UBC Sustainable Farming Tool

LITEFARM

Prepared by: Craig Yu, UBC Sustainability Scholar, 2018
Prepared for: Dr. Zia Mehrabi, Research Associate, The UBC Policy School, Institute for Environment Resources and Sustainability, and UBC Farm.
August, 2018

Acknowledgements

The author would like to thank Karen Taylor and Dr. Zia Mehrabi for their support and feedback throughout this project.

Cover photo courtesy of UBC Communications and Marketing

Contents

Executive Summary

LiteFarm is a multifunctional web app that aims to help farmers make day-to-day decisions, and encourage them to farm more sustainably. The feature set in LiteFarm was created based on a farmer participatory design process, including iterative user experience and user interface testing. The project is ongoing but my work during the UBC Sustainability Scholars program fell within the key phase of implementing a high fidelity prototype of the app.

Currently we are developing a Minimal Variable Product (or MVP), which "is a product with just enough features to satisfy early customers, and to provide feedback for future product development" (*Wikipedia, 2018*). The MVP development was initiated in 2 phases: a back-end phase, and a front-end phase. As a UBC Sustainability Scholar, I worked as part of a developer team to (1) create a technology plan and design operating procedures, (2) map a detailed back end architecture to a normalized database design, (3) populate the back-end with a range of default data for data driven user experience, (4) write API for posting and getting information from the database, and (5) implement the user sign up, profile, mapping, logging features of the front-end. Details of my work during my time as a Sustainability Scholar with the LiteFarm application project are further explained in this document.

Introduction

Farming in the 21st century requires complex logistics. This is exacerbated by the need to produce food in the most environmentally and socially responsible way possible. With the ever-evolving technologies, in rich nations farm sizes are getting larger, and fewer people are required to work on an acre of field each year *(Roser, 2018).* However in poorer nations, the size of farms is only getting smaller: these are often highly diverse and require large amounts of labor. In both cases data collection and reporting responsibilities are increasingly placed on farmers. Juggling different management software tools can be daunting and time consuming, and in some cases – particularly for diversified and complex farming system technologies, easy to use solutions do not exist. Our project aims to develop an open-source multifunctional web application that helps farmers to manage their farms more efficiently, while promoting sustainable farming. It was coined LiteFarm because our primary target users are diversified farm owners and workers, who have trouble with complex cropping problems, and technology adoption. The brand represents a tool that is easy, light, and something that users won't even notice they are carrying in their pocket.

LiteFarm has 8 major features, which perform the following functions:,

1. Profile: Sign up, access control, worker management, information settings.
2. Fields: Provide access to satellite imagery and a mapping tool to help farmers design their field plans, and manage rotations.
3. Finances: ales, Expenses, and on the fly calculation of profit and loss by crop.
4. Logs: Fertilizer, pesticide, irrigation, scouting, and a variety of other activity logs for record keeping
5. Notifications: Smart notification to farmers such as pest outbreak activities, and extreme weather.
6. Shifts: Easy labor tracking.
7. To-do lists: Create to-dos and assign them to farm staff
8. Reports: End of season reporting of socio-ecological outcomes, including price trends, profit and loss, water use efficiency, nitrogen use efficiency, biodiversity on farm, soil health, labor wellbeing, and food supply.

Features such as logs and finance help farmers to understand their farms' operation with ease. Farmers can track their usage of fertilizers and pesticides to make farming more sustainable. The log records can also help farmers to get their organic produce certified. LiteFarm as an open-source free web app lowers the technological barrier to entry, encourages more farmers to use our app thus induce more sustainable farming. The development of this web app is the core of this project, so this report is mainly focused on the developing process.

General background

For a typical web app to function there are two broad components needed to be developed and tested. They are the 'back-end' and the 'front-end'. The back-end belongs to the data-access layer of an application. It is an aggregation of services on the server-side that distributes necessary data to clients. The front-end belongs to the presentation layer, or the client side of the application. It gathers data using back-end services and presents them to users in a useable, interactive way. In this document we will examine two components of their design: the technology and their implementation in our project. Figure 1 shows a rough blueprint of how front and back-end are linked.

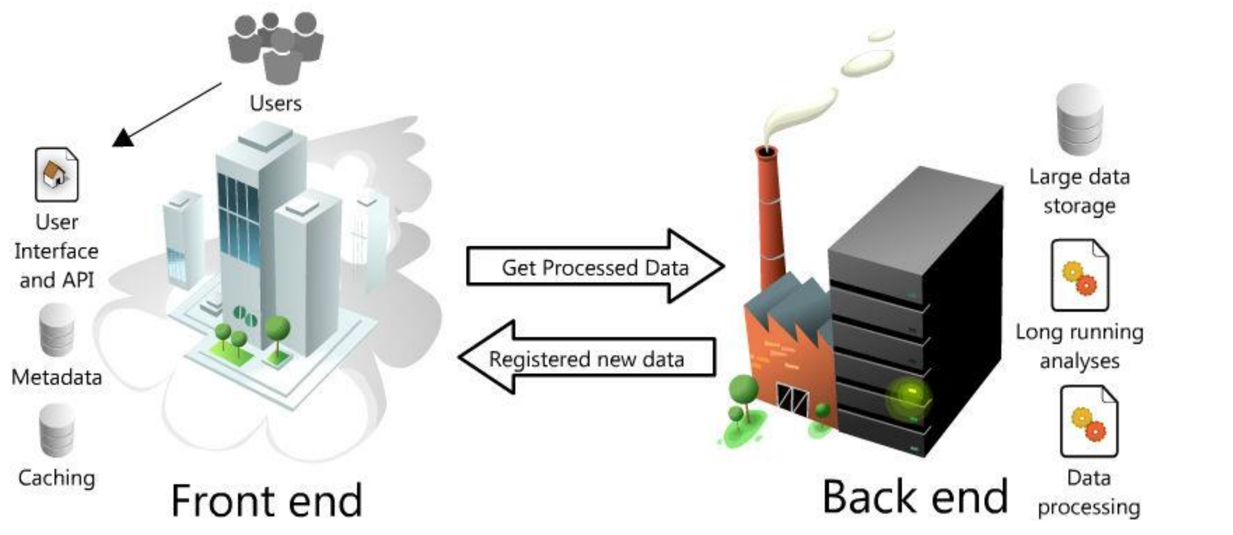


Figure 1 **Credit:** The Map of Life

The main communication method between the front and back-end is through the use of API, namely a REST API. API stands for Application Programming Interface: a set of predefined calls that can be made by a client to a server. REST stands for Representational State Transfer, which is an API architectural style. There are 4 commonly used methods in a REST service.

METHOD	USAGE
GET	Read
PUT	Update/Replace
POST	Create
DELETE	Delete

For example, when you go to a website to check foreign exchange rate, the website will make a GET call to a server, which will send back an object that contains all the necessary information. You can try this (https://exchangeratesapi.io/api/latest) in your browser to see the object that gets sent back!

In the next two sections I will go on to explain the work that I managed to accomplish on both the front and back end components of the application, and explain the technologies and way they were implemented. Two cross-cutting technologies that were used across both the front and back end are listed below for reference.

Node.js:

A JavaScript run-time environment that executes scripts written in JavaScript on the server-side.

npm:

npm stands for node package manager, it lets developers to install/uninstall packages made available by other developers without worrying about dependencies.

If the web app is a phone, then the operating system on that phone is Node.js, and the App Store/Play Store is npm. They are the backbone of both the front and back-end.

Development Approach

There are many development philosophies of how to implement a web app. We chose to develop the back-end first because in our opinion it is the most efficient way, given the size of our team, the time constraint, and the detailed mapping of data and algorithms that are required for the front end to run. The idea, was that, once the back-end development was complete, we can make actual API calls to our server to retrieve or modify data instead of mocking the data for the front-end to use. The process helped us to understand the full scope of the application.

Before we started coding, we worked as a team to make a plan of action and prepared a range of planning documents to help provide guidance during the development process. The idea is that this would provide us with a structured workflow, and also help future developers to quickly integrating into our team.

The documents we worked on included:

DOCUMENT	PURPOSE
Coding standard	A set of rules to enforce a coding standard to make code cleaner and more coherent. This is essential for a group project as each developer has their own coding style preferences. The standard is enforced by using ESLint.
Technology plan	Provides a list of technology that will be used for development. Including comparisons of different technologies to inform the right technology choices before starting the project.
Operation procedures	Establish procedure rules for the development. Such as meeting frequencies and branch merging procedure.

Development Process

Phase I: Back-end

Upon starting the project, the required features, a high fidelity prototype of the user interface, and a detailed information architecture, were already in place. This enabled us to design the database and define necessary API upfront. Before we designed the database, we had to decide on which type of database we will use. The choices are a relational database(SQL) or non-relational database(NoSQL). This article provides more information on the differences between those two.

PostgreSQL

Based on the web app's features, we chose to go with a relational database for its scalability and reliability. PostgreSQL is a relational database management system, it allows us to manage the database with Structured Query Language.

With an information architecture diagram of the back-end, using relational database design principles, our team created a database schema that would allow us to implement the database.

Figure 2 shows a snippet of an early iteration of the database schema representing the crop table, and each attribute is a column name of the table.

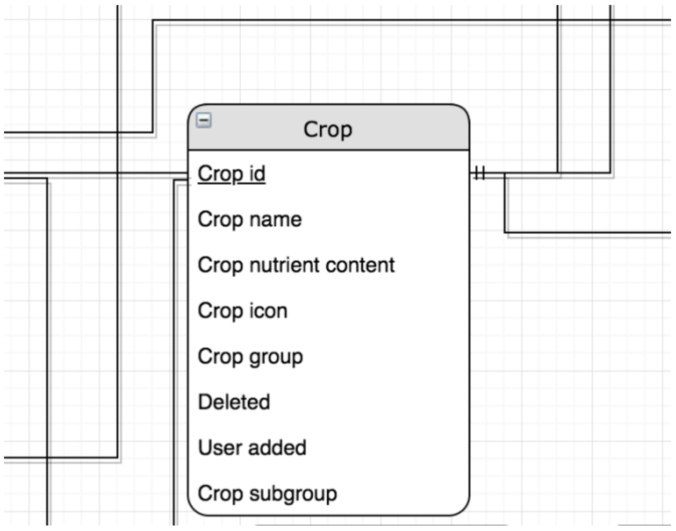


Figure 2

In order to achieve the data driven features of the web app, we needed to collate a range of data on which the back-end algorithms were to run. The challenge for this was that no crop related online data sources are available for us to use right away. The data being either in the form of web page, PDF, or raster file. By using a combination of Chrome plugins and Python package called Beautiful Soup, I was able to scrape the data from web pages and store them as CSVs. The scraped data needed more 'wrangling' to be usable for the database. For example, the crop table in our database needed crop groups, crop names, crop nutrient contents, crop rooting depths, and crop coefficients. All three of those data came from different sources, but they needed to be integrated into one table. Another challenge was that a common crop name can be associated with many scientific names, and vice versa. That made the merging of the data much more difficult. In the end this was done by using multiple Python scripts and manual check for data integrity.

After we created the database from the schema, and populated it with the data, it was time to implement the API for our app. Our API consists multiple endpoints. The endpoints I implemented and tested for the team (who also worked on a range of others not shown) are shown in the table below.

REQUEST METHOD	ENDPOINT	PURPOSE
GET	/user/:id	Get user by user id
GET	/user/farm/:farm_id	Get all users under the farm id
POST	/user	Create a user
PUT	/user/:id	Update a user by user id
DELETE	/user/:id	Delete a user by user id
GET	/crop/:id	Get a crop's info by crop id
GET	/crop/farm/:farm_id	Get a list of crops by farm id, the list includes a default crops and custom crops added by the users under the farm id
PUT	/crop/:id	Update a crop by crop id
POST	/crop	Add a default crop
DELETE	/crop/:id	Delete a crop by crop id
GET	/farm/:id	Get farm info by farm id
POST	/farm	Create a farm
PUT	/farm/:id	Update a farm by farm id
DELETE	/farm/:id	Delete a farm by farm id
GET	/field/farm/:farm_id	Get a list of fields under a farm id

POST	/field	Create a field by field id
PUT	/field/:id	Update a field by field id
DELETE	/field/:id	Delete a field by field id
GET	/farm_crop/farm/:farm_id	Get a list of crops planted under a farm id
POST	/farm_crop	Create a crop being planted in a farm
PUT	/farm_crop/:id	Update the crop being planted in a farm by farm crop id
DELETE	/farm_crop/:id	Delete the crop being planted in a farm by farm crop id
GET	/shift/:id	Get a shift's info by shift id
POST	/shift	Create a shift
PUT	/shift/:id	Update a shift by shift id
DELETE	/shift/:id	Delete a shift
GET	/task_type	Get all task types
GET	/task_type/:id	Get a task type by task type id
POST	/task_type	Create a task type
DELETE	/task_type/:id	Delete a task type by task id

A detailed example

In this section, we will look at one example of how we implemented the back-end system, so that it can send back a list of crops upon a client's request. The same idea applies to the endpoints listed above, and others included in the application.

When a client initiates a request of the crop list to the server in the form of a GET call, the server should parse the call to know what the client is requesting and run the scripts that would query the database and send the list back. This is achieved by using Express. Express is a Node.js framework that provides features needed to build our API.

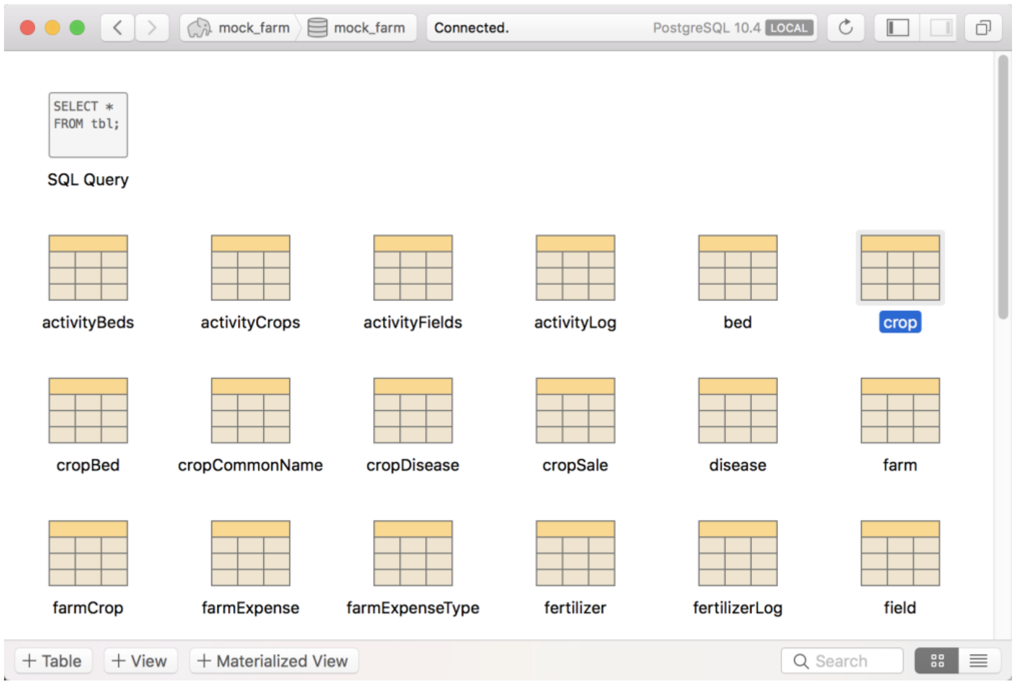
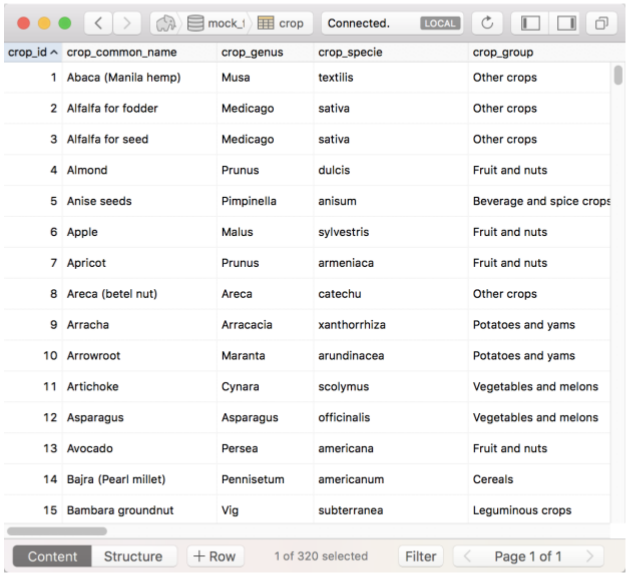


Figure 2

Figure 3 shows a snapshot of the database. The crop list is stored in the highlighted crop table. Inside the table contains all the necessary information of all crops as shown in figure 4.



crop_id	crop_common_name	crop_genus	crop_specie	crop_group
1	Abaca (Manila hemp)	Musa	textilis	Other crops
2	Alfalfa for fodder	Medicago	sativa	Other crops
3	Alfalfa for seed	Medicago	sativa	Other crops
4	Almond	Prunus	dulcis	Fruit and nuts
5	Anise seeds	Pimpinella	anisum	Beverage and spice crops
6	Apple	Malus	sylvestris	Fruit and nuts
7	Apricot	Prunus	armeniaca	Fruit and nuts
8	Areca (betel nut)	Areca	catechu	Other crops
9	Arracha	Arracacia	xanthorrhiza	Potatoes and yams
10	Arrowroot	Maranta	arundinacea	Potatoes and yams
11	Artichoke	Cynara	scolymus	Vegetables and melons
12	Asparagus	Asparagus	officinalis	Vegetables and melons
13	Avocado	Persea	americana	Fruit and nuts
14	Bajra (Pearl millet)	Pennisetum	americanum	Cereals
15	Bambara groundnut	Vig	subterranea	Leguminous crops

Figure 4

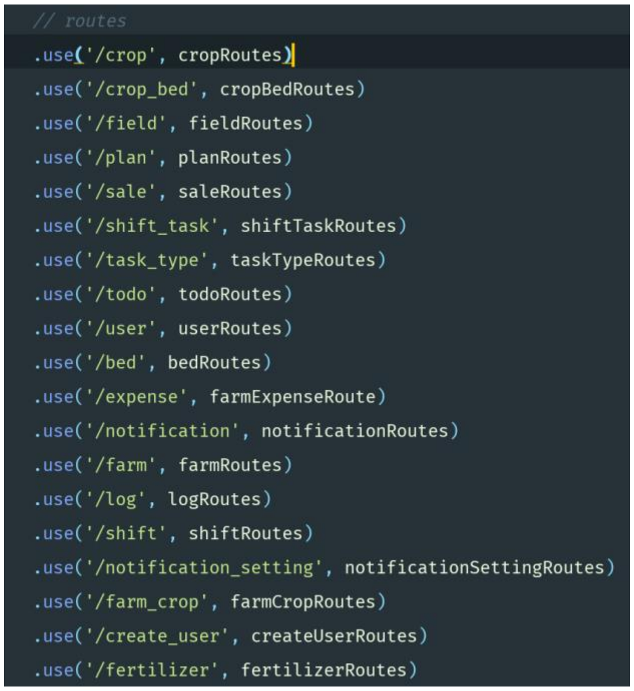
Step 1: accept a call

The endpoints that are currently registered in our server can be seen in figure 5.

When a user make a request of:

```
GET
http://localhost:5000/crop
```

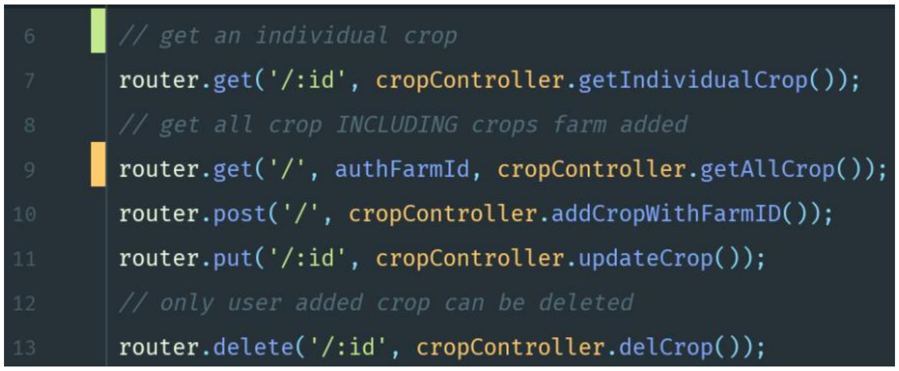
The server takes the `/crop` part of the url to know which script to run, in this case it will run the `cropRoutes` script.



```
// routes
.use('/crop', cropRoutes)
.use('/crop_bed', cropBedRoutes)
.use('/field', fieldRoutes)
.use('/plan', planRoutes)
.use('/sale', saleRoutes)
.use('/shift_task', shiftTaskRoutes)
.use('/task_type', taskTypeRoutes)
.use('/todo', todoRoutes)
.use('/user', userRoutes)
.use('/bed', bedRoutes)
.use('/expense', farmExpenseRoute)
.use('/notification', notificationRoutes)
.use('/farm', farmRoutes)
.use('/log', logRoutes)
.use('/shift', shiftRoutes)
.use('/notification_setting', notificationSettingRoutes)
.use('/farm_crop', farmCropRoutes)
.use('/create_user', createUserRoutes)
.use('/fertilizer', fertilizerRoutes)
```

Figure 5

Step 2: execute a call



```
6   // get an individual crop
7   router.get('/:id', cropController.getIndividualCrop());
8   // get all crop INCLUDING crops farm added
9   router.get('/', authFarmId, cropController.getAllCrop());
10  router.post('/', cropController.addCropWithFarmID());
11  router.put('/:id', cropController.updateCrop());
12  // only user added crop can be deleted
13  router.delete('/:id', cropController.delCrop());
```

Figure 6

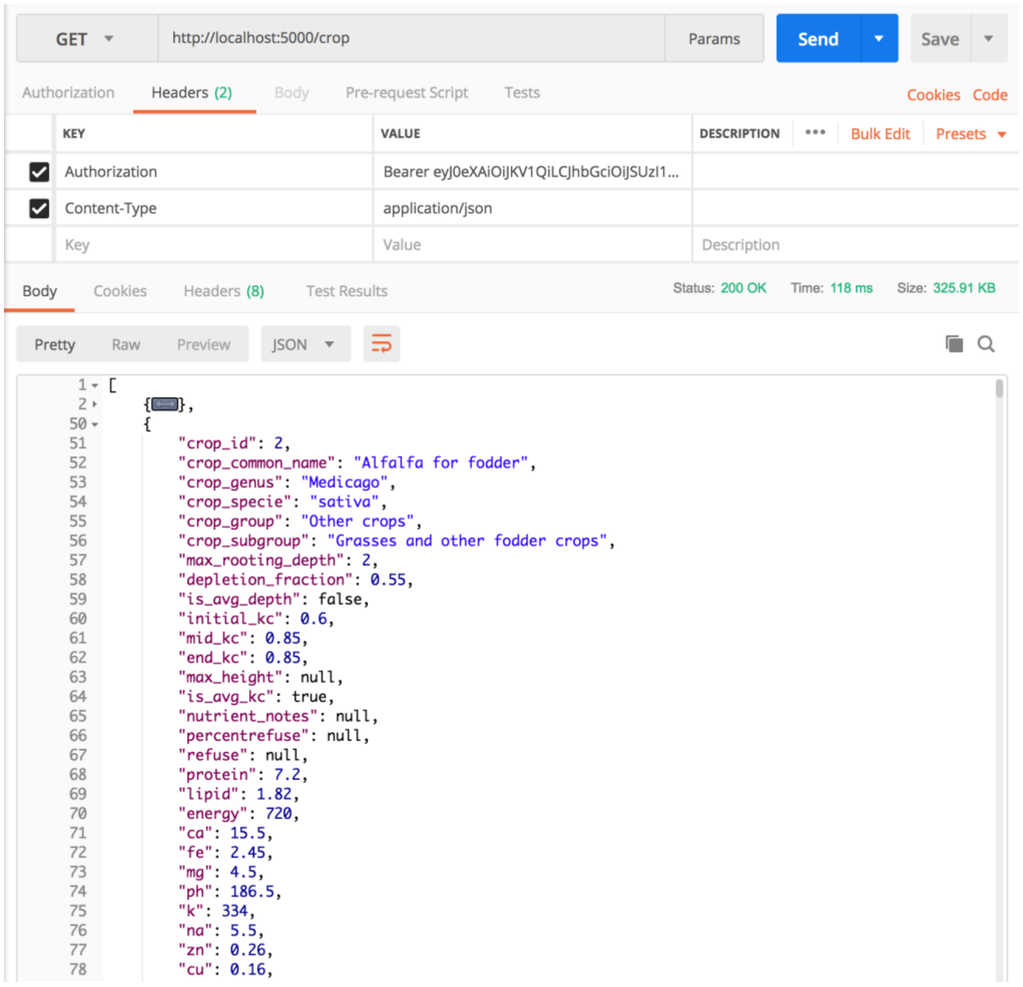
Since the client is making a simple GET call, the `cropRoutes` script will call the function on line 9 in figure 6. The `getAllCrop()` function will then make an inquiry to the database to retrieve the data in the crop table.

Step 3: complete a call

In the event that the inquiry is valid and successful, the database will return the data requested to the server. Then the server would send all the data to the client along with a `200 OK` status indicating the call was successful.

If some errors occurred during the inquiry process, the server would send back an error status code indicating the type of error along with an optional error message. For example, if the database is missing the crop table, then the server would send back a `500` error code indicating an internal server error.

An overview of the whole example



KEY	VALUE	DESCRIPTION
✓ Authorization	Bearer eyJ0eXAiOiJKV1QiLCJhbGciOiJSUzI1...	
✓ Content-Type	application/json	
Key	Value	Description

Status: 200 OK Time: 118 ms Size: 325.91 KB

```
[
    {…},
    {
        "crop_id": 2,
        "crop_common_name": "Alfalfa for fodder",
        "crop_genus": "Medicago",
        "crop_specie": "sativa",
        "crop_group": "Other crops",
        "crop_subgroup": "Grasses and other fodder crops",
        "max_rooting_depth": 2,
        "depletion_fraction": 0.55,
        "is_avg_depth": false,
        "initial_kc": 0.6,
        "mid_kc": 0.85,
        "end_kc": 0.85,
        "max_height": null,
        "is_avg_kc": true,
        "nutrient_notes": null,
        "percentrefuse": null,
        "refuse": null,
        "protein": 7.2,
        "lipid": 1.82,
        "energy": 720,
        "ca": 15.5,
        "fe": 2.45,
        "mg": 4.5,
        "ph": 186.5,
        "k": 334,
        "na": 5.5,
        "zn": 0.26,
        "cu": 0.16,
```

Figure 7

Figure 7 shows the call being made to the server on the top, and the response got sent back from the server at the bottom. The crop list is an array of objects, with each object being an row in the crop table.

Phase II: Front-end

The web app we are building is a single-page application (SPA). Unlike the traditional static HTML web that needs to refresh every time when something is changing, SPA renders the changing element dynamically using JavaScript on the same HTML page. An analogy would be watching cartoons using a flip book versus using a TV. This allows us to provide a much more smooth and true to a native application user experience.

We are building the SPA using React. React is a JavaScript library to build user interface developed by Facebook. It is a robust library that can be easily integrated with packages available on npm.

For instance, we needed a calendar component (figure 8) for user to choose a date. Instead of spending days making and testing our own calendar, we just installed a package called `react-dates` using npm. It is a calendar created and made available to the public by the good people at Airbnb. Though some styling and modification were still needed for the calendar, this dramatically reduced our development time.

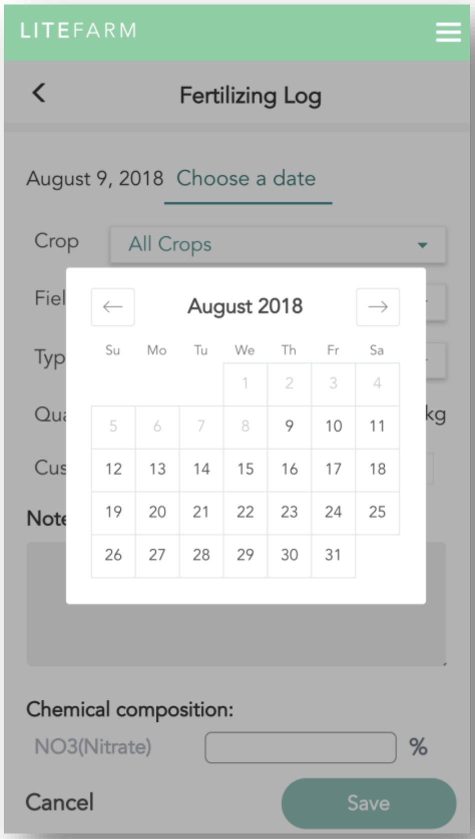
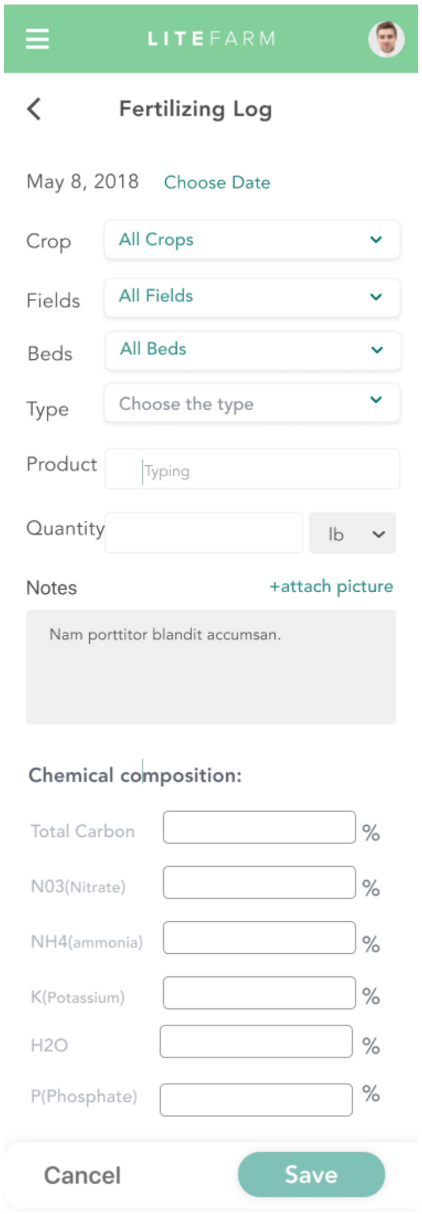


Figure 8 A calendar component

A key component of this web-app is offline capability. The plan (to be implemented in the coming weeks) is to use Progressive Web App (PWA) technology to achieve this goal. The advantage of a PWA compares to a normal web app is that users can save our web app onto their phones, and the app will still have some features working without a network connection. This can come in handy for farmers in rural area without easy access to the Internet when on a field.

Most of the front-end work is implementing the design made by the UX and UI designers on the project. Figure 9 shows the UI design and Figure 10 shows the current page being implemented. As you can see there are some differences between these, but since we are making an MVP we focus more on the implementation of functionalities, we opted out from pixel perfect mapping. The styling will be updated in the later stage of Phase II. About half of the front-end work is still in progress, but the project will be continued after August 10th, 2018.



May 8, 2018

Crop	All Crops
Fields	All Fields
Beds	All Beds
Type	Choose the type
Product	Typing
Quantity	lb

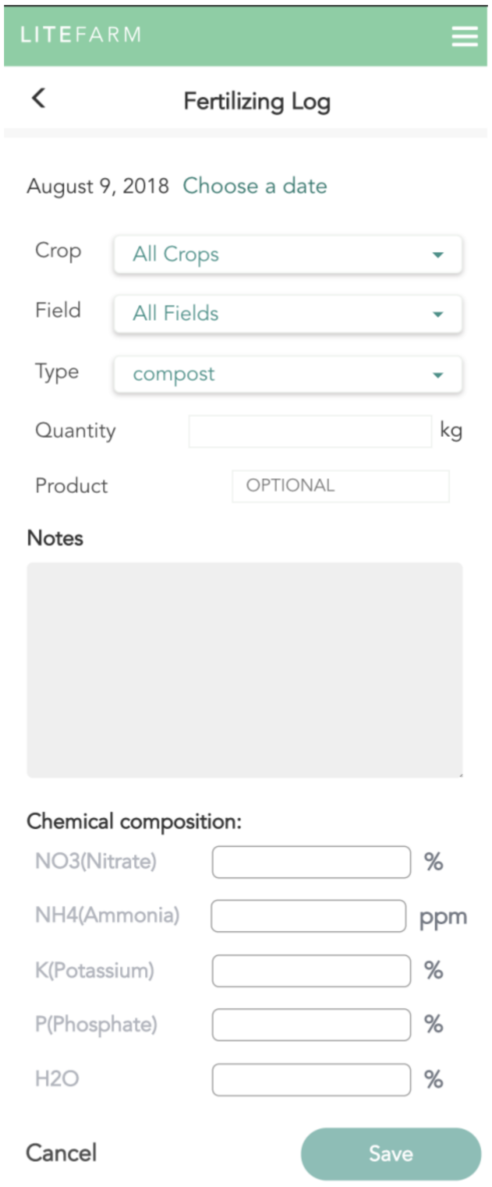
Notes

Nam porttitor blandit accumsan.

Chemical composition:

Total Carbon	%
N03(Nitrate)	%
NH4(ammonia)	%
K(Potassium)	%
H2O	%
P(Phosphate)	%

Figure 9



August 9, 2018

Crop	All Crops
Field	All Fields
Type	compost
Quantity	kg
Product	OPTIONAL

Notes

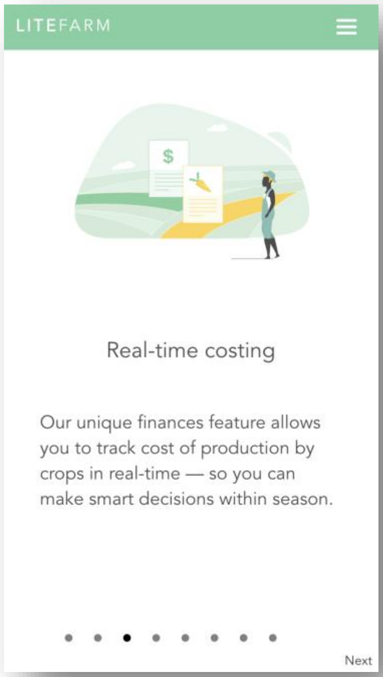
Chemical composition:

NO3(Nitrate)	%
NH4(Ammonia)	ppm
K(Potassium)	%
P(Phosphate)	%
H2O	%

Figure 10

Front end work

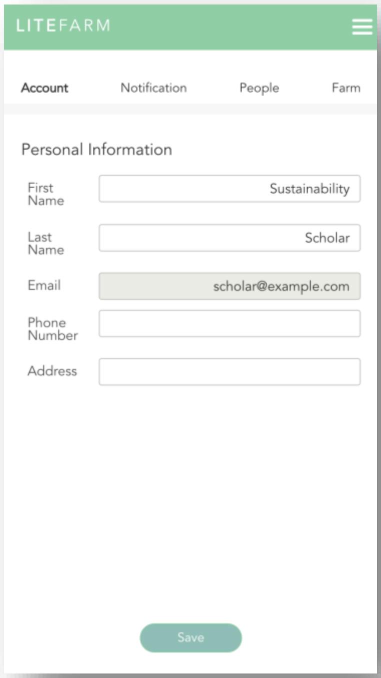
I managed to work on a rage of the key features of this application during my time on the project. These included, the profile, logs features and various components. Screenshots below show an overview of examples of pages that I have implemented..



Real-time costing

Our unique finances feature allows you to track cost of production by crops in real-time — so you can make smart decisions within season.

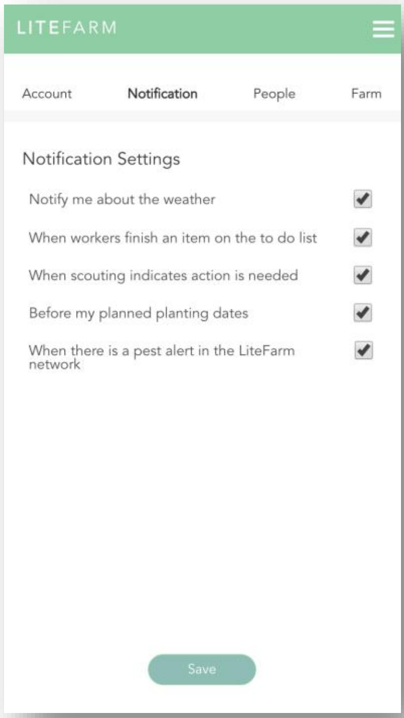
Figure 11 Onboarding Page



Personal Information

First Name	Sustainability
Last Name	Scholar
Email	scholar@example.com
Phone Number	
Address	

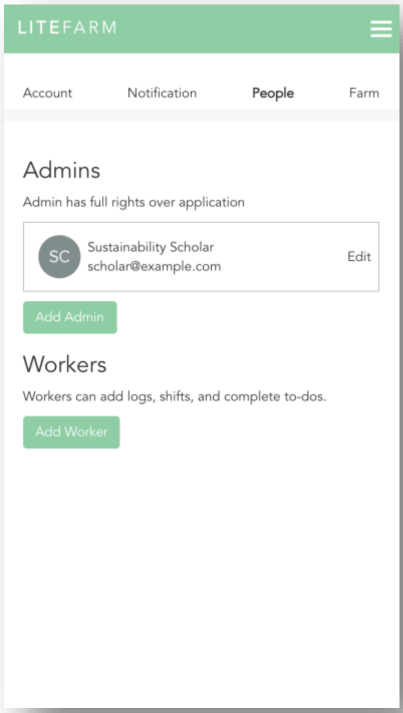
Figure 12 Profile - Account Setting



Notification Settings

Notify me about the weather	✓
When workers finish an item on the to do list	✓
When scouting indicates action is needed	✓
Before my planned planting dates	✓
When there is a pest alert in the LiteFarm network	✓

Figure 13 Profile - Notification Setting



Admins

Admin has full rights over application

SC Sustainability Scholar scholar@example.com

Workers

Workers can add logs, shifts, and complete to-dos.

Figure 14 Profile - People

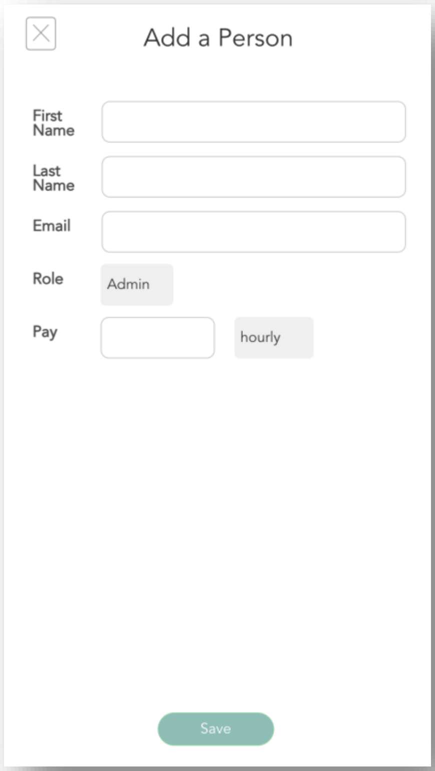


Figure 15 Profile - People - Add a person

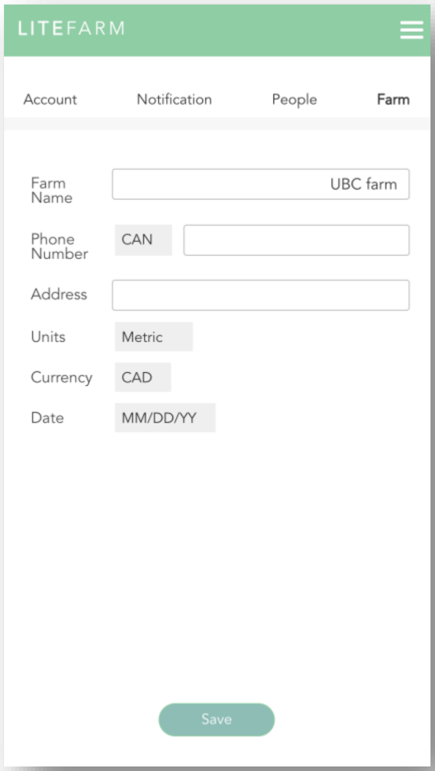


Figure 16 Profile - Farm setting

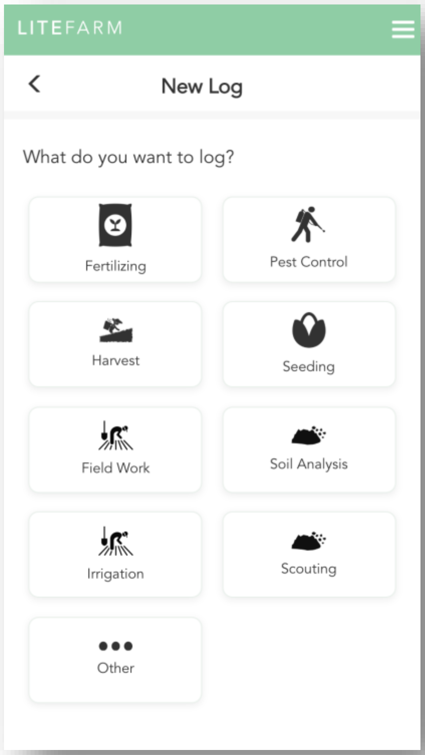


Figure 17 Add a new log

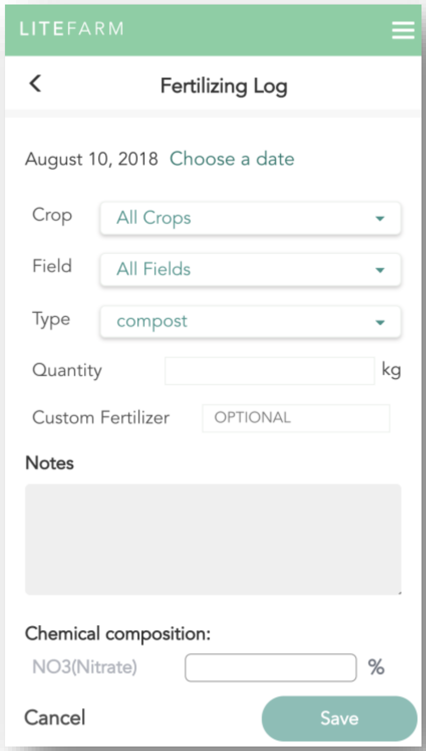


Figure 18 Add a Fertilizer Log

Summary

LiteFarm presents high potential to attract its target audience. The feature-rich web app provides a single platform for farmers to manage their farms while maintaining a degree of simplicity. Users of LiteFarm can better track their production process and make smarter decisions during the season. With historical data provided to the farmers, they can make better plan with their operations. All of these would make their farming more sustainable and profitable. Being an open source project encourages more people to participate in the projects, which in turn making LiteFarm more easily to be updated and improved by the community in the future.

References

Wikipedia. “Minimum Viable Product.” *Wikipedia*, Wikimedia Foundation, 24 July 2018, en.wikipedia.org/wiki/Minimum_viable_product.

Roser, Max. “Employment in Agriculture.” *Our World in Data*, 2018, ourworldindata.org/employment-in-agriculture.