Intro to Software Engineering
ECSE 321

Kevin Chuong, 260742781
Jeremy Davis, 260744431
Oscar Décéus, 260744646
Elias Al Homsi, 260797449

DELIVERABLE #2

Presented to Mr. Daniel Varro
Faculty of Engineering

McGill University
December 11, 2018

Table of Contents

1.0 Architecture

1.1 Description of subsystems and interfaces

MVC Architecture:

Model subsystem

The model subsystem functions as the domain knowledge of the system. Holistically, it represents the data storage (persistence) of the entire application. It provides business data for both the view and controller subsystems to work with. But most importantly, the model subsystem does not need to know what happens in the other 2 subsystems to function and is therefore independent.

View subsystem

The view subsystem functions as the display of domain objects to users. It is a visual representation of data (android and web frontend) and serves as the medium of interaction between the user and the controller subsystem. It can receive data from the model under the form of data transfer objects from the controller. The view subsystem depends on the controller subsystem and the model subsystem.

Controller subsystem

The controller subsystem manages interactions with the model subsystem and updates data changes of the former to the view subsystem using data transfer objects. It deals with the business logic (ex. Sign up -> Updates model with new user) and enforces business rules (ex. Wrong password -> Updates view with error message). The controller subsystem depends on the model subsystem. The Controller subsystem includes the capability of hashing and salting a password for safe storage. Also, it generates tokens to keep track of logged in users.

Client-Server Architecture:

Clients

Clients will have access to the system through the frontend. More specifically, Client 1 (Scientists and Urban Foresters) will have access to the web frontend and Client 2 will only have access to the Android frontend. They will be able to send and receive data to the appropriate services provided by the server via HTTP calls using the REST API. However, there are some restrictions concerning the services that each client can use. Client 1 will be able to send REST API calls to all the services while Client 2 will only be able to use the Tree Management service and the User Management service.

Services

Four different services are implemented within the backend of the system: Tree Management, User Management, Sustainability Calculator and Forecasting Generator. Each of them deals with different types of requests and manages different types of data. Service 1 (Tree Management) deals with the different queries related to trees (e.g., mark, cut down, plant, list) and send back (if needed) the data transfer objects to the appropriate client. Service 2 (User Management) deals with the queries related to users, especially with their information… Service 3 (Sustainability Calculator) deals with the queries related to the sustainability attributes. Finally, Service 4 deals with the what-if scenarios.

MVVM Architecture:

This architecture was chosen for the website frontend because it fits best the required specifications.

Model

The model subsystem contains the business logic and data for the system. The model is saved with the persistence layer of Xstream which saves it in the format of xml files. The model is transferred into view-model format on the front-end in order to make it possible to interact with the view.

View Model

The view model is responsible for converting the different data objects from the model in a way that objects are easily managed and presented to the users. The model view interacts with the view in order to update the information presented to the user.

View

The view is the website itself coded in html, css and vue.js. The view job here is to show the information and relay it back to the user in order to show the internal state of the system.

1.2 Evaluation of architectural styles

Model-View-Controller Architecture

The architecture of MVC was used because it fits best the requirement. Since the application itself specifies a backend part running on a server and a front end part running on different server (possibly the same). Also, the unified backend would serve both the android and the website frontends. MVC is simple elegant solution to the problem presented and it allows us to focus on the source of the problem (Service, Controller, View, Model) and fix it. Working on a modular structure such as MVC helped the team perform well and works best with the agile development method.

Layered Architecture

The layered architecture was used to wrap the business logic in the service module with the restful controller. Instead of making one controller, the project used separation of concerns and layered architecture. The Restcontroller comes on top of the service and provides HTTP requests for the service layer. It helped the team to focus on the business logic and the communication problems separately.

Repository Architecture

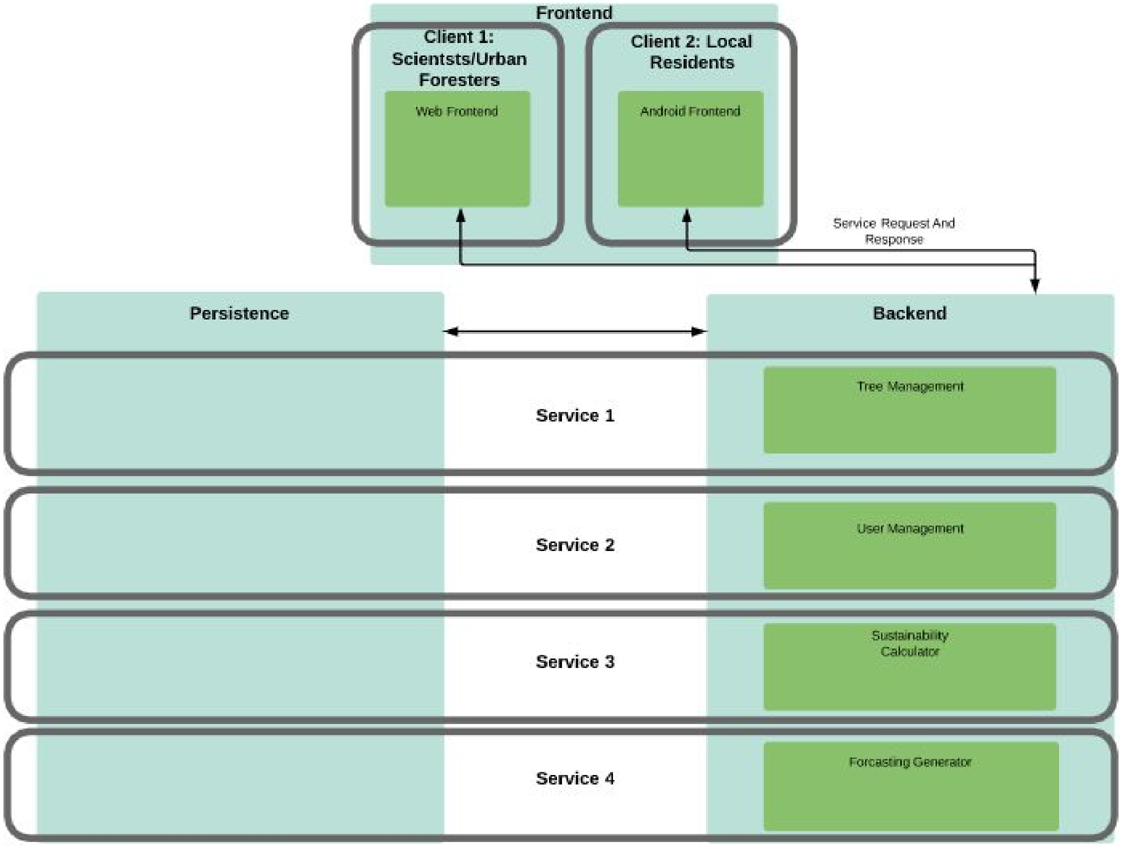
Repository architecture is common when developing an IDE or a revision system. The application given does not fit this purpose or the general architecture. It also presents one point of failure which damages the requirements of reliability and availability. Repository Architecture simply does not fit the project description.

Client-Server Architecture

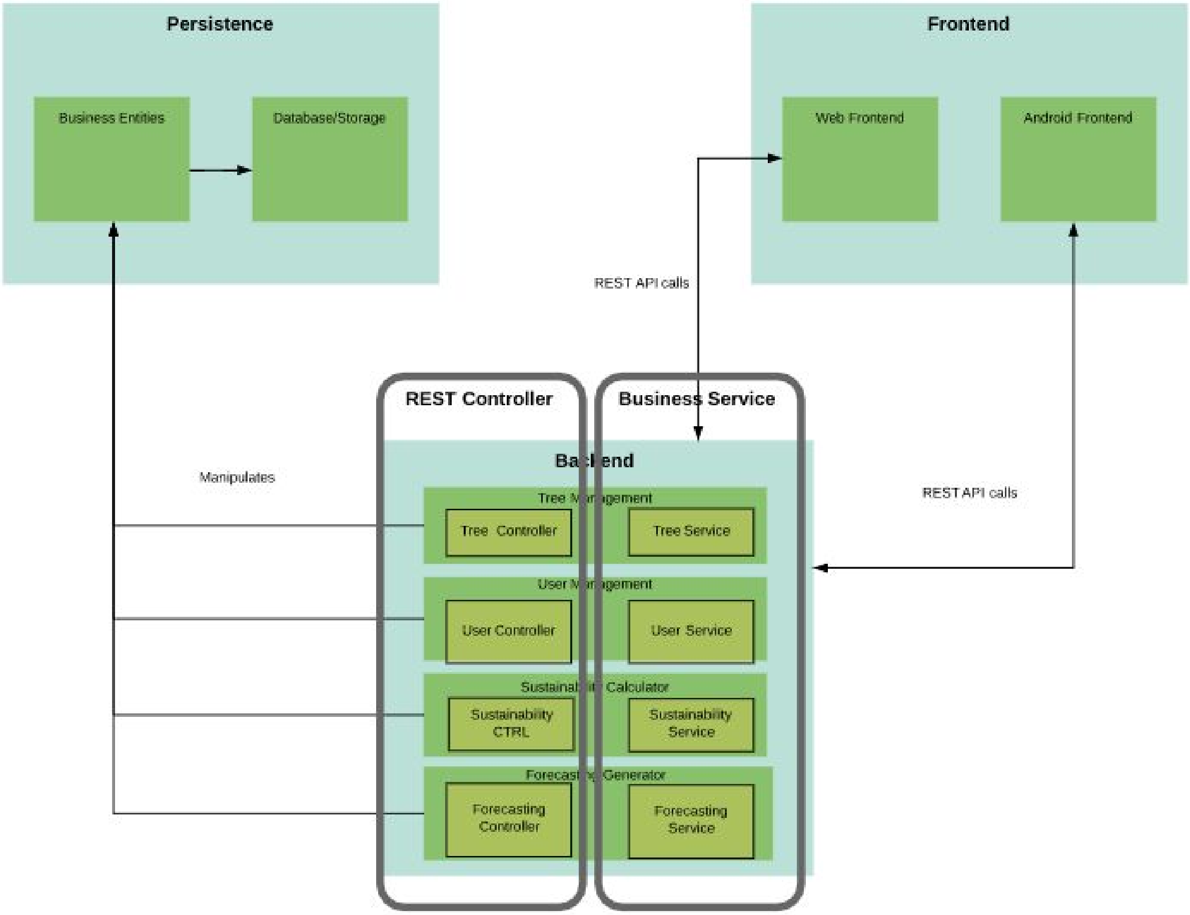
Many clients are going to use the services provided by the project. Clients would connect using their phones and browsers to make modification to the model and adding information. Client-Server Architecture fits the service description of users connecting remotely and requesting information and services. The thin-client programs being represented as the website and the android front-end. The server being the backend installed on a seperate server (preferably) and providing services to the clients.

1.3 Block Diagram(s)

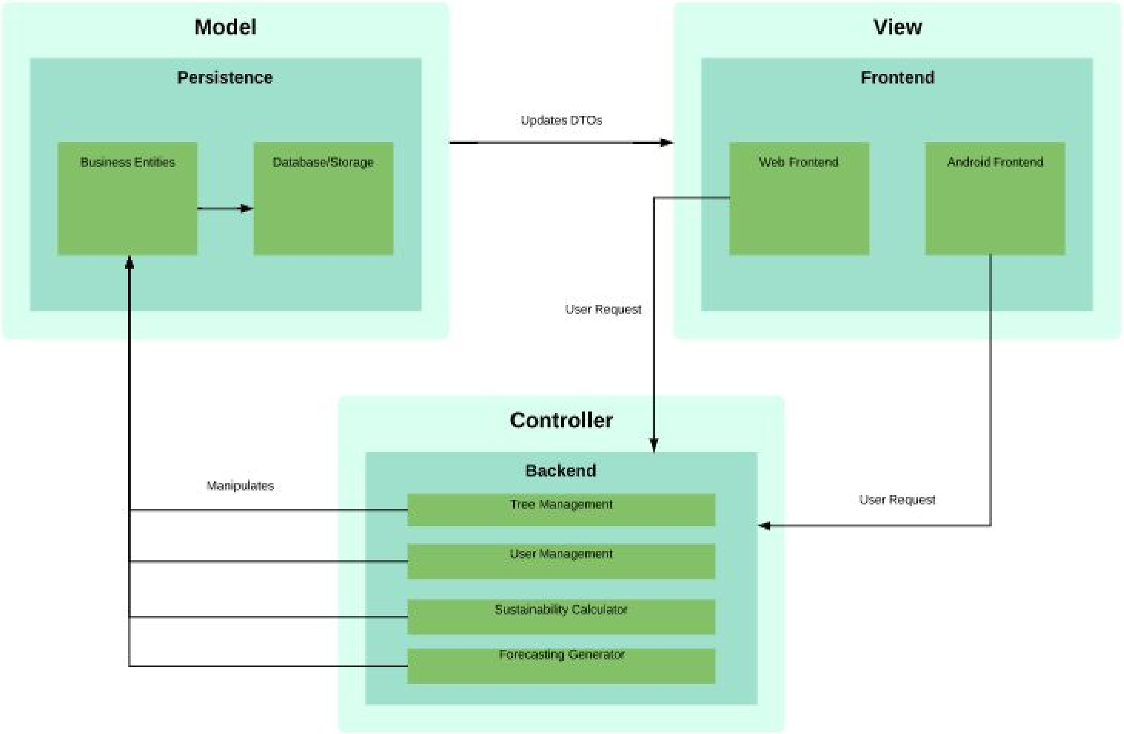
Client-Server Architecture



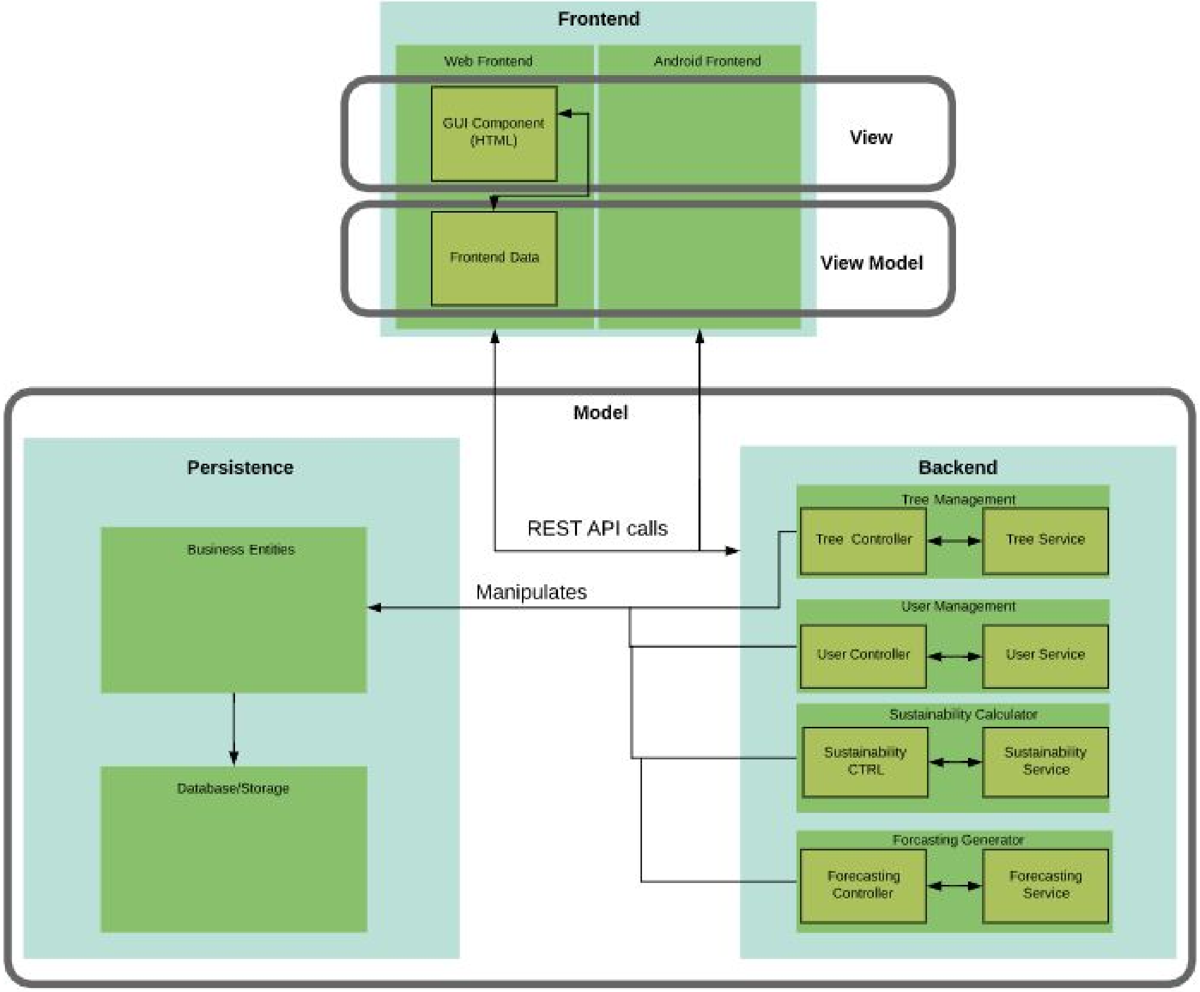
Layered Architecture



Model-View-Controller (MVC) Architecture



Model-View-ViewModel Architecture



2.0 Detailed Design

2.1 Entity classes + DTOs (description + operations)

The full list of Entity classes are:

1. Tree: The tree entity with its attributes.
2. Resident and subtypes are Environmental Scientist and Municipal Arborists
3. Location: longitude and latitude
4. TreePLE System: The overall system that links everything else.
5. Transaction: links a resident with a tree he/she trying to modify. Everytime a tree is marked as cutdown or diseased a transaction is registered into the system.
6. Municipality: The muncipality that will be linked to the tree.

Since those entities contains information that the system does not necessary want to show to users such as User's hashed passwords and other private information. DTOs are created in order to transfer the information to the front-ends.

A TreeDTO from the entity Tree.
A ResidentDTO from the entity Resident.
A MuncipalityDTOfrom the entity Municipality.
A LocationDTO from the entity Location.
A TransactionDTO from the entity Transaction.

2.2 Controller classes (description + operations)

TreePLEService

The main controller of the entire system, takes care of all backend business logic in the following list of operations:

1. genToken: Generates a token (cookie) to keep track of logged in users
2. checkPassword: Checks password of corresponding user
3. CreateTree: Creates a new Tree object and saves to persistence
4. CreateResident Creates a new user of the system. The type specifies which type of users is requested to be created
5. FindAllTrees: List all trees as an array of JSON objects.
6. FindAllMunicipalities: Lists all municipalities available.

7. FindAllResidents: Added for debugging purposes. Lists the residents currently registered in the system.
8. FindAllTransactions: Changing a tree status requries a transaction. A transaction is a simple entity that registeres which resident modified which tree and when. This method lists all transactions.
9. createMunicipality: creates a new entity.

The controller also have a list of active tokens (cookies) that logged in users are currently owning. A token work by generating a random string of characters and giving it back to the user once logged in. The user would submit the token in future transaction in order to verify identity. Once a token has not been used over a threshold of time it becomes stale and the system would not accept it. It is true that this might not be the best way to manage security. However, it is good enough for the current implementation (non banking application).

2.3 Boundary classes (description + operations)

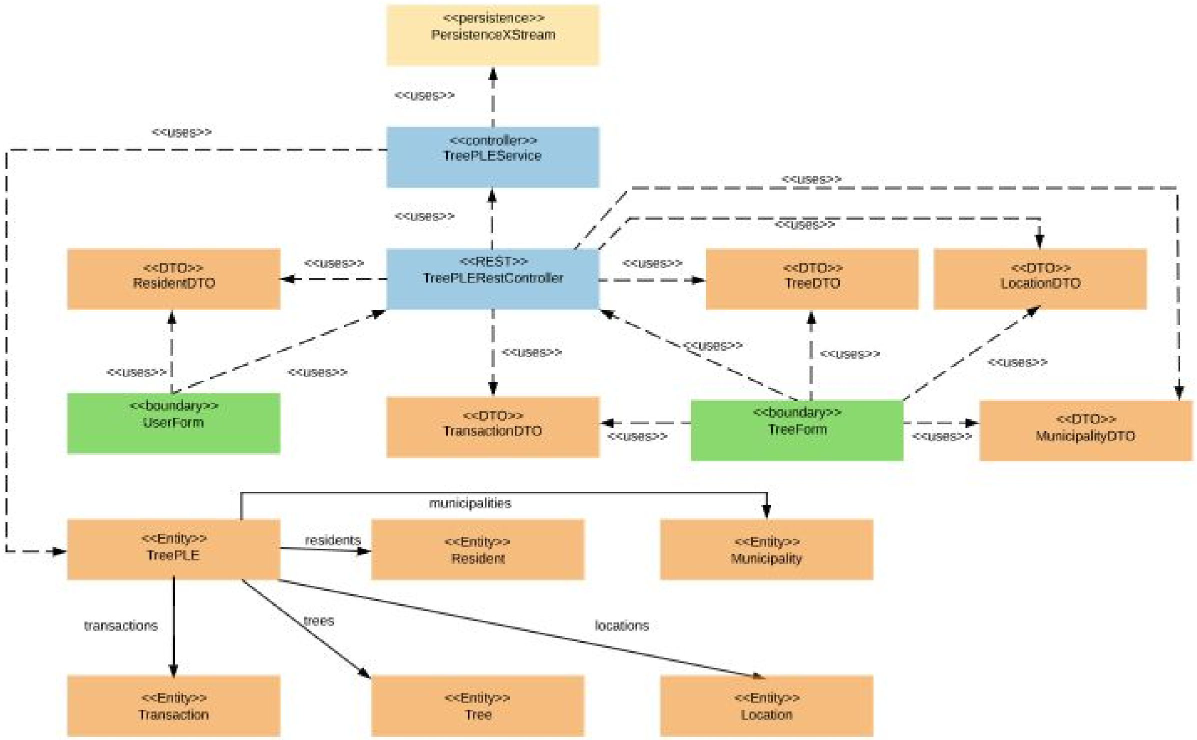
TreeForm

Manages the exchange of information of trees between a user and the system. Includes the information about which tree, its location, its municipality and the appropriate transaction.

UserForm

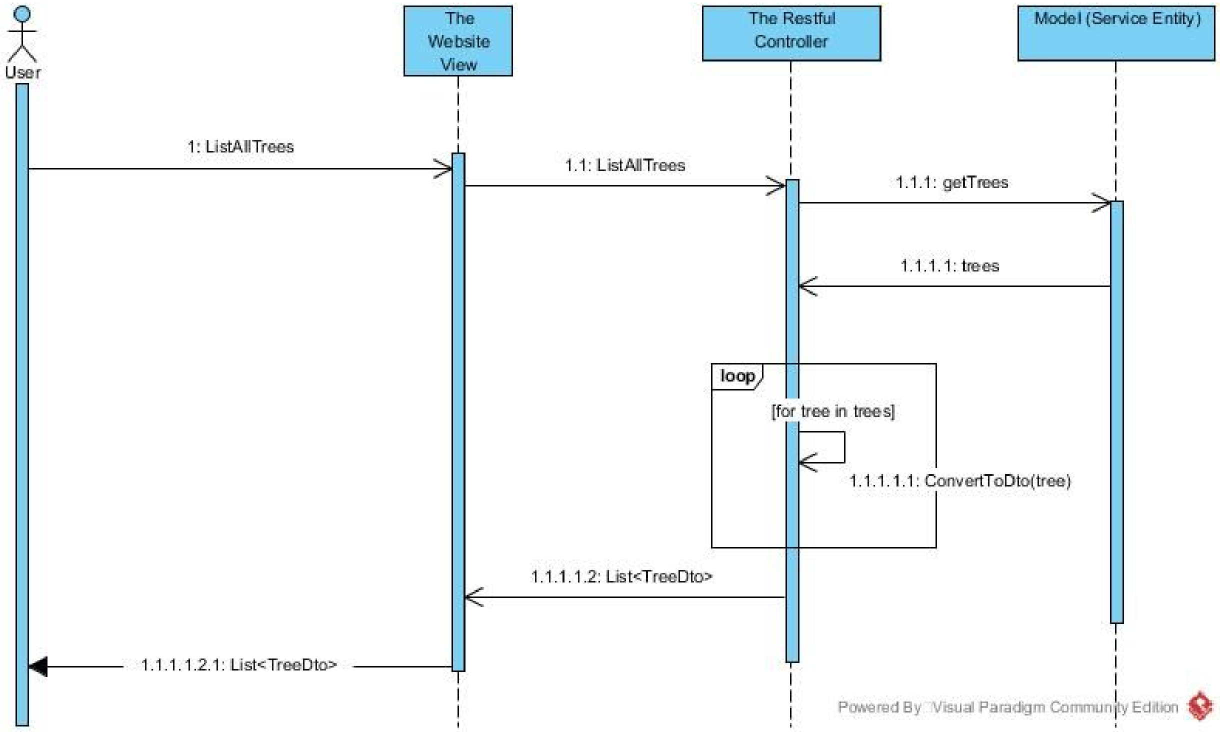
Manages the exchange of user's information between a user and the system. Includes the information of a user's name and password.

2.4 Class Diagram(s)



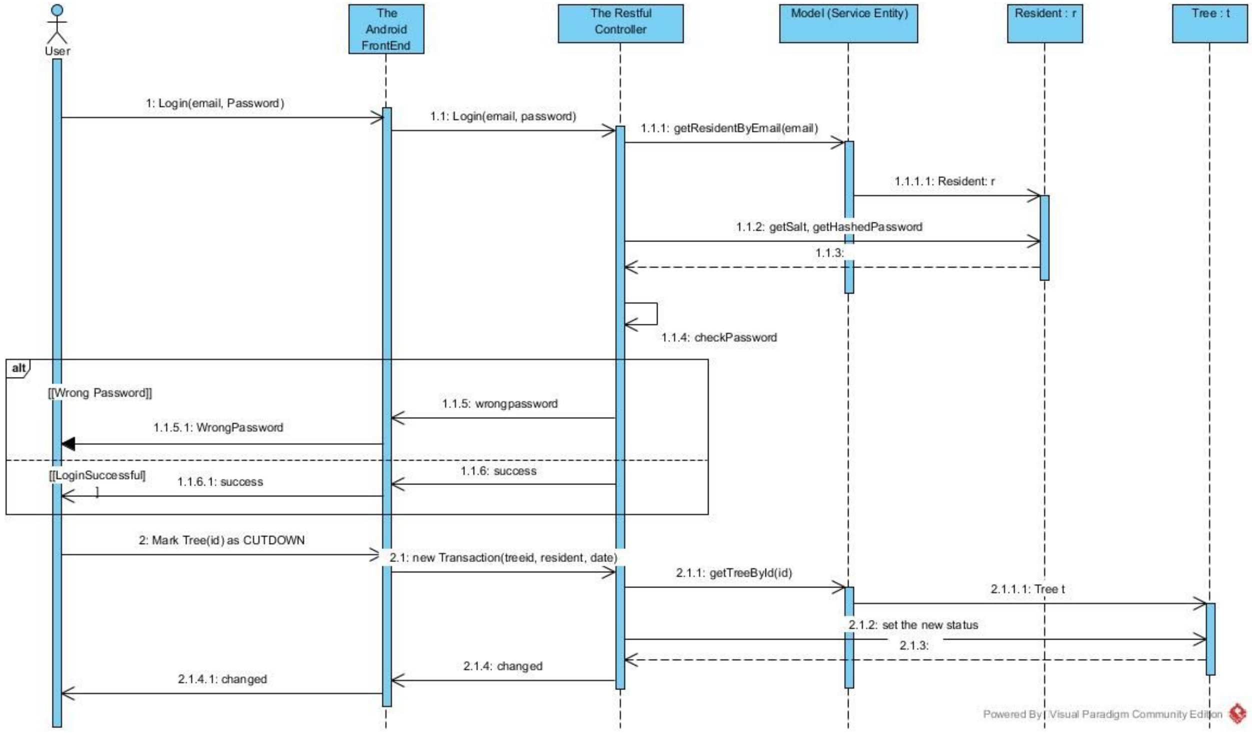
2.5 Sequence Diagrams

Sequence Diagram of listing all trees:



The Sequence Diagram above clearly explains the flow of events when the user first hits the page. An implicit message for listing all trees is generated and therefore the view (Vue.js) relay back the message to the backend using the AXIOS API. The AXIOS API provide the asynchronous ability of making post/get requests while not holding resources such as the page. The Controller then make a request to the model in order to get all tree objects. One by one, the controller transform those objects into DTOs and relay them back to the view and hence back to the user.

Marking a Tree as planted or cutdown:



In the system we are requiring the user to login in order to keep track of who changed the tree status and at what time. Also, some modifications are only available to scientists and other special users. This way the system could keep who is doing which modifications and if they have enough privileges. This feature is yet to be implemented in future iterations.