UCLouvain

École polytechnique de Louvain

Design and development of a market gardening application

Lauzplan, a user friendly and non intrusive application

Author: **Guillaume DE MOFFARTS**
Supervisors: **Guillaume LOBET, Kim MENS**
Readers: **Céline DEKNOP, Rémi DESMET**
Academic year 2020–2021
Master [120] in Computer Science

Acknowledgement

I thank both my advisors Prof. Kim Mens and Prof. Guillaume Lobet very much for their guidance all over the year and for their trust and confidence in my work, allowing me to work autonomously and make my own proposals.

I am also grateful to Rémi Desmet and Prof. Guillaume Lobet for having introduced me to the market gardening world.

Contents

Chapter 1

Introduction

This master thesis is about the development of an application targeted to market gardeners, supporting them to improve their profitability. It aims at accompanying them to better plan their work and to define the prices of their products.

1.1 Context

Traditional farming fields that extend over great distances where all work is done mechanically with tractors are well known by people. This way of producing is harmful for biodiversity compared to market gardening, which works at a much smaller scale. With market gardening, 2.5 to 10-hectare fields are used, with diverse varieties growing close to each other [1]. Studies have shown that this way of producing leads to much better biodiversity. In addition to the benefit for biodiversity, the pollution is also reduced since most of the work is typically done with manual tools instead of polluting tractors.

At the moment, very few studies have been conducted on the topic of market gardening. Therefore, the *Université catholique de Louvain* (UCLouvain) bought the Lauzelle's farm to conduct research on various aspects of market gardening.

1.2 Problems

Nowadays, most gardeners define the prices of their products based on the ones from organic wholesalers. Most market gardening work is done manually and on a much smaller scale than usual organic farming. It is therefore far more challenging to make a living from this activity. Therefore, the regular prices do not always reflect the amount of work needed to grow a product. If market gardeners want to revise their prices, they need to explain to their clients why they are so different from the supermarket ones.

It is also challenging for market gardeners to do an effective planning to manage their crops. Such an effective planning would allow them to maximize the usage of their land and consequently to generate more profit.

1.3 Motivation

Since market gardening is much better for biodiversity, it is therefore critical to develop this activity. Market gardeners need to be helped to make their activity more profitable, even if many do it by passion and are supported by volunteers. Several software applications already exist that they could use. Unfortunately, none of them are targeted to professional market gardening.

1.4 Objectives

Around 2017, the UCLouvain started to do some market gardening studies, but lacked data. Researchers thought of an application that would help gardeners in their work and, at the same time, give them the data they need. This application was designed and developed by Zélie Mulders for her master thesis [2]. Regrettably, gardeners are not using it.

The first objective of this master thesis is to identify why the present application is not used and how to adapt it to help gardeners. Several questions have to be answered. Are there any missing features that gardeners need? Is the application too difficult to use? Does it take too much time for gardeners to use the application? Answers to these questions will allow deciding whether new features should be added, or existing ones improved. It is possible that the whole application, or only some of its parts, should be redesigned.

The second objective is to develop an application taking into account the answers to the previously mentioned questions. It will either be a completely new one or an evolution of the existing one. Finally, the developed application will be validated to assess whether it better addresses the needs of the market gardeners.

1.5 Approach and Contributions

The first thing to do before even looking at the existing application is to get a general understanding of its domain. It includes the difficulties and needs that both researchers and gardeners have. Several sources have been used for this first analysis. Researchers at UCLouvain wrote a paper precisely describing these difficulties and needs [1]. A report on a research project that wanted to test a model of market gardening was also included [3] in our researches. Based on the collected

background, the existing software solution needs to be assessed to understand the problems preventing gardeners from using it.

The existing software is a web application, which should still be the case with the new one. The second thing to do is to analyze and compare different technological choices for developing web applications. The current software solution will then be assessed based on this comparison.

Finally, based on the understanding of market gardeners' needs and the identification of the problems of the existing application, what should be kept and what should be changed will be identified. The best technologies and the architecture that fit the needs can then be selected, should they remain the same or be new choices.

1.6 Roadmap

This master thesis starts by giving some background material to familiarize the reader with market gardening and web technologies. It then presents the application developed for a previous master thesis and other works related to the presented research. After that, the next chapter states the faced problems and the solution proposed to solve them. Then, the master thesis discusses how the solution has been designed and implemented. Finally, the last chapter presents how the proposed solution has been checked to evaluate whether it actually solves the identified problems and if it has been correctly implemented.

Chapter 2

Sneak preview

Before moving to the core of this master thesis, this chapter shows a sneak preview of the applications that have been developed for this master thesis. It contains several briefly commented screenshots of the application, organised along the main features, to help the reader visualise what has been realised.

2.1 Web application

The first application of the developed solution is a web application. The main features of this latter allow market gardeners to manage their gardens and to define the parcels and beds they contain.

When the application is launched for the first time, the main area displays a warning indicating that no garden have been created in the application yet, as shown on Figure 2.1. Creating a new one is very easy and done by clicking on the *"Crér un jardin"* button. A new garden named *"Nouveau jardin"* is then created and can be renamed later on.

When several gardens have been defined, a menu on the top left of the screen offers the possibility to switch between gardens as shown on Figure 2.2. The currently displayed garden is highlighted with a light blue background. This menu also contains a button to create a new garden, since the previously mentioned one is not accessible anymore.

When the market gardener has selected a garden, he can define the parcels of the garden by directly drawing them on the map that shows in the main area of the application. Figure 2.3 shows a parcel being drawn on the satellite view. Three points of the parcel have already been fixed and the user is currently moving the fourth one. While the parcel is being drawn, two pieces of information are shown: the area of the current parcel and the length of the line being drawn. When a parcel has been drawn, it is possible to edit or delete it by clicking on it and using the toolbox showing on the top of the screen.

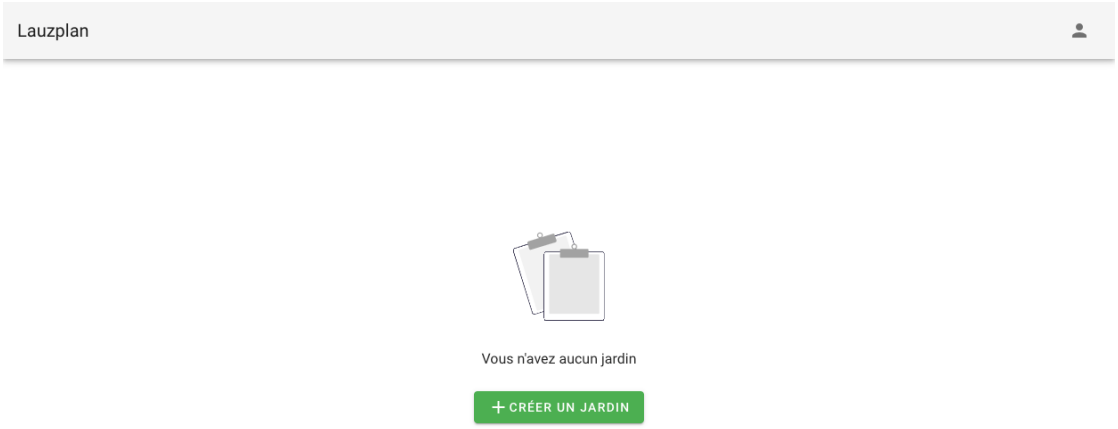

Figure 2.1: When no garden have been created, the application displays a message saying so, with a button to create one.

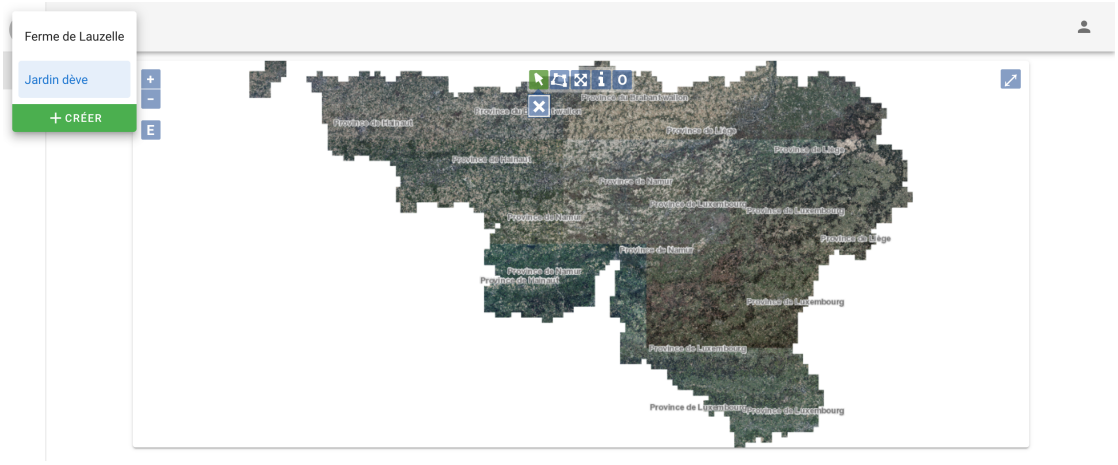

Figure 2.2: A menu on the top left of the application can be used to switch between gardens.

From a drawn parcel, it is possible to define its beds. To do so, the last button of the toolbox has first to be clicked. The application then asks the user to select one of the border of the parcel for which beds have to be defined, as shown on Figure 2.4. Beds are then automatically generated inside the parcel parallel to the selected border. Their sizes are by default predefined in the application (one-meter wide with 40cm between the beds).

It is also possible for the user to define custom sizes for the beds. For that, the sizes have to be entered from the panel shown on the bottom left of the map, before selecting a border and therefore before the automatic beds generation.

When the market gardener has defined several parcels, he or she can navigate through all the parcels and get information from them, as shown on Figure 2.5. At all time information about the parcels are available below the map. Some information are also available directly on the map by clicking the i button in the toolbox and hovering the parcels with the mouse.

2.2 Mobile application

The second application of the developed solution is a mobile application. This latter is to be used by market gardeners for the timekeeping activities. Figure 2.6 shows four screenshots of four different screens of the mobile application.

In the two first screens, the market gardener selects the variety he or she is working on. The third screenshot shows the chronometer that market gardeners use to keep track of the time spent on the selected variety. Finally, once he or she stopped the chronometer, a summary of the recorded start and end times is shown. It is still possible from this last screen to update the times, in case the market gardener forgot to start or stop the timer at the moment he or she started or finished to work.

2.3 Future work

The web application presented in the first section is not finished yet, as the time-keeping visualisation feature has still to be developed. Figure 2.7 shows a wire-frame illustrating what kind of visualisation will be available on the web application after future developments.

The first graph is a histogram showing the time spent by variety measured as hours per squared meters. The second graph is a line plot of time spent monthly over the year for the selected species or varieties, again measured in hours per squared meters. Finally, the last visual element is a tabular view of all the collected statistics. From this table, market gardeners can add new measures or edit existing ones, a feature useful for gardeners who do not want to use the mobile application.

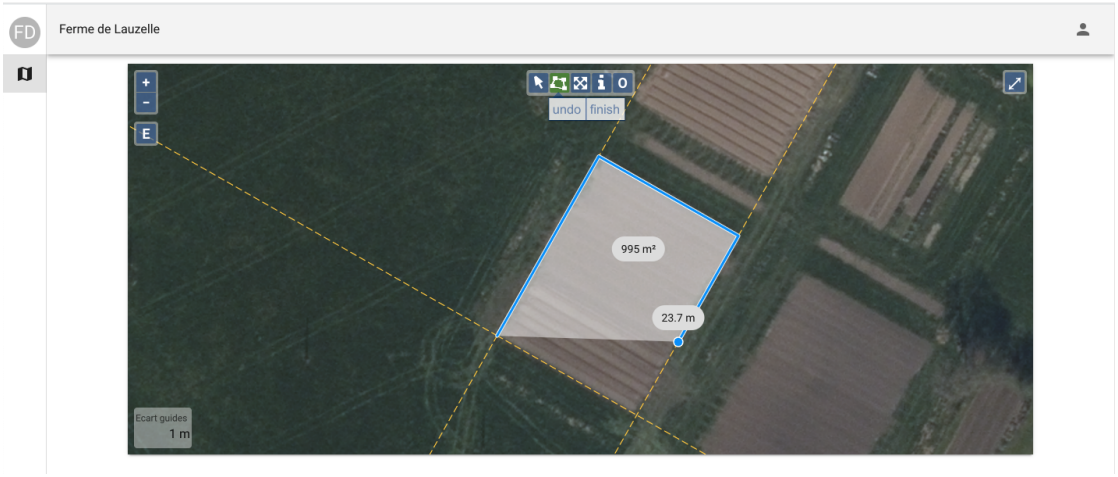

Figure 2.3: A parcel is directly drawn on map with a satellite view, by defining its border with a polygon.

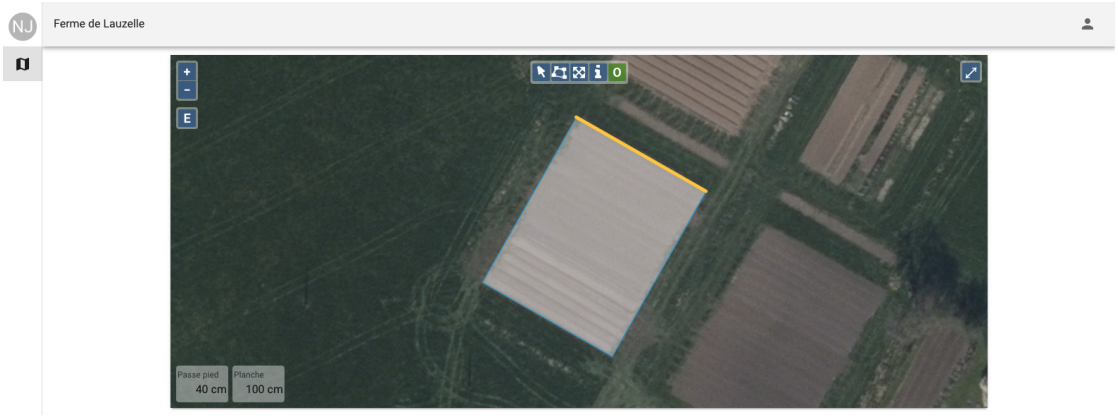

Figure 2.4: To create the beds for a parcel, a distance has to be chosen and a border of the parcel has to be selected.

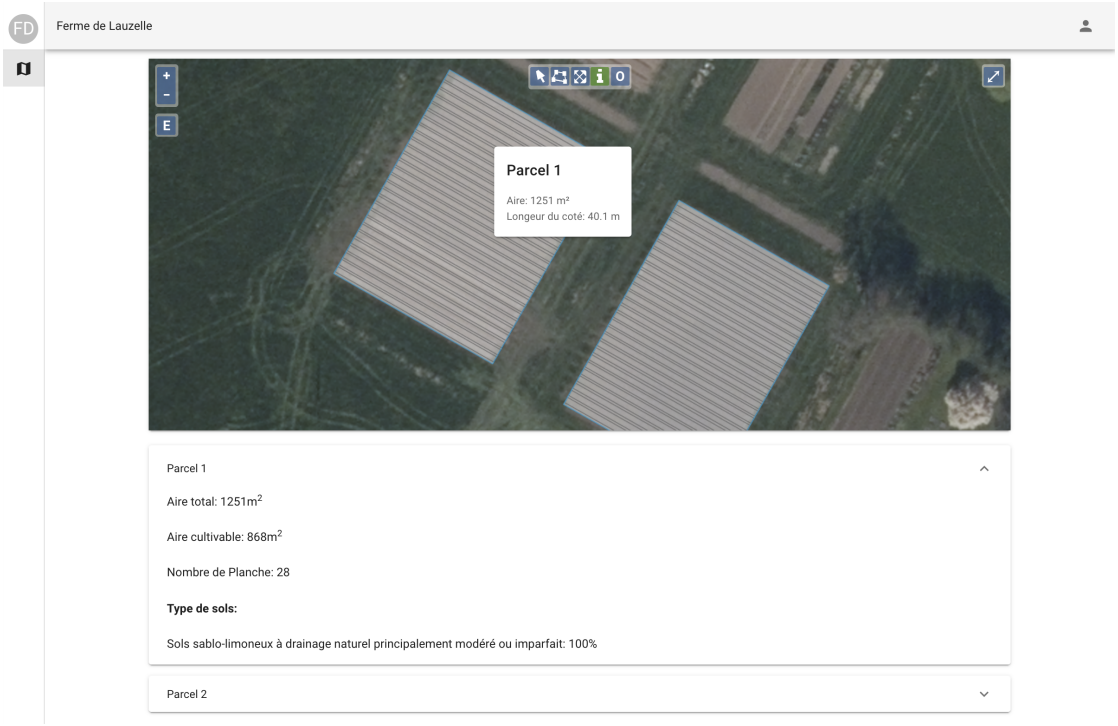

Figure 2.5: A floating box and several panels at the bottom of the page display pieces of information about the parcels.

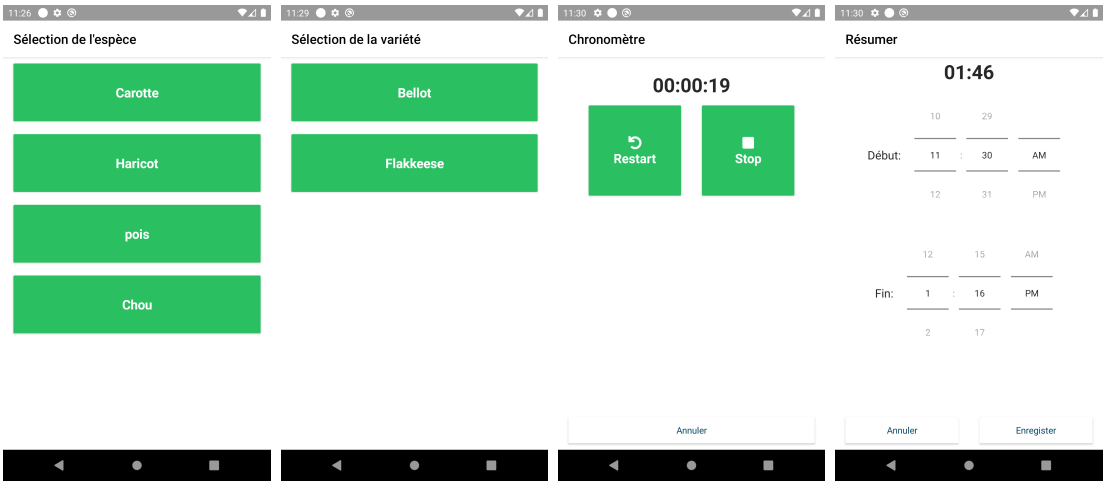

Figure 2.6: The time-keeping feature on the mobile application has four main steps.

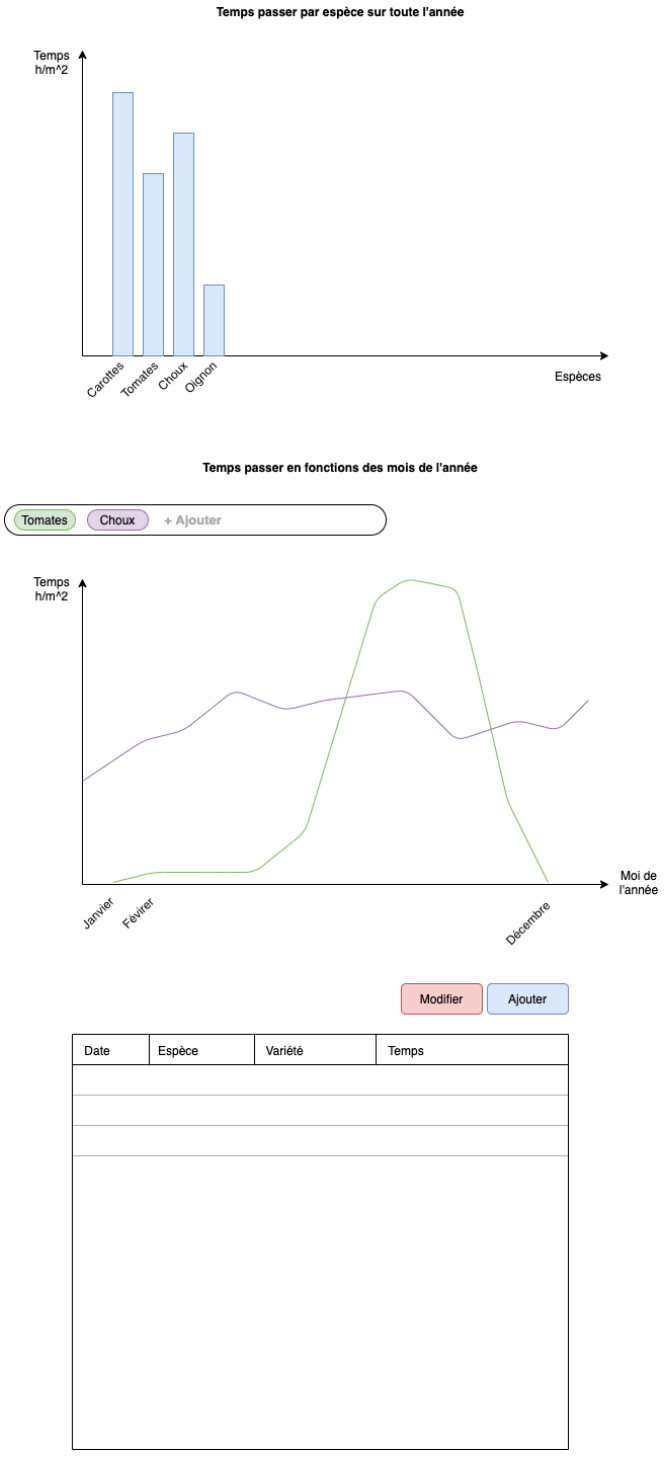

Figure 2.7: Several graphs will be available on the web application to display statistics related to the time-keeping activities.

Chapter 3

Background material

Before going deeper into this master thesis, this chapter introduces some concepts for the unfamiliar reader. It starts with an introduction to the basics of market gardening. It then introduces technological concepts and the libraries and frameworks discussed in the document.

3.1 Market gardening

Market gardening is the production of fruits and vegetables on a professional level. In this master thesis, a garden refers to the whole land that a gardener uses to produce fruits and vegetables and is structured as follows. A garden is divided into parcels which are themselves divided into beds. The structure of Lauzelle's farm is shown in Figure 3.1.

Vegetables and fruits can be divided into families called species, themselves divided into different varieties. For example, tomato is a species. Cherry and beefsteak tomatoes are varieties. During the year, gardeners do some cultivation operations on the vegetables, like seeding or treating the plant.

3.2 Software technologies

This section describes all the main technological concepts that are used throughout this master thesis to explain the existing and developed applications.

3.2.1 Types of applications

Different types of applications are referred to in this master thesis. The main type is the web application or web app for short. It is a kind of application that usually

Figure 3.1: Lauzelle's farm is structured as one garden (in blue) divided in several parcels (one shown in red) consisting of several beds (a few shown in green).

runs inside a web browser. One can see it as an "advanced" website. The websites of YouTube, Netflix, Facebook, and Twitter are examples of web applications.

Web applications are accessed from any web browser, just like a website. Therefore, they can run on any computer with any operating system as long as it has access to the internet. Progressive Web Applications (PWA) are a category of web applications that can be accessed offline. They run in a separate browser window to feel more like a standalone application. Finally, native applications directly run on a computer or a smartphone. They do not rely on a web browser, but need to be first installed on the computer.

3.2.2 Web technologies

Most websites and web applications are developed using two main languages. The first one is HTML, a declarative language, meaning that there are no instructions as in Python, C or Java. Instead, it is used to describe to the computer what you

want, and it will find a way to do it. HTML is used to describe to the browser what content needs to be displayed on the screen. The browser interprets this description and translates it to a Document Object Model (DOM), which is easier for the computer to work with. This DOM is a set of elements (DOMElement) such as images, texts, etc.

The second language that is used is called JavaScript. It is an imperative language, telling the computer what to do with instructions. JavaScript is used to access and manipulate the DOM. It offers the ability to add new elements to the DOM, and to modify or remove existing ones. It also makes it possible to listen to events happening on DOM elements. For example, new content can be added to a page after a button has been pressed.

To ease web development, libraries and frameworks are being developed. They are used to avoid the many DOM manipulations required to create a web application. Several libraries and frameworks are compared later in this master thesis.

3.2.3 API

An API (Application programming interface) is a way for software and services to communicate together given a set of rules. In the web applications domain, an API usually refers to a server that provides data in a specific way.

The most common type of APIs for the web are REST APIs. With this kind of API, the server does not store any state, making it possible for the server to quickly scale up if the number of users increases. A REST API is composed of multiple endpoints that can be contacted to retrieve or update data. Each of these endpoints gives access to a specific set of data. There is usually a documentation that describes which endpoint provides access to what data. Figure 3.2 shows two calls to a REST API. The first one retrieves a list of gardens, and the second one asks for more information about the first retrieved garden.

Another emerging type of API is GraphQL. GraphQL APIs are also stateless, but have only one endpoint. This latter is used to query the data, both to access and update it. They are defined with a set of interconnecting types that form a graph. These types also serve as the documentation for the API. Figure 3.3 shows a query to a GraphQL API, getting all the desired data in a single API call.

Thanks to this chapter, the basics of market gardening are now clear. The structure of a garden has been presented, along with how vegetables are divided into groups. Also, the basics of web development have been exposed, along with how multiple services can communicate with each other. The next chapter shows how the current application developed in Zélie Mulders's master thesis works. It also compares existing solutions. Finally, it discusses a paper summarizing the problems that market gardening faces.

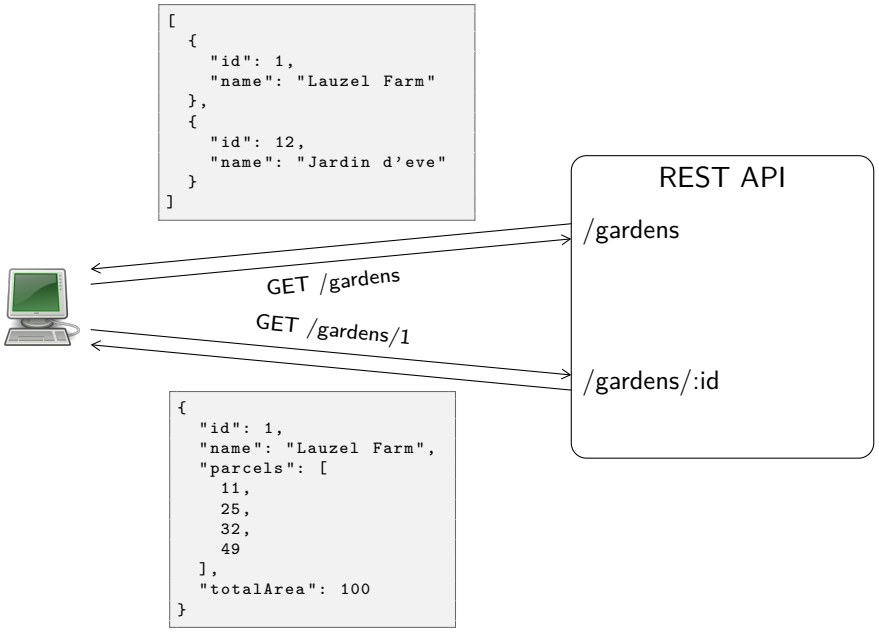

Figure 3.2: To use a REST API, a specific endpoint has to be contacted, depending on the operation to be performed.

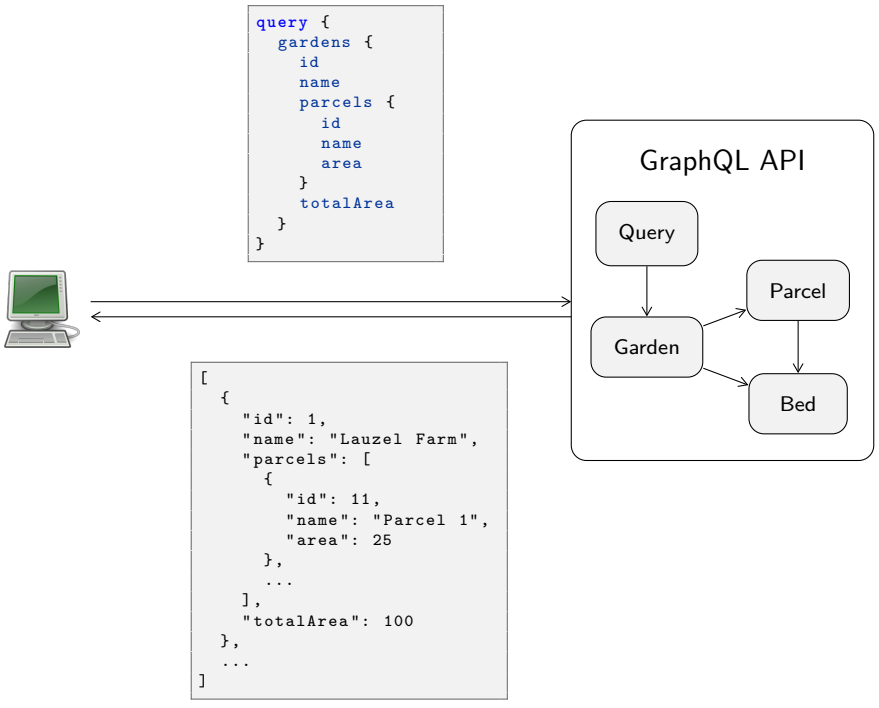

Figure 3.3: To use a GraphQL API, a unique endpoint has to be contacted, specifying which parts of the types graph to retrieve (grey rounded boxes).

Chapter 4

Related work

This chapter first looks at a paper summarizing the problems that market gardening faces. Then, it shows how the current Lauzplan application works. Finally, it looks at other existing software solutions that could be used to solve the problems.

4.1 Difficulties of market gardening

This section discusses the paper *"Les petits producteurs maraîchers diversifiés"* [1], looking at the parts explaining how market gardeners are selling their products.

There are three ways in which market gardeners can sell their products. The first one is direct sales to the consumer. The sale is either made through a shop at the farm, a market, or consumers can come and harvest what they want themselves. Another way is to sell the production to the consumer through an intermediary, for example restaurants or other shops. The last way is to sell the vegetables to wholesalers, adding yet another intermediary to the consumer.

Market gardeners having their own shop might want to sell vegetables they do not harvest themselves, so that they can offer a broader selection to their customers. There are multiple reasons why gardeners would like to buy additional vegetables. First, they might not have enough resources to grow everything by themselves. Second, it is sometimes just too expensive to do it. For example, carrots are not profitable when grown at the small scale used with market gardening. Market gardeners' primary way to access those other vegetables is to buy them from organic wholesalers. As one can imagine, this will significantly influence the price at which gardeners sell their production. Indeed, if they bought half of the tomatoes they are selling, they will sell all of them at the purchase prices plus a margin. This means that the sale prices are aligned with the ones of traditional organic farming, even if it requires more work per vegetable. These prices make it very difficult for market gardeners to make a profit out of their activity.

4.2 Lauzplan

This section presents the main features of the Lauzplan application that has been developed in a previous master thesis [2].

The first step is to create a garden and then to define all its parcels and beds with the form shown in Figure 4.1. A summary view of all the parcels and beds can be obtained from a page shown in Figure 4.2. This is a simplified example, Lauzelle's farm having seven parcels with about ten beds each.

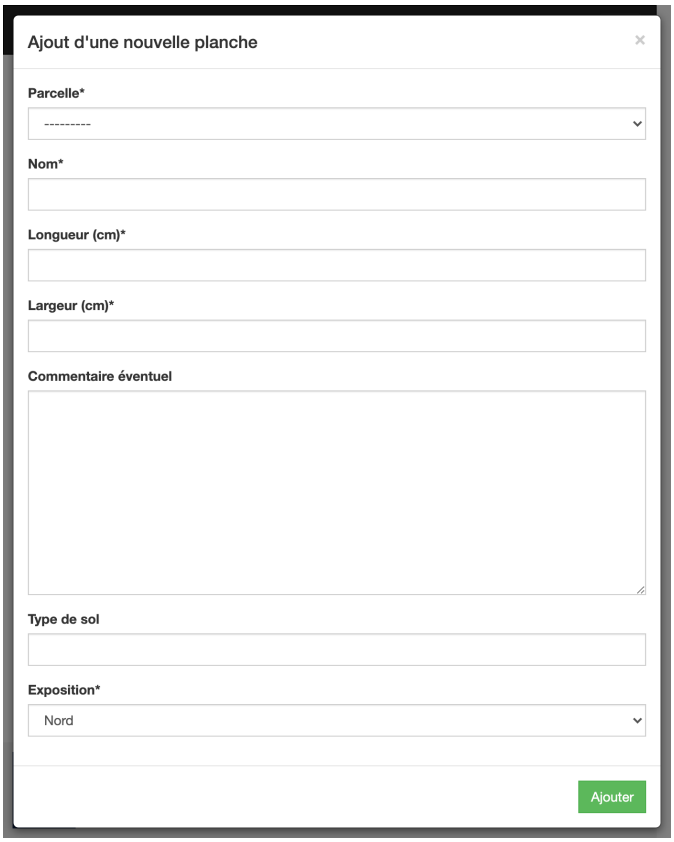

Figure 4.1: The form to add a new bed asks for several information about the bed such as its name and dimensions.

Once the garden structure is created in the application, vegetables with the operations to be done at a specific date and the corresponding durations can be encoded. There is also the possibility to import vegetables from a library, to avoid gardeners to encode everything by hand. Figure 4.3 shows a page summarizing all the created and imported vegetables. They can then be edited from this page and new ones can also be created. Finally, another page is needed to add the created vegetables to beds, which finishes the configuration.

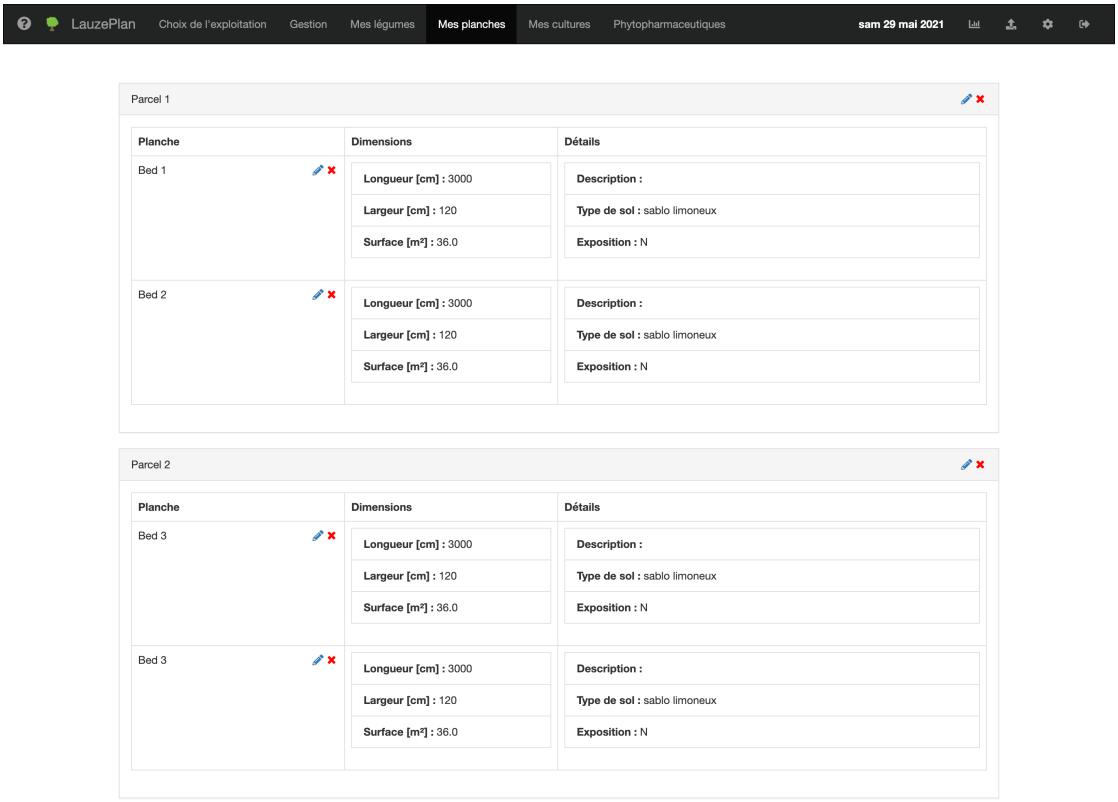

Figure 4.2: A summary view shows all the parcels of the current garden and the beds that they contain.

Creating all the beds and vegetables of a given parcel from a garden can be very time-consuming. The problems of this solution are discussed in more detail in the next chapter.

4.3 Other solutions

In her master thesis, Zélie Mulders already analyzed several solutions [2]. Unfortunately, none of them are dedicated to market gardening. There is an application called *"Mes petit légumes"* only dedicated to amateur gardeners [4]. There is also LEA that is focused on the management of big farms [5].

Finally, there is another software solution called Tend that fits the needs of this work but is not in French and is too expensive [6]. Having this application for free would have been nice, to help even the smallest market gardeners. However, data of the gardeners should be accessible to researchers so that they can conduct studies, which is not a feature of Tend.

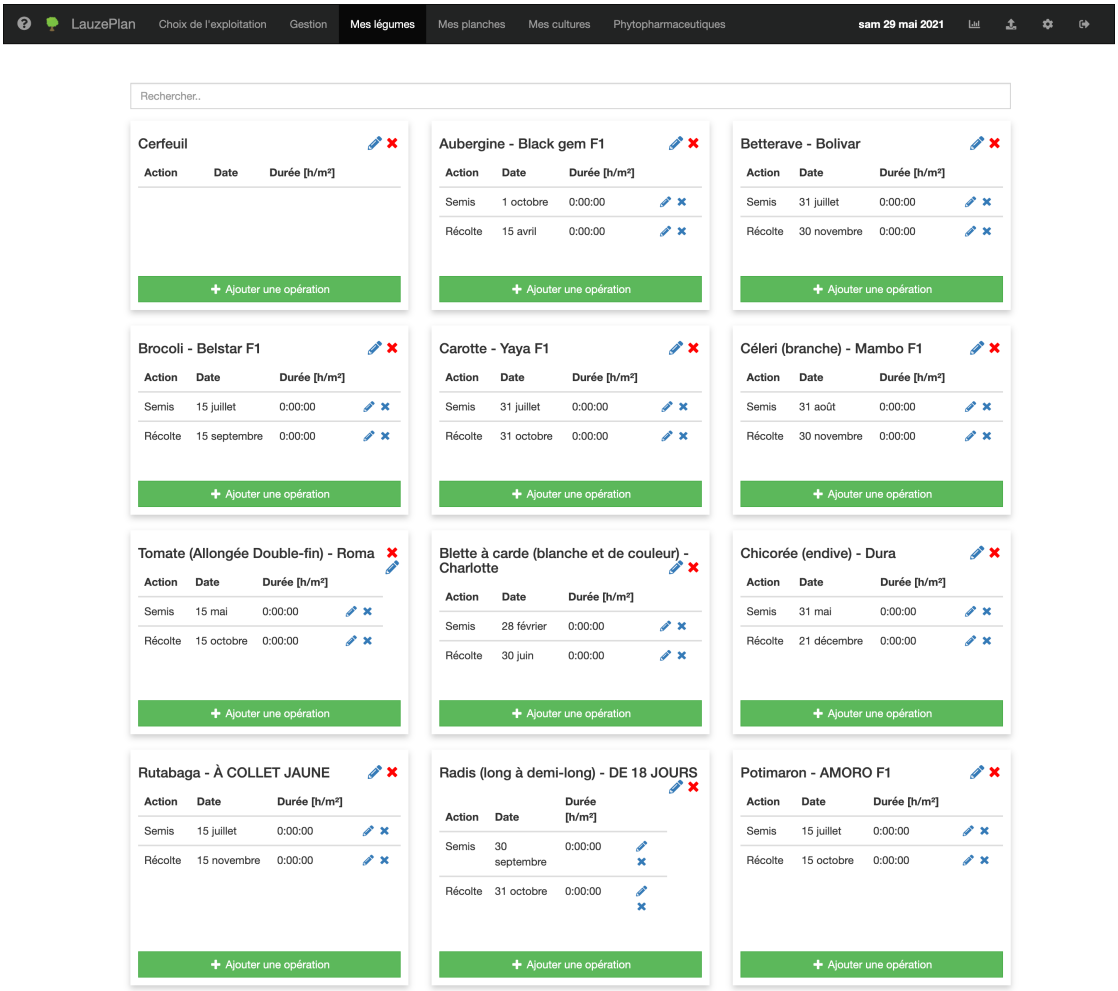

Figure 4.3: Vegetables can be created and then modified from a single page in the application.

New applications are being developed since the previous master thesis. For example, there is a free and open-source software called Qrop that can be used to help with the planning [7]. However, it is still in a very early stage of development and is not recommended for use yet.

Chapter 5

Requirements

This chapter is about the requirements' analysis for the application to be developed. It first presents the functional requirements and then the non-functional ones.

5.1 Problem statement

This master thesis is a follow-up on the work done by Zélie Mulders. Unfortunately, gardeners are not using the developed application. Discussions with the staff at Lauzelle's farm and a short survey highlighted that the features gardeners needed were already implemented in the application. Therefore, this is not the cause of our problem. However, several potential issues with some features might explain why the current application is not used.

First, providing all the necessary information to create a garden in the application is too time-consuming for gardeners. For example, they have to create every bed for each parcel by giving them a name and a size, one by one, even if they usually all have the same dimensions.

Also, the planning generated thanks to the vegetable information was very hard to follow. The application produces a list of tasks to be performed. Every day, gardeners check the list of tasks, but they could not always perform all of them for various reasons. For example, some tasks require specific weather conditions or need a couple of days after rain.

The conclusion of the discussions and the survey are that some of the identified problems are caused by a significant lack of dynamism in the application. The technology choice that was made can explain this. The framework used to build the existing application, namely Django [8], is better to make applications displaying content and less used for those needing a lot of user input.

5.2 Functional requirements

To quickly deliver an application without implementing a version for each operating system, a web application would be the best choice. Therefore, most of the features will be implemented in a web app. However, one of the identified features will be implemented in a separate smartphone application.

5.2.1 Garden creation

To easily and quickly create the structure of a garden, gardener should be able to draw the parcels directly on a map with a satellite view. Also, beds should be automatically generated to fill in each parcel along a selected direction. With the geographical position of the parcels, the soil type should be automatically identified and displayed. The sun orientation should also be automatically retrieved for the drawn parcels.

This way of designing the garden will be much quicker and simpler than before. Most information that needed to be entered by hand will now be automatically input. It also gives a good representation of the garden on the land that the gardener knows very well.

5.2.2 Planning

Planning with fixed dates is much more complex to solve. The day a task will be executed indeed depends on too many factors. That is why a completely different approach is proposed, in which gardeners are not given tasks anymore. Gardeners know their jobs very well and do not want an application to tell them what and when to do.

Instead, the application will help them maximize their land usage by maximizing the amount of rotation on a bed. Only the information provided by seed sellers will be used, which are two or three date ranges. The first one, which is optional, is the warm seedling dates range. The second range is the planting and the last one is the harvesting dates range. The estimated amount of time a plant needs to grow before harvesting can be deduced from those ranges. Gardeners will use that information combined with a Gantt chart to help optimize their land usage.

5.2.3 Time-keeping

The last requirement for the application is a time-keeping system. Gardeners would like to know how much time they spend on each variety to help them estimating the value of each product.

To do so, a mobile application with the possibility to quickly start a chronometer for a specific operation on a variety is required. This application needs to be accessible on a smartphone with a bad internet connection or even offline.

On the web app, we want to have a visualization of the total time spent for each variety. It should also be possible for gardeners to manually add new entries, a feature that will be useful for those that do not have a smartphone.

5.2.4 Researchers access

The researchers' interface of the current application is not very useful. Indeed, researchers would only access this page to extract the data they need and then analyze them with another tool. Therefore, this interface can safely be removed.

However, an API providing a direct access to the data for the researchers will be implemented. With this, they will have the possibility to directly access the data from a Python or a R script, for example.

5.3 Non-functional requirements

Both the functional and non-functional requirements are essential for the developed application. Indeed, all gardeners are not very comfortable with web technologies and smartphones. This is why the application must be as user-friendly as possible.

Gardeners should also spend the least amount of time possible on the application. It should be a tool to help them, and it must not be intrusive.

5.4 What to keep from previous work

Now that the requirements for the application to be developed are clear, elements that can be kept from the existing application can be identified. As already discussed, the Django framework is not the best choice for dynamic web applications. There exist ways to bring dynamism, but they do not seem to produce habitable code and may confuse future developers. That is why only the Django data models will be kept from the previous application. In the new solution, it will only be used as the back-end, and a more modern framework will be used for the front-end. The new architecture is explained in the next chapter.

Chapter 6

Technology analysis

Now that the requirements are defined, the different technologies that may be used to develop the application have to be identified. This chapter compares several possible technologies and the motivated choices are presented in the next chapter.

As a reminder, a web and a mobile application have to be developed as well as an API for researchers. In addition to that, each client needs a way to communicate so that cultivation operations made on the mobile app are reflected on the web application, and researchers also have access to this data.

6.1 Web clients

Three main types of web clients can be considered. The difference between them is the way they handle the rendering of the user interface on the browser.

6.1.1 Static pages

In this kind of client, the web browser makes a request for specific HTML files to a server. These files are already present on the server which just serves them to the browser which then renders the HTML. Figure 6.1 illustrates the communication between a client and a server. Working with static pages does not allow much user interaction since the HTML cannot be changed.

There are two main ways to have the HTML files on the server. The simplest one is to code the files by hand, requiring no specific tools besides a file editor. Unfortunately, the files must be updated for each change to be done. To solve this problem, another way is to have the static HTML files automatically generated by a generation library or framework. This latter has just to be told how to generate each page based on given data. The advantage is that the website is developed once, and the static files are rebuilt when the given data are updated.

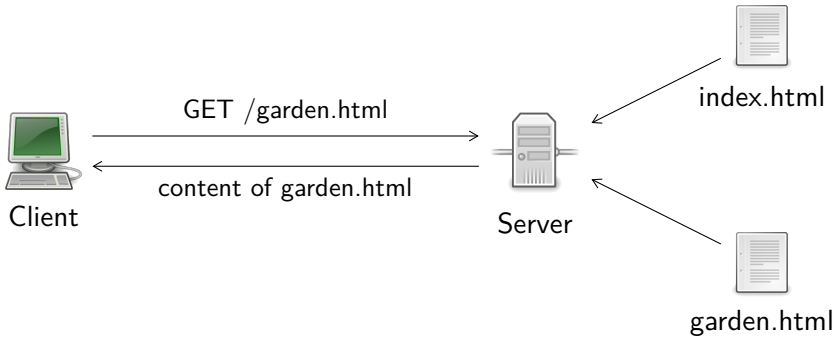

Figure 6.1: With static pages, the client requests pages to the server which already has them ready to be sent back to the client.

6.1.2 Dynamic pages

Static pages do not offer any customization for the user and dynamic pages solve this issue. The client is mostly the same as with static pages. It requests a page from the server, but can enrich the request with additional information. The big difference is on the server side.

As shown on Figure 6.2, there are no HTML files on the server anymore. The HTML content is generated on-the-fly by the server using the requested page and the additional information from the request. This makes it possible to have content that is specific to each user, depending on the additional information sent with the request. Compared to static pages, doing the generation on-the-fly has a cost on the server side, even if the server is usually powerful enough to generate the HTML quickly.

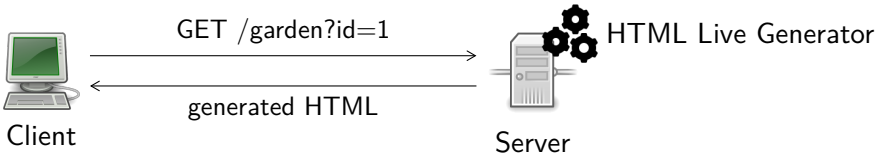

Figure 6.2: With dynamic page generation, the server has the ability to produce different content specific to the user, for the same request.

6.1.3 Client-side rendering

Dynamic page generation is better for user customization. However, it is still not ideal because it lacks dynamic user interactions as HTML content still needs to be loaded from the server, for each action from the user.

To solve this, JavaScript is combined with the static or the dynamic pages solution. This JavaScript code will then be able to modify the rendering on the browser. For example, it can fetch data in the background and render elements based on these data on the browser.

With this method, the server is not the only responsible for the rendering anymore, making it lighter. Unfortunately, it makes the client heavier, slowing down the client's computer, if there is much rendering to do.

6.2 Mobile application

In the early days of mobile applications, there was only one way to develop them. A separate application had to be developed for each mobile operating system using a native framework and API. The most significant advantage is that the application is more efficient, responsive and lightweight since it directly uses the native tools of the phone. Nowadays, two alternatives exist to make it possible to develop a single application that can run on different operating systems.

6.2.1 WebView

The first and simplest alternative to native development is WebView. It makes it possible to develop an application only using web technologies. It will then be a kind of "mini-browser" rendering HTML. Libraries can be used to access native functionalities of the phone, working as a relay to the native API calls.

Using WebView is great because the same technologies can be used to build a website and a mobile application. Unfortunately, this comes with some downsides. Since the app is a mini-browser, it can be heavier than if it had been built it using the native framework. Also, new native API features can only be used when the developers of the libraries have integrated them.

6.2.2 Native compiler

The second alternative is in-between native development and WebView. The application is still developed using web technologies such as JavaScript, but HTML is not used for rendering. Instead, a framework will do native rendering. This means that it will use native UI elements such as layouts, buttons, and text areas, for example.

Therefore, the mini-browser is not needed anymore for the rendering of the user interface. This makes the application lighter and more responsive. Moreover, the application looks and feels closer to a native application since it uses native UI components.

Chapter 7

Architecture

The new application has a different architecture compared to the existing one, going from a three-tier to a four-tier. It indeed consists of a database, a server, an identity provider, and clients.

The significant change is in the responsibilities that the clients and servers have. In the current application, the Django server is responsible for the vast majority of the work. It includes accessing the data, processing it, and generating HTML from it. The client only has to display what is provided by the server. Moreover, the current Django server is also responsible for identifying the users.

Several responsibilities have been removed from the server in the new application. They have been moved to the client or to another server. First, the server will not do any rendering anymore. Instead, it will pass the processed data to the clients thanks to an API, letting them do all the rendering. A huge advantage of this way of working is that we could re-implement any of the tiers to use new technology.

Figure 7.1 shows the different parts of the new architecture and the choices of technologies. The remainder of the chapter describes in more detail the different parts of this architecture.

7.1 Client-server communication

As discussed above, the server has to pass data to the client and vice-versa. In order to do that, an API has to be defined. Nowadays, RESTful and GraphQL APIs are the most common options for the web. A GraphQL API has been chosen because there are multiple clients that need to access the API. Each of them has very different needs in terms of the shapes of data.

On the one hand, a web application dedicated to garden creation with the planning part will mainly be accessed from a computer and requires more data. On the other hand, we have a mobile application for time tracking, which will be

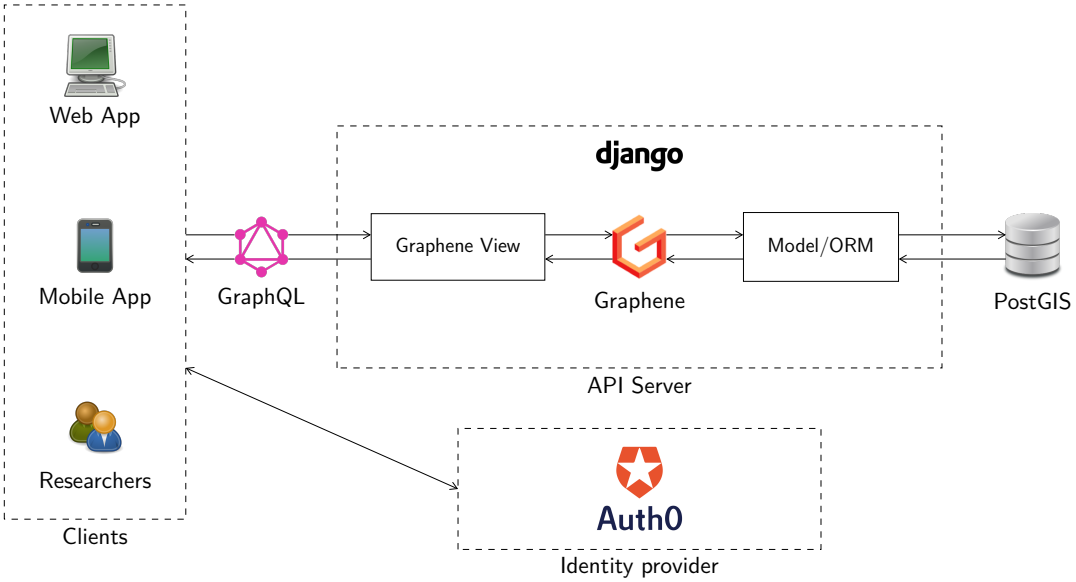

Figure 7.1: The global architecture of the proposed software consists of four tiers communicating between them.

accessed on phones with a poor internet connection, requiring that as little data as possible is exchanged so that the application still loads quickly.

The last client is for researchers' access. They will call the API directly without going through a graphical interface. The shape of the data they will need is unknown as it will depend on the research they are conducting.

To sum up, there are three clients that need to access overlapping data sets but with different shapes. For example, researchers may need to list all the beds with their respective soil types. The gardeners' web interface may need a list of parcels including their beds with their respective soil types. Finally, the time-tracking web interface only needs the name of each bed and parcel.

Satisfying those constraints could have been done with a RESTful API, using parameters and filters on the request, but it would have required more work and would have quickly become complex. The GraphQL API is a huge advantage for researcher access as the API will not have to be modified every time a researcher needs a different type of data.

7.2 Server

To propose a GraphQL API, the technology chosen to develop the server must support GraphQL server development. As explained in Section 5.4, part of the data models previously created by Zélie Mulders should be kept, with some minor

modifications, in order to save development time. From a technical point of view, it means that a library making it possible to create a GraphQL server while integrating well with Django is required.

The choice that has been made for the server is to use Graphene with its built-in integration with Django [9]. This library can be used to create the API and to map Django models to Graphene types. With this solution, it is possible to very quickly create the GraphQL API with a very few lines of code.

7.3 Web client

As of today, there are two widely used frameworks/libraries to build web clients, namely React [10] and Vue.js [11]. Both technologies could probably be used to build the application. However, Vue is a framework, and React is a library. It means that Vue offers a more complete solution than React. The latter would require multiple third-party libraries to achieve the same result. Another aspect that should be considered is the previous experience of the author with Vue. This means that code will be produced faster, reducing the number of bugs that a beginner could make, and producing cleaner and more habitable code. Furthermore, Vue is easy to learn and extremely well documented, allowing future developers and students to understand the code quickly.

To make the code even more straightforward and produce the application faster, the Nuxt.js [12] zero-configuration framework built on top of Vue has been chosen. With this framework, the application pages can be built just like it would have been done with Vue, but easing all the configurations. For example, page routing is done automatically based on the directory structure, instead of doing it by hand with code. A direct consequence is that every project built with Nuxt follows the same structure. New developers directly know where to find most of the code and can start to work faster on the project.

7.3.1 Server communication and local state

Making web requests with a query is the way used to communicate with the server and retrieve data. Of course, this can be done just by using the request API of the browser. However, we chose to use Apollo client [13], an excellent library built to access the application data with GraphQL queries. Application data refers to any data that the application needs to work. It could be multiple APIs, data stored on the browser (local storage), or temporary data in memory (local state). Having a library that aggregates and processes data from different sources is very useful to avoid code duplication. Indeed, the data processing will appear only once in the code instead of each time data is requested.

Another great feature is that Apollo can store the results of queries in a cache memory. When a query is made, Apollo immediately returns the result present in the cache matching the query, if any. At the same time, it makes the actual request to the different APIs and then updates the cache with the retrieved results, matching the actual data from the server. The result first returned by Apollo is kept in sync with the cache. This means that it will be automatically updated with any result of API requests related to the data, not only the one made for this query. Figure 7.2 depicts this behavior. This has two huge advantages. First, data can be immediately displayed to the user when a query is made, even if the web request is not finished yet, which gives us a more responsive and dynamic application. Second, coherent data is available throughout the whole application.

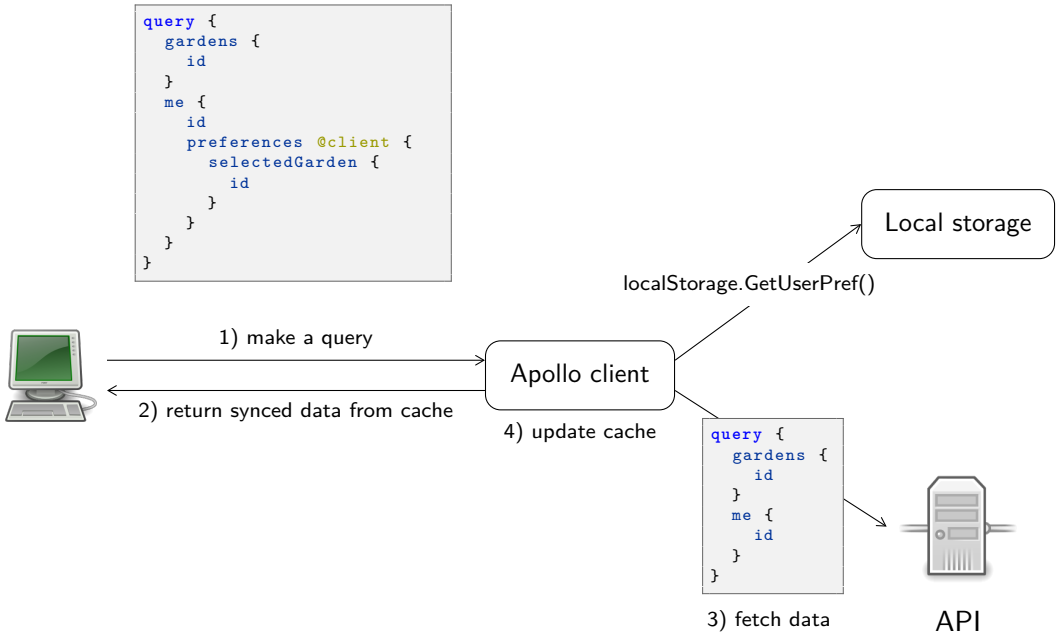

Figure 7.2: Data fetching with Apollo client

7.3.2 Map

One of the functional requirements is that parcels should be drawn directly on a map. To do so, a map provider with a satellite imagery set and a JavaScript library to render the map in the browser are both required. Some map providers also have their library to display their maps.

Three libraries including 2D drawing features have been found. The first one is MapBox [14] which gives excellent results, but can only be used with their paid map provider. The two others are open-source and not linked to any map provider,

namely Leaflet [15] and OpenLayers [16]. They both have similar features, but OpenLayers has been chosen since it resulted in a more fluid rendering. OpenLayers is also more mature and has excellent documentation but has a smaller community than Leaflet. Finally, it also requires less third-party libraries than Leaflet since many features are directly included with OpenLayers.

For the map providers, there are plenty to choose from. MapBox, Google Maps [17], and Bing Maps [18] have been investigated. They all are not free but include labels on the maps, such as road names and places. There also exists free map providers, including the ESRI [19] world imagery and the imagery set of WallOnMap [20]. With both of them, a layer with labels has to be added on top of their imagery.

The two factors that came into account to choose the map provider are the price and the fact that the imagery is up-to-date, so that parcels are already visible, and the gardener can draw over them. Google maps and the imagery set of WallOnMap are both up-to-date, but only WallOnMap is free. That is why the latter has been chosen. The downside is that it only includes Wallonia and not the whole world as Google Maps. Thankfully, since it is easy to change the map provider, it can be switched to Google Maps in the future.

7.4 Identity provider

In the clients, some actions are specific to a user. For example, users are only allowed to modify their garden, not the ones of other gardeners. Researchers are only allowed to see some data and cannot change anything. For that, the API needs to be able to identify the user. We chose to use a service called Auth0 [21] to do so. This service allows us to add authentication securely to our applications effortlessly. Indeed, it can be secure to have a company specialized in the domain to store your users' information and password than doing it yourself. This service is free up to 7000 active users per month, which is enough since the application is targeted to gardeners in Wallonia. If the number increase in the future, there are open-source equivalents such as authlib [22].

Chapter 8

Implementation choices

This chapter describes several general concepts about important implementation choices. It starts with the integration of OpenLayers with Vue using data injection. It then explains how the management of events from the map has been simplified thanks to reactive programming.

The purpose of this section is to help future developers that would like to work on the application. Nontechnical people can therefore safely skip it. Indeed, the principles used to integrate the map are an advanced topic in Vue. Furthermore, they can in fact be used to integrate any library that does the rendering outside the front-end framework. For example, it can be used to integrate a drawing library such as Canvas [23]. Also, reactive programming might not be known by students who would like to continue the work on this application. It is therefore quickly introduced in this chapter, along with its integration with Vue.

8.1 Map integration

Vue uses an extended HTML syntax, named the template, and a small amount of JavaScript to build and render small components, having each a single responsibility. The template handles the rendering while the JavaScript usually handles and processes data as well as events.

OpenLayers is a library that renders maps using object-oriented programming in JavaScript. Every class of this library renders an element of the map. For example, the `Map` class renders the map in the given place on the page. Likewise, the `Overlay` class adds an overlay on the map at a given position.

This does not fit nicely with the principles of Vue and the rendering through templates since the rendering will be done in JavaScript. Figure 8.1 shows how the rendering is done in OO style and how it should be in Vue. To solve that, classes of the library are mapped to Vue components that can be used in a template.

The remainder of this section discusses three techniques used to achieve the expected result of Figure 8.1. They are illustrated with the example of a map overlay that displays a DOM element on a map coordinate. First, life-cycle hooks are used to add and remove elements from the map at the right time. Second, template referencing is used to access the DOM elements. Finally, data injection is used to pass an instance of the map to every child component.

```
let map = new Map({
  layers: [new Layer({
      ...
      })],
  target: 'map',
  view: new View({
      ...
      })
})

map.addOverlay(new Overlay({...}))
```

```
<template>
  <v-map>
    <view>
      <layer>
        <overlay :position="[0,0]"/>
      </layer>
    </view>
  </v-map>
</template>
```

Figure 8.1: Comparison of object oriented style rendering (left) and Vue template rendering (right) to display a map in a page.

8.1.1 Life-cycle hooks

Life-cycle hooks are functions called at some point during the lifetime of a component. Two of them are used to integrate OpenLayers with Vue. First, the `mounted` hook, which is called when the component has finished its initialization, is used to instantiate and add the corresponding element to the map. Second, the `beforeDestroy` hook, which is called just before the component starts its destroy process, is used to remove the corresponding element from the map. Figure 8.2 shows a simplified version of the `Overlay` component.

```
export default{
  data() {
    return { overlay: null }
  },
  mounted() {
    this.overlay = new overlay({...})
    this.map.addOverlay(this.overlay)
  },
  beforeDestroy() {
    this.map.removeOverlay(this.overlay)
  }
}
```

Figure 8.2: Two life-cycle hooks are used in the `Overlay` component.

8.1.2 Template references

DOM elements that are generated by the template of a component can be accessed with Vue. An example where this is very useful is again the overlay. As explained previously, it displays a DOM element on a map. Without Vue, DOM elements have to be created by hand using JavaScript, which is cumbersome, especially when the elements have to be modified later on. With Vue, the one generated thanks to the template can be directly used. Figure 8.3 shows how this works.

```
<template>
   <div>This is an <span>Overlay</span></div>
</template>

export default {
  data() {
    return { overlay: null }
  },
  mounted() {
    // this.$el reference the DOM generated by the component
    this.overlay = new Overlay({element: this.$el})
  }
}
```

Figure 8.3: Vue makes it possible to directly reference the DOM elements associated to a template.

8.1.3 Data injection

In the code of Figure 8.2, a map instance is accessed with `this.map`. However, this example does not show where this instance comes from. The map component should pass its map instance to any of its descendants, like the overlay component that needs it. To do so, the provide and inject features of Vue have to be used. The map component is **providing** a map instance, and the overlay is requesting the map instance to be **inject**ed in its component. This is illustrated by Figure 8.4.

8.2 A tooltip with reactive programming

This section shows how a tooltip showing a help message when the mouse stopped moving for some time and hiding it when the mouse starts moving again has been built. The tooltip should also be hidden when the mouse is not on the map. This is very complex to do with regular JavaScript. Multiple events should be monitored, local variables updated, and timers started and stopped.

However, it becomes trivial with the reactive programming paradigm. It is a declarative paradigm with which it is possible to handle streams of data that can

```
{
  mounted() {
    this.map = new Map()
  },
  provide: {
    getMapInstance: () =>
        this.map
  }
}
```

(a) Provide in map component

```
{
  inject: ['getMapInstance'],
  computed: {
    map() {
      return this.getMapInstance()
    }
  }
}
```

(b) Injection in a descendent component

Figure 8.4: The provide and injection features of Vue can be used on the map instance to make it available to any of its descendants.

be manipulated with operators. In our case, the focus is on data coming from events. Indeed the whole behavior of the tooltip is based on the mouse movements. Rxjs [24] is a library that makes it possible to use this paradigm in JavaScript.

Figure 8.5 presents a marble diagram showing a simplified version of the tooltip that is only based on the mouse movements. Marble diagrams are used to explain how the streams are manipulated. Horizontal lines show the streams with time. Each of them has a name displayed on the left. Marbles on the streams represent data that is flowing. For example, on the `mouseEvent$` line, each grey marble represents a mouse movement with its position. The rounded rectangles show operations done on the streams. For example, the grey marbles of the `mouseEvent$` line are mapped to red marbles to define a new stream.

In Figure 8.5, the `show$` stream identifies when to show (green) or not (red) the tooltip. However, such a stream of data cannot be used directly as is in a Vue template. Thankfully, vue-rx [25] is a library that plugs streams into the reactivity system of Vue. This means that it will be possible to access the last value of a stream directly in a Vue template.

This chapter presented how Vue and reactive programming make it possible to derive data from multiple events effortlessly. It also explained how using the template rendering of Vue avoids generating DOM by hand. For both those reasons, using the Vue framework to develop the application has several obvious advantages, compared to what would have been achieved with other frameworks such as Angular [26] or React [10].

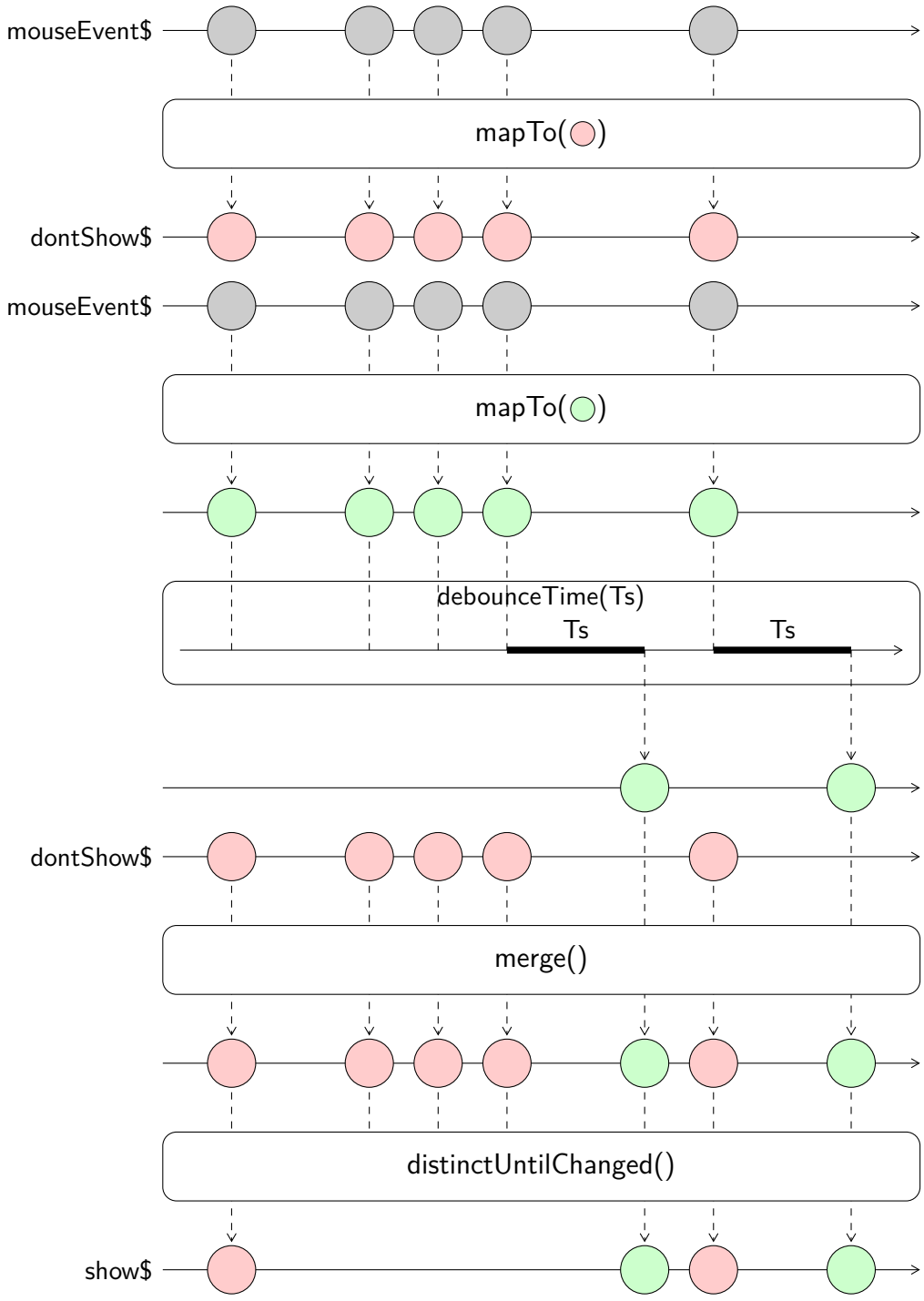

Figure 8.5: Marble diagram of the show variable of a tooltip

Chapter 9

Validation

The previous chapters identify the problems to be solved and propose a solution to hopefully solve them. The proposed solution has now to be verified. Two steps to prove the validity of the proposed solution have been identified. First, the fact that the software application can indeed help market gardeners as expected should be proved. Of course, the correctness of the implementation and the fact that it matches the imagined solution should also be discussed.

This chapter starts by stating all the methods used to verify the validity of the application. Then, it shows and discusses the results of the validation. Finally, it makes some reflections on the validity of the validation methods used.

9.1 Method

This section discusses how to ensure that the software implementation is correct using component and end-to-end testing. After that, it presents how a System Usability Scale score (SUS Score) can be used to assess how useful the users find the application.

9.1.1 Component testing

As presented in the previous chapters, Vue uses single responsibility components to build a user interface. The first step to prove that the software is correctly implemented is to check that each component behaves as expected. To be more future-proof, black-box testing is the preferable method to use. This means that some input are provided to the component and the resulting output are examined.

The internal workings of the components are therefore not examined at all. The benefit of such an approach is that the implementation of the component can be completely changed without having to modify the tests. The vue-testing-library

has been used, using its feature making it possible to render a single component and provide it the input it expects. Then, the produced DOM is queried to verify that it matches what is expected. For example, the component that displays the name of the selected garden can be tested as follows. A name is chosen and given to the component as an input. The test then checks that, once initializes, the DOM of the component contains a text box displaying the given name.

The DOM is queried with the help of accessibility roles [27]. These are roles given to DOM elements such as buttons, text boxes, or menus that, for example, allow blind people to use an application. As every principal element will be queried using these roles in the test, it has the benefit of also making sure that the application is accessible. An example is presented later on Figure 9.2 on page 41.

9.1.2 End-to-end testing

Testing each component individually makes sure that they behave as expected in isolation. However, it does not ensure that they are working well together. This is where End-to-End testing comes in. This kind of tests checks whether scenarios, that we expect a typical user would do on the application, are properly working.

With end-to-end testing, the tests are not focusing on small parts of the application as with component testing, but cover the application as a whole. For that, the cypress.io [28] library has been used. It runs a web application in a browser that can be programmatically accessed. Functions can be used to click and navigate the website just like how a user would do it. As with component testing, DOM elements can be queried to ensure that the correct data are displayed.

9.1.3 System Usability Scale

The System Usability Scale is a quick tool that measures the usability of a product. In software development, this score can be used to measure the user experience of an application. It computes a score between zero and one hundred based on the answers on ten very generic questions such as *"I think I would need help to use this system"* or *"I think this system is very easy to use"* asked to users.

By analyzing many results of SUS scores, [29] provide an interpretation of this score. Figure 9.1 summarizes their results showing that, for an application to be acceptable, it needs to achieve a score of at least 70.

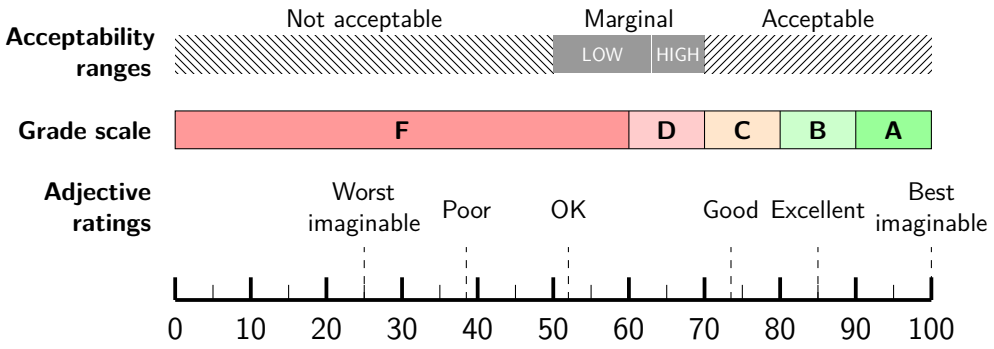

Figure 9.1: The score obtained on the system usability scale can be used to quickly measure the usability of an application.

9.2 Results and discussion

This section presents the results of the validation methods presented in the previous section and discusses what can be concluded from them.

9.2.1 Component and end-to-end testing

Component and end-to-end testings result in tests passing or not. A test that passes means that the output corresponds to what was expected for the given input. On the other hand, when a test does not produce the expected output and therefore fails, the software can be directly corrected to pass the test.

About 50 tests covering most of the UI components of the main application have been written. Figure 9.2 show an example of component testing for the `ErrorManager` component, which purpose is to display the errors to the user.

```
test('should display the given error for 2s', async () => {
    // Initialize the component
    const { updateProps, getByRole } = render(ErrorManager)

    // Update the input of the component
    await updateProps({ error: { message: 'some error' } })

    // Check that the error message is displayed
    expect(getByRole('alert', { name: 'some error' })).toBeVisible()

    // Wait for two second
    jest.runTimersToTime(2000)

    // Check that the error message is not displayed anymore
    expect(getByRole('alert', { name: 'some error' })).not.toBeVisible()
  })
```

Figure 9.2: Example of a component testing for the `ErrorManager` component.

Unfortunately, components built to render the OpenLayers map have not been tested due to some technical difficulties. Indeed, those components are tightly coupled together and are not meant to be used separately. This makes it challenging to use them in isolation. The API has not been tested yet as there are only a few lines of code. However, some difficult parts such as the generation of beds or the computing of the soil type under a bed must be tested, in the future. The mobile application has also not been tested yet as it is still in an early stage. The specification of the components are indeed still constantly changing, making the definition of stable tests not possible.

For end-to-end testing, several scenarios have been tested. A first one checks that the user can log in and log out. Other tests check that the user can draw, modify and delete parcels and that the information shown on the screen is changed accordingly. The possibility to generate the beds for a parcel has also been tested. Figure 9.3 shows an example of end-to-end testing, for the log in scenario.

```
it('should login', () => {
    cy.contains("Accéder à l'application").click()
    cy.get('#username').type('kim.mens@uclouvain.be')
    cy.get('#password').type('Het_Wachtw00rd_Van_Kim')
    cy.get('button[type=submit]').first().click()
    cy.url().should('eq', 'http://localhost:3000/no-garden')
  })
```

Figure 9.3: Example of a end-to-end testing for the log in scenario.

Finally, one of the features which is an overlay on the map giving information about the parcel hovered by the mouse has not been tested automatically yet, since the `hover` function is not available with `cypress.io`. However, it has been tested manually from time to time.

9.2.2 System Usability Scale

Only three people answered the survey to evaluate the SUS score of the developed application. This is not much but it provides a first idea on how well the application already looks at the current stage. As shown in Figure 9.4, it obtained a 68.33 SUS mean score. According to [29], this means that the application is in the higher part of being marginal. As the development is still in an early stage, it is a rather satisfactory result. The score could be improved by making the application more robust and by giving more information to the user, when errors occur.

Table 9.1 shows all the questions of the survey and the scores obtained for each one of them. Odd-numbered questions are positive statements while even-numbered ones are negative. The score column contains a value between zero and four, zero being the worst possible score and four the best one.

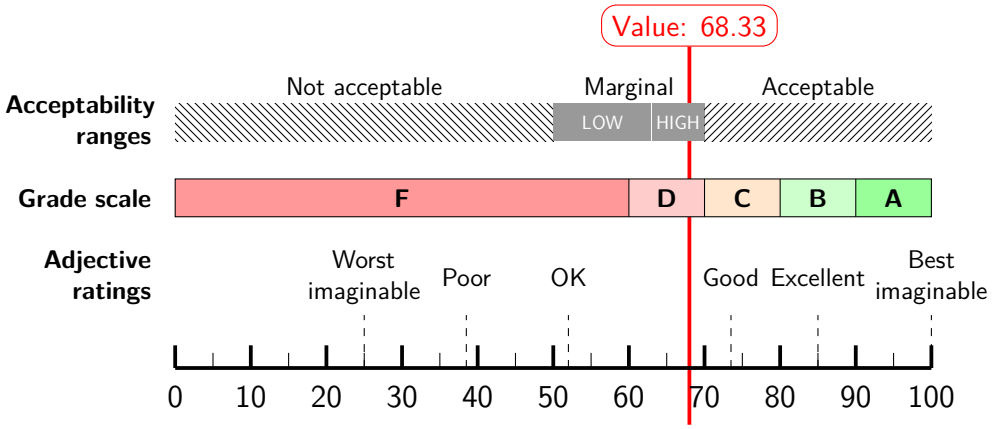

Figure 9.4: The average SUS Score obtained for the application from the survey puts it in the marginal high range.

Results presented in the table shows that all the negative questions got a score of three out of four, meaning that people who tested the application do not see it negatively. However, for the positive questions, two of them got a score below average. First, question 1 got a score of 1.33, which might be result from the fact that people who tested the application are not market gardeners and therefore do not need to use the application. Another reason might be that there are still important features that are not implemented yet. Then, question 5 got a score of 1.67. In the feedback collected from the users, they reported that it was not so easy to find some of the functionalities. This observation might explain the low score obtained for question 5. Finally, the best result is question 9 that got a score of 2.67, meaning that users are not afraid of breaking the system.

9.3 Validity

This section discusses the validity of the validation methods that have been used. It first starts with the validity of the component and end-to-end testings. It then discusses the system usability scale.

9.3.1 Component and end-to-end testing

In software development, it is generally impossible to prove that an application is 100% correct and does not contain any bugs. It is only possible to prove that some identified situations behave correctly. Indeed, only specific scenarios are tested to assess whether they produce the expected result. To prove that an application is correct, an infinity of scenarios would be required.

	Question	Average	Score
1	I think that I would like to use this system frequently	2,33	1,33
2	I found the system unnecessarily complex	2	3
3	I thought the system was easy to use	3,33	2,33
4	I think that I would need the support of a technical person to be able to use this system	2	3
5	I found the various functions in this system were well integrated	2,67	1,67
6	I thought there was too much inconsistency in this system	2	3
7	I would imagine that most people would learn to use this system very quickly	3,33	2,33
8	I found the system very cumbersome to use	2	3
9	I felt very confident using the system	3,67	2,67
10	I needed to learn a lot of things before I could get going with this system.	2	3

Table 9.1: Score for each individual questions.

However, testing increases the confidence of the developers that the software is correct. It also makes the developers more confident about the fact that the correctness of features will not regress with the software evolution. New functionalities are indeed added to the software and existing ones can be changed. Testing checks that these evolutions does not break existing features.

To be confident about the fact that the components are correctly implemented, a measure of how well the components are tested is needed. Thankfully, the chosen testing framework computes some metrics, called code coverage. The values of these metrics give some insights into the tests' quality. The easiest one to understand is the statement coverage. It represents the ratio of statements executed during the tests on all the statements of the application. If a statement has not been executed during testing, it might mean that some unexpected behavior may arise when the uncovered statements are executed. All but one component tested have a 100% statement coverage. However, one of the components still has a 85% coverage due to the fact that a way to test all the code of this component has not been found.

9.3.2 System Usability Scale

The goal of the SUS was to prove that the software effectively helps market gardeners. However, this scale only measures how usable an application is. It does not directly assess whether it solves or not the problems of market gardeners. However, having a good usability for the proposed system can increase the confidence that it can help gardeners. Indeed, if the software has all the features that they might want but is overly complex to use and have a terrible usability score, it will not be used by gardeners and will therefore not be able to help them. To have more confidence that the application solves gardeners' problems, a survey has to be conducted asking them what they think about the features, and whether they solve some of their problems.

Combining the methods described above increases the confidence that the proposed solution is on a good track, even if there is still some work, in particular to have a more robust application giving better feedback to its users.

Chapter 10

Conclusion and Future work

The goal of this master thesis is to improve an application developed in a previous master thesis [2]. The purpose of this application is to help market gardeners in their daily work. One of the aspects is to help them be more profitable. The prices at which they are selling their products are indeed aligned with traditional organic farming, even if market gardening usually requires more work.

The first analysis of the existing application quickly revealed that the current user experience needed to be improved by, for example, having a more dynamic application. However, changing the current application to do so would have produced a hard-to-maintain and a very complex code. The goals of the master thesis then shifted to the deep understanding of the problems of the current application to build a new one with most of the same features, but with a better user experience. Chapter 5 establishes a list of functional and non-functional requirements based on the analysis of the application and its domain.

The remainder of this conclusion goes through the requirements, the architecture and the user feedback and discusses what was achieved in this master thesis. It finishes by presenting some possible future work and perspectives.

10.1 Requirements

The new application has to offer a way to create a garden and its structure. While the information that we have on the garden structure stayed similar to the previous application, the way these data are provided to the system is completely different. With the new application, the time needed to create a garden has been reduced and simplified thanks to the automatic bed generation. Since the parcels are directly drawn on a map, the sun orientation does not have to be entered anymore and the soil types parcels are now automatically retrieved. The user experience moved from entering all the data by hand to just drawing parcels on a map.

The second established requirement is the creation of a planning feature with a completely different approach from the one taken from the previous application. The imagined approach was a simple planning that guides gardeners instead of telling them exactly what to do. Unfortunately, this is a huge task that may require a whole master thesis by itself.

Another feature that was completely changed from the original application is the time-keeping of the cultivation tasks. In the previous application, it was tightly linked to the planning. The latter gave farmers a task to do, and they could validate it by giving the time spent on the task. There is now a separate mobile application dedicated to time-keeping and that is not linked to any planning. When starting a task, the species, the variety, and the operation are selected, which starts a timer. It can be stopped by going back to the app when the task is done. In the previous application, users had to estimate the time per square meter spent on a task, and they now just have to tap on three buttons to start a timer and on another one to stop it when done.

This application is not fully functional yet. A way to store the task done offline and to sync them when the phone connects to the internet still has to be implemented. Also, a visualization of the encoded data should still be developed on the web application.

Finally, researchers can now access all the data they want directly with an API instead of connecting to the app and extracting data from the researchers' page.

10.2 Architecture

The final choices of architecture and technologies made for the developed application are relevant and contribute to its quality. As a reminder, three clients have been defined, fetching data from a GraphQL API made with Django and graphene. The first one is the web application made with Vue. Then, there is the mobile application made with NativeScript. The last client is the researcher's access that can be used to access data directly.

This architecture is flexible enough that it was possible to change plans during development, in an agile way. In the beginning, a dedicated application for the time-keeping was not planned at all. Instead, it was supposed to be part of the web application. Also, working with an API makes it easy to define and add new clients. As discussed in Chapter 8, the choice of Vue for the web application helps a lot for many aspects such the management of events and DOM generation. Django also helped to quickly build the API thanks to its ORM. However, even if it is easy to understand the basics of the framework, it can be very difficult when moving into advanced topics. Indeed, the documentation of Django is more like a huge tutorial, which makes it challenging to have an overall view of the functionalities.

10.3 User feedback

Feedback collected from users and the SUS score obtained for the application tend to show that the application is on the right track, even if there is still work to do.

However, a tutorial section should be added for the application, to quickly explain the new users how it works and what are its main features. Additional information about what each button does should also appear when users leave the mouse on a button for some time. Also, a new way to focus on a parcel should be added, to help users finding a specific one when the map is zoomed out. Finally, another feature recommended by a test would be to have a pin put on the map at all the places where there is a parcel, when the map is zoomed out.

10.4 Future work

In the future, the first thing that should be done is to finish the time-keeping feature so that gardeners can use the applications in their daily work. Also, the developed OpenLayers map components are very repetitive from each other. Code duplication should be removed from these components thanks to a complete refactoring taking into account a global view of all of them. The existing vue-layers [30] library does exactly what the developed OpenLayers map components do. However, it has a significant problem with the modification of drawn features which made it unusable for the developed application. It is a good idea to check, in the future, if the modification feature of this library has been fixed and then use it instead, resulting in less maintenance for the developed application.

As discussed in a previous chapter, the testing has to be improved to cover more parts of the system. It will consequently improve the confidence about the fact that all the applications are working as expected, which will lead to a better usability of the software.

Finally, many features can still be added that would be useful to market gardeners. First, there is the planning feature that would help gardeners maximize their land usage, having the least possible amount of time with nothing on the beds. This feature may also help gardeners to decide what species to put next to each other as it could also help with bed rotation. Another helpful feature would be to give gardeners a profitability index when selecting which vegetables to plant on the beds. This index could be based on the computed amount of work per square meter obtained from the previous years for each variety.

To conclude on a more personal note, we are happy with how the master thesis turned out. Even if not all the goals were met, we still did most of the analysis that will allow developers to finish the unmet goals in a short amount of time.

Bibliography

[1] Adrien Dockx, Rachel Sundar Raj, and Philippe Baret. Les petits producteurs maraîchers diversifiés. situation, enjeux, besoins et regroupement. Technical report, Université catholique de Louvain, 2019.

[2] Zélie Mulders. Conception of a market garden management application in partnership with lauzelle farm. Master's thesis, Université catholique de Louvain, 2018. Prom.: Kim Mens, Philippe Baret and Adrien Dockx.

[3] Evaluation de la performance écologique et économique d'un système diversifié de polyculture maraîchage-arboriculture fruitière (modèle "verger-maraîcher") sur petite surface en région wallonne. Technical report, Crabe ASBL, 2018.

[4] Mes Petits Légumes. https://mespetitslegumes.com, 2014.

[5] LEA | Logiciel pour Entreprise de Travaux Agricoles. https://www.lea-agri.com, 2021.

[6] Tend | Organic Farm Management Software. https://www.tend.com, 2021.

[7] Qrop. https://qrop.frama.io, Retrieved on June 2021.

[8] Django Software Foundation and individual contributors. The Web framework for perfectionists with deadlines | Django. https://www.djangoproject.com, 2021.

[9] Graphene-Python. https://graphene-python.org, Retrieved on June 2021.

[10] Facebook Inc. React - A JavaScript library for building user interfaces. https://reactjs.org, 2021.

[11] Evan You. Vue.js. https://vuejs.org, 2021.

[12] Nuxt.js - The intuitive Vue Framework. https://nuxtjs.org, Retrieved on June 2021.

[13] Introduction to Apollo Client - Client (React) - Apollo GraphQL Docs. https://www.apollographql.com/docs/react, Retrieved on June 2021.

[14] Maps, geocoding, and navigation APIs & SDKs | Mapbox. https://www.mapbox.com, Retrieved on June 2021.

[15] Vladimir Agafonkin. Leaflet - a JavaScript library for interactive maps. https://leafletjs.com, 2021.

[16] OpenLayers. http://openlayers.org, Retrieved on June 2021.

[17] Overview | Maps JavaScript API | Google Developers. https://developers.google.com/maps/documentation/javascript, Retrieved on June 2021.

[18] Microsoft Corporation. Bing Maps Dev Center - Bing Maps Dev Center. https://www.bingmapsportal.com, 2021.

[19] GIS Mapping Software, Location Intelligence & Spatial Analytics | Esri. https://www.esri.com, Retrieved on June 2021.

[20] WalOnMap | Géoportail de la Wallonie. https://geoportail.wallonie.be/walonmap, Retrieved on June 2021.

[21] Auth0: Secure access for everyone. But not just anyone. https://auth0.com, Retrieved on June 2021.

[22] Authlib. https://authlib.org, 2017.

[23] Canvas API. https://developer.mozilla.org/fr/docs/Web/API/Canvas_API, 2021.

[24] ReactiveX. http://reactivex.io, Retrieved on June 2021.

[25] RxJS integration for Vue.js. https://github.com/vuejs/vue-rx, 2019.

[26] Angular. https://angular.io/, 2021.

[27] MDN contributors. WAI-ARIA Roles - Accessibility | MDN. https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles, 2021.

[28] JavaScript End to End Testing Framework | cypress.io. https://www.cypress.io, 2021.

[29] Aaron Bangor, Philip Kortum, and James Miller. Determining what individual SUS scores mean: Adding an adjective rating scale. *Journal of Usability Studies*, 4(3):114–123, 2009.

[30] Vladimir Vershinin. VueLayers. https://vuelayers.github.io, 2019.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**