Table of Contents

Final Documentation

The aim of the document is to highlight the development efforts of the second team responsible for further developing the MoT Trees project, as well as guide future developers for continued development. The team consisted of Philippos Andrianos Kalatzis, Nikolaos Lintas, Ioannis Angelos Chaidemenos and George Tamvakas. The document will present the project's requirements, analysis, designs, implementations, and testing efforts made for readers to obtain a holistic view of the project's progress. For ease of continued development, appropriate instructions will be provided for the installation, execution, and deployment of all the projects artifacts.

Requirements

For the purposes of grasping the progression of the project, both the initial requirements left from the former development team and current requirements worked on will be highlighted. For all requirements, their completion status will be marked in a tabular format in order to clarify the remaining work that needs to be performed.

The previous team had successfully completed the following requirements:

1. Login/Register page
2. View tree details
3. View my trees
4. Mobile and Desktop support
5. View all trees on a map
6. Make a dynamic interactable map
7. Adopt a tree
8. Water a tree
9. Limit max adopted trees
10. Distinguish between adopted trees, my adopted trees and not adopted trees.
11. Un-adopt a tree
12. Add mobile app support (PWA)
13. Deploy on Heroku

The previous team then highlighted the following incomplete requirements, which became the project's starting requirements to base future efforts on:

1. Distinguish age between trees on the map (was claimed to work, but wasn't actually)
2. Report a tree's issues
3. Landing Page
4. Admin Page
5. Revoke tree ownership

6. Admin responds to requests
7. Make the adoption of a tree satisfying/ceremonial
8. Inform admins that rubbish is removed from a tree
9. Contact us page
10. Share trees between people
11. Add a custom name to trees
12. Add notifications
13. Promote users to admins
14. Track the weather and consider it into the watering algorithm
15. Add watering requirements per tree type
16. Gamification (Points, Prizes)
17. Competition/Leaderboards
18. Regional Leaderboards
19. View other people's adopted trees
20. Comment on trees

The following table below depicts the status of all requirements worked on during this project, indicating their completion. Requirements will not only include unfinished ones from the previous project, but also derived requirements. The available statuses are: 'Done', 'Started' and '-' (signifying has not been worked on). The 'Story ID' column matches with the ID of the sprint backlog user stories. The 'List ID' refers to the number of the requirements in the list mentioned prior with the remaining requirements.

Story ID	**List ID**	**Functional Requirements**	**Status**
S31	1	Distinguish age between trees on the map	-
S9, S11, S12, S13	2	Report a tree's issues. Reports have a status, and admin's can dynamically change statuses of reports.	Done
S7	3	Landing Page	Done
S1, S8, S10, S11, S12, S13, S30	4	Admin Dashboard Page	Started
S10	5	Revoke Tree Ownership (Block Users)	Done
S11, S12	6	Admin responds to requests	Started

-	7	Make the adoption of a tree satisfying/ceremonial	-
-	8	Inform admins that rubbish is removed from a tree	-
S14	9	Contact us page	Done
-	10	Share trees between people	Started
-	11	Add a custom name to trees	-
-	12	Notifications (inside web app and email)	Done
S1	13	Promote users to admins	Done
-	14	Track the weather and consider it into the watering algorithm	-
-	15	Add watering requirements per tree type	-
-	16	Gamification (Points, Prizes)	-
-	17	Competition/Leaderboards	-
-	18	Regional Leaderboards	-
-	19	View other people’s adopted trees	-
-	20	Comment on trees	-
S15	-	Deploy to Google Play	Done
-	-	Deploy to cPanel	Done
-	-	Migrate from MongoDB to MySQL	Done
-	-	Decouple frontend from database layer (Use response objects/stable key value pairs)	Done
-	-	Decouple database from controller code in the backend	Done
-	-	Constrain CSS rules so that they do not produce side effects in other pages in the frontend	Done
S22		Users upload a photo when reporting a tree problem	-
S21	-	Users upload a photo to customize their tree	-
-	-	Protect access to API endpoints	Done
S24, S15	-	Change email and password	Done

S28	-	Multi-language support (look here for details of progress)	Started
S29	-	Admin username at navigation bar	Done
S25, S26	4	Report pagination, sorting and searching (look here for details of progress)	Started

There is also a smaller amount of non-functional requirements that were addressed in this project.

Non-Functional Requirements	**Status**
Improve user experience (UX)	Done
Change Colour Theme (yellow - grey => green - white)	Done
Unify UI Design (components are similar in all pages)	-
Enhance User State Awareness (in web app notifications)	Done
Add Iconography (improves reusability - learning in UX and overall design)	Done

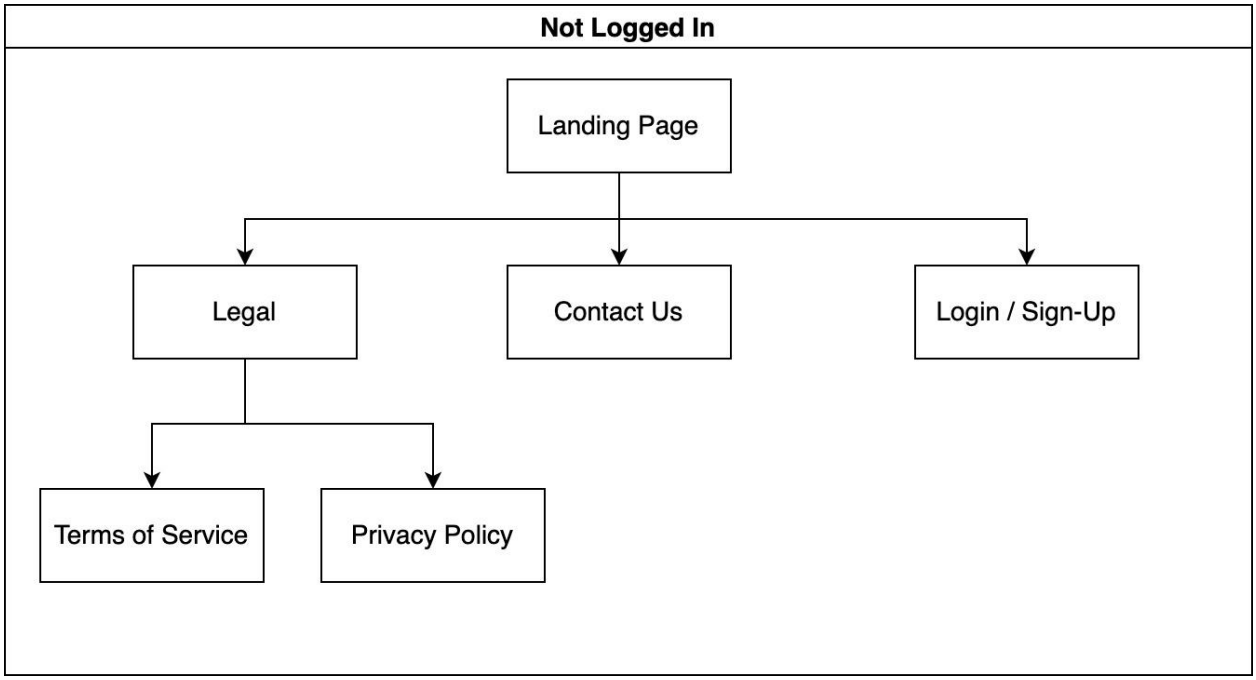
For more information regarding unfinished or partially finished requirements, please go to the Evaluation section in this document, which should help highlight key issues. Any issues or pieces of advice that can help future development resume faster can be found there.

Analysis

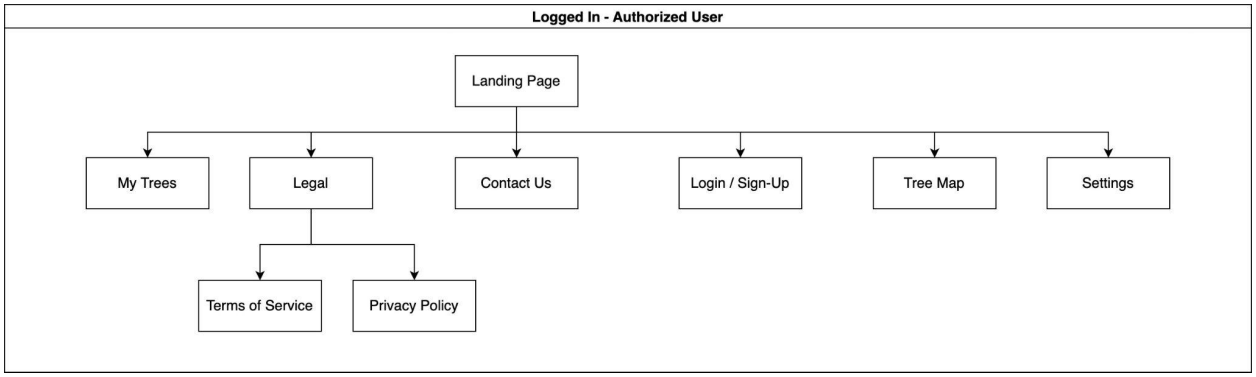
This section contains the entirety of analysis performed on occasions before proper designs or implementations were made. Notably, the creation of sitemaps, mockup UIs, and database requirements analysis was performed.

Sitemap

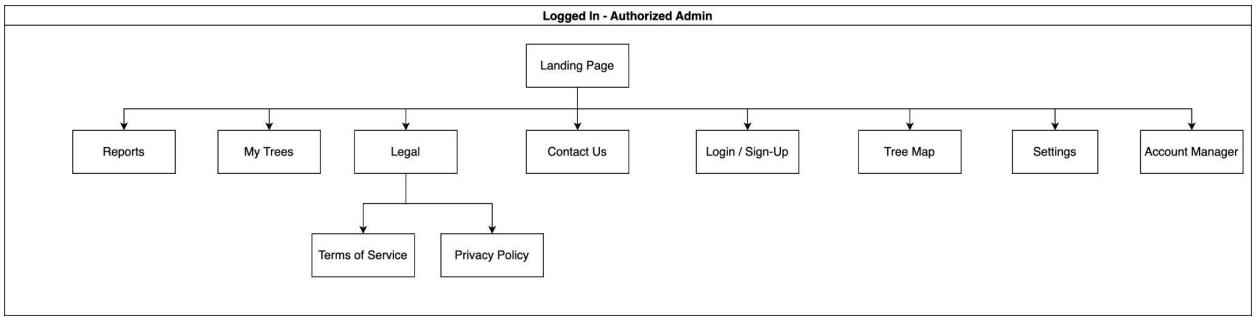
The sitemap of the web app is available in three distinct versions. Firstly, when a user is not logged in they cannot use the services offered, but they can contact the municipality for more information on the project, they can view the landing page and any legal obligations.



Once users register or login, they gain access to the main functionalities of the web application. These are the trees map where they can adopt and view all trees in Thessaloniki and my trees where they can make reports on a tree, locate the tree, water a tree or un-adopt (delete) a tree. Additionally, they can change their accounts email or password in their settings page.



If the users are promoted to an administrator, then they gain access to the account manager and user reports pages. In the account manager, they can block misbehaving users and promote other accounts to administrators. In the user reports, they gain access to all reports, and they can assign themselves to solve the issue or mark a report as resolved in the web application.

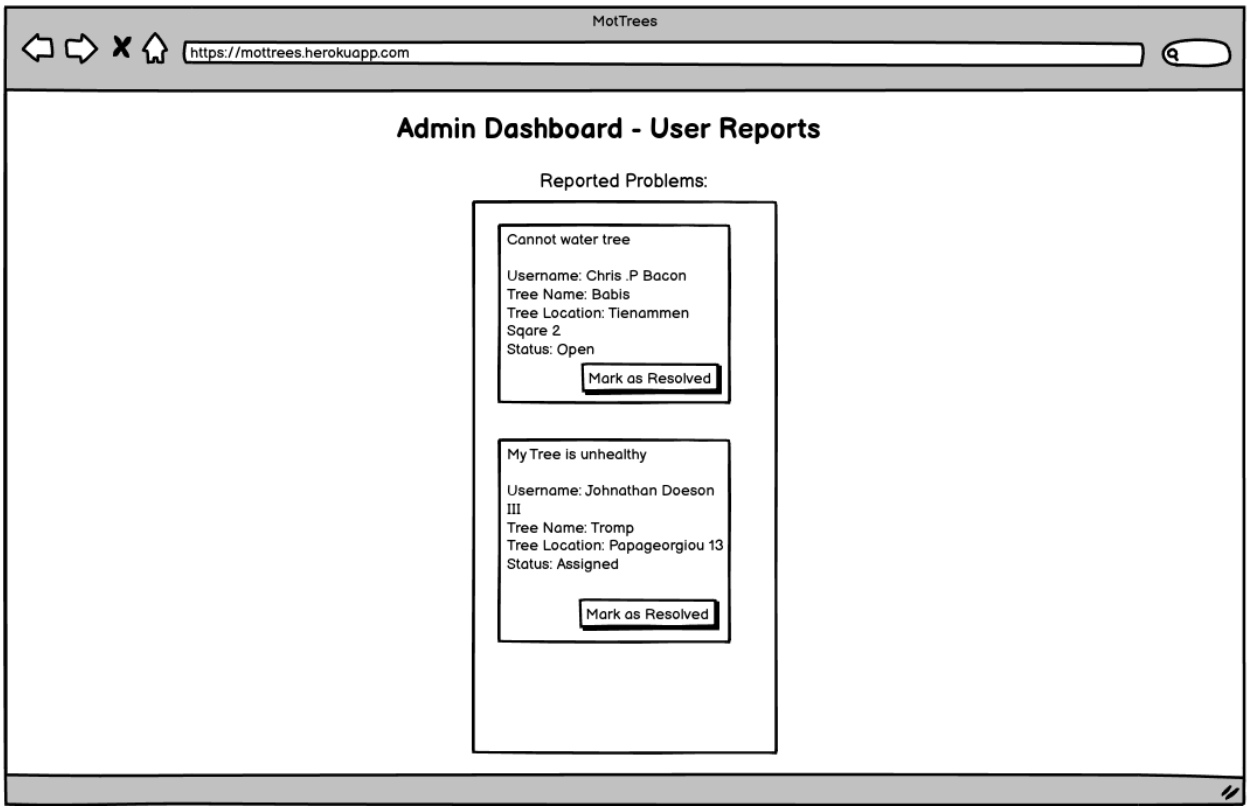


UI Mockups

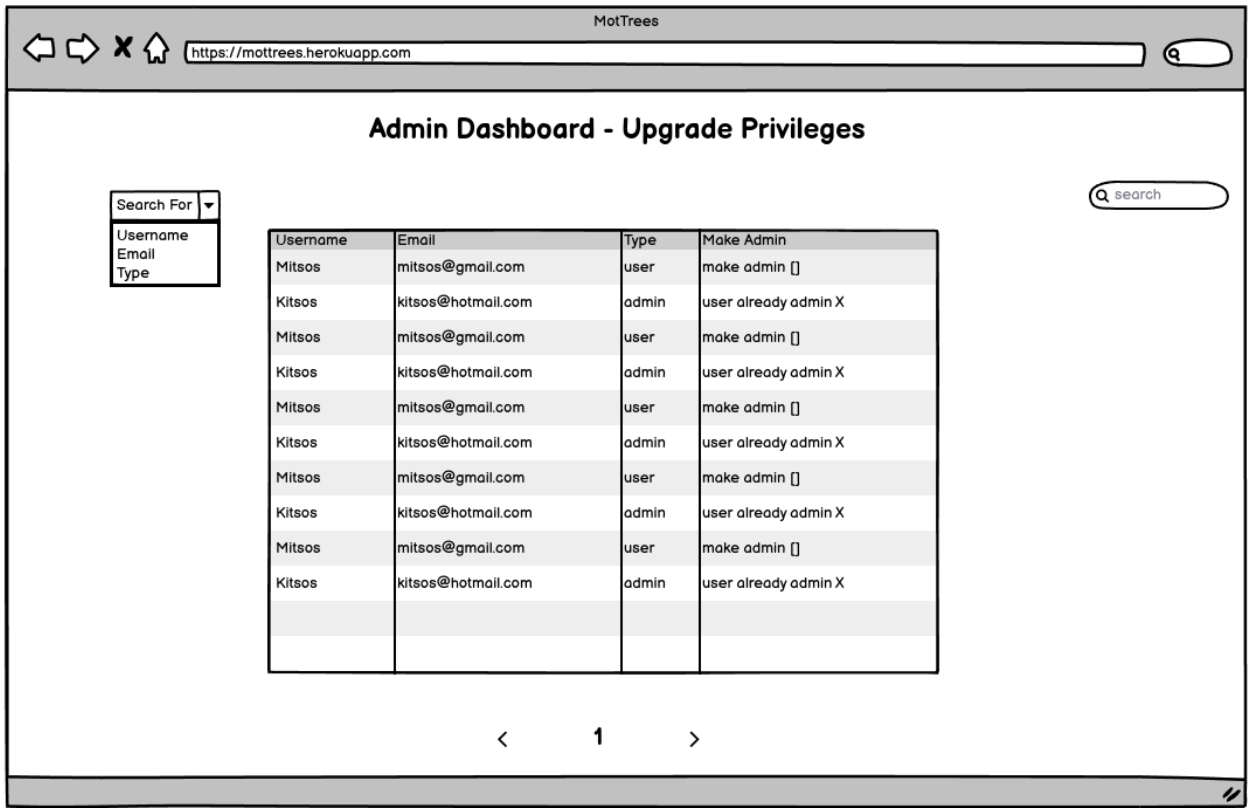
User Interface mockups are simple visual representations of a specific requirement. They provide a visualization of how a specific interface should look but without many details like fonts, colouring and other complex stylistic elements. They can potentially be very helpful for the developers for two reasons:

- They allow developers to properly communicate with the clients about how something should look and more easily understand what the clients expect in requirements.
- They can save time as clients can see a mockup made quickly using tools like Balsamiq, and agree or disagree with designs and propose changes. As a result, less unexpected changes can occur as a surprise in the future.
- Mockups prove a good way to communicate between backend developers and frontend developers, indicating what the frontend is creating and the capabilities that should be supported. This allows developers to work at their own pace without the need of synchronization where one team depends on another to finish first.

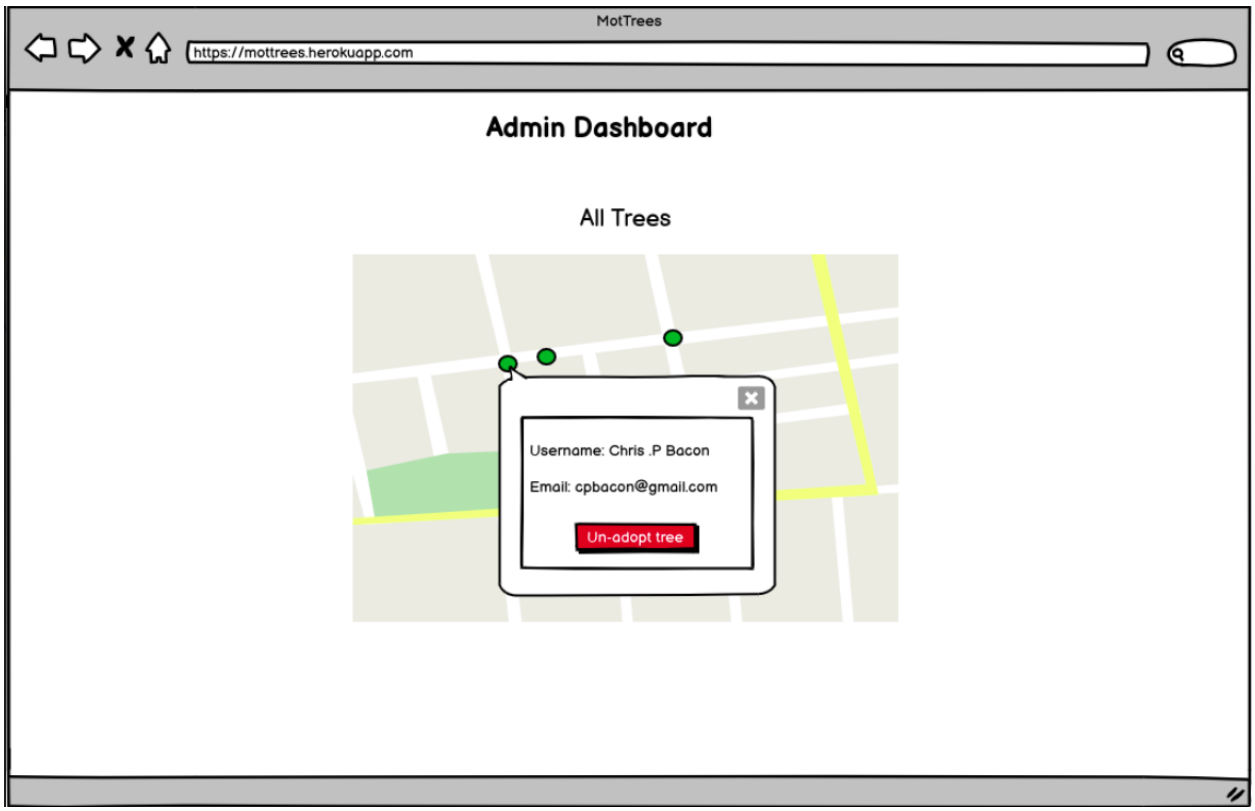
The mockups that were created for the purposes of developing this project can be seen below.



Admin Dashboard - List of user reports



Admin Dashboard - List of users with privileges and ability to upgrade/downgrade them.



Admin Dashboard - All trees with owner details

Database

This section will perform a basic analysis of requirements as to guide the future designs and implementations of a new SQL based database. Requirements will be specified as to abstractly clarify the data and responsibilities the database is expected to fulfill. The requirements of the current database solution were extracted from a prior (NoSQL) database implementation, as well as being derived from user requirements. As of this version of documentation, the database must fulfill the following requirements:

- **Contain all trees in Thessaloniki:** The database must store all individual trees identified in Thessaloniki with information. The most important information stored for each tree is their exact latitude and longitude coordinates and adoption status from other users.
- **Maintain a list of users:** The registration, authentication, and management of user accounts must be possible.

- **Facilitate role based authorisation:** Role based logic should be used for the purposes of controlling user authorisation. Every user will be associated with one or more roles, indicating their permitted actions.
- **Facilitate the management of user reports:** User created reports should be stored, which must be resolved by an administrator. Each report should specify an issue with a specific tree.
- **Aid in the process of resetting user passwords:** The database should facilitate the process of users resetting their passwords.

From the previous requirements and findings, the exact data requirements that should be stored are:

- **User:** Email, Password, Blocked Status (Boolean).
- **Role:** Name.
- **Tree:** Name, Needs Watering (Boolean), is Alive (Boolean), Type, Address, Latitude, Longitude, Zip Code, Last Watered Date, Creation Date.
- **Report:** Title, is Resolved (Boolean), is Active (Boolean), Description, Creator ID, Admin ID, Tree ID, Creation Date.
- **Password Resetting:** Hashed ID, User ID, Creation Date.

Design

This section will cover all design efforts made before implementations began, specifically, the design of the SQL based database used and separations of concerns in the backend are discussed. Additionally, some topics were researched before implementation for packages or in terminology. This came useful when designing some implementational elements and understanding deployment terminologies mentioned from the previous team and relevant solutions.

Researched Topics

In an effort to provide more context and understanding of used technologies or concepts, this section covers a variety of topics or technologies by providing brief explanations within subsections. Most notably, the use of Sequelize as a technology and Android PWA creation tools are covered. Additionally, the multi-language localization tools are briefly discussed.

Sequelize

Sequelize is an Object–Relational Mapping (ORM) tool which provides layers of abstraction when communicating with SQL based databases. In the event the underlying database solution used is replaced or updated in schema, Sequelize provides convenience by ensuring transactions execute without major refactoring or implementations. Sequelize works primarily through the use of developer defined models, with each model representing a corresponding table in a database. Every model can also set its relations with other models (e.g. One-to-One, Many-to-Many etc.). For these reasons, the Sequelize library is used as the underlying implementation for database communication and utilization.

PWA

A PWA is an application which leverages multiple technologies and patterns to take advantage of both web and native mobile application features. Notably, web applications developed as PWAs (based on certain criteria or key principles) allow users to be able to “install” the web application on the home screen of their mobile devices via a shortcut provided by their web browsers. As a result, users are able to experience a native app-like experience without having to visit the Google Play Store or App Store. Importantly, a PWA is not a complete standalone application consisting of an entirety of a web application, but a proxy application which makes hardcoded calls to remote resources, remaining small in storage size. This is done via the use of “Service Workers” which are a set of Javascript implementations.

A PWA alone however is not enough for Google Play Store or App Store deployment. By using external libraries and tools, web applications built can be converted into a TWA which can be installed natively on mobile devices when listed on app stores. To achieve this, Bubblewrap and PWA-Builder can be used. The final applications can then be deployed on the Google Play Store or App Store, increasing the potential user base being covered.

PWA advantages include:

- Client side caching of web application data, enabling the use of an application offline, provided the data has already been downloaded before.
- Client side data is updated when changes are detected from the web application.
- A more integrated “Look and Feel” for mobile devices.
- Increased ease of use for notifications and push messages

Lighthouse

Lighthouse is an open source automated tool by Google which can perform quality verification of web applications or PWAs based on criteria. By running a series of audits, the tool generates reports indicating the quality and performance of a web application and its pages, as well as its conformance to being a PWA. Such a tool can aid in setting the bare-minimum of implementations needed to ensure that a web application is as compliant to being a “proper” PWA as possible.

TWA

TWA is a software solution which extends upon existing PWAs to improve user experience and application independence. TWA apps run from within user independent and integrated browsers which provide a “full screen” experience, hiding any UI browser elements. In effect, the illusion of a fully fledged mobile application is being presented to users, enhanced by the fact that TWAs can be installed from mobile app stores. Because of the nature of TWAs, such apps share similar benefits to PWAs, especially for data caching, native app experience, performance and more.

Bubblewrap

Bubblewrap is an open-source software solution which allows developers to create or convert existing Node.js based applications into PWA (Progressive Web Apps) for Android devices using TWA (Trusted Web Activity) as an underlying solution. When using Bubblewrap, the solution can help create the necessary files needed when performing the conversion into PWA or utilize any existing files made in preparation.

PWA-Builder

PWA-Builder is a web application founded by Microsoft that provides a convenient user-experience for the creation of PWAs using Bubblewrap as its core solution. Greater convenience is provided when creating and configuring existing PWAs in comparison to Bubblewrap and its dedicated CLI (Command Line Interface). Importantly however, PWA-Builder allows for the additional creation of both Windows and IOS applications with likewise ease of use.

Multi-Language Support

An effort was put to multilingual support, but it was not implemented in time. What the team identified were solutions in terms of dependencies, and a small explanation of them will be provided below.

Tolgee: It is best used when the client needs to enter translations directly to the data set. Stores translations on a REST API server which can be self-hosted, used with a free tier or paid tier with their servers (coming soon). The translations are retrieved using the TolgeeProvider component, which wraps around the main application (in index.js). Then, in the UI components, the useTranslate() hook provided can be used to access JSON from the API using the key from the key-value pairs. In order to switch language, the package provides the useSetLanguage() and useCurrentLanguage() hooks appropriately, which send a request to the remote specifying the new language.

I18next: The most suitable candidate, stores the language data in i18n.js which is a component with JSON configuration data. Here, we can specify the languages available and their translations. Translations are key value pairs where the key is either the value of the main language or custom key names. A fallback language can also be specified. In order to translate text, there are several ways according to the steps taken prior. Firstly, using the key in a <Trans> tag allows for a custom value to appear inside. Secondly, <Trans> tag standalone can be used if the key used for the text is the text from the primary language. Finally, a custom t{} wrapper can be used instead of <Trans> as described in the second instance.
Useful Links:

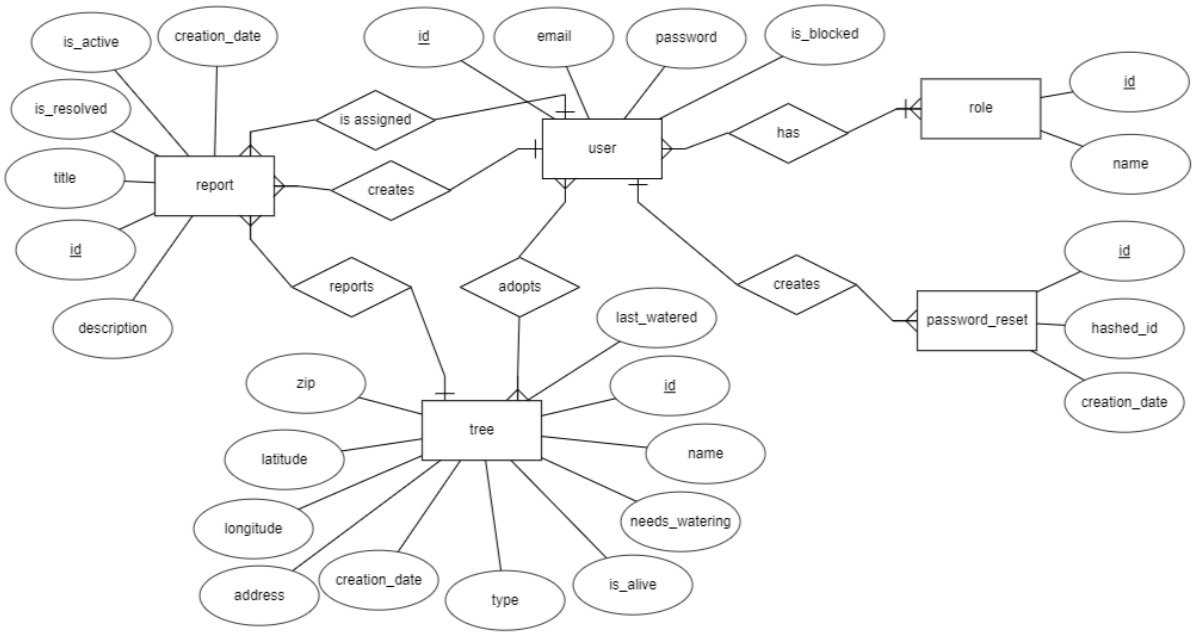
- Playground
- Guide 1
- Guide 2

Database

This subsection aims to discuss the design of an SQL based database for the project after a migration from a NoSQL solution. Brief justification will be provided for important decisions made, as well as indicate the intended use of the database. An ERD (Entity Relationship Diagram) diagram will be presented as a simple overview of the database design in structure and relations. Lastly, a more elaborate database schema will be presented with final attribute names and data types used. For an examination of backend specific implementations using the database, please refer to the Implementation section.

Entity Relationship Diagrams

Based on previous requirements and conclusions made during analysis, the ERD design can be presented as follows in the diagram below.



There are notably five tables: user, report, role, tree, and password reset. In the current design, every user is able to “adopt” a number of trees, with every tree able to be adopted by many users. Every user “has” one or more roles, each role can indicate if the user is a generic “user” or for example an “admin” for authorisation purposes. Reports are “created” from a single user and resolved by a single “assigned” administrator, each report, “reports” an issue for a specific tree. Upon request, every user “creates” a new password reset entry which will persist until it expires or is utilised until the user confirms their new password (via an email notification).

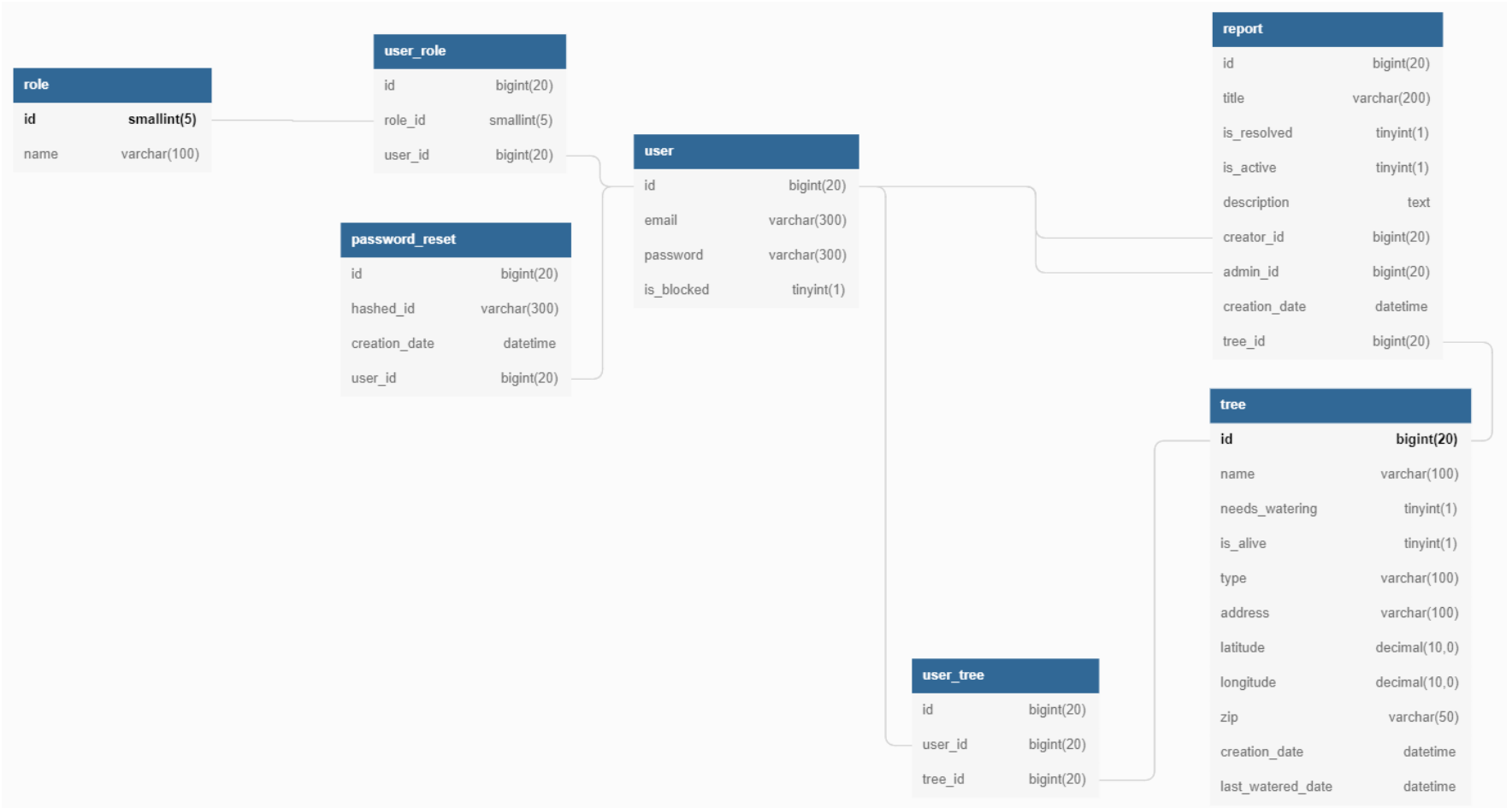
Schema Diagrams

Expanding upon the previous ERD section, the database schema can be seen in the diagram below. Notably, there are two associative tables: user_role, user_tree. The table “user_role” aids in associating users to one or more roles. The table “user_tree” aids in associating users with one or more trees when adopting.

Notable database design choices are:

- All primary keys (and consequently foreign keys) are unsigned and auto incrementing, as negative valued primary keys will never be used.
- The password of every user tuple must be hashed before insertion for bare-minimum security in the event of data breaches.
- All “creation_date” attributes are calculated automatically on tuple insertion for convenience, unless manually set.

- For the “password_reset” table, the database solution is expected to remove every tuple after a certain amount of time has elapsed from the creation date of each tuple. Tuples are not expected to persist forever. This is achieved through the use of a scheduled event in the database.


role
id smallint(5)
name varchar(100)
user_role
id bigint(20)
role_id smallint(5)
user_id bigint(20)
password_reset
id bigint(20)
hashed_id varchar(300)
creation_date datetime
user_id bigint(20)
user
id bigint(20)
email varchar(300)
password varchar(300)
is_blocked tinyint(1)
report
id bigint(20)
title varchar(200)
is_resolved tinyint(1)
is_active tinyint(1)
description text
creator_id bigint(20)
admin_id bigint(20)
creation_date datetime
tree_id bigint(20)
tree
id bigint(20)
name varchar(100)
needs_watering tinyint(1)
is_alive tinyint(1)
type varchar(100)
address varchar(100)
latitude decimal(10,0)
longitude decimal(10,0)
zip varchar(50)
creation_date datetime
last_watered_date datetime
user_tree
id bigint(20)
user_id bigint(20)
tree_id bigint(20)

MVC & Repository Pattern

When improving, and implementing features in the backend of the web application, the migration to a new SQL database was used as an opportunity to increase separation of concerns. Specifically, efforts were desired to be made to expand upon the Model-View-Controller pattern, as although controller implementations existed, proper Views and Models did not. At the same time, the existing codebase in the controllers were directly coupled to the implementations of the previous NoSQL database, providing little abstraction as only an Object Relation Mapping Tool was used. As a result, the repository pattern will be used to encapsulate the implementations of SQL databases from controller classes.

With these design plans, a number of repositories will be created to represent a resource of interest, namely: users, admins ,reports, and tree data. Each repository will provide a number of functions (migrated from existing NoSQL implementations) or based on project requirements. The resulting data of each repository function should be returned to controllers in the form of models which can be transformed into views or returned from controllers as a result (avoiding tight coupling). Naturally, each repository can be invoked on demand by controllers to acquire results.

Component Based Architecture

The frontend makes use of component-based architecture, an architectural paradigm that influences the structure of code. Components are basically atomic sets of frontend functionality which may contact an API (sendRequest), update the data (React states and context), structure the data (JSX) and present the data (JSX + React + CSS) all in a single structure. In addition, components are unit testable, hence the dependency with other components is removed. The level of encapsulation is controlled by the developer, but the most used is folder layer abstraction. In folder layer abstraction it is expected that each folder contains its main page and/or children pages, its styles (.css) either inside the page files or externally, its tests and any required helper/utility classes. In React, separating code into components is called composition. Our project structure follows this paradigm, as can be seen in the frontend structure section. Using this layered approach allows the application to have several additional advantages and disadvantages:

+ Abstractness

+ Reusability

+ Separation of concerns

+ Decoupling

+ Clarity in data and state passing between objects

+ Testability

+ Project cost decrease

- More LOC (Lines of Code)

- Restricted future use of code (must be designed abstractly and leverage external input)

Implementation

This section will highlight the most important implementations made for both the frontend and backend. The core concepts and components used in implementation will be discussed, indicating their expected use and function.

Backend

This subsection examines the most important implementations created in the backend. The core implementations of the backend will be mentioned, highlighting the uses of controllers, databases interaction, repositories, models, special implementations, and use of environment variables.

Controllers

Controllers are responsible for serving HTTP requests from the frontend through provided REST API endpoints. The endpoint design follows the following structure:

1. An endpoint method will be responsible for processing a single type of request according to its associated route. Routes are managed from the main file (app.js) and the associated files in the 'routes' directory. A route must be aware of:

    a. The HTTP method expected

    b. JWT token verification (if necessary)

    c. Actual URL endpoint where it expects requests at

    d. The relevant controller method that needs to be triggered when a request is received.

    An example of a route can be seen below:

```
1. //Un-adopt trees
2. router.patch("/:tid/delete", verifyToken);
3. router.patch("/:tid/delete",
   treesControllers.removeTreeAdoption);
```

2. Communication with the associated repository method/s occur. There is usually one method for performing the main operation of the endpoint and optionally one or more endpoints used for validation of provided parameters, authentication or authorization purposes. An example of a controller-to-repository interaction is visible below.

```
a. const deleteReport = async (req, res, next) => {
b.   const id = req.params.id;
c.
d.   //Authorisation
e.   if (req.roles.includes("admin")) {
f.     try {
g.       await ReportsRepository.delete(id);
h.     } catch (e) {
i.       const error = new HttpError(e, 500);
j.     }
k.     res.send(204);
l.   }
m.   else {
n.     //Unauthorised
o.     res.send(403)
p.   }
q. };
```

3. After validation and performance of the operation, either an expected result or exception will be received.

    a. On success, the operation's result will be sent back as JSON with a successful HTTP code.

    b. The controller will handle the received exception and send the appropriate HTTP responses to the resource requester. A message which obfuscates internal system design will be sent back along with a valid (operation failure) HTTP code.

Database

The database that was originally used for the web application was MongoDB which is a NoSQL database. From the start of the project, implementations made use of this existing solution, but in order to facilitate the deployment of the web application to cPanel a migration had to occur to MySQL. As a result, a complete rewrite of the backend took place by changing every method that was previously using mongoose, which is an ORM for MongoDB, and replacing it with Sequelize. Sequelize, which is also an ORM can be used with multiple relational databases like MySQL, PostgreSQL, and more, remaining a reliable choice. For the transition to be performed smoothly and to also facilitate future changes, various repository classes were created for this purpose, as mentioned previously. The database of the deployed application on CPanel can be viewed on phpMyAdmin in CPanel by visiting https://192.232.236.47:2083/ .

The credentials for logging in are:

- **Username**:

adoptatree
- **Password**:
  a569dof450!

After logging in go to **Databases -> phpMyAdmin.** The credentials for accessing the databases are:

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

In order to access the database locally, any tool can be used like phpMyAdmin, MySQL Workbench, TablePlus, etc. When running the application locally it is necessary to add certain environmental variables in order for the backend to be able to communicate with the database. The environment variables are explained in the section Environment Variables.

All interactions between the database and backend happen through Sequelize using created models (discussed later). Each model created in the project represents a table in the database. Currently the models created are:

- **PasswordReset:** Represents the password_reset table
- **Report:** Represents the report table
- **Role:** Represents the role table
- **Tree:** Represents the tree table
- **User:** Represents the user table
- **UserRole:** Represents an associative user_role table
- **UserTree:** Represents an associative user_tree table

Overall, each repository uses Sequilize based models to perform any manipulation or retrieval of database data. The actual location where the creation of a new database connection is made is in the file “Sequilize.js”. The code can be seen below:

```
1. const { Sequelize } = require('sequelize');
2. require('dotenv').config({                                    path:
   `./.env.${process.env.NODE_ENV.trim()}` })
3.
4. module.exports    =    new    Sequelize(`${process.env.DB_NAME}`,
   `${process.env.DB_USER}`, `${process.env.DB_PASSWORD}`, {
5.   host: `${process.env.DB_HOST}`,
6.   dialect: "mysql",
7.   pool: {
8.     max: 4,
9.     min: 0,
10.       acquire: 30000,
11.       idle: 10000
12.     }
13.  });
```

Repositories

When creating designs and implementations, abstractions can potentially improve the maintainability and future enhancement of code. Previously, Mongoose and its database access operations were coupled with controllers directly, leading to a complete refactoring of existing code in order to switch ORM systems. For this reason, the backend accesses database related features via classes known as repositories, each representing a resource of interest. This is done to prevent tight coupling of code, especially with Sequelize to external classes (such as controllers) which are only interested in pure data results. An example of a repository is a “User Repository” which can provide basic CRUD operations when manipulating user data in a database. For repositories to work, Sequelize is used as the underlying method of performing database transactions. As a result, in the event of major database changes or replacement of Sequelize, remaining code implementations can remain unaffected as repository interfaces remain consistent

Overall the repositories created are:

- **AdminRepository:** Contains all functions to satisfy admin related requirements. Examples are for blocking users, modifying admin privileges.
- **ReportsRepository:** Contains all functions to satisfy reports related requirements. Reports can be toggled as active, resolved, retrieved, created, and deleted.
- **TreeRepository:** Contains all functions to satisfy tree related requirements. Trees can be retrieved by ID, retrieved by user ID, get all tree data, watered, and adopted.
- **UserRepository:** Contains all functions to satisfy user related requirements. User password and emails can be changed, users can be retrieved by ID or searched, users can be deleted, created, and logged in to.

Unfortunately, as of this documentation, current implementations possess certain faults in design due to project deadlines. Ideally, repositories should return resources or Data-Access-Objects (DAO). As of now, repositories (with exceptions) are implemented to return complete response objects (containing the requested data) to be immediately returned by controllers. This implementation tightly couples the repositories to the responses which normally controllers or separate services classes (encompassing business logic) should generate.

Models

There are two types of models in the backend. The database and response models.
The response models are used to create response objects that populate themselves automatically on construction and decouple the Sequelize ORM responses from database communication. The automatic population helps because otherwise you would need key/value pairs in all endpoint methods in the controller when sending a response. There are response objects for Users, Trees and Reports and their collections. In the file, you will notice a set of attributes and a constructor which populates these attributes.

On the other hand, there are the database models. These are used by Sequelize, the Object Relational Mapper (ORM) solution used to simplify communication between the database and Node.js backend. For every table in the database, there is a database model. Sequelize requires the **definition of '.define**()` of tables. All tables have a modelName, attributes, database options and relations. The model name is the reference name of this table in model format; it should match your class name for this file.

Then, the attributes describe:

- The type (BIGINT, DATE etc.)
- If they can be null
- If these should be auto-incremented (in case of id's for example)
- If they are primary keys
- If they are foreign keys (references: takes another model by its modelName and a key which is its primary key)
- If they get a default value

The database options are simple. Provide the table name present in the database. Then mention if this table uses timestamps, createdAt and updatedAt.
Finally, the relations describe the relational aspect of the database (MySQL is an RDBMS). Relations are defined on the model that shares their primary key using a foreign key to another model/table (notice the relations between User.js and PasswordReset.js). To describe a one to one relation you need to write it like so:

```
1. User.hasOne(PasswordReset, { foreignKey: "user_id" })
2. PasswordReset.belongsTo(User, { foreignKey: "user_id" });
```

In the snippet above 'User' is the current model which shares their primary key and `PasswordReset` is the model receiving the foreign key. The `foreignKey` needs the exact name of the foreign key column in the database (in our case, the PasswordReset table in the database has a column user_id which will hold the foreign key of the user table's id column).

To describe a many to many relation you need to write it like so:

```
1. User.belongsToMany(Role, { through: "user_role", foreignKey:
   "user_id" })
2. Role.belongsToMany(User, { through: "user_role", foreignKey:
   "role_id" })
```

In the above snippet, a User has many roles and many roles may belong to a User. In other words, users and roles share a new table which acts as a many-to-many relationship containing however only a reference to each other's foreign key references. This table above is named 'user_role' and the two foreign key columns are 'role_id' and 'user_id'.

Another method of declaring relationships is the previously mentioned references method, which will be shown below in more detail:

```
creator_id: {
  type: DataTypes.BIGINT.UNSIGNED,
  allowNull: false,
  references: {
    model: User,
    key: 'id'
  }
},
```

Here, the creator_id is a column of the Report table and Report model, and it mentions that it possesses a reference to a User model with a primary key id. The above is an example of a one to many.

Authentication & Authorisation

The backend facilitates the process of authenticating users based on provided email addresses and password inputs. When the provided email address and password of a user matches with the values stored in the database, they are authenticated. Once authenticated (via the users controller), users receive a JWT based token, containing their email, user ID, and roles. The frontend then receives this token and includes it in requests when needed, especially for authorisation purposes in the future. Notably, when new user accounts are created, their password inputs are hashed for security purposes before being stored in the database. The implementation of authentication logic can be found in the user's controller and repository classes. An example of a JWT token being created is shown below:

```
token = jwt.sign(
  { userId: user.id, email: user.email, roles: user.roles },
  process.env.JWT_KEY,
  {expiresIn: "1h"}
);
```

Authorisation in the web application is accomplished via a roles-based solution. Every user in the web application is assigned one or more roles, where each role indicates their authorization to perform actions. Currently, only two roles exist (hardcoded in the database), “user” and “admin”. The actual logic which determines the capabilities of these roles is hardcoded in the implementation of the backends controllers. Any change of roles permissions requires the updating of affected backend controller classes. Importantly, every new user account created will always possess the basic role of “user”.

To perform authorisation of requests, the backend utilizes a custom middleware implementation which extracts the roles of a user when they send a request for a specific endpoint. The middleware which is responsible for decoding extracted tokens is “authJWT.js”. The roles are

extracted from a decoded JWT token provided in the header of responses under “Authorization”. The format of JWT tokens provided in requests is as “JWT <token value>”. The roles acquired are then appended to the original request made to the backend (as an array of strings) and the request is forwarded to the intended controller endpoint. The actual conditional checks which are made to determine access are done so in the controllers.

Authorisation middleware example:

```
1. //Function performs authorisation by retrieving the current user from a database
2. //using the a ID provided by a JWT token provided in the requests headers
3. const verifyToken = async (req, res, next) => {
4.   //Find and decode JWT token from request header
5.         if    (req.headers    &&    req.headers.authorization    &&    req.headers.authorization.split(' ')[0] === 'JWT') {
6.             jwt.verify(req.headers.authorization.split('   ')[1], process.env.JWT_KEY, function (err, decode) {
7.
8.       //Check if token could decoded
9.       if(!err){
10.           //Append user roles to request and resume execution
11.           req.roles = decode.roles;
12.           next();
13.         }
14.         else{
15.           //Token verification error
16.           next("/error");
17.         }
18.       });
19.     } else {
20.       //Token identification error
21.       next("/error");
22.     }
23.   };
```

Controller authorisation checking example:

```
1. //Authorisation Check
2. if (req.roles.includes("admin")) {
3.   ..code
4. }
5. else {
6.   //Unauthorised Error
7.   const error = new HttpError(
8.     "Unauthorised",
9.     403
10.     )
11.     return next(error)
12.   }
```

For the middleware to take effect, a new “route” must be created for a respective endpoint requiring authorisation. The middleware will not execute on every request but only when set in

advance in the configuration of routes. To add authorisation checking, a new route must be added before the intended endpoint is reached, an example can be seen below:

```
1. //Un-adopt trees
2. router.patch("/:tid/delete", verifyToken);
3. router.patch("/:tid/delete",
   treesControllers.removeTreeAdoption);
```

For the code to work, ensure that the following import exists:

```
1. verifyToken = require('../middlewares/authJWT');
```

User & Report Pagination & Querying

There are two methods dedicated to providing paginated results with querying options in the backend, “getUsers” and “getAllReports”. These methods can be found in the user and report repositories respectively. Both the searching, querying and paginating of results for either reports or users themselves are done in a similar manner.

In regards to pagination, each time a request is received in either the users or reports methods, both methods will count the total number of matched records. At the same time, the methods will retrieve a limited number of matching rows (data) to be returned as a result. The limit is based on an input parameter “limit”. Both the total number of matching records and the provided limit in the original request determine the total number of paginated “pages” the requestor can go through. To be able to go to the next “page” of results, a “skip” value can be provided, which indicates the number of matching records which should be skipped when returning results. Overall, on successful completion, both methods will return their collected data, the total count of matched records, the current page the requestor is on and the total number of pages. An example is visible below:

```
1. //Send result
2. return {
3.   data: users,
4.   paging: {
5.     total: usersCollectionCount,
6.     page: currentPage,
7.     pages: totalPages,
8.   },
9. };
```

To perform querying of data, both methods differ in options and implementations. For users, querying is performed via a provided email input. However, depending on the format of the email address, the method adapts accordingly. If a full email address is provided (based on if a “@” is visible), an exact email search is performed. If a partial email address is provided (i.e. the first half of an email address without a “@”), similar named emails are matched with when searching. When querying reports, multiple inputs can be provided. Reports can be searched by “resolved status”, “is active”, the ID of the original report creator, and the report title set. In the

current implementation, all inputs are inclusive (i.e. all values provided must match for a row to be returned). Titles provided to not have to be an exact match. Notably, any combination of inputs can be provided as the method adapts its queries accordingly.

Email Notifications Helper

In order to implement certain functionalities in the project, there was the need to be able to send emails. One of the most important functionalities was for users to have the ability to reset their account password in case they forgot it. The forgot-password functionality required the use of an NPM library "nodemailer". With this library, it is possible (using a provided email address) to send emails to specific end-receivers. A normal, success scenario of the *forgot-password* is briefly outlined below:

- User clicks forgot-password and is redirected to a new page
- User inputs email
- The system sends email to the user containing a link to reset the password
- User clicks the link and is redirected to a new page where they can type their new password

Another scenario where the need of sending emails came up, was to allow the users to report a problem with their tree or any other tree that they believe has an issue. Once the reported issue has been resolved, an email is sent to the user who reported the issue to inform them that it has been resolved. Nodemailer was also used in this case since it was the best free option. A typical success scenario of reporting an issue would be:

- User clicks report an issue for a specific tree
- User selects one of the options for the issue or adds their own description
- User submits the issue
- User receives an email once the issue has been resolved

The code which is responsible in the backend for sending emails is found inside the helper class "EmailHelper.js". This class contains a single method implementation which is responsible for sending an email to a single email address. Both the subject and HTML contents of emails must be provided to form a valid email. The function code can be seen below:

```
1. //The from email parameter needs to be a valid email.
2. function sendEmail(from, to, subject, htmlContent) {
3.
4.   //Set mail options
5.   let mailOptions = {
6.     from: from,
7.     to: to,
8.     subject: subject,
9.     html: htmlContent
10.     }
11.
```

```
12.     //Send mail based on mail options
13.     transporter.sendMail(mailOptions, function (err, info) {
14.       if (err) {
15.         console.log(err);
16.       } else {
17.         console.log(info);
18.       }
19.     })
20.   }
```

Environment Variables

The backend has three separate environment files which are used to inject values during backend execution. The files are:

- ".env.development": Contains environment values for a development environment.
- ".env.production": Contains environment values for a production environment.
- "nodemon.json": Used by the nodemon library to also inject development environment values.

Both the ".env.development" and "nodemon.json" files are expected to contain values for use during development. Depending on how the backend is executed, only one of the two files is ever used to inject environment values. If nodemon is used, the "nodemon.json" file is used, otherwise "env.development" is used. For more information regarding executing the project, please refer to Project Execution.

The backend has the following environment variables:

- DB_USER: The user of the database to authenticate as when connecting.
- DB_NAME: The name of the database to connect to.
- DB_PASSWORD: The password of the database user being authenticated with.
- DB_HOST: The URL of the active database.
- CORS_ORIGIN: This is used to prevent CORS errors when developing locally. This should be left untouched.
- DELETED_USER_ID: This environment value contains the database ID of a user tuple, which is used as a placeholder when a user's account is deleted. The placeholder user is then displayed for each report created by a deleted user.
- JWT_KEY: This variable is used when creating new JWT tokens.
- NOTIFICATIONS_EMAIL: The email address which will be used to send notifications.
- NOTIFICATIONS_EMAIL_PASSWORD: The password of the email address which will be used to send notifications.

- NOTIFICATIONS_EMAIL_SERVICE: This is used to specify to nodemailer the email service being used.
- BASE_DOMAIN: This variable is used to set the base domain of all requests or links used.

Frontend

This subsection discusses the implementations and structure of the frontend with its uses of components, authorisation and environment variables.

Structure

The structure of the frontend mainly exhibits a collection of folders. The most important of these folders is the ‘components’ folder. Here, the way components are arranged is the following:

1. For every style file (.css) you must use a separate folder
2. Every folder must contain its helpers, functions or any other components related to its main component. The main component is identified by using the same name as the folder. All of the previously mentioned are JavaScript (.js) files. Note: All components are functions too.
3. There may be some folders which group other folders/components with each other.

Another folder is the ‘assets’ one, which is meant to contain all images, videos, icons, or other raw media types. Then the context folder contains all globally accessible context hooks. The ‘hooks’ are responsible for holding globally available hooks (methods) that can be used in various components. The ‘models’ folder will contain objects that will encapsulate the response data retrieved from the backend APIs. Finally, ‘util’ is meant to contain utility functions (equivalent to helpers in the backend) available globally across the frontend.

Page Components

The frontend has the following components:

- Account Manager: an admin only page which provides access control functionality over users. Admins can block/unblock other accounts here. Alternatively, they can also upgrade the rights of users to admin and reverse administrators to users.
- Auth: This is the login/sign-up page. The user also has the option to reset their password if they try to login using their email. When signing up some necessary information about user data is displayed to conform to Google Policies.

- Contact Us: A simple page with contact information provided by the municipality of Thessaloniki.

- Landing Page: The index page of the application, aims to provide a humoristic entry point to the web app. It also contains a call to action to login if users are not logged in already.

- Legals: This page provides access to the terms of service and privacy policy that conform to EU and US policies (US necessary for Google Platform). These are the only pages in the application written in English due to Google Policies.

- Map: It contains two types of maps. The main map through which the user can adopt trees and see them across Thessaloniki. On this map, they can also identify trees already adopted by other people. The second map type is a smaller one used to show the position of a single tree in my trees page.

- Navigation: This component contains the mobile menu, main navigation (left side with logo and username and nav links (right side, general navigation).

- Reports: This folder contains all report pages. There is the main report page which contains a report list with report items. These are tightly coupled together, hence why they are in the same folder. Here, administrators can view all user reports on trees and they can investigate/abandon a report and resolve them.

- Reset Password: This page is used when the reset password process is triggered and the email has been verified. Here the users input their new password and are then redirected to the main application.

- Settings: This page is used to change email and password or alternatively delete the user's account altogether. The email changing process logs the user out. The password change process is the same as forgot password.

- Trees: This page provides access to my trees which show all the user's adopted trees along with several actions they can perform on those trees. Such actions are to report a problem, water, view on the map or remove from adopted trees. When a tree report is created a description and a title is sent to the reports page which can be viewed by administrators. Viewing the tree on the map shows a small local map around the tree.

Custom UI Components

In this project, there are custom UI elements that can be reusable and accessible every time you want to create something UI related.

- Backdrop: The backdrop component will add a dimmed layer over your application's current screen. Can be used to emphasize a new application state according to a user's action (event).

- Card: Cards are self-contained pieces of information that may contain actions. Elements, like text and images, can be hierarchically placed in them. They have 3 principles:
    - Contained: A card is identifiable as a single, contained unit.
    - Independent: A card that can stand alone, without relying on surrounding elements
    - Individual: A card cannot merge with another card or divide into multiple cards.

- ConfirmationDialogBox: It is a custom dialog box that contains "agree/decline" or "agree/disagree" buttons and based on the context it can be useful for specific scenarios.

- FormElements: Contains the elements that are necessary for a form. The most usual ones are buttons and inputs
    - Button: buttons are elements that execute specific actions, especially when they are clicked or hovered. You can set default colors and interactions when users are hovering them. You can create classes for each button, so they can be reusable based on the situation(red buttons for error, orange buttons for warnings etc)
    - Input: fields where users are typing whatever they need to type. You can set rules based on the input(valid email format, password with minimum 8 characters etc)

- LoadingSpinner: An animated element that indicates loading is in process. It will appear when data is loading slowly or when the connection is interrupted while loading. Very useful in order for users to not stare down a static white screen while waiting for data.

- Modals: Modals are very similar with the dialog boxes, but the difference is that you can insert whatever content you want based on the context. It is not limited only to the "agree/disagree" concept. Also, there are the error modals which are related with the modals. They are automated modals that are shown whenever they is a specific error in the content of the modal. Modals consist of a lot of properties, but the most essential ones are:
    - Show: the name of the modal that you want to show
    - onCancel: when the modal should be removed
    - Header: the header of the modal
    - Footer: the content that the modal should have

- Toast: A Toast is a non-modal, unobtrusive window element used to display brief, auto-expiring windows of information to a user.

Authentication & Authorization

Authentication is relatively straightforward. In the signup screen, a valid email address is checked, and the password must be at least 8 characters long. Also, the user must agree with the terms of service to proceed with a new account. When logging in, the user must provide their existing account information and to aid them in avoiding mistakes, the same validation methods on email and password are performed.

Authorization in the frontend relies on the use of the globally available context. The context is set during the login process and kept in persistent storage on client side (local storage). The full process becomes apparent if the following files are traced in this order:

1. Auth.js: The login/sign-up page. This is where the login/sign-up hook is triggered.
2. Auth-hook.js: This is a globally accessible hook which allows the frontend to perform the login, sign-up or logout processes. It saves and clears the data in local storage. Note: The order of the parameters and props is significant!
3. App.js: The main file will load the data from the context during runtime and make it available to all components. You can call this data and use it using the context hook like so: ` const auth = useContext(AuthContext);`

Environment Variables

Environment variables are variables that are available through a global hook using `process.env ObjectYouWant`. The environment variables in the frontend are:

- REACT_APP_GOOGLE_API_KEY: An Google API key is a simple, encrypted string that identifies an application without any principal. They are useful for accessing public data anonymously, and are used to associate API requests with your project for quota and billing.
- REACT_APP_BACKEND_URL: It represents the application's url and can be used most of the time whenever you are trying to make a request. Instead of writing the whole url all the time, with this variable you can just type the name of the endpoint
- PHONE: It represents the phone number for communicating with the project's team.
- EMAIL: It represents the email address of the project

Furthermore, there is also the .env.production which contains the environment variables but for the production. This will be used after the project's implementation, and the only essential variables are the Google API key and the application's backend url.

Enabling Email Notifications

By default, the web application (and more specifically backend) does not have a functioning email notification system for when users request to change their passwords or when a submitted report is resolved. This is mainly due to a missing valid email address and email password input in the final backend source code. To enable email notifications to work as expected, two steps need to be performed:

1. Enable “Less secure app access” in the Gmail account, which you will use to send notifications. This can be done by going to “Manage your Google Account” and searching for “Less secure app access”.

2. Modify all “.env” (environment) files in the backend, specifically the values:

    NOTIFICATIONS_EMAIL: <New Email Address>
    NOTIFICATIONS_EMAIL_PASSWORD: <New Email Password>

With the new values in place, restart the backend and perform testing to ensure that the new email is being used without any issues. If the email's credentials are incorrect, an error message will appear on the terminal executing the backend.

Continuous Integration/Deployment

The application has one continuous integration script which builds the application, tests the validity of all dependencies and tests the application for compatibility with various node versions on both backend and frontend. Also, the frontend is build so that any compile time errors can be caught. When creating pull requests to merge with the main branch, the below CI will block the operation if any errors are thrown during the process. This CI triggers on very push to the remove repository from any branch, in the case that costs start to accumulate use it only on feature branches and minimizes commits to the remote. This can be done by using git squash before committing to the remote. The file performing this operation is named `build CI` and it is compatible only with GitHub Actions servers. Here is an explanation of some of the code:

```
1. strategy:
2.   matrix:
3.     node-version: [14.x, 16.x, 17.x, 18.x]
```

The above determines which versions are to be tested for compatibility.

```
1. # Get Node
2. - name: Use Node.js ${{ matrix.node-version }}
3.   uses: actions/setup-node@v3
```

The above sets the node version so that it runs in all versions.

```
# Caching
with:
  node-version: ${{ matrix.node-version }}
  cache: "npm"
    # Required for 'npm ci' command to use the existing package-lock.json
  cache-dependency-path: "./backend/package-lock.json"
```

The above performs a caching of the dependencies on the server so not all of the dependencies have to load again and only the ones that changed.

```
- name: Upgrade Container NPM
  run: npm install -g npm
```

The above upgrades the node package manager version for this run because the GitHub server version is not compatible with the higher versions of node tested prior.

```
- name: CI
  run: CI=false npm ci
```

The above runs the CI version of npm install which uses the package-lock.json file to check dependencies.

In the frontend side, the difference is that in addition to the above, a build function is available to catch compilation errors related to React or JSX. This is done through the code below:

```
- name: Build
  run: CI=false npm run build --if-present
```

Although the above may be perceived as very simple CI methodology, there were more than 5 different bugs and more than 10 instances that it showed that a commit had issues and should not be merged with the main branch.

Testing

When the product was received by the previous developers, no testing framework was included and no testing was performed. Once the requirements were prioritized, automated testing received low priority, with a higher focus on adding features and swapping deployment methods to more realistic options. This led to testing receiving no further attention compared to the previous team, either in terms of performing tests or integrating a framework for the backend or the frontend.

Running the Project

This section aims to provide basic instructions for running the project on a developers local machine, the assumption is made that project files used are readily available. Inside the project folder there are two notable folders named “frontend” and “backend”. These folders respectively

encompass the entirety of the frontend or backend source code separately and thus, must be executed separately. Before executing any commands, ensure that Node JS is installed and NPM (Node Package Manager). The node version should ideally be “v16.15.0”.

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:developer's
        i. “npm run dev_win”
        ii. “npm run start_win”
    b. If you are using a UNIX based operating system
        i. “npm run dev_unix”
        ii. “npm run start_unix”

The backend has installed an NPM dependency called “nodemon”. This library allows for the execution of the backend with automatic restarts on code changes, this is valuable for developers convenience. Thus to execute with nodemon, use the commands that include the keyword “dev”. It should be noted that both the standard and “dev” commands inject separate environment values on use, please refer to the backend's Environment Variables for more details. 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.

Local Database Installation & Execution

To execute the MySQL database the project will use, a variety of tools can be used as long as the underlying database contains the correct schema and connects with the backend. For simplicity, the following section will provide instructions for the local installation of MySQL database and its connection to the backend of the project using AMPPS. This section makes the assumption that the SQL file used to import the latest developed database is readily available.

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. If prompted to login, the default username is: “root” and the 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. A single event should exist which is responsible for automatically deleting password reset attempts 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.

Remote Web App Installation & Deployment

This section will outline the steps necessary to create, install or otherwise deploy the web application of the project or its Android application derivative. Sections will be dedicated for both the web application and Android application separately.

For future reference and to prevent confusion, the following terms are used:

- **Backend:** This statement refers to the “backend” folder found inside the web application project. This folder contains the entirety of the backend source code.
- **Frontend:** This statement refers to the “frontend” folder found inside the web application project. This folder contains the entirety of the frontend source code.
- **Project Folder:** This refers to the folder which contains both the frontend and backend folders, together with any additional project files.

Web Application

This subsection will first describe the process for building the web application into a production-ready state, followed by another subsection dedicated to cPanel deployment.

Project Building

Before the web application can be built, as a preliminary step, the command “npm install” must be executed for both the frontend and backend subprojects. Once all of the dependencies have been downloaded and installed, the project can be properly built.

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 previous steps, the backend can be executed with “npm run start” to ensure that the built web application works as expected, encompassing the frontend. Overall, the backend project folder has in effect become the final built web application.

cPanel Installation & Deployment

As of this documentation, cPanel is the service provider used to deploy the final web application and serve it to end-users. For this reason, instructions will be focused on the use of cPanel as well as production deployment of the previously built web application. If the web application has not been built, please refer to: Project Building.

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”.

Database

This subsection provides instructions on how to deploy or update the remote MySQL database solution used on cPanel. The assumption is made that a newer version of a database has been exported and is ready for use to be imported 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 “With Selected” select field. When dropping all tables, disable “Enable Foreing 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.

Android

Before an Android application can be generated (as a PWA), a successful installation and deployment must first be made for the latest web application. If this has not been accomplished, please refer to: Web Application section.

Conformance to Google Policies

Every application should provide users with a reasonable degree of functionality and a respectful user experience, but without sacrificing privacy. For that reason, it is very important to follow user protection policies which in this case are that Google Play Store’s publishing policies, Google’s general policies and the EU Privacy Shield Framework. These are not only useful for protecting our user base, but also enforces good practice on developers. In the case of this project it was a necessity since a deployment to the Google Play Store was part of the requirements.

The current version of the app conforms to the following policies:

- Collecting and using user’s personal data. We store some data in the database (email) and in local storage. We state these things in the legal pages and on sign-up as requested by Google’s guidelines. Functionality such as change email, password and deleted account (along with all user data) are a requirement in the policies, so they had to be implemented.
- Links to other websites (occurs in the contact us page, where a link is directed to the municipality website).
- Contact us (a page that includes the phone number, the website, the email and the location of the municipality of Thessaloniki in case the user wants to communicate with it.)
- Changes to this privacy policy (From time to time, there is a possibility for some updates/changes in the application’s privacy policy. Of course,users will be notified via email and/or a prominent notice on the app’s service for any kind of change).

PWA Creation

The creation of an Android application can be achieved by using PWABuilder, for more information please refer to: PWABuilder. During the creation of a new PWA it is possible through configuration to specify changes that affect the visual presentation and the “look and feel” of the application. These changes can be made if desired through PWABuilder. To create a new PWA/Android Application:

1. Deploy the complete web application to a live production server.
2. Follow the instructions in “https://docs.pwabuilder.com/#/builder/android?id=packaging”

Important: When generating a new application, place in a secure location the “signing.keystore” and “signing-key-info.txt” files which were generated. These files are vital for future application updates and should not be lost.

Uploading a new Android Application

Before uploading a new Android application, ensure that you have completed PWA Creation. To upload a new Android application to the Google Console Android, simply follow the instructions in “https://docs.pwabuilder.com/#/builder/android?id=publish”.

Notes: When uploading the **“**assetlinks.json**”** file to the live production server, place the file in the “public/.well-known”directory of the backend project. This can be done using cPanels file manager user interface.

Updating an existing Android Application

To update an existing Android application, you must be in possession of the original “signing.keystore” and “signing-key-info.txt” files which were generated when using PWABuilder to create the application the first time. Without the original files, it is impossible to perform a successful update without creating a new application project on Google Console. To update an existing Android application on Google Console Android, follow the instructions written in “https://docs.pwabuilder.com/#/builder/android?id=update-existing-pwa”.

Important: If you are using existing files used by the current project team, as of this documentation specially attention is required when using PWABuilder. When following the instructions in the link above, you can find the “key alias”, “key password”, “store password” values from the original “signing-key-info.txt” file. However, it is vital that the package name has been renamed to “com.herokuapp.mottrees.twa” during the building process if the files of this project are used.

Notes: When a new .”abb” bundle file has been created, navigate to the “App Bundle Explorer” on Google Console Android and upload the file as a new version of the latest app bundle file used. If an error occurs that the bundle file uploaded is not the latest version, repeat the

instructions in "https://docs.pwabuilder.com/#/builder/android?id=update-existing-pwa" and ensure that the version number has been incremented.

Google Play Store Deployment

Before following this guide to formally deploy to the Google Play Store, please complete the "Uploading a new Android Application" or "Updating an existing Android Application" sections. By completing these sections, the assumption is made that the latest .aab app bundle is already uploaded to the Google Console Android. Please note that in order to deploy a new Android application to the public store, a formal review process is performed by Google lasting up to at most seven days. If the Android application submitted does not meet Google standards (according to their policies), any formal releases will be refused. For the purpose of simplicity, the instructions provided will not include a testing period phase which can normally be performed.

To perform a new release of an Android application:

1. Log into the Google Console for Android developers using a developers account. This is assumed to be readily available.
2. Select the Android application project being used.
3. In the navigation menu on the left, press the "Production" button.
4. Press "Create new release".
5. Select the latest app bundle which should be released. Fill in any other information if desired.
6. When ready, press "save" then "Review release".
7. From the next page, you can select the countries which should be targeted for release. After filling in any desired information, press "Start rollout to Production".
8. In the navigation bar, press "Publishing overview". From here you can see and manipulate all of the final details before final deployment. Special attention should be placed at the "Main store listing" section and "App Content" within "Data Safety". These sections should be examined closely by pressing their links and setting descriptions and completing a "data safety" questionnaire which is used to state the data requirements of the application.
9. The latest release created should be visible and ready for review. When ready, press "Send for review".

Evaluation

This section briefly mentions all notable issues or unfinished work which were not completed in time. The information here should aid in indicating which features are partially complete or issues that should be addressed in future works. In some cases, suggestions from the team will be provided here on how to proceed.

Known issues:

- There is currently nothing implemented to prevent a single user from resetting their password too many times or reporting too many issues.
- The report sorting, searching and pagination are not done in the frontend. The backend has already been performed, however, look here.
- The multi-language functionality had been started at the research stage, but no implementation took place in the frontend.
- There is no way for a report to be put back as unresolved or to be outright deleted, although technically possible in the database.
- 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.
- 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.

Unfinished requirements:

- Distinguish age visually between trees on the map. This was previously attempted using different colours to show the age of the tree. Potentially different shades of green.
- 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.
- 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. The only way communication between administrators and users could be performed would be through a user contacting the municipality from the contact us page.
- 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.

- Add a custom name to trees. This is supported by the database, but needs an endpoint in the backend and the appropriate frontend to be performed.

- Track the weather and implement it into a watering notifications algorithm. This requires an external API to get the data. The algorithm would have to run on the server (backend) and manage issuing these notifications.

- Add watering requirements per tree type. This requires a knowledge base for each tree. This means that either a human feeds data to the database adding the watering requirements per tree type or an automated solution would use an external API.

- 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.

- Competition/Leaderboards. These are meant to show in the whole country the users with the most points down to the users with the least amount of points, just like a game leaderboard.

- Regional Leaderboards. This is an enhancement of leaderboards where in addition to a country wide leaderboard there is an option for a leaderboard only for the local population per municipality.

- View other people's adopted trees. It is expected here 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.

- Comment on trees. Trees would take the form of a post which has comments. Might also have to consider a reply system.

- 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.

- Inform admins that rubbish is removed from a tree. Right now, the current options for the trees are "view it from the map", "water it", "delete it from your trees" and report a problem to the municipality. Another option can be added where the user can inform them that any rubbish has been removed from their adopted tree. Alternatively, the existing report option can include an optional photo section.

- 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.

- 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.

- Limit the maximum number of tree adoptions. The actual limit should be communicated

with the clients. The limit must take place in both the backend and the frontend to ensure it cannot be bypassed.

Sprint Reviews

SPRINT NUMBER: 1

From date: 17/4/2022 To date: 6/5/2022

Planned stories

Table of stories initially planned to be implemented in the past SPRINT and their completion status

Status types:

- **On hold**: we are waiting for information from the client, placeholders are used (lorem ipsum).
- **Done**: front-end, back-end and content are done.
- **Partially**: either back-end or front-end tasks are finished and waiting for the other to be ready.

Story	**Status**
As a user, I want to view a page where I can see what the project is about, because I am new to the system.	On hold
As an admin, I want to modify user account privileges, because I want to manage users.	Partially
As a user, I need to report issues (includes tree being unhealthy) with the application, because I need to inform the municipality	On hold
As an admin, I want to be able to stop users from having adopted a tree if they neglect it, because it is abusing the system.	Partially
As an administrator, I should be able to view user reported problems, because I want to resolve problems with the system.	On hold
As a user, I want to be able to contact the municipality from the system, because I need to inform them about the system.	On hold
As a user, I would like to download the app from the Google Play Store, because it is convenient.	On hold

Major obstacles during the SPRINT

Resolved

Report any major obstacles encountered and how they were resolved.

- Builds with errors were being pushed to the remote, as a result, initial continuous integration has been configured:
    - Added Node Build continuous integration.
- GitHub repository was not configured:
    - Added issue and pr templates.
    - Configured automated kanban board with reviews.
    - Add branch protection rules.
    - Add Dependabot dependency version manager.
- When people started working, everyone was working on the project in their own way, but it was resolved by creating guidelines:
    - Project Work Guidelines document.
- Database backups prevented case of failures or accidents:
    - A locally installed MongoDB tool was installed, which allows for local backups to be created from an already deployed database instance. Documentation was created as to how to achieve this.
- Authorisation was added to the backend that prevented the proper addition of administrators:
    - New middleware was added which automatically decodes provided JWT tokens and retrieves user data from a database. User data retrieved is used for authorization purposes in requests.
- Major dependency support added:
    - React 18.
    - React Router 6.
- Landing page was added which shows necessary content for the project's purpose
- MongoDB as a database solution was kept:
    - Research was made into the factors of NoSQL against SQL. Considerations were made as to the risks to refactor.

Unresolved

Report any major obstacles encountered and how they are going to be resolved in the next SPRINT.

- Major dependency support issue identified:
    - Mongoose ORM v6.
- Inability to add Bootstrap (CSS library) due to breaking changes to existing code:
    - Time will have to be dedicated to converting existing CSS into Bootstrap equivalent. All affected pages will have to be refactored.
- Refactoring of existing code needs to be performed:

    - To improve the quality and consistency of code (such as with comments), all existing code needs to be refactored. This process will be lengthy in time.
- Several instabilities with CI (Continuous Integration) are discovered with continued progression, it still needs further configuration.
- A Google developer account is necessary for the deployment of the application to the Google Play Store. As no other ways exist to test if deployment will work, the process is on hold until the lecturer resolves the issue.
- Interviews need to be made to the client:
    - How the municipality wants to be informed from the users' reports.
    - The content of the landing page.
    - Details about the "Contact us" page.
- Front-end oriented task have not been completed due to needed client answers and content:
    - Interviews need to be performed with the client.
    - Pages need to be created that are not directly related to tasks to be completed.

Team reflection on the SPRINT

What was successful in the SPRINT? Why was it successful?

The team was able to start working on a considerable number of user stories, which provided valuable progress. Many bugs and dependency issues were also resolved during the first sprint which allowed for the development process to run smoothly. Proper documentation of guidelines (development standards) helped ensure the team developed at consistent quality and performed predictably. This especially helped with the use of GitHub.

How can this success be repeated in the next SPRINT?

In terms of tackling a lot of user stories, this can be achieved by:

- Minimizing documentation.
- Communicating between the back-end and front-end developers with cheap mock-ups the expected behaviors of various system components.
- Working in pairs depending on the technologies we are tasked with.

What went wrong in the SPRINT? Why did it go wrong?

Valuable time was spent fixing dependencies and understanding NPM's (Node Package Manager) integration with Node. One of the developers has spent most of their time working on that and the associated issues that occur with the CI pipeline and errors which break the development branch. Several user stories were placed on hold due to factors beyond the team's control.

How was the problem corrected or recovered?

The issue with NPM and dependencies has been resolved, but not fully as it becomes clear that issues still reveal themselves given enough time.

How could the problems be avoided in the next SPRINT?

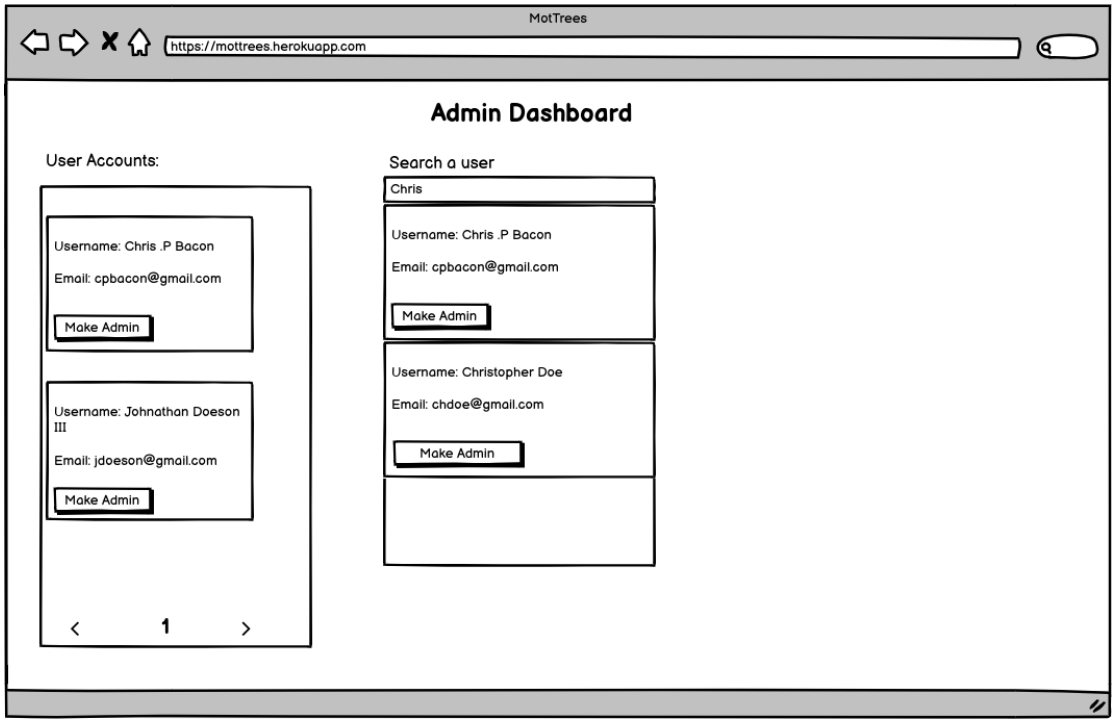
The NPM and dependencies issue is unavoidable; only with experience and further research effort can a resolution be reached. About user stories, we will contact the client for more information about the content and design decisions of some webpages.

First SPRINT documentation

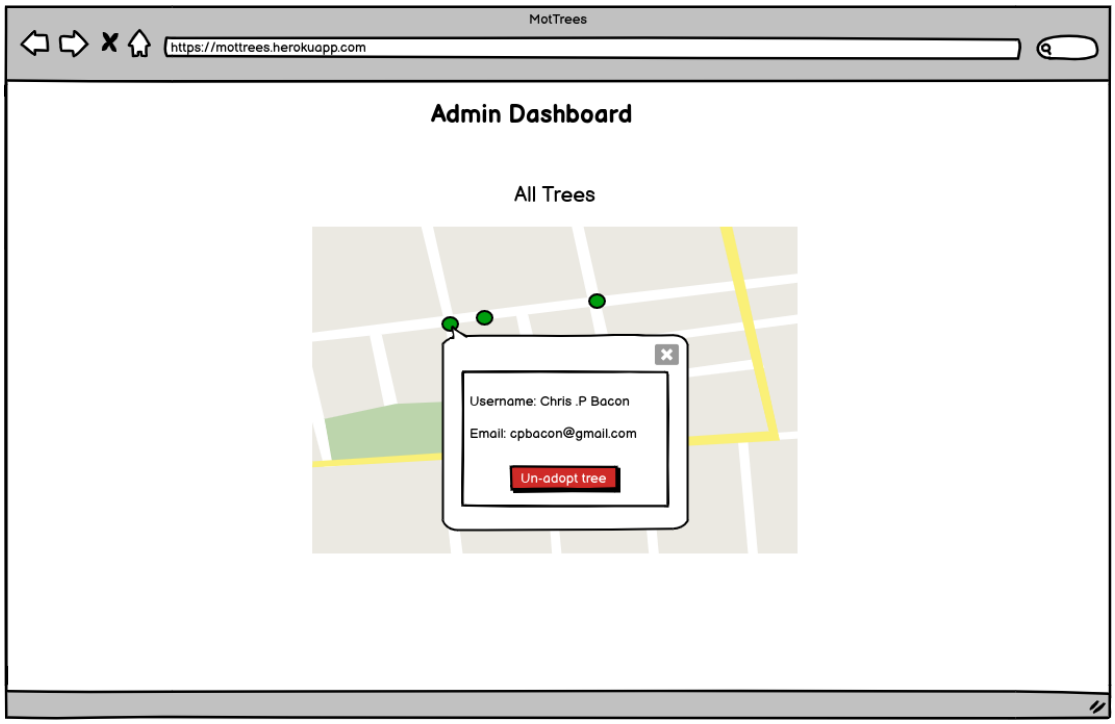
Here you can put any complete or work-in-progress products that have been produced during the SPRINT.

These might include:

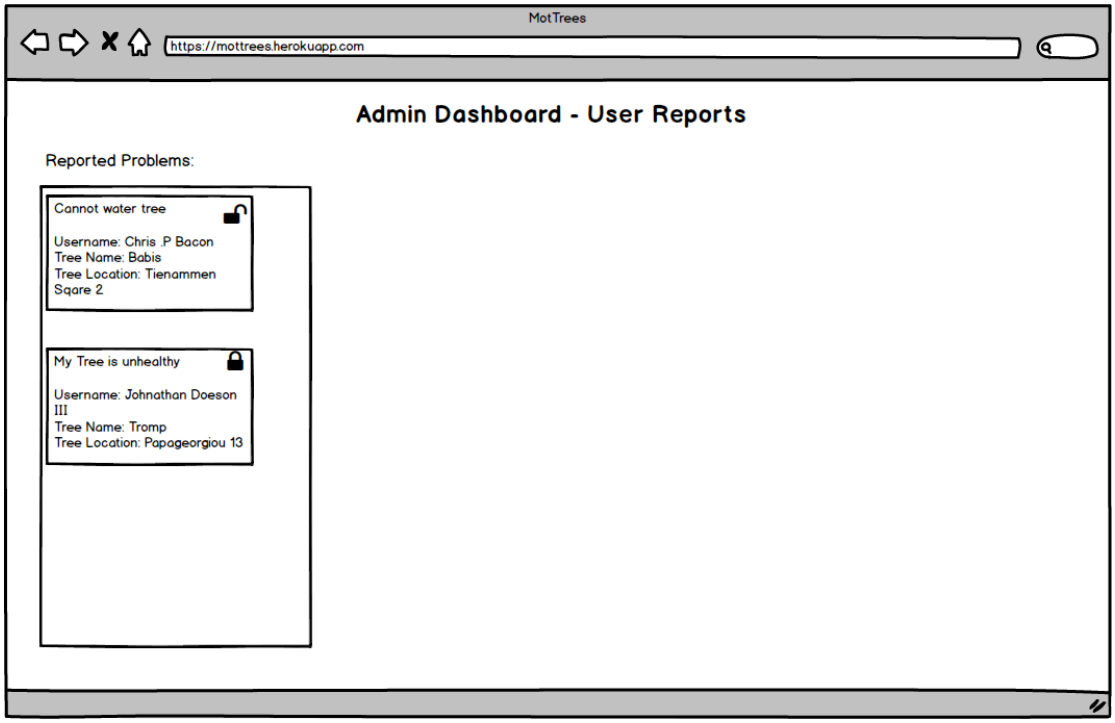
- Project Documents:
  - Project Work Guidelines
- PWA Documentation:
  - PWA Conversion
- MongoDB Documentation
  - MongoDB Backups Handling
- UI prototypes:
  - Upgrade user to administrator account



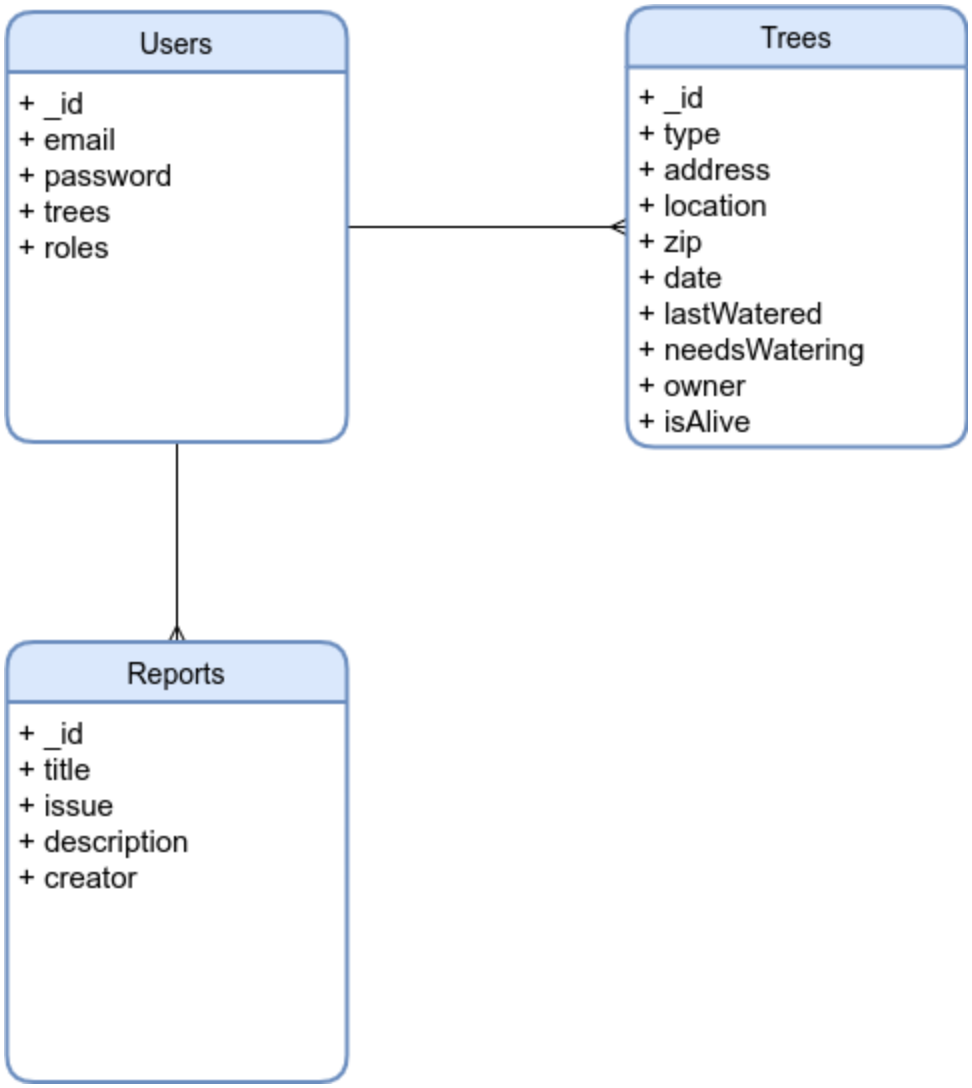
  - Admin removes tree from users



- Admin view user reported problems



- Database ERD



SPRINT NUMBER: 2

From date: 5/7/2022 To date: 5/20/2022

Planned stories

Table of stories initially planned to be implemented in the past SPRINT and their completion status is visible in the table below.

ID	Story	Completed (check)

S7	As a user, I want to view a page where I can see what the project is about, because I am new to the system.	✓
S8	As an admin, I want to modify user account privileges, because I want to manage users.	✓
S9	As a user, I need to report issues (includes tree being unhealthy) with the application, because I need to inform the municipality	✓
S10	As an admin, I want to be able to stop users from having adopted a tree if they neglect it, because it is abusing the system.	✓
S11	As an administrator, I should be able to view user reported problems, because I want to resolve problems with the system.	✓
S12	As an administrator, I should be able to mark reported issues as resolved, because I want to inform the system that the issue is done.	✓
S13	As a user, I need to be informed with a notification when an administrator has inspected my reported issue, because I want to know the progress of my reports	✓
S14	As a user, I want to be able to contact the municipality from the system, because I need to inform them about the system.	✓
S15	As a user, I would like to download the app from the Google Play Store, because it is convenient.	×

S17	As a user, I can personalize my tree (like give them a name), because it makes users feel closer to their trees.	×
S18	As a user, I should be reminded to water my tree when the time to water is nearing, because I want to make sure I don't forget (notification).	×
S20	As a user, I would like to see who has adopted each tree, because I would like to know.	×
S22	As a user, I would like to upload a photo to prove that I performed an action (reporting unhealthy tree and when tree has garbage)	×

Major obstacles during the SPRINT

Resolved

- Creating a developer account was a major obstacle from the beginning because of the cost that was needed to create one. This issue has now been resolved by the college as we now have access to a Google developer account, which will allow us to proceed with the Google Play Store deployment process.

Unresolved

There were no major (unresolved) obstacles encountered during the sprint which hindered or prevented the completion of stories (and tasks).

Team reflection on the SPRINT

What was successful in the SPRINT? Why was it successful?

Overall, the sprint is considered mostly successful as a majority of the user stories of this and the leftover stories from the previous sprint were completed. Success can be attributed to the fact that very few obstacles were encountered and (as suggested), concentration of teamwork was emphasized as to properly complete more stories. An important achievement was being able to send transactional emails (as notifications) using a service provider. A reasonable service provider was found and implemented.

How can this success be repeated in the next SPRINT?

As demonstrated in the previous sprint, by concentrating developers wherever possible, more progress can be made. Otherwise, further development will proceed as usual.

What went wrong in the SPRINT? Why did it go wrong?

Despite the lack of completion for specific planned stories, further attention needs to be placed on the improvement of UX (User Experience) and refactoring of existing code. Apart from that, as mentioned, no major obstacles were encountered and caused significant issues in the sprint.

How was the problem corrected or recovered?

The problems are to be recovered in Sprint 3 by applying the necessary suggestions from the supervisor.

How could the problems be avoided in the next SPRINT?

When developing web pages, increased effort will be placed on providing a sensible UX.

Second SPRINT documentation

No documents were created which are of value to future developers.

SPRINT NUMBER: 3

From date: 23/5/2022 To date: 3/6/2022

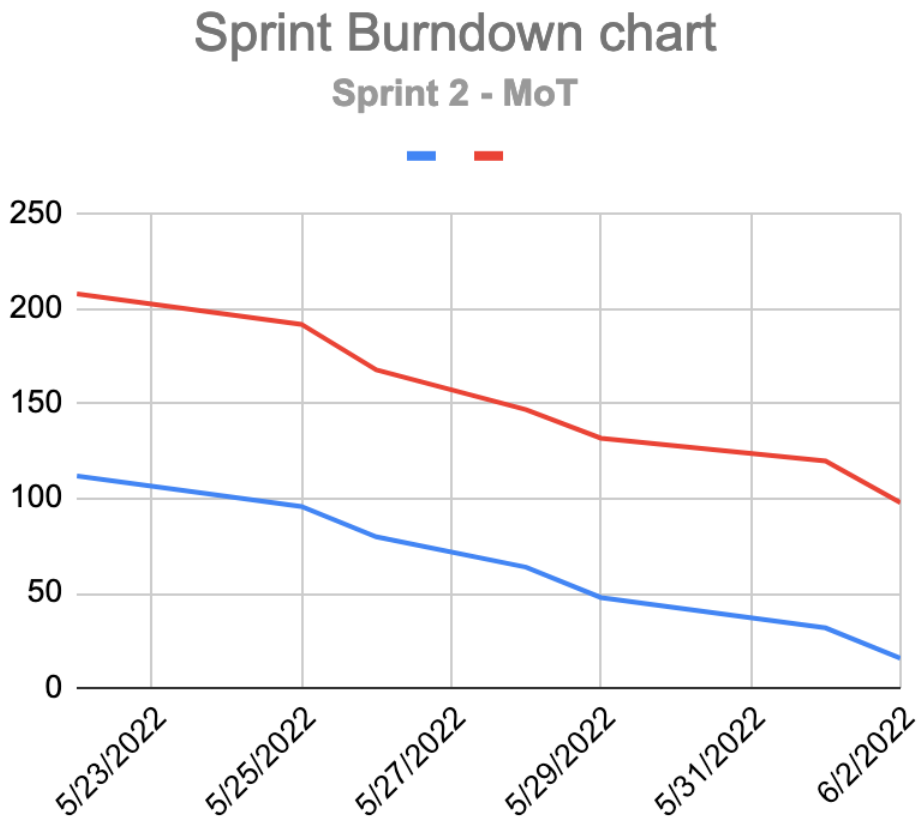
Planned stories

Table of stories initially planned to be implemented in the past SPRINT and their completion status

ID	Story	Completed (check)
S22	As a user, I would like to upload a photo to prove that I performed an action (reporting unhealthy tree and when tree has garbage)	
S20	As a user, I would like to see who has adopted each tree, because I would like to know.	
S17	As a user, I can personalize my tree (like give them a name), because it makes users feel closer to their trees.	
S18	As a user, I should be reminded to water my tree when the time to water is nearing, because I want to make sure I don't forget (notification).	
S15	As a user, I would like to download the app from the Google Play Store, because it is convenient.	
S24	As a user I need to be able to reset my password, because I need access to my account if I forget.	
S25	As an admin I need to be able filter and search the user reports, because it helps me do my work easier.	
S26	As an admin I need to be able to view paginated results for the user reports, because there are too many reports on the screen at once and I am getting confused.	
S27	As a user I need to add a description to my reports. because I need to tell the municipality the specific issue.	
S28	As a user I need access to multiple languages across the website, because I don't understand Greek	

S29	As an admin I would like to see my username in every page so I know which account I am using	
S30	As an admin I need to be able to view which admin account a user report is currently associated with and awaiting resolution.	
-	Migrate Mongoose to Sequelize	X
-	Add abstraction between Sequelize and Controllers	X
-	Migrate Database from MongoDB Atlas to local MySQL	X
-	Migrate frontend from Mongoose to abstracted responses from backend	

Screenshot of BURNDOWN CHART



Major obstacles during the SPRINT

Resolved

Report any major obstacles encountered and how they were resolved.

- Frontend developers had to familiarize themselves with the backend code that handled routing, controllers and ORM. At the same time, they had to follow the migration deadlines.

Unresolved

Report any major obstacles encountered and how they are going to be resolved in the next SPRINT.

- Nothing to report.

Team reflection on the SPRINT

What was successful in the SPRINT? Why was it successful?

- The migration from mongoose to sequelize was successful for the backend of the project.

How can this success be repeated in the next SPRINT?

- Whenever a major issue has to be solved, if the whole team is included in the process, code will be delivered within time.

What went wrong in the SPRINT? Why did it go wrong?

- No major issues occurred that are worth mentioning.

How was the problem corrected or recovered?

- Nothing to report.

How could the problems be avoided in the next SPRINT?

- Nothing to report.

Third SPRINT documentation

Here you can put any complete or work-in-progress products that have been produced during the SPRINT.

Documentation Created:

- Localization
- Database Design
- Implementation