UBC Farm Web Application Design Document & Report

UBC Sustainability Scholars Program Project:
Stewardship science technology for monitoring the socio-ecological outputs of farming activities

Xingyu Tao

August 2017

Table of Contents

Acknowledgements

With special thanks to Prof. Hannah Wittman and Dr Zia Mehrabi for all the support provided in completing this project.

Abstract

The UBC Farm Socio-ecological Monitoring Application (hereafter "UBC Farm App") is an ongoing project that aims to develop a dynamic monitoring and reporting system for UBC Farm that can also be adapted to other scales of farming operations (local, regional, global). Its ultimate goal is to make farming more sustainable by closely tracking and reporting on every aspect of a Farm's operation, from resources used, hours worked, to bio-diversity levels, climate change impacts, finances, and food produced. Through several research, design, and implementation cycles, it has reached its current iteration which combines an array of intercommunicating modules together to offer real-time tracking and reporting functions.

This document aims to provide a detailed report on the system architecture and design choices of the application's current iteration (April – September 2017). It accompanies the online documentation and code for the application found https://github.com/ubc-farm and includes an overview of the following features currently in the implementation queue of the project:

1. Fields	Completed
2. Tasks	Completed
3. Inventories	Completed
4. Finances	Completed
5. Employees	Implementing
6. Reports	Implementing

System Architecture and Rationale

This application makes use of a layered architecture to separate the database, business logic, and UI to help facilitate code development. This follows naturally from the "MERN" (MongoDB, Express, React, Node) stack's design philosophy. The restriction of imposing an established framework such as React on the project improves the stability of the web application, and it also makes incorporating new features easier due to extensive existing reference.

The figure below gives a visual representation of each component's role in the current UBC Farm application.

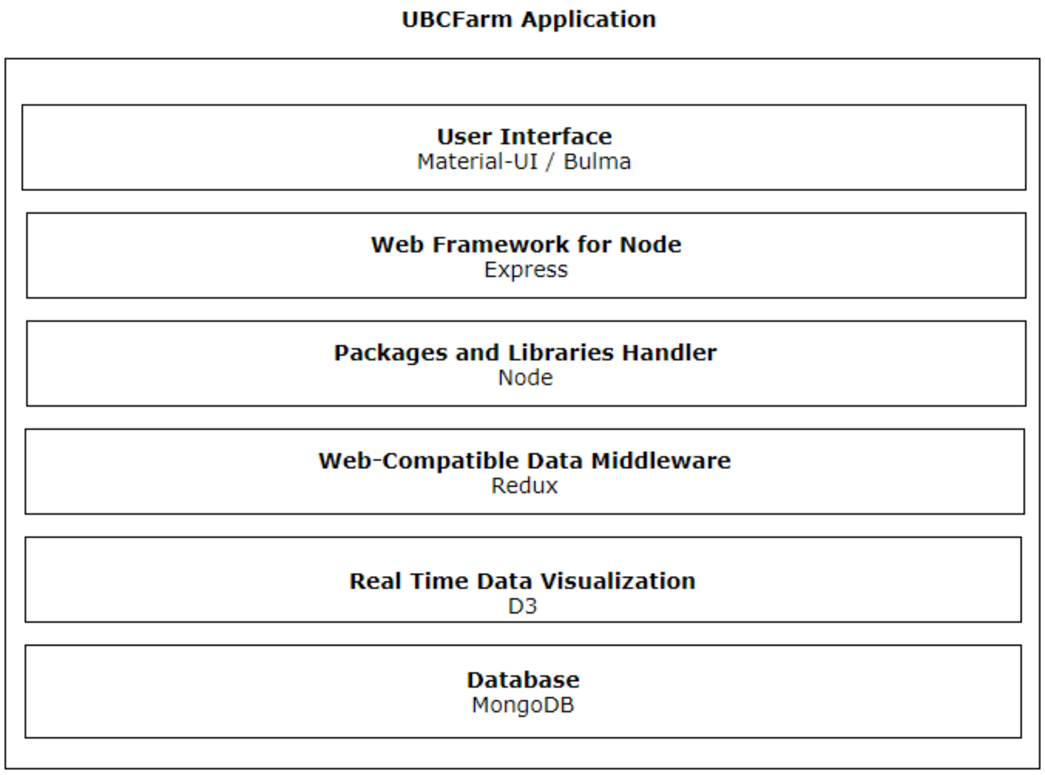


Figure 1 - Layered Component Diagram

Each component in the UBC Farm App's layered architecture fulfills a well-defined role. This separation of tasks also facilitates the control for the data flowing between each layer, so that sensitive information can be kept secure as required for private farm owners and as specified by UBC's information security policies.[1] Briefly, the specific roles of each of the component will be elaborated in the following pages.

[1] *Acceptable Use and Security of UBC Electronic Information and Systems*. PDF. Vancouver: The University of British Columbia Board of Governors, June 2013.

Material UI & Bulma

At the top level, for styling and responsive design, this app uses two CSS Frameworks: Material UI and Bulma. Material UI is a mobile-first design widely used on Android and Chrome, which contains a wide array of ready-to-use components such as search bars, form fields, and date selectors. Bulma is a web framework useful for its sleek fonts and responsive columns. Together they provide the styling for this application's components and are the most important factors controlling the aesthetic appearance of the user interface.

Express

As a minimalist and flexible Node Framework for web applications, Express offers a way to standardize development of web applications in an object-oriented way. It is a service that abstracts the HTML document's elements into objects by using JavaScript. The way it accomplishes this and manages to escape the pitfalls of the classical HTML/CSS/JAVASCRIPT pattern is by transpiling (translating code) from object oriented syntax into pure JavaScript code, on top of the root HTML document. This approach to programming web applications has many benefits including more accurate modeling of complex objects, and more readable code for collaboration purposes.

Node

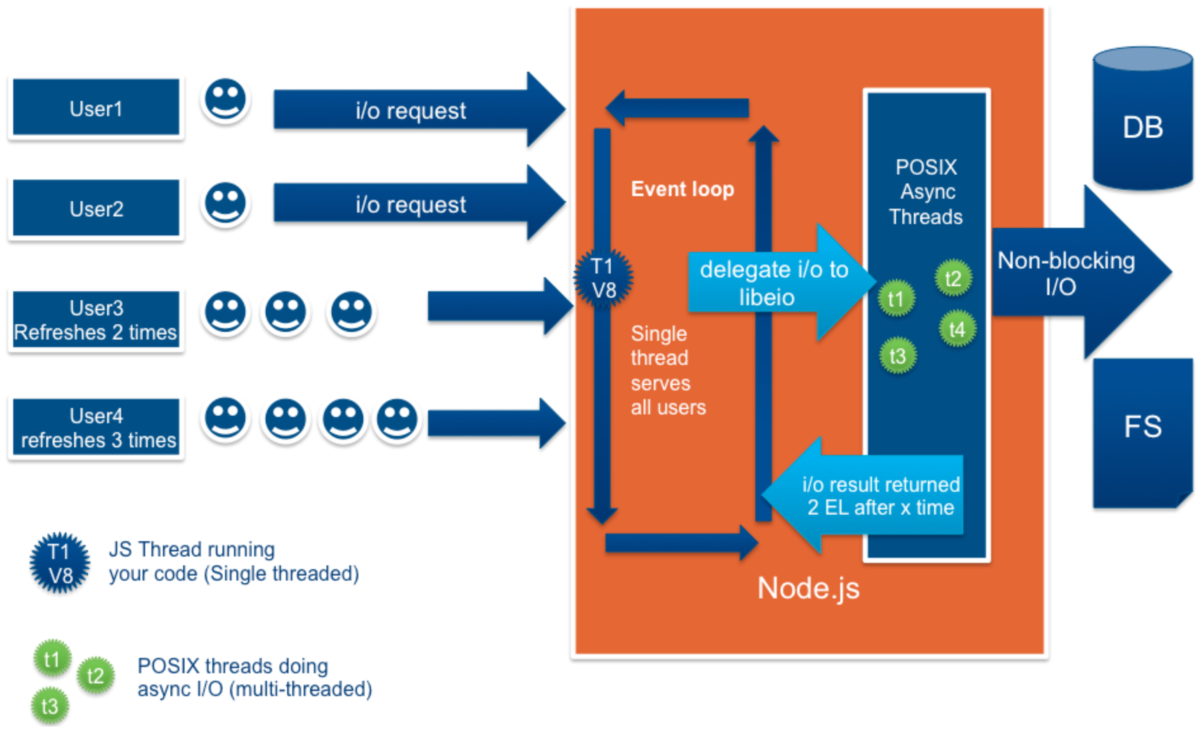


Figure 2 - Node Workflow

Node provides an event-driven JavaScript runtime for web applications. Its Input/Output mechanism is non-blocking, which allows the UBC Farm App to support asynchronous (random) updates. This is an especially important component that drives the real-time updating features of the UBC Farm App. Its package environment and online libraries (npm) contains the largest repository of open source codes that provide services such as authentication schemes, css frameworks, and database APIs (Application Programming Interfaces). Virtually all services from Google Maps to MongoDB have their APIs available in the form of npm packages.

Redux & React

Redux and React are two separate components that are tightly linked together. React is essentially a JavaScript library that allows the incorporation of states into HTML elements. This is crucial for building an interactive application that requires communication across different modules. React keeps track of each element's state, alerts modules of changes, and updates the User Interface accordingly, mimicking the function of a state machine. An element's state can be defined as anything, but most often it contains the element's current attributes.

Redux builds on React by providing the link between React states and a third party Database (in our case MongoDB). Redux uses React's control over the Document Object Model (DOM) element to dynamically update the UI according the changes in the Database, which it keeps a copy of in the *Store*. In doing so, Redux acts as the application's event handler for user triggered actions such as creating, editing, and deleting information. Redux maintains consistency across all instances of the application, so it avoids race cases and deadlocks (two users try to change one value at the same time, creating inconsistencies).

[2]

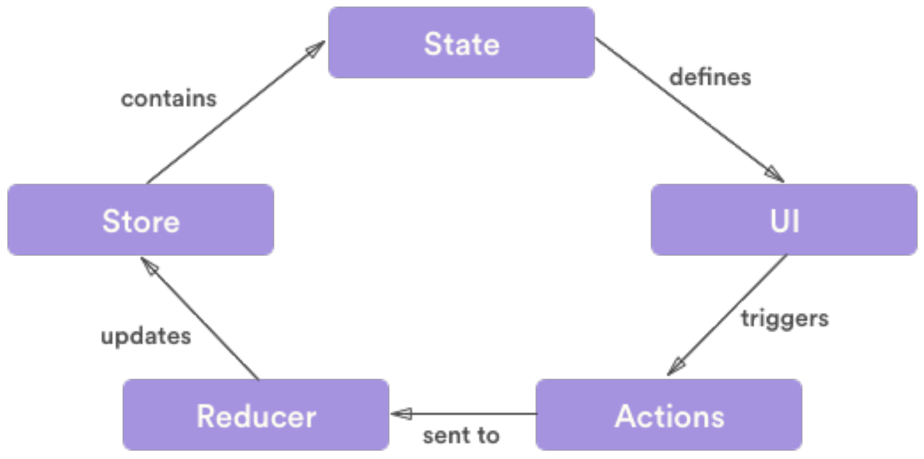


Figure 3 - Redux React Workflow

[2] Goutay, Nicholas. "Redux Workflow." Digital image. Getting Started with React, Redux and Immutable: a Test-Driven Tutorial (Part 2). March 30, 2016. Accessed August 3, 2017. https://www.theodo.fr/blog/2016/03/getting-started-with-react-redux-and-immutable-a-test-driven-tutorial-part-2/.

D3

D3.js is a well-known JavaScript library that provides various data visualization functionalities. In the UBC Farm App, D3 is used to visualize time-series as well as geographical data and with the help of React Redux the visuals created with D3 can be updated dynamically with user inputs.

MongoDB

MongoDB is a document-based, object-oriented Database. Compared to table-based structures such as SQL, MongoDB offers an easier way to store data as objects, as well as crossing and nesting them. If needed it is also easily transformed into tabular form. This way, data about objects can be stored the same way they are modeled. This in turn makes the application as a whole more scalable, and ensures consistency in data models between the frontend and the backend of the application. It also makes incorporating third party data with JSON formatting a trivially easy task. For this application we will be interacting with MongoDB through Mongoose. Mongoose is a schema based framework for modeling application data, streamlining interactions with MongoDB.

[3]

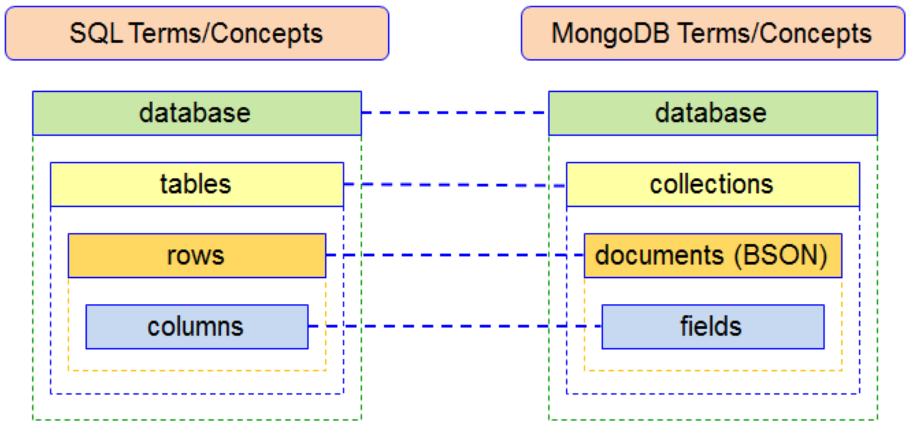


Figure 4 - SQL vs. MongoDB Structure

[3] Malaya, Victoria. SQL vs. MongoDB Terms/Concepts. Digital image. Sql-vs-nosql. November 8, 2013. Accessed August 3, 2017. http://sql-vs-nosql.blogspot.ca/2013/11/indexes-comparison-mongodb-vs-mssqlserver.html.

Detailed Design

With the help of the different services and frameworks discussed in the previous section, the application is built in a modularized fashion. The visual below provides an UML diagram view of the overall relationships between different modules of the UBC Farm App.

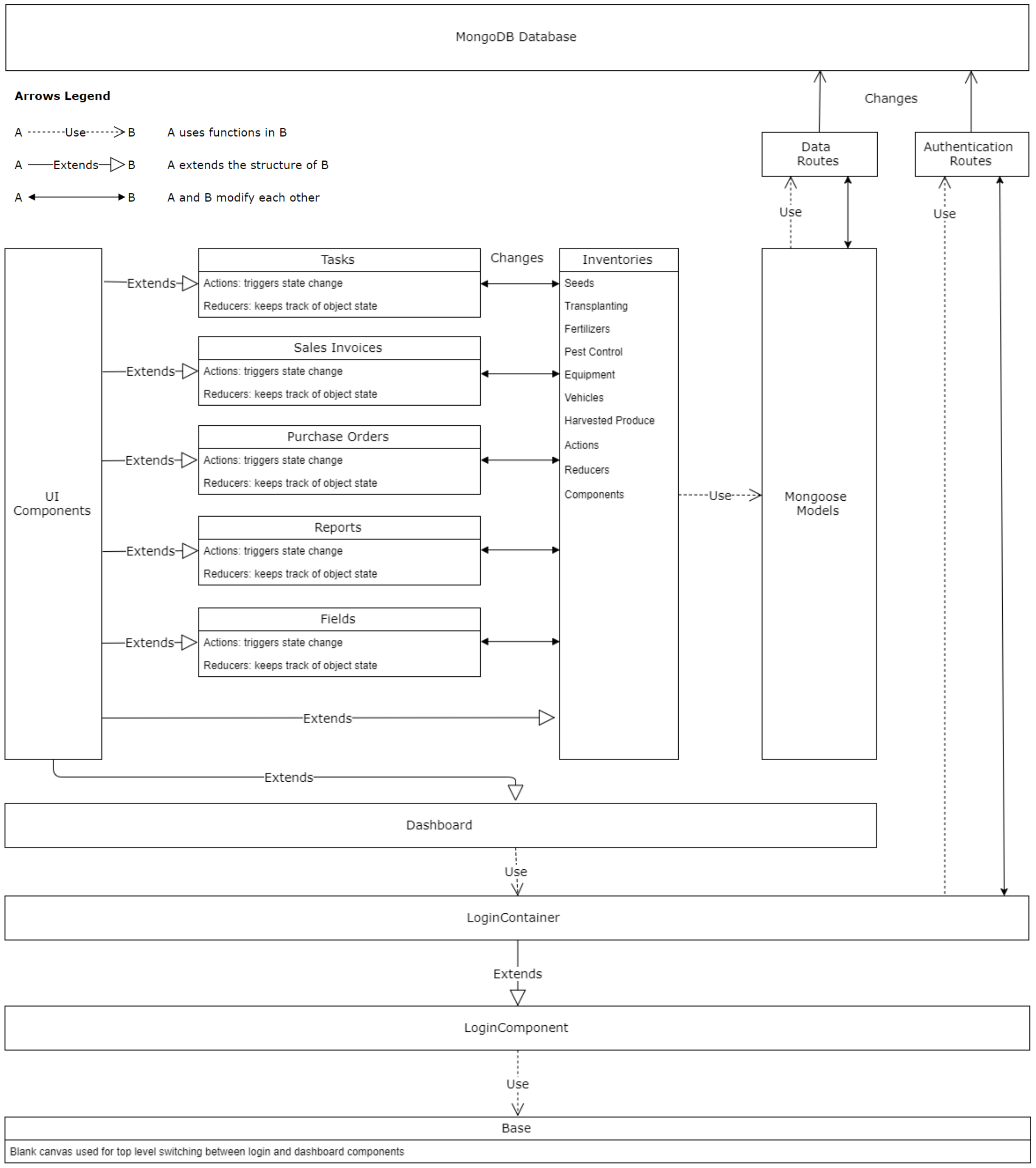


Figure 5 – UBC Farm App Detailed Design UML Diagram

Design Patterns

Several well-established design patterns provide the foundation for adding new modules to the current project. The *Component Container Pattern* separates functional components from presentational components so that these tasks are handled separately. The *Redux Store* and *Redux React Pattern* links the app to a third party Database, in this case MongoDB, to perform Data storage and retrieval.

Component Container Pattern

The *Component Container Pattern* is a method to separate the tasks of data interaction from those concerned with User Interface. Separating code by function is also a way to make collaborative development and debugging easier.

The diagram to the right describes the basic workflow of a module constructed according to the *Component Container Pattern.* As illustrated, the *Component* interacts with the data source while the *Container* interacts with the User. Most of the components in this application are constructed this way, with a *Component* JavaScript file interacting with Mongoose files and a *Container* JavaScript file specifying the User Interface layouts.

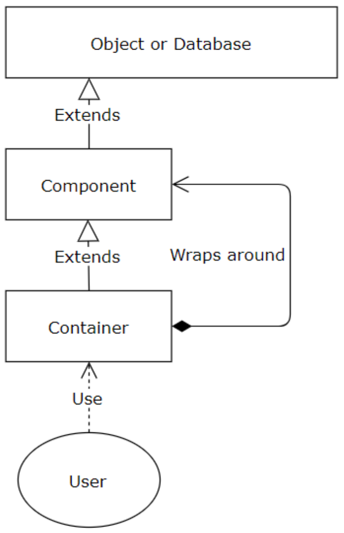


Figure 6 - Component Container Pattern

Redux Store, Redux React Patterns

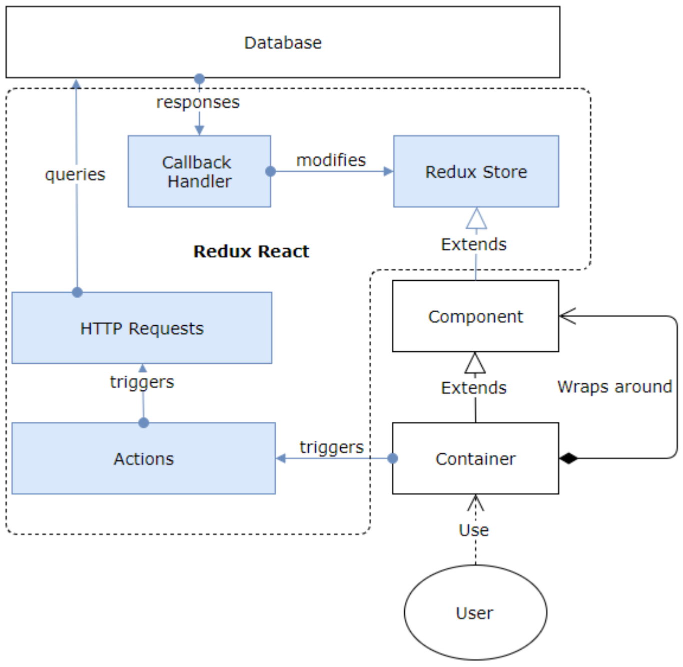


Figure 7 - Redux React Pattern

It is no surprise that Redux and its services are highly connected with React. Redux is meant to act as a middle-man between Databases and React Components. Its main concern is to keep the frontend and backend consistent with each other. One dra wback of Redux and React worth mentioning is that it interrupts the normal workflow of the DOM object. This means that other services that also interact with the DOM object (such as D3) need their own Redux wrappers as well in order to work smoothly.

Introduction to Pages

The main functionalities of the UBC Farm App are grouped by pages, which appear as tabs in the top navigation bar of the webpage. They are *Fields, Tasks, Inventories, Reports, Finances,* and *Users*. All of them are connected to the Redux Store via Reducer files, which encapsulate actions that each object triggers when Users interact with them. As a result, they are all dynamic, and user controlled. Data updates and rendering changes on the User Interface are instantaneous, which is a crucial advantage that Redux React provides.

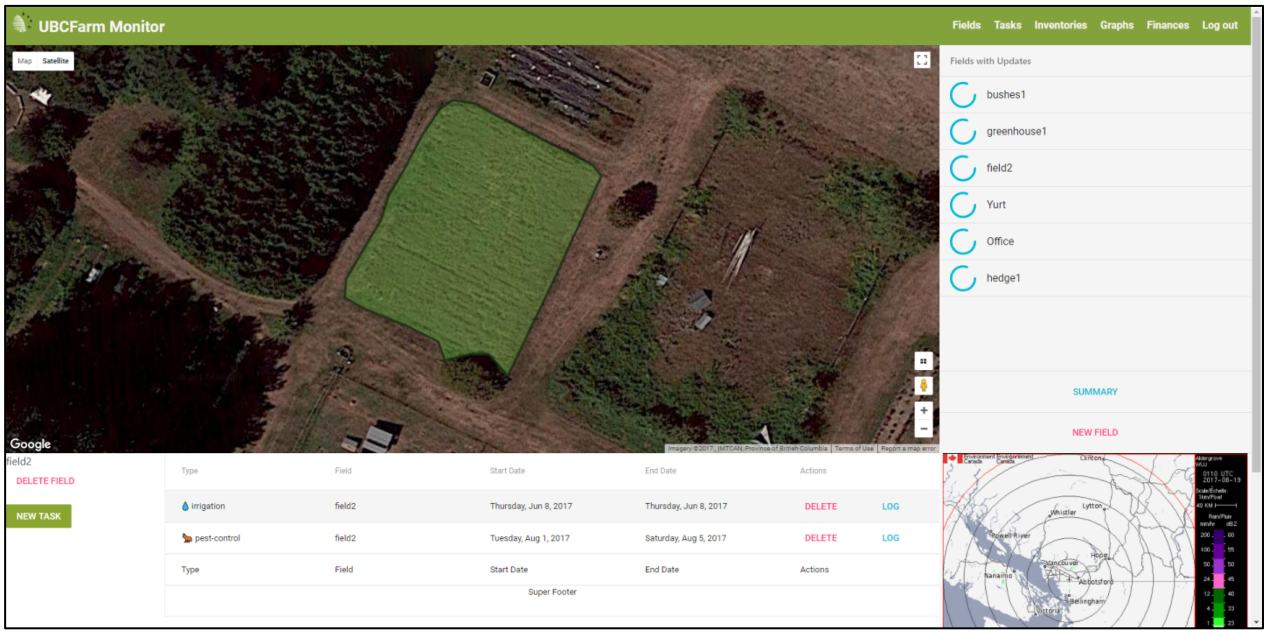
Fields

The *Fields* page contains a prominent Google Maps Component that visualizes the geographical data associated with farm buildings and fields. Users can use the list to the right to focus on a specific field or building, which has its associated attributes and active tasks displayed below the Map. The page also contains a weather module, which will be expanded in the future to include more features. The main fields in a *Field* object model are its *name*, its associated *GEOJson Polygon* representing its geographical area, and a *TaskList* representing the current tasks that are associated with the field. Standard CRUD actions can be performed on the field, including *deleting*, *fetching*, and *saving*. The *select* action is added to support field selection on the *Fields* page.

Field
Fields
name
polygon GEOJson
taskList
Actions
delete
fetch
save
select

Figure 8 - Field Object Model

Google Maps API



Type	Field	Start Date	End Date	Actions	
irrigation	field2	Thursday, Jun 8, 2017	Thursday, Jun 8, 2017	DELETE	LOG
pest-control	field2	Tuesday, Aug 1, 2017	Saturday, Aug 5, 2017	DELETE	LOG
Type	Field	Start Date	End Date	Actions	

Super Footer

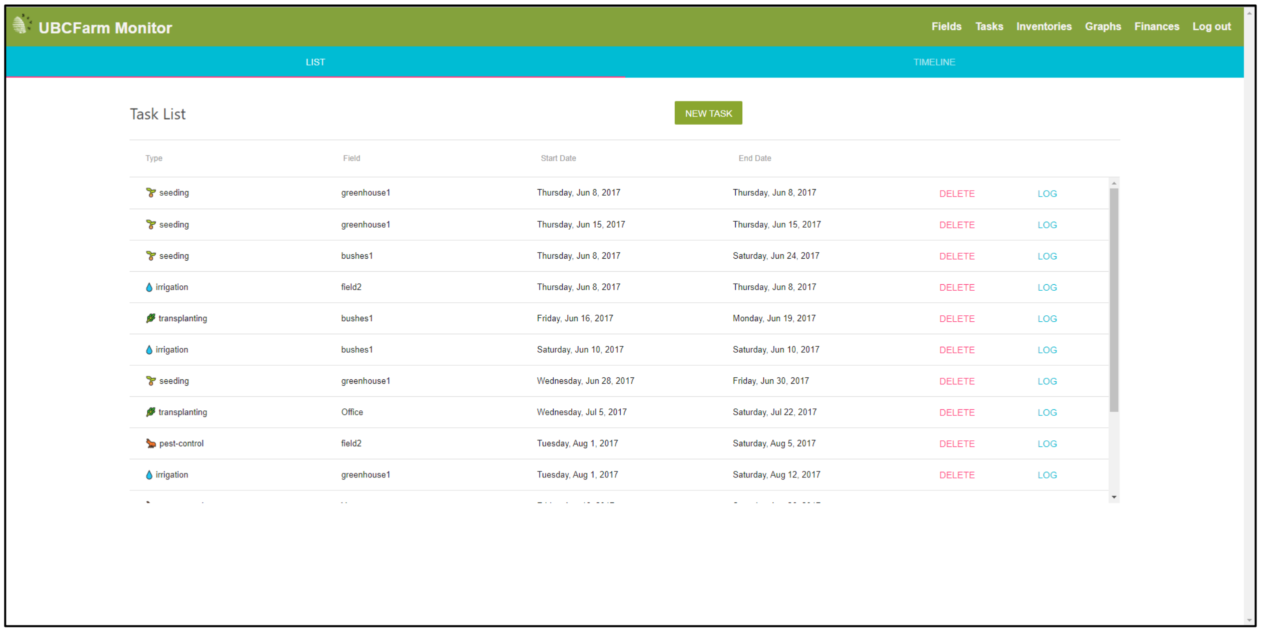
Figure 9 - Fields Page

The *Fields* page uses Google Maps API to visualize the geographical information of fields and buildings, including calculating their coverage area, as well as performing accurate distance measurements. To make the Google Maps API compatible with React Components, a helper *npm* package called *react-google-maps* is used, as well as some custom JavaScript code.

Tasks

There are 8 types of tasks that can be set in the current prototype: *seeding, irrigation, pest-control, transplanting, soil-sampling, scouting-harvest, scouting-pest,* and *fertilizing* (additional tasks for e.g. bed preparation, weed control, and custom user defined tasks, will be included prior to release). Each task is associated with a specific field or building. When a user is logging a task, resources can be subtracted from or added to inventories. These changes are tracked by the inventories, and form the basis from which data can be gathered for report generation.

The *Tasks* page contains two tabs that allow users to switch between a List view of the tasks and a Timeline view of the tasks.



Task List

Type	Field	Start Date	End Date		
seeding	greenhouse1	Thursday, Jun 8, 2017	Thursday, Jun 8, 2017	DELETE	LOG
seeding	greenhouse1	Thursday, Jun 15, 2017	Thursday, Jun 15, 2017	DELETE	LOG
seeding	bushes1	Thursday, Jun 8, 2017	Saturday, Jun 24, 2017	DELETE	LOG
irrigation	field2	Thursday, Jun 8, 2017	Thursday, Jun 8, 2017	DELETE	LOG
transplanting	bushes1	Friday, Jun 16, 2017	Monday, Jun 19, 2017	DELETE	LOG
irrigation	bushes1	Saturday, Jun 10, 2017	Saturday, Jun 10, 2017	DELETE	LOG
seeding	greenhouse1	Wednesday, Jun 28, 2017	Friday, Jun 30, 2017	DELETE	LOG
transplanting	Office	Wednesday, Jul 5, 2017	Saturday, Jul 22, 2017	DELETE	LOG
pest-control	field2	Tuesday, Aug 1, 2017	Saturday, Aug 5, 2017	DELETE	LOG
irrigation	greenhouse1	Tuesday, Aug 1, 2017	Saturday, Aug 12, 2017	DELETE	LOG

Figure 10 - Tasks List View

The list view clearly shows the type, dates, and field associated with each specific task. Buttons for logging tasks and deleting them are also easily accessible. Tasks can also be set from the *Fields* page as well, where each field has its own Task List displayed, containing only tasks that are associated with it.

VIS Timeline API

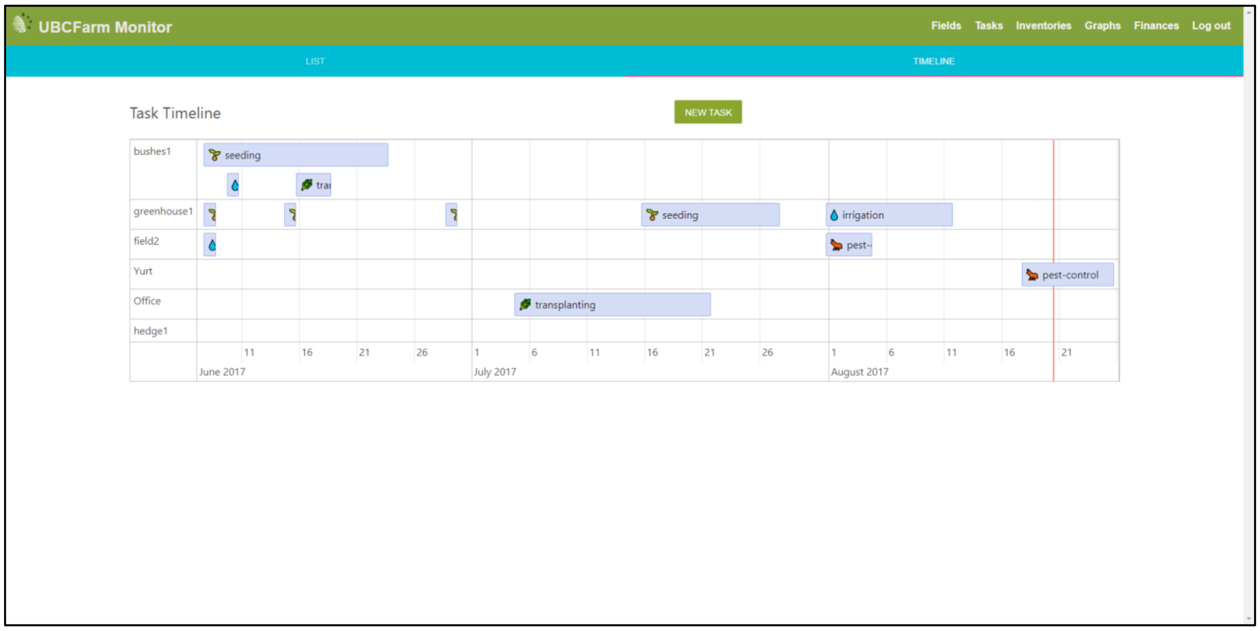


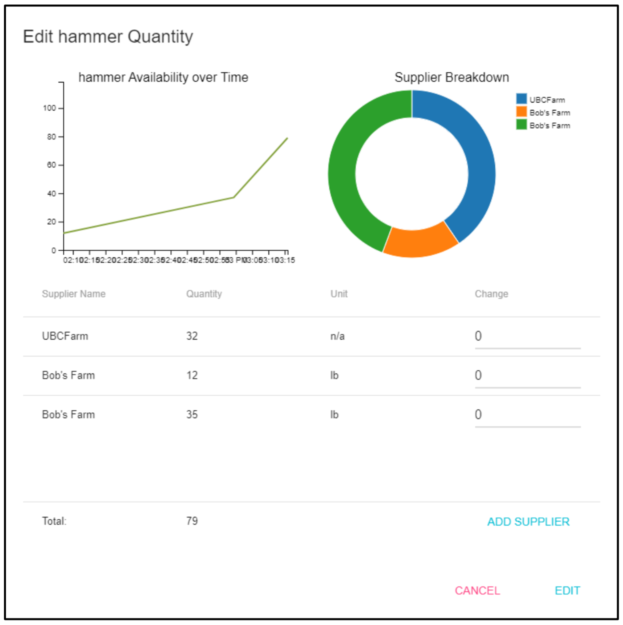
Figure 11 - Tasks Timeline View

To provide a more visual way of representing the tasks, the *vis.js*[4]Timeline package was used. In order to make the package compatible with React Components using it, a React wrapper file was necessary. The Timeline shown in the figure above adjusts automatically to new tasks being set, and users can freely zoom in and out to adjust the timeframe.

Inventories

The *Inventories* page is a collection of Lists representing all resources being tracked. The 7 main categories of resources identified are: *Seeds, Transplanting, Fertilizers, Pest Control, Equipment, Vehicles,* and *Harvested Produce*.

Each object in each collection is further subdivided by *supplier* if applicable, to separate cases where the same resource has several different sources. This increases the granularity of data available and allows for more accurate tracking of resource usage.

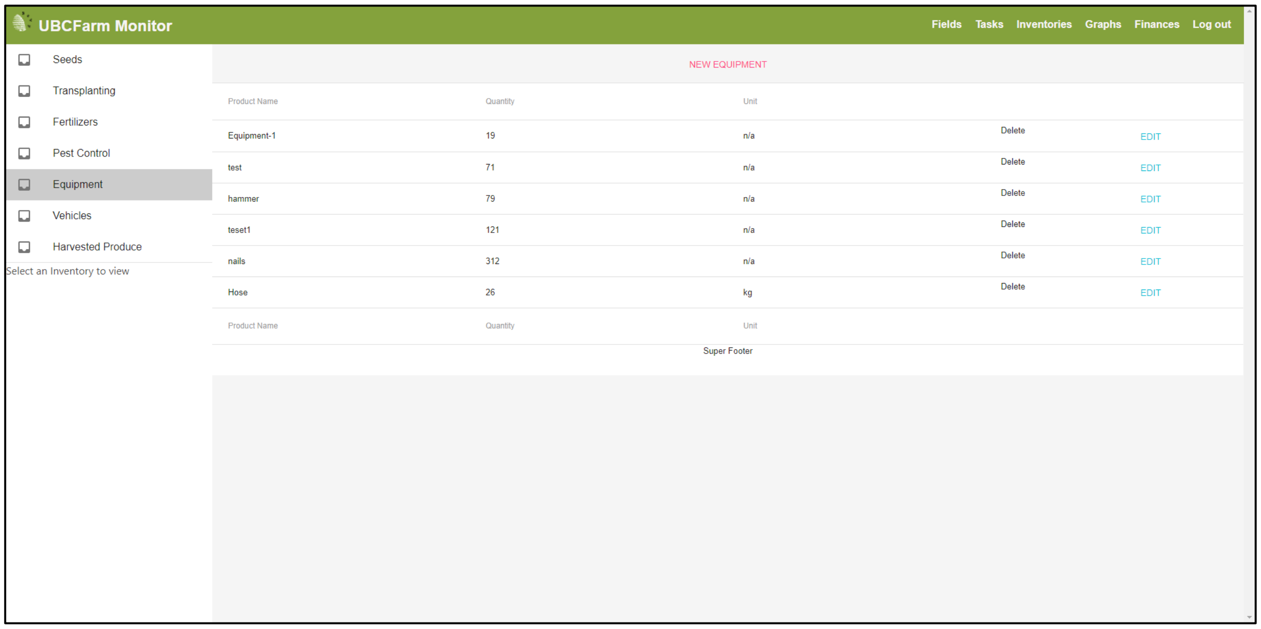


Edit hammer Quantity

Supplier Name	Quantity	Unit	Change
UBCFarm	32	n/a	0
Bob's Farm	12	lb	0
Bob's Farm	35	lb	0
Total:	79		

Figure 12 – Inventory Item Modal Example

[4] Documentation available at *visjs.org*



Product Name	Quantity	Unit		
Equipment-1	19	n/a	Delete	EDIT
test	71	n/a	Delete	EDIT
hammer	79	n/a	Delete	EDIT
teset1	121	n/a	Delete	EDIT
nails	312	n/a	Delete	EDIT
Hose	26	kg	Delete	EDIT
Product Name	Quantity	Unit		

Super Footer

Figure 13 – Inventory Page

D3 API

In order to visualize the composition of different Inventory Items, D3 is used to create line and pie charts detailing the availability of items over time, as shown in figure 12. Although being a declarative way to build visuals, because D3 has its own update workflow it is harder to incorporate with React compared to other APIs such as Google Maps or VIS. Currently, the D3 components are updated by manually triggering a re-rendering of the containing React component. This issue is further discussed in the Design Challenges and Recommendations section of this report.

Finances

The *Finances* page is subdivided into 5 tabs: *Summary, Clients, Suppliers, Invoice,* and *Purchase Form*. *Summary* presents an overview of the financial inputs and outputs of the Farm over a specified amount of time adjustable by the user. *Clients* and *Suppliers* each maintain a list of current clients and suppliers and keeps record of their historic activities by tracking purchase and sales items. *Invoice* and *Purchase Form* are two form pages where users create new invoices and purchase forms.

Users & Reports

The *Users* page keeps a list of active users/employees of the Farm. These can be edited with administrator access. The *Reports* page contains various reports generated with D3 using data gathered through all the different application modules. The data is aggregated and calculated automatically.

Design Challenges and Recommendation

Throughout the development of the current iteration of the UBC Farm App, there have been many obstacles to overcome. Listed below are two challenges that had significant impact on the way the application is built. These points should be taken into account when designing future components.

Integrating Modules with React

There are many different third party packages and modules incorporated into this project. Integrating all of them with React smoothly is then very important in order to assure efficiency and to avoid bugs. This report mentions three such modules: Google Maps, Vis Timeline, and D3.

D3 in particular was hard to integrate with React since it has its own update cycle separate from that of the DOM object (the backbone of any website). It is therefore separate from React's workflow as well. In order to ensure that updates from one of these two will trigger updates in the other, there exists two possible solutions:

1) Re-render the SVG visual generated by D3 with new data when changes occur
   - Pros: Data will be consistently updated
   - Cons: Updates might become choppy if internet is slow (affects user experience)
2) Hand the handling of the on-click event completely to D3
   - Pros: Updates will be smooth, allowing better animation and transition
   - Cons: Listeners must be set up on both sides to relay user inputs and new data

In this case, option 2 was chosen because it costs little extra work to set up listeners and it provides a better user experience with smoother transitions.

Interconnectivity

This is an issue that the *Redux React* pattern is aimed at solving. Without it, the state of each object needs to be monitored separately and all must communicate with a shared database. *Redux React* takes control of this process, using its *Store* as a mockup of the database. While it seems like just a copy of information form the database, it fulfills the important job of keeping the state of every object very organized from the developer's point of view. Behind the scene, it also updates the DOM element affected by changes automatically, so no JavaScript event listeners need to be manually set up.

Conclusion

The UBC Farm App presents great potential for building an agriculture oriented task management and reporting system that increases the sustainability of farming practices by tracking the socio-ecological footprint of farming. This project is a great opportunity for both research into building scalable software applications and learning more about sustainable farming practices. The most important result from this Sustainability Scholars project is a solid foundation on which more improvements and new modules can be added to in the future.