and the person involved. Another major issue our clients face is that sharing data they collect can take up 8 hours. They currently would have to upload the data to google drive, then the person who wants to access the data would need to download it. This is a very inefficient system, especially when considering we are dealing with a large amount of data. This team was formed to create a mobile version of the current application. Along with creating a mobile application, the team is implementing multiple upgrades to certain features. Making the whole process of collecting data and sharing data much easier and safer.

Our solution involves developing an application which can be used on Android devices. This provides a portable and affordable platform which is easier to take up into trees because of the one-handed use. Our application is able to read, store, visualize, merge, and upload data to the cloud for the user.

As a team, we are very pleased with the final result of the project, and very pleased we had the opportunity to work with a group of like-minded peers on a piece of legitimately useful software. We are so glad we had the chance to develop our skills in many aspects of the full software lifecycle, including but not limited to: architecture, design, requirements acquisition, design reviews and general workflow, embedded systems development, Android development,

Java best practices, Google Firebase API and cloud development, and many more. We look forward to seeing how this application will be used by our clients to conduct more efficient research, and how their research will impact our world. Overall we can confidently say that we had a very successful and very informative experience through this Capstone project!

# 10 Glossary

---

If you use any terms that have special meaning (domain-specific terminology, for example), lay out the definitions here.

## Appendix A: Development Environment and Toolchain (absolutely required)

Remember how uninformed you were on practical mechanics of how to actually develop code in your chosen environment when you started? How long it took you to figure out your toolchain and development cycle? One of the key pieces of “organizational knowledge” for a project is exactly this: how should a new team member configure their development machine, and what is the process leading from code to production of a runnable product? This is what you want to explain in this appendix. Write it as a “how-to” for setting up your machine:

- **Hardware**: Start with an overview of your environment: what platform(s) did your team develop on (Linux, Mac, etc.)? Give a rough overview of the tech specs (processor, memory) of the machines. Comment on whether you feel there are any minimum hardware requirements for the effective development of your software, beyond “a decent machine”.
- **Toolchain**: Next introduce and discuss all the software tools you tools used: development environment/editor + plug-ins that were useful/critical, backend databases, other supportive tools/packages (e.g. package managers, etc.) that you installed to make life easier, or that are required for other pieces of the software chain to work right. For each one, name it, explain briefly what it does in general, and then why/how it was helpful/needed for this project.
- **Setup**: Now discuss how to actually set up your environment. This should be a step-by-step guide: install this, install that, place this in that directory, etc.

You don’t have to discuss in detail how to install individual packages (instructions for that are presumably on the site for that package), but do point out any special settings or configurations that should be made in installing it that are relevant to this project. Just imagine yourself walking a new team member through setting up their machine.

- **Production Cycle**: At this point, your instructions should have allowed a newbie to completely set up his/her machine for action. Now let’s get down to work: explain the production cycle, i.e., walk through how the edit-compile-deploy process works. It might be helpful here to focus it around a specific example: explain how one might change, for instance, the text that appears in some obvious dialogue in the application...then what you’d do to build and produce a “new version” of your app. To avoid missing small but critical steps, it is useful to just do a little edit yourself and pay attention and write down each step in the process...

In sum, reading this appendix and following the instructions should literally allow a new team member to effectively edit your code and push out a new product version. With that, they are off and running!

**Other appendices (optional)**

Include any other documents that you feel make your design specification easier to understand but are not central to the project's description.