Butu- Food forest app Design Report

Manage tasks, map the forest & store knowledge

Enrique Ramos Adamik – s2960397
Luc Haaijer – s2960974
Thom Kastelein – s2749793
Konstantin Milev – s2925389
Ruud Rupert – s2938855
Hugo van Wijngaarden – s2836823

EEMCS Faculty
TCS Design Project
Group 17

Supervisor – Dennis Reidsma

April 16, 2025
University of Twente

Table of contents

1. Introduction

Voedselbos Boekelose Bleekgaard is a community-driven operation in Boekelo that strives to manage a food forest. It is managed by volunteers who take care of the many flowers, herbs, and other flora while attempting to reduce the effort required to maintain them. This is done by setting up a reasonably complex ecosystem that can continue to exist without extensive human supervision.

The community already has a system in place to handle the recording of the many plants and paths existing inside of the food forest. This is done by a subscription-based geographic information system called ArcGIS. Volunteers have used this system to navigate the premises, record the different plants, and partition them into dedicated zones. Additionally, information surrounding the harvests of the plants, their scientific names, and other facts have been put into ArcGIS. Although this system is functional, the ArcGIS software was not meant for this functionality, as inserting these facts into the system can be tedious to do in the field and requires duplicating information for each entity in question.

Although the forest is capable of maintaining its diverse vegetation for a prolonged time, volunteers still have tasks to perform to maintain the ecosystem as it is or to improve its accessibility. Currently, these tasks are mostly passed around in a group chat filled with volunteers, though this system does require a lot of mental note-taking and can be quite tiresome to manage. Additionally, the lack of a clear view containing the information surrounding these tasks can make it hard to meet deadlines since volunteers can forget their tasks or are only reminded of them when the deadline is approaching.

As such, this report discusses the development of a platform that integrates a map feature similar to the one provided by ArcGIS, storing the data about plants and objects that are inside the food forest while also allowing the management of tasks for volunteers to do. The intention is to produce a user-friendly web app capable of these functionalities.

2. Background

2.1 Project domain

In this chapter, the specific domain surrounding the project will be discussed. By analyzing the current systems in place, the stakeholders, and their demands, we aim to form an understanding of their expectations and possible implementations. As a result, our view of the fully functioning system will be more concise, preventing changes during later stages of development.

2.2 Domain food forest

The Bloekelose Bleekgaard is a 0.74-hectare field on which community members maintain a biodiverse food forest (https://boekelosebleekgaard.nl/). The forest is used as a semi-artificial ecosystem wherein various types of plant life grow and flourish in harmony. Besides plant life, the forest also contains multiple roads and small creeks, which both require maintenance from volunteers to prevent overgrowth from nearby plants or possible obstructions.

2.3 Domain ArcGIS

The volunteers currently make use of the software services provided by ArcGIS. ArcGIS Field Map is an app that is capable of providing users with data-driven maps that assist in location-based data-capturing and asset-finding (Esri, n.d.). Users are capable of adding objects of interest or tasks to be done, but also use this data in the field as their location is being tracked by the application. These functionalities make ArcGIS ideal for the purposes of Food Forest, though the scale of the Food Forest limits the usability for volunteers. The vast amount of geographical objects within the Food Forest prevents the UI from helping with searching for particular plants. The additional lack of a search bar to globally look for objects of interest immensely reduces the ease of use for the application. Facts that relate to a geographical object, such as its species or name, are also redundantly spread onto all instances of it, leading to difficulties as all instances would need their properties to be edited when modifying information. Furthermore, the application is not meant for task management, and as such, this functionality would need to be part of a separate system instead of being built on top of the pre-existing application. Finally, the cost of using ArcGIS's services disincentivizes its usage. Because of this, the client desires a transition to free and open-source software.

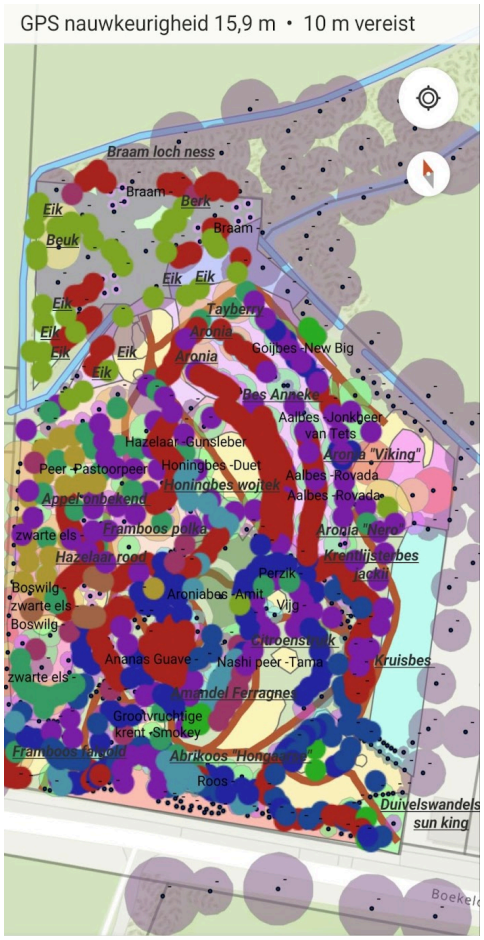


Figure 1 – Current ArcGIS-based system

2.4 Stakeholders

The future system will primarily be used by two types of users: volunteers and administrators. Administrators are those who are responsible for approving new tasks proposed by volunteers and approving new users. Administrators can also do anything that volunteers can, such as creating and editing tasks and knowledge-base information.

Volunteers are those who carry out the tasks given by administrators. To assist them with this, they need to have map functionalities similar to those provided by ArcGIS. Upon completing a task, they will need to be able to record this in the system. Volunteers might also be relied upon to submit tasks due to their presence in the forest as they have a better overview of the current conditions. Because of that, they may want to add tasks themselves. To prevent ambiguity when it comes to formed tasks, administrators are

required to assess these tasks first and potentially reject or modify them directly before acceptance.

2.5 Software deliberation

Geographical data

The system to be built can be implemented in numerous ways. One aspect to be considered is the choice of software for its development. As stated earlier in the domain analysis, the current system makes extensive use of ArcGIS to allow for navigating the food forest and recording information. The downsides to this have already been documented, and as such, we have attempted to consider other software which are free and open-source. During our initial interview with the client, we were heavily encouraged to make use of existing standards to enable better interoperability if required later on. Since both mobile and desktop support was required, in addition to programmability being important, the focus was kept on web platforms. Various frameworks were evaluated, such as Mapbox, MapLibre, OpenLayers, and Leaflet. An open-source alternative version of ArcGIS called OpenGIS was additionally investigated for use, though we ended up deciding on a more lightweight approach for our system. Due to its openness and programmability, we chose Leaflet. Besides being open source, it is also being actively maintained and has built up a large community of tools over its lifetime, which will assist with its integration.

Map objects are stored in the GeoJSON standard format, which enables easy migration from and to other platforms. Since ESRI/ArcGIS uses its own JSON standard, this was converted to GeoJSON using the well-supported terraformer/arcgis library. The script used for this can be found in the code repository of our project.

One hiccup that was run into was the use of Rijksdriehoekscoördinaten in the original dataset. By default, Leaflet makes use of WGS84 (EPSG:4326) coordinates and 'Web Mercator' projection (EPSG:3857). Unfortunately, it does not support the Dutch RD (EPSG:28992) standard. To complicate things further, the ESRI (arcgis) maps use a slightly different level of detail and origin, making things a few meters off. While Leaflet supports implementing custom coordinate systems, we have gone for the easier option of transforming the coordinate system in the initial transformation. We made great use of the coordinate reference transformation approach (Compuron, 2021) when dealing with Rijksdriehoek coordinates, which allowed us to use Leaflet's preferred coordinate system.

Lastly, various tile layer providers were evaluated. These are the actual images that make up the map. Originally, the default OpenStreetMap tiles were used, but these had no detail on the field. Since only a relatively small area is being mapped, and open data was a hard requirement, the tile layers from PDOK, part of the Dutch Kadaster, were used. These are made up of air photos with 7.5cm resolution and served to the application on demand over the OpenGIS Web-map-services (WMS) protocol.

App framework

Since our system will be used on both mobile by volunteers in the field and on the desktop by administrators, the usage of a web app will help speed up development and allow for cross-compatibility between devices. The implementation of this would be done by a full-stack *TypeScript* system running *ExpressJS*.

The choice for an express-based TypeScript backend was due to a few reasons. Firstly, it was chosen because most team members already had some experience with TypeScript or its counterpart, JavaScript. In addition, the TS/JS family of languages works well with web applications due to JS being heavily supported by nearly all browsers. Specifically, TypeScript, a superset of JavaScript, was chosen since having a typed language makes development in larger groups less error-prone in comparison to a dynamically typed language.

Front-end framework

For the front-end, it was chosen to again use TypeScript, together with our own custom ElementFactory framework. This framework enables components to be re-used across pages, type validation for HTML elements with TypeScript, and additional helper functions to simplify component creation.

A separate API layer has been created to allow easy access to the back-end API from any front-end code. This layer is responsible for sending requests and casting the responses to the right type. By separating the API layer, changes can be made in a single place instead of all around the project.

To build the TypeScript code, webpack is used because it is common, highly configurable, and fits all the needs for this project. As for styling, we made use of Sass, another superset but in this case for CSS, which is cleaner to use and also helps with larger-scale projects.

Database

PostgresSQL was chosen to manage our databases due to its robustness and scalability. Moreover, it is open-source and offers additional data types, which we made use of (such as UUIDs and custom enums), and richer JSON support. Additionally, all members of the team were already familiar with it, so it was a natural choice over other solutions such as *MySQL* and *MongoDB*. Other choices could have been made, but they would likely not have resulted in largely different results. Since a lot of the GIS data is dealt with as JSON, some NoSQL options were also evaluated, but from experience, these were expected to result in a worse development experience for the target application.

3. Planning

In this section, we outline the methods we used to plan our development of the project, including regular meetings with the client, an extensive specification of requirements, a Trello board, and a Gantt chart.

3.1 Client Meetings

At the beginning of the project, we interviewed our client, who is also our supervisor and an additional stakeholder. He is one of the volunteers managing the Food Forest and has also been in charge of the existing ArcGIS page for it. These fruitful discussions resulted in several requirements (covered in more detail in the next section) that were crucial to our planning process. In addition, we agreed to hold regular progress meetings with our client/supervisor every Tuesday morning.

3.2 Requirement specification

The requirements for the system were noted and ranked according to MoSCoW prioritization techniques (Must, Should, Could, Won't). These requirements were gathered (some of them in the form of user stories) after the aforementioned interviews with two stakeholders who are managers of the food forest. Additionally, they were reviewed and approved by one of those managers. These requirements were an artifact that we could reference later on to ensure that our development was properly focused.

Must-haves

These "must" requirements are required to be completed before the end of the module for the MVP and describe essential functionalities.

1. *Account system*
    - Users are able to create accounts using an e-mail address and password.
    - Users are able to log in to their accounts after creating them.
    - Users have different roles (admin, worker, volunteers).
        - The ways in which a user can interact with different parts of the system are controlled by their role.

2. *Geographic information system*
    - Geopositional data is stored in a centralized database.
        - Objects can be stored with a name, description, and position.
        - Areas/overlays can be stored with a name and shape.
    - Users (with appropriate permissions) are able to retrieve geopositional data.
        - These users can view this data within the web application.
    - Users (with appropriate permissions) can insert new data into the database using the web UI.
        - These users can insert new objects/POI.
        - These users can insert new areas.
    - Users (with appropriate permissions) can delete objects and areas from the database using the web UI.
3. *Centrally stored tasks*
    - Information about tasks is stored in a centralized database.
    - Users (with appropriate permissions) are able to insert new tasks into the database using the web UI.
    - Users (with appropriate permissions) are able to view what tasks exist in the system using the web UI.
    - Users (with appropriate permissions) are able to edit tasks that are already in the database using the web UI.
    - Users (with appropriate permissions) are able to remove tasks from the database using the web UI.
    - Task fields
        - Task entries contain a name and a description.
        - Task entries are able to contain a location.
        - A GIS object or area is able to be linked to a task entry.
        - Task entries contain an estimated duration.
        - Task entries contain a fulfillment period.
    - Users of the mobile frontend are able to add new tasks.
        - These users are able to provide a name and description for the task.
        - These users are able to pick a location to associate with the new task.
    - Users of the desktop frontend are able to provide additional details for tasks.
        - These users are able to specify an estimated duration.
        - These users are able to provide a fulfillment period.

Should-haves

These “should” requirements are not mandatory to be completed before the end of the module but should be considered of high importance for user convenience and or usability.

1. *Centrally stored tasks*
    - Users of the mobile frontend are able to add new tasks. *(cont.)*
        - These users are able to find closeby GIS objects to associate with the new task.
        - These users are able to take a picture and link it to the new task.
        - These users can assign tags to tasks
    - Task fields
        - Task entries are able to contain images.
        - These users are able to provide a checklist of subtasks.
        - These users are able to set a recurrence frequency.
        - Tasks automatically recur by the given recurrence frequency.
        - Tasks can be assigned tags
    - Mobile users are able to view a list of nearby tasks.
2. *Knowledge base*
    - Information about plants and POI is stored in a centralized database.
    - Information in the knowledge base is attached to POI on the map
    - Users (with appropriate permissions) are able to insert new entries into the database using the desktop UI.
    - Users (with appropriate permissions) are able to view entries in the knowledge base from both the mobile and desktop UIs.
        - These users are able to search for specific knowledge entries by their name.
    - Users (with appropriate permissions) are able to edit entries in the knowledge base using the web UI.
    - Users (with appropriate permissions) are able to delete entries from the knowledge base.

Could-haves

These “could” requirements are not mandatory to be completed before the end of the module but could be of use for users. They could be implemented, but due to time constraints or lack of importance, they should be held for later.

- Users are able to view a timeline visualization of upcoming tasks through the desktop UI.
- Users are notified of upcoming tasks.
    - Users are notified by email of upcoming tasks.
    - Users receive push notifications about upcoming tasks.
- Users of the desktop UI are able to view aggregate statistics about the system.
    - The interface contains the total number of GIS objects.
    - The interface shows an overview of how many tasks were completed each week, month, and year.
    - The interface shows the names of the most active users.
- The GIS data of the old system is transferred to the new system.
- Make the project deployable as a Docker service to allow for easy setup.

Won’t-haves

These “won’t” requirements are out of scope for this project and will not be sought out to be implemented. They would be of use to the users but require massive overhauls, time to develop, or prolonged maintenance to be done well after the system has been developed.

- The system will not support multi-tenancy. This means that every organization that wants to use the system will have to host their own instance of it.
- The system will not support in-forest pathfinding. When someone uses the system, we assume they already know how to navigate around the premises.
- The system will not have a pre-filled knowledge base. Any data for it must be provided by the client.

3.3 Gantt Chart

Following a recommendation from our supervisor, we drew up a Gantt chart to keep us on track during development. The goal of such a chart is to visualise how long the development of different elements of the project would take and how they would overlap. This was kept up to date throughout the project.

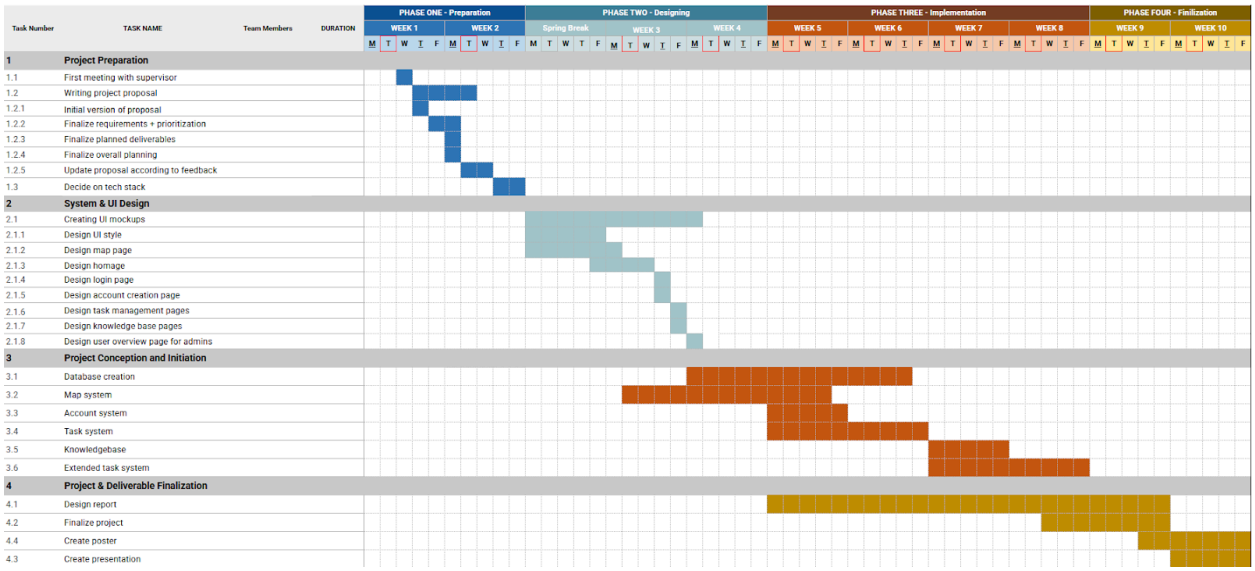


Figure 2 – Gantt chart for organisation

3.4 Trello Board

To ensure good communication when it comes to task allocation, we prepared a Trello board. We created cards to communicate tasks to be done and which member is responsible for which task. Besides work distribution surrounding the requirements we have documented, the Trello board additionally allows us to quickly record newly found bugs and issues, which can then be handled as any other task.

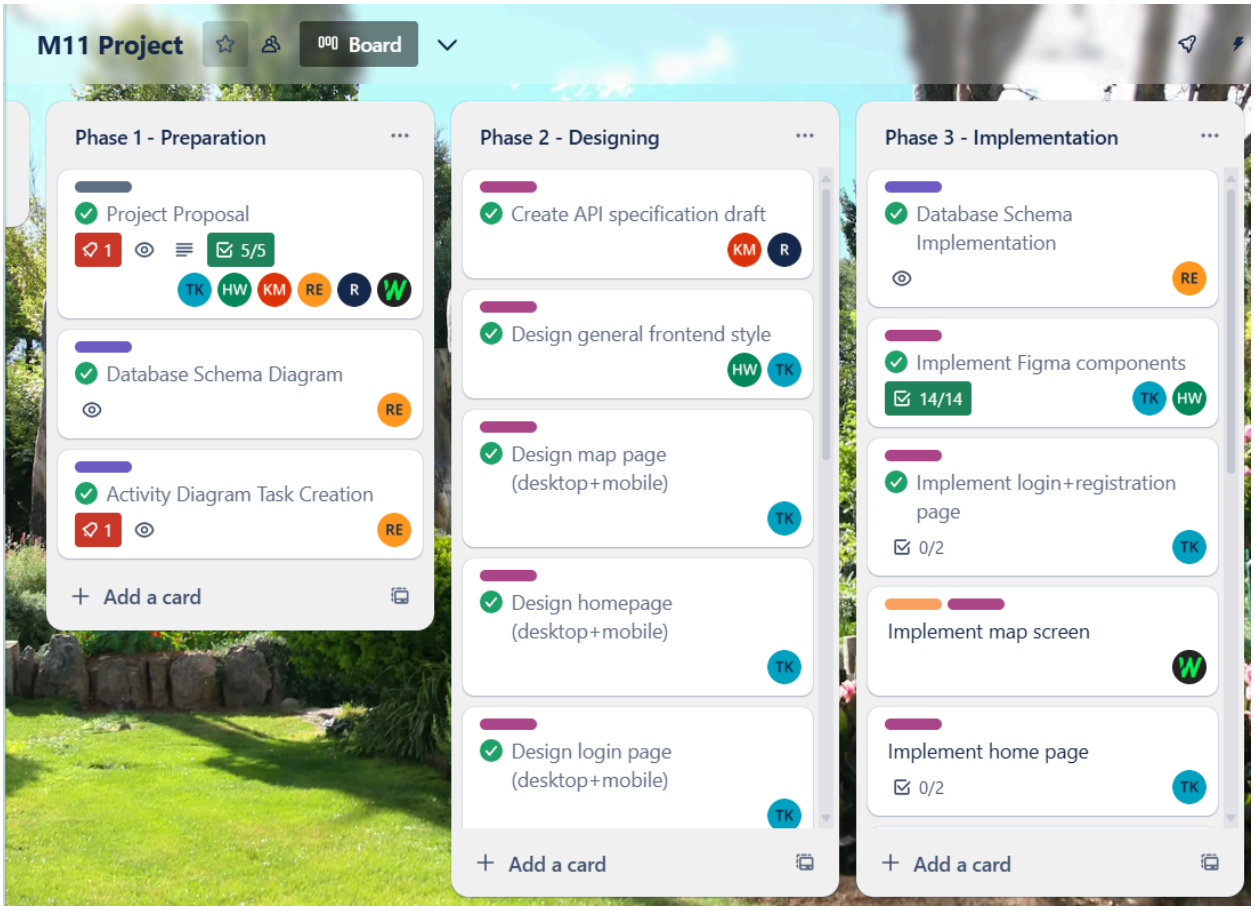


Figure 3 – Trello board for management

3.5 Risk Assessment

Finally, we conducted a risk assessment where we evaluated the possible risks that could potentially impede the process of development and addressed how we could mitigate them if they happened to occur.

Risk	**Mitigation**
The team runs out of time	Careful scheduling, reasonable management of scope, and focussing on our priorities.
Internal team conflicts	Internal mediation and, in case of escalation, communication with the supervisor.
Unequal distribution of work	Communicate with teammates who are reluctant to work together on a solution that satisfies everyone.
Security/use concerns	Follow industry standard practices. Adhere to security by design Ensure secure authentication.
Technical difficulties	Look for help from other team members who may have more experience. Seek advice from our supervisor in case of persistent difficulties in meeting requirements.

4. Design

In this chapter, multiple concepts surrounding the design of the system have been noted and described. These range from the processes that take place within the front-end design to the back-end database models.

4.1 Processes

Our system is capable of providing numerous functionalities to its users. For the essential features, the processes have been described in reasonable detail with the use of diagrams.

Task management

Task management is one of the primary components of our system, so we designed several activity diagrams to determine how it should function. The exact functionality of the task system was iterated upon during development, and so these activity diagrams have been revised to be more accurate to the final system. Older versions of these diagrams can be found in Appendix I.

One of our “must” requirements requires the ability to make tasks. Tasks are created by volunteers and administrators, either out in the field with the mobile version of the app or with the desktop version. When a task is created out in the field, only some of the fields need to be filled out since our system prioritizes fast and easy task submission. Once submitted, the initial task can be reviewed and edited in more detail by managers. Once a task has been accepted, it can be completed by volunteers. These can view the available tasks to be performed and assign themselves to them. Finished tasks can be marked as complete.

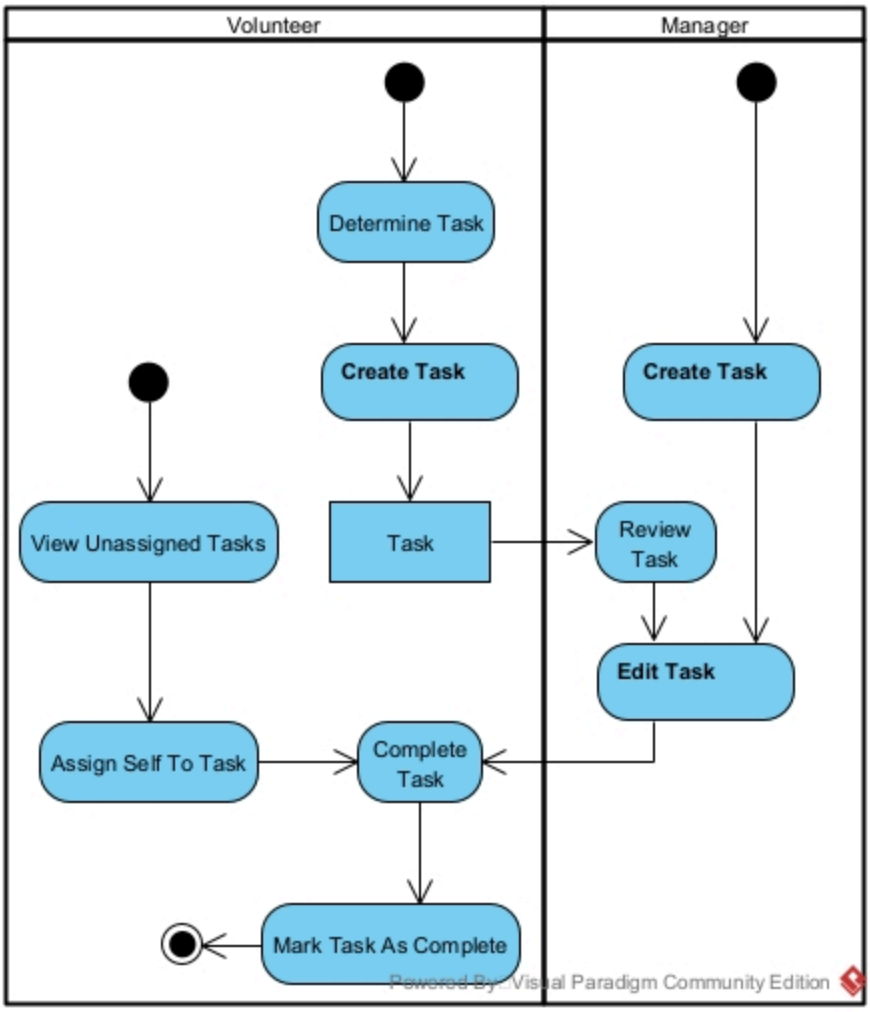


Figure 4 – Activity diagram of task management

The creation of a task first requires a location to be chosen, either by selecting a pre-existing object (such as a plant or path) or by picking a desired arbitrary location on the map. From here, users can optionally choose to assign a name, description, or files to the task before saving it. If no name is chosen, the task is saved with a default name. Files that can be uploaded are images, videos, and audio.

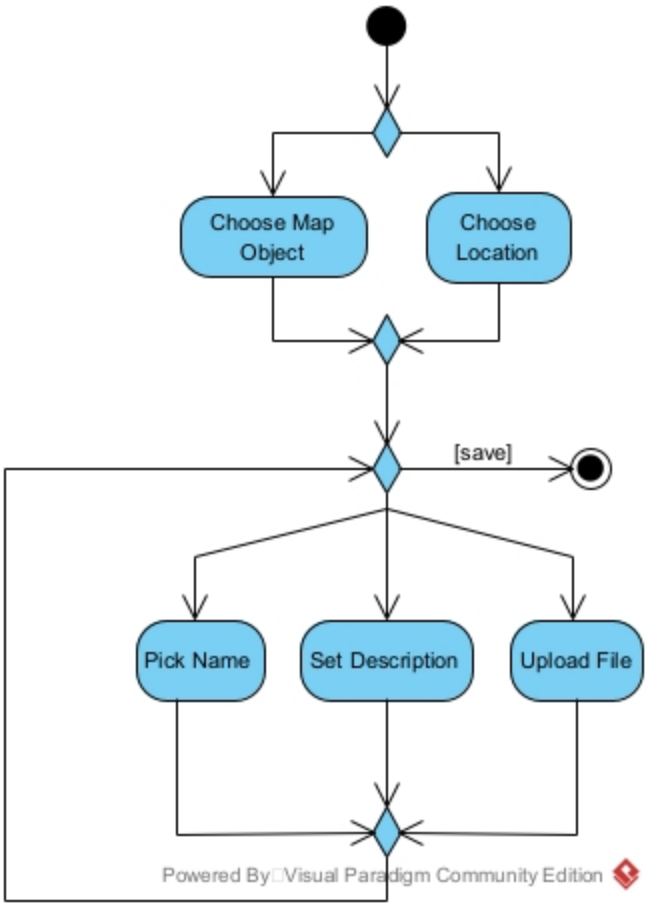


Figure 5 – Activity diagram of task creation

Editing a task allows all of the elements of a task to be modified. This includes changing the name or description, adding or removing tags, setting the urgency of the task, modifying start and end dates, selecting how often the task will reoccur, uploading/removing files, and finally, choosing which objects on the map should be associated with the task. All of these fields are optional. Due to its complexity, editing tasks is primarily intended for the desktop version; creating tasks is much easier and therefore better suited for mobile work in the food forest.

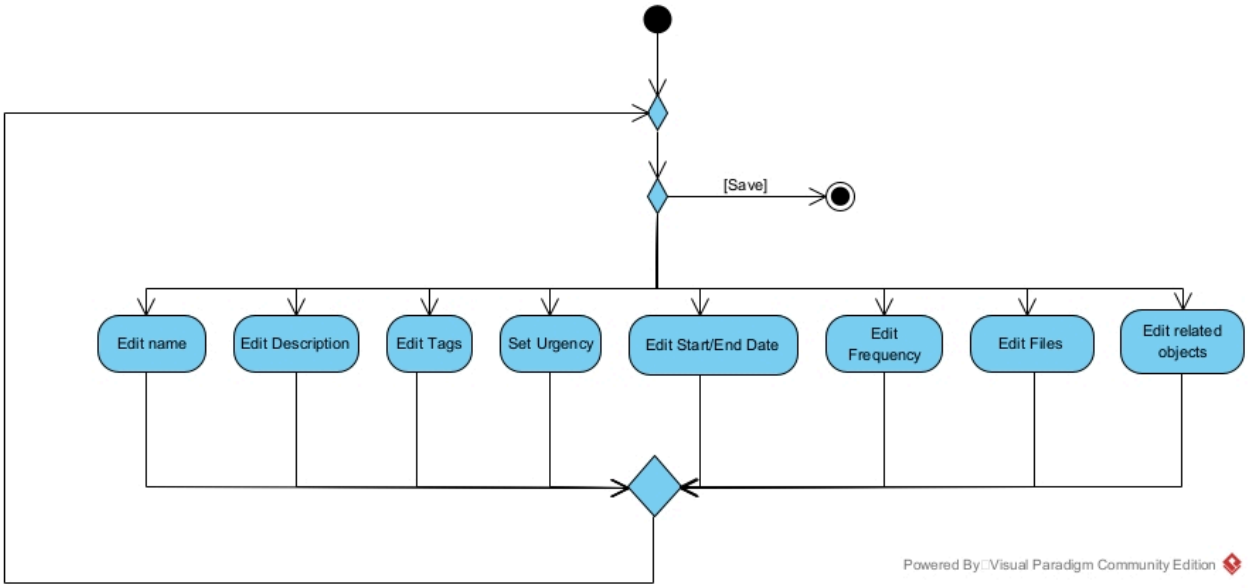


Figure 6 – Activity diagram of task editing

Task lifecycle

The following sequence diagram represents the lifecycle of a task. Volunteers create tasks either on mobile or on desktop; tasks created on desktop can be provided with more details. The system receives the new task and confirms that it is now “pending”. Pending tasks must be approved by an administrator, who must choose to confirm it. The administrator can also edit the task however they desire. The task is then saved as “unassigned”. Afterwards, willing volunteers may choose to choose the unassigned task and assign themselves to it, thereby making it “assigned”. Finally, after the task has been carried out, the volunteer can mark it as “complete.

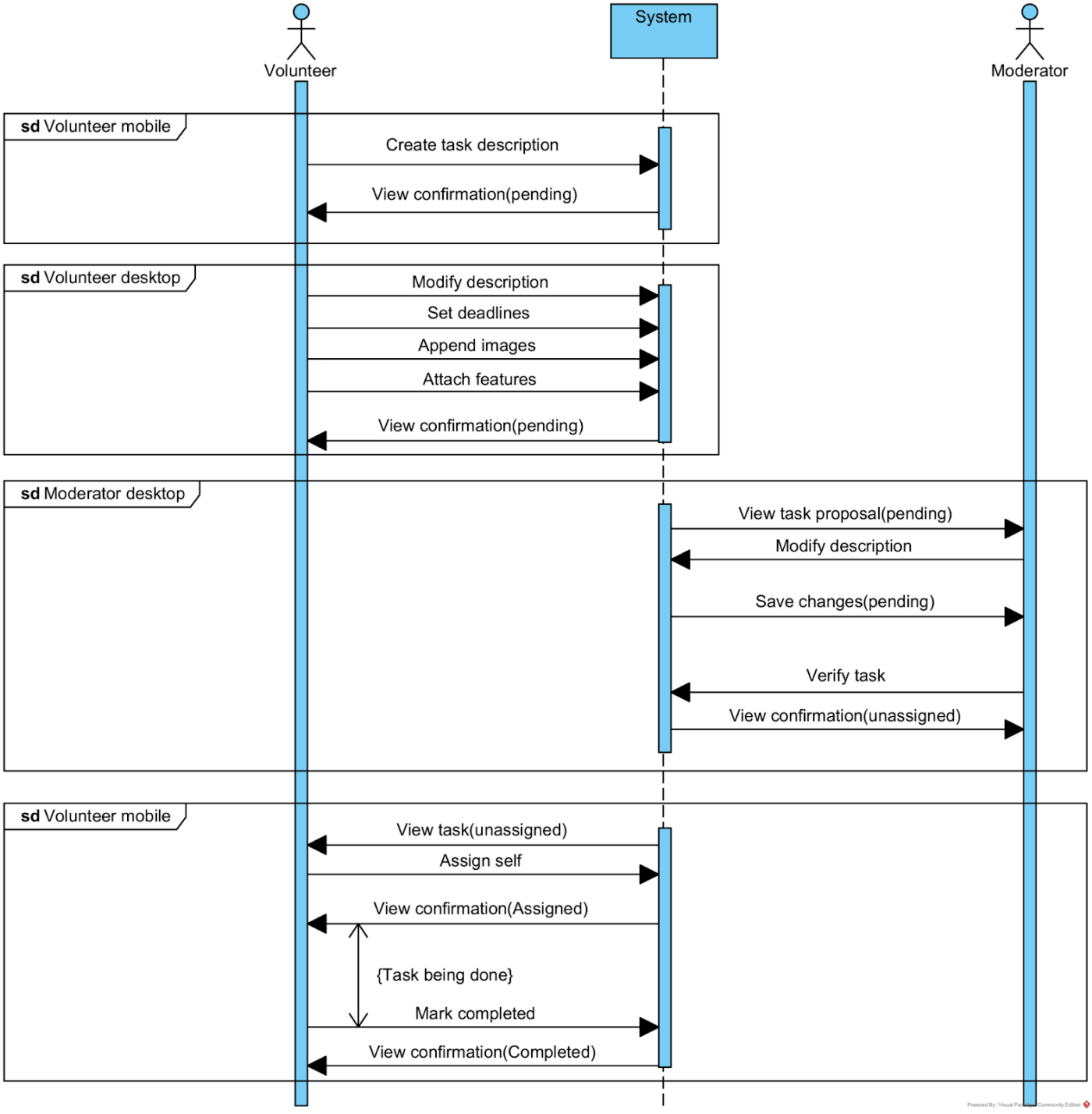


Figure 7 – Sequence diagram of task lifecycle

Knowledge Entry Creation

Knowledge entries exist separately from the objects which belong to them. A knowledge entry can exist independently at first, containing only a title and description. Afterwards, GIS objects can be attached to it, each of which can only belong to one knowledge entry. Besides geographical objects, multiple facts representing small elements of knowledge can be attached to a given knowledge entry. Finally, knowledge must be saved manually.

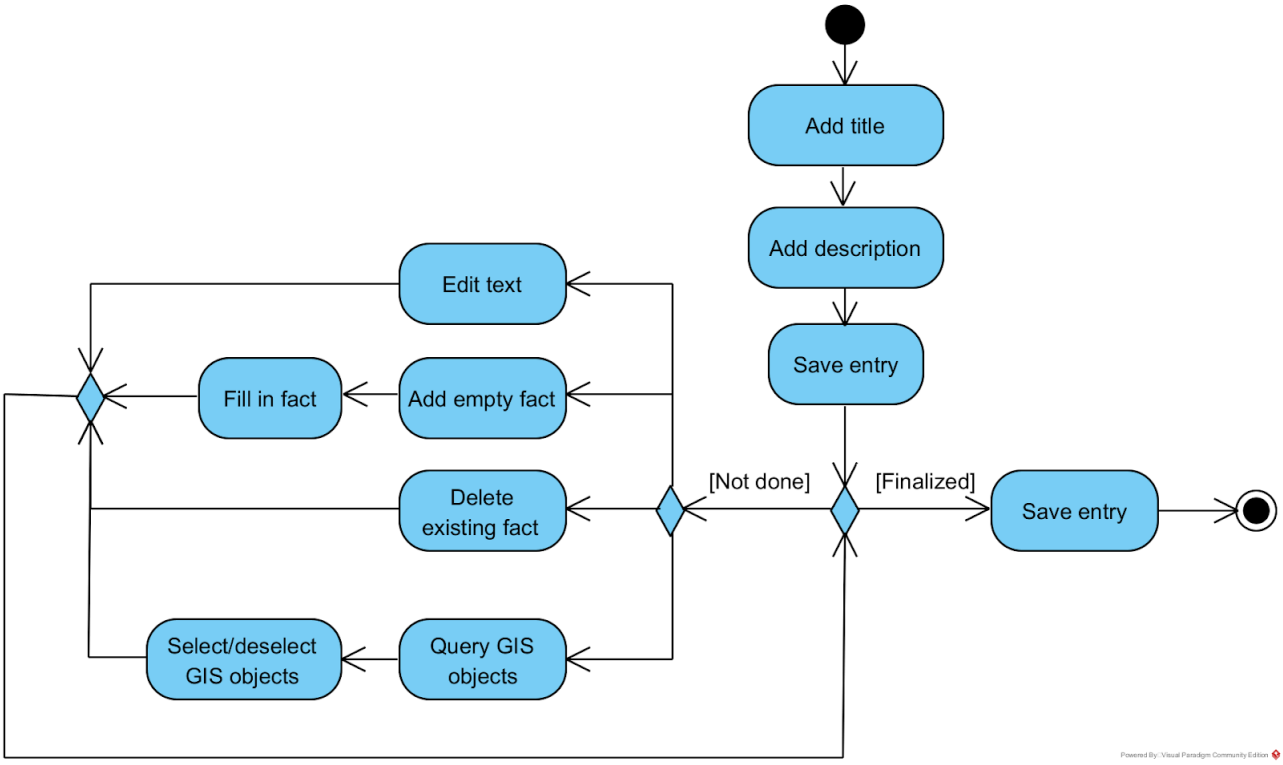


Figure 8 – Activity diagram of creating a knowledge entry

4.2 Backend

For the backend, our team decided on a Node.js environment along with the Express framework. Both are very common and well-supported. We considered alternatives, such as a Java server or the Python-based Django framework, but ultimately, we chose Express due to its minimalism. Additionally, we chose to utilise TypeScript, an extended version of JavaScript that is type-safe and is transpiled to JavaScript at runtime.

API

As mentioned earlier, the express JS framework was used, which allows for easy routing to RESTful API endpoints that provide convenient resource identification. We created a specification to define which endpoints we would be creating; this specification was essential for keeping the API consistent and coordinated. A set of endpoints was assigned to each main database table: Tasks, GIS, Tags, Knowledge, Files, Users, and Auth. For each endpoint, we specified the form of the path and what data would be sent or received.

Type	System	URL	Receives (JSON / Path params/ Header)	Returns (JSON)	Status	Purpose

Post	Auth (ready)	api/auth/register	JSON {
			name: string,
			email: string,
			password: string
			}
				Cookies: user_id, token	200 -> Redirect	Creates an account
						No authentication required

Put	Auth (ready)	api/auth/update	token, { new_password: string }	*Cookies: token*	200 -> Redirect 401 -> Unauthorized	Update password
Post	Auth (ready)	api/auth/login	{ email: string, password:string }	*Cookies: user_id, token*	200 -> Redirect	Login into an account **No authentication required**
Get	Auth (ready)	api/auth/check	-	success, message?, user_id?	200 -> Received data 401 -> Unauthorized	Returns current login status
Post	Auth (ready)	api/auth/logout	Cookies: token	-	200 -> Redirect 401 -> Unauthorized	Logout
Get	Users (ready)	api/users	-	userid:{ user_id: string, name: string, email: string, role: string, tasks?: number[] }[]	200 -> Received data 403 -> Insufficient permission 401 -> Unauthorized	Returns data of all registered users, including admins and self **Admin only** **Recently changed**

Get	Users (ready)	api/users/{id}	-	{
				user_id: string,
				name: string,
				email: string,
				role: string,
				tasks?: number[]
				}
					200 -> Received data
					403 -> Insufficient permission
					401 -> Unauthorized
						Returns data of a specific user.
						If admin, can fetch other user IDs
						Admin only
						Recently changed

Put	Users (ready)	api/users/{id}	{ role: string }	-	200 -> Received data 403 -> Insufficient permission 401 -> Unauthorized	Updates role of user (userid) **Admin only** **Recently changed**
Delete	Users (ready)	api/users/{id}	-	-	200 -> OK 403 -> Insufficient permission 401 -> Unauthorized	Deletes a user **Admin only if other user ID**

Figure 9 – Example of API specification

Database

Furthermore, the system required a way to store data, such as tasks and users, so that it could be used later. To this end, we implemented a PostgreSQL database. The decision to use Postgres was made on the basis that it is a popular, open-source option that supports JSON types, which would be useful for storing data about geographical objects. Below is an entity relationship diagram of the final database schema. Our database schema was updated several times during development; the initial version can be found in the appendix.

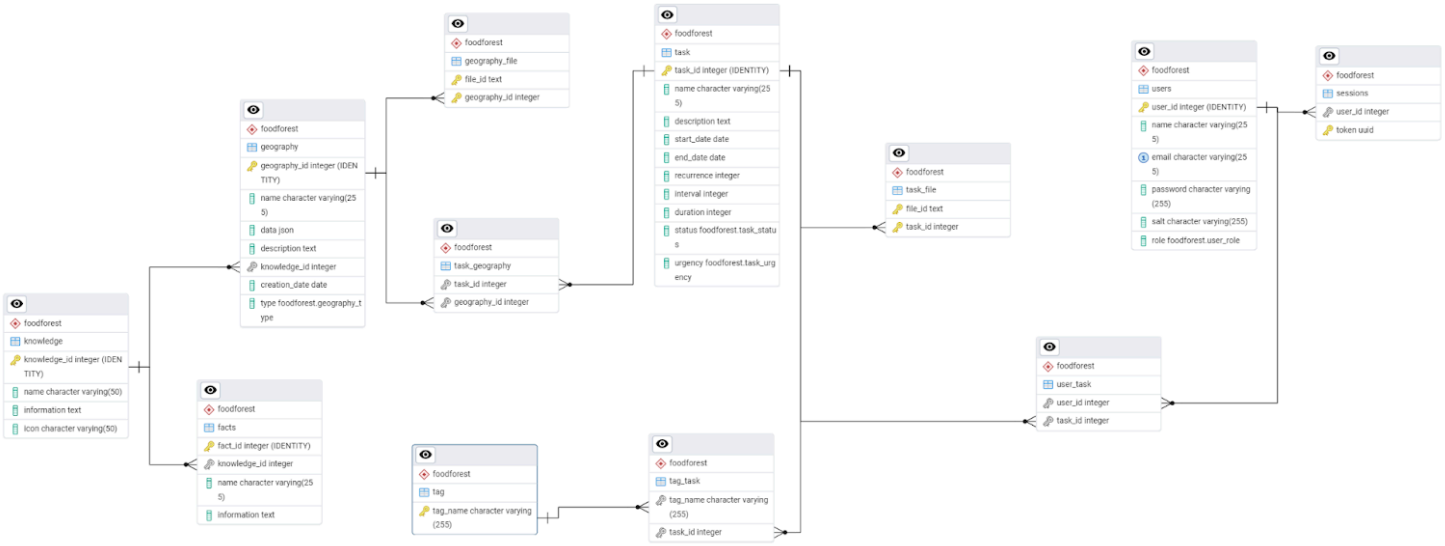


Figure 10 – Final database diagram

Each task in the schema must have a *name*, *task_id*, *status*, and *urgency*, but the other columns are optional. One major concern surrounding tasks is how to document their deadlines and possible repetitions. Notably, instead of just one date, there is a *start_date* and *end_date*; this allows tasks to span variable amounts of time since some tasks must be done immediately while others might simply need to be done within the week or even the season. Additionally, there are columns for recurrence interval and frequency for recurring tasks. There is no table for subtasks, as we didn't have time to implement that feature. The *status* and *urgency* columns have custom enum types to allow for recording in what state a task lies in and if volunteers should prioritize it. Additional files related to a task, such as images, are stored using a *file_id*, which is recorded in the *task_file* table. This table has a *task_id* foreign key to relate it with tasks.

Tasks can also have tags, which is a many-to-many relationship supported by the tag_task associative table. Furthermore, users are also connected to tasks, again with an associative table. This allows volunteers to be associated with multiple tasks. Geographical objects also have a many-to-many relation to tasks, as one task could be part of multiple objects, or a geographical object could have multiple tasks related to it.

Geography entries must have a *geography_id*, a name, geographical data (in JSON) and a type, which is an enum. Additionally, they can have a description, and they get an automatically generated creation date. The data column holds the GeoJson data of the

object, such as its coordinates. Like the task table, the Geography table is related to a *geography_file* table that stores *file_id*-s for geographical objects.

The *Knowledge* table holds knowledge base entries and has a one-to-many relationship with a *Geography*, since, for example, many of the same plants would reference the same knowledge. *Knowledge* has a name, information, and an icon, which relates to a string that corresponds to a Google Web Font icon. There is also a *fact* table, which has a many-to-one relationship with *Knowledge*; facts are small, specific pieces of information that can be added to a *knowledge* base entry.

As stated earlier, there is a separate table for files belonging to *Geography* and *Task*. The goal here is to avoid a table with two foreign key columns where one would always be null. Having two separate tables is a cleaner solution.

The *Users* table has columns for *user_id*, *name*, *email*, *password*, *salt*, and *role*, none of which are optional. The role field is an enum, and the email field must be unique. There is also a *Sessions* table, which stores unique tokens to represent active user sessions. These tokens are automatically generated UUIDs (universally unique identifiers). *Sessions* have a many to one relationship with *Users*, since one user may have multiple independent sessions, such as one on mobile and another on desktop.

4.3 Frontend

Mock-up

The views that we intend to showcase to users have been modeled within Figma. This mock-up consists of a desktop version, which is meant for administrators, and a mobile version to be used by volunteers. Although the Figma is a conceptualization of what the system looks like, the actual implementation could see alterations due to constraints or, if possible, improvements on the design. On a side note, the letters BUTU are derived from the Dutch words “Buren tuin”, which translate to “Neighbourhood garden”.

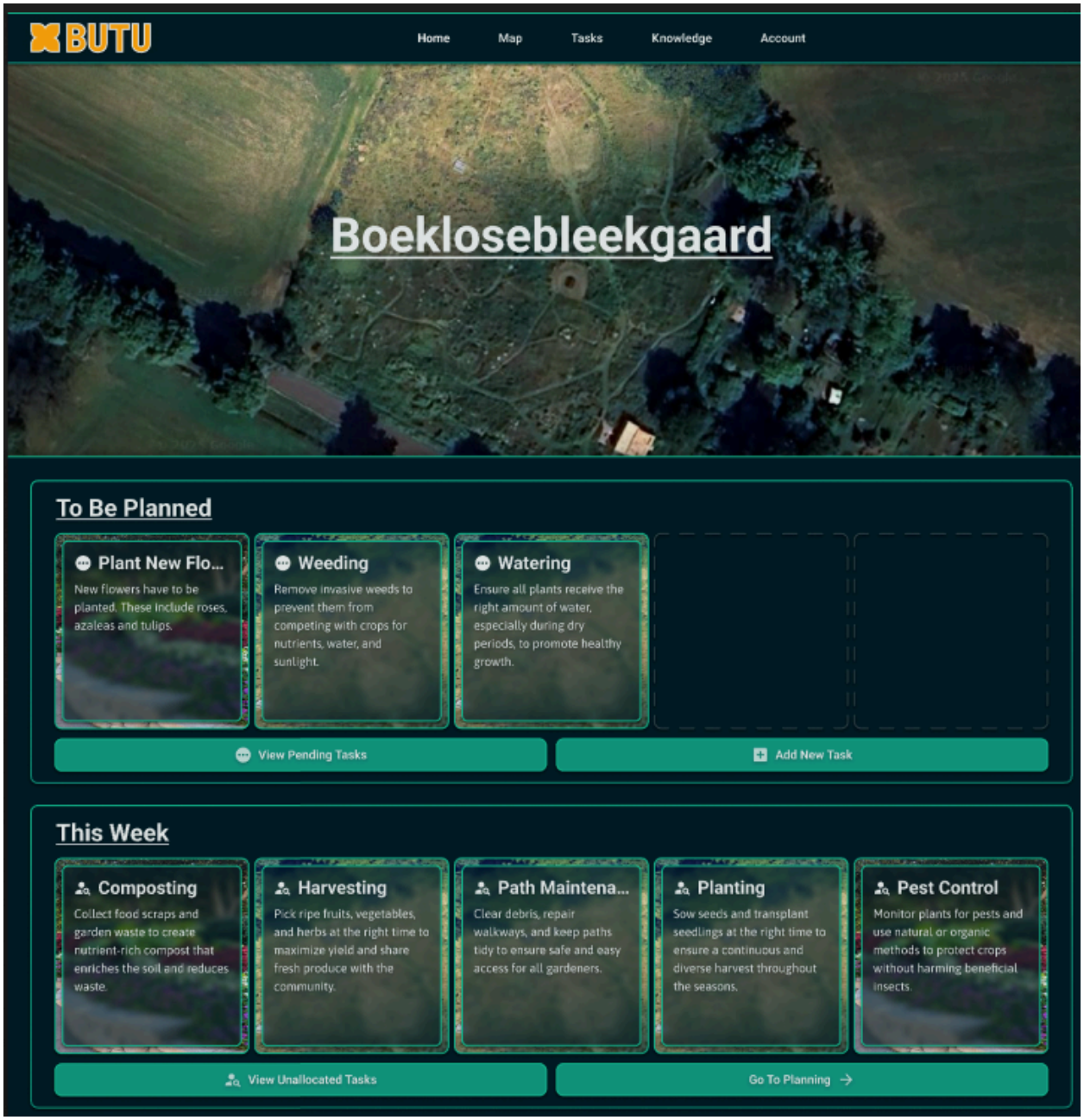


Figure 11 – Example frontend mockup

This is an example of one of the Figma mock-up pages that we created. Our final frontend looks quite similar to the mock-ups, although there were some changes as we iterated on the original frontend. The full mock-up is included under Appendix I: Full Mock-up.

Mapping

For mapping, we have already decided on using Leaflet to view the Food Forest. Leaflet is a popular, open-source option for making maps. The open-source nature of Leaflet is very important since one of the issues with the current system is that it relies on ArcGIS for the map, which is expensive and closed-source. To check the feasibility of this, we prototyped a strict client-side version that is capable of viewing, creating, and deleting new geographical objects or layers. Since this prototype possesses no backend, the data is only stored in local-memory, though modifying it to send requests to a server which will store the elements on a database would require only minor alterations.

Figure 12 – Initial mapping prototype

Framework

To build the frontend of our application, rather than relying on an existing framework (React, Svelte, etc.), we decided to utilize the tools that we had previously built ourselves. This way, we could use code that we were already familiar with, thus speeding up the development process. These tools consist of two main libraries:

1) *A library for checking the types of variables at run-time.*
   As a consequence of TypeScript having to be transpiled to JavaScript so that it can run in a web browser, the custom types we define in our code are inaccessible during run-time. However, using this library, we have defined predicate functions that can verify that incoming data is actually of the data type that we expect it to be.

With this, we were able to catch type errors as soon as possible, which is the main reason we chose to use TypeScript over JavaScript.

2) A library for creating DOM objects.
   This library has a focus on the creation of reusable builder objects specifically for DOM elements. Normally, the code that would create these DOM elements is long and explicit. With these builder objects, however, most of this code can be abstracted away. The builder objects are then able to be reused to more efficiently create elements whose contents are similar. In practice, this allowed us to rewrite the components from our Figma design as these builder objects. Through this, we ensured a consistent design that aligns with our original vision.

```
const buttonText = document.createElement("h3");
buttonText.textContent = "Click Me!";
buttonText.classList.add("bold");

const button = document.createElement("button");
button.toggleAttribute("clickable", true);
buttonText.appendChild(buttonText);

button.addEventListener("click", () => {
    button.classList.toggle("red");
});
```

Figure 13 – Default code structure

```
// create builder object
const ASM = EF.button({ text: "Click me!" })
    .attribute("clickable")
    .children(
        (_, params) => EF.h3({}, params.text)
            .classes("bold")
    )
    .on("click", (_, self) => {
        self.classList.toggle("red");
    });

const button1 = ASM.make();
// make another
const button2 = ASM.make();
// change text
const button3 = ASM.make({ text: "Click me too!" });
```

Figure 14 – Applied builder pattern

5. Testing

In this chapter, the types of testing we have implemented are discussed. In general, we describe the unit testing we have performed during development, the testing of multiple modules, and usability testing to check if the features provided are up to par with the client's expectations.

5.1 Test plan

For testing we have had multiple objectives in mind: specifying the intended behaviour of our code, checking the correctness of our code, and having high enough coverage to test as many different lines of code. To achieve this, we focus on the API HTTP calls, as the majority of our code is present within these functions. We intended to have a broad range of coverage for our system though achieving full coverage would be difficult.

5.2 Unit testing

Unit testing describes the validation and testing of isolated functions. For this, we use the inbuilt test runner provided by Node.js, as it is well-suited for asynchronous code. The majority of our unit testing involves the routing functions and API endpoints. Although we refer to it as unit testing, throughout the tests, overlap does happen between the separate modules. Especially when it comes to tasks and knowledge entries as they can interact with GIS features.

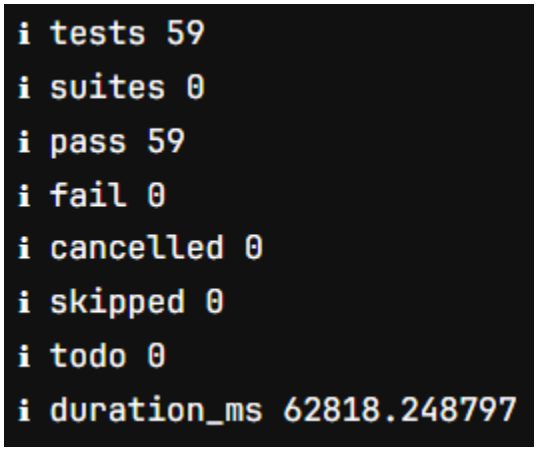


Figure 15 - Test results

To further help with code correctness, unit testing has been integrated within the project's GitLab pipeline. After the project was compiled and the frontend was built, the system would run the unit tests before deploying within the hosted Docker environment. This helped with figuring out which commits led to faulty behavior and where to start with debugging.

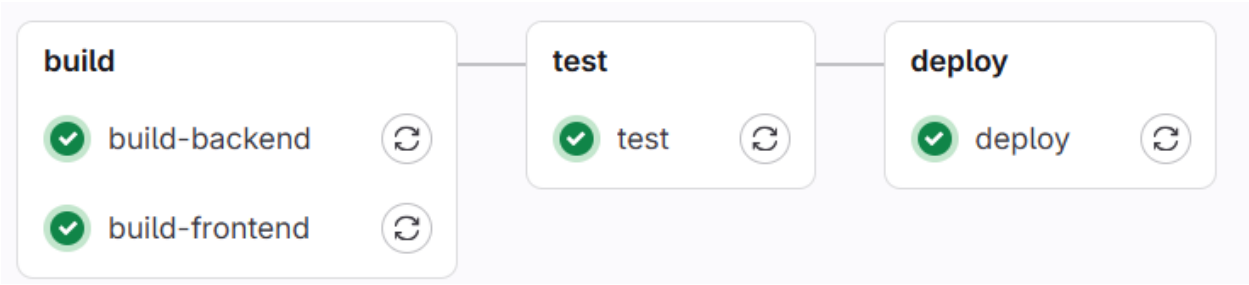


Figure 16 - CI/CD pipeline

5.3 User testing

Next to automated testing, which was mostly oriented towards testing the correctness of the program, continuous user testing is also a vital part of the process.

Weekly meetings took place with one of the stakeholders to showcase the current state of the application. This gave us feedback on features that may need more work or potential bugs to be fixed. Besides meetings with our client, through meetings with other groups, we were able to gather feedback from other students. This gave us valuable insights, mostly on the UI/UX field of our system.

Before the final release, we will go over the end product to find any last bugs and to determine if all the requirements were met. We will also have a last meeting with the customer to determine if the product works as envisioned and to iron out any small changes.

5.3 Linting

To reduce the number of bugs from coding mistakes, ESLint is enforced for all commits. The main use of this tool is to detect likely problematic design patterns and requires them to be improved before code can be accepted.

6. Implementation

In this chapter, the final implementation of the system is discussed and evaluated. Further considerations are taken to review the testing results and the changes made since the originally designed concepts.

6.1 System backend

We use Express for back-end routing to create different routes for API endpoints. The index.ts back-end file defines which routes are available and acts as middleware that checks if users are authorized. Each main database table has a corresponding API route (e.g., API/tasks for the Task table operations). There is a TypeScript file for every route, and in these files, the API endpoints are defined for various HTTP methods. These endpoints use the 'pg' client to send queries to the database as prepared statements, which prevents SQL injection attacks.

Authorization is done using UUIDs: universally unique identifiers. Whenever a user logs in, a uuid is generated as a session token and stored in the database. This token is sent to the user as a cookie when they log in or register. Authorization is done every time the user makes a request to the backend, using their token against the tokens in sessions to verify if it belongs to their account.

A Docker environment is used to allow easy development and deployment across different working environments. Docker automatically sets up the application and a database from the *schema.sql* file. Four docker-compose files have been created: Development, Production, Convert, and Test. Development enables automatically recompiling files on change, production compiles everything only once for the best performance, Convert seeds the database with ArcGIS data, and Test runs all the unit tests.

Use is made of the Continuous Integration / Continuous Deployment model. This means that on every commit, the program is type-checked, linted, built, and tested. Only when all tests pass is the program automatically deployed to a virtual machine over SSH. This workflow is implemented using the GitLab CI platform.

6.2 System frontend

Front-end pages utilize Typescript and SCSS, which must be compiled into JavaScript and CSS before they can be used. SCSS is a superset of CSS that allows good styling to be implemented more easily such as by allowing for the nesting of style sheet properties within each other.

The use of TypeScript instead of just JavaScript allows for much stricter type checking and, therefore, more reliable pages. In addition, it allows for the use of the aforementioned ElementFactory framework, which allows for complicated HTML elements to be easily created by specifying them in TypeScript by using the provided methods.

Besides the framework, there are also several API helper classes written in TypeScript using our own type-checking library. These classes provide useful methods and types for each API resource, allowing the endpoints to be accessed more easily. The type-checking library ensures that information sent or received from the API is as expected. It does this by providing a number of methods that automatically check the type of each field.

```
    export namespace Entry {
        export const checkType = getMappedChecker({
            task_id: isNumber,
            name: isString,
            description: nullable(isString),
            start_date: nullable(isString),
            end_date: nullable(isString),
            recurrence: nullable(isNumber),
            interval: nullable(isNumber),
            status: isStatus,
            urgency: isUrgency,
            tags: getArrayChecker(isString),
            files: getArrayChecker(isString),
            geographies: getArrayChecker(isNumber),
            users: getArrayChecker(isNumber)
        });
        export const safeCast = TypeChecker.cast(checkType,
"TasksAPI.TaskResponse");
    }
```

Figure 17 – Example of type-checked API type

6.3 Limitations

Though the system is capable of most of the requirements given, numerous restrictions are still present. For one, users are limited in potential roles as currently only administrators and volunteers are present. A more fine-grained rule set would help with preventing bad actors or regular users from (accidentally) modifying entries within the system. Additionally, individual geographical objects do not have dedicated pages, which would have aided in the management of the map. Another restriction would be the lack of heavy caching. Although our application already caches requests when possible, querying the map and the numerous features present within it can be quite bandwidth-heavy. This conflicts with mobile device users as they tend to want to keep data usage to a minimum. Even when taking URL redirects to a minimum and attempting to reuse as much of the already retrieved data as possible, complexity becomes quite large and reduces the simplicity of the code. This is especially the case on the front-end where multiple views are present (mobile or desktop), which require different functionalities.

6.4 Conclusion

The final system meets all of the must-have requirements. It is capable of keeping track of all geographical objects of the food forest and any tasks that need to be done. All of this information can be displayed on a map of the food forest. In addition, data from the old system has been imported. Tasks are versatile and can be assigned to geographical objects and given start/end dates. There is a distinction between volunteers and administrators. A working knowledge base allows more detailed information to be recorded. Finally, the website works both on desktop and mobile, with a comfortable interface.

6.5 Future

The system can still be improved upon in numerous ways by adding functionalities that we had to omit due to time constraints. These include but are not limited to: multi-tenant system so that the system can hold multiple segregated but concurrently running applications, abstraction to a general task management system so that it can be applied for other map and task-based objectives (not just garden related ones), and live updating from external systems in case the old system still wants to be kept in use (regularly updating the data to include new ArcGIS entries). More minor improvements could be made to the visual aspects of the UI both on mobile and desktop as deadline visualization could be worked on to convey urgency better.

7. Individual contributions

While the design project was very much a team effort, some work was split up to better fit each student's skill-set and learning goals. However, sometimes, overlap was present, and members helped one another as problems presented themselves. We roughly divided the work as indicated in the table and were content with the work every member delivered.

Enrique	Report, Backend, Frontend (map, global), Diagrams
Luc	Report, Frontend (map, global), Infrastructure, Prototyping
Thom	Report, Frontend (tasks, global, framework), UI design
Konstantin	Report, Backend, Infrastructure, Security
Ruud	Report, Testing, Frontend (knowledge), Diagrams
Hugo	Frontend (admin, global), UI design

8. Bibliography

Confuron (2021). *Leaflet & Rijksdriehoek*. Retrieved from: https://www.compuron.nl/leaflet/index.html

Esri. (n.d.). Field Data Collection App for mobile workers | ArcGIS Field Maps. https://www.esri.com/en-us/arcgis/products/arcgis-field-maps/overview

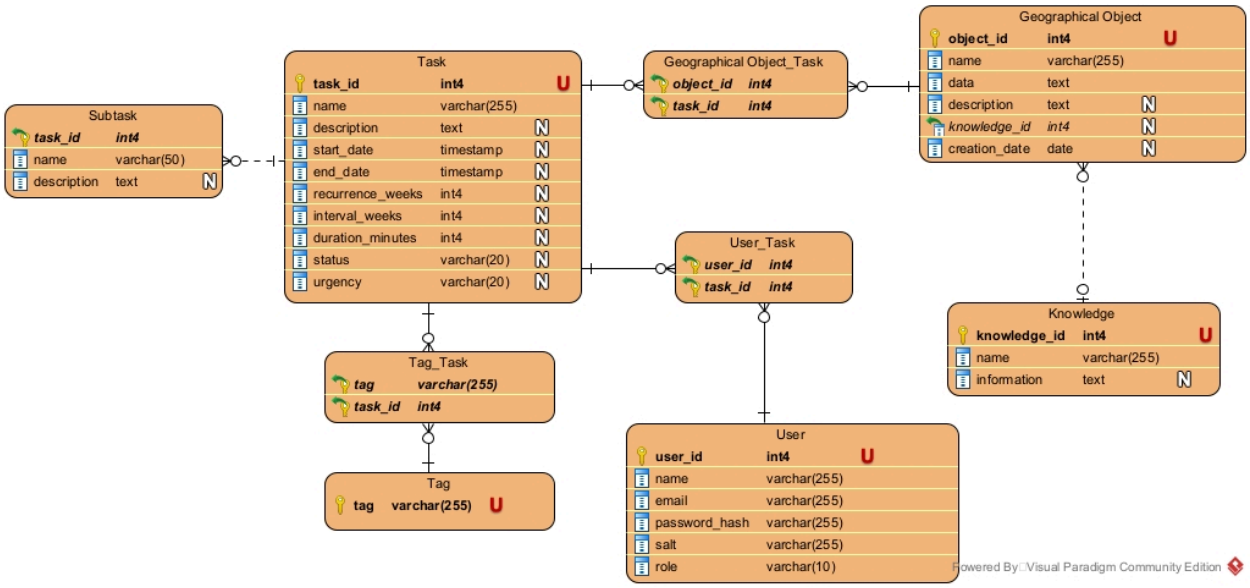
The illustration on the cover of this document is sourced from undraw.co by Katerina Limpitsouni.
https://undraw.co

9. Appendices

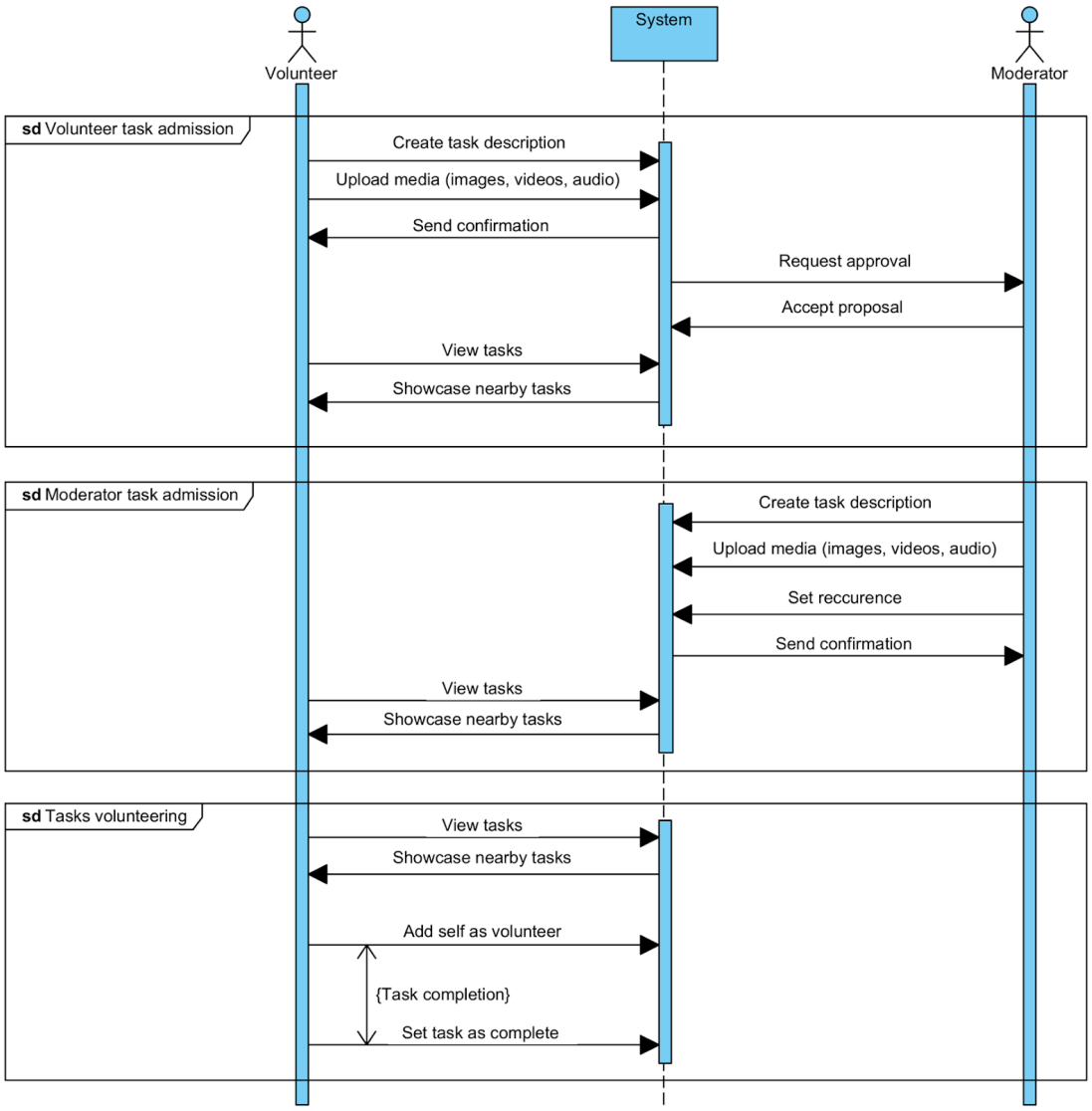
- Appendix I: Additional Diagrams
- Appendix II: AI Statement
- Appendix ||: Manual

Appendix I: Additional diagrams

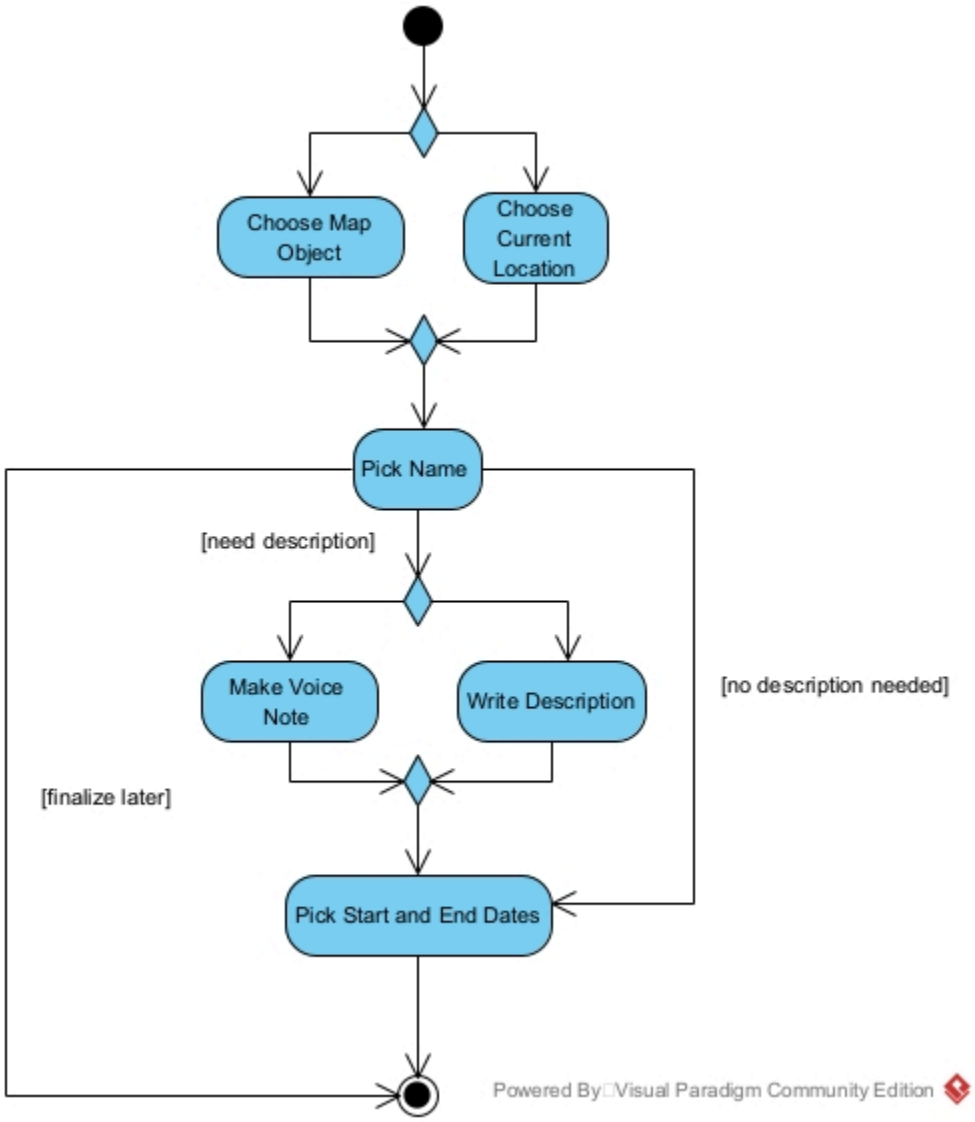
Initial Database ERD Diagram



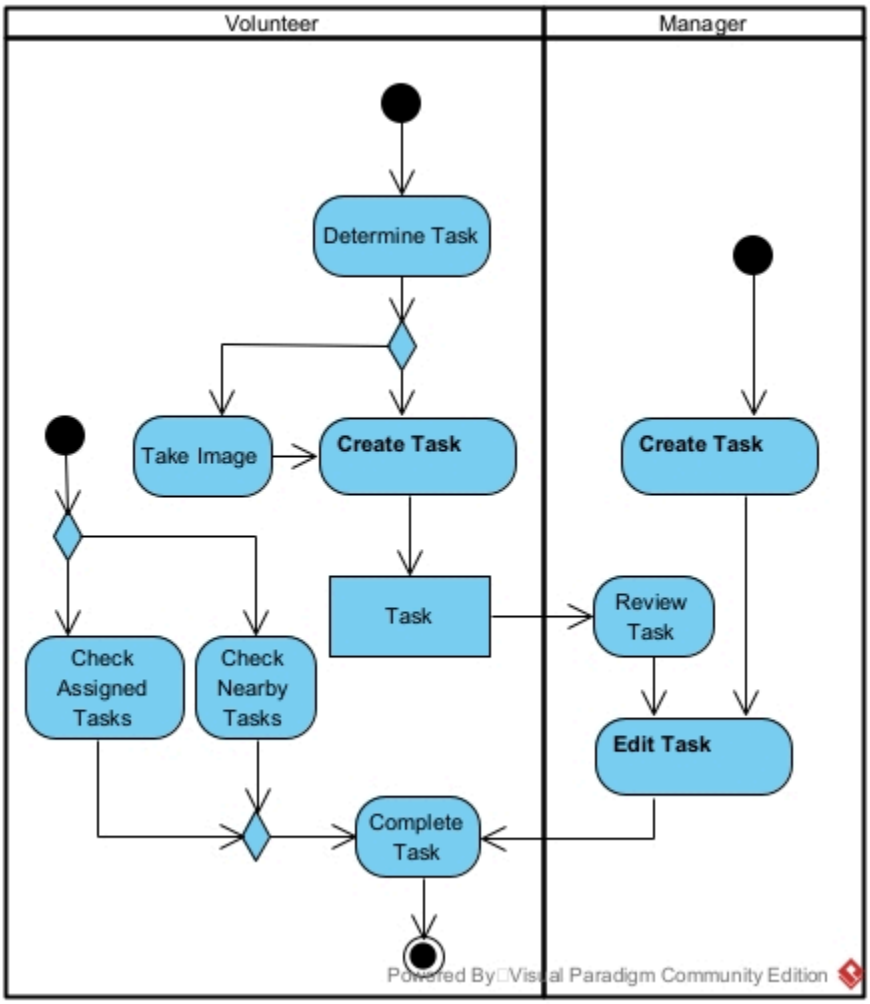
Initial Task Lifecycle Sequence Diagram



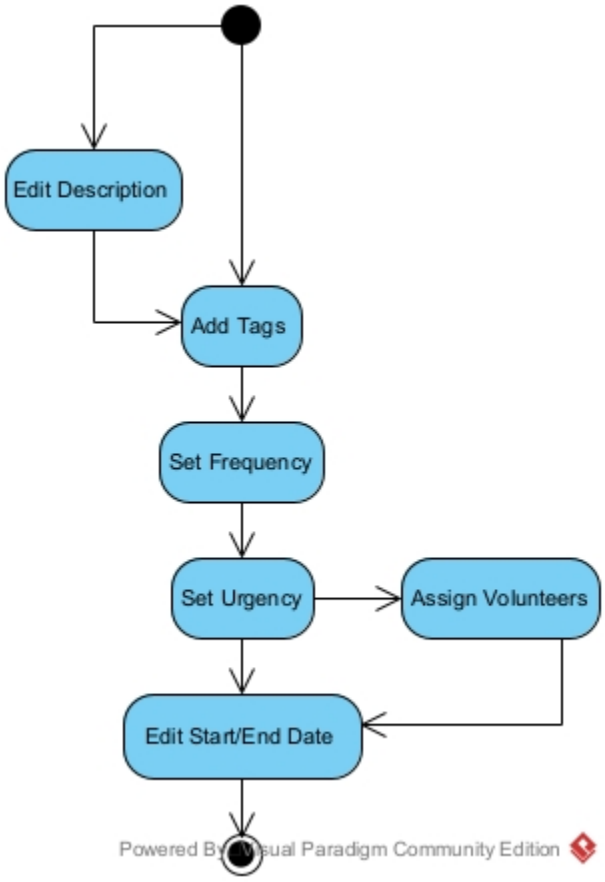
Initial Create Task Activity Diagram



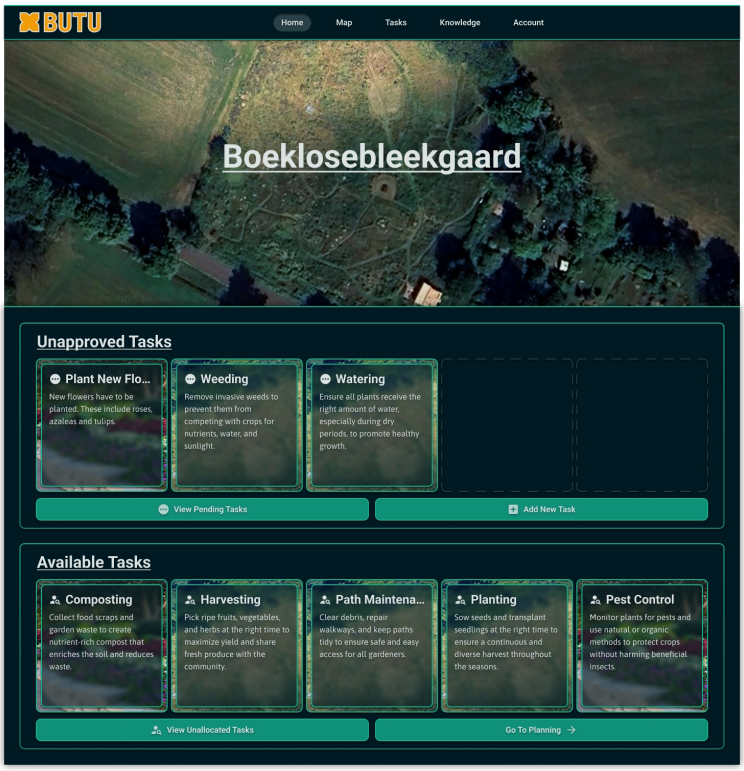
Initial Task Management Activity Diagram



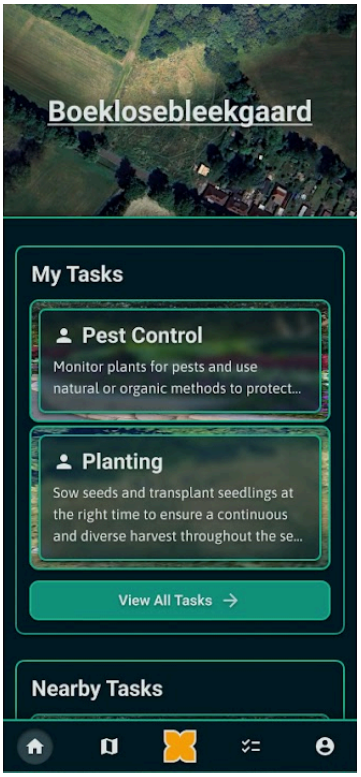
Initial Edit Task Activity Diagram



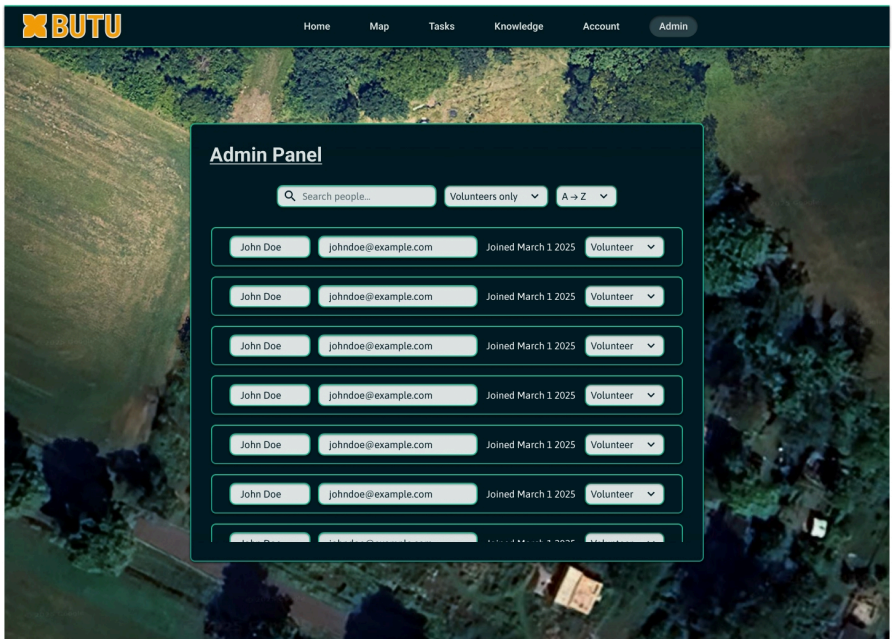
Full Mock-Up



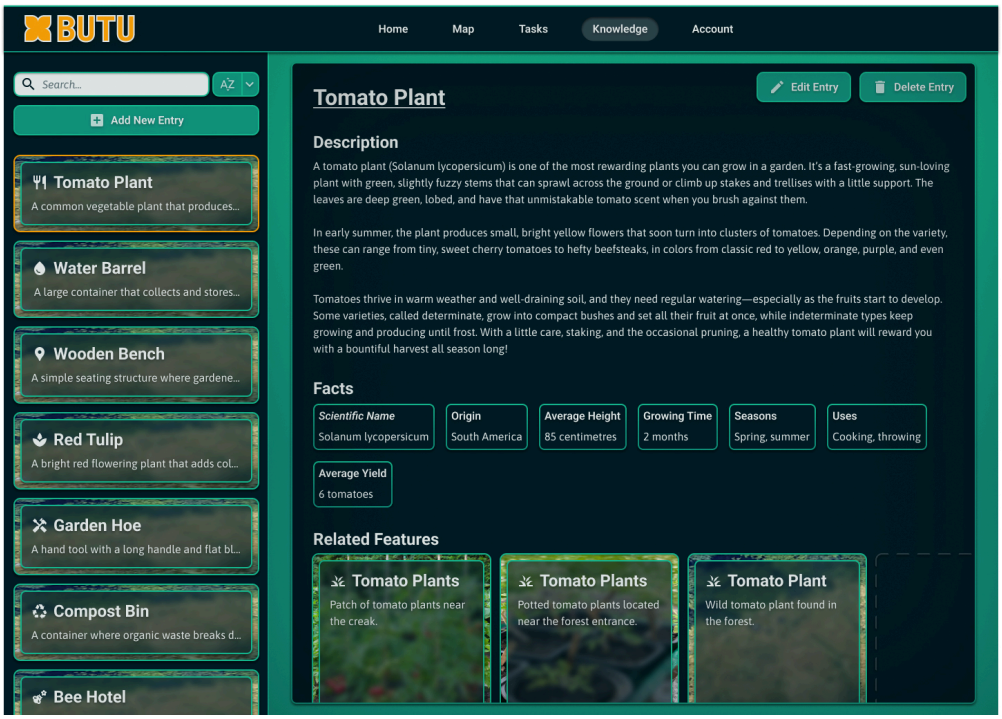
Homepage (desktop)



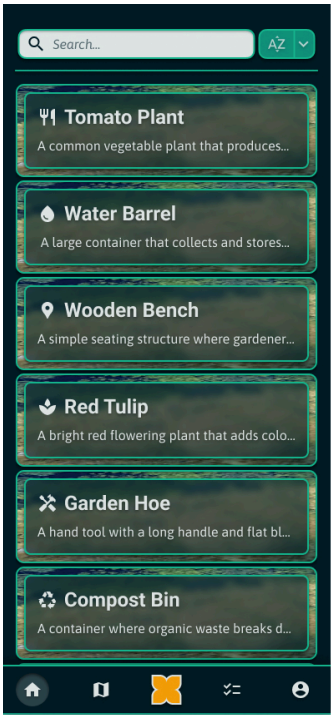
Homepage (mobile)



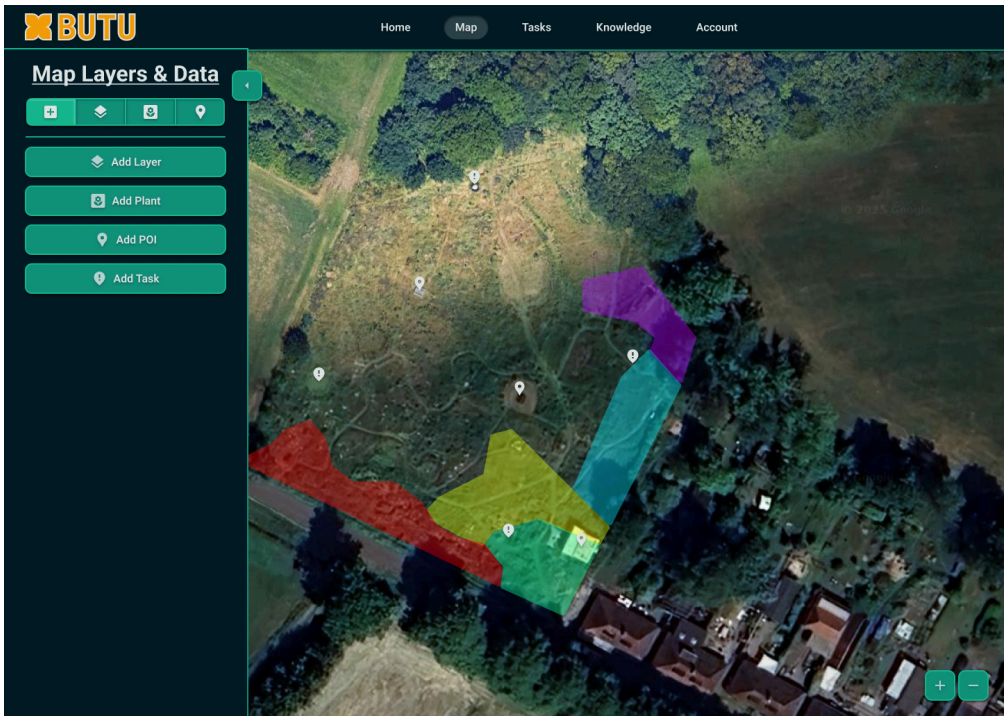
Admin Panel



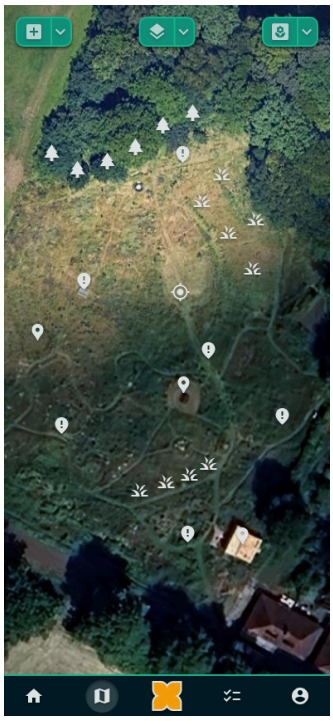
Knowledge Base (desktop)



Knowledge Base (mobile)



Map (desktop)



Map (mobile)

Appendix II: AI Statement

For this project, some use was made of AI tooling. All code is checked and understood by humans, and we take full responsibility. Git history is visible to the supervisor to ensure full transparency.

- GitHub Copilot was enabled while making small, repetitive parts of the backend.

Appendix III: Manual

Login/Registration

To log in, press the “Inloggen” button on the navbar at the top of the screen. This will take you to the login page, where you can enter your email and password to log in. If you do not have an account, press “registreer dan hier”, which will take you to the registration page. Here, you can enter your name, email, and password (twice) to create an account. If you are the first user, you will be an administrator with full access. Otherwise, you will be a pending user that must be verified by an administrator before you can do anything.

Administration Panel

To go to the administration panel, click on the “Admistratiepaneel” button on the navbar. On this page, admins can search, edit, and delete both users and tags. The two buttons at the top of the panel let you switch from editing users to editing tasks. When editing users, you can use the search bar to look for users by name. You can see and modify the names, emails, and roles of users. To verify users, change their roles here. Finally, you can delete users. To save your changes, use the save button below. When editing tags, things work similarly: you can search tags by name and choose to delete them. You can also add new tags by typing them in the bar below and pressing the plus button.

Account page

To go to the account page, click on the “Mijn Account” button on the navbar. This is a very simple page where you can see and modify your name and email if you wish. You must save your changes with the save button. You can also log out here.

Home Page

To go to the home page, click on the “Homepagina” button on the navbar. On this page, you can see an overview of pending and available tasks in addition to recently added locations. Pending tasks are tasks that were added by volunteers and have not yet been approved by an administrator. The pending tasks box also has buttons to go to the task page for more planning or to the map page to add new tasks. Available tasks are tasks that have no assigned users. Tasks can be clicked on in order to view their individual pages.

Tasks Page

To go to the main tasks page, press “Taken” on the navbar. On this screen, you can see an overview of tasks and their current states. First, there are tasks that are unplanned and do not have set times to be completed. Then, you have different displays for tasks that are due this week, next week, next month, and later than next month. Finally, you can see the completed tasks. You can press on tasks to go to their individual pages.

Map page

To go to the map page, press “Kaart” on the navbar. On this page, you can view and add new plants, zones, and paths. Additionally, you can create tasks, either with their own location or for specific objects.

To begin with, on the top right, there is a button that opens a submenu of filters. You can use these filters to toggle elements of the map. The filters, from top to bottom, are tasks, plants, areas, and paths.

On the top left of the map, you have a series of buttons. The topmost one allows you to zoom in or out. Under that, the next button (with 4 arrows) allows you to move objects on the map; their new positions will be saved. The next button has an eraser icon and allows you to delete objects by clicking them. Finally, there is a button for your current location and a button for returning to the food forest.

On the bottom left, there is a plus button that opens a submenu when pressed. In that menu, you can choose from tasks, zones, objects/plants, and paths. By selecting an option, you can create a new object of that type by clicking on the map. To create tasks or plants, click once. To create zones, place some points to make a shape and finish by pressing the

starting point. To make a path, place some points in a line and press the final point to stop. Once you place a new object, you will see a panel on the left where you can edit it.

To edit an object, press on it. This will display a panel on the left where you can edit it in detail. This panel will have a box with a form that lets you enter a name and description. To save any changes, press the “Opslaan” button. You can add more tasks to objects (that are not tasks themselves) by pressing the “Nieuw Taak” button, which will add a new form for a task. Each form on this panel has a drop-down button above it that allows you to collapse it. In addition to the “Opslaan” button, there are number of other button that can appear on a form: the “Details” button allows you to edit a task, the “Kennis” button takes you to a knowledge base entry for the object, and the “Bestanden” button allows you to add images or videos to a task.

Individual Task Page

To manage a task in more detail, you can visit its page by pressing on it in the homepage or the tasks page. This page displays all of the information about a task: its name, tags, description, timing, and attached objects. Under the information, there are four buttons: The top left button allows you to assign and unassign yourself from the task. The top right button allows you to accept a pending task (if you are an admin) or complete a current task. The bottom left button allows you to edit a task. The bottom right button deletes the task. If you choose to edit the task, you can change each element as you wish. You must save your edits with the button at the bottom.

Knowledge Page

To go to the knowledge page, press “Kennis” on the navbar. On the left side of the screen, you can search, add, or select knowledge entries. Once selected, you can view a knowledge entry on the right, including its name, description, facts, and map objects. At the top right of this display, you can either delete the entry or edit it. When editing, you can change any aspect of the entry. You must save with the button in the top right to confirm any changes made. In case no knowledge entries are present, the user will automatically be given the creation screen for knowledge entries. There, they can create a simplified entry, after which they can edit it for more details. There is also an additional create entry button which is intended to be used on desktop to create more knowledge.