Conception of a market garden management application

in partnership with Lauzelle farm

Dissertation presented by
Zélie MULDERS

for obtaining the Master's degree in
Computer Science and Engineering

Supervisor(s)
Kim MENS, Philippe BARET, Adrien DOCKX

Reader(s)
Hélène VERHAEGE

Academic year 2017-2018

Contents

1 Introduction **5**
1.1 Context . . . 5
1.2 Problem . . . 5
1.3 Motivation . . . 5
1.4 Objectives . . . 5
1.5 Approach . . . 6
1.6 Roadmap . . . 6

2 Problem Analysis: Market Gardening **8**
2.1 Terminology . . . 8
2.2 Daily life scenario . . . 9
2.3 Profitability of market gardens . . . 10
2.4 Research in gardening . . . 10
2.5 Existing tools . . . 10
2.6 Conclusion . . . 12

3 Problem Analysis: Software Engineering **14**
3.1 Description and requirements of the clients . . . 14
3.2 Terminology . . . 15
3.3 Application platform . . . 18
3.4 Adopted methodology . . . 18
3.5 Programming language . . . 18
3.6 Web framework . . . 19
3.7 Database . . . 19
3.8 Development tools . . . 20

4 Requirements **23**
4.1 Initial Mindmap . . . 23
4.2 Meeting with gardeners . . . 23
4.3 Conclusion . . . 23

5 Solution **26**
5.1 Market gardeners . . . 26
5.2 Research . . . 33
5.3 Library . . . 33
5.4 Privacy aspects . . . 33
5.5 Encoding . . . 35
5.6 Conclusion . . . 35

6 Architecture **37**
6.1 High-level design . . . 37
6.2 Low-level design . . . 39

7 Implementation **43**
7.1 Permissions . . . 43
7.2 Handling two databases . . . 43
7.3 Production . . . 45

8 Validation **53**
8.1 Unit tests . . . 53
8.2 Functional testing . . . 53
8.3 Manual tests . . . 53
8.4 Human validation by gardeners . . . 54
8.5 Human validation by researchers . . . 55
8.6 Conclusion . . . 56

9 Conclusion and future work **57**
9.1 Objectives . . . 57
9.2 Future work . . . 57
9.3 Conclusion . . . 58

Acknowledgements

I would first like to thank my supervisor and professor Kim Mens from the EPL Faculty for his time and listening throughout this semester. I would also like to thank my two other supervisors from the bioscience Faculty, Adrien Dockx and Professor Philippe Baret. It has been a real pleasure to work and discuss with them. I am grateful to Prof. P. Baret to have proposed this subject that brought me its share of learnings and meetings.

In addition, I would like to thank the market gardeners, especially Damien, Prisca, Annick and Camille, who took part in this project by giving their time, their comments and sharing their experience.

I would also like to thank Tanguy for his support and help during this project and Faustine for being such a good friend that she took the time, during her exam period, to read over part of this dissertation.

And last but not least, I would also like to thank my family for supporting me. I would particularly like to thank Dounia, Cyril, Salomé and Andrew for their wise comments and corrections.

Abstract

Even though market gardening is attracting increased attention from consumers and researchers, it is not always easy for gardeners to make ends meet. For the practitioners, not only is gardening a tough task by the huge work labour it imposes, but it also requires important efforts for its planning and organisation. The study of different gardening techniques and the associated profitability of vegetables draws much attention among some researchers from UCLouvain. To support these studies, the university has recently started a project on a farm in Louvain-la-Neuve. In this context, this master thesis aims to provide a software tool which could support both gardeners and researchers.

The first step of our work consisted in meeting some gardeners in order to better identify the main daily challenges they face in their planning and crop management. Meanwhile, it was emphasized that a key issue hampering research was the difficulty to gather reliable data sets.

Following these observations, this dissertation builds a web application offering, on the one hand, a planning tool for gardeners and, on the other, a data gathering platform for researchers. Our solution allows a gardener to create a virtual representation of his garden and to divide his land into parcels and beds. He can either create or use existing identity cards of vegetables. Each of these cards contains a roadmap of successive cultivation operations for the cropping of the concerned vegetables. Our tool provides reminders of these cultivation operations throughout the year and helps the gardener keeping track of his actions.

Several software engineering techniques were used throughout the development of this solution. It is important that the developed tool can be easily reused and maintained in the future. By using Python as main language, we ensure that most of the code is understandable by many developers, but also by most researchers. Concerning robustness, a large bench of automated tests is set up, continuous integration is used through Travis and maintainability is guaranteed by CodeClimate.

We have successfully developed an application that is now online and available for gardeners and researchers. To validate the usefulness and relevance of the developed application, we presented our solution to several gardeners. The development of this application showed the gardeners' interest in using technology to release them from a tedious planning. It also revealed the complexity of building a generic crop model.

Chapter 1

Introduction

1.1 Context

Feeding the world in a sustainable way is becoming a major problem nowadays. The conventional way of producing food does not seem to offer a long-term solution and tends to decrease soil quality. Traditional market gardens of human size is sometimes proposed as a more viable alternative, although their sustainability has not been demonstrated yet. Indeed, the efficiency of small scale vegetable gardening techniques is not well-known, as very little research has been conducted about this topic. On the other hand, gardeners with small scale vegetable gardening appear to have troubles making a good living. In addition, they have lots of paperwork to manage and their crop planning is also taking them a lot of time.

1.2 Problem

Vegetable growers should spend most of their time in their crops, taking care of their production, and not behind their desk doing planning and administration. Gardeners can be excellent vegetables growers but terrible at filling administrative tasks and thinking about an optimal cropping plan. However, planning and paperwork are two time consuming activities that need to be done. Gardeners should therefore receive support in order to ease the fulfilment of these tasks as nobody needs vegetable growers with a PhD in optimisation and accounting.

1.3 Motivation

It is often claimed that that market gardening is necessary to feed the world and yet it has been put aside by the recent technological evolutions. However, a new generation of gardeners who grew up with technology is eager to incorporate it into their work. At the same time, the Université catholique de Louvain conducts research in a farm located in Louvain-La-Neuve on different market gardening techniques in order to study their efficiency. The project around this new farm aims to bring people and students from different faculties together.

1.4 Objectives

This thesis aims to build a software system that will help market gardeners in their daily planning. It will also provide an interface for researchers where they can collect data relevant to their research but also from private gardeners. The objective is thus to have a gardener interface that helps gardeners in their daily planning by reminding them about forthcoming operations. The application will also provide a way of planning future crops. Finally, we want to provide access

to an overall view of the profitability of each vegetable based on the required working hours of the previous years and selling prices of these crops.

1.5 Approach

Our work started in June 2017, by pinpointing the eagerness expressed by the bioscience engineering Faculty to initiate research in the field of market gardening.This motivated us to design a software system aiming to ease such future studies. First, it was necessary to meet professionals in the field in order to identify their practical needs. We participated in several meetings with gardeners. This helped us define in a more concrete way the technical specifications of the software system.

After that, we were able to start the implementation part. We regularly met gardeners to get their feedback about our software and to keep an incremental approach. The validation of our system was two-fold: on one hand we implemented some automated tests, and on the other hand we also met new gardeners who were not implied in the development process to present them our system. Finally, we managed to build a solution useful for gardeners and to fulfil part of our initial requirements. However, following a highly incremental approach, these requirements and their priorities evolved during the project.

1.6 Roadmap

In this dissertation we will describe the approaches we adopted and detail the proposed solution as follows. We have chosen to organise our chapters so that at each new chapter we go one step deeper in the abstraction tree. This abstraction tree is represented in figure 1.1.

At the highest level we have the requirements definition that are detailed in the first three chapters. These chapters first delineate more precisely the two fields concerned by this thesis, namely market gardening and software engineering. At this stage, we also further justify why we decided to tackle this problem, what are the needs of the future users and what motivated our choice of Django as a web framework. In addition, we briefly present the evolution of the requirements.

Afterwards, our solution is presented and the implemented features are highlighted in Chapter 5. This part is dedicated to the description of the application from the users' point of view.

After these descriptive chapters, Chapter 6 and 7 go deeper in abstraction by describing first the architecture of the solution and then some specific implementation choices.

After that, the validations of the implemented system are presented. In particular, we describe (1) the tools that we used, such as Travis for automating our tests, and (2) all the tests we made to ensure the robustness of our system. In addition, the validation operated by real gardeners is presented.

Finally, we conclude in Chapter 9 by discussing to which extent our objectives were met, and we provide some ideas for future improvements.

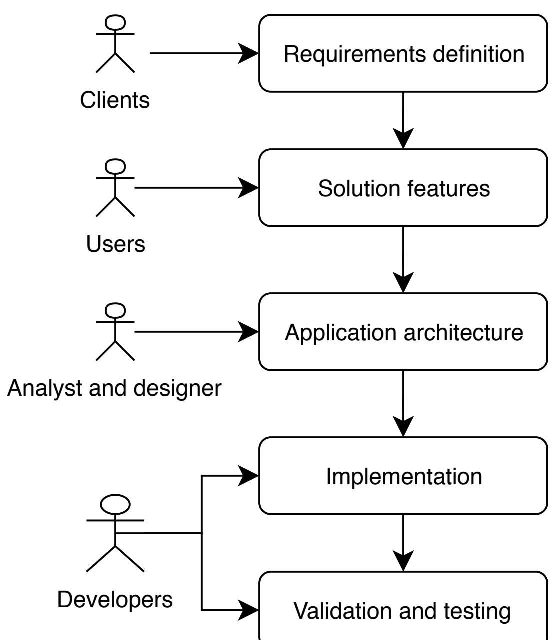


Figure 1.1: Abstraction tree for our system.

Chapter 2

Problem Analysis: Market Gardening

In this chapter, we will talk about market gardening. It is self-contained, as it allows the reader to understand this master thesis even without any any prior knowledge of this field. We will analyse this field and define different problems encountered during the management of a market gardening farm.

2.1 Terminology

First, it can be useful to define some common terms used in gardening.

Market gardening A market gardener, or a vegetable grower, is someone who produces fruit and vegetables on a relatively small area (under 10 hectares). The difference between a farmer and a market gardener is principally in the type of final product. Whereas a farmer will produce cereals, a market gardener is specialized in vegetables and small fruits.

Bed A bed in a market garden is a surface of production, usually a line. It is used to divide the field in smaller cultivated areas. A picture of beds is shown in Figure 2.1. Market gardeners usually choose the width of their beds according to the width of the tools they are going to use.

Figure 2.1: Beds in a market garden.

Crop rotation In order to preserve the soil and to eliminate some diseases specific to some plant species, some market gardeners rotate their cropping. If they plant one type of vegetable on a bed, the year after they will plant another type of vegetable on this bed. They will not plant two years in a row the same vegetable on the same bed.

Intercropping Growing different species on the same surface at the same time in order to improve productivity.

Cultivation operation An operation that has to be done on crops at a certain point in time. This can weeding, treating or anything else.

2.2 Daily life scenario

Seasons Market gardeners live by the rhythm of the seasons: they have a peak of work during spring and summer. Harvests continue during autumn but during winter they usually have less things to do in the garden.

Planning Most gardeners plan their cropping during winter [9], when they have more time to think about what they want to grow this year. Planning the coming year has several advantages :

- Know in advance what amounts of seeds and fertilizers they will have to order;
- Take the time to decide what to grow and in what quantities;
- Look back at the previous year to see which vegetables were the most profitable and adjust cropping according to this experience;
- Gain time during the rush season by having clearly in mind what has to be done;
- Organise the year to spread the work as much as possible (everything can not be seeded the same week).

While really useful, this planning part is not always done by market gardeners. Our application will provide help for the two last points. However, we did not implement a planning tool capable of generating a cropping plan and an order form for seeds and seedling. Such a system would be very interesting, but really complex if we want to take everything into account.

Adaptations Once the work season has started, this planning has to be adapted to the reality on the terrain. Weather is the major factor of changes to the planning. Indeed, some seeding requires several days of dry weather followed by one day of rain for example. In the case of difficult weather (late frost, high humidity,...), whatever was the initial plan, the gardener will have to adapt his schedule. Other factors that disrupt the work set-up can be diseases in the crops, short staffing or hardware issues. One example scenario could be: we are the first week of July, the season is in full swing. Tim is a market gardener and had planned to plant endives this week. The weather conditions are perfect, so he could stick to his plan. Unfortunately, his tomatoes have mildew[1] and if he wants to save his tomatoes' crops, he has to treat them immediately.

This example shows that some events have priority over the initial plan and confirms the idea that initial planning is meant to evolve.

It is essential for a market gardener to be able to adapt his plans to specific situations and to keep a clear head as the season progresses. Planning is already not an easy task, but adapting to changes is even harder.

[1] an epidemic fungus `https://en.wikipedia.org/wiki/Mildew`

2.3 Profitability of market gardens

Workforce Market gardeners don't count their hours as regular workers. They work all day in order to reach their objectives of the day. Most of them have no idea of how long each culture takes. It also means that they have no idea how profitable their cultures are. Moreover, they often need external workers to help them during the peak season. These external workers represent 50% of production costs [5]. Consequently, organizing the work to reduce the need of external workers can have a considerable impact on the garden's profitability.

Vegetables profitability Some vegetables are more profitable than others. For example, in his book, Jean-Martin Fortier [5] gives data about the profitability of the vegetables he's growing. However, most farmers don't do this analysis on their production and have therefore no idea of which cropping is the most profitable. Even in the table of Jean-Martin Fortier, we have no idea of the work time needed for each culture. And yet we have seen before that workforce represents a significant cost. Moreover, from one area to another it is reasonable to think that some crops will be more profitable than others. Depending on the clients' preferences or the soil type, some vegetables will be easier to sell or to crop. Gardeners are mostly not analysts and don't have the right tools to give them an idea of how profitable their business is and how they could be more efficient.

Others profitability factors There are two other profitability factors that are worth mentioning even if we did not take them into account in our requirements and specifications :

- Retail strategy: different retails strategies will give different revenues, the more intermediaries there are between the producer and the client, the less the producer will gain.
- Pricing strategy: of course, the selling price of vegetables will affect the profitability of these vegetables. Depending on which retail strategy is chosen, prices will be more or less flexible.

Antoinette Dumont has dedicated her doctoral thesis on the subject of market gardens' profitability.[4] We did not focus on these factors in our application, but they are interesting trails for future work.

2.4 Research in gardening

Vegetable crops have not been studied that much. For example, there are several studies for intercropping with cereals, but very few studies exist about intercropping for vegetables. However, lots of gardeners and lots of non-scientific books talk about intercropping with vegetables. A good reference for intercropping is given by Jean-Paul Thorez in his book [12] in the form of a double-entry table. This table is widely used but not scientifically proven. This is why the UCL has decided to initiate research in that field with the recent acquisition of the farm. To carry out such research it is essential to do experiments and to gather data in different situations with changes to some parameters (soil type, water supply, exposure,...).

2.5 Existing tools

From our research, not a lot of software exists to help farmers of all kind in their daily life. Furthermore, most of this software is not open source.

First, we found software like *Mes petits légumes* [7] intended for non-professional market gardeners, with a large library of data about lots of vegetables. The software can be bought

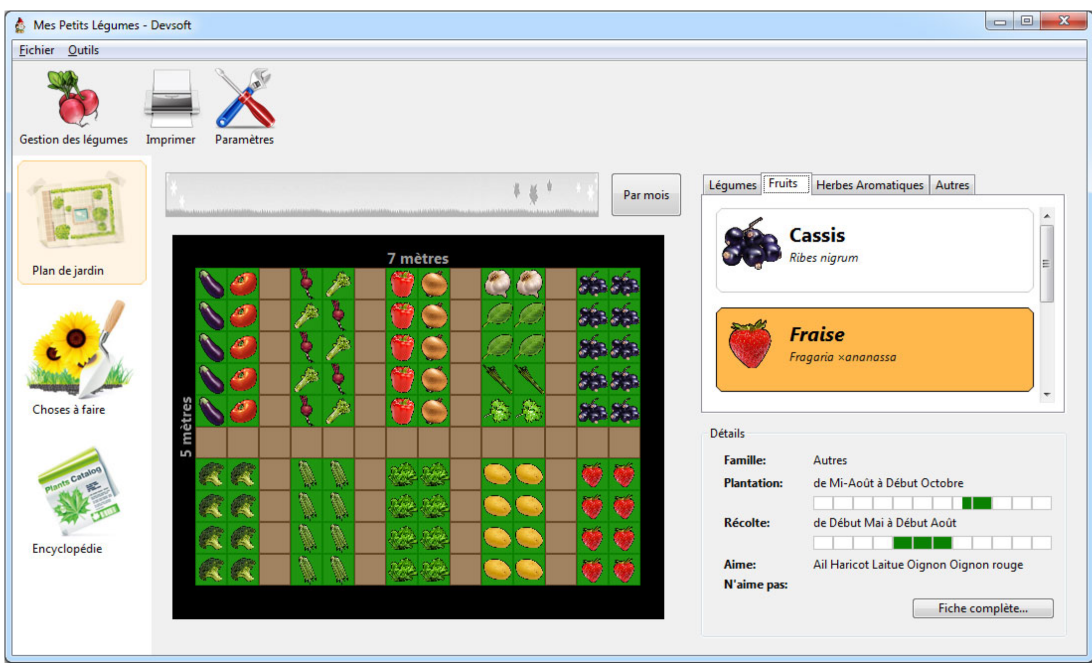


Figure 2.2: Visual representation of a garden planning.

Figure 2.3: Intercropping pieces of advice.

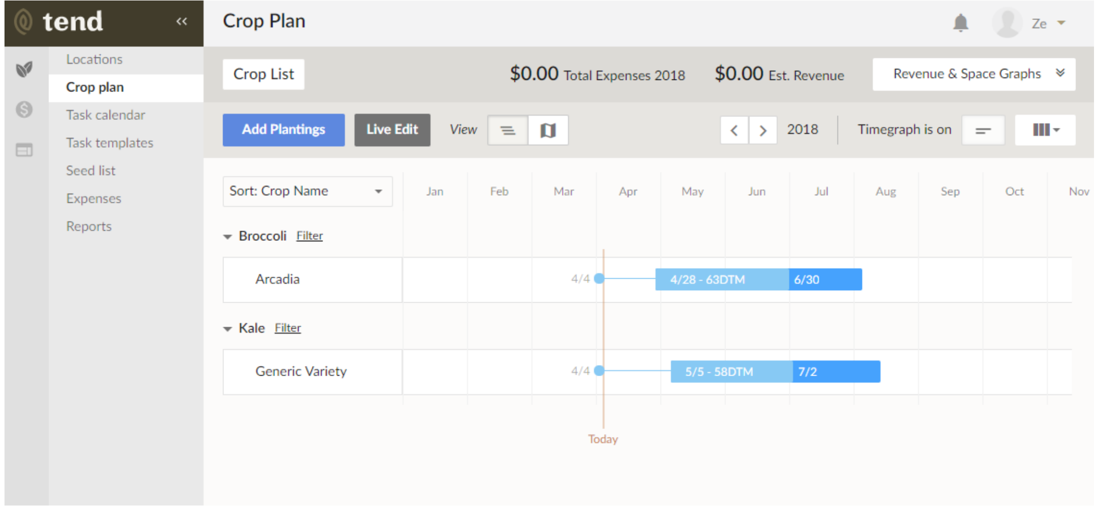


Figure 2.4: Planning view.

once for 19 € or one can use the free incomplete version. Two screenshots of this application are shown in Fig 2.2 and 2.3.

Then, we have software like *LEA* [6] which is more focused on the management of the business and intended for big farms. It can generate invoices for farm machines. It helps managing stocks and the use of fertilizers. Once again, the software is not open source: a paid subscription is required to use it.

Finally, we have found a software system that seems to have a purpose and a target audience similar to this project. *Tend* [11] is a software system developed in the USA by a startup since 2016. It has lots of features, including a databases of vegetables, a task calendar and an expenses section. Creating a crop plan by adding plantings is shown in Fig 2.4 and 2.5. This software is organised around three axes : grow, sell and advertise. The *grow* part looks like our application: a gardener can add locations, manage a crop plan, look at his task calendar, his seeds list and keep record of his expenses. The two other parts are linked since they ease the selling of products via a customised mailing list or website. They also have developed a mobile application where we can find most of the features again. This tool is quite complete and seems really adapted for the management of anglophone's farms (the whole application is in english and it uses the system of imperial units). The use of this software requires a subscription of 399$ per year or 39$ per month.

These software systems show that farmers are in need of tools to help them in their planning and management. The poorness of software really adapted to their needs show that this field has been mostly forgotten by technology.

2.6 Conclusion

From this analysis of the field of market gardening, we have seen that it is a field that is starting to get more attention from scientists. Therefore, there is a need for collecting data in order to conduct research. We can also say that market gardeners would gain in efficiency by having computing tools designed to help them in their daily planning. This is where this thesis comes in, trying to satisfy the scientists' need of data and the need for a planning tool for market gardeners. These two entities would both benefit from working together as data from one is useful for the other. Indeed, data collected from private gardens can be useful for the researchers and research results can help gardeners improve their techniques and knowledge.

In the next chapter we will analyse the problem in a software engineering way, i.e. describe the technical needs and requirements.

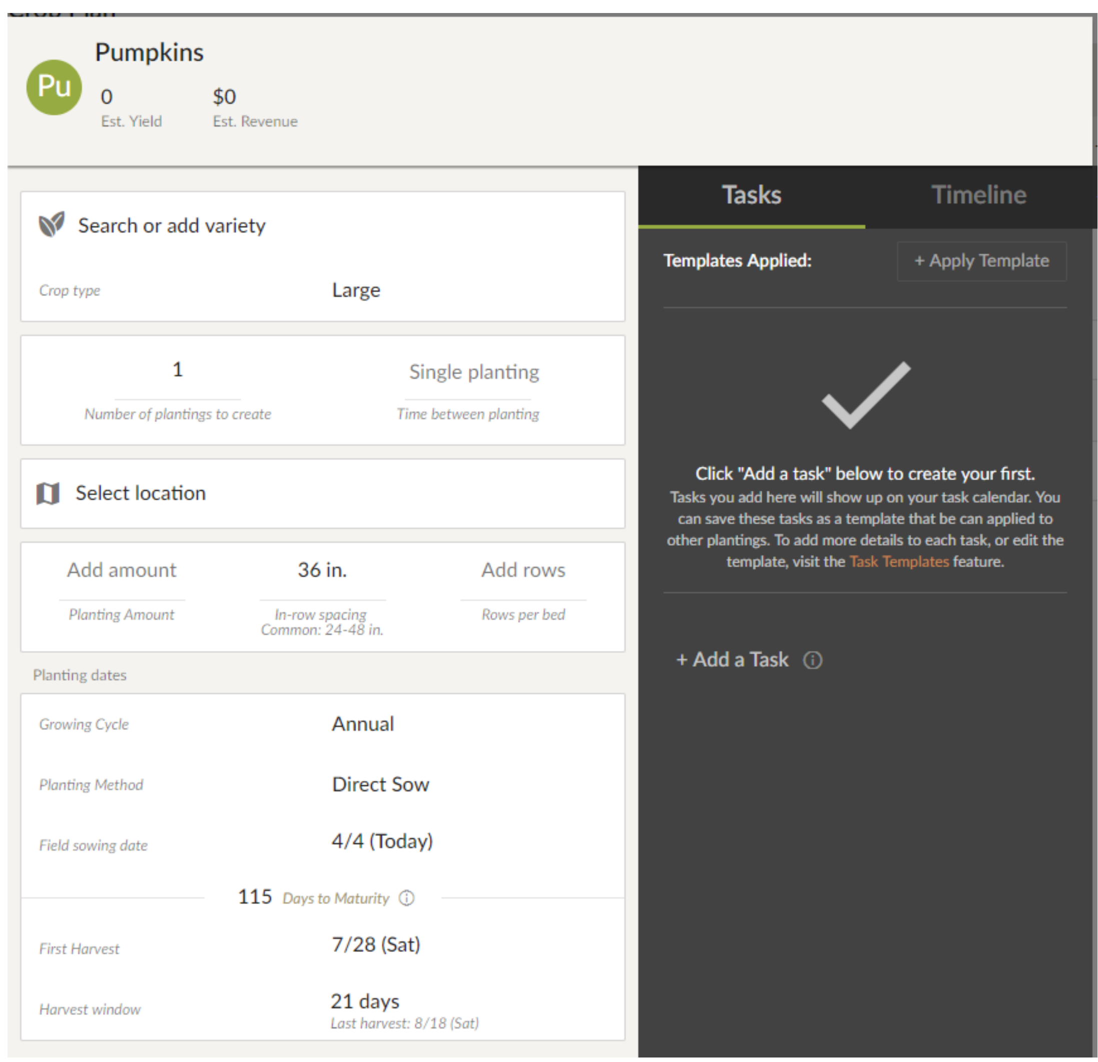


Figure 2.5: Adding a new planting.

Chapter 3

Problem Analysis: Software Engineering

In this chapter we will first describe the requirements from the point of view of both clients and then analyse and describe the technological choices that we faced.

3.1 Description and requirements of the clients

Our two main clients are the UCL and market gardeners. Their requirements are presented in this section. This section presents their requirements, while Chapter 4 focuses on their evolution through the different stages of the solution development

3.1.1 UCL

As already mentioned, market gardening is getting more and more interest from researchers over the last few years. The UCL is currently leading research in this field. The university has recently took in charge an old farm and plans to do some research in market gardening on the fields surrounding that farm.[1] The point of this research is to gather data about the viability, efficiency and profitability of different gardening principles. There are many theories about gardening on small surfaces, but not that much research on this topic and the efficiency of most of these theories has not been proven. Hence, UCL would like a web application gathering data from this forthcoming gardening project but also from gardens around the country. From our meetings with UCL's researchers we have defined some requirements:

- The application should have a researchers dashboard to allow easy access to gardens' data;
- Growers should be able to choose which pieces of information they agree to share with the university;
- Every task should have a *note* field so growers can give more details about anything they do;
- Every task should have a *duration* field in order to collect data on the time needed for each cropping.

3.1.2 Market gardeners

Before starting the project, we met several vegetable growers in order to obtain their opinion on the application and their advice. As the application is intended to help them, it was essential to meet them. From these meetings, we defined what were their requirements:

[1] `http://fermedelauzelle.be`

- The application should help them in their planning;
- In order to help their planning, the application should remind them about cultivation operations;
- The application should give them information about previous years and harvests;
- The application should have an economical side: one should be able to see which cropping is the most profitable.

3.2 Terminology

This section defines some concepts specific to software engineering, , which will be used in the following sections. This section can be skipped by experts on the subject.

Agile Methodology An agile methodology is a set of techniques and principles for conducting a development project. The main principle of agile software development is to iterate over short periods (called sprints) divided in sub-phases in order to build the final product in an incremental way. A sprint lasts between one and two weeks and is divided as follows:

- At the start of each iteration, we plan with the client what we are going to do this iteration;
- Next step is to think about how to build a good design to achieve the objectives;
- Then, we develop the features;
- After, we test these features. If we switch these last two steps, we apply what is called Test-Driven Development. With this methodology, a team writes tests before developing the corresponding features, ensuring that the tests will cover all cases;
- At the end of each sprint, we meet the client again to validate the changes and new features, collect his feedback about the project's progress and to define together the future work.

A good summary of the agile methodology is the Agile Manifesto[1], which states the main principles of this methodology.

Agile Manifesto [1]

> *Individuals and interactions* over processes and tools.
> *Working software* over comprehensive documentation.
> *Customer collaboration* over contract negotiation.
> *Responding to change* over following a plan.
> That is, while there is value in the items on the right, we value the items on the left more.

Fig. 3.1 presents a visual representation of this iterative approach.

In addition to its great flexibility, this methodology entails regular feedbacks of the client, improving the definition of the product to design.

Software framework In computer programming, a software framework is an abstraction in which software providing generic functionality can be selectively changed by additional user-written code, thus providing application-specific software. A software framework provides a standard way to build and deploy applications [10].

It aims to help developers creating software faster by providing mechanisms that are common to a family of IT solutions. Developers using a framework can specialize multiple parts of the software to handle their particular needs.

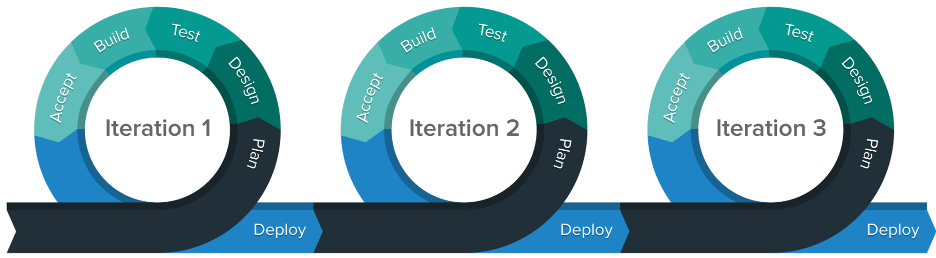


Figure 3.1: Iterations using Agile methodology.

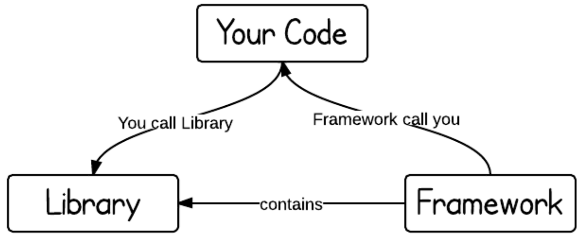


Figure 3.2: Framework versus library.

A framework is different from a software library in the sense that when using a library, we call the methods of the library while when using a framework, our code is called by the framework. This difference is illustrated in Figure 3.2.

The advantages of using a framework is the speed and ease of building a software solution because lots of functionalities are available out of the box. Furthermore, when using a framework, a developer has to follow some guidelines which lead to more reusable and maintainable code.

For example, a web framework is a software framework that provides lots of features common to many web applications, like database connection, template support, session management, etc. [14] During this master thesis, we used a popular web framework named *Django* that will be described in detail in Section 3.6.

Object Relational Mapping An Object Relational Mapper (**ORM**) is a software component that provides a bridge between data stored in a relational database and objects (in the sense of Object Oriented Programming). Figure 3.3 illustrates this principle. With such a tool, developers can abstract almost totally the database and focus mostly on the object-oriented model. The model is written using an Object Oriented language like Java or Python. It is then easy for developers to query, insert, update and delete objects in the database: the ORM abstracts the SQL part and developers just create and update classical objects.

The advantages of using an ORM are multiple but the main one is the abstraction of the

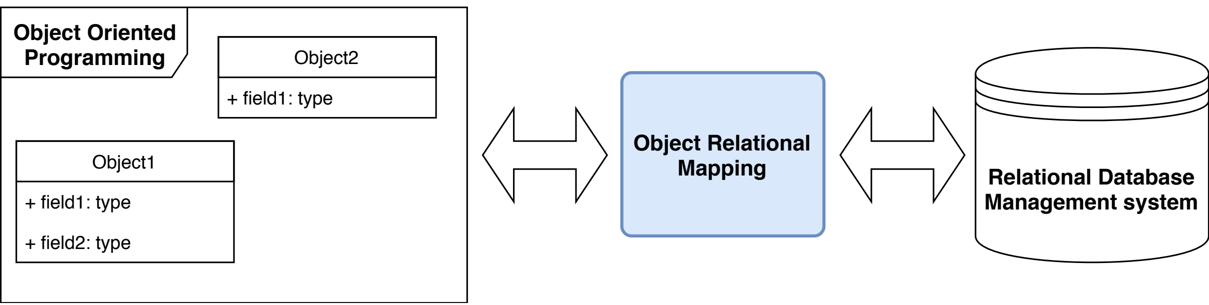


Figure 3.3: Illustration of Object Relational Mapping.

```python
class SimpleModel(models.Model):
    firstname = models.CharField(max_length=30)
    lastname  = models.CharField(max_length=30)
```

Listing 3.1: Simple python model

```
+---------------------------------+
|SimpleModel                      |
+---------------------------------+
|- id: number <Primary Key>       |
|- firstname: varchar(30) NOT NULL |
|- lastname: varchar(30) NOT NULL  |
+---------------------------------+
```

Listing 3.2: Simple model database table

database engine: it does not matter if you use a MySQL or a PostgreSQL database, the code will be the same. Another advantage is the ease of readability of a project: using an ORM, reading the model is enough to understand the structure of the data. Some ORMs are even able to create or update the database schema based on the model or the opposite: generate the model based on an existing database schema. It is worth noting that ORM also has drawbacks. The main disadvantage of using an ORM is the tight coupling between the object model and the database which makes using object-oriented features hard to implement (for example, databases do not support inheritance). Hence, if we want to use specific object-oriented features we have to add a layer between our object oriented design and the ORM, this layer is called Data Access Layer. Another drawback can be efficiency, depending on the models and the queries to be done, since the developer does not have control over the generated SQL code, queries can become slow. However, most ORM try to optimize queries. Another major drawback is that since models have been built with the ORM, we cannot reuse them somewhere else or even with another ORM. This is the reason why some ORMs are able to generate models based on the database schema. Nevertheless it is often easy to switch from one ORDBMS (object-relational database management system) to another.

Django provides its own ORM [2]. Each model is mapped to a single database table. Snippet 3.1 shows a simple Python model while snippet 3.2 shows the corresponding database table.

Continuous Integration Continuous integration (CI) is a software engineering practice to help development teams working on a fixed repository. When using CI, we automate a set of checks at each push request on a repository and developers should merge (or integrate) to this repository regularly (or continuously). With continuous integration, we will detect bugs faster and more easily. We can also add tests to our continuous integration tool. These tests will then be run every time new code is published and we will receive a notification if they fail. Thanks to that, failure origins can be detected easily. If we do small regular commits, it will be really easy to find where the failure comes from.

Section 3.8.2 describes Travis, the continuous integration software we used in this project. It is worth noting that, since we were only one developer on the project, the "integration" part of CI was not used as defined above (i.e. to check that the merged code of multiple developers do not break the application). However it was used to regularly run our tests suite and ensure that our modifications kept our system healthy. We tried to publish our changes very often to get regular feedback from our CI tool. Thanks to this practice, we were quickly aware of any breaking change.

[2] https://docs.djangoproject.com/en/2.0/topics/db/

3.3 Application platform

An important question to ask is which platform we are going to choose. We have two main clients; researchers at the UCL and gardeners. While researchers work mostly on their computer, gardeners spend most of their time on the field and don't always have regular access to a computer. We could have considered developing a mobile application, but then we have to deal with different operating systems (Android and iOS mainly). For these reasons, we have opted for a web application. A web application is available on every device having access to the internet, whatever the operating system is and the size of the device. With this solution we assure satisfaction of both clients.

3.4 Adopted methodology

At the start of the project, we had no clear specifications on the final result. The requirements have evolved over weeks. Because of this incremental definition of requirements and specification, it was natural to follow an Agile methodology (Section 3.2). As we were only one developer working on the project, we did not apply all principles of this methodology, however we kept the iterative approach by doing regular meetings with the clients. We had meetings with the UCL client once per week, with a gardener once per month and with the professor once per week or two weeks. In this project, the professor played the role of scrum master and technical referent. For each meeting, new features were presented to the client in order to have his approval (or not). These meetings were also the good moment to define priority for the coming weeks.

3.5 Programming language

To guide our selection of the ideal programming language we had several guidelines:

- As we want the application to be reusable and maintainable by future students or researchers from the UCL, we had to choose a language easy to learn and to understand, with a fast learning curve. From next year onwards, first year students at the EPL Faculty will be taught Python as first programming language and in the bioscience engineering Faculty, it is common that researchers use Python;
- As we are targeting a web application, we needed a language with web frameworks available;
- Because of the nature of our application and the requirements defined at the beginning of the project, we also wanted a language with object-oriented features.

From these guidelines it was natural to choose *Python* as programming language.

Python Python is a programming language used in many fields. Its main advantage is great readability. Its first release was in 1991 and its creator is Guido van Rossum. Python is an interpreted language which means there is no compilation stage before running the program. The interpreter executes it directly. This feature makes it fast and easy to use even for very small projects, but for bigger projects Python is known as being slow because there is no compilation optimisation possible. [13] However, Python has a big active community and thus great support and documentation which makes it a reliable language. Python is currently at the fourth place of the TIOBE index [3] which suggests that Python is a good choice of programming language.

[3] The TIOBE index measure the popularity of programming languages: `https://www.tiobe.com/tiobe-index/`

Figure 3.4: Four popular Python Web Framework.

3.6 Web framework

By choosing the Python language, we already narrowed the range of available web frameworks. Besides Python as requirement, the principal guidelines for the choice of a web framework were the following:

- It should be easy to use and understand as we have chosen Python partly for its readability;
- It should be stable and reliable, so with a good documentation and a community behind it, reporting bugs and keeping the framework up to date;
- It should have basic web features such as authentication, since we do not want to reinvent the wheel and re-implement existing modules.

From these guidelines, we spotted 4 popular Python web frameworks (Fig 3.4): Pyramid[4], Bottle[5], Django[6] and Flask[7].

Bottle and Flask are more intended for small projects and do not offer lots of support for bigger applications with more needs. For example, they don't have a built-in authentication module. Their key advantages are their small size and their fast installation. We dropped those choices (and other similar lightweight web frameworks) early on. Django and Pyramid are both designed to create web applications of medium to large size. They are both open source and have a large community. The main difference between Django and Pyramid is while the latest is a lightweight framework, the former is a high-level web framework. This implies that Pyramid, compared to Django, is well-known as being a lot more flexible. There is not a framework better than the other, they have different features and we have to choose which ones are the most important for us in our project. We did not choose Pyramid because we did not need the flexibility it offers. Instead, we opted for Django because of its maturity, its popularity and its wide range of built-in modules. We relied upon its ORM[8], its authentication module, its administration interface, its MVT[9] design pattern architecture and other features that will be discussed later in our implementation chapter.

3.7 Database

Once again, the choice of Django reduced the choices we had concerning the database we are going to use. First of all, Django does not officially support NoSQL databases. Official documentation is only given for PostgreSQL, MySQL, SQLite and Oracle databases. Our choice was made between PostgreSQL and MySQL because SQLite is intended for small databases and Oracle is not a free alternative. Once again, PostgreSQL and MySQL have both their own advantages. We have chosen PostgreSQL because it is the most popular relational database management system (RDMS) used with Django.

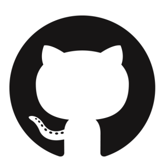
[4] https://trypyramid.com/
[5] http://bottlepy.org/docs/dev/
[6] https://www.djangoproject.com/
[7] http://flask.pocoo.org/
[8] Object Relational Mapping
[9] Model View Template

3.8 Development tools

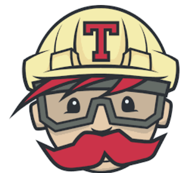
To ensure that we produce a quality software system, we used different tools. From version control to code coverage tools, we will present what we used and why we used them in this section.

3.8.1 Github

First of all, this project is open source. We put the source code online to make it easy for other people to contribute to the project. However, we added a non-commercial license[10] to restrict the usage of the project and prevent strangers from reusing the code to make profit.



We used Github and our repository can be found here : `https://github.com/ZelieM/MT-planting_planner`. Github is a version control system, that is, it manages all the changes brought to the project and allows several team members to work on the same project. External users can also choose to fork the repository and to make push requests. It means that anyone can suggest changes to the code.



3.8.2 Travis

To ensure our application behaves as expected, we wrote different kinds of tests. They are described in more detail in the Validation chapter 8.

Travis[11] is a Continuous Integration application. It is very popular in the open source community because it has a free plan for those projects. We took advantage of this offer to set up our CI process. The executions for our project can be found at `https://travis-ci.org/ZelieM/MT-planting_planner`. Travis simply requires to write a dedicated file *.travis.yml* containing our instructions. Our configuration, shown in snippet 3.3 is simple:

- Define the version of python we need;
- Declare that we need a PostgreSQL database;
- Define the version of Django that we use;
- Install the requirements of our application and a tool to post the code coverage to CodeClimate (presented in Section 3.8.3);
- Run our tests;
- Upload the code coverage to CodeClimate.

Travis can also be linked to multiple communication tools like Slack but since our team has only one developer, we kept the default basic notification system: email. As soon as a build failed (i.e. some tests were failing), we received an email and our time was then dedicated to fix the errors causing the failure.

[10] `https://creativecommons.org/licenses/by-nc-nd/4.0/legalcode`
[11] `https://travis-ci.org/`

```
language: python
python:
  - "3.5"

services:
  - postgresql

# setup environment
env:
  - DJANGO_VERSION=2.0.1

install:
  - pip install -r requirements.txt
  - pip install codeclimate-test-reporter

script:
  - export DJANGO_SECRETKEY=secret
  - coverage run --source='.' manage.py test --settings=planting_planner.
    settings.tests

after_script:
  - codeclimate-test-reporter --file .coverage
```

Listing 3.3: Configuration file for Travis

3.8.3 CodeClimate

CodeClimate[12] is a platform performing static code analysis[13] and collecting code coverage reports.

Like Travis, they propose a free plan for open source repositories. The dashboard for our project on CodeClimate can be found at `https://codeclimate.com/github/ZelieM/MT-planting_planner`. This tool is integrated with our Github repository so that it analyses our code for each commit published.

To analyse our source code, CodeClimate provides a set of plugins covering different programming languages. For example, we use a plugin named *pep8* to analyse our Python source code.

Maintainability Based on the results of the analytic tools, CodeClimate gives a grade to each version of our project. During the development of our application, we tried very hard to maintain this grade to "A", the best grade possible. This grade is shown in Figure 8.1. It represents an appreciation of the technical debt[14] and the time needed to fix it.

To compute this grade, CodeClimate considers the following aspects:

- Code smells: Typical bad patterns found by static analysis tools.
- Duplication: Code duplication should be avoided based on the Don't Repeat Yourself

[12] `https://codeclimate.com`

[13] Static analysis is performed without actually executing the code. It can cover many different aspect of code like ensuring code style (indentation, spaces, naming convention, etc.) or detect well-known bad patterns and code duplication.

[14] A technical debt appears when a developer choose an easy solution instead of a better (more maintainable) solution that would take longer to implement. The term "debt" comes from the fact that, the longer a developer waits before fixing the problem, the most difficult it becomes (the longer you wait before you "pay" it, the more "expensive" it becomes to fix).

(DRY) principle. Duplication leads to less maintainable code because any change made to one piece of code must also be done to the replicated code.

- Others: CodeClimate takes into account some metrics like the *Cyclomatic Complexity*[15], the length of files or the number of arguments per function.

Test coverage Running our tests generate a code coverage report. This report indicates which portions of our code are indeed tested and which are not. CodeClimate is able to receive this report and give us a percentage of code that is covered. Of course, the higher the test coverage is, the better.

However, it is foolish to think that a project must absolutely reach 100% of test coverage. Some portions of code do not require to be tested because they are so simple that it would actually take too much time and code to write and maintain a test for it. The tests could even be bigger and harder to understand than the code itself [16].

The opposite is foolish too: a project without any tests can easily break without even noticing. So it is important to find the good balance: test the critical parts and accept that some simple lines of code are not tested.

In our project, we reached a code coverage of 83% which seems reasonable to us. We think we managed to write the correct amount of tests to give us enough confidence that our application behaves as expected.

[15] The Cyclomatic Complexity is a count of the linearly independent paths through source code.[3]

[16] `https://dev.to/danlebrero/the-tragedy-of-100-code-coverage`

Chapter 4

Requirements

In this chapter, we will briefly describe the evolution of the requirements. Having to combine two clients, we had to follow a highly incremental approach.

4.1 Initial Mindmap

Initially, we were only working and cogitating with the bioscience engineering Faculty and their needs. Our first idea of the software has been put into a mindmap shown in figure 4.1. We can see that the features were divided in three poles: a planning, a management and a database part.

Planning : We thought about having a planning tool that would allow gardeners to split their garden into beds and to encode a cropping plan. On the server side, we would have implemented a module that checks that the cropping plan respects certain constraints. We pointed out three types of constraints: spatial, temporal and financial constraints. Spatial constraints would check if every bed is farmed and if intercropping is done, whether it is done properly (some types of vegetables go well with each others). Temporal constraints would make sure that crop rotation is done properly and that the gardener produces as much as he wants for a given period. Finally, financial constraints would check that the expected earnings are optimal.

Garden's management : We thought about the application as being helpful to manage a garden day by day, by encoding seeding, weeding or harvest. This daily encoding would generate statistics about the time spent on each crop or the selling price of each vegetable for example.

Database : This application should also maintain a vegetable library with data about a set of vegetables (not an exhaustive list).

4.2 Meeting with gardeners

After having discussed with the bioscience engineering Faculty it was time to go present our idea to the real future users: gardeners. From our meeting with the gardeners, we gave more priority to the management part of the application. Indeed, even if the planning part with the implementation and the verification of constraints is really interesting, they have a real need to have tools to help them in their day to day planning.

4.3 Conclusion

After this initial requirements phase, we kept having regular meetings with the bioscience engineering Faculty and gardeners to assess the relevance of our implementation choices and

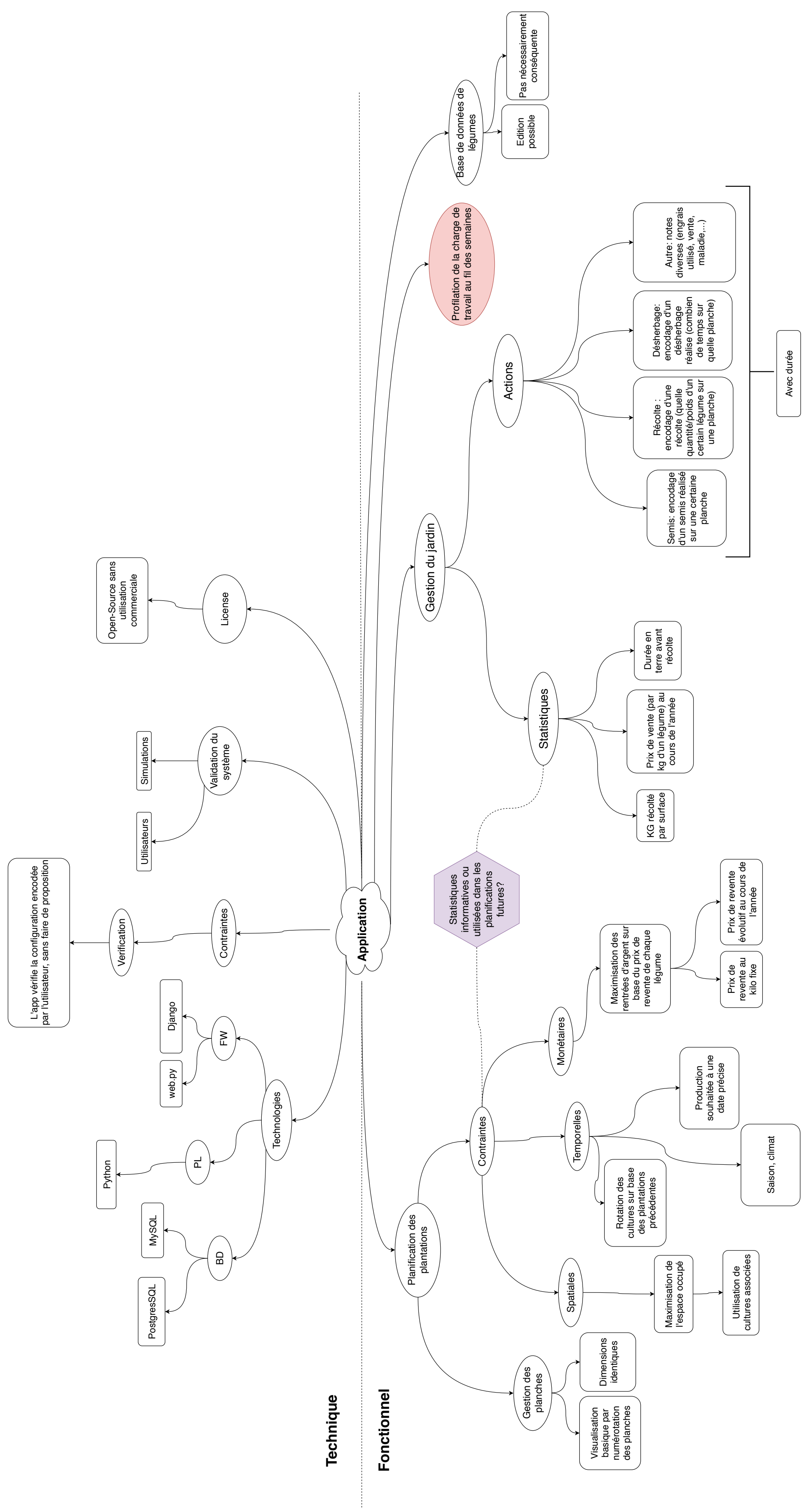


Figure 4.1: Initial mindmap.

ensure we were going in the right direction. This way of working directly implied to follow an Agile Methodology as stated before (Section 3.4). In the next chapter we will describe what we implemented in the final version keeping in mind the initial requirements described here.

Chapter 5

Solution

In this chapter we will describe in detail the features of our solution. First we will talk about the part of the application intended for gardeners, then the part for researchers and finally we will talk about the shared library. The solution is available online at the following url : `https://lauzeplan.sipr.ucl.ac.be`

5.1 Market gardeners

The main part of the application is accessible for market gardeners. They can use it to manage their gardens, vegetables and current plantings. In this section, we will describe the different features available for gardeners.

5.1.1 Gardens

First of all, any user of the application can create a garden and be associated to as many gardens as he wants. A garden is an abstract representation of a physical garden located somewhere on earth, with a history of plantings, a set of vegetables and other specificities. So the first thing to do after the creation of an account is to create a garden or to ask to be added to an existing garden. Every garden has at least a name and a postal code.

5.1.2 Beds

A garden has a set of beds that can be used to grow vegetables. As defined in Section 2.1, a bed is simply a surface. This view is shown in Figure 5.1. We can see that beds can be deleted and edited easily. The red round button can be used to either create a new bed or print the list of QR codes. This last feature will be discussed in more detail in Section 5.1.10. Every bed has at least a name, a width and a length, but as we can see in figure 5.1, the user can also give more detail such as a specific description (it could be that this bed does not retain water properly, or that it is located just next to the road), a soil type and an exposure.

From our interviews with gardeners in the validation phase (Chapter 8.4), we have added the concept of parcel which regroups a set of beds. This helps gardener organise better their beds.

It can be important for some gardeners to have a visual representation of their beds to have a clear idea of their location and their possible interaction in case of intercropping or crop rotation. For this reason, we implemented a very basic tool that allow the gardener to order his beds that can be seen in figure 5.2. However, this feature is more a proof of concept than a real feature. It could be worth developing this idea in the future and link the graphic beds' representation to a map location.

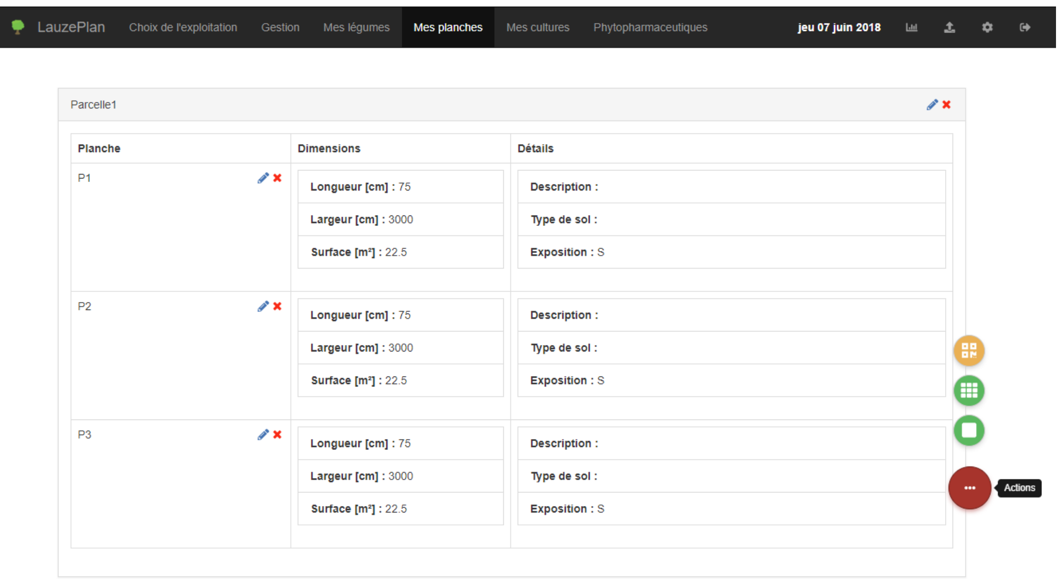


Figure 5.1: Screenshot of the beds view.

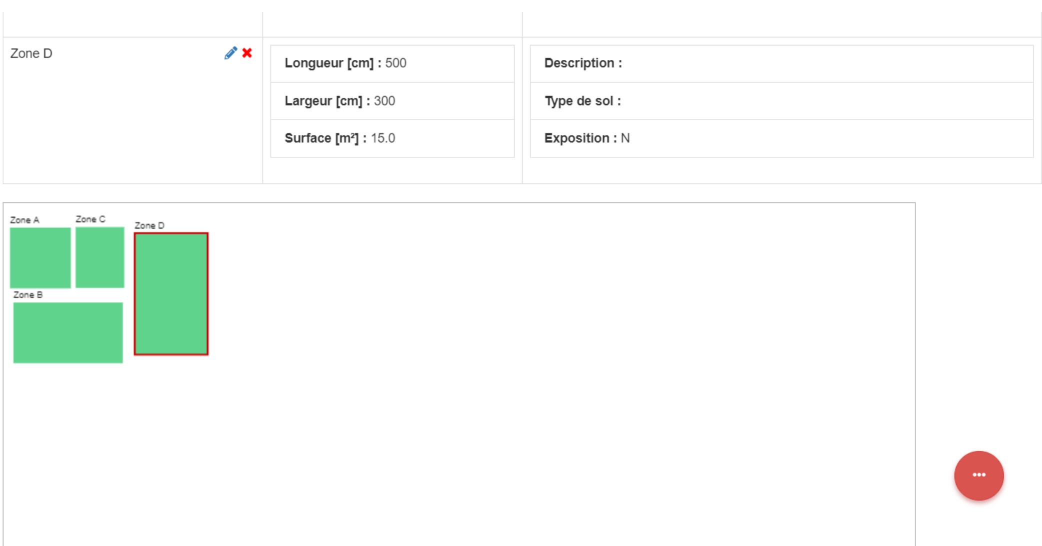


Figure 5.2: Screenshot of the beds visualisation tool.

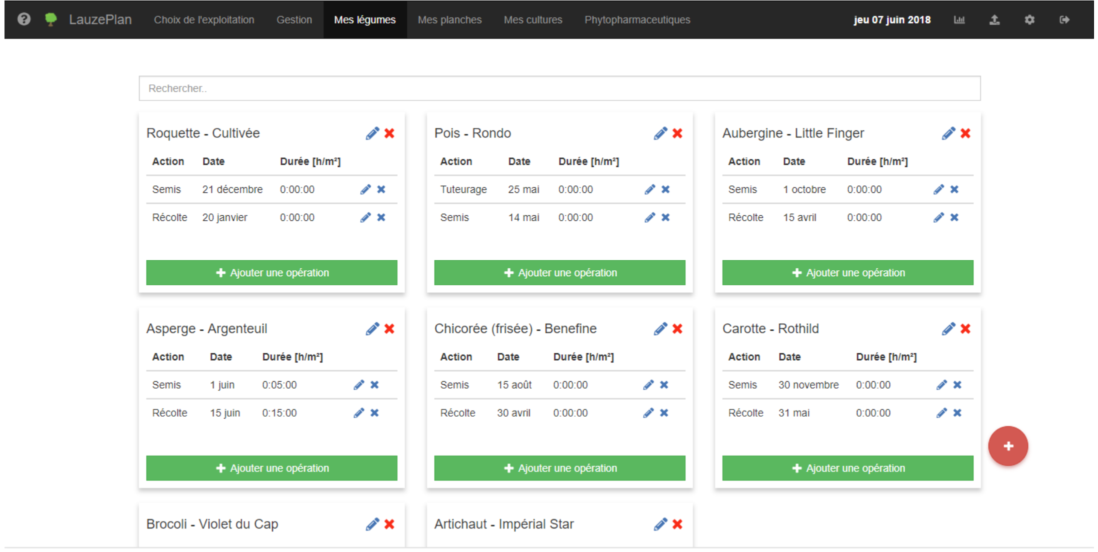


Figure 5.3: Screenshot of the vegetables view.

5.1.3 Vegetables

Each garden has a set of vegetables that any member of the garden can decide to grow. A vegetable has a name, a variety and some cultivation operations to be done during the growing period. Vegetables can be created from scratch or imported from a library. This library of vegetables will be discussed in more detail in Section 5.3. A screenshot of the vegetables page is shown in Figure 5.3. The import or the creation of vegetables is done via the red plus button. We can edit, delete and create operations. About the import, once a vegetable has been imported to a garden it is not synchronised with the library. This means that if the library's data changes it will have no impact on private gardens. Inversely, every change made by gardeners do not affect the original vegetables in the library. However, once copied in a private garden, the vegetable keeps a reference to the id of the original vegetable in the library. It is necessary to keep this reference so that we cannot import the same vegetable multiple times. We chose not to have a continuous synchronisation between the library and private gardens. We wanted gardeners to have their own instances that are not related to any other garden or public information. A gardener can choose to have completely different cultivation operations compare to his neighbour and still growing the exact same vegetable. For example, an organic grower will not spray his crops according to the same routine as a non organic grower.

5.1.4 Planning

The main feature of the application is to provide alerts and monitoring of what has been done and what remains to be done. This page is shown in Figure 5.4. We can see that a gardener can consult the future operations to be done, mark them as done, delete them or postpone them. When validating an alert, the application provides a form where the user can add more detail: the duration and an optional comment. This form does not have to be filled. Regarding

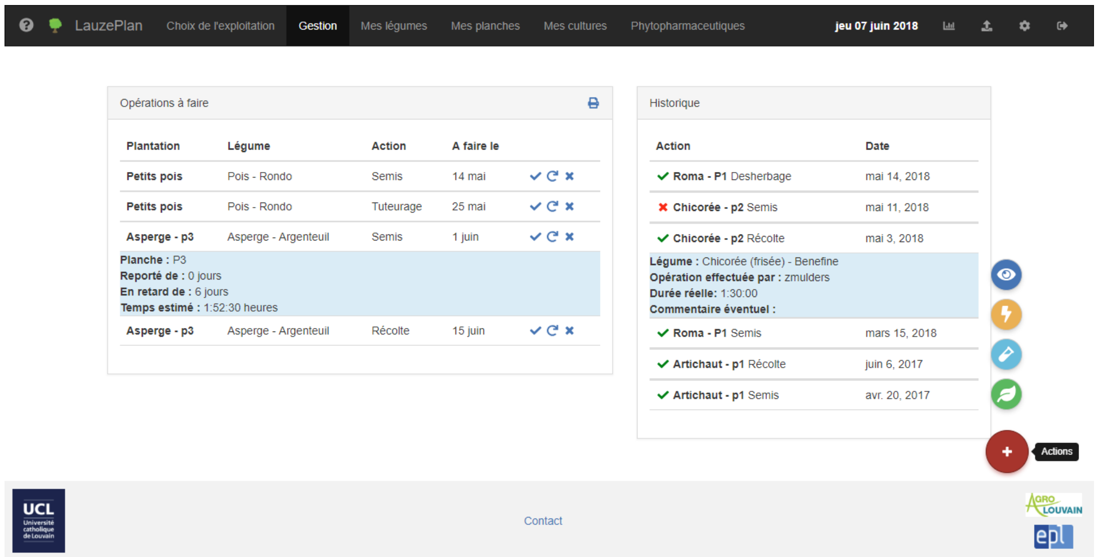


Figure 5.4: Screenshot of the main view.

the deletion, there are two options. Either the operation was useless and we only delete this particular operation. Or else, the entire crop has been destroyed and all following operations will also be deleted. The printer button will be explained in Section 5.1.11.

From this page, a gardener can also add a use of phytosanitary product, a punctual operation, an observation or a new planting. Adding a new planting will add all the cultivation operations related to the vegetable seeded to the future operations to be done. Finally, gardeners can consult the history of past operations in the right column. Every operation has at least an executor, a date and an area concerned. More comments can be added and consulted in the history. As we can see in Figure 5.4, every item, in alerts and history, is expandable (by simply clicking on it), providing more information on the operation.

5.1.5 Statistics

Another part of the application is about statistics. Gardeners encode estimated work hours by square meter for each cultivation operation. From this data and the surface seeded, we collect data and produce a graph of the estimated work hours. On this graph, we superimpose the real time spent on each crops. This statistics section is shown in Figure 5.5.

5.1.6 Crops

Another important page is the one listing all the crops. This page is shown in Figure 5.6. We can see the different beds with their current crops and their previous crops. We can also choose to sort our crops by vegetables instead of beds. A terminated crop has a productivity computed from the amount harvested and the selling price.

5.1.7 Phytosanitary products

The Federal Agency for the Safety of the Food Chain (AFSCA) requires that gardeners keep track of phytosanitary products that they buy and use. For this reason and by discussing with gardeners, we implemented a managing section for these products. In the phytosanitary view (figure 5.7), gardeners can register an incoming product or an usage of phytosanitary product. The information encoded can then be exported in csv format. We have based our models of

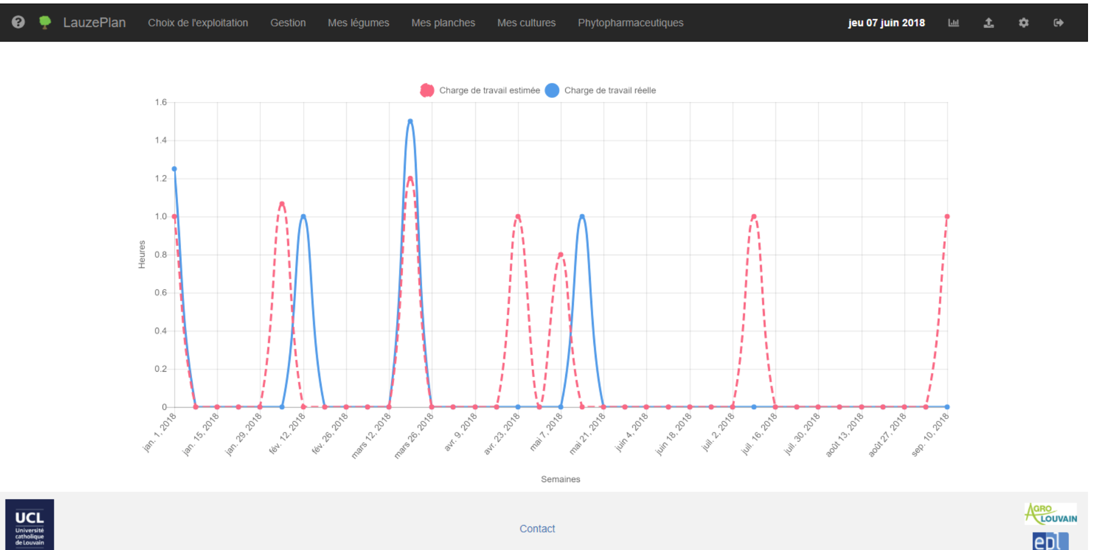


Figure 5.5: Screenshot of the statistics view.

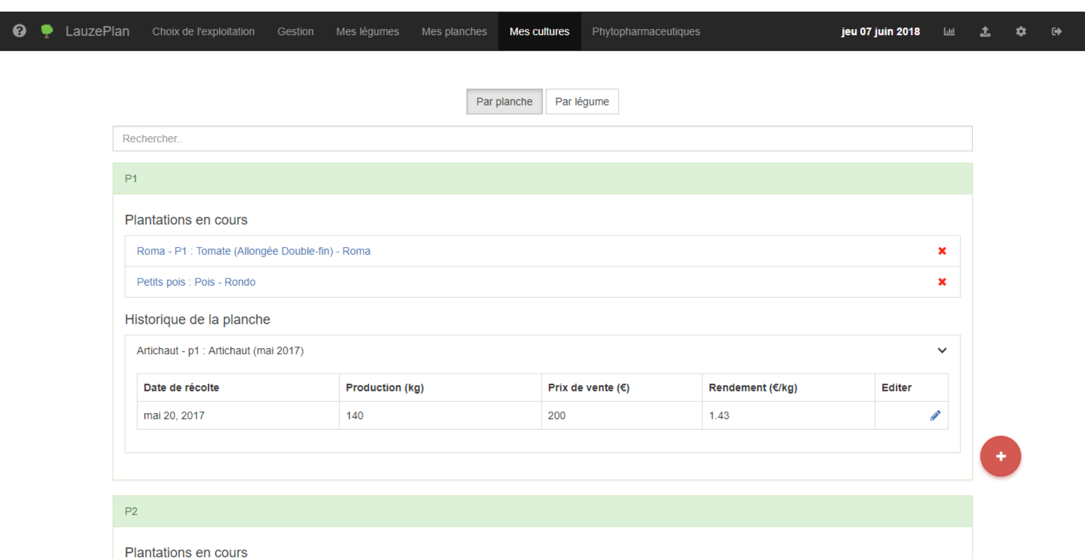


Figure 5.6: Screenshot of the crops view.

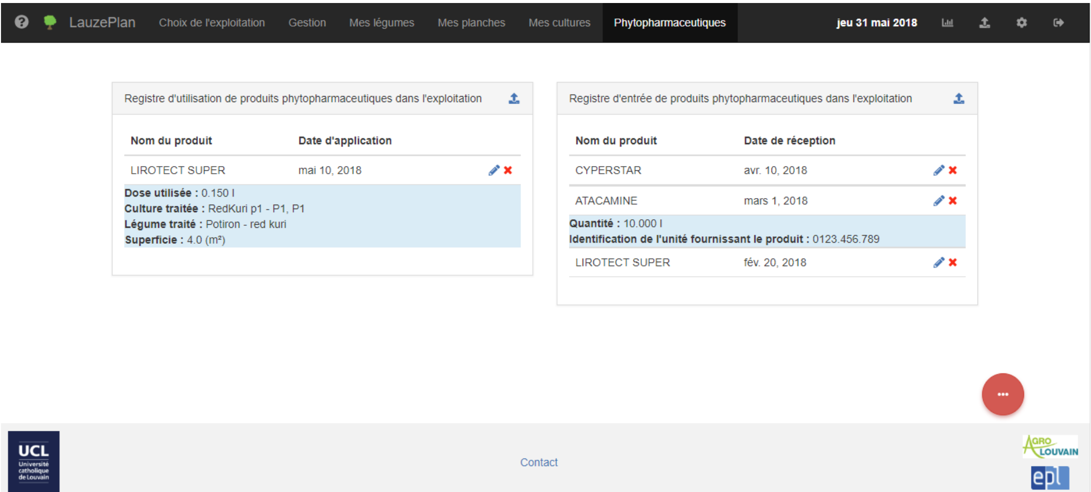


Figure 5.7: Screenshot of the phytosanitary products management view.

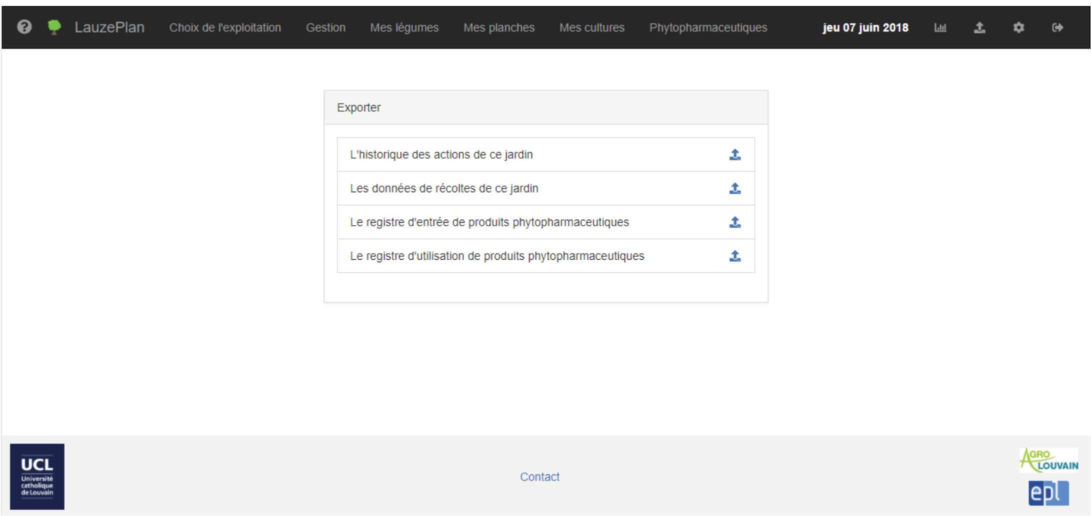


Figure 5.8: Screenshot of the export view.

usage and entry register on the vade-mecum for farmers written by the AFSCA [1]. By basing our models on this official document, we ensure that these exportable registers can be used in the context of an AFSCA check.

5.1.8 Export history data

Some gardeners like to have paper follow-up of their work. For this reason, we have implemented an export page (figure 5.8). On this page, we can export a csv file with the history of all operations done on this garden between two dates. We can also export harvest details or the phytosanitary registers we mentioned above.

5.1.9 Settings

On the settings page (figure 5.9), gardeners can add and edit pieces of information about the garden such as the garden's name, the garden's postal code, the soil type and a short description.

[1] http://www.afsca.be/publicationsthematiques/_documents/2012-07-20_Gewasbeschermingsmid_Produitsphyto_FR_V4_2018-01-25_web.pdf

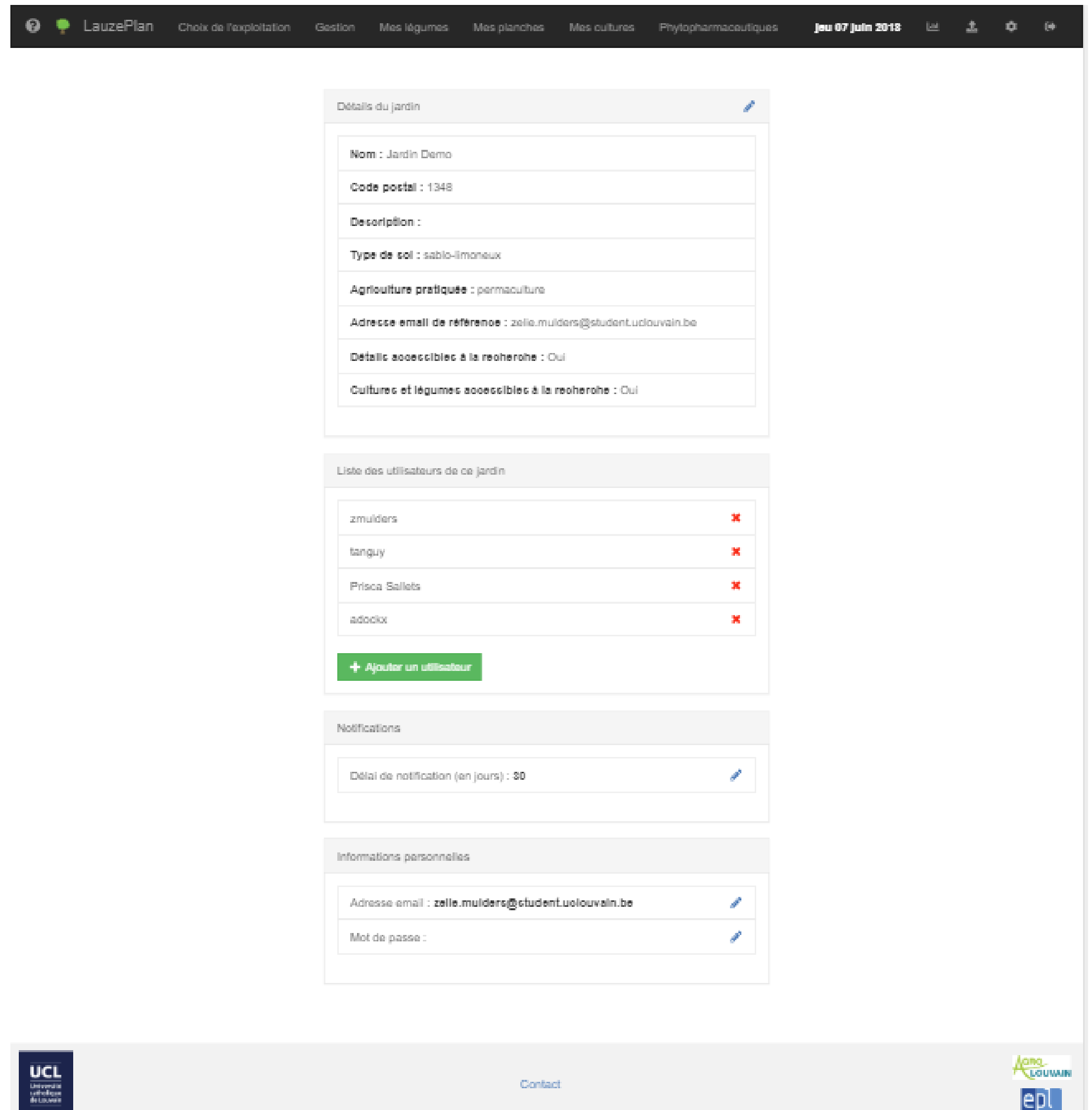


Figure 5.9: Screenshot of the settings view.

They can manage active gardeners on the current garden. They can also change their privacy preferences and their private information (email and password).

5.1.10 Geek gardener: QR codes

We have detected two types of gardeners. First we have gardeners that are comfortable with new technologies. Second, we have old school gardeners that prefer to work with paper and pen. For these two kind of profiles, we have developed specific features. First, for the geek ones, we have implemented a QR code feature. From the bed view, one can print QR codes that can then be placed next to the real bed. When someone scans the QR code, it will access a page where he can see what is currently seeded on this bed. He can also add a new planting. Then by clicking on an ongoing crop he will be redirected to the operation that needs to be done for this specific crop.

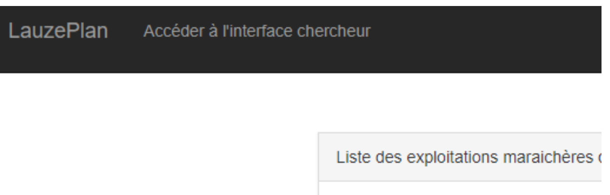


Figure 5.10: Research interface access.

5.1.11 Old school gardener: printable roadmap

As said before, there are two main type of gardeners. For gardeners that prefer to work with paper and pen, we have added a print button on the alert view (figure 5.4). The user will have the choice of the number of days he wants to print and will then have a printable file of the forthcoming operations. Of course, if he wants the application to be up to date he will still have to encode his actions online after having performed them.

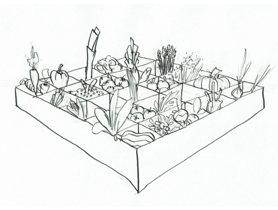
5.2 Research

Researchers have access to an interface from which they can consult data about gardeners using the application. Only administrators of the website are able to add researchers in the research team. Once added to the research team, the user can see a new button on the garden selection page (figure 5.10).

This will redirect to the following url : `http://lauzeplan.sipr.ucl.ac.be/research` (accessible only to research members).

5.2.1 Export

The research part has been designed to ease the access to the application data for research in the bioscience engineering Faculty. Hence, we find there the possibility to export in csv format data about registered gardens but also vegetables from the library. The two main views for exporting are shown in Fig 5.11 and 5.12.



5.3 Library

A library of vegetables and their data is available both for researchers and for market gardeners. Researchers are the only ones able to add, remove or edit vegetables in this library. Gardeners can copy them for their garden if they do not want to encode their own vegetables.

5.4 Privacy aspects

Researchers cannot use data from market gardens without the approval of the gardeners. To solve this issue we have added privacy settings where the gardener can choose what he agrees to share with the University. The settings are relative to a garden and not to user data. Indeed, researchers do not need to have access to the email address of every gardener of the garden, a reference email is sufficient. Furthermore the most valuable data concerns vegetables and crops and are therefore relative to garden, not to gardener. There are two degrees of security:

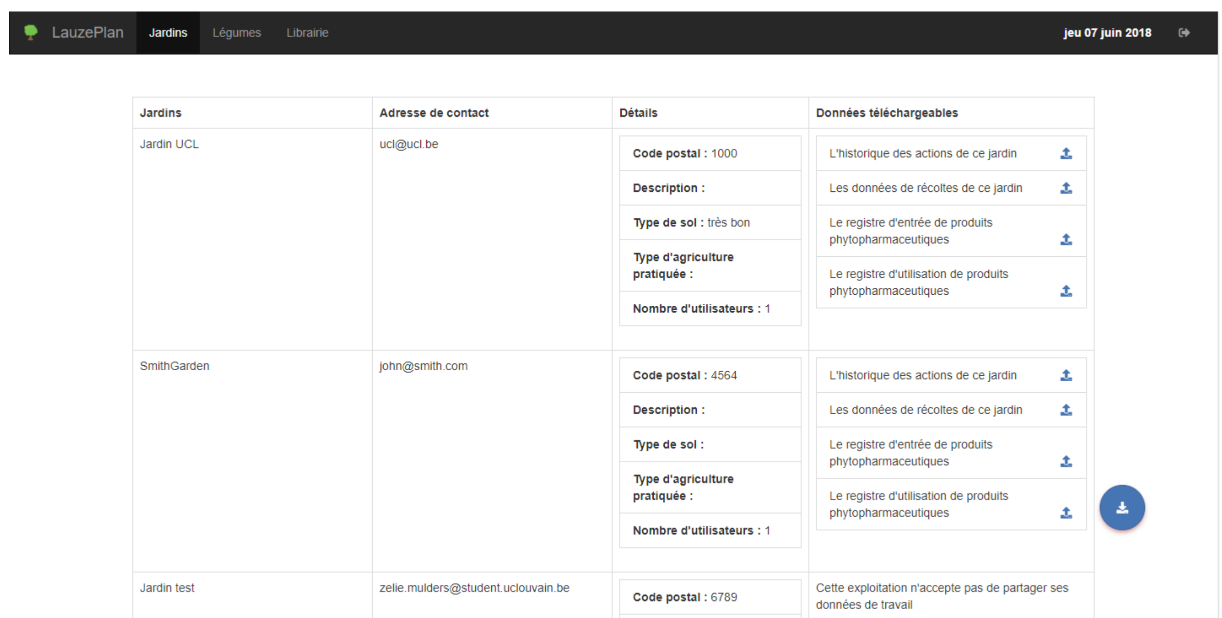


Figure 5.11: Research interface gardens data.

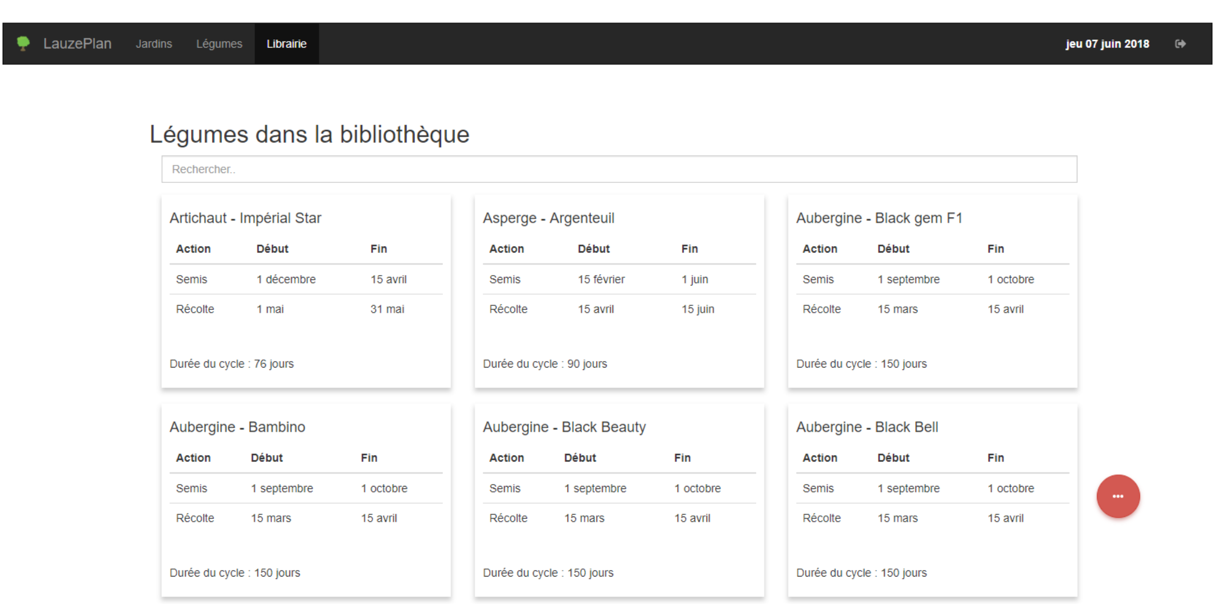


Figure 5.12: Research interface library data.

1. The garden agrees to share all personal information such as postal code, reference email, soil type and beds.
2. The garden agrees to share all vegetables and crops related data.

By default, these two settings are set to true and the gardener agrees to share his garden's information. Gardens' privacy settings are stored in the database and then used in the research part. Indeed, only the data from gardens in agreement are shown and exportable. These privacy settings can be updated by any gardener of the garden, but as we do not share critical data, we assume these settings will not be updated regularly.

5.5 Encoding

It is always hard to ask people to encode data. For this reason and for the application to be valuable from the start, we had to encode vegetables in the library. We contacted several seed sellers to ask if they agree to share part of their data. We got in touch with Agrosemens [2], Cycle en terre [3] and Semailles [4]. However, all of them replied that their database is not open source and hence they do not agree to share its content. Such a database takes time to be build and is therefore a valuable good. Therefore, we asked two students to encode data from Agrosemens' catalog to fill up our database. The encoding has been done via the Django's admin interface, which is the easiest and the most reliable way to encode valid data in the database.

5.6 Conclusion

In this chapter, we have skimmed the functions of our application. To summarize, we can have a look at figure 5.13 which puts together the different entities concerned by the features of the application. We put at the center three types of devices because we made sure or application is responsive and usable as well on a computer's screen as on a smartphone's screen. We even made our application a Progressive Web App (Section 7.3.9) to give the illusion it is an app on smartphones and tablets, more details are given in Section 7.3.9.

Hence the center of our figure 5.13 represents first the vegetables library part of our application, gathering information about a set of vegetables. Second, we have the frontend application, available on every device type and intended for gardeners and third we have the databases of our application, used to collect data and dealing with them in backend. The research part of our application is represented by the lowest third, where we can see researchers from the bioscience engineering Faculty working with data from the application. Next to it, we highlight the collaboration with the engineering school. Finally, on the top third of this image, we have the gardeners which are the main clients of this application (and as said before, we can distinguish two type of gardeners profiles).

In the next chapter, we will describe how we tackled this problem at the architecture level.

[2] `http://www.agrosemens.com/`
[3] `http://www.cycle-en-terre.be/`
[4] `https://www.semaille.com/fr/`

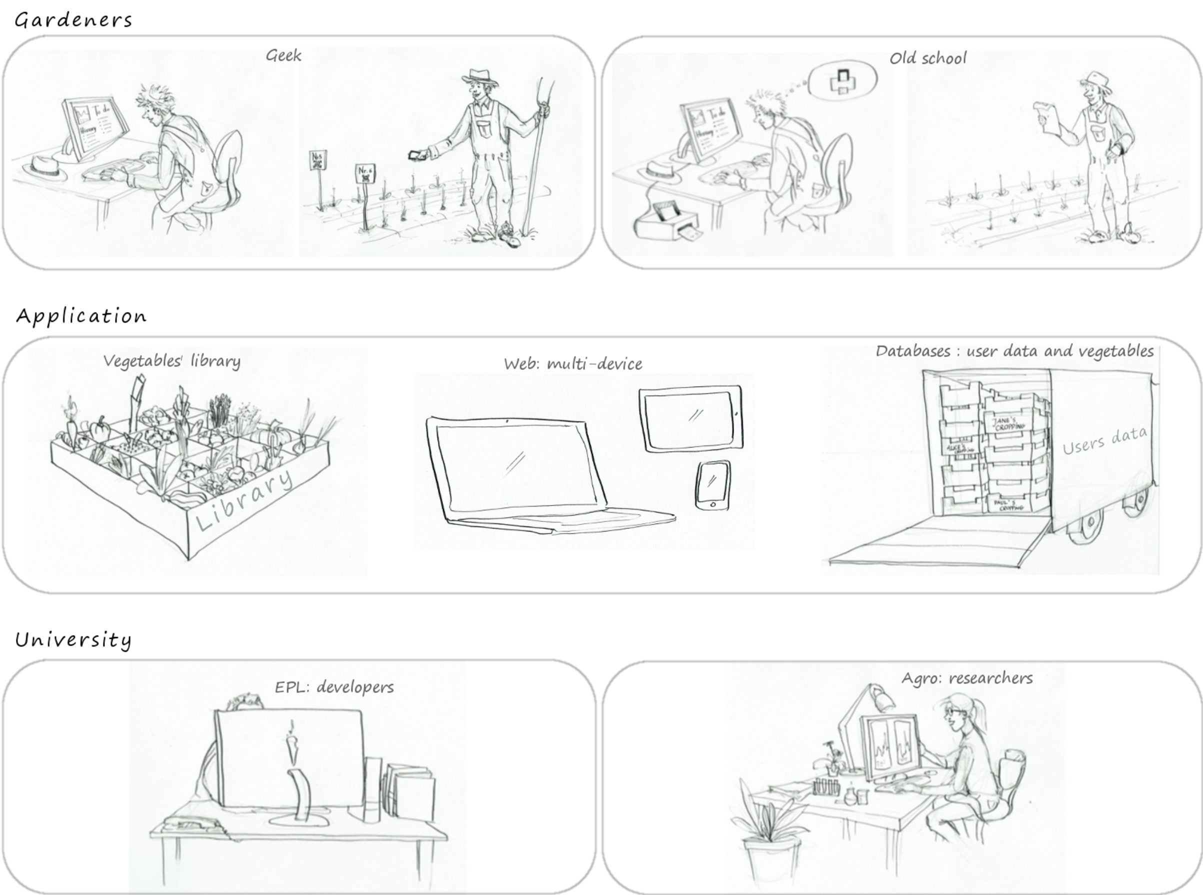


Figure 5.13: Functions summary.

Chapter 6

Architecture

Now that we have defined all the features of our solution, in this chapter we will get one level lower in the abstraction tree and talk about the architecture of the application. We will start with a very global view of the organisation and then go into more detail.

6.1 High-level design

In this section we will talk about the software architecture of the solution and the design pattern used by Django (model view template).

6.1.1 Software architecture

Our solution can be seen as three interacting parts. First, we have the users concerned by the application. Second, we have the application itself that can be divided in three parts. Finally, we have two databases that manage all the data generated and used by the three application parts. This decomposition is shown in Figure 6.1.

Users First the users, as we have already said the application is targeting two types of users. First, we have gardeners that will use the application to plan their season, gather data about their production, and consult some statistics. And then we have the researchers that are interested by data generated by the gardeners to conduct research.

Application's parts The application has three main parts :

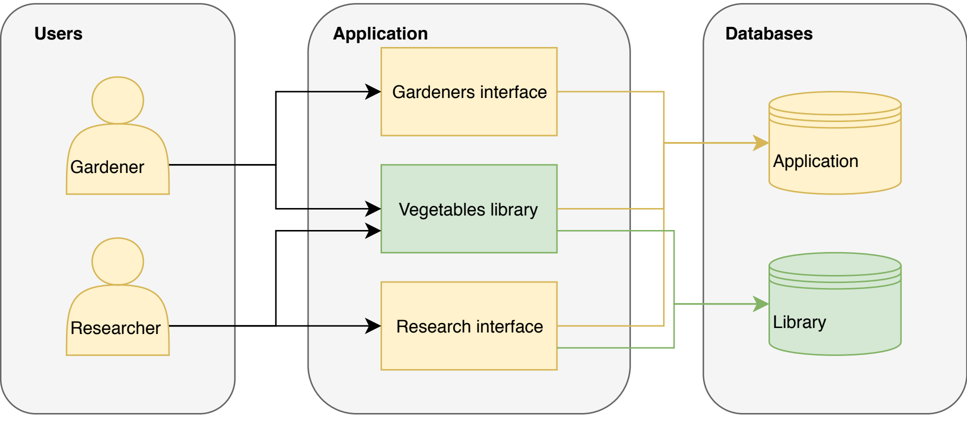


Figure 6.1: Software architecture.

1. Gardeners interface: this interface is dedicated to gardeners. The features of this part have already been described in Section 5.1
2. Vegetables library: the part of the application that contains data about vegetables that are accessible to everyone. This part is described in Section 5.3
3. Research interface: this last part is dedicated to researchers to access data of the two others parts of the application. More details are given in Section 5.2

Databases We have separated the vegetable library from the rest of the application so that if the library evolves, it will not affect the main application. To keep things clear and separated we have also created a second database. The default database concerns the application and all data generated by registered gardens while the library database contains all indexed vegetables. The configuration details of the two databases for production is shown on the snippet 6.1. We can see that they are both Postgresql databases located on the same server.

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'lauzeplan',
        'USER': 'lauzeplan',
        'PASSWORD': 'Jd5uHTFg',
        'HOST': 'pgsql.uclouvain.be',
        'PORT': '5440',
    },
    'db_vegetables_library': {
        'ENGINE': 'django.db.backends.postgresql',
        'NAME': 'lauzeplan_library',
        'USER': 'lauzeplan',
        'PASSWORD': 'Jd5uHTFg',
        'HOST': 'pgsql.uclouvain.be',
        'PORT': '5440'
    }
}
```

Listing 6.1: Databases' settings (settings/production.py)

6.1.2 Model View Template of Django

Django uses the model view template design pattern. This pattern is shown in Figure 6.2. By following the yellow arrows, we can reconstruct the path taken by a http request. First, a user goes to an url and sends a http request to the server. After a routing step (not represented in the schema) that matches urls to views, we get to the view. The view requested may need to collect some objects. Objects are themselves retrieved from the database thanks to the object relational mapper of Django. Then, the view can use these objects to create the context required by the template to be rendered. Once the context is ready, the view will render an html template and send it as response to the user.

Models define the database objects. Django has an integrated object relational mapper which handles the databases queries and the creation of databases tables. If we take a simple object such as a vegetable attached to a garden, we have the object defined in snippet 6.2 that will be transformed to a table in PostgreSQL with the following columns :

id	name	variety	garden_id	extern_id
[PK] integer	character varying (100)	character varying (100)	integer	integer

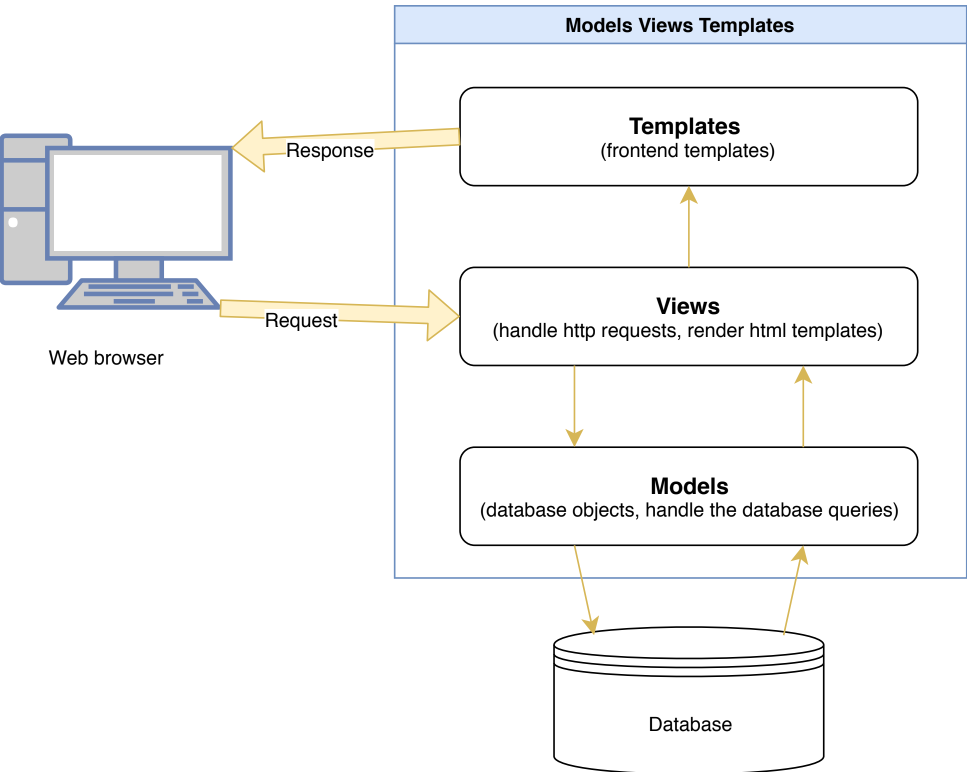


Figure 6.2: Model View Template design pattern.

```
class Vegetable(models.Model):
    name = models.CharField(max_length=100)
    variety = models.CharField(max_length=100, blank=True, default="")
    garden = models.ForeignKey(Garden, on_delete=models.CASCADE)
    # Field filled with primary key of vegetable from the library when exporting
    extern_id = models.IntegerField(null=True)
```

Listing 6.2: Vegetable object

So we can see that Djano's ORM adds a primary key by default to every object created. To sum up, models (or objects) are used in the views and stored in the database.

Views handle http requests and render the appropriate html template. Actually, http requests are first processed by a routing step that bind an url to a view. Afterwards, the view collects or computes data to be served and renders a template with the appropriate context. The views are implemented in Python.

Templates are html files that are meant to be rendered in a web browser. A context can be passed to templates and then we can call this context using double brackets (Ex : {{ context.object }}) in the template.

6.2 Low-level design

Low-Level design is when we talk about the inner components of a software system and their structure. We won't talk much about this level of detail as it can get quite long and cumbersome to itemize every component of our application.

Class diagrams : UML (Unified Modeling Language) diagrams model the structure and the relation between our different classes. Such class diagrams have been generated from Django by a Django extension and they can be found in figures 6.3 and 6.4. They are split in two because of the separation between the two main Django's applications: the application itself and the vegetables library.

The most interesting diagram is the one of the planner application (figure 6.3). We have the general objects such as a garden, a vegetable, a bed and a parcel. Then we have the concept of cultivation operation which is linked to a vegetable and has either a date or an offset and a previous operation. These objects represent the technical itinerary to follow along the year for every vegetable. We can see that a garden has a set of cultivated areas, it represents the current or previous crops: a cultivated area links a vegetable to a bed and is either active (currently cropped) or inactive (already harvested). Finally we have the concept of alerts or *ForthcomingOperations* which is linked to an original cultural operation, a cultivated area and can be postponed. Once an operation is marked as done, it stays in the table of *ForthcomingOperation* but *is_done* is set to true and we add the execution date. In addition to that we create an history item for this done area. History items can be either punctual operations, observations or alerts marked as done. It was important for researchers to keep a track of who has done every operation, that is why we have an executor field for every history item. In brief, a garden has a set of virtual representation of physical things such as a bed or a vegetable and a set of operations done or to be done.

The diagram of the vegetables library is simpler (figure 6.4) but has more detail for each vegetable, we have given more importance to plant taxonomy.

Database design: As said before, Django has an integrated Object Relational Mapper, the database design is thus directly translated from the model design. Therefore, UML diagrams are also representative of the database schema.

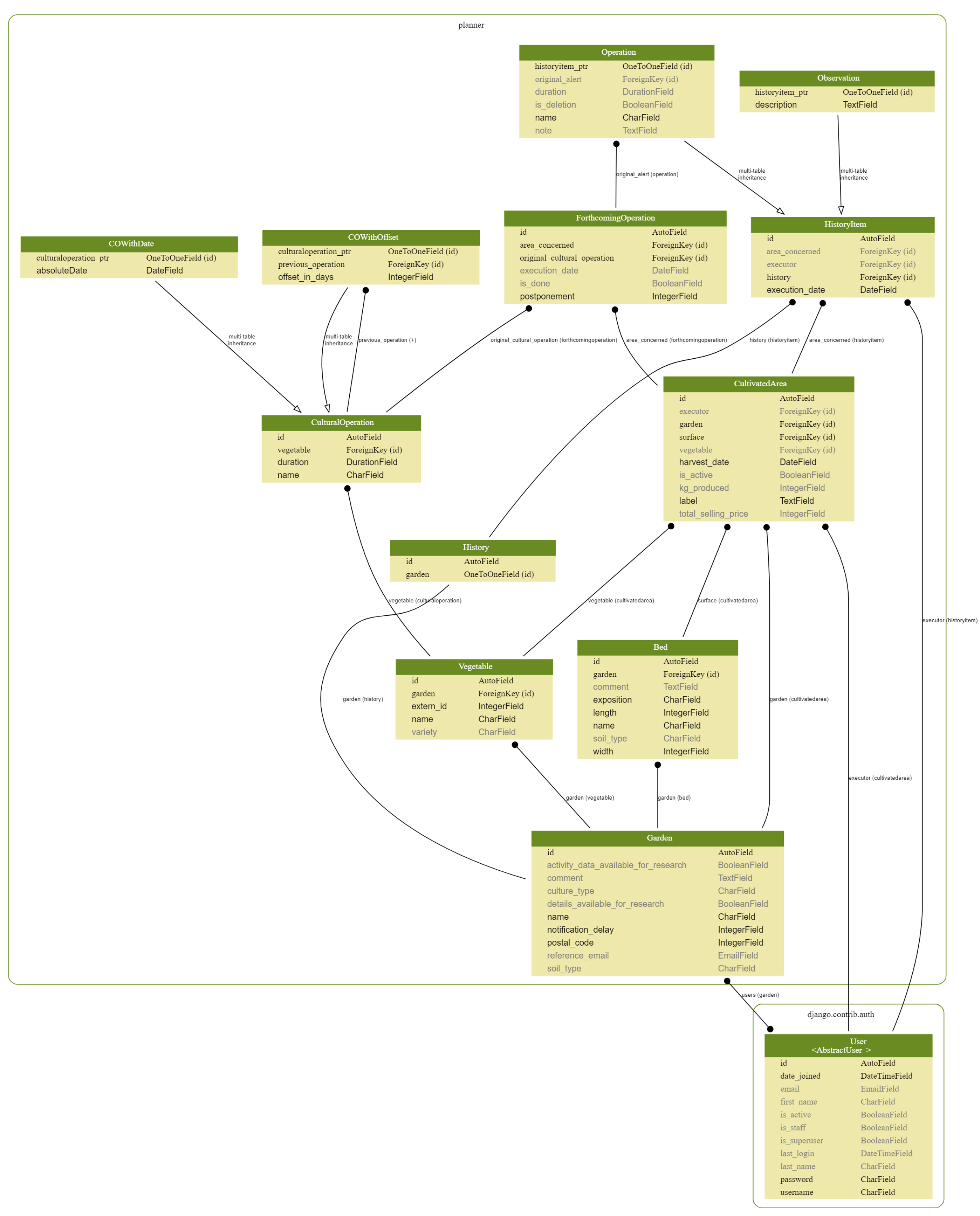


Figure 6.3: Planner application models.

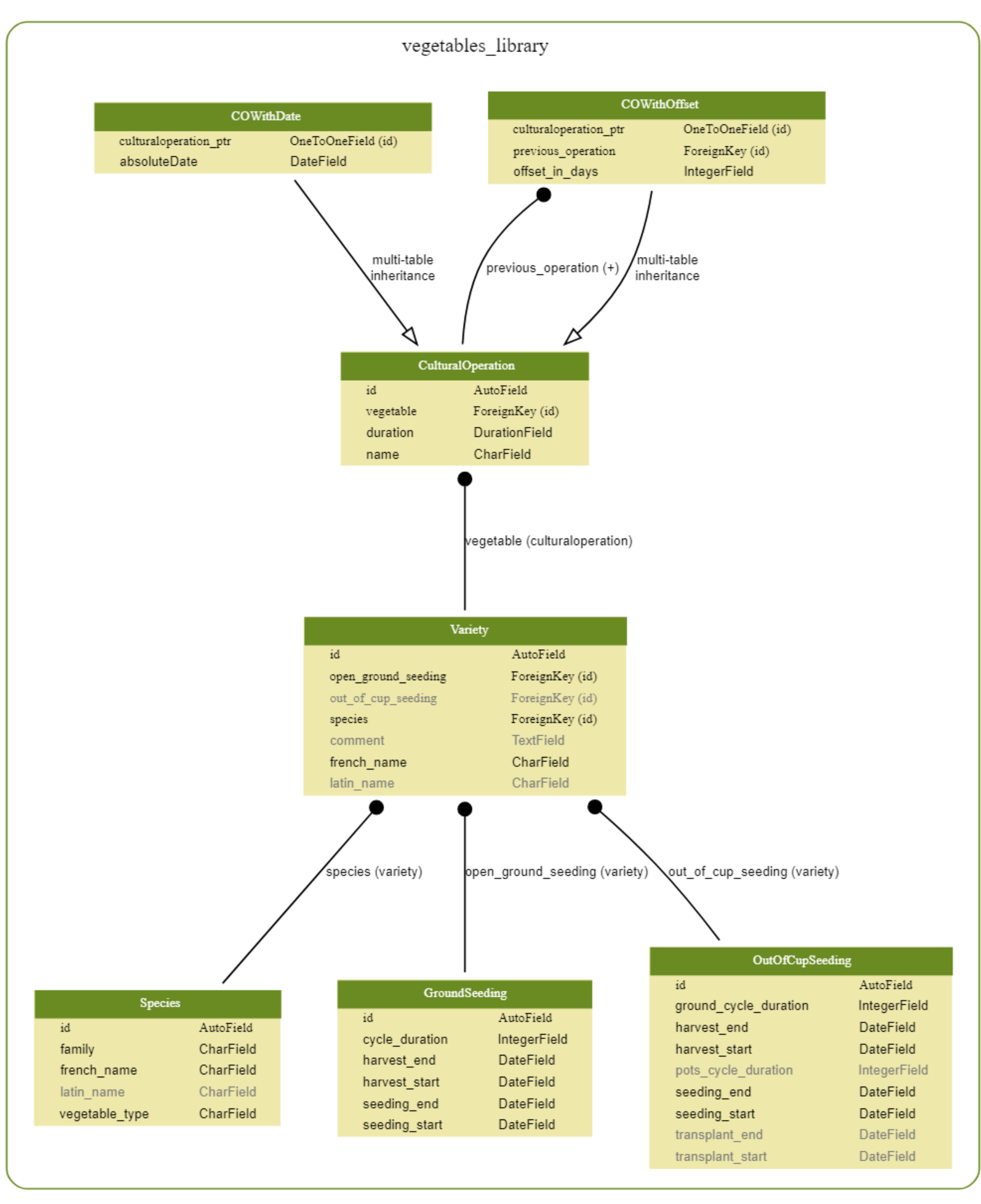


Figure 6.4: Vegetables library models.

Chapter 7

Implementation

In this chapter, we will point out some specificities of our implementation. We will not go through every implementation choice, but we will present some parts that are worth noticing.

7.1 Permissions

In this section we will explain how we solved the problem of the research interface. This interface should only be accessible by researchers, but researchers should be able to register and login as regular users. Hence, we need to give more permissions to some users identified as researchers. Django generates by default three types of permissions for each object created: add, delete and change this object. These permissions are used in the administration interface and can be given to users or group of users. However, for the research interface, this was not what we were looking for since we wanted a permission that regulates access to a set of pages. Instead, we created an empty model (see snippet 7.1) that only has meta informations. Managed set to false implied that there will be no model created in the database, the following line creates a new Django permission. In the administration interface, we have created a research group that has this permission and then we can add users to this group. Every page of the research interface checks that the current user has the permission to access this part of the application. This check is made by a custom decorator used on every view that ensures access to this view is allowed for the current user.

```
2 class ResearchMember(models.Model):
      class Meta:
4         managed = False
          permissions = (
6             ("is_searcher", "Can access research interface"),
          )
```

7.2 Handling two databases

We have seen before that we have separated the library's database from the default database that contains all user information. This choice has implied the development of two intermediate modules. First, a routing module, redirecting all operations on library's objects to the library's database. Second, an import module that allows a gardener to import vegetables from the library to his garden. In this section, we will describe these two modules.

7.2.1 Database Routing

Since we rely upon Django's integrated ORM for all database accesses, we had to tell Django that every model of the library should be stored in the vegetables library database. In order to achieve this, we set up a database router (see snippet 7.1). This router has four main functions that all rely on the meta informations of the models; every Django model has meta information containing, among others, *app_label* which is the name of the Django application they belong to. Therefore, this router tells Django that every read, write, relation and migration concerning a model from the *vegetables_library* application should be perform on the database *db_vegetables_library*.

```
# DB router for the library app


class VegetablesLibraryDBRouter(object):
    """
    A router to control vegetables_library db operations
    """
    def db_for_read(self, model, **hints):
        "Point all operations on vegetables_library models to '
    db_vegetables_library'"
        if model._meta.app_label == 'vegetables_library':
            return 'db_vegetables_library'
        return None

    def db_for_write(self, model, **hints):
        "Point all operations on vegetables_library models to '
    db_vegetables_library'"
        if model._meta.app_label == 'vegetables_library':
            return 'db_vegetables_library'
        return None

    def allow_relation(self, obj1, obj2, **hints):
        "Allow any relation if a model in vegetables_library is involved"
        if obj1._meta.app_label == 'vegetables_library' or \
                obj2._meta.app_label == 'vegetables_library':
            return True
        return None

    def allow_migrate(self, db, app_label, model_name=None, **hints):
        "Make sure the vegetables_library app only appears on the '
    vegetables_library' db"
        if app_label == 'vegetables_library':
            return db == 'db_vegetables_library'
        return db == 'default'
```

Listing 7.1: Database Router

7.2.2 Import module

Since gardeners can import vegetables from the library that are not stored in the same database as their own vegetables, we also had to implement an import module. Indeed, the vegetable's model in the shared library is different from the one in the garden's database. Functions performing this import can be found in file *planner/import_vegetables_helpers.py*. The import copies information from the library and creates a new vegetable object linked to the garden from where the import is done. Thanks to this separation of models, the model from the library can evolve without having any impact on the gardens' models.

7.3 Production

Such a project requires to be tested by real users (i.e. real gardeners). To do so, it is necessary to deploy our application to a real web server and make it accessible to the world. This is commonly called "Deployment to production".

7.3.1 Host machine

We have asked the UCL to provide a machine to be used as a web server. They gave us a virtual machine with Ubuntu 16.04 already installed and a full SSH access to it. This machine has 2 VCPU[1], 2 GO of RAM and a hard drive of 100 GO. Also, the virtual machine has a fixed public IP and the ports 80 and 443 are accessible from the outside of the UCL. The DNS record `lauzeplan.sipr.ucl.ac.be` is configured to point to the IP of our machine (which is `130.104.12.56`).

7.3.2 PostgreSQL cluster

To host our databases, we had two choices: host them on our virtual machine or use the dedicated PostgreSQL cluster of the UCL. We chose the second solution for two reasons. First, it is a good practice to separate the data and the application so that if we make our host crash, it does not impact our data. The second reason is that the PostgreSQL cluster provided by the UCL is configured to perform automatic backups every nights. When dealing with production data (i.e. data from real users), it is very important to make sure they won't be lost if a problem occurs. Thanks to those automatic backups, this is guaranteed.

7.3.3 Apache

We chose Apache[2], a famous, free and open-source, web server to host our application. Since our project is written in Python, we needed a special module to make Apache able to serve our web pages: *mod_wsgi*. This module is able to host any Python WSGI[3] application.

After installing Apache, Python3 and mod_wsgi, the configuration of the website was quite easy. The full configuration file is shown in snippet 7.2.

7.3.4 Git branches

In order to be able to track the status of the production deployment, we created a new branch on our git repository. Our main development branch became *dev* and the *master* branch became our production branch. This means that the code that is deployed to our production server is the one from the *master* branch.

7.3.5 Django configuration

One last thing we had to do: configure Django to run in production mode, with the databases hosted on the dedicated cluster, no more debug messages, etc. To achieve this behavior, there is a built-in way in Django that rely on the environment variable named *DJANGO_SETTINGS_MODULE*. When we specify this variable, Django will load the associated settings. So we created a folder *settings* containing one file per environment in addition to some default settings common to all environments. The common settings contained in *defaults.py* are imported by the others settings *dev.py*, *production.py* and *tests.py*.

[1] A Virtual CPU (VCPU) is a physical CPU that is assigned to a virtual machine.

[2] `https://httpd.apache.org/`

[3] Web Server Gateway Interface (WSGI) is a specification describing how a web server (Apache in our case) can communicate with a web application (Django for us).[15]

```
WSGIPythonPath /home/zmulders/MT-planting_planner

<VirtualHost *:80>
  ServerName lauzeplan.sipr.ucl.ac.be
  Redirect permanent / https://lauzeplan.sipr.ucl.ac.be/
</VirtualHost>


<VirtualHost *:443>
  ServerName lauzeplan.sipr.ucl.ac.be
  DocumentRoot /home/zmulders/MT-planting_planner/planting_planner

  # Specify the path where Apache is authorized to run CGI scripts
  # /webhook/ is the path in the URL
  # the second path is the location of the scripts
  ScriptAlias /webhook/ /home/zmulders/webhook_github_master/
  # Activate CGI for the scripts' folder
  <Directory /home/zmulders/webhook_github_master/>
    Options ExecCGI
    Require all granted
  </Directory>

  # Below is the Django application's configuration
  WSGIScriptAlias / /home/zmulders/MT-planting_planner/planting_planner/wsgi.py
  Alias /static /home/zmulders/MT-planting_planner/planner/static

  <Directory /home/zmulders/MT-planting_planner/planner/static>
    Require all granted
  </Directory>

  <Directory /home/zmulders/MT-planting_planner/planting_planner>
    <Files wsgi.py>
      Require all granted
    </Files>
  </Directory>

  SSLCertificateFile /etc/ssl/lauzeplan_sipr_ucl_ac_be.crt
  SSLCertificateKeyFile /etc/ssl/lauzeplan.sipr.ucl.ac.be.key
  SSLCertificateChainFile /etc/ssl/DigiCertCA.crt
</VirtualHost>
```

Listing 7.2: Apache configuration for lauzeplan.sipr.ucl.ac.be

On the server, we simply had to set the environment variable to the correct value to make Django use the production environment. However, because we use Apache and WSGI, we did not find an easy way to tell Apache to give a custom environment variable to the WSGI process. So we simply updated the file *wsgi.py* to set itself this environment variable. This modification implies that every time our application is run using WSGI, it uses the production settings. This behavior is good for us because when we are developing, we do not use WSGI but a simple Python web server.

7.3.6 Deployment

When we need to deploy a new version of our application to the production server, the first thing we need do is to merge the code to deploy into the *master* branch. Then, we establish a SSH connection to our web server and simply pull the latest changes from the git repository. If some migrations are pending, we need to run them manually. Finally, we need to restart the Apache server.

Automatic deployment

To avoid having to manually connect to our server, we tried to set up an automated way of performing the above operations. We imagined a solution working with the webhooks proposed by Github. A webhook is an URL that Github can call when a specific operation occurs. In our case, we want to call a dedicated URL (`https://lauzeplan.sipr.ucl.ac.be/webhook/webhook_github.perl`) that will trigger a custom script every time a push operation is performed. Configuring the webhook is easy in Github and configuring Apache to run a script when an URL is called is not very hard, using the Common Gateway Interface (CGI) module.

We have reused a small script in Perl [4] found on Github [5]. This script is quite simple and performs one of the operations described above: fetch the latest version of our code. The script is shown in snippet 7.3.

Using a CGI script implies some limitations. For example it is impossible to restart the Apache server from this kind of script. Indeed, it is the Apache server that runs the script. To restart Apache, we first need to stop it, which would stop the CGI script itself.

So our solution for automatic deployment is not fully automated and will not be until we find a solution to avoid restarting Apache.

7.3.7 Uptime robot

Uptime Robot[6] is a free service used to monitor websites. It performs a simple HTTP request every five minutes and sends us an email if the website did not respond correctly. So if our website breaks, we are warned within five minutes and we can inspect the problem and try to solve it.

7.3.8 HTTPS with SSL certificate

In February 2018, Google has announced their plan to mark all HTTP website as not secured[7]. For this reason, and also because we want to protect the data of our users, we secured our website with an SSL certificate.

Since the UCL provided us the web server and the domain name, we also asked them to give us the needed SSL certificate. Once we had this certificate and the associated private key file, it

[4] Perl is a scripting, interpreted, language: `https://www.perl.org/`

[5] `https://gist.github.com/mugifly/5087897/`

[6] `https://uptimerobot.com`

[7] `https://security.googleblog.com/2018/02/a-secure-web-is-here-to-stay.html`

```perl
#!/usr/bin/env perl
# GitHub Webhook receiver & updater cgi (for Hosting-web-server)
# https://gist.github.com/mugifly/5087897/
use strict;
use warnings;
use utf8;
use lib 'lib';
use CGI;
use JSON::PP;

our $GIT_PATH = '/usr/bin/git';
our $TARGET_DIR = '/home/zmulders/MT-planting_planner/';
our $TARGET_REPONAME = 'origin';
our $TARGET_BRANCH = 'master';

our $q = new CGI;

load_payload();
git_pull();

print "Content-type: text/html\n\n";
print "OK";
exit;

sub load_payload {
        unless(defined($q->param('payload'))){ error('Invalid parameter'); }
        my $payload = JSON::PP::decode_json($q->param('payload'));
        if ($TARGET_BRANCH ne "" && $payload->{ref} ne 'refs/heads/'.
    $TARGET_BRANCH){
                error('Not target refs');
        }
}

sub git_pull {
        chdir($TARGET_DIR);
        `$GIT_PATH pull $TARGET_REPONAME $TARGET_BRANCH`;
}

sub error {
        my $msg = shift;
        print "Content-type: text/html\n\n";
        print "Error:".$msg;
        exit;
}
```

Listing 7.3: Perl script for automatic deployment

was easy to configure Apache to redirect all the traffic to HTTPS and to enable the usage of SSL with the ssl module of Apache.

The certificate we use is valid until the 20th of May 2020. Before this date, it will be needed to renew it or the application will be marked as unsecured by most web browsers.

7.3.9 Progressive Web App

Progressive Web App (PWA)[8] is an emerging technology (and philosophy) developed by Google and adopted by most of the major actors of the web. It allows a website to look and behave like a native application[9], meaning that a PWA have the following abilities.

- Can be installed on the homescreen of the device;
- Can run offline;
- Can receive and display push notifications;
- Can adapt to all size of screens;
- Can run in fullscreen.

At the time we are writing this master thesis, PWA is not yet widely supported, but it is well on track. Google Chrome, Firefox and Opera support it well on Windows, Linux and Android (we are not sure about macOS and iOS). Microsoft announced[10] that PWA will be available for installation from the Microsoft Store. Apple started to implement Service Worker (described below) on Safari a few weeks ago[11] on their release 46. However, on iOS, PWA is not completely functional yet.

But even if Apple does not fully support Progressive Web App, it is still worthy to set it up. Because building a PWA is about ensuring that the user will have the best experience possible by making our application behave well on different sizes of screens and load as fast as possible.

Manifest The first thing that a PWA needs is a JSON file called *Manifest*[12] telling the browser basic information about our application like its name or its icon.

Service worker The core principle of Progressive Web App is the service worker[13]: a JavaScript script that is installed by the browser and runs in background. The service worker can cache any kinds of resources (style sheets, HTML pages, JavaScript files, images, etc.). It intercepts every network request and is able to serve a cached version of the requested resource.

Once installed, the service worker can make our website starts instantaneously by serving a cached version of our home page for example. If a device has a poor connection or no connection at all, our application can still run thanks to the service worker: it can again serve a cached version of your page or it can display a custom page to indicate to the user that a connection is required to continue.

[8] `https://developers.google.com/web/progressive-web-apps`

[9] A native application is an application that is developed for a targeted Operating System and is installed from a dedicated store.

[10] `https://blogs.windows.com/msedgedev/2018/02/06/welcoming-progressive-web-apps-edge-windows-10/#0eVsoxrHYlso6vcS.97`

[11] `https://developer.apple.com/safari/technology-preview/release-notes/#r46`

[12] `https://developers.google.com/web/fundamentals/web-app-manifest`

[13] `https://developers.google.com/web/fundamentals/primers/service-workers/`

Progressive Web App checklist With the Manifest and a Service Worker, here are the others requirements a website must meet to be called a Progressive Web App[14].

- Use HTTPS exclusively;
- Responsive content;
- All URLs load even offline;
- First load time must be under ten seconds.

Advantages Here is a list of some important advantages PWA brings. The first one is an advantage over natives applications. The others points are advantages over classical websites.

- No installation from a store (Play Store, App Store, etc.) needed. Just access a website and install it as an application;
- Work offline;
- Load fast;
- Home screen installation;
- Can receive Push notifications.

Progressive Web App and Django At the end of the implementation of our application, we decided to set up the basic blocks needed to build a Progressive Web App. We took this decision because our website will potentially be used by gardeners from the field, with their phones. So it is a good point if they can install our website as an application and have a shortcut on their home screen. To do this, we used a Django app named *django-progressive-web-app*[15].

Our manifest is generated by this app based on some pieces of information provided in our settings.

For our Service Worker, we made the choice to show a warning page when no connection is available. It was the most logical choice for us because our application cannot really work without an internet access since Django is a pure server-side framework. All our data are stored in a distant database and the web pages are generated by Django. So an offline version of our application is not really a good idea. It would lead to outdated data presented to the user and we would need to implement a complicated offline management system to allow the user to encode new vegetables, operations, ... and publish them to the server once our app goes back online.

The complete Service Worker code is shown on snippet 7.4. It simply stores a cached version of the page `https://lauzeplan.sipr.ucl.ac.be/offline`, shown in Figure 7.1 and serves it whenever the network fails.

We configured our website to run over HTTPS as explained in Section 7.3.8. We tested with Google Chrome on our laptop and our Android phone that the application can indeed be installed on our home screen. This operation is automatically proposed by Google Chrome under some conditions[16]:

- The user has interacted with the application for at least thirty seconds;
- The application is not already installed;
- A service worker is active;

[14] `https://developers.google.com/web/progressive-web-apps/checklist`
[15] `https://github.com/svvitale/django-progressive-web-app`
[16] `https://developers.google.com/web/fundamentals/app-install-banners`

```
var cacheName = 'lauzeplan-offline';
//Install stage sets up the offline page in the cahche and opens a new cache
self.addEventListener('install', function(event) {
  var offlinePage = new Request('offline');
  event.waitUntil(
  fetch(offlinePage).then(function(response) {
    return caches.open(cacheName).then(function(cache) {
      console.log('Cached offline page during Install'+ response.url);
      return cache.put(offlinePage, response);
    });
  }));
});

//If any fetch fails, it will show the offline page.
self.addEventListener('fetch', function(event) {
  event.respondWith(
    fetch(event.request).catch(function(error) {
        console.error( 'Network request Failed. Serving offline page ');
        return caches.open(cacheName).then(function(cache) {
          return cache.match('offline');
      });
    }));
});
```

Listing 7.4: Basic Service Worker script showing the page "/offline" when no connection

Figure 7.1: Offline page shown when no network is available.

Figure 7.2: Splashscreen shown when application is loaded from the home screen.

- The checklist presented above is respected.

Once installed, the icon of our application appears on the home screen of our device. When launching the app from there, a splashscreen (shown in Figure 7.2) first appears, quickly replaced by the content of the application itself. The website is now in full screen mode and looks like any other native app.

Chapter 8

Validation

In this chapter we will explain how we validated our system. First, we will talk about the technical validation, that is, making sure our system is stable, maintainable and robust. Tests are relative to Django's application and can be found in *planner/tests* and *research/tests*. Then, we will talk about the usefulness of our system by presenting it to vegetable growers and researchers.

8.1 Unit tests

Unit tests are meant to ensure the correctness of a small part of code, usually a function. So the first thing we have done is some unit tests for our helpers functions. We have not tested our models and our requests to the databases since it is pure Django.

8.2 Functional testing

A very important part to test is the views. Django has a Client class that can act as client instance for tests. We have used this client to test all our views. Django's client can perform get and post requests, so we were able to test all our forms.

We show the simplest test possible done on each view on snippet 8.1. This very simple test helped us ensure that every page is reachable. However, in addition to this simple access test, we also implemented tests on post requests. We have tested all our forms and whether they are creating objects properly and redirecting to the right page.

```
1   def test_index_view(self):
        self.client.login(username=self.username, password=self.password)
3       response = self.client.get('/{}/vegetables'.format(self.garden.id))
        self.assertEqual(response.status_code, 200)
```

Listing 8.1: Example of a simple test done on each view

All our tests together, unit and functional tests, we managed to reach 83% of code coverage as shown in Figure 8.1. This result can be found on CodeClimate. As said before, we have done continuous integration testing; Travis launches our tests at every push to the repository and sends the coverage report to CodeClimate. If one of our tests failed, Travis' build will fail and we directly receive and email.

8.3 Manual tests

Manual tests consist of testing the application by hand, that is, the developer goes to the application and makes sure everything is working as expected. For example, a typical testing scenario is to register a user, connect to the application, create a garden, a vegetable, and add a planting. From there, we can check that the others features such as validating an alert with some

Codebase summary

MAINTAINABILITY

A 2 days

TEST COVERAGE

B 83%

Figure 8.1: Codeclimate summary.

information and making sure it appears in the history. Something that was way easier to test by hand was the printing and exporting features. Testing that when clicking on the printing button we get a pdf document (and a csv document for the export) has been tested in the functional tests. However, making sure the content of the printed or exported file is correct has been done by hand. Another feature tested only by hand is the QR code generation and making sure the QR code are redirecting to the correct URLs. Hence, for some specific tests and for end-to-end testing, we have done manual tests.

8.4 Human validation by gardeners

To validate our product, we went back towards gardeners to show them what we have done. Some of them tried it and gave us feedback. We will go through their comments in this section.

Emile et Camille

5590 Les Basses-Serinchamps
They grow vegetables on about two hectares since 5 years. They promote organic and family farming.

We went to their place to show them the application and explain the different features. They were really enthusiastic. They told us that sometimes they wake up in the middle of the night, remembering that they forgot to seed something. This application could help them sleep tight and stop feeling guilty about forgetting something. It would give them a clear idea of what they have to do and help them planning and advance their work. As they are two workers, they think it would also help them coordinate their work and be aware of what the other has done.

However, they are old school gardeners and don't use smartphone or tablets. They prefer a paper-based planner. So they could benefit from the printing feature.

Prisca Sallets

She is a technical advisor in market gardening at Biowallonie.

We did not meet Prisca, she tried the application by her own. It was a good test to ensure the user friendliness of our product. Unfortunately, she did not understand the structure of the software. However, she gave us a feedback of what she did not understand and how she would use the software.

First of all, we should definitely add a tutorial or a home page explaining how to use the application and what do the different concepts mean and represent (gardening's jargon is not always the same from one gardener to the other). For example, she got a little lost at the first page when we ask her to create a *Jardin*, because in her jargon, a *Jardin* is a set of beds and a market garden has a set of *Jardin*. From there, it seems important to add the concept of *Jardin* and hence to add a gradation for cultivated surfaces (a set of beds constitute a *Jardin*). Moreover, it often happens that she seeds the same thing on a whole *Jardin* (i.e. several beds). She does not want to encode this planting for each bed of the *Jardin*, she would prefer to be able to add a planting to a bed, several beds or a *Jardin*.

She usually knows what has to be done on her crops every day. For her, the most interesting part is the encoding of spent time on each crop. But once again, she wants to be able to say that she has weeded a bunch of beds in a certain amount of time and then this amount is split between the beds concerned.

Annick Noiset

1390 Grez-Doiceau

She is at her third season of vegetables production on about 65 ares. She is growing around 200 different varieties.

She used to plan her season in January in an excel sheet with a set of tasks to be done for each week. At the start of the season, she prints her planning and then try to stick to it. However, sometimes, she does not manage to do everything she planned on a week and since her planning is on paper, she quickly gets lost on what she has done and what remains to be done. She is doing organic market gardening and therefore she has to write down everything she does. She does it in a notebook and then she has to transcribe it on her computer. She is eager to use technology to improve her organization. She really believes that she could save an important amount of time by using a tool such as ours.

She is using a smartphone and she is interested to be kept informed of the evolution of this project. She could not use it now since the season already started, but she thinks that she could use it during winter to plan her season.

Nonetheless she noticed some flaws in our product: her bed planning evolve during the season. Since several crops follow each other on a same bed, sometimes a bed is not ready (the soil still needs to be worked or another crop is still in place) to welcome the planned crop and hence she has to plant it in a free bed. Second, she thinks it is really hard to estimate how long it takes per square meter to do each task.

8.5 Human validation by researchers

This tool is going to be used as part of the gardening project of Lauzelle farm, the farm taken in charge by UCLouvain. From next year, the cultivation operations made on the farm's field will be encoded in our tool. After that, data encoded from the application could be used within the scope of research.

Another interest for researchers is the access to the vegetables library. As we have discussed before, such a library is hard to get since it is valuable for seed sellers. Hence, having such a library easily and freely accessible can be useful for some researchers.

8.6 Conclusion

To sum up, we managed to get a code that seems maintainable and adequately tested. This is very important to ensure the project can persist and be reused and maintained by other developers. On a more practical point of view, from our interviews with gardeners and researchers, we have learned different things. First of all, this tool arouses interest from gardeners. However, as every gardener has his way of working and planning it is not always easy to have a generic tool that is understandable by all and that meets everyone's needs. To conclude, this kind of tool has a bright future but is still at its early stages.

Chapter 9

Conclusion and future work

In this chapter we will go through what we have done compared to our initial requirements. We will also go through a set of additional features that could be worth considering if the application is extended. This chapter serves as a conclusion of our work in the context of our master thesis.

9.1 Objectives

Looking back at our initial requirements and comparing them with what we produced, we can affirm that we partly met our initial objectives. As a reminder, the initial objective of this project was to provide a software tool for gardeners that often have troubles planning their crops. Furthermore, researchers at the UCL in the bioscience engineering Faculty are currently doing research on market gardening and the software had to take this aspect into account since they are eager to gather data from gardeners. As a matter of fact, we managed to produce a usable software application that could help gardeners in their day to day planning. This same application can also be used by researchers to gather data about gardening. Nevertheless,we did not succeed in developing some features, for example we have not had the time and the information required to implement a planning tool checking a set of constraints.

A secondary objective was to mix technology and market gardening that are two fields that used to be pulled apart. We completely achieved this goal as we received really positive feedbacks from gardeners, interested in our work.

9.2 Future work

We hope that our project will be maintained and extended by the bioscience engineering Faculty. Indeed, we have a lot of ideas to extend our software system and we will present some of them in this section.

9.2.1 Planning tool

We could implement a planning tool as initially discussed in the mindmap. This would imply a research part, defining which constraints should be applied during spatial planning. Indeed duration of crop rotation and intercropping are two fields of research with not a lot of proven data and gardeners often have their own theories. Moreover, crop rotation can depend on soil type. Such a planning tool is of great interest for research as the model concerned is inherently complex. We could also imagine a highly generic constraints tool where the gardener could encode himself the constraints he is using in his cropping plan.

9.2.2 Cost price estimation

We implemented a harvest part where gardeners can enter details about harvest. However, we could have go further in this part and we think it is really interesting to have data about cost price of vegetables. A possible extension of the software would be to estimate the cost price of vegetables depending on the work hours spent on the crops and the selling price and then, make an estimation of the incomes of an entire season based on the initial planning.

9.2.3 Bed visualisation

We could improve the beds' visualisation part by adding a drawing tool on a map. The gardener can locate precisely on a map where his beds are located.

9.2.4 Pictures

We could allow users to add pictures for the vegetables, but also for gardens. As there are hundreds of varieties for some species, it could be useful to have a picture linked to each variety to have an idea of its shape, colour,... Gardeners could also choose to add a picture of their field or current crops. However, populating a database of vegetables already takes time, but adding pictures could double this time. For this reason, we think it could be interesting to have a partnership with a seed seller.

9.2.5 Partnership

As said before, populating a database is time-consuming and these data have already been encoded by several seed sellers. A partnership with a seed seller could be a good idea to have access to such a database. In return, seed seller could add advertisement for his seeds for example. While contacting gardeners to show them our product, we have been contacted back by someone developing a similar application. He is possibly interested by a partnership with the university. He has been working on his product for three years, but he realised that he cannot do everything on his own because it takes too much time. Because he has put lots of effort in it, he would like his software product to be used and eventually to get money from it.

9.3 Conclusion

We have built a software system usable by gardeners, however we also have discovered the complexity hidden behind gardening. It can be really challenging to build a model that would be generic and specific enough to capture every crop's characteristic. Designing a software that would be simple, intuitive and still complete is a tough task. This thesis has proved the relevance of interest in this area and the challenges encountered when mixing software engineering, research in bioscience engineering and contact with gardeners which are not always technology-oriented people. In software engineering, this thesis was also a good place to put into practice what we have learned during our studies. We have implemented a web application from A to Z, including the meetings with clients and the research of innovative solutions.

Bibliography

[1] Manifesto for Agile Software Development. http://agilemanifesto.org/, 2001. [Consulted 22-March-2018].

[2] Continuous integration. https://www.thoughtworks.com/continuous-integration. [Consulted 26-March-2018].

[3] Cyclomatic Complexity. https://docs.codeclimate.com/docs/cyclomatic-complexity. [Consulted 28-February-2018].

[4] Antoinette Dumont. *Analyse systémique des conditions de travail et d'emploi dans la production de légumes pour le marché du frais en Région wallonne (Belgique), dans une perspective de transition agroécologique.* PhD thesis, Université Catholique de Louvain, 2017.

[5] Jean-Martin Fortier. *Le jardinier-maraîcher : manuel d'agriculture biologique sur petite surface.* Ecosociete Eds, 2016.

[6] LEA. https://www.lea-agri.com/. [Consulted 10-January-2018].

[7] Mes petits légumes. http://mespetitslegumes.com/. [Consulted 15-August-2017].

[8] Mireille Navarrete and Marianne Le Bail. Saladplan: a model of the decision-making process in lettuce and endive cropping. *Agronomy for sustainable development*, 27(3):209–221, 2007.

[9] Horizon permaculture. Planification des cultures : méthodologie. https://fermesdavenir.org/fermes-davenir/outils/planification-cultures-methodologie. [Consulted 15-November-2017].

[10] Software framework - Wikipedia. https://en.wikipedia.org/wiki/Software_framework. [Consulted 08-April-2018].

[11] Tend. https://www.tend.ag/, 2016-2018. [Consulted 26-March-2018].

[12] Jean-Paul Thorez. *Le petit guide du jardinage biologique : potager et verger.* Terre Vivante, 1985.

[13] Jake VanderPlas. Why Python is Slow: Looking Under the Hood. https://jakevdp.github.io/blog/2014/05/09/why-python-is-slow/, 2014. [Consulted 10-January-2018].

[14] Web framework - Wikipedia. https://en.wikipedia.org/wiki/Web_framework. [Consulted 08-April-2018].

[15] What is WSGI? WSGI.org. http://wsgi.readthedocs.io/en/latest/what.html. [Consulted 29-Mars-2018].

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve www.uclouvain.be/epl