Industrial Project

Tree Adoption Application for MoT

Final Documentation

Computer Science Department City College,
International Faculty of University of Sheffield

Tron Baraku
Chase Burton Taylor
Marino Osmanllari
Yusuf Demirhan

February, 2023

Table of Contents

1. Final Documentation

The final document's objectives are to guide upcoming developers for ongoing development as well as to highlight the development efforts of the third team in charge of furthering the MoT Trees project. The third team working on this project consists of Tron Baraku, Chase Burton Taylor, Marino Osmanllari, and Yusuf Demirhan Kerem. The project's requirements, analyses, designs, implementations, and testing will all be covered in the document so that readers can get a complete picture of how the project is coming along. Appropriate instructions will be given for the installation, execution, and deployment of all project artifacts to make future development easier.

2. Requirements

Both the initial requirements left by the previous development team and the requirements added after meeting with the client will be highlighted to help understand how the project is progressing. The status of each requirement's completion will be shown in a table to showcase the amount of work left to be completed.

Requirement	Status
Renew SSL Certificates	Done
Redeploy new Google Maps API	Done
Redesign Application for Mobile users	Done
Add Tree Filtering	Done
Add Zip Code Map Search	Done
Add Zoom to Tree Button	Done
Add Water All Trees Button	Done
Prevent multiple users adopting the same tree	Done
Admin responds to requests	
Report sorting, searching and pagination	Started
Resolved reports deleted or archived for a time period	Started
Email notifications for password resets, email changes, etc	Done
Add email verification (also when changing the email)	Done
Fix password resetting cache problems	

Map local caching	Done
Limit user actions (rate limiting)	Done
Add node clustering	Done
Track the weather and implement it into a watering notifications algorithm	
Add watering requirements per tree type	
Distinguish age visually between trees on the map	Done
Admin Dashboard (Admin Panel)	Done
Admin Statistics Page (Application Statistics)	
Create an admin hierarchy	
Comment on trees	
Users upload a photo when reporting a tree problem	
Limit the maximum number of tree adoptions	Done
View other people's adopted trees	Started
Implementing multi-language functionality	Started
Add a custom name to trees	Done
Restrict map to map of Thessaloniki	Done
Add global filter option	Done
Users upload a photo to customize their tree	
Gamification (Points, Prizes)	
Introduce user level system where maximum number of adoptable trees is linked to user level	
Competition/Leaderboards	
Share trees between people	
Make the adoption of a tree satisfying/ceremonial	Started

Requirement breakdown

Very High Priority

- **Renew SSL Certificates.** Initially users were unable to login and register on the deployed project. This was due to the expiration of SSL Certificates. Resolving this issue has a very high priority as it is a core requirement.

- **Redeploy new Google Maps API Key.** Creating and deploying a new Google API key was critical as the previous one had expired. The map functionality would not load without a valid Google API key.

- **Redesign Application View for Mobile users.** This requirement was decided at an internal group meeting. Creating a better user experience for mobile users was the top priority assigned to our group. Further details and design requirements are discussed in section 3.1.

High Priority

- **Add Tree Filtering.** Mobile users are able to filter the tree map based on the ownership status of the trees. Currently, three options are provided: Show All Trees, Show Only My Trees, and Show Only Available Trees.

- **Add Zip Code Map Search.** Mobile users are able to search for a specific area on the map based on the zip code they provide. The current solution finds a tree based on the provided zip code and reecenters the map to the coordinates of that tree.

- **Add Zoom To Tree Button**. Mobile users are able to easily find their adopted trees on the map page by clicking on a button that zooms to their trees. If the user has more than one tree, subsequent button clicks iterate and zoom over the adopted trees (first click goes to first tree, second click goes to second tree and so on).

- **Add Water All Trees Button.** Mobile users are able to easily mark all their adopted trees as watered with the click of a single button. This button is displayed in the *MyTrees* page if the user has more than one tree.

- **Prevent multiple users adopting the same tree.** Users of the application should not be able to adopt the same tree if they click to adopt a specific tree at the same time. A check in the backend has been added to prevent this from happening.

- **Admin responds to requests.** Admins right now can manage reports between each other and resolve them, however they cannot ask users for additional information or talk to them in any way. Currently, the only communication channel between administrators and users is through the contact us page. This functionality is of high priority as gathering information from users is a core feature.

- **The report sorting, searching and pagination are not done in the reports page.** The reports appear clustered as boxes in the reports page. This feature would help administrators to easily manage and search for reports.

- **Resolved reports deleted or archived for a time period.** There is no way for a report to be put back as unresolved or to be outright deleted, although technically possible in the database. This is necessary functionality in order for admins to easily determine the amount of new problems. Separating resolved reports from active reports would help with the clustering in the reports page.

- **Email notifications for password resets, email changes, etc.** This was mostly implemented by the previous team but inoperable. This is a high priority issue as it deals with account security.

- **Add email verification (also when changing the email).** Users must confirm their email when creating an account or changing their email. This is high priority both for ensuring the user will receive the notifications and to prevent bots from creating unlimited accounts.

- **Password resetting cache problems.** In order to reset the password in production due to the service workers caching the responses, users need to refresh their cache. This is not intended and needs to be fixed. This is a high priority issue as it deals with account security.

- **Map local caching.** The map has been optimized to reduce the map load time. The map is a core requirement which heavily impacts the usability of the application. Static data regarding the trees is cached after the user loads the map for the first time. The size of the tree data has also been reduced. Fetching from the cache significantly reduces the map loading time as only data regarding the users (owners) needs to be fetched from the server.

- **Limit user actions (rate limiting).** Providing limits for the number of times users can perform certain actions: resetting their password, reporting an issue, adopting / removing one or more trees, renaming their tree and similar actions have a high priority because they deal with the security and reliability of the application. In addition, as requests made to the Google Maps API come at a cost, this action should be controlled to prevent misuse by users.

- **Add node clustering.** The performance of the server is enhanced through node clustering, which is an approach to run multiple instances of Node through child processes.

Medium Priority

- **Track the weather and implement it into a watering notifications algorithm.** Users can be reminded through a notification about when they need to water their trees. This requires an external API to get the data. The algorithm would have to run on the server (backend) and manage issuing these notifications. This is medium priority because while it would provide utility, it is not a core function.

- **Add watering requirements per tree type.** This requires a knowledge base for each tree. A list of all trees has been sent to the client so that they can provide this information but there has been no response yet.

- **Distinguish age visually between trees on the map.** This functionality was implemented by the first group and misidentified as a requirement by the second group so the status has been changed to ‘Done’.

- **Admin Dashboard Page.** This page would act as a central hub and link to other admin pages such as user reports and account management pages. It has been created for mobile users.

- **Admin Statistic Page.** This page would display statistics about the application (number of users, number of adopted trees, number of trees watered, etc).

- **Create an admin hierarchy.** Currently, admins can promote other users to admins and they can also remove other admins which can be dangerous. An admin hierarchy would specify the permissions that admins have.

- **Comment on trees.** Trees would take the form of a post which has comments. Might also have to consider a reply system. This is medium priority because it will improve user engagement but require considerable development time to ensure moderation features work as intended.

- **Users upload a photo when reporting a tree problem.** Right now, users can insert a title and a description for the report. It would be more descriptive if the user could also upload a photo in order to show the specific problem. This is medium priority as in most cases the standard category selection and description will be sufficient.

- **Limit the maximum number of tree adoptions.** This was identified as an important requirement before launching the application. Having a limit prevents application misuse and ensures that the trees are distributed among users. The solution has been developed and implemented. The current limit is 3 trees per user.

- **View other people's adopted trees.** It is expected that when users click on trees on the map, they can view who owns these trees. Alternatively, it can be added on top of the map itself, although this might prove more challenging. This is considered medium priority as it is not a core feature and will require extensive consideration.

- **Implementing multi-language functionality.** Errors in the backend are currently thrown from the repositories to the controllers. This works, but the issue is that these exceptions are sent to the frontend and this poses a threat to localization as this package is only available in the frontend. A solution to this would be to receive an error from the backend, and it serves as a key to the localization package so that it can be translated according to the users' language. This is considered medium priority as the vast majority of users are expected to have Greek as their first language. An effort to facilitate this functionality has been made as a "language.js" file which contains the labels in Greek, has been created.

Low Priority

- **Add a custom name to trees.** The frontend and backend solution has been developed and implemented. Users can add custom nicknames to their trees.

- **Restrict map to map of Thessaloniki.** The map has been restricted to prevent users from panning or zooming outside the bounds of Thessaloniki.
- **Users upload a photo to customize their tree.** Right now, whenever the user views their trees, there is a default picture that doesn't represent the current tree. There can be an option where the user can upload a photo of their adopted trees. This has to be built from the database up, it has not been accounted for so far. This is considered low priority as this is not a core functionality and moderation features will be required.
- **Gamification (Points, Prizes).** Whenever actions are performed, reward users with points. Such actions could be tree watering, making a tree report or adopting a tree. Once they reach a certain level of points, they can shop for prizes in a shop. These prizes may be physical or digital. This is low priority as it adds little functionality and would require extensive planning to ensure it could not be exploited.
- **Introduce user level system where maximum number of adoptable trees is linked to user level.** Currently the maximum amount of trees that a user can adopt is 3. In the future, a user level system could provide benefits based on the level of the user such as unlocking more trees.
- **Competition/Leaderboards.** These are meant to show the users with the most points, just like a game leaderboard. This is a low priority issue because it is not a core functionality and exists primarily to increase user retention.
- **Share trees between people.** Currently, once a tree is adopted, it can no longer be adopted. The database does support multiple tree adoptions, but neither the backend nor frontend account for this so far. This is low priority because it is not a core feature and until the number of users exceeds the number of trees in Thessaloniki it will likely remain as such.
- **Make the adoption of a tree satisfying/ceremonial.** Whenever a user clicks the adopt button, present him with something nice, maybe include a GIF, a sound, an animation, emojis, whatever makes the process exciting and playful. This is a low priority as it serves primarily to encourage additional tree adoption.

3. Analysis

This section includes all analyses done at various points before appropriate designs or implementations were produced. While there are numerous features that are incomplete or unattempted, only a few are crucial for deployment. Therefore, we have prioritized the tasks in accordance with necessary user actions. Requirements that are high priority should be the focus to ensure that the basic features are robust and reliable. Medium priority corresponds to requirements that further improve the application through increased functionality. Low priority requirements are quality of life improvements and other non-essential functionality.

As we have worked to implement and improve high priority features, undocumented issues presented themselves. These included:

- Problems with dependencies - when setting up the project locally, dependency issues were encountered and the command “npm install –force” had to be used on the frontend folder. To fix the problem a deprecated dependency had to be removed.
- Users unable to login and register due to expired SSL certificates on the deployed project which is being hosted in cPanel.
- Google Maps API key had expired and the Tree Map was not functional.
- The Map containing the trees required a substantial wait time to load.

A meeting with the client was scheduled and completed at the early stages of the project. Raised questions and recommendations were communicated with the client to redirect the path of functionalities. The meeting addressed some issues of high importance that need to be prioritized. After exchanging ideas for future features with the clients and the lecturer, they briefly proposed some interesting touchpoints, such as:

- Create admin dashboard page for showing application statistics
- Display water requirements for each tree in the “MyTrees” page
- Introduce user levels and perks associated with them such as being able to adopt more than three trees
- Create admin hierarchy to regulate cases where admins can remove other admins

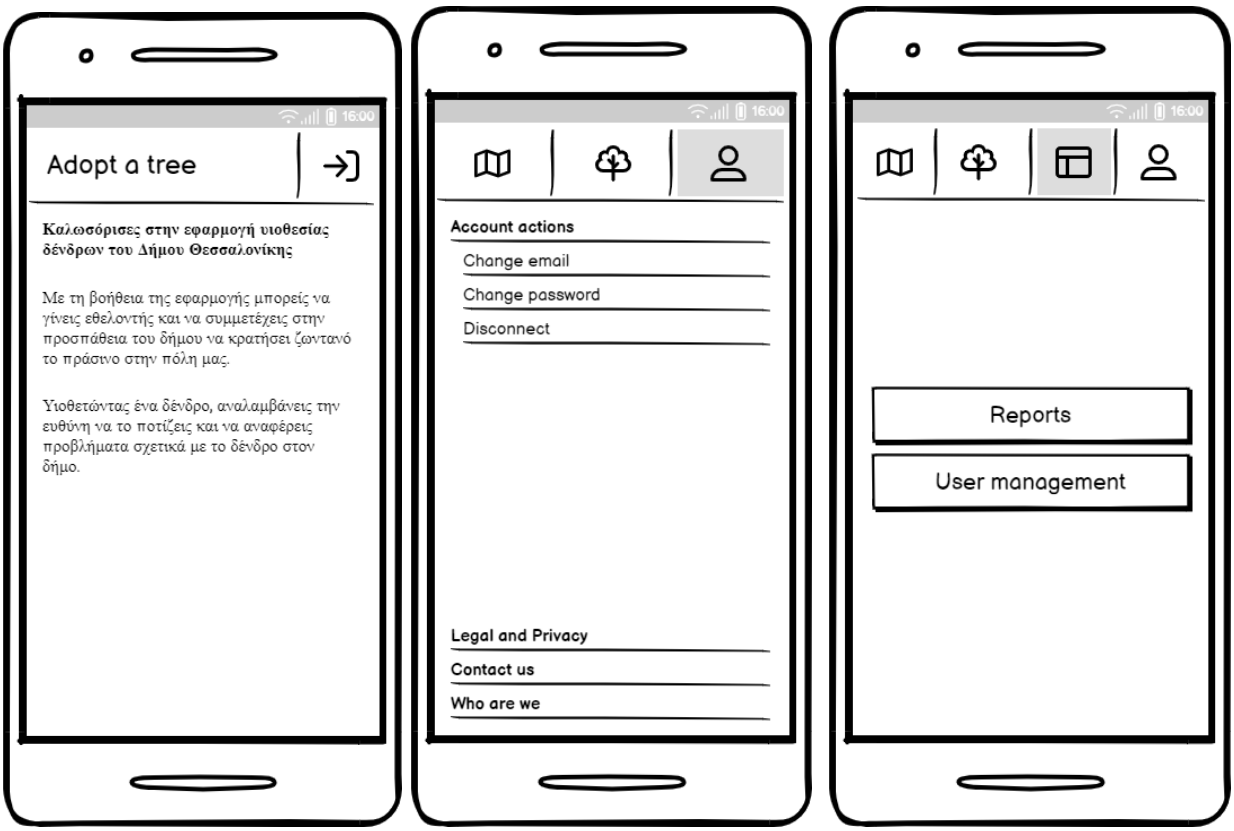
At this time, administrators cannot view statistical data about the application in a convenient way. Information regarding the number of users, number of adopted trees, number of trees that have been watered is not provided within the application. Access to this data is only possible by manually querying the database. This approach is not optimal and should be addressed in the future. Another issue stems from the fact that administrators have the ability to remove other admins. This implementation is dangerous as it can be misused in the case that a user is accidentally promoted to administrator and removes all other admins. On the user side of the application, once a user adopts a tree they are unable to view the watering requirements for that tree. This missing information is problematic because a user would have to know or research the watering requirements of their trees. This could lead to confusion among users and mistakes made on the amount of watering that a tree receives. As a result, the aforementioned points were added to the list of requirements.

3.1 Mobile Application Redesign

During the development of the application, an internal meeting was held where the user experience and the overall design of the application was discussed. It was decided that improvements should be made towards changing the way mobile users interact with the application. Assuming that most of the users will indeed be mobile users, the changes that were discussed rank highly in the list of priorities. Mockup screens were built using Balsamiq Wireframes and they have been included below.

3.1.1 UI Mockups

Mockups for user interfaces are straightforward visual representations of a particular requirement. They offer a visual representation of how a particular interface might appear but lack many specifics like fonts, colors, and other intricate stylistic components. The mockups that were made in order to improve the mobile application are displayed and described in this section.

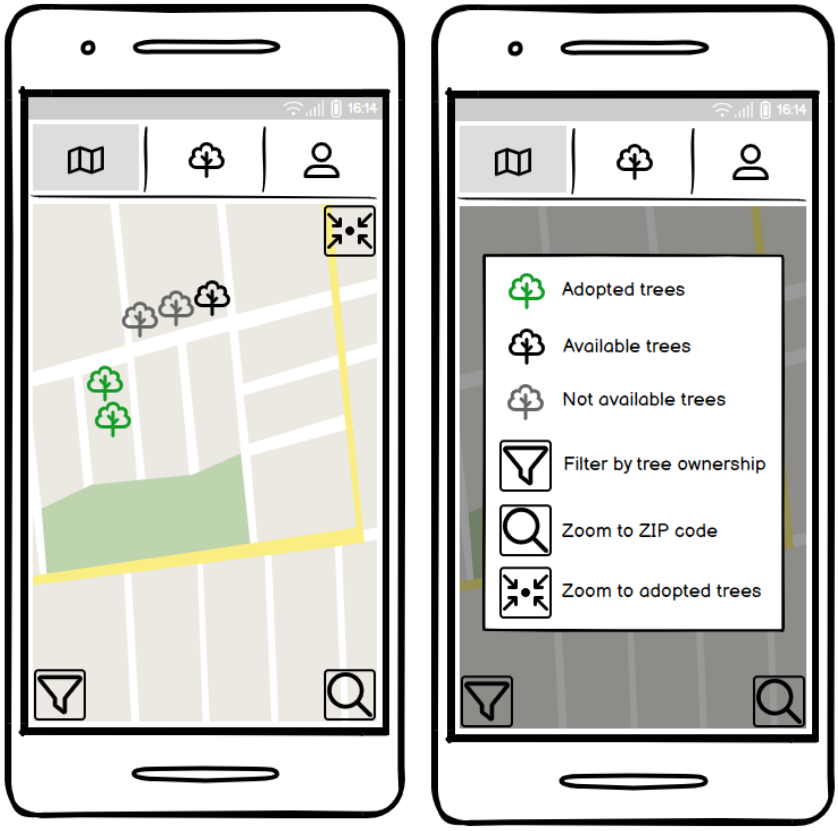


Landing Page | User Settings | Admin Page

The landing and user settings pages have been simplified. The existing images on the landing page were deemed unnecessary and inaccurate. The navigation bar is going to be redesigned to show icons instead of the existing sidebar. These icons will redirect the user to three different pages: “My Trees”, “Map” and “User Settings”. Admins will have an extra icon which will give them access to admin actions such as viewing reports and managing users.

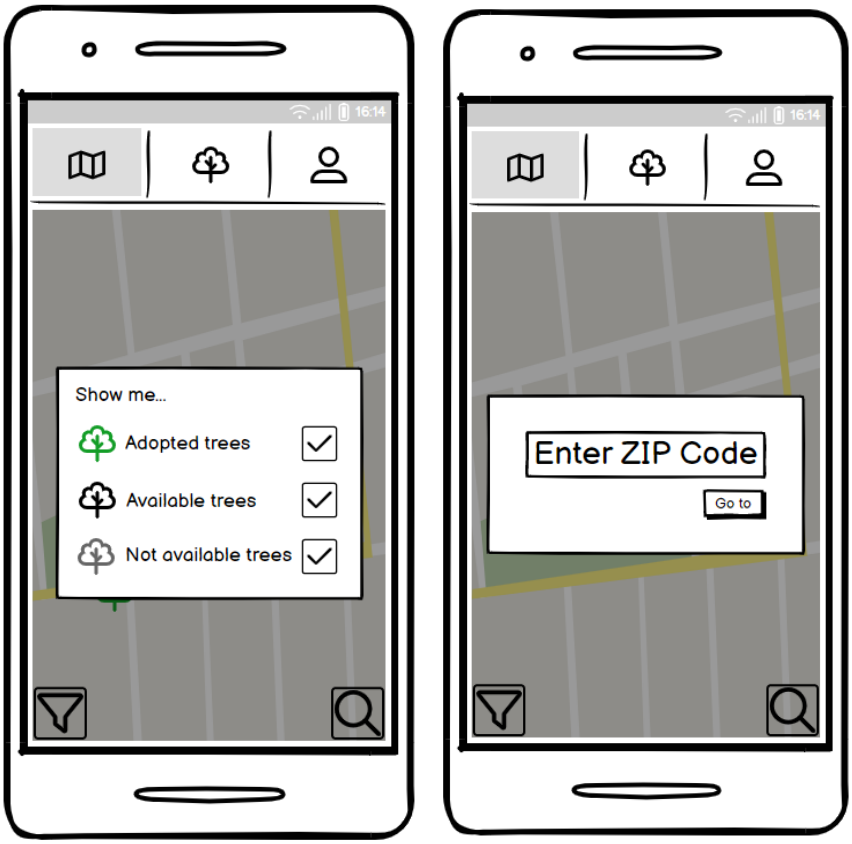
“My Trees” Page

The most significant changes can be observed on the “My Trees” page. The way users view and interact with their trees has been rethought and redesigned. Upon visiting this page for the first time, the users have the option to click on the “Adopt a Tree” button which will redirect them to the map where they will be able to adopt trees. This button is displayed as long as the users have not reached the maximum amount of trees adopted. Users that have adopted trees are shown the address of the tree and have the option to water that specific tree or view the tree on the map. The latter action will redirect the users to the map page and zoom in on their specific tree. In order to facilitate the process of flagging trees as watered, a “Water All Trees” button has been added. By simply pressing this button, all the trees adopted by the user will be marked as watered. Lastly, recently watered trees will be differentiated from unwatered trees by applying a different color to the tree card.



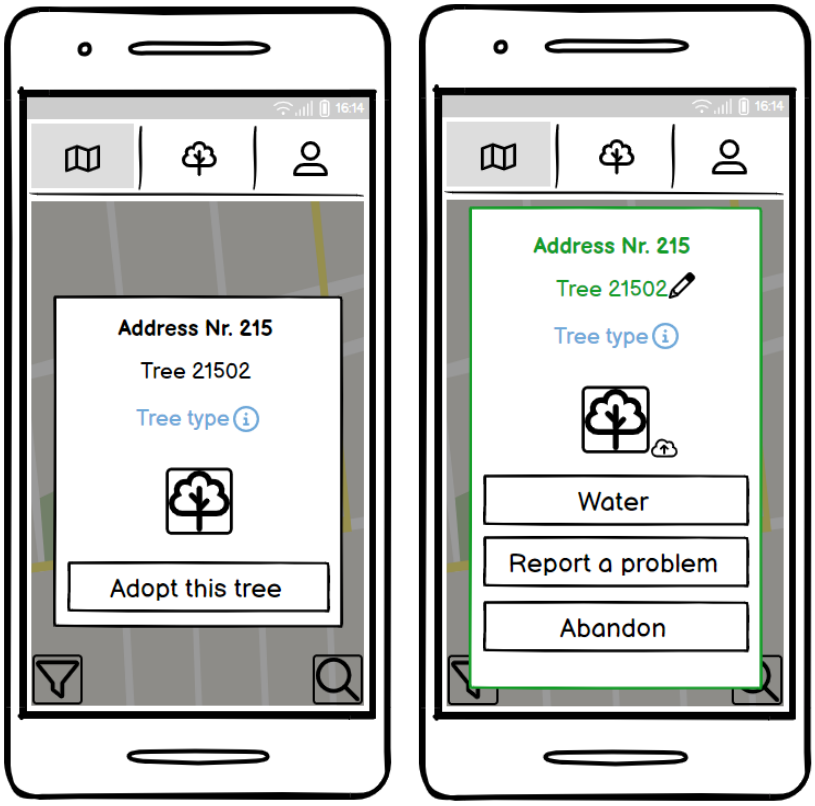
Map Page Map Tutorial

Regarding the map of the trees, buttons will be added to provide functionality such as tree filtering, zip code searching, and zooming to the trees of the user. Moreover, an informational card will provide information about the map and describe each visible icon.



Tree Filtering Zip Code Search

The tree filtering functionality will be provided through a pop up that will allow the user to select their filtering preferences. The map should dynamically update to show the changes based on the selected option. Similarly, the zip code searching will be offered through a pop up that prompts the user to enter a zip code. Thereafter, the map should be updated to zoom at the specified area.



Modal for adopting a tree Modal for managing a tree

The interaction between the user and the trees was redesigned by creating two new dialog modals that are activated when a mobile user clicks on a tree in the map. In the case that the clicked tree is available (green dot on the map), the modal displays information about the tree such as the location, type, and name of the tree. In addition, this dialog enables the users to adopt the tree. The second modal that has been designed handles the interaction between a user and their trees. When a user clicks on a tree that belongs to them (purple dot on the map), they are provided with information about their tree and options that enable them to manage the tree. From this dialog they can select to rename their tree, water the tree, report a problem, or abandon the tree. The created map modals allow for efficient and intuitive navigation within the application, as the users can view and manage their adopted trees directly from the map page.

4. Design

This section covers all design efforts made before implementations were made. Due to the team's inexperience with the related technologies, a significant amount of research was necessary to find appropriate solutions.

4.1 SSL Certificates

Digital Certificates are necessary to encrypt the connection and authenticate a website's identity. Websites require SSL certificates to protect user data, confirm the site's ownership, prevent attackers from developing a fake version of the site, and convey trust in users. The domain of the application is certified through cPanel using a 2048-bit RSA key. This certification expires after April 22, 2023. After the expiration, logging in and registering will not function and fetching errors will appear as pop-up messages. Although cPanel claims that the certificates will automatically renew through their AutoSSL feature, in our experience this was not the case. Nevertheless, upon expiration, the certificates can be renewed by following this process:

1. Visit cPanel Web Application Management & Tools: https://192.232.236.47:2083/
2. Log in using the cPanel credentials:
    - Username: adoptatree
    - Password: a569dof450!
3. Find the "SSL/TLS Status" option under "Security" tools
4. Press the "Run AutoSSL" button

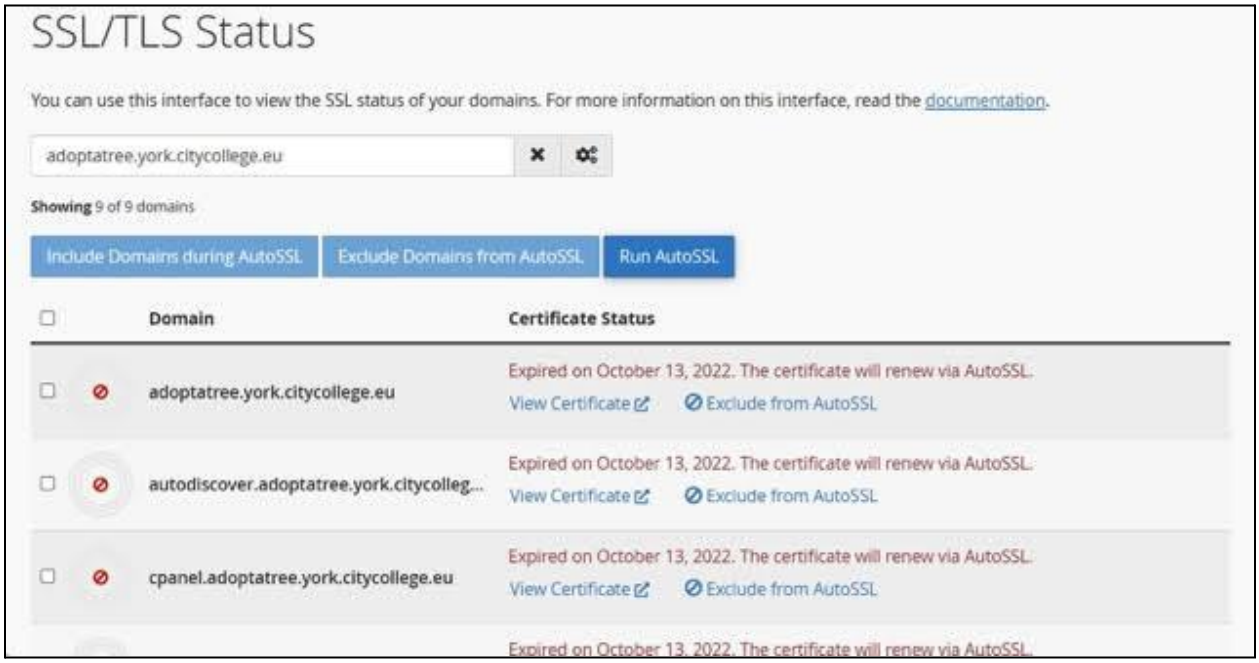


Image 1: Expired SSL certificates

4.2 OpenStreetMap

A free, editable map of the whole world, known as OpenStreetMap, is created by volunteers and distributed under an open-content license. All map pictures and underlying map data are available for free access under the OpenStreetMap License. The switch from Google Maps to OpenStreetMaps was considered due to the pricing of the Google Maps API as Google charges based on the number of requests. Although Google offers some free initial credit for the usage of the API, switching to a free map API might be necessary in the future. Another aspect to

consider is the cost of switching as the map is a complex module and performing this transition could prove to be time consuming and could hinder the progress of the project. Currently, two separate API keys are used (one for production and one for development) which can be found as environmental variables named REACT_APP_GOOGLE_API_KEY under the frontend .env and .env.production files.

4.3 Deck.gl

Deck is a WebGL-powered framework for visual exploratory data analysis specifically designed for large datasets. This framework is used to create visual and interactive layers that display large amounts of data in real-time. The current implementation utilizes the GeoJsonLayer to display the trees as dots on the map. Upon initial research, it seems that DeckGL is compatible with OpenStreetMaps through a mapping library such as Leaflet or Mapbox.

4.4 OAuth2 Certificate

A problem documented by the previous team concerns the email notification system. Their instructions are to update all ".env" files with the correct credentials and enable less secure apps in the GMail account. The latter is not a long term solution as Google deprecated this feature in May 2022, meaning it will require additional credentials to continue to work as intended. To create these credentials, go to the Google Developers Console and create a new project. Select OAuth consent screen then choose external. Create, then under Application name, enter the name of the project (Adopt A Tree). Under Authorized domains, enter the domain name associated with the project. Select Save, then go to Credentials. Under create credentials you will be able to select Web Application, and enter the URL. After that you will be able to create new credentials and update the backend ".env" with them.

4.5 Email Authentication

In the original system, there was no way to verify an email address exists or belongs to the person registering. Because of this, malicious individuals could register an unlimited number of accounts to fake addresses or register accounts under someone else's name. To prevent this, modern web standards utilize a verification process to confirm both aspects of request legitimacy. This process can involve an email or text message being sent to the user, containing a code associated with their account. Once the user enters this code, their account becomes logged-in. This implementation utilized the existing auth component route infrastructure, which had the advantage of reducing complexity but inherits the problems with the original design. For example, if the authentication infrastructure were separated into login / signup, the signup component could send users to an entirely separate verification page. This would allow a verification link to be included in the email. This was attempted and largely successful but was scrapped because it required a lot of redundant code and the routing failed under specific circumstances. This was because it was taking a hashed identifier and the user email, then redirecting the user to a verification component. The problem arose when the user copied the link from one browser to another, or one device to another. In these cases the redirection failed

and it was found that solving the problem would likely require entirely redesigning the process. That being said, the final implementation is highly functional and has no other known issues.

4.6 Google Maps API

Google Maps API key had expired and the Tree Map was not functional. As a result the functionality of adopting a tree could not be accessed. The ‘InvalidKeyMapError’ on the console revealed that the previous API key had expired and needed to be replaced.

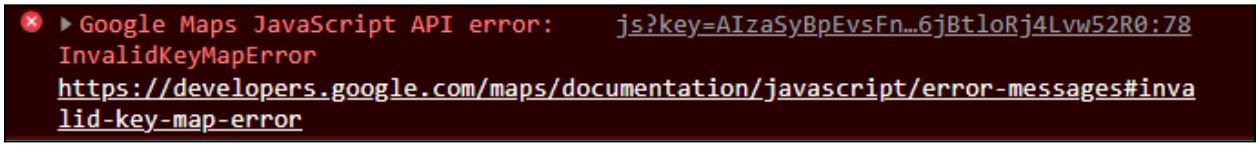


Image 2: Expired API Key Error

Due to changes in the Google API policy, a Billing Google Cloud account is needed to generate this key. The new key was provided by the College and website traffic needs to be monitored due to the charges that Google applies based on usage.

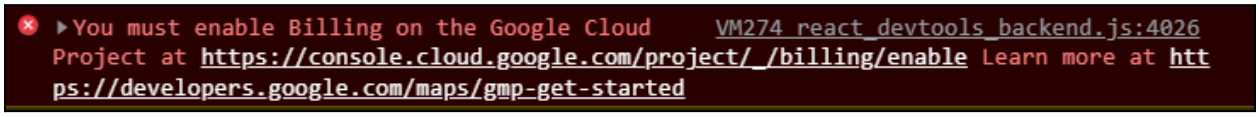


Image 3: Google Cloud Billing Account Required Error

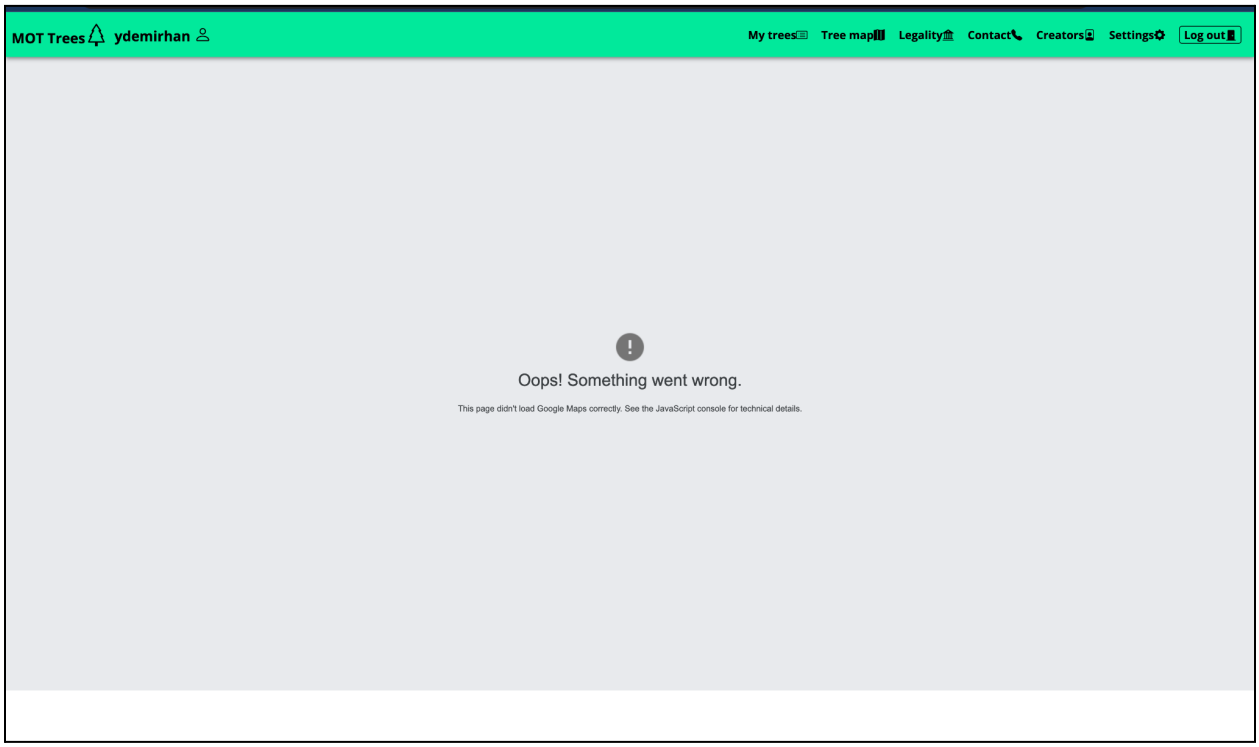


Image 4: Map Not Loading Error

5. Implementation

The most significant frontend and backend implementations will be highlighted in this section. The fundamental ideas and elements utilized in implementation will be covered, along with information on how they should be used and why.

5.1 Backend

This section provides a general overview of the structure and technologies used in the backend of the application. Please note that most of the structure was inherited by the second group and has already been extensively documented in their documentation. Nevertheless, this section covers all the significant additions and efforts made in improving the project in relation to the backend side of the application.

5.1.1 Database

The application uses a MySQL database which is an SQL database compatible with the cPanel site management platform where the application is currently deployed. In addition, the application utilizes Sequelize, an Object-Relational Mapping (ORM) library for Node.js which provides an easy-to-use interface for interacting with relational databases. The database of the deployed application can be viewed on *phpMyAdmin* in cPanel by visiting https://192.232.236.47:2083/.

Log in to cPanel using these credentials:

- Username: *adoptatree*
- Password: *a569dof450!*

After logging in, go to *Databases* and find *phpMyAdmin*. The credentials for accessing the databases are:

- Username: *adoptatree_root*
- Password: *qk@5@gp#p$aej7c#3f&^*sya6*4g!xht4#**

For reference, the SQL database inherited with the project had a schema which consisted of 7 tables: *tree*, *user_tree*, *user*, *user_role*, *role*, *report*, and *password_reset*. The development of various features required modifications to the database. These modifications have been described in the following subsections.

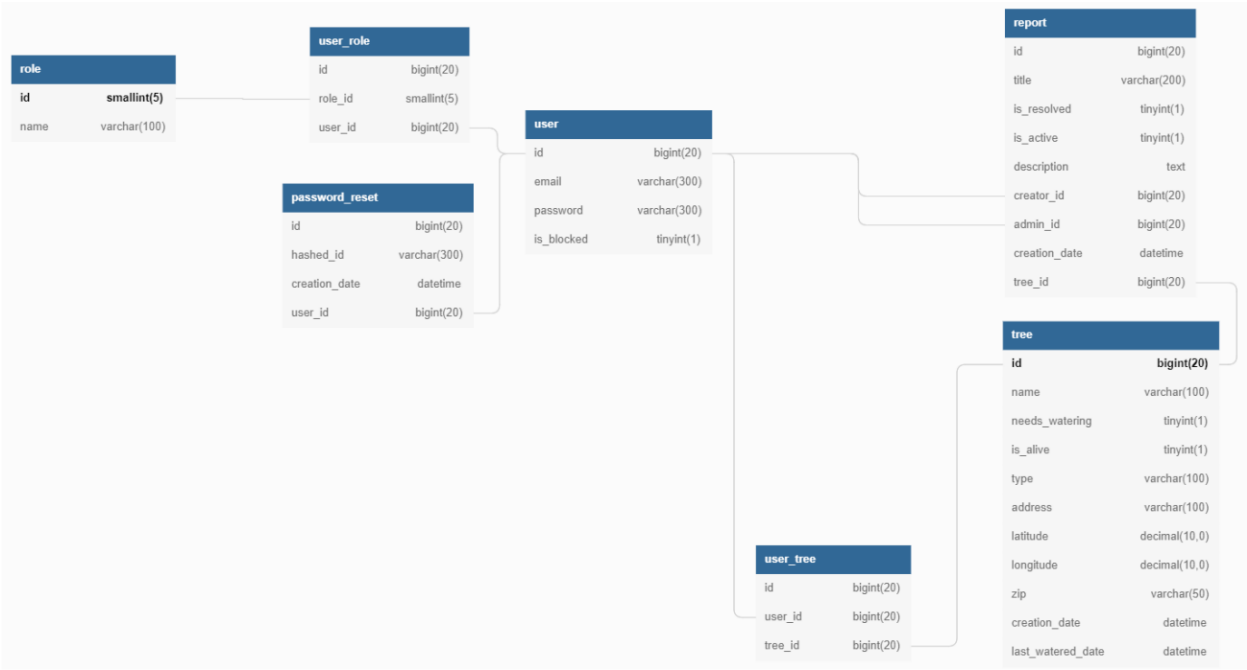


Image 5: Original Database Schema

Tree Nickname

A column named *tree_nickname* has been added to the *user_tree* table. The field was placed in this table as only adopted trees can be given a nickname. Conveniently, storing the tree nickname in this table, guarantees that a nickname given to a tree by a user is automatically deleted if the user no longer owns that tree.

Global Filter

A *global_filter* column has been added to the *user* table. This field holds information about the preference of the user regarding the filtering that should be applied to trees. This field can take one of four integer values which are used to represent the filtering preference and the way filtering should be applied. By default, the field has a value of 0, which indicates that the filter selected by the user should not be remembered and every time they load the map, the user should view all the trees. Positive integer values (1,2 or 3) indicate that the global filter toggle is switched on and the filter selection of the user should be remembered so that when they visit the tree map, the filter they had previously selected is applied. For more details regarding the global filtering functionality refer to section 5.2.2.

Deletion Script

Unverified users are deleted if they are older than 15 minutes. This ensures that “spam” accounts are not stored and allows users who leave the verification page to recreate their account. The process is similar to the pre-existing reset password deletion script.

Removal of DateTime

Originally, the *creation_date* field in the *tree* table had the DateTime format. As this type of format includes the time as well as the date, it takes up more storage space. Considering that currently there are 38441 trees in the database, the difference in size is significant when making requests to transfer all the tree data. In an effort to increase efficiency and reduce the time it takes for the map to load, this field has been changed to Date format. As a result, the transfer size of the requests and the size occupied by the tree data has been reduced (significant for storing all the trees in the users local storage).

Updated Schema

After applying all the changes, the schema of the database takes the form of:

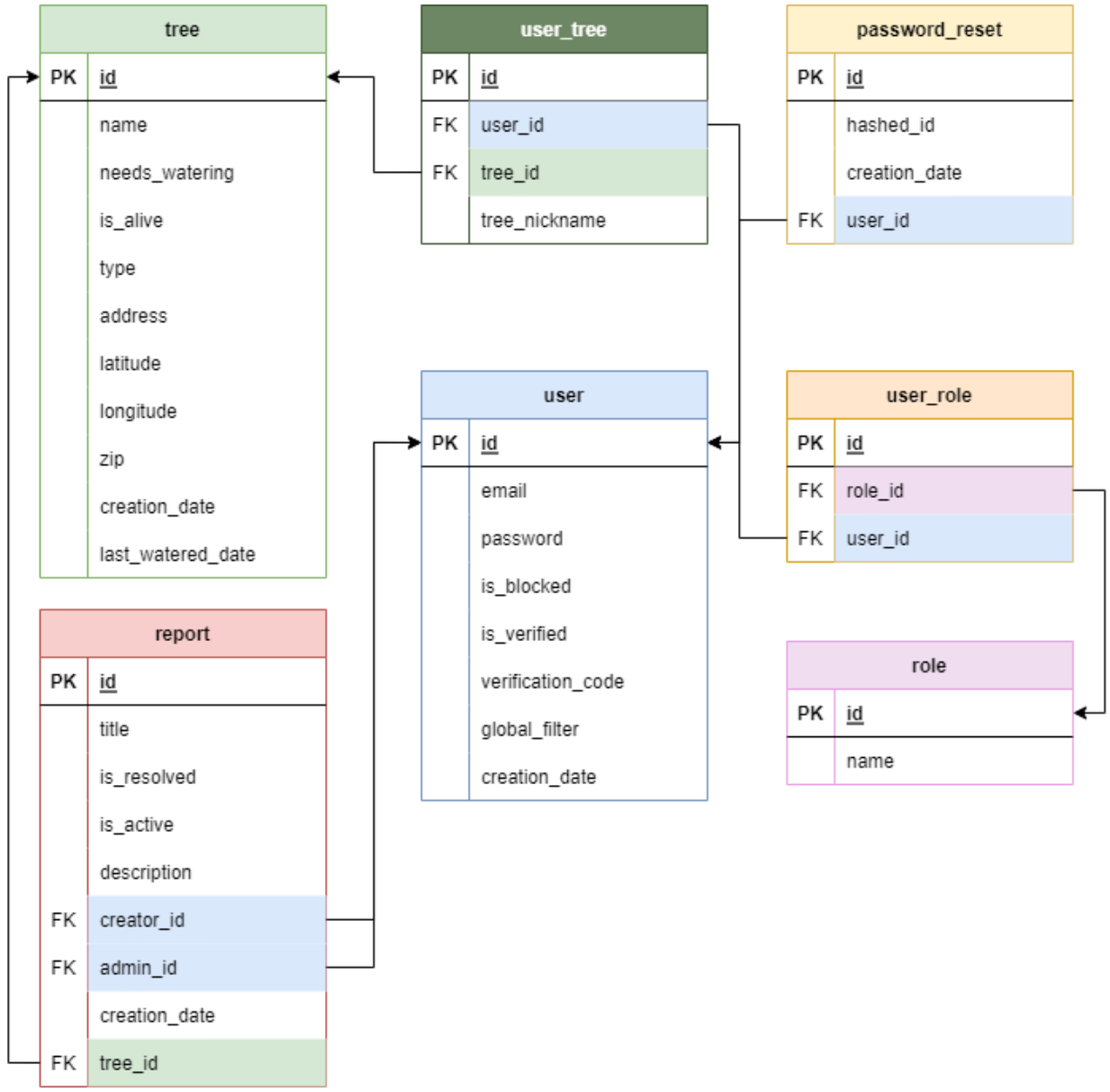


Image 6: Updated Database Schema

5.1.2 Structure

The backend of the application has been designed based on the Model-View-Controller (MVC) paradigm. This design pattern provides higher separation of concerns, reusability, scalability, testability, and flexibility. This subsection covers the most important backend concepts such as: Controllers, Repositories, Models, and Routes. Having a comprehensive understanding of backend concepts and the interactions between them, facilitates future work and improvements. A typical workflow in the MVC pattern involves the following steps:

1. User interacts with View and sends a request
2. Controller receives the request and applies the business logic
3. Controller invokes appropriate methods in the Model to interact with the database
4. Model interacts with the database and returns result to Controller
5. Controller receives result from the Model, formats it, and passes it to the View
6. View receives the data from Controller and displays it to the user

Controllers

Controllers act as an intermediary between the model and the view. They are in charge of responding to HTTP requests coming from the front end using the provided REST API endpoints. Controllers are important as they typically contain the business logic of the application. There are four controller files in the application:

- **Admins-Controllers**
- **Reports-Controllers**
- **Trees-Controllers**
- **Users-Controllers**

When a user sends a request to a specific URL, the routing system takes care of matching the request to a specific action inside the controller file. This is achieved based on the specified URL path and HTTP method. The routing system is managed from the main file (app.js) and four route files that correspond to the controller files:

- **Admins-Routes**
- **Reports-Routes**
- **Trees-Routes**
- **Users-Routes**

Note: When defining new routes, be specific and descriptive with the URL path. In addition, pay attention to the order in which routes are defined. The routing system matches incoming requests to the first route that matches the URL pattern. As a result, routes that pass parameters can be incorrectly matched and cause undesired behavior. For example, if the */user/:userId* is defined before the route */user/create*, the URL */user/create* will be matched by the first route (*/user/:userId*) and the value of the *userId* will be extracted as "*create*". Evidently, this situation is undesired and should be avoided. In this example, to process the URLs separately, the route that passes parameters should be defined last (this behavior in Express is explained here).

Repositories

Repositories are responsible for managing database operations and communicating with the model layer. They provide an interface between the application and the database and serve as an intermediary between the model and the controller, offering an abstraction layer that separates the application logic from the data storage mechanism underneath. The four repository files mentioned below provide methods that utilize Sequelize to perform CRUD operations in the database.

- **AdminRepository**
- **ReportsRepository**
- **TreeRepository**
- **UserRepository**

After performing an operation on the database, a repository typically returns the result of the operation to the calling controller. Ideally, this result should be a resource or a Data-Access-Object (DAO). The current implementation has certain design flaws as there are cases where complete response objects are returned as responses. This creates tight coupling between repositories and responses.

Models

The models encapsulate the data and logic of the application and they provide an interface for other components, such as controllers, to interact with the data. There are two types of models used for the application: response models and database models.

Response models are used to design and create objects that hold the data received from the database. Typically, the repositories utilize the response models to construct response objects which are then returned to the controller. Response models are useful in representing the data in a different format than the one used in the database model. The existing response models are:

- **ReportRes**: Constructs a report object
- **TreeDataRes**: Construct a tree object containing only tree data
- **TreeRes:** Constructs a tree object that contains only the id, owner, coordinates, and creation date.
- **TreesCollectionRes:** Creates a collection of trees that includes owners (deprecated)
- **TreesDataCollectionRes:** Creates a collection of all tree data necessary for the map (uses the TreeDataRes class to construct the collection of only tree data)
- **UserCollectionRes:** Constructs a collection of users (deprecated)
- **UserRes:** Constructs a user object
- **UserTreeRes:** Constructs a response object containing data from the UserTree table

Note: The deprecated response models were previously used to construct a response which contained all the tree data (together with the owners of the trees). Our group has separated the tree data from the user data to improve performance by implementing caching of the tree data.

Database models are used to represent the database schema and simplify communication between the database and Node.js backend. Sequelize necessitates the use of *.define()* to specify the details of each table in the database. Every table should have a corresponding database model which includes a modelName, attributes, database options, and relations. The modelName serves as the table's reference name in model format and should correspond with the class name of the file. The relations of the database are defined in a separate associations.js file.

Note: When performing changes to the database, remember to update the corresponding database models.

5.1.3 Account Verification

Account verification is critical for both security and account management (reset password, change email). When a new user signs up, they receive an email containing a unique code associated with their account (Image 8). This code is stored in the verification_code column in the user table. When the user successfully enters the code linked to their account, the column is_verified changes from 0 to 1. User verification status is checked during authentication, similar to the is_blocked field. If a user fails to enter their code within the allotted time, their account is deleted by a script in the database. Once a user completes verification, they are logged-in automatically.

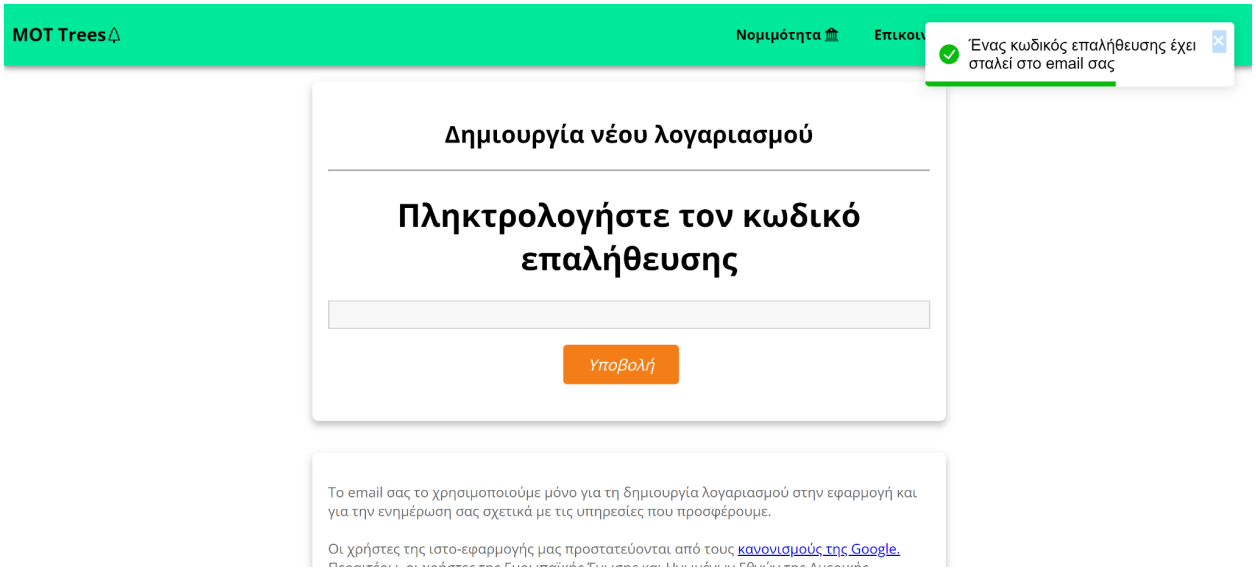


Image 7: Verification Page

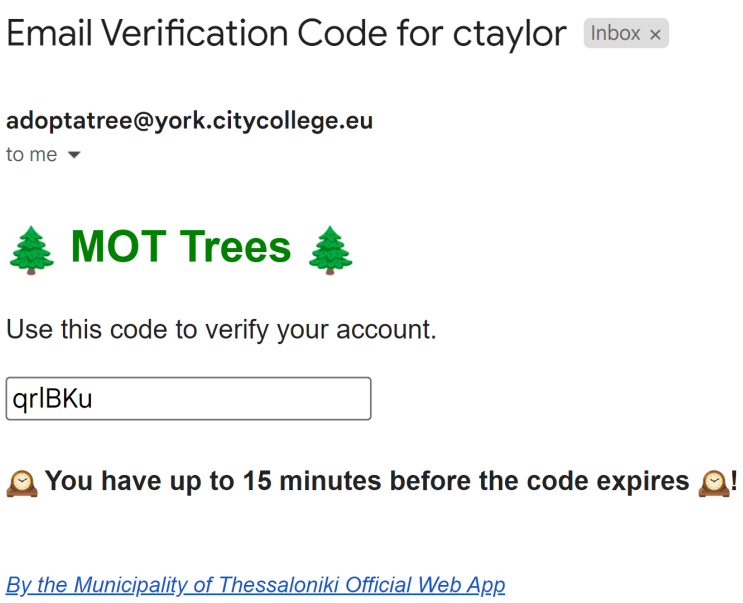


Image 8: Verification Code

5.1.4 Node Clustering

Clustering is a technique used to improve the performance and availability of the application. The main benefits of using a cluster are increased scalability, reliability, and fault tolerance. By distributing resources across multiple nodes, more requests can be handled. This was implemented by using the built-in cluster module in Node (Image 9). The implementation forks the number of processes according to the number of CPU cores available. These forked processes are logged in both the console and the file “logs.txt”. Conversely, the master process simply logs the number of forked processes and handles the termination of a worker process. It should be noted that cPanel requires the use of the syntax “node:cluster” instead of simply “cluster”.

```
22  if (cluster.isMaster) {
23    new RateLimiterClusterMaster();
24
25    const numWorkers = os.cpus().length;
26    console.log(`Master cluster setting up ${numWorkers} workers...`);
27    logToFile(`Master cluster setting up ${numWorkers} workers...`);
28
29    // Fork workers
30    for (let i = 0; i < numWorkers; i++) {
31      cluster.fork();
32    }
33
34    cluster.on("online", (worker) => {
35      console.log(`Worker ${worker.process.pid} is online`);
36    });
37
38    cluster.on("exit", (worker, code, signal) => {
39      console.log(
40        `Worker ${worker.process.pid} died with code: ${code}, and signal: ${signal}`
41      );
42      console.log("Starting a new worker");
43      cluster.fork();
44    });
```

Image 9: Node Cluster Implementation

5.1.5 Rate Limiting

One technique to regulate the amount of times a resource can be accessed over a specified duration is through rate limiting. Primarily, it is implemented to prevent abuse or overuse of a particular service, such as the access to a website or an API. Considering the Google API has a financial cost based on the number of requests, it is prudent to restrict the number of requests from a specific IP address over a span of time. While this is relatively trivial, using it in conjunction with node clustering without using an external tool such as Redis (we were informed the final production server does not support this) can be quite complicated. This problem was solved using the rate-limiter-flexible library, specifically the RateLimitCluster module. It operates by having each worker process communicate with the master which updates the rate limit information for the client IP and instructs the worker to either accept or reject the request. An example can be seen here. Relevant information about each request such as the client IP address, the process id, and the rate limiting details are logged within each worker (Image 10).

You can use tools such as POSTMAN or CURL to test the rate limiter. Here is an example:

```
curl -X POST -H "Content-Type: application/json" -d
"{\"email\":\"test@example.com\"}"
https://adoptatree.york.citycollege.eu/api/users/password-reset
```

```
45  } else {
46    const rateLimiter = new RateLimiterCluster({
47      keyPrefix: "rl", // communication channel
48      points: 300, // number of allowed requests
49      duration: 60 * 60, // duration in seconds
50      timeoutMs: 3000, // if master does not respond in time, reject request
51    });
52
53    const rateLimiterMiddleware = (req, res, next) => {
54      const client = req.headers["x-forwarded-for"] || req.socket.remoteAddress;
55      rateLimiter
56        .consume(client)
57        .then((rateLimiterRes) => {
58          // Send message to worker to update the consumed points
59          console.log("Request received from " + client);
60          console.log("Process ID: " + process.pid);
61          console.log(rateLimiterRes);
62          logToFile("Request received from " + client);
63          logToFile("Process ID: " + process.pid);
64          logToFile(rateLimiterRes);
65          next();
66        })
67        .catch((rateLimiterRes) => {
68          // Error message
69          console.log("Request rejected from " + client);
70          console.log("Process ID: " + process.pid);
71          console.log(rateLimiterRes);
72          logToFile("Request rejected from " + client);
73          logToFile("Process ID: " + process.pid);
74          logToFile(rateLimiterRes);
75          res.status(429).json({
76            message: "Too many requests. Please try again in an hour.",
77          });
78        });
79    };
```

Image 10: Rate Limiting Implementation

Note: Just as cPanel requires using “node:cluster” the syntax must be reflected in the node_module folder where the library is found. Go to the RateLimitCluster file and update line 22 to reflect the “node:cluster” syntax. This will enable it to work on cPanel.

5.1.6 Limit Tree Adoption Per User

This feature was selected to be worked on as it is an important feature that needs to be implemented before launching the application. Without a tree adoption limit, malicious users can adopt a large number of trees without assuming the responsibility of watering them which would ruin the experience for everyone. Therefore, adding this restriction ensures that the responsibility of taking care of the trees in the city is distributed between the users.

The problem is solved by adding a check in the controller function responsible for tree adoptions. This check ensures that the user has not exceeded the maximum number of trees allowed per user before making an adoption. To find out the number of trees adopted by the user, the *adoptTree* function inside *tree-controllers* calls the *countUserTrees* function of the *TreeRepository* file. This function queries the *UserTree* table to find the number of trees adopted by the specified user and then returns this value to the controller. Currently, the limit is **three** trees per user. In the case that the user has already reached the limit, an appropriate message is displayed to the user.

```
118   //Find the amount of trees that the user has adopted
119   static async countUserTrees(uid) {
120     try {
121       return await UserTree.count({
122         where: {
123           user_id: uid,
124         },
125       });
126     } catch (err) {
127       throw "Internal Server Error at countUserTrees";
128     }
129   }
```

Image 11: Counting the number of trees adopted by the user

5.1.7 Prevent Two Users Owning Same Tree

An issue that was noticed in the deployed version of the application was a situation where multiple users would load the map at the same time and try to adopt the same tree. This would result in both users succeeding in adopting the same tree. To remedy this situation, a check has been added in the backend function responsible for tree adoption. This check ensures that the tree that has been requested to be adopted does not have an owner. If two users attempt to adopt the same tree at the same time, one of the users should succeed while the other user should get a message saying that the tree has already been adopted. The *checkTreeAvailability* method searches the *UserTree* table for an existing tree, which would indicate that the tree is already adopted.

```
static async checkTreeAvailability(treeId) {
  try {
    let tree = await UserTree.findOne({
      where: {
        tree_id: treeId,
      },
    });
    if (tree) {
      return false;
    }
    return true;
  } catch (err) {
    throw "Internal Server Error at checkTreeAvailability";
  }
}
```

Image 12: Checking the tree availability

5.2 Frontend

The frontend's implementations, structure, and usage of components, authorization, and environment variables are covered in this subsection. The frontend received special attention as one of the most significant priorities was the redesign of the application for mobile users and the implementation of features to improve the user experience.

5.2.1 Structure

The frontend's structure mostly shows a hierarchy of folders, which was created by the previous groups. The 'components' folder is the most significant of these directories. With the aim of separating concerns and providing modularity, the components have been arranged using the following rules:

1. For each style file (.css) there must be a separate folder.
2. Each folder must include any helpers, functions, or other components that are related to its main component. The main component is identified by using the same name as the folder. Note: All components are functions too.
3. Some folders may group together different components or folders.

An informational overview of the directories and files in the frontend folder has been provided below:

- Assets - contains all images, videos, icons, or other raw media types
- Components - contains all application components
- Context - contains all globally accessible context hooks. The ‘hooks’ are responsible for holding globally available hooks (methods) that can be used in various components
- Hooks - contains implementations of custom hooks
- Models - contains objects that will encapsulate the response data retrieved from the backend APIs

- Util - contain utility functions (equivalent to helpers in the backend) available globally across the frontend (such as the language.js file)
    - Language.js file - contains greek translations for various labels. All Greek text used in the application should be inside this file.
- App.js file - renders the top-level components and provides global configuration
- Env files - contain frontends environment variables

5.2.2 Adding functionality & Redesigning Application

This section describes all the efforts made to improve the application and the user experience. As discussed before, the main focus was the redesign of the application for mobile users. Since this was determined to be the main priority, most of the development was focused on implementing the necessary changes for improving the experience of mobile users.

Email notification system

While the previous team acknowledged the latent issues with the email notification system, it remained unimplemented and insecure. To remedy this and ensure a reliable solution the creation of OAuth2 security credentials and a refresh token were necessary. Due to a change in Google policy in May 2022, the previous solution would no longer suffice as the enable “less secure apps” feature was deprecated. After creating the correct security tokens, changes were made to the email helper and “.env” files to reflect the new values (Image 13). The new implementation should only require alteration if the domain name or email address were to change, as the refresh token prevents the authentication token from expiring.

```
12  NOTIFICATIONS_EMAIL: adoptatree@york.citycollege.eu
13  NOTIFICATIONS_EMAIL_PASSWORD: 250rax700
14  NOTIFICATIONS_EMAIL_SERVICE: gmail
15
16  CLIENT_ID: 532291279438-vtar4g5ri948t1i2idmltbval9iqbchi.apps.googleusercontent.com
17  CLIENT_SECRET: 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
18  REFRESH_TOKEN: 1//04iEPHuYyByJvCgYIARAAGAQSNwF-L9IrI6zrYtFvcdGTVf-WKIZdoVCa1be4HJ_JwRt_rOHCy8z_KqzsY8bE7-S4Z6seNIaYSqY
19  ACCESS_TOKEN: ya29.a0Aa4xrXNMoiX3N2VCq0XG18wg1W1cIYNk6kSonmZIN2XMmX5V0_bbEs0Jtma5F8RTExmi7W2I6eyBupwJfZuu4s-8yT-1TOqw0rCPHH6TrJCKzWVECBV1jodErSFg_Ki
20  EXPIRES: 3599
```

Image 13: Environment variables

Adding custom tree names

Although the requirement of allowing users to rename their adopted trees has a low priority as it is not critical for the launch of the application, we decided to work on this feature to gain a deeper understanding of the overall code used in the project. Moreover, an argument can be made that allowing users to customize the name of their trees facilitates a connection between the user and the tree. The frontend solution has been developed by utilizing the Material-UI library so that when the user clicks the rename icon they are prompted with a modal for completing this process. This action sends a request to the backend so that a nickname for the tree can be added to the database. We decided that the tree nickname field should be added to the “UserTree” table instead of the “Tree” table. This decision was based on the fact that only adopted trees should have custom nicknames. Saving the nickname in the “UserTree” table guarantees that the tree belongs to a user and in the case that the user decides to relinquish ownership of a tree, the nickname will be deleted so that future owners are not presented with the nicknames used by previous owners. Moreover, this approach is more efficient when

considering that unadopted trees are not going to have a nickname. Storing the nickname of trees in the “Tree” table would be space inefficient and would cause extra overhead when fetching the trees.

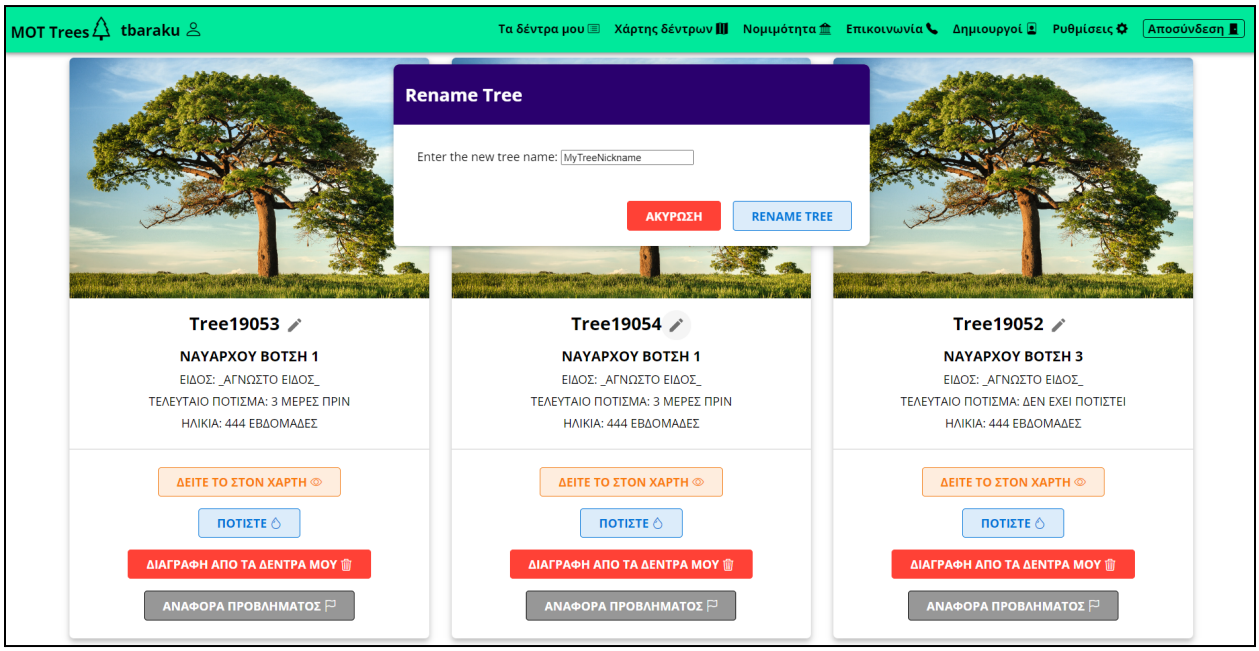


Image 14: Process of renaming a tree

Apart from the process of adding a nickname to a tree, there is another action that checks if a tree has a nickname, fetches that nickname and displays it to the user.

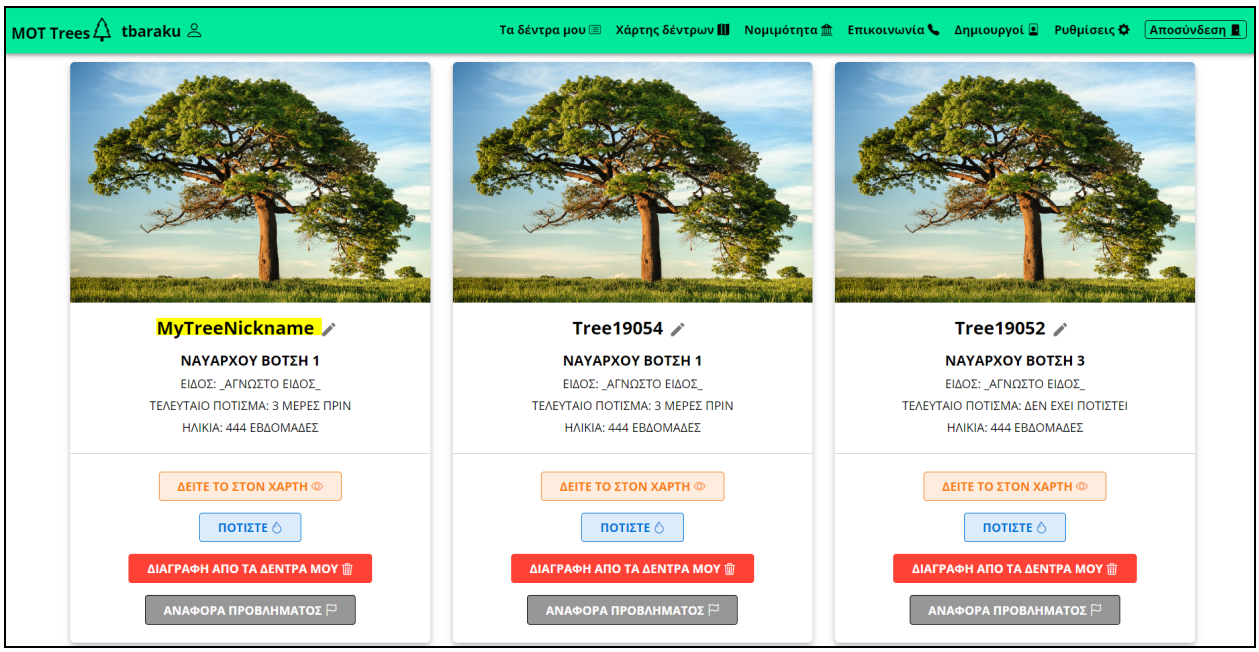


Image 15: Successfully displaying the Tree Nickname to the user

Note: The maximum size of tree nicknames has been set to 15 characters. This ensures that the nicknames fit inside the tree cards on most screens.

Tree Components of Card

The page from where users can view their trees, referred to as the “MyTrees” page, has been completely redesigned. In the web application, the trees adopted by the user are displayed in a card with all the information about the tree and all the available actions. In a mobile application, the space is limited and displaying all this information would consume multiple pages. As a result, the tree cards have been simplified to only show the address of the tree and some important functionality such as watering the tree and viewing the tree on the map. Regarding the ‘My Trees’ Page, there are more implementations mentioned in the next sections.

Image 16: ‘Tree Card’ in Web View

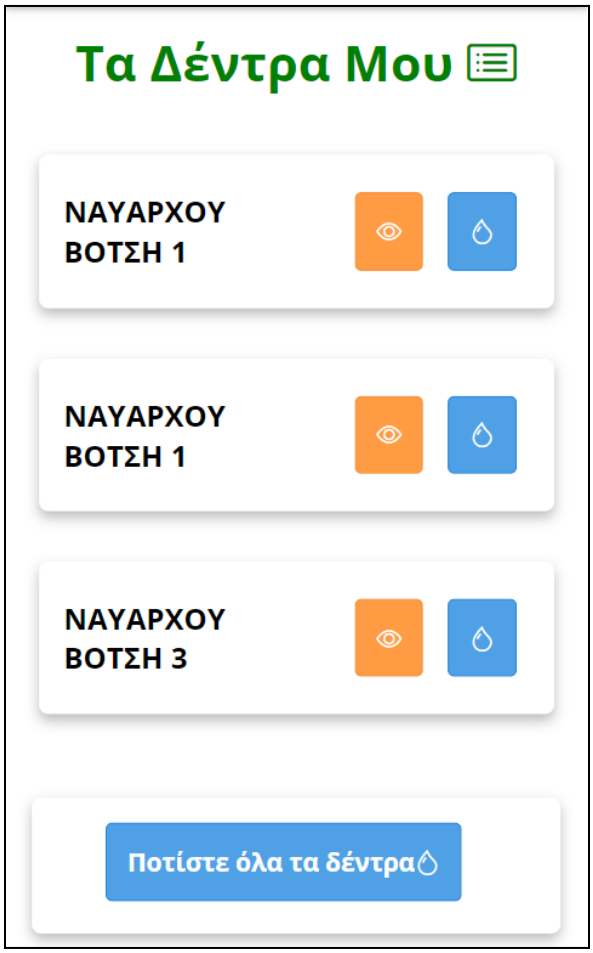


Image 17: ‘My Trees’ in Mobile View

Redirect user to their tree

This is another requirement that was identified from the redesign prototypes of the mobile version of the application. On the “View My Trees” page there is a button next to each of their trees that redirects the user to the map and zooms in on their specific tree. This feature is intended to enhance the user experience and make it easier for the user to distinguish the

location of their tree. When the user clicks this button, the coordinates of that tree are passed to the map component so that after being redirected to the map page, the map is centered and zoomed into the specific tree.

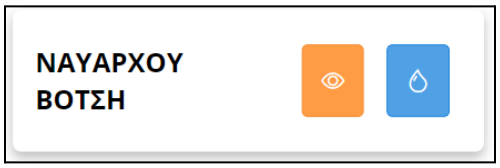


Image 18: View Tree on Map Icon

Water All My Trees

This requirement was identified during the redesign of the application for mobile users. This button would allow users to easily mark all the trees that they have adopted as watered. To implement this feature, first a button was designed on the frontend which is displayed on small devices at the bottom of the 'MyTrees' page. 'Water all my trees' button appears only if there are two or more trees adopted by the user. Upon clicking this button, a request is sent to the backend where all the trees owned by the user are watered.

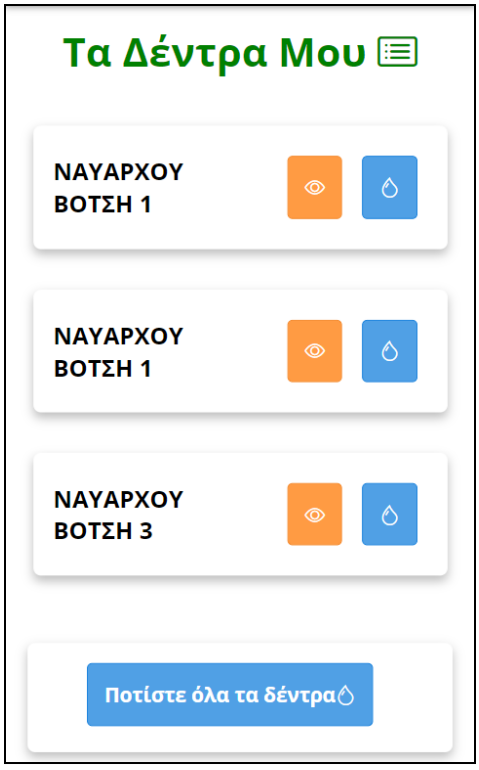


Image 19: Water All My Trees Button

Adopt a tree button

In the 'My Trees' page, if the page is visited by a user that does not have any adopted trees, the 'adopt a tree' button is displayed to encourage them to adopt a tree. The button redirects the user to the map page, where they can select and adopt trees. After the first tree is adopted the button is shifted below the tree card and it is shown to the user until the limit of the trees is

reached. An additional functionality tied to this button is the redirection of the user near their trees after having adopted a tree. This feature is further explained further in the below functionality.

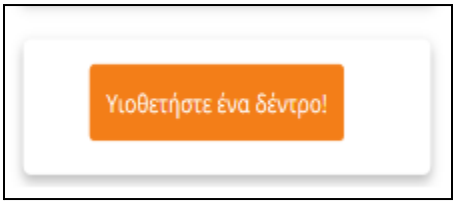


Image 20: Adopt a tree button

Redirect near trees

Once a user has already adopted a tree, they can view it on their “MyTrees” page. As long as they have not reached the limit of the adopted trees, a button is displayed on this page, which encourages them to adopt another tree (the “Adopt a tree button”). Upon clicking this button, the user is redirected to the map page. Assuming that users would want to adopt trees relatively near the location of the trees they have already adopted, this redirection has been modified so that the loaded map is centered and zoomed at the position of their trees. In the case that the user only has one adopted tree, then the coordinates of that tree are used. If the user has adopted two trees, the coordinates of their second tree will be used for the redirection. The method responsible for handling this redirection is included below.

```
const redirectNextToTree = () => {
  //Values used for redirecting
  let coords = [];
  let zoomLevel = 20;

  //If the user has only one tree, use the coordinates of that tree
  if (loadedTrees.length === 1) {
    coords = [
      { lat: loadedTrees[0].latitude, lng: loadedTrees[0].longitude },
    ];
    //If the user has two trees, redirect them to the second tree (more recent)
  } else if (loadedTrees.length === 2) {
    coords = [
      { lat: loadedTrees[1].latitude, lng: loadedTrees[1].longitude },
    ];
  }
  navigate("/map", {
    state: {
      coordinates: coords[0],
      zoom: zoomLevel,
    },
  });
};
```

Image 21: Method for redirecting user next to their tree

‘Account View’ Page

Visual improvements were made to the ‘Settings’ page. It previously contained three buttons for changing the email, the password, and deleting the account. On this page, blank spaces were

visible and the large buttons were inadequate. With the changes made, the page has been renamed to ‘Account View’ and navigation links have been added to redirect the user to the ‘Legal and Privacy’ page, ‘Contact us’ page, or the ‘Who are we’ page. These links previously were accessible in the navigation bar and now resemble footer links in the new ‘Account View’ page.

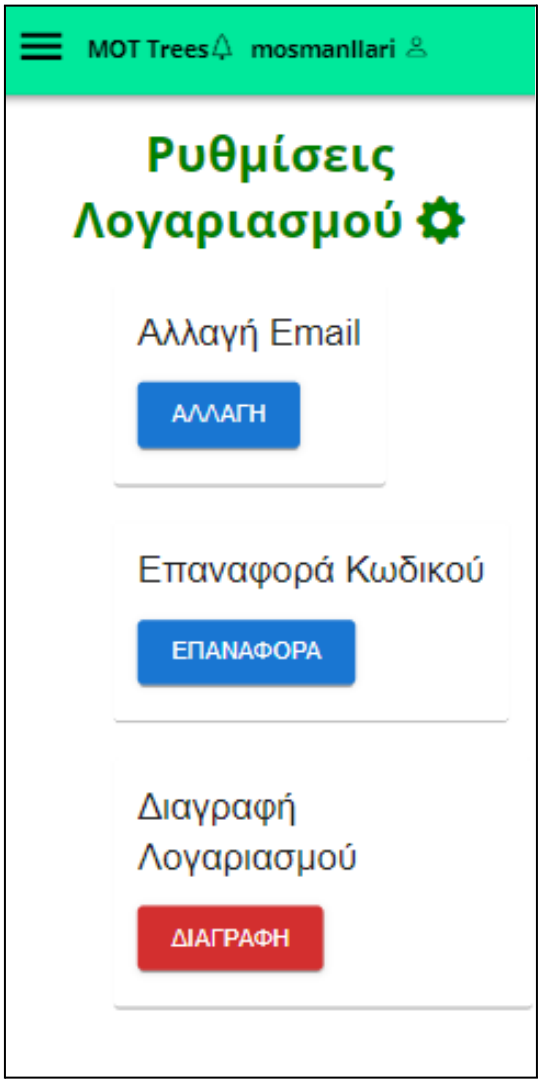


Image 22: Old Settings’ Page

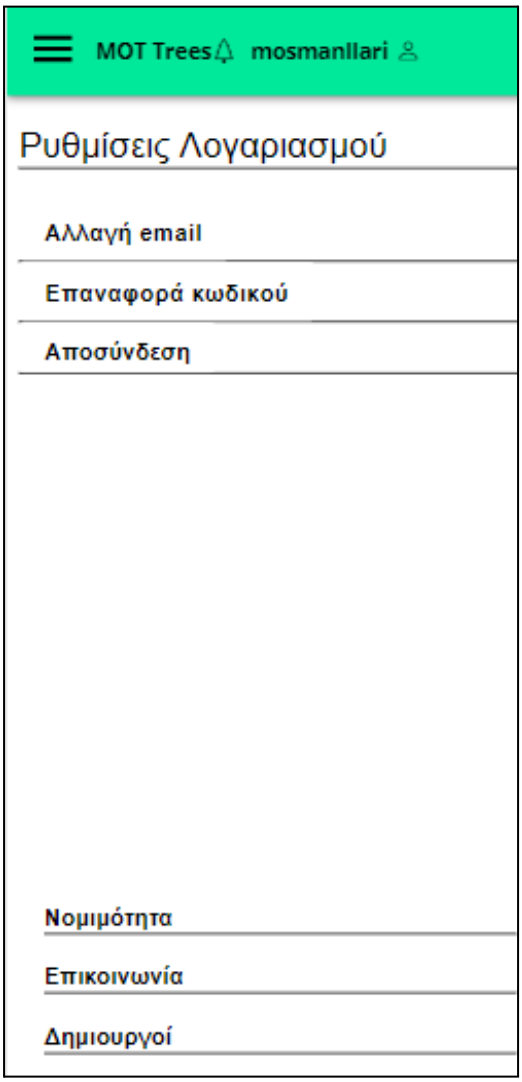


Image 23: New ‘Account View’ Page

Restrict map to map of Thessaloniki

The existing implementation of Google Maps, allowed users to zoom out and move to any part of the world. However, since the scope of the project only includes trees within Thessaloniki, it was decided that the maps should be restricted to show only the map of Thessaloniki. To accomplish this, the latitude and longitude coordinates have been defined. The “strictBounds” value has an effect on the overall size and level of zooming out that is allowed on the map. These values are used to restrict the map by setting the “option” attribute of the “GoogleMapReact” tag.

```
const options = {
  restriction: {
    latLngBounds: {
      north: 40.69085,
      south: 40.550558,
      west: 22.872475,
      east: 23.039703,
    },
    strictBounds: true,
  },
};
```

```
<GoogleMapReact
  options={options}
  bootstrapURLKeys={{
    key: process.env.REACT_APP_GOOGLE_API_KEY,
  }}
  defaultCenter={center}
  defaultZoom={defaultZoom}
  yesIWantToUseGoogleMapApiInternals
  onGoogleApiLoaded={({ map }) => {
    mapRef.current = map;
    deckOverlay.setMap(map);
  }}
></GoogleMapReact>
```

Image 24, 25: Implementing Map Restriction

Navigation Bar

Significant changes were made to the navigation bar for the mobile view of the application in order to improve its appearance and functionality. One of the key changes was the removal of the sliding bar, which provided access to all of the site's pages. It was determined that this feature was not necessary and that a simpler, more intuitive design would be more effective. As a result, the navigation bar has been streamlined to resemble other popular social media platforms like Instagram. These updates aim to enhance the user experience and make it easier for mobile users to find the information and resources they need.

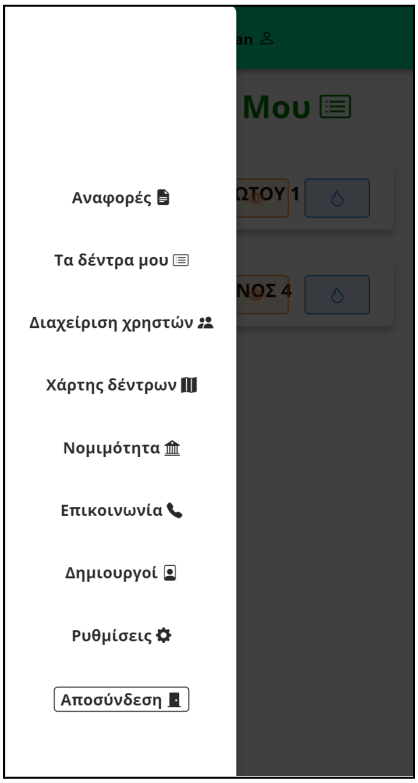


Image 26: Old navbar slider

There is a slight difference between the navigation bar available to regular users and the additional pages available to admins. Admins have access to the ‘User Management’ and ‘User Reports’ pages.

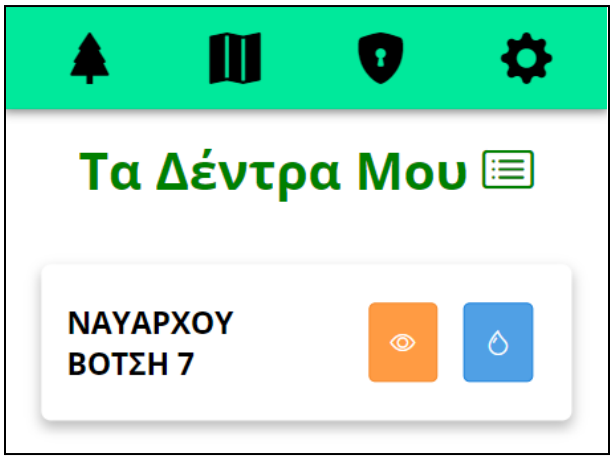


Image: 27 Admin nav bar

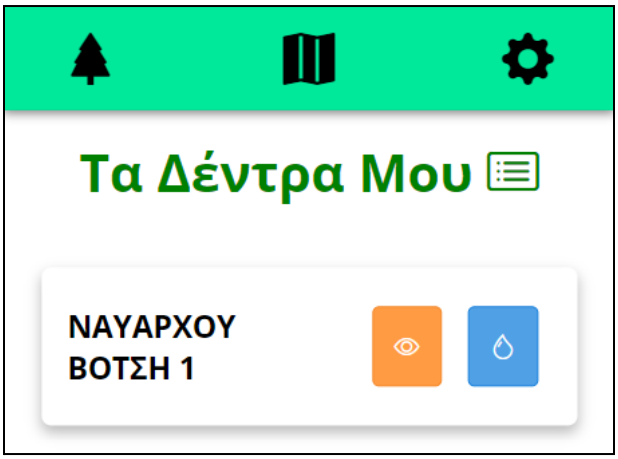


Image: 28 User nav bar

Separation of tree and user data

The separation of the tree and user data was one of the most important and challenging features of the application. The previous implementation was based on combining the tree data and the user data in the backend using a single query and sending that data to the frontend. Combining the user and tree data is necessary so that the trees can be differentiated and accurately displayed based on their owner. However, the approach used was inefficient as the large number of trees was being queried and sent from the backend to the frontend every time that the map was being opened. Consequently, this heavily affected the time it took for the trees to load.

In response to this, a solution was devised to separate the tree data and user data into two different GET requests with the intent of storing the tree data in the users localstorage. One of the requests fetches the data about all the trees whereas the other request fetches the user data that contains information about which tree belongs to which user. The tree data can be considered as static information as it rarely ever changes (unless there are more trees that have been planted and need to be added to the database). However, data about the owners of the trees changes frequently and this information needs to be fetched from the UserTree table. When users load the map page for the first time, a request is sent to fetch all the trees. This is the only instance when data about all the trees needs to be transferred. In all other cases, the tree data is simply fetched from the localstorage, thus significantly reducing the time it takes for the trees to load.

As localstorage size is limited to 5MB by most modern browsers, to preserve space, only the unique tree attributes required for displaying the trees must be stored. This includes the ID of the tree and the coordinates of its location. Other attributes such as the shape used for displaying the tree on the map, can be appended to the tree data using javascript functions. An important function called “combineData” adds the information about the owner of the tree to the

tree objects. This is achieved by looping through the tree data and the data fetched from the UserTree table and assigning an owner to the tree if there is a match on the “tree_id”.

```
//Function that retrieves the combined data of the trees and the users
const getTrees = async () => {
  //Fetch the data about the users that have trees
  await fetchUserTrees();
  //Check to see if the tree data is already stored in localstorage
  if (localStorage.hasOwnProperty("treeData") === false) {
    //If it is not, fetch the tree data from the backend (gets the data and stores it in localstorage)
    await fetchTrees();
  }
  //Get the tree data from localstorage
  treeData = getStoredTreeData();
  //Combine the tree data and the user data
  let combinedData = combineData(treeData, userData);
  //Set the sourceData to the combined data
  setSourceData(combinedData);
};
```

Image 29: *getTrees* Function

```
//combine the data from the trees and the users
const combineData = (treeData, userData) => {
  let combinedData = treeData;
  for (let i = 0; i < treeData.length; i++) {
    for (let j = 0; j < userData.length; j++) {
      if (treeData[i].properties.id === userData[j].tree_id) {
        treeData[i].properties.owner[0] = userData[j].user_id;
      }
    }
  }
  return combinedData;
};
```

Image 30: Function that adds the owners of the trees to the tree data

Admin Panel

After notable modifications were done to the navigation, one of the admin route links was non functional. Admin Panel is a new page that is used only by administrators just for routing to two other pages, ‘Reports’ and ‘User Management’. The Admin Panel page has a simple design which consists of two buttons for the admins to redirect themselves to other pages that contain administrative functionality.

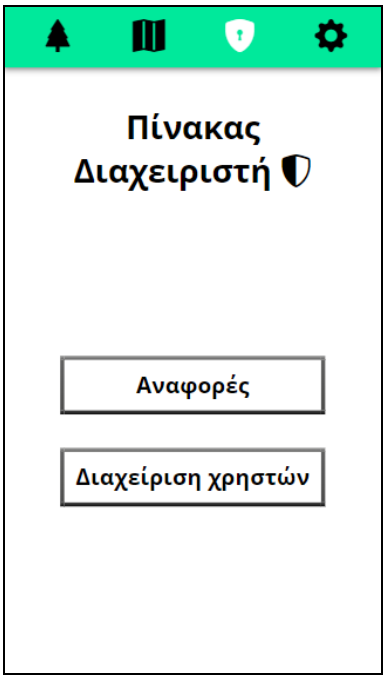


Image 31: Admin Panel Page

Filter modal

Initially, according to the Balsamiq Wireframes prototype, this modal was supposed to be created with checkboxes so the user could select more than one option to reflect their preferences. During implementation, it was decided that a better approach would be to use radio buttons to provide mutually exclusive options for filtering the trees. As a result, the filtering process has been simplified without compromising in terms of the offered functionality. The offered options include:

- Show All Trees
- Show Adopted Trees
- Show Available Trees

To dynamically change the data displayed in the map, the GoogleMapsOverlay layer had to be updated. This was achieved by calling the *finalize()* method on the deckOverlay constant that contains the layer that is placed on top of the map to display the trees as dots. This method removes the existing overlay to allow for an updated overlay, which contains the filtered tree data, to replace the old one. The *finalize()* method is called twice after updating the *sourceData,* which contains the tree data that is used to create a DeckGL overlay layer. The *sourceData* is updated after the trees have been filtered and it is also updated when a user adopts a tree so that the color of the tree can be updated.

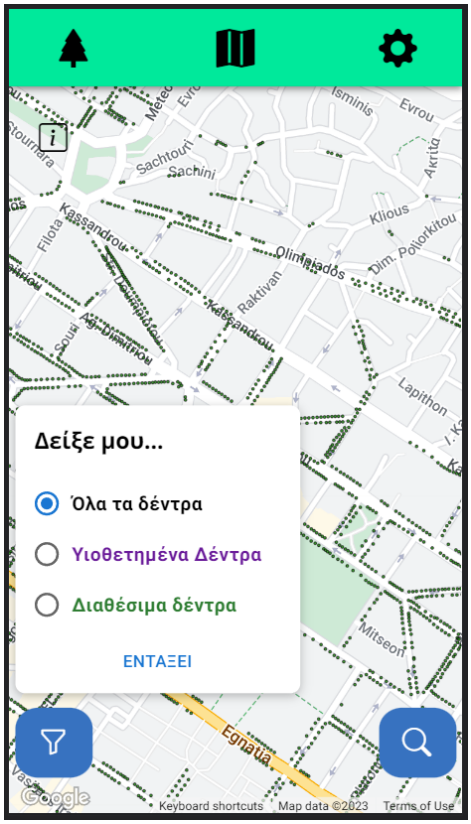


Image 32: Filter Modal

Global Filter

Having implemented the filtering functionality, a debate sparked regarding the expected behavior of the application in relation to whether the application should keep track of the filtering choice selected by the user. Arguments can be made that some users might always expect to see all the trees when loading the map whereas some other users might expect that their filtering choice is remembered and applied every time they load the map. It was decided that this choice should be offered to the user.

In the settings page, a toggle switch has been added which allows users to set their preferences on whether the filtering should be applied globally or not. If this switch is enabled, the filter applied to the map is tracked in the *global_filter* field of the *user* table in the database. The default value of the global_filter field is 0, which indicates that the switch is turned off. Non-zero values correspond to the switch being on. The numerical values used to determine how the trees should be filtered initially (when the map loads):

- Value ‘0’ - Show all trees
- Value ‘1’ - Show all trees
- Value ‘2’ - Show only trees adopted by the user
- Value ‘3’ - Show only available trees

The implementation of this functionality was quite problematic. In the settings page and in map page two separate requests functions are needed. The *getGlobalFilter()* function makes

requests to fetch the value of the global filter from the database while the *setGlobalFilter()* makes requests to modify it. Problems were encountered when attempting to get this value in the map page and using it to set the filter of the trees prior to the map loading. In some instances, the map would load while the value of the global filter was still null. When this occurred, the choice of the user was being disregarded and the map layer was being loaded with all the trees. To remedy this problem, the *getTrees()* method has been modified to wait and check the value of the global filter before setting the source data of the map layer. This ensures that an initial filter is applied to the map according to the value of the global filter (if the option is enabled by the user).

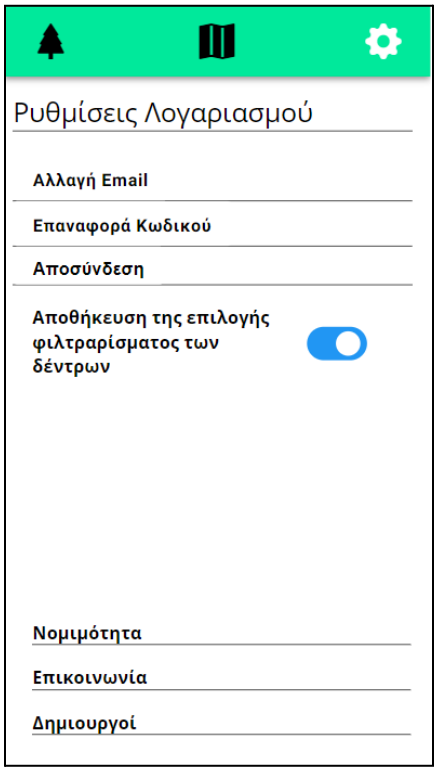


Image 33: Global Filter Switch

Search with ZIP code modal

As defined during the design phase, a search modal will be offered to the users so that they can search the tree map using zip codes. This involves users typing a 5 digit zip code and clicking the search button, which sends a request to the backend to search the database for a tree that belongs to that zip code. If a tree is found with the specified zip code, the coordinates of that tree are used to recenter the map. If a tree is not found or the user enters an inaccurate zip code, a message is displayed to the user to notify them that the search failed and they should try again.

As an alternative to this implementation, the use of the Google Maps Geocoding API was considered, however, this would result in an increased cost for the API usage as each search request would come with a cost. Therefore, it was decided that the implemented solution, which utilizes the stored zip code information, is suitable for the purposes of the application.

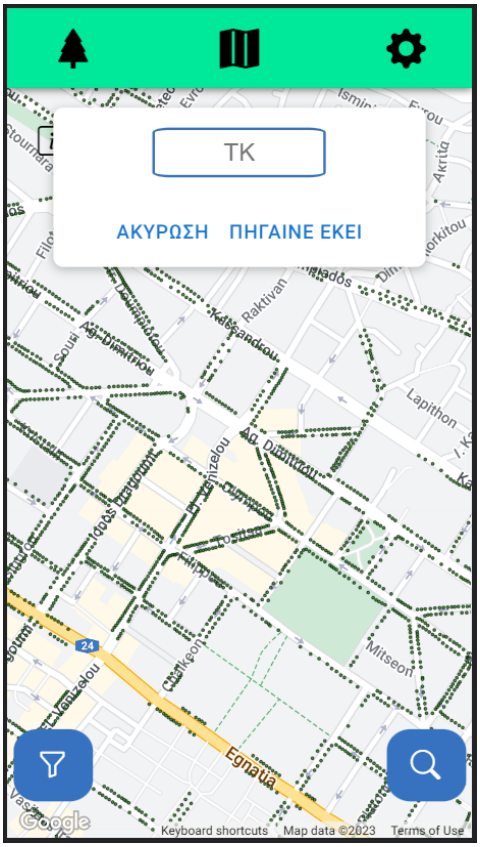


Image 34: Zip Search Modal

Tree map modals

The interaction between the user and the trees has been redesigned by creating two new dialog modals that are activated when a mobile user clicks on a tree in the map. In the case that the clicked tree is available (green dot on the map), the modal displays information about the tree such as the location, type, and name of the tree. In addition, this dialog enables the users to adopt the tree. The second modal that has been created handles the interaction between a user and their trees. When a user clicks on a tree that belongs to them (purple dot on the map), they are provided with information about their tree and options that enable them to manage the tree. From this dialog they can select to rename their tree, water the tree, report a problem, or abandon the tree. The created map modals allow for efficient and intuitive navigation within the application, as the users can view and manage their adopted trees directly from the map page.

Image 35: Available Tree Modal

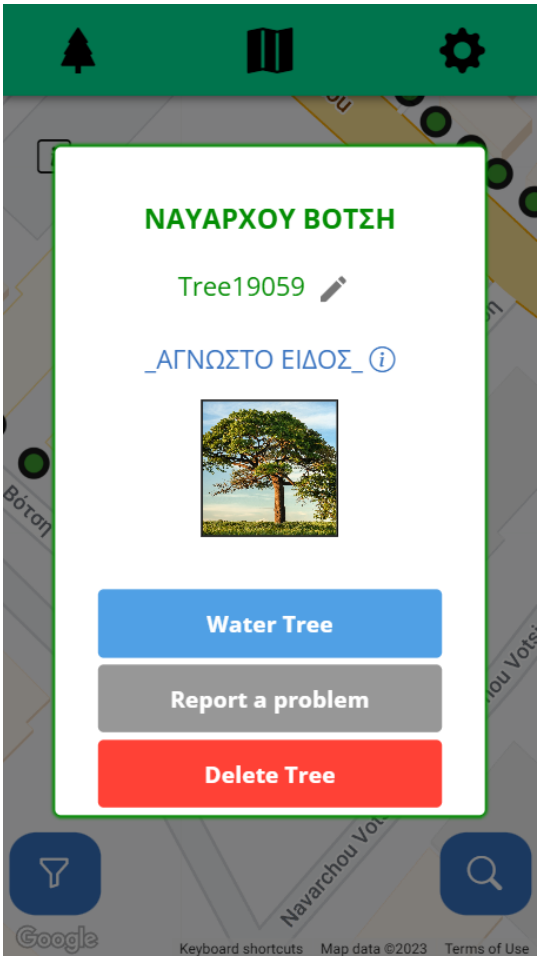


Image 36: Adopted Tree Modal

Removing circular dependency warnings

The circular dependency warnings were interfering with testing both on the development and production environments, so the warnings for them were removed. A circular dependency warning occurs when two or more modules import each other, and this structure was inherited from the previous team. While circular dependencies can make debugging and comprehending code more challenging, they are not necessarily indicative of poor design. For example, having circular dependencies can improve simplicity by reducing the number of components and modules needed. They can also improve reusability by allowing modules to import and use other components without going through an intermediary module. Finally, they can also increase performance by reducing the number of lookups and imports needed. In the future, restructuring the codebase to account for the circular dependencies may prove worthwhile, but that was not found to be the case at the current phase of the project.

Zoom Button

Alongside the filter, search and tutorial buttons, a new button is presented to provide functionality to mobile users. The “Zoom to all trees” button allows users to easily find and view each tree they have adopted by zooming in on each one. Each click is equivalent to the “Next”

command, which takes the user to their next tree and loops back to the beginning when all trees have been viewed. In the case that the user only has one tree, then the button simply zooms to that tree every time it is clicked.

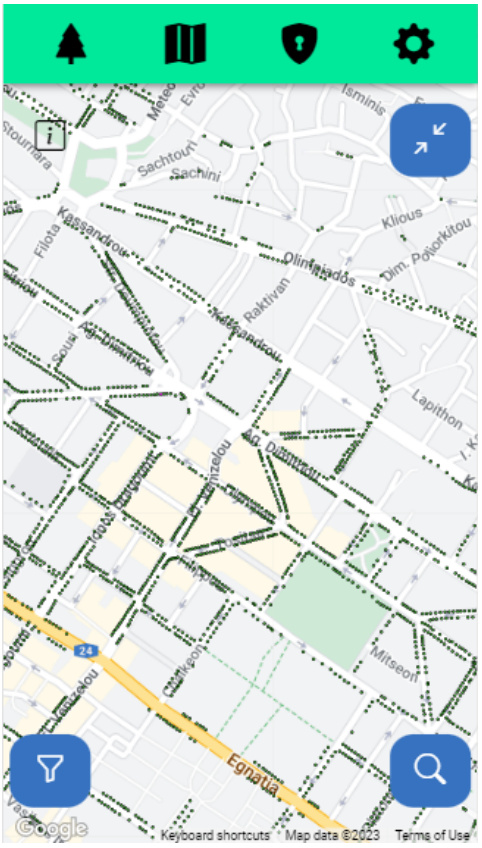


Image 37: Zoom Button (top right)

Tutorial Button

With all the added functionality and the new icons used in the map, a “Tutorial Button” has been devised which acts as a map legend and serves the purpose of informing the user regarding the buttons and the features of the application. The button resides in the top left corner of the map and it is symbolized by an “i” icon which stands for information. The tutorial button has a minimalistic design and a transparent background so as to not get in the way of the map.

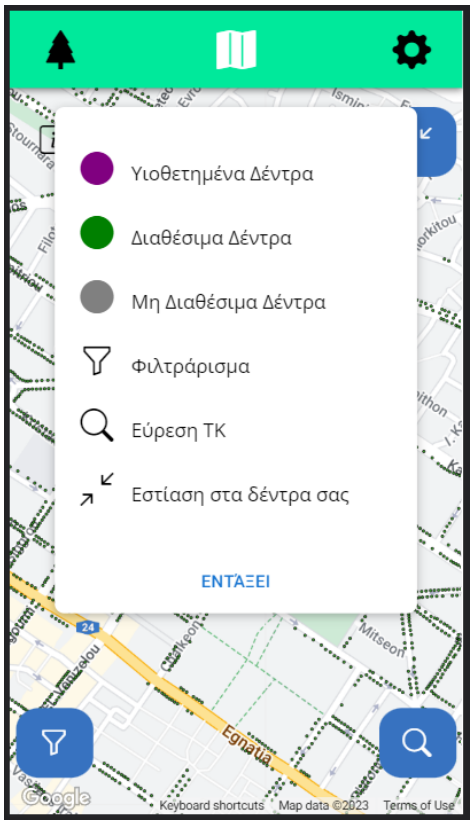


Image 38: Tutorial Modal

Tree Scaling

In an effort to make it easier for users to locate their trees on the map, a tree scaling factor has been introduced for trees that are adopted by the user. The *calculatePointRadius* function is called inside the GoogleMapsOverlay layer and it returns the radius of the tree based on the current zoom level of the map. The *finalize()* method is called to clear the existing overlay. Without clearing the existing overlay, the trees appear stacked on top of each other. The zoom level is kept in a React state hook which is updated every time the map changes.

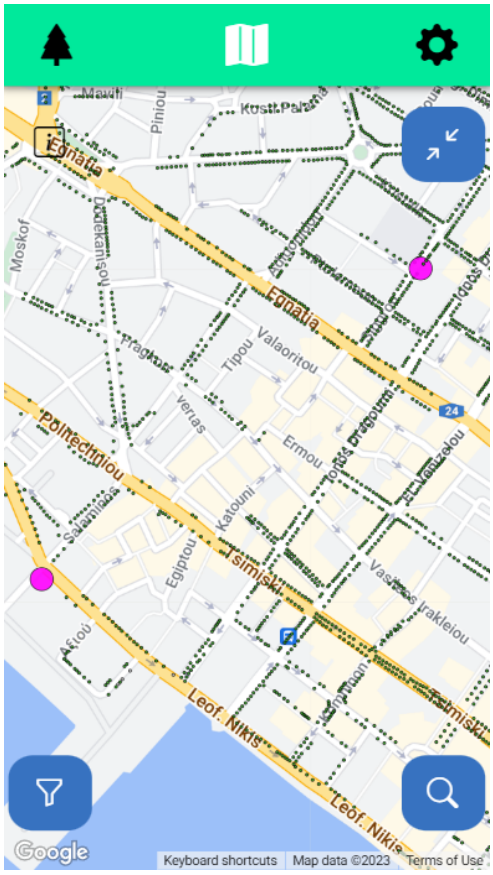


Image 39: Tree Scaling in Map

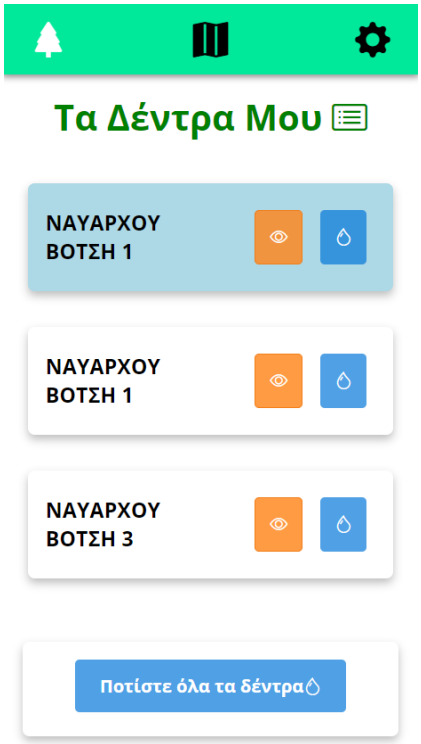
```
//const used for scaling the trees based on the zoom level
const refZoom = 20;

//calculates the radius of the trees based on the zoom level
const calculatePointRadius = useCallback(
  (owner) => {
    // Clear the existing overlay
    if (deckOverlay) {
      deckOverlay.finalize();
    }
    // If the user is the owner of the tree, calculate the radius based on the zoom level
    if (owner[0] === userId) {
      return Math.max(2, Math.pow(2, refZoom - currentZoom));
    } else {
      return 2;
    }
  },
  [currentZoom]
);
```

Image 40: calculatePointRadius function

Tree Watering Color

As an indicator to the users that they have watered their tree, the color of the tree card has been updated to display a shade of blue. This helps users by visually distinguishing the trees that have been watered and the trees that have not. Currently, the blue color is attributed to trees

that have been watered in the past two days but the implementation allows for modifications by simply modifying the values used in comparison to the *dayDifferenceSinceLastWatered* variable.




```
if (lastWateredReceived != null) {
  // Parse dates since last watered
  let today = new Date();
  let lastWatered = new Date(lastWateredReceived);
  // Fetches the time according to year, month and day
  const timeLast = lastWatered.getTime();
  const timeToday = today.getTime();
  // Date difference algorithm
  let dayDifferenceSinceLastWatered = Math.floor(
    (timeLast - timeToday) / (24 * 3600 * 1000)
  );
  // Increase the day by 1 as it counts today as well and make the number positive
  //(unless its 0 where it becomes -0).
  dayDifferenceSinceLastWatered = (dayDifferenceSinceLastWatered + 1) * -1;

  // Display last watered on UI
  if (dayDifferenceSinceLastWatered === -0) {
    wateringStatus = greekLabels.TODAY;
    statusColor = "lightblue";
  } else if (dayDifferenceSinceLastWatered === 1) {
    wateringStatus = greekLabels.YESTERDAY;
    statusColor = "lightblue";
  } else {
    // Reverse negative number to positive
    wateringStatus = dayDifferenceSinceLastWatered + greekLabels.DAYS_BEFORE;
    statusColor = "white";
  }
} else {
  wateringStatus = greekLabels.NOT_WATERED;
  statusColor = "white";
}
```

Image 41,42: Changing the color of the tree card

6. Testing

The functionality of the application was manually tested when the product was obtained by the previous developers. When the requirements were prioritized, adding functionality and resolving existing issues were given more priority than the addition of automated testing. Nevertheless, this section describes the testing that was performed and gives instructions on how to proceed in regard to manually testing the application.

6.1 Manual testing

This subsection presents important scenarios that need to be tested. These functionalities are vital to the experience and performance of the application, as such they should work as intended.

Scenario 1: Registering
Step 1: Provide a valid email address and 8 character password in the Registration page
Step 2: Clicking the register button, an email should be sent to the provided email containing the confirmation code
Step 3: Copy the unique code and enter it into the code verification page

Success: This scenario is successful and the user's account is successfully created. The creation of the account can be verified by querying the *user* table in the database to examine the account that has been created and verified.

Scenario 2: Logging In
Step 1: Having successfully performed Scenario 1, visit the Login page
Step 2: Provide correct credentials and press the login button
Success: The user should be redirected to *'my trees'* page and a userData entry should have been created in the localstorage (Inspect tools → Application → Local Storage). This entry contains information about the user and the session.

Scenario 3: Simultaneous tree adoption
Step 1: Log In
Step 2: Visit the tree map page
Step 3: Open another instance of the application (can be done through another browser) and complete steps 1 and 2 again for a different account.
Step 4: Locate a tree on both of the opened instances and try to adopt the same tree at the same time.
Success: One of the accounts should succeed in adopting the tree while the other account should receive an error message saying that the tree has already been adopted.

Scenario 4: Adding a New Tree
Step 1: Navigate to the map page
Step 2: Select a tree to adopt
Step 3: Click the adopt button
Step 4: Verify that the new tree has been added to the my tree page
Success: The new tree should be visible on the map and my tree page.

Scenario 5: Deleting Existing Tree
Step 1: Navigate to my tree page
Step 2: Click the show tree location
Step 3: Click on the tree
Step 4: Click the delete button *(Αφαίρεση από τα δέντρα μου)*
Success: The tree has been deleted from 'my tree' page and removed from the map too

Scenario 6: Global Filter trees
Step 1: Navigate to the settings page
Step 1: Switch on the global filter
Step 2: Navigate to the tree map page
Step 3: Click on the filter button
Step 4: Select one of the three filter options
Success: According to the selection of the filter, it is displayed in the map. Moreover, this filter option remains the same, whatever other action the user does.

Scenario 7: Water one tree / water all trees
Step 1: Navigate to my trees page

Step 2: Click the water droplet button on the tree you want to water if there are more than 1
Success: Tree card turns blue to indicate it has been watered recently
Step 3: Click the water all trees button at the bottom of the page
Success: All three cards turns blue for the same reason

Scenario 8: Zoom to next tree
Step 1: Navigate to my trees page
Step 2: Click the zoom-to-all-trees button at the top right (multiple times)
Success: Screen zooms according to the number of trees adopted. Initially, it zooms into the first tree, then to the rest one by one. After reaching the maximum number of adopted trees, with the next click it loops back to the first tree resetting the implemented counter.

Scenario 9: Search with ZIP code
Step 1: Navigate to the map page
Step 2: Click the search button in bottom right corner
Step 3: Enter a valid 5 digit zip code and click approve
Success: The screen automatically goes to the specific ZIP code location

Scenario 10: Change password
Step 1: Log in to your account
Step 2: Navigate to the account settings page
Step 3: Click the reset password option
Step 4: Open your email and click the provided link
Step 5: Enter the new password
Success: The password is successfully updated and the user can log in using the new selected password.

7. Project Setup Guide

This section covers instructions on how to set up the project. Most of these instructions were already documented by the second group and they remain unchanged so they remain relevant and useful. This section covers:

- Local Database Installation
- Project Installation & Execution
- Project Execution
- Remote Web App Deployment to cPanel

Note: Topics such as Progressive Web App (PWA) Creation and Google Play Store Deployment were outside the scope of this project and they have already been covered extensively by the documentation presented by the second group which can be found inside the docs folder.

7.1 Local Database Installation

The MoT project uses a MySQL database which needs to be installed in the local machine for the application to run locally. This guide assumes that the latest SQL file which is used to import the SQL database is readily available. This file is located in the “database backup” folder inside “docs”. Alternatively, the SQL file can be exported from the existing database deployed at cPanel. This guide uses the AMPPS Stack application to facilitate the process of connecting the database to the backend of the project.

1. Download and install the tool AMPPS from https://ampps.com/.
2. Start the AMPPS program and press the home icon.
3. Press on “phpMyAdmin” under “Database Tools”.
4. Login using the default credentials: username: “root” and password: “mysql”.
5. On the left-hand side, create a new database with “New”. Enter the database name as “mot_trees” and press the create button.
6. Select the newly created “mot_trees” database on the left-hand side.
7. On the top center of the page, select the “Import” tab.
8. Import the SQL file, which contains the database schema and data.
9. Under “mot_trees”, select “Events” and ensure the event scheduler is enabled. Two events should exist which are responsible for automatically deleting password reset attempts and registration code verifications after 15 minutes.

After successfully setting up the database environment, simply execute AMPPS and enable “Apache” and “MySQL” to start the database. The next time the backend executes, it should successfully connect to the locally running database with the default values. Should any issues occur during connection, the backend environment variables can be modified with the correct database credentials.

7.2 Project Setup & Installation

This subsection describes the steps needed to initially set up the project in a local machine. This guide assumes that the project has been cloned from the Github repository. In addition, the NodeJS and NPM (Node Package Manager) should already be installed. The recommended Node version is “v16.15.0”. Moreover, it is assumed that the local database has been installed and set up correctly, if not refer to the previous section. To set up the project follow these steps:

1. Open a terminal and navigate to the backend folder.
2. Run “npm install” on the backend folder.

3. Use the terminal to navigate to the frontend folder.

4. Run “npm install” on the frontend folder.

These commands should successfully install the necessary packages and dependencies. A list of the packages and their versions can be found in the package.json files.

7.3 Project Execution

Having successfully set up the project and the database in a local machine, the application can be executed by separately executing the frontend and backend folders.

To execute the frontend:

1. Open a new terminal at the frontend folder as a directory.
2. Run the command “npm run start”.

To execute the backend:

1. Open a new terminal at the backend folder as a directory.
2. Run only one of the following commands.
   a. If you are using the Windows operating system:
      i. “npm run dev_win” (developers)
      ii. “npm run start_win”

   b. If you are using a UNIX based operating system
      i. “npm run dev_unix” (developers)
      ii. “npm run start_unix”

Note: The “nodemon” NPM dependency has been installed in the backend. For the convenience of the developer, this library automatically restarts the execution of the backend on code changes. To execute with nodemon, use the commands that include the “dev” keyword. When running on localhost, the frontend can be accessed at “localhost:3000” and the backend at “localhost:5000”.

Alternatively, a `run_project.sh` shell script is provided (compatible with Linux/Mac OS systems) which will run the projects for you both in a single terminal window. Steps to run the script:

1. Open a terminal window in the project’s root directory (where the frontend and backend folders are)
2. Drag the script on the terminal, which will input the absolute path (or manually get the path through the CLI). Press Enter.
   a. In case you get a permissions error run “chmod 777 pathToRunScript” to authorize the script to run.
   b. In case something else is failing, please check the files `.env.development` and `nodemon.json` in the backend. These hold values used for the development

running stage, which you might need to modify to make the project run. An example can be a used port or a different password to the local database.

7.4 Remote Web App Deployment

This section provides detailed instructions on how to deploy the application in the web hosting platform, cPanel. These instructions were provided by the second group, however they remain relevant and have been included in this documentation for the convenience of the developers. To deploy the application, the first step is to build a production-ready state application.

7.4.1 Project Building

To successfully build the application, the command “npm install” must be executed for both the frontend and backend subprojects (refer to steps in section 7.2). If the dependencies have already been installed, the project can be built using the following steps:

To build the web application:

1. Run the command “npm run build” at the frontend.
2. Inside the frontend, copy all the contents of the “build” folder.
3. Inside the backend, delete all files on the “public” folder.
4. Inside the backend, paste the previously copied files from “build” into the frontends “public” folder.

After completing the listed steps, the backend can be executed with “npm run start_win” or “npm run start_unix” to ensure that the built web application works as expected, encompassing the frontend. Overall, the backend project folder has become the final built web application as it contains both the frontend and the backend folders.

7.4.2 cPanel Installation & Deployment

Having completed the process of building the application (described in the previous subsection), the application can be installed or deployed to cPanel using the guides provided in this subsection.

Note: When performing a production deployment to cPanel, cPanel will automatically execute the Node JS web application in a production environment. For this reason, it is unnecessary to change any environment values manually in code. As of this documentation, the default production values are satisfactory for deployment unless otherwise desired. The environment variables for the backend can be found in the files “.env.deployment” and “.env.production”. The environment variables for the frontend can be found in “.env” and “.env.production”.

cPanel Credentials:

- Username: *adoptatree*

- Password: *a569dof450!*

cPanel Initial Setup:

- Node: v16.15.1
- NPM: 8.11.0

cPanel Access:

- Domain: http://adoptatree.york.citycollege.eu/
- Web Application Management & Tools: https://192.232.236.47:2083/
- SSH: 192.232.236.47:22122 (Use cPanel credentials)

To install the built web application to cPanel:

1. Create a new compressed zip file containing all of the files of the built web application (the contents of the backend folder). For future reference, the zip file shall be named “Compressed-Web-Application.zip”.
2. Log into cPanel using the credentials provided above.
3. In the “Files” category, enter into the “File Manager”.
4. From the root directory “/home/adoptatree”, enter into the “nodejsapp” folder.
5. If necessary, delete all files present in the “nodejsapp” folder, this can be done by pressing “Select All” and then “Delete”. Preferably, enable the “Skip the trash and permanently delete the files” checkbox to delete the existing files immediately.
6. Within the same folder as in step 4, press the “Upload” button. Use the user interface to upload the zip file created in step 1 (“Compressed-Web-Application.zip”). Once complete, return to the “nodejsapp” folder.
7. You should be able to extract the zip file uploaded by right-clicking on the file using the cPanel file manager and “Extract” into the same folder. The entirety of the built web application source code should be visible inside “nodejsapp”.

To deploy the web application on cPanel:

1. Log into cPanel using the credentials provided above.
2. In the “Software” category, enter into the “Application Manager”.
    a. In the event an existing application already exists:
        i. Re-enable the web application through a slider under the “Status” column.
        ii. Press the “Ensure Dependencies” button to reinstall NPM dependencies.
    b. In the event an existing application does not exist:
        i. Press “Register Application”.

ii. Provide an application name.

iii. Select the domain for the web application, the domain value should be “adoptatree.york.citycollege.eu”.

iv. Set the application path to “nodejsapp”. The path will specify where the web application source code is located.

v. Select the deployment environment as “Production”.

7.4.3 Database Deployment

This subsection provides instructions on how to deploy or update the remote MySQL database solution used on cPanel.

To deploy a new version of a database to cPanel

1. Log into cPanel using the credentials provided above.
2. In the “Databases” category, enter into the “phpMyAdmin”.
3. In contrast to local development, the database in cPanel is named “adoptatree_db”. This is the database which will be manipulated. Ensure to select this database on the left-hand side.
4. Select all tables with the checkbox “Check all” and drop all tables using the “With Selected” select field. When dropping all tables, disable “Enable Foreign Key Checks".
5. With the same database selected, select the “Import” tab.
6. Import the new database scheme from an SQL file. If an error occurs during importing regarding events and user privileges, this is normal and can be ignored, the issue occurs as cPanel does not allow the use of custom database events.

To update or deploy a new database event to cPanel:

1. Log into cPanel using the credentials provided above.
2. In the “Files” category, enter into the “File Manager”.
3. Enter into the directory “/home/adoptatree/cron_job”.
4. Insert or adopt any script files with execute SQL statements. For example, a “password_reset_script.sh” script file can be found, this file is executed every 15 minutes to execute SQL to delete old password reset requests from the database.
5. Return to the cPanel tools page, in the ”Advanced” category, enter into the “Cron Jobs”.
6. From here, create or modify any existing cron jobs to execute any existing or new script files added in “/home/adoptatree/cron_job” after a specified amount of time.

8. Evaluation

This section briefly discusses any significant problems encountered or work that was not completed on time. This information should help in identifying which features are incomplete or problems that need to be fixed by succeeding groups. Lastly, suggestions and ideas will be mentioned.

8.1 Known issues

- Some component files are overloaded with code and logic which makes the files hard to read (especially the MapDeckGl.js file and files relating to tree components). Ideally, all logic should be moved to external and util files. An attempt has already been made to decluster the tree component files through the creation of the TreeUtil folder.

- Circular Dependency Warnings (Currently Suppressed) - Circular Dependency Warnings in React occur when two or more components or modules depend on each other in a way that creates an infinite loop. This creates a circular reference, and can cause issues in the application. Circular dependency warnings can result in reduced performance and unexpected behavior, among other issues. They can also be challenging to detect and debug since the circular reference might not always be obvious.

- Multi-language functionality - This group encountered some difficulty in working with an interface that contained labels that were not in English. The *language.js* file was created to ensure that all the code and all the labels across all files appears in English. The only files that contain Greek labels should be in these language files and then imported from there (there is one in the frontend and one in the backend). Although this helps with future implementations of multi-language functionality, it is not a complete solution. Creating a Multi-language functionality can be achieved in several ways. One common approach is to use a localization library like **i18n** or **react-intl**.

- cPanel - An issue that caused a lot of complications is how cPanel uses Node dependencies. This results in built-in modules such as “cluster” working locally, but requiring “node:cluster” syntax to work on cPanel. This issue extends to libraries that use these modules, specifically the rate-limiter-flexible module RateLimitCluster requires access to the cluster module. So, for it to work on cPanel, you must go to the node_module folder, find the rate-limiter-flexible folder, inside the lib folder a file named RateLimitCluster can be found. On line 22, the syntax must be updated to “node:cluster” for it to work on cPanel. The exact reason for cPanel working this way is unknown. Further investigation into this issue is recommended, though after migrating to a dedicated server it is unlikely you will encounter this issue.

- ESLint warnings - ESLint warnings are messages generated by the ESLint tool to alert developers about potential issues in their JavaScript code. ESLint is a static code analysis tool that helps developers identify problematic patterns or potential errors in

their codebase. Although most of the ESLint warnings have been resolved, there still remain some warnings that should be considered (mostly regarding React hooks).

- Slow Tree Scaling - Currently adopted trees are scaled on the map based on the zoom level of the map. As this solution waits for the map to update and then calculates the size of the tree point, the scaling does not appear to be that smooth.

8.2 Unfinished requirements

- Admin responds to requests - The current communication system between administrators and users within the platform is limited, as it only allows for indirect communication through a general contact page. To address this, we need to design and implement a messaging feature that allows direct communication between administrators and users. This feature could be integrated within the report page, enabling administrators to request additional information from users, provide support, or resolve issues.

- Add toast notification to notify users that they have successfully watered a tree - This task requires little effort as the code for the toast notifications is already available and being used to show notifications when users report a tree.

- Report sorting, searching and pagination - To make the report system more user-friendly, create an interface that resembles an email inbox. Use different fields for sorting and searching, and implement a pagination system that displays a limited number of reports per page with easy navigation. This approach is similar to what the users requested and will help to improve the overall usability of the report system.

- Resolved reports deleted or archived for a time period - The issue with the reports page is that it starts to create confusion when there are a lot of reports incoming or stored there. In the same page there are resolved and unresolved reports which is not the optimal state of this page. To improve the functionality, there should be a consideration of the amount of incoming reports and how they are handled. To address this, a solution needs to be implemented for automatically archiving or deleting resolved reports after a certain period of time. This would ensure that the page only displays unresolved reports, making it easier for the admin to manage them. Meanwhile, the other page where all the resolved reports are, can be used as the admin archive to go back and look at details. The archiving or deletion process could be based on a set time period or a threshold number of reports. This system should be designed to efficiently store and retrieve archived reports if necessary.

- Users upload a photo when reporting a tree problem - To enhance the reporting of tree problems, implement an upload photo button that allows users to upload photos and associate them with specific reports. Moderation and filtering tools should be considered to prevent inappropriate or spammy photos. To ensure secure and efficient storage of the photos, backup and recovery measures must be put in place. This feature will allow

users to provide visual evidence of tree problems, improving the accuracy of the reports and facilitating more effective responses from staff.

- View other people's adopted trees - This functionality was started and was almost completed but after the UI redesigns instructions, it was decided that for now users should not be able to interact with gray trees (trees adopted by other users). For this reason, this functionality was abandoned.

- Make the adoption of a tree satisfying/ceremonial - The implementation of the confetti thrower was ready to be merged with the rest of the code but some issues canceled this functionality from being completed. After adopting any tree, confetti was thrown to the interface of the user to make it more satisfying, but there were two problems. First, confetti made the screen sluggish to the point it could freeze for a second if the *<Dialog>* stayed open for too long. Even though this functionality looked fine in essence, after running tests in GitHub, Frontend CI showed errors, presumably from the library imported *(ReactConfetti from 'react-confetti').*

- Track the weather and implement it into a watering notifications algorithm - To elevate the watering system of the application, an algorithm could be implemented that could track rainy days to automatically update the status of the trees and mark them as watered. It could prove to be challenging but not impossible. Ideally, the weather algorithm will require an external API to send automated notifications for each tree type.

- Add watering requirements per tree type - There are more than 40 different tree types in the application's database and every tree type has different watering needs. The idea was to provide this information to users so that they know how often they need to water their trees. There already is an informational ("i") icon next to the tree type in the tree modals that were designed for mobile users, which could be used to display this information when clicked. A list of all the tree types was sent to the client so that they could provide the watering requirements, however they did not respond.

- Introduce user level system where maximum number of adoptable trees is linked to user level - In relation to the Gamification, if a user surpasses a certain level, they get rewards. One of them could be to increase the limit for the number of trees they can adopt.

- Admin Statistics Page - This page should display statistics about the application as explained in the requirements section. Different types of charts with different characteristics could be placed here.

- Admin Hierarchy - Some administrator privileges should be restricted to prevent misuse. One idea is to create a role between admin and user, named *moderator,* which must have only general permissions.

- Fix password resetting cache problems / Users upload a photo to customize their tree / Comment on trees / Gamification (Points, Prizes) /Competition/Leaderboards / Share trees between people - these are all requirements explained in the first section and by

the previous groups. Due to time limits, no attempt has been made to develop these functionalities so they remain as unfinished requirements.