Creating Transparency Within The Organic Food Supply Chain

Samuel John Morgan
912438

Submitted to Swansea University in fulfilment
of the requirements for the Degree of Bachelor of Science

Bachelor of Science

Department of Computer Science
Swansea University

May, 2020

Declaration

This work has not been previously accepted in substance for any degree and is not being con-currently submitted in candidature for any degree.

Signed

Date 7/5/2020

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed

Date 7/5/2020

Statement 2

I hereby give my consent for my thesis, if accepted, to be made available for photocopying and inter-library loan, and for the title and summary to be made available to outside organisations.

Signed

Date 7/5/2020

Abstract

This dissertation presents an attempt at creating transparency within the organic food supply chain using Ethereum smart contracts. The system allows farmers, processors and retailers to transfer livestock (products) and produce between each other. Once a product reaches a retailer, it is given a label id that can be inserted into a decentralised web app which will then present the products organic properties and the path it has taken through the supply chain. Organic certifications can be distributed to a farmer by an organic body that is defaulted to 'The Welsh Organic Scheme'.

Table of Contents

Table of Figures

Chapter 1

Introduction

Food is seen as being organic if it is grown and handled in a manner that adheres to standards set by a governing body. The United States Department of Agriculture requires that crops should not be subject to genetic engineering, artificial colours, and synthetic fertilisers (Federal Register, 2010). Livestock is required to have year-round access to the outdoors, which is dependent on the weather (Federal Register, 2010). The desire for organic food has seen a 107% increase in the EU market between 2006 and 2015 (Willer, et al., 2017). This is in part due to the past food scandals such as the fipronil egg scandal where the outlawed insecticide fipronil affected 700,000 eggs imported into Britain (BBC News, 2017) (European Commission, 2019). With supermarkets now stocking a broad range of products such as free-range chicken, it is becoming more accessible than ever to go organic (Sainsbury's, n.d.).

The Ethereum blockchain (Buterin, 2013) is a cryptographically secure immutable group of records in which complex smart contracts can be created. Smart contracts allow the peer-to-peer transfer of currency and data. A contract is fulfilled once terms decided by the buyer and seller are met removing the need for a middle man. These transactions are immutable and visible for the public to see, therefore decreasing the chance of corruption. Blockchain technology has seen investment from large corporations such as Wal-Mart who have completed two proof-of-concept tests on the hyper ledger blockchain (Hyperledger, 2016).

1.1 Motivations

The primary motivation of this dissertation is to solve issues of distrust between customers and entities within the supply chain. Gaining the trust of the customers is crucial if the organic food industry wishes to succeed. Without trust, customers may not see the value in spending a premium for organic products. This trust has been tested in the past, an example of this being when in August 2019 a US District Judge found Randy Constant guilty of lying to consumers about the authenticity of his organic products and was sentenced to 10 years in prison (Foley, 2019). This dissertation aims to solve this problem by using the properties of the Ethereum blockchain, to create a system which provides an immutable and public record of products being transferred through the supply chain. An emphasis will be put on the different organic properties each product holds, such as pesticides used. Currently, there is not a system that is accessible to the public, so this dissertation is venturing into new territory. The lack of current systems may in part be due to the difficulty in scaling a system that will be able to encompass large supply chains.

My personal motivation for creating such a system is to hold companies that mistreat their livestock accountable for their actions by displaying their farming methods on immutable public records. This gives consumers a greater understanding of what they are buying and hopefully encourage them to purchase ethical products.

1.1.1 Aims

This dissertation aims to prove the possibility of creating transparency in the organic food supply chain using Ethereum smart contracts. To achieve this, a set of initial aims were created:

- Create software to track produce as it is transported through three different parts of the supply chain.
  - Farmer
  - Processor
  - Retailer
- Create a distributed application for web browsers.
  - Create a portal for farmers, processors, retailers, governing bodies and customers.
- Store organic certification issued to farmers by governing bodies on the blockchain.
- Accommodate for produce.
  - Store the types of pesticides and fertilisers used to grow produce on the blockchain.
- Accommodate for livestock.
  - Store the types of feed, where the livestock is held and how long they spend outdoors on the blockchain.
- Give the user an understanding of how organic a compound product (ready meal) is.
  - Provide a percentage of the ingredients that are organic.
- Users should not be able to insert invalid data into the decentralised web application.

Chapter 2

Background

In order to gain an understanding of the past and current research being made into fields related to the topic, underlying Ethereum Blockchain technology, current supply chains, and similar systems have been studied.

2.1 Related Work

2.1.1 Ethereum Whitepaper (Buterin, 2013)

The Ethereum Blockchain is an immutable group of records. Immutability is achieved using a hashing algorithm calculated using double-SHA256 algorithm; each block/transaction on the blockchain is given a unique hash value. The hash value provides immutability as every block on the blockchain stores the hash of the previous block. Therefore if the hash of the block were to change an effect would be seen throughout.

Vitalik Buterin and Gavin Wooisn co-founded Ethereum after Bitcoin inspired Vitalik. Vitalik wished for Ethereum to have a broader range of use cases than Bitcoin, which only has use cases in the financial sector. Ethereum attempts this by allowing users to create and host smart contracts on the Ethereum Virtual Network, made possible by the semi Turing complete nature of the EVM. Smart Contracts specify conditions that are to be met for a transaction to occur on the Ethereum Blockchain. Transactions can be monetary or solely data. Due to the decentralised nature of smart contracts and the Ethereum Blockchain as a whole, third parties are not required during a transaction.

A user with an EOA (externally owned account) on the blockchain can interface with smart contracts that are distinguishable by its contract account address. An EOA can send a transaction to another EOA or a contract account. Each transaction consists of a receiver address, amount of ether, data, gas price and gas limit. Gas is a method Ethereum uses to measure how much computational power is required to complete a transaction. The sender of a transaction is required to pay the gas in ether and can specify the maximum gas that they are willing to spend, which is called gas limit. Gas price is the value that the sender pays for every computation. Gas prices were taken into account when writing smart contracts for the system. During the creation of the systems, smart contracts specific focus was placed on not including unnecessary values in the 'Struct' data types (Solidity, 2019). Storing too much data can lead to a smart contract being infeasible due to the cost of each transaction.

2.2 Related Systems

Previous attempts at creating transparency within the food supply chains have been made and implemented with some success.

2.2.1 IBM Food Trust (IBM, n.d.)

IBM has implemented a smart contract solution to create transparency between entities in the supply chain. On the contrary, the system created for this dissertation aims to serve the customers by providing publically available data on products within the supply chain. The hope is that by serving the customer, it will indirectly benefit the entities within the organic supply chain due to factors such as increased sales.

IBM Food Trust can give the power of privacy to entities on the supply chain as it runs a Blockchain called the hyper ledger (Hyperledger, 2016). IBM hopes that giving private companies the ability to track the movement of products through supply chains will simplify the process of recalling products. In the past, if there were safety issues with produce, everything had to recalled meaning a significant loss of profit. In some cases, it has taken two weeks to trace issues with produce back to their origin, such as with the E. coli Spinach outbreak in 2006 (Cohn & Yiannas, 2017).

IBM provides a demo (*figure 1*) in which the individual products that make up a fruit and nut bar are listed. Information presented by IBM presents a detailed map of a products data, listing all warehouses, primary producers, manufacturers, distributors and stores the products that make up a fruit and nut bar has passed through.

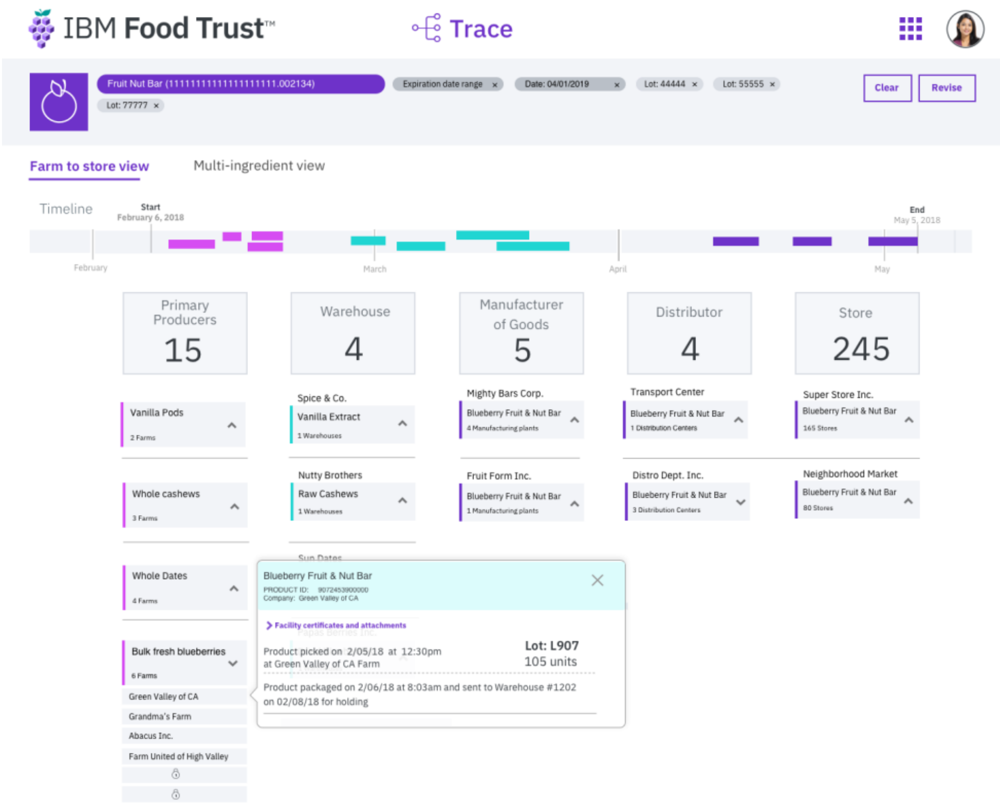


*Figure 1: IBM Food Trust Demo (*image supplied by **(IBM, n.d.)**)

The system created for this dissertation is naturally not as detailed as the IBM Food Trust. However, IBM Food Trusts data is quite tricky to understand, requiring some level of training for its users. Unlike my system, IBM Food Trust has a composite product feature fully implemented (*figure 1*).

2.2.2 Provenance (Provenance, 2020)

Provenance is a system with the slogan "Every product has a story" (Provenance, 2020) that aims to create transparency in the supply chain to better "the wellbeing of people, animals and communities." (Provenance, 2020).

Provenance attempts to solve more than just transparency within the organic food supply chain; *figure 2* displays the business claims of 'Rebel Kitchen'. Business claims also include 'Female Owned Business', 'Fair Payment', 'Handmade' and many more. The blockchain can validate business claims made by 'Rebel Kitchen' as seen in *figure 2*. Business claims in the system created in this dissertation are only organic certifications, as it currently stands the default business claims are ‘FAWL’, ‘Welsh Organic Scheme’ and ‘Dairy Assurance’.

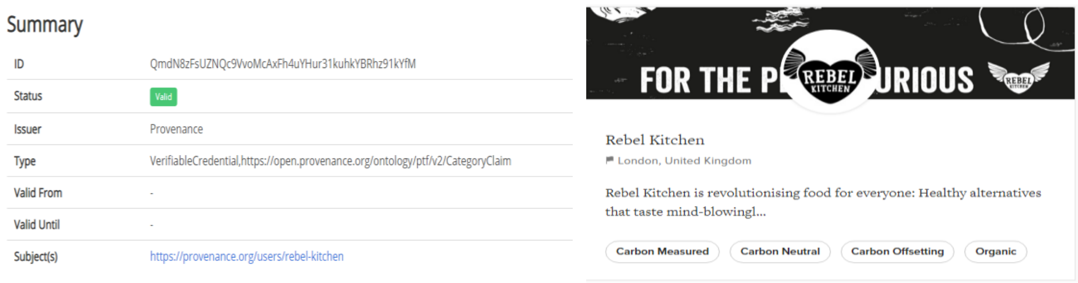


*Figure 2: Carbon Measured Blockchain and Rebel Kitchen Business Claims (*image supplied by **(Provenance, 2020)**)

Product tracking in Provenance works in a similar way to my system. Each product has an ID that can be inserted into a web portal (*figure 3*).

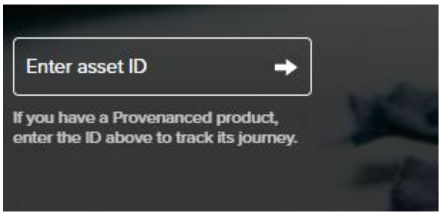


*Figure 3: Asset ID Text Box (*image supplied by **(Provenance, 2020)**)

Once entered, the user will be taken to a screen that displays basic information about the product and the journey it has taken thus far through the supply chain (*figure 4*). Each transaction of a product is verifiable through the blockchain.

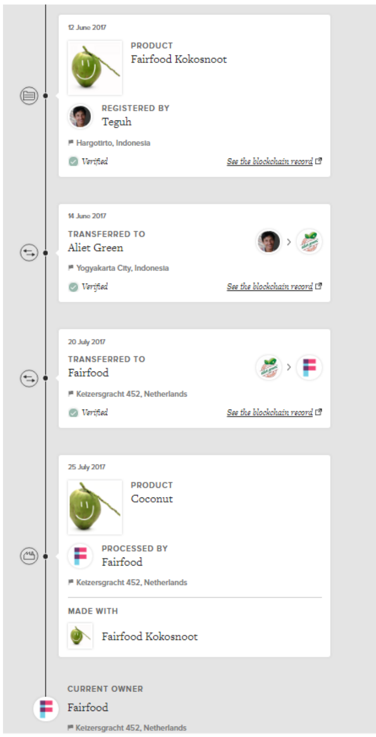


*Figure 4: Provenance Product Journey (*image supplied by **(Provenance, 2020)**)

Provenance and the system being created in this system share technological similarities as they are both hosted on the Ethereum Blockchain. However, the loading times of the information and journey of a coconut on Provenance takes approximately 15 seconds while my system takes <1 second. Factors that could affect the time may be the more substantial amount of data that Provenance pulls from the blockchain and the fact that I am only running my system on a test network.

Provenance has an attractive user interface, including profile images for entities and products in the supply chain. The user interface of the system developed in this dissertation is minimalistic and does not include any images.

Chapter 3

Project Specification

Before creating the system, requirements were outlined; these requirements would ensure that the project meets the aims previously outlined.

3.1 Features

3.1.1 Requirements

- Keep track of products throughout the supply chain by placing data on the blockchain.
  - A farmer harvests a product, gives it a type (Livestock or Produce), weight and accompanying organic information. Organic information will be dependant on whether the product is of type Livestock or Produce.
  - A farmer sends a specific amount of a product to a processor.
  - On arrival of the product the processor states that the product has arrived.
  - The processor can then pass a certain amount of the product onwards to a retailer in a similar fashion to how the farmer sent the product to the processor. On sending the product it is allocated a unique label id.
  - A retailer can then receive a product inserting the unique label id into the decentralised web app.

- All individual products within a retailer will have a label id that a customer can insert into the distributed web app. The label Id provides the user with the name, weight, organic certifications and properties that the farmer holds. Additionally, the path that the product has taken through the supply chain will be displayed.
  - Farmer data field will display:
    - Time/Date harvested
    - Time/Date sent
    - Weight harvested
    - Weight sent
    - Farmer address and name
  - Processor data field will display
    - Time/Date received
    - Time/Date sent
    - Weight received

        - Weight sent
        - Processor address and name
    - Retailer data field will display
        - Time/Date received
        - Weigh received
        - Retailer address and name

- Store data about farmers on the blockchain.
    - Address on blockchain
    - Name
    - Their organic certifications
    - Inventory

- Store data about processors on the blockchain.
    - Address on blockchain
    - Name
    - Inventory

- Store data about retailers on the blockchain.
    - Address on blockchain
    - Name
    - Inventory

- Store data about governing bodies which distribute organic certifications to farmers on the blockchain.
    - Name
    - Organic certification belonging to the governing body.
    - Address on Blockchain

- Store data about livestock on the blockchain.
    - Name
    - Amount of time they spend outside per day
    - What they are fed
    - Where they are kept while indoors

- Store data about produce on the blockchain.
    - Name
    - Pesticides used
    - Fertilisers used

- All code written must be readable.
    - Comment all code
    - Use constant code practices within each language used.

- All data inserted into the web application will be validated.
    - Farmer should not be able to give themselves an organic certification.
    - All entities should not be able to send a weight of a product that they do not currently have in their inventory.
    - All entities should not be able to receive a product that was not sent to them.
    - A single Ethereum address can make only one of each entity.

- Composite products can be created by combining individual raw products.
    - Display all products within the composite product.
    - Display percentage of products within the composite product that is organic.

3.2 Languages

Solidity (Solidity, 2019) is a high-level object-oriented programming language for creating smart contracts for the Ethereum blockchain. The Solidity compiler compiles high-level Solidity code down to Ethereum bytecode which can be run on the Ethereum Virtual Machine. Solidity provides utilities that helped in the creation of the system, such as events. On emit, a record will be created on the blockchain. An event accepts a user-specified number of parameters. Parameters in an event can be indexed and parameters that are indexed can be used to search for a specific event. Solidity is one of the most popular languages for blockchain programming with 9.5% of all blockchain searches on stack overflow referencing Solidity compared to the next popular being JavaScript with 4.8% (Mix, 2019). Theoretically, Solidity is heavily documented, therefore decreasing development time. The system is running on Solidity v0.4.20; this is not the most up to date stable version as issues arose while attempting to communicate with Web3.js using a version higher than v0.4.20.

JavaScript (MDN contributors, 2020) is a client-side object orientated scripting language with the abilities to create dynamic web pages.

JQuery (JQuery, 2020) is a Javascript library which was used to manipulate the data that is received from both smart contracts and user inputs. JQuery is lightweight and easy to learn. The system is running JQuery 3.4.1, which is the most up to date stable version.

HTML (w3schools, n.d.) is a mark-up language used to define how data will be displayed on a user's monitor within their browser. The system is running HTML 5, which is the most up to date stable version.

CSS (MDN, 2020) defines how HTML tags will be displayed on the user's screen.

3.3 Frameworks

Bootstrap (Bootstrap, 2020) is a framework that provides CSS templates. In the system, Bootstrap is used to make an aesthetically pleasing graphical user interface for the decentralised application. Bootstrap saved a significant amount of time during the front-end development phase as it meant a custom CSS did not have to be written for the system. The system runs Bootstrap v4.4.1, which is the most up to date stable version.

Web3.js (Web3js, 2020) is a framework that allows for interaction between Javascript and Ethereum Smart Contracts. The system currently runs on web3 v1.2.6 which is the most up to date stable version.

Truffle (Truffle, 2020) is a framework that makes the development of smart contracts easier. The system is currently being run on the test network that truffle provides. The system is currently running v6.0.3 the most up to date stable version.

3.4 Development Tools

Remix (Ethereum, 2019) is a browser-based IDE used to write and deploy smart contracts on to the Ethereum blockchain.

Atom (Atom, 2020) is a customisable text editor with Git capabilities. Atom was used to write the HTML, JQuery, and CSS for the decentralised web app. The Git capabilities streamlined the development process, and as a result, there was no need to switch between several windows.

Git (Git, n.d.) is a version control system which allows small to large groups of people to work on the same project simultaneously. It also allows forking which benefited the development of the system allowing for confident experimentation with new features before committing to the master branch. Rollbacks also played a large part in the development process as it was possible to return to past versions of the system.

GitHub (Github Inc, 2020) is a web hosting service for the git vcs, allowing access to project files from any computer. Furthermore, GitHub creates an external backup of the system, and if hard drives computer were to fail, work would not be lost.

Npm (npmjs, 2020) is a packet manager tool which was used in the system to install and manage Web3.js and Bootstrap dependencies. The system is currently running v6.14.4 which is the most up to date stable version.

Chapter 4

Implementation

Divided into four sections; Smart Contracts, JQuery, Front-end and Testing, this chapter will present and explain the system created.

The first three sections are implemented following the MVC (Model View Controller) framework. The MVC framework is a cycle of a user making a request to the view. The view then interfaces with the controller that in turn, interfaces with the model to retrieve data from the blockchain. The model then provides the data via the controller and view. MVC provides modularity that can lead to readable and maintainable code. (Microsoft, 2020)

4.1 Smart Contracts

This section is a compilation of all the smart contracts; there are three smart contracts Farm.sol, Processor.sol and Retailer.sol.

The smart contracts have three leading roles; defining how data will be stored on the blockchain, verification, validation and retrieving data from the blockchain. In terms of the MVC framework, the smart contracts would be considered the models.

The .sol extension is the solidity smart contract extension.

4.1.1 Farm.sol

This smart contract deals with the creation and storage of farmers on the blockchain. Each farmer can harvest produce, harvest livestock and send a product. Furthermore, this contract deals with the storage of governing bodies on the blockchain and their distribution of organic certificates to farmers.

```
pragma solidity ^0.4.20;

/*
* This smart contract hndles farmer data and the products they have in their inventory.
* It also stores organic properties and governing bodies that distribute certificates.
*/

contract Farm {

    uint numberFarmers = 0;

    //Structure defining a product
    struct Product {
        uint productId;
        string typeStr; //type is either Livestock or Produce
        string name;
        uint weight; //weight in grams
        uint8 allocated; //0 = not allocated, 1 = allocated
        address farmerAddress; //owner of product

    }

    mapping (uint => Product) products; //all products stored here with productId as index
    Product[] productsArray;

    //Structure defining organic properties for produce
    struct OrganicPropertiesProduce {
        uint productId;
        string pesticideUsed;
        string fertilisersUsed;
    }

    mapping (uint => OrganicPropertiesProduce) organicPropertiesProduce; //organic properties for produce stored here indexed using a counter

    //Structure defining organic properties for Livestock
    struct OrganicPropertiesLivestock {
        uint productId;
        uint timeSpentOutdoors; //measured in minutes
        string feedUsed;
        string housing;
    }

    mapping (uint => OrganicPropertiesLivestock) organicPropertiesLivestock; //organic properties for livestock stored here indexed using a counter

    uint numberOfProducts = 0; //number of products in the ecosystem
```

Figure 5: Products and Organic Certifications

Figure 5 presents how all the data related to the products are stored on the blockchain. All product data is stored using three structs which are 'Product', 'OrganicPropertiesProduce' and 'OrganicPropertiesLivestock'. The struct ‘Product’ stores basic information about a product and has a primary key 'productId' which is used as a foreign key within 'OrganicPropertiesProduce' and 'OrganicPropertiesLivestock' depending on the product type.

```
//Structure defining a farmer
struct Farm {
    address farmer;
    string name;
    uint[] certifications; //stores id of certifications that the farmer owns
    uint8 allocated; //0 = not allocated, 1 = allocated
}

mapping (address => Farm) farmers; //farmers stored here indexed using the farmers address
address[] farmersLUT; //array of farmer addresses

//Defining governing body which is used to distribute certifications
struct GoverningBody {
    address governingAddress;
    string name;
    uint[] certificationIds;
    uint allocated; //0 = not allocated, 1 = allocated
}

mapping (address => GoverningBody) governingBodies; //governing bodies stored here indexed using address
uint numberOfGoverningBodies = 0;

//Structure defining a certificate
struct Certification {
    uint certificationId;
    string name;
}

mapping (uint => Certification) certifications; //certifications stored here using a numberOfCertifications as a index

uint numberOfCertifications = 0;

event CreateFarmer(string name);
event Harvest(uint indexed _productId, string _name, uint weight, address indexed _farmerAddress, uint256 time);
event SendProduct(address _farmerAddress, address _reciever, uint indexed _productId, uint weight, uint256 time);
```

Figure 6: Farmers and Governing Bodies

Figure 6 presents three structs' Farm', 'GoverningBody' and 'Certification'. The ‘Farm’ struct stores address, name, certifications and allocated status. The id’s of a certificate received by a farmer are stored in the certification array. The name correlating to each certification id can be found within the ‘certifications’ mapping.

Three events are defined (*event explanation found in chapter 3.2*) both 'Harvest' and 'SendProduct' are called in *figure 30* for tracking purposes. They are both indexed by '_productId' so that information about the product can be deduced from the '_productId'. A benefit of this is that it is not necessary to store all the data about a product when it passed through the supply chain thereby saving gas.

```
constructor() public {
    //Hard codes a Governing Body Quality Welsh Food Certification Ltd (GB-ORG-13)
    certifications[0] = Certification(numberOfCertifications++, "FAWL");
    certifications[1] = Certification(numberOfCertifications++, "Welsh Organic Scheme");
    certifications[2] = Certification(numberOfCertifications++, "Dairy Assurance");
    uint[] memory certificationTMP;
    governingBodies[msg.sender] = GoverningBody(msg.sender, "Quality Welsh Food Certification Ltd", certificationTMP, 1);
    governingBodies[msg.sender].certificationIds.push(certifications[0].certificationId);
    governingBodies[msg.sender].certificationIds.push(certifications[1].certificationId);
    governingBodies[msg.sender].certificationIds.push(certifications[2].certificationId);

    numberOfGoverningBodies = numberOfGoverningBodies + 1;
}
```

Figure 7: Governing Body Constructor

On the deployment of Farm.sol the constructor (*figure 7*) is called, hard coding three certifications and allocating them to a governing body. The governing body's address is determined by the address of the account that deployed the contract to the blockchain.

```
function createFarmer(string memory _name) public {
    require (farmers[msg.sender].allocated == 0, "Already exists");
    uint[] certifications;
    farmersLUT.push(msg.sender); //add current address of message sender to the farmers look up table
    farmers[msg.sender] = Farm(msg.sender, _name, certifications, 1); //initalise farmer object and store in farmers array
    numberFarmers = numberFarmers + 1;
    emit CreateFarmer(_name);
}

function harvestLivestock(string memory _typeStr, string memory _name, uint _weight, uint _timeSpentOutdoors, string _feedUsed, string _housing) public {
    require (farmers[msg.sender].allocated == 1, "Farmer doesn't exist");
    require (products[numberOfProducts].allocated == 0, "Product already exists");
    products[numberOfProducts] = Product(numberOfProducts, _typeStr, _name, _weight, 1, msg.sender); //initialise product object and store in products array
    organicPropertiesLivestock[numberOfProducts] = OrganicPropertiesLivestock(numberOfProducts, _timeSpentOutdoors, _feedUsed, _housing); //initialise organic livesotock properties for the current product
    numberOfProducts = numberOfProducts + 1;
    emit Harvest(numberOfProducts, _name, _weight, msg.sender, block.timestamp);
}

function harvestProduce(string memory _typeStr, string memory _name, uint _weight, string _pesticideUsed, string _fertiliserUsed) public {
    require (farmers[msg.sender].allocated == 1, "Farmer doesn't exist");
    require (products[numberOfProducts].allocated == 0, "Product already exists");
    products[numberOfProducts] = Product(numberOfProducts, _typeStr, _name, _weight, 1, msg.sender); //initialise product object and store in products array
    organicPropertiesProduce[numberOfProducts] = OrganicPropertiesProduce(numberOfProducts, _pesticideUsed, _fertiliserUsed); //initialise organic produce properties for the current product
    numberOfProducts = numberOfProducts + 1;
    emit Harvest(numberOfProducts, _name, _weight, msg.sender, block.timestamp);
}

function sendProduct(address _reciever, uint _productId, uint weight) public {
    require (farmers[msg.sender].allocated == 1, "Farmer doesn't exist");
    require (products[_productId].weight > 0, "Not in stock");
    require (weight <= products[_productId].weight, "This is not a valid weight");
    products[_productId].weight = products[_productId].weight - weight; //minus amount of weight being sent from current weight for specified product
    emit SendProduct(msg.sender, _reciever, _productId, weight, block.timestamp);
}
```

Figure 8: Farmer Abilities

Figure 8 displays all the abilities of a farmer.

The function ‘createFarmer’ ensures that a message senders EOA does not already have a farmer account. An empty array of certification is initialised. The EOA is placed into the lookup table of all farmers. Farm struct is initialised storing it in the farmers mapping, indexed by EOA. The lookup table is helpful as it is less computationally expensive to iterate over compared to a mapping of farmers that is indexed by address.

The functions ‘harvestLivestock’ and ‘havestProduce’ ensures that a farmer exists and that the product id is not already allocated, ensuring that each new harvest product gets a unique product id. Organic properties are then allocated using either the

‘OrganicPropertiesProduce’ or ‘OrganicPropertiesLivestock’ and stored in their respective mappings *(figure 5)*. Next ‘numOfProducts’ is incremented by one, this will be the id of the next product.

The ‘sendProduct’ function ensures that a sender exists, the weight of the product being sent is greater than zero and sending weight is less than or equal to current weight. The amount of weight being sent is then subtracted from the current weight and stored.

```
function setFarmCertification(address _farmerAddress, uint _certificationId) public {
    require (governingBodies[msg.sender].allocated == 1, "Not a governing body");
    bool owned = false;
    for (uint i = 0; i < governingBodies[msg.sender].certificationIds.length; i++) {
        if (_certificationId == governingBodies[msg.sender].certificationIds[i]) { //
            owned = true; //when the certificates that match are found owned is set to true
        }
    }
    require (owned == true, "Governing body does not own this certification");
    farmers[_farmerAddress].certifications.push(_certificationId); //add certification to array of certifications
}
```

Figure 9: Set Farmer Certifications

Figure 9 is the function that will be called when a governing body wishes to give a farmer an organic certification. The first line of code ensures that the message sender is a governing body. Following that, it ensures that the certification belongs to the governing body. Finally adding a certification id to the farmers' certification id array.

```
//farmer getters ----------------------------------------------------------------
function getFarmerName(address _farmerAddress) external view returns (string memory) {
    return farmers[_farmerAddress].name;
}

function getFarmerAllocated(address _farmerAddress) external view returns (uint) {
    return farmers[_farmerAddress].allocated;
}

function getNumberOfFarmers() external view returns (uint) {
    return farmersLUT.length;
}

function getFarmerAddress(uint _count) external view returns (address) {
    return farmersLUT[_count];
}

function getCertificationArray(address _farmerAddress) external view returns (uint[]) {
    return farmers[_farmerAddress].certifications;
}
//farmer getters end ----------------------------------------------------------------
```

```
//product getters ----------------------------------------------------------------
function getProductTypeStr(uint _id) external view returns (string memory) {
    return products[_id].typeStr;
}

function getProductName(uint _id) external view returns (string memory) {
    return products[_id].name;
}

function getProductFarmerAddress(uint _id) external view returns (address) {
    return products[_id].farmerAddress;
}

function getProductWeight(uint _id) external view returns (uint) {
    return products[_id].weight;
}

function getAllocated(uint _id) external view returns (uint) {
    return products[_id].allocated;
}

function getNumberOfProducts()  external view returns (uint) {
    return numberOfProducts;
}
//product getters end ----------------------------------------------------------------
```

Figure 10: Farmer Entity Getters

Figure 11: Farmer Product Getters

```
//organic properties getters
function getTimeSpentOutdoors(uint _id) external view returns (uint) {
    return organicPropertiesLivestock[_id].timeSpentOutdoors;
}

function getFeedUsed(uint _id) external view returns (string memory) {
    return organicPropertiesLivestock[_id].feedUsed;
}

function getHousing(uint _id) external view returns (string memory) {
    return organicPropertiesLivestock[_id].housing;
}

function getPesticidesUsed(uint _id) external view returns (string memeory) {
    return organicPropertiesProduce[_id].pesticideUsed;
}

function getFertilisersUsed(uint _id) external view returns (string memeory) {
    return organicPropertiesProduce[_id].fertilisersUsed;
}
//organic properites getters end ------------------------------------------------
```

```
//governing bodies getters ----------------------------------------------------------
function getGoverningBodyName(address _governingBodyAddress) public view returns (string memory) {
    return governingBodies[_governingBodyAddress].name;
}

function getCertificateName(uint _certificationId) public view returns (string memory) {
    return certifications[_certificationId].name;
}

function getGoverningBodiesCertificates(address _governingBodyAddress) public view returns (uint[]) {
    return governingBodies[_governingBodyAddress].certificationIds;
}
//governing bodies getters ----------------------------------------------------------
```

Figure 12: Organic Properties Getters

Figure 13: Governing Body Getters

4.1.2 Processor.sol

This smart contract deals with the creation and storage of processors on the blockchain. Each processor can receive and send a product.

```
pragma solidity ^0.4.20;

/*
* This smart contract handles processors data and the products they have in their inventory.
*
*/

contract Processor {

    //Structure defining a product
    struct Product {
        uint productId;
        uint weight;
        uint8 allocated; //0 = not allocated, 1 = allocated
        address processorAddress; //owner of product
    }

    mapping (uint => Product) products; //all products stored here with productId as index

    uint latestProductId = 0;
    uint label = 0;

    //Structure defining a processor
    struct Processor {
        address processor;
        string name;
        uint8 allocated;
    }

    mapping (address => Processor) processors; //all processors stored here with address as index
    address[] processorsLUT; //array of processor addresses
```

Figure 14: Product and Processor

In *figure 14* ‘Product’ struct contains the ‘productId’ which acts as a foreign key for a product harvested by a farmer meaning there is no need to store ‘name’, ‘farmerAddress’ and ‘typeStr’ multiple times. Much like in *figure 6,* a product is stored in a mapping. The ‘latestProductId’ variable is the product id of the last product received by the processors. The ‘label’ variable is the label id of the next product being sent by a processor, ensuring that each label is unique.

The ‘Processor’ struct works the same as a ‘Farm’ struct in *figure 6,* although ‘Processor’ does not have any certifications.

```
event CreateProcessor(string _name);
event ProductRecieved(address _sender, address _reciever, uint indexed _productId, uint _weight, uint256 time);
event ProductSent(address indexed _sender, address indexed _reciever, uint indexed _label, uint _productId, uint _weight, uint256 time);

function createProcessor(string memory _name) public {
    require (processors[msg.sender].allocated == 0, "Already exists");
    processorsLUT.push(msg.sender); //add current address of message sender to the processors look up table
    processors[msg.sender] = Processor(msg.sender, _name, 1); //initalise procesor object and store in farmers array
    emit CreateProcessor(_name);
}

function recieveProduct(address _sender, uint _productId, uint _weight) public {
    require (processors[msg.sender].allocated == 1, "Processor doesn't exist");
    //If product doesn't exist create new product otherwise add weight to already existing product
    if (products[_productId].allocated == 0) {
        products[_productId] = Product(_productId, _weight, 1, msg.sender);
        latestProductId = _productId;
    } else {
        products[_productId].weight = products[_productId].weight + _weight;
    }
    emit ProductRecieved(_sender, msg.sender, _productId, _weight, block.timestamp);

}

function sendProduct(address _reciever, uint _productId, uint _weight) public {
    require (processors[msg.sender].allocated == 1, "Processor doesn't exist");
    require (products[_productId].weight > 0, "Not in stock");
    products[_productId].weight = products[_productId].weight - _weight; //removes weight sent from current weight
    label = label + 1; //iterates the label so that eaxh label is unique
    emit ProductSent(msg.sender, _reciever, label, _productId, _weight, block.timestamp);
}
```

Figure 15: Events and Processor Abilities

Figure 15's ‘createProcessor’ and ‘sendProduct’ functions work in the same manner as *figure 8's* ‘createFarmer’ and ‘sendProduct’ functions.

The ‘receiveProduct’ function ensures that the EOA has a processor account. When a product is not allocated, a new product is added to the ‘products’ mapping, and ‘latestProductId’ is updated. When the product received is already owned by the processor, the received weight is added to the current weight.

```
//Product Getters-------------------------------------------------------------
function getProductProcessorName(address _processorAddress) external view returns (string memory) {
    return processors[_processorAddress].name;
}

function getLatestProductId()  external view returns (uint) {
    return latestProductId;
}

function getLatestLabel()  external view returns (uint) {
    return label;
}

function getProductProcessorAddress(uint _id) external view returns (address) {
    return products[_id].processorAddress;
}

function getProductWeight(uint _id) external view returns (uint) {
    return products[_id].weight;
}

function getAllocated(uint _id) external view returns (uint) {
    return products[_id].allocated;
}
//End Product Getters---------------------------------------------------------
```

```
//Processor Getterss----------------------------------------------------------
function getProcessorAllocated(address _address) external view returns (uint) {
    return processors[_address].allocated;
}

function getNumberOfProcessors() external view returns (uint) {
    return processorsLUT.length;
}

function getProcessorAddress(uint _count) external view returns (address) {
    return processorsLUT[_count];
}

function getProcessorName(address _processorAddress) external view returns (string) {
    return processors[_processorAddress].name;
}
//End Processor Getters-------------------------------------------------------
```

Figure 16: Processor Product Getters

Figure 17: Processor Entity Getters

4.1.3 Retailer.sol

This smart contract deals with the creation and storage of retailers on the blockchain. Each retailer can receive a product.

```
pragma solidity ^0.4.20;

/*
* This smart contract handles retailers data and the products they have in their inventory.
*
*/

contract Retailer {

    //Structure defining a product
    struct Product {
        uint productId;
        uint label; //customer will use the label to track their item
        uint weight;
        uint8 allocated; //0 = not allocated, 1 = allocated
        address retailerAddress; //owner of product
    }

    uint latestLabel = 0;
    mapping (uint => Product) products;  //all products stored here with label as index

    //Structure defining a retailer
    struct Retailer {
        address retailer;
        string name;
        uint8 allocated; //all products stored here with productId as index
    }

    mapping (address => Retailer) retailers; //all retailers stored here with address as index
    address[] retailersLUT; //array of retailer addresses

    event CreateRetailer(string name);
    event ProductRecieved(address _sender, address _reciever, uint indexed _label, uint weight, uint256 time);
```

Figure 18: Retailer Structs

Figure 18 acts in the same way as *figure 14*.

```
    function createRetailer(string memory _name) public {
        require (retailers[msg.sender].allocated == 0, "Already exists");
        retailersLUT.push(msg.sender);  //add current address of message sender to the retailer look up table
        retailers[msg.sender] = Retailer(msg.sender, _name, 1); //initalise retailer object and store in farmers array
        emit CreateRetailer(_name);
    }

    function recieveProduct(address _sender, uint _productId, uint _label, uint _weight) public {
        require (retailers[msg.sender].allocated == 1, "Retailer must exist");
        require (products[_label].allocated == 0, "Product already exists");
        products[_label] = Product(_productId, _label, _weight, 1, _sender); //crate new product object and store in products mapping
        latestLabel = _label;
        emit ProductRecieved(_sender, msg.sender, _label, _weight, block.timestamp);
    }
```

Figure 19: Retailer Abilities

Figure 19 acts in the same way as *figure 15*.

```
    //product getters ------------------------------------------------------------------------
    function getLatestLabel() external view returns (uint) {
        return latestLabel;
    }

    function getProductWeight(uint _label) external view returns (uint) {
        return products[_label].weight;
    }

    function getAllocated(uint _label) external view returns (uint) {
        return products[_label].allocated;
    }

    function getProductRetailerAddress(uint _label) external view returns (address) {
        return products[_label].retailerAddress;
    }

    function getProductId(uint _label) external view returns (uint) {
        return products[_label].productId;
    }
    //end product getters ------------------------------------------------------------------------
```

Figure 20: Retailer Product Getters

```
    //retailers getters ------------------------------------------------------------------------
    function getRetailerName(address _address) external view returns (string memory) {
        return retailers[_address].name;
    }

    function getRetailerAllocated(address _address) external view returns (uint) {
        return retailers[_address].allocated;
    }

    function getNumberOfRetailers() external view returns (uint) {
        return retailersLUT.length;
    }

    function getRetailerAddress(uint _count) external view returns (address) {
        return retailersLUT[_count];
    }
    //end retailer getters ------------------------------------------------------------------------


}
```

Figure 21: Retailer Getters

4.2 JQuery Controllers

In the MVC framework, the JQuery code acts as a controller.

4.2.1 contract.js

Contract.js handles the creation of all entities, the passing of a product from entity to entity and displaying products in inventories.

```
/*
* Description: This js file interacts with both the html files and
* solidity smart contracts to pass data through the supply chain.
*/
App = {
  contracts: {},

  load: async () => {
    await App.loadWeb3() //imports web3 into the file
    await App.loadAccount() //sets default account
    await App.loadAllContracts() //load all contracts from the blockchain
    document.getElementById("current_address").innerHTML = App.account; //parses current address to html
  },

  loadWeb3: async() => {
    if (typeof web3 !== 'undefined') {
      web3 = new Web3(web3.currentProvider);
    } else {
      web3 = new Web3(new Web3.providers.HttpProvider("http://localhost:8545"));
    }
  },

  loadAccount: async() => {
      const accounts = await web3.eth.getAccounts(); //returns array of accounts
      App.account = accounts[6]; //select a singular array
  },

  loadAllContracts: async() => {
    await App.loadFarmerContract()
    await App.loadProcessorContract()
    await App.loadRetailerContract()
  },

  loadFarmerContract: async() => {
    const farmAbi = [⋯}] //defines functions within farm contract
    const farmAddress = '0xf694C0A7293A92d23FAc8f1f3662983b647b776c' //contract address
    App.farmContract = new web3.eth.Contract(farmAbi, farmAddress);
  },

  loadProcessorContract: async() => {
    const processorAbi = [⋯] //defines functions within processor contract
    const processorAddress = '0x277038B5C99e4454d08Fca1198012317369F7013' //contract address
    App.processorContract = new web3.eth.Contract(processorAbi, processorAddress);
  },

  loadRetailerContract: async() => {
    const retailerAbi = [⋯}] //defines functions within retailer contract
    const retailerAddress = '0x445389308222C0A145D3deD8eB46269da5dA9507' //contract address
    App.retailerContract = new web3.eth.Contract(retailerAbi, retailerAddress);
  },
```

Figure 22: Load Web3 and All Contracts

Figure 22 shows the code that loads Web3.js for interfacing between the JQuery code and the smart contracts. Using web3 functions, all three contracts are called from the test network using their application binary interface and the contract account address. The application binary interface is a JSON definition of a smart contract. An example of an application binary interface can be seen below.

```
"constant": true,
"inputs": [],
"name": "getNumberOfFarmers",
"outputs": [
        {
                "name": "",
                "type": "uint256"
        }
],
"payable": false,
"stateMutability": "view",
"type": "function"
},
```

Figure 23: ABI for getNumberOfFarmers

```
createFarmer: async (data) => {
  let name = $('[name="farmer_name"]').val();
  const allocated = await App.farmContract.methods.getFarmerAllocated(App.account).call();
  if (name == "") {
    alert('name field can not be empty');
  } else {
    //if farmrer isn't currently allocated create new farmer otherise throw alert
    if (allocated == "0") {
      await App.farmContract.methods.createFarmer(name).send({from: App.account, gasLimit: 4712388});
      alert('created farmer');
    } else {
      alert('farmer already exists');
    }
  }
},

createProcessor: async (data) => {
  let name = $('[name="processor_name"]').val();
  const allocated = await App.processorContract.methods.getProcessorAllocated(App.account).call();
  if (name == "") {
    alert('name field can not be empty');
  } else {
    //if processor isn't currently allocated create new farmer otherise throw alert
    if (allocated == "0") {
      await App.processorContract.methods.createProcessor(name).send({from: App.account, gasLimit: 4712388});
      alert('created processor');
    } else {
      alert('processor already exists');
    }
  }
},

createRetailer: async (data) => {
  let name = $('[name="retailer_name"]').val();
  const allocated = await App.retailerContract.methods.getRetailerAllocated(App.account).call();
  if (name == "") {
    alert('name field can not be empty');
  } else {
  //if retailer isn't currently allocated create new farmer otherise throw alert
    if (allocated == "0") {
      await App.retailerContract.methods.createRetailer(name).send({from: App.account, gasLimit: 4712388});
      alert('created retailer');
    } else {
      alert('retailer already exists');
    }
  }
},
```

Figure 24: Create New Farmer, Processor and Retailer

Figure 24 handles the creation of entities within the supply chain, ensuring that they are not already allocated.

```javascript
harvestProduce: async (data) => {
  //read all values inputted by user in the farmer_homepage.html form
  let name = $('[name="produce_name"]').val();
  let weight = $('[name="produce_weight"]').val();
  let pesticides_used = $('[name="pesticides_used"]').val();
  let fertilisers_used = $('[name="fertilisers_used"]').val();
  var weight_int = parseInt(weight, 10);
  //check if input is valid, if input is valid add product to farmer inventory
  if (name == "" || weight == "") {
    alert('weight or name is empty');
  } else if (isNaN(weight_int)) {
    alert('weight is not a int');
  } else {
    await App.farmContract.methods.harvestProduce('Produce', name, weight_int, pesticides_used, fertilisers_used).send({from: App.account, gasLimit: 4712388});
    alert('harvested');
  }

},

harvestLivestock: async (data) => {
  //read all values inputted by user in the farmer_homepage.html form
  let name = $('[name="livestock_name"]').val();
  let weight = $('[name="livestock_weight"]').val();
  let time_spent_outdoors = $('[name="time_spent_outdoors"]').val();
  let feed_used = $('[name="feed_used"]').val();
  let housing = $('[name="housing"]').val();
  var weight_int = parseInt(weight, 10);
  var time_spent_outdoors_int = parseInt(time_spent_outdoors, 10);
  //check if input is valid, if input is valid add product to farmer inventory
  if (name == "" || weight == "") {
    alert('weight or name is empty');
  } else if (isNaN(weight_int)) {
    alert('Please insert a integer in the weight field');
  } else if (isNaN(time_spent_outdoors_int)) {
    alert('Please insert a integer in the time spent ourdoors field');
  } else {
    await App.farmContract.methods.harvestLivestock('Livestock', name, weight_int, time_spent_outdoors_int, feed_used, housing).send({from: App.account, gasLimit: 4712388});
    alert('harvested');
  }
},
```

Figure 25: Harvest

Figure 25 handles the harvesting of livestock and produce. Dependant on the type of product harvested different functions; 'harvestLivestock' or 'harvestProduce' will be called. Both functions accept different organic properties. They both then go onto validate the input. If the input passes validation, smart contracts methods are called to add the product to the farmers' inventory.

```javascript
sendProductFarm: async (data) => {
  var items = document.getElementsByClassName("list-group-item active"); //get currently activated list item in farmer_homepage.html
  var product_id = items[0].firstChild.nextElementSibling.innerText.slice(4); //slice first 4 characters giving the product_id
  var product_id_int = parseInt(product_id, 10);
  //read all values inputted by user in the farmer_homepage.html form
  let reciever = $('[name="reciever"]').val();
  let weight = $('[name="send_weight"]').val();
  var weight_int = parseInt(weight, 10);
  //check if input is valid, if input is valid send desired amount of selected product to the reciever
  if ((reciever == ""|| weight == "") || isNaN(weight_int)) {
    alert('insufficient values');
  } else {
    const allocated = await App.processorContract.methods.getProcessorAllocated(reciever).call();
    const inventory_weight = await App.farmContract.methods.getProductWeight(product_id_int).call();
    var inventory_weight_int = parseInt(inventory_weight);
    if (inventory_weight_int < weight_int) {
      alert('insert valid weight');
    } else {
      if (allocated == '0') {
        alert('not allocated');
      } else if (allocated == '1') {
        await App.farmContract.methods.sendProduct(reciever, product_id_int, weight_int).send({from: App.account, gasLimit: 4712388});
        alert('sent');
      }
    }
  }
},
```

Figure 26: Send Product as Farmer

Figure 26 handles the transfer of a product from a farmer to a processor. It takes user input from 'list-group-item' that is active at the time of function call. List group item is created by the 'getHarvestedProducts' (*Figure 27*). The receiver's address and weight are read from the view then validated ensuring none of the fields is empty, and weight is an integer. Furthermore, it ensures that the receiver's address is allocated as a processor in the blockchain.

```
getHarvestedProducts: async (data) => {
  $('#currently_harvested').empty(); //removes all products currently on screen
  //$( ".currentlyHarvested" ).remove();s
  const numberOfProducts = await App.farmContract.methods.getNumberOfProducts().call(); //number of products in the ecosystem
  const currentlyHarvestedTemplate = $('#currently_harvested'); //template that will hold individual harvested products

  //iterate over all products and getting those that have a weight greater than 0 and belong to the current farmer
  for (var i = 0; i < numberOfProducts; i++) {
    var i_int = parseInt(i, 10);
    //get current product info
    const product_weight = await App.farmContract.methods.getProductWeight(i_int).call();
    const product_allocated = await App.farmContract.methods.getAllocated(i_int).call();
    const farmer_address = await App.farmContract.methods.getProductFarmerAddress(i_int).call();

    var product_weight_int = parseInt(product_weight, 10);
    var product_allocated_int = parseInt(product_allocated, 10);
    if (product_weight_int > 0 && farmer_address == App.account) {
      let product_id = i_int;
      let name = await App.farmContract.methods.getProductName(i_int).call();
      let weight = product_weight_int;
      //product values to be output to the screen
      let output =
        `<a href="#" class="list-group-item list-group-item-action" data-toggle="list" role="tab">
          <div class="d-flex w-100 justify-content-between">
            <h5 class="mb-1" name="product_id">ID: `+ product_id +`</h5>
          </div>
            <p class="mb-1" name="product_id">Name: `+ name +`</p>
            <p class="mb-1">Weight: `+ weight +`g</p>
        </a>`;

      currentlyHarvestedTemplate.append(output);

    }
  }
},
```

Figure 27: Get Harvested Products

Figure 27 iterates over all products within the farmer contract and where one is owned by the global account address and the weight is greater than zero. The product_id, name and weight is output.

```
recieveProductProcessor: async (data) => {
  //read all values inputted by user in the processor_homepage.html form
  let product_id = $('[name="product_id"]').val();
  let weight = $('[name="processor_weight"]').val();
  var product_id_int = parseInt(product_id, 10);
  var weight_int = parseInt(weight, 10);
  if ((product_id == ""|| weight == "") || isNaN(weight_int) || isNaN(product_id_int)){
    alert('invalid input');
  } else {
    let sender = await App.farmContract.methods.getProductFarmerAddress(product_id_int).call(); //get sender address
    await App.processorContract.methods.recieveProduct(sender, product_id_int, weight_int).send({from: App.account, gasLimit: 4712388});
    alert('received');
  }
},

sendProductProcessor: async (data) => {
  var items = document.getElementsByClassName("list-group-item active"); //get currently activated list item in processor_homepage.html
  var product_id = items[0].firstChild.nextElementSibling.innerText.slice(4); //slice first 4 characters giving the product_id
  var product_id_int = parseInt(product_id, 10);

  //read all values inputted by user in the processor_homepage.html form
  let reciever = $('[name="reciever"]').val();
  let weight = $('[name="send_weight"]').val();
  var weight_int = parseInt(weight, 10);
  //check if input is valid, if input is valid send desired amount of selected product to the reciever
  if ((reciever == ""|| weight == "") || isNaN(weight_int)) {
    alert('insufficient values');
  } else {
    const allocated = await App.retailerContract.methods.getRetailerAllocated(reciever).call();
    const inventory_weight = await App.processorContract.methods.getProductWeight(product_id_int).call();
    var inventory_weight_int = parseInt(inventory_weight);
    if (inventory_weight_int < weight_int) {
      alert('insert valid weight');
    } else {
      if (allocated == '0') {
        alert('not allocated');
      } else if (allocated == '1') {
        await App.processorContract.methods.sendProduct(reciever, product_id_int, weight_int).send({from: App.account, gasLimit: 4712388});
        const latestProductId = await App.processorContract.methods.getLatestLabel().call();
        alert('sent to label is ' + latestProductId);
      }
     }
   }
},
```

Figure 28: Recieve and Send Product as Processor

Figure 28 deals with both receiving and sending a product as a processor. RecieiveProductProcessor takes product_id and processor_weight and gets the sender's address by querying farm contract using the product_id to find the farmer that is linked to the product_id.

SendProductProcessor works the same as *figure 26.*

```
getProcessorInventory: async (data) => {
  $('#processor_inventory').empty(); //removes all products currently on screen
  const latestProductId = await App.processorContract.methods.getLatestProductId().call(); //number of products in the ecosystem
  const currentlyInInventoryTemplate = $('#processor_inventory'); //template that will hold inventory

  //iterate over all products and getting those that have a weight greater than 0 and belong to the current processor
  for (var i = 0; i <= latestProductId; i++) {
    var i_int = parseInt(i, 10);
    //get current product info
    const product_weight = await App.processorContract.methods.getProductWeight(i_int).call();
    const product_allocated = await App.processorContract.methods.getAllocated(i_int).call();
    const sender_address = await App.farmContract.methods.getProductFarmerAddress(i_int).call();
    const processor_address = await App.processorContract.methods.getProductProcessorAddress(i_int).call();
    var product_weight_int = parseInt(product_weight, 10);
    var product_allocated_int = parseInt(product_allocated, 10);
    var i_int = parseInt(i, 10);
    if (product_weight_int > 0 && processor_address == App.account) {
      let product_id = i_int;
      let name = await App.farmContract.methods.getProductName(i_int).call();
      let weight = product_weight_int;
      //product values to be output to the screen
      let output =
        `<a href="#" class="list-group-item list-group-item-action" data-toggle="list" role="tab">
            <div class="d-flex w-100 justify-content-between">
              <h5 class="mb-1" name="product_id">ID: `+ product_id +`</h5>
            </div>
              <p class="mb-1" name="product_id">Name: `+ name +`</p>
              <p class="mb-1">Weight: `+ weight +`g</p>
          </a>`;

        currentlyInInventoryTemplate.append(output);

    }
  }
},
```

Figure 29: Get Processor Inventory

Figure 29 works the same way as *figure 27.*

```
recieveProductRetailer: async (data) => {
  //read all values inputted by user in the retailer_homepage.html form
  let product_id = $('[name="product_id"]').val();
  let label_id = $('[name="label_id"]').val();
  let weight = $('[name="retailer_weight"]').val();
  var label_id_int = parseInt(label_id, 10);
  var product_id_int = parseInt(product_id, 10);
  var weight_int = parseInt(weight, 10);
  if ((product_id == ""|| weight == "") || isNaN(weight_int) || isNaN(product_id_int) || isNaN(label_id_int)) {
    alert('invalid input');
  } else {
    let sender = await App.processorContract.methods.getProductProcessorAddress(product_id_int).call(); //get sender address
    await App.retailerContract.methods.recieveProduct(sender, product_id_int, label_id_int, weight_int).send({from: App.account, gasLimit: 4712388});
    alert('recieved');
  }
},
```

Figure 30: Receive Product as Retailer

Figure 30 works the same way as 'recieveProductProcessor' in *figure 28,* but unlike *figure 28,* label_id is a parameter. The extra parameter (label_id) is necessary as two separate farmers can send a product of the same product_id to a retailer, meaning it would not be possible to differentiate the senders’ addresses.

```javascript
getRetailerInventory: async (data) => {
  $('#retailer_inventory').empty(); //removes all products currently on screen
  const numberOfProducts = await App.retailerContract.methods.getLatestLabel().call(); //number of products in the ecosystem
  const currentlyInInventoryTemplate = $('#retailer_inventory'); //template that will hold inventory

  //iterate over all products and getting those that have a weight greater than 0 and belong to the current retailer
  for (var i = 0; i <= numberOfProducts; i++) {
    var i_int = parseInt(i, 10);
    //get current product info
    const product_weight = await App.retailerContract.methods.getProductWeight(i_int).call();
    const product_allocated = await App.retailerContract.methods.getAllocated(i_int).call();
    const product_id = await App.retailerContract.methods.getProductId(i_int).call();
    var product_id_int = parseInt(product_id, 10);
    const retailer_address = await App.retailerContract.methods.getProductRetailerAddress(i_int).call();
    var product_weight_int = parseInt(product_weight, 10);
    var product_allocated_int = parseInt(product_allocated, 10);
    var i_int = parseInt(i, 10);
    if (product_weight_int > 0 && retailer_address == App.account) {
      let name = await App.farmContract.methods.getProductName(product_id_int).call();
      let weight = product_weight_int;
      //product values to be output to the screen
      let output =
          `<a href="#" class="list-group-item list-group-item-action" data-toggle="list" role="tab">
              <div class="d-flex w-100 justify-content-between">
                <h5 class="mb-1" name="product_id">ID: `+ product_id +`</h5>
              </div>
                <p class="mb-1">Label ID: `+ i_int +`</p>
                <p class="mb-1" name="product_id">Name: `+ name +`</p>
                <p class="mb-1">Weight: `+ weight +`g</p>
            </a>`;

        currentlyInInventoryTemplate.append(output);

    }
   }
 }
}


$(() => {
    App.load();
})
```

Figure 31: Get Retailer Inventory

Figure 31 works the same as *figure 27* and *29.*

4.2.2 search.js

Search.js deals with the tracking of a product based on its label_id once it has reached a retailer.

```javascript
/*
* Description: This js file interacts with both the html files and
* solidity smart contracts to present the path in which a product has taken through the supplychain.
*/

App = {
  contracts: {},

  load: async () => {
    await App.loadWeb3() //imports web3 into the file
    await App.loadAllContracts() //load all contracts from the blockchain
  },

  loadWeb3: async() => {⋯},

  loadAllContracts: async() => {⋯},

  loadFarmerContract: async() => {⋯},

  loadProcessorContract: async() => {⋯},

  loadRetailerContract: async() => {⋯},
```

Figure 32: Load Contracts Search

Figure 32 loads all contracts.

```
organicProperties: async() => {
  //gets label value inserted by a user
  var label = $('[name="label"]').val();
  var label_int = parseInt(label, 10);
  const allocated = await App.retailerContract.methods.getAllocated(label_int).call();
  //checks if product with the label that the user has inserted is in the label text box
  if (allocated == "0") {
    alert("Product does not exist");
  } else {
    //get all attributes related to the label inserted
    const product_id = await App.retailerContract.methods.getProductId(label_int).call();
    var product_id_int = parseInt(product_id, 10);
    const type = await App.farmContract.methods.getProductTypeStr(product_id_int).call();
    const name = await App.farmContract.methods.getProductName(product_id).call();
    const weight = await App.retailerContract.methods.getProductWeight(label_int).call();
    const farmer_address = await App.farmContract.methods.getProductFarmerAddress(product_id_int).call();
    const organic_certifications = await App.farmContract.methods.getCertificationArray(farmer_address).call();
    organic_certifications[0];
    const organicPropertiesTemplate = $('#organicProperties');
    //check the type which defines what will be output on the screen
    if (type == "Produce") {
      let pesticides_used = await App.farmContract.methods.getPesticidesUsed(product_id_int).call();
      let fertilisers_used = await App.farmContract.methods.getFertilisersUsed(product_id_int).call();
      //data to be output
      let output =
         `<h5> Product Name: `+ name + `</h5>
          <h5> Product Weight: `+ weight + `</h5>
          <h5>Pesticides Used: `+ pesticides_used +`</h5>
          <h5>Fertilisers Used: `+ fertilisers_used +`</h5>
          </div>`;

      organicPropertiesTemplate.append(output);
    } else if (type == "Livestock") {
      const time_spent_outdoors = await App.farmContract.methods.getTimeSpentOutdoors(product_id_int).call();
      const feed_used = await App.farmContract.methods.getFeedUsed(product_id_int).call();
      const housing = await App.farmContract.methods.getHousing(product_id_int).call();
      //data to be output
      let output =
         `<h5> Proudct Name: `+ name + `</h5>
          <h5> Proudct Weight: `+ weight + `</h5>
          <h5>Time Spent Outdoors (Daily): `+ time_spent_outdoors +` minutes</h5>
          <h5>Feed Used: `+ feed_used +`</h5>
          <h5>Housing Of Livestock: `+ housing +`</h5>`;
      organicPropertiesTemplate.append(output);
    }

    const organic_certifications_template = $('#organicCertifications');
    organic_certifications_template.append(`<h4>Farmer Certifications</h4>`);
    //loop through all organic certifications held by a farmer and output them to the screen
    for(var i = 0; i < organic_certifications.length; i++) {
      let certification_name = await App.farmContract.methods.getCertificateName(organic_certifications[i]).call();
      let output =
        `<h5>`+ certification_name +`</h5>`;
      organic_certifications_template.append(output);
    }
  }
},
```

Figure 33: Organic Properties

Figure 33 interfaces with the retailer and farmer contract to retrieve the organic properties of a product. The user passes in a label from _id product_history.html form. The output is dependent on the type of the product, if the type is 'livestock' output time spent outdoors, feed used and housing if the type is 'produce' output pesticides and fertilisers used.

```
tracking: async() => {
  //gets label value inserted by a user
  var label = $('[name="label"]').val();
  var label_int = parseInt(label, 10);
  const product_id = await App.retailerContract.methods.getProductId(label_int).call();
  product_id_int = parseInt(product_id, 10);
  const allocated = await App.retailerContract.methods.getAllocated(label_int).call();
  //checks if product with the label that the user has inserted is in the label text box
  if (allocated == "0") {
    alert("Product does not exist");
  } else {
    //get all attributes for the product that are held in the retailer contract
    const retailer_address = await App.retailerContract.methods.getProductRetailerAddress(label_int).call();
    document.getElementById("retailer_address").innerHTML = retailer_address;
    const retailer_name = await App.retailerContract.methods.getRetailerName(retailer_address).call();
    document.getElementById("retailer_name").innerHTML = retailer_name;
    const retailer_weight = await App.retailerContract.methods.getProductWeight(label_int).call();
    document.getElementById("retailer_weight").innerHTML = retailer_weight;
    const retailerReceivedEvent = await App.retailerContract.getPastEvents('ProductRecieved', { filter: {_label: label_int}, fromBlock: 0, toBlock: 'latest'});
    document.getElementById("retailer_time_received").innerHTML = new Date(retailerReceivedEvent[0].returnValues.time*1000);

    //get all attributes for the product that are held in the processor contract
    const processorSendEvent = await App.processorContract.getPastEvents('ProductSent', { filter: {_label: label_int}, fromBlock: 0, toBlock: 'latest'});
    const processorRecievedEvent = await App.processorContract.getPastEvents('ProductRecieved', { filter: {_productId: product_id_int}, fromBlock: 0, toBlock: 'latest'});
    const processor_address = processorSendEvent[0].returnValues._sender;
    document.getElementById("processor_address").innerHTML = processor_address;
    const processor_name = await App.processorContract.methods.getProductProcessorName(processor_address).call();
    document.getElementById("processor_name").innerHTML = processor_name;
    document.getElementById("processor_weight_received").innerHTML = processorRecievedEvent[0].returnValues._weight;
    document.getElementById("processor_time_received").innerHTML = new Date(processorRecievedEvent[0].returnValues.time*1000);
    document.getElementById("processor_weight_sent").innerHTML = processorSendEvent[0].returnValues._weight;
    document.getElementById("processor_time_sent").innerHTML = new Date(processorSendEvent[0].returnValues.time*1000);

    //get all attributes for the product that are held in the farmer contract
    const farmSendEvent = await App.farmContract.getPastEvents('SendProduct', { filter: {_productId: product_id_int, _reciever: processor_address}, fromBlock: 0, toBlock: 'latest'});
    const farmHarvestEvent = await App.farmContract.getPastEvents('Harvest', { filter: {_productId: product_id_int}, fromBlock: 0, toBlock: 'latest'});
    const farmer_address = await App.farmContract.methods.getProductFarmerAddress(product_id_int).call();
    document.getElementById("farmer_address").innerHTML = farmer_address;
    const farmer_name = await App.farmContract.methods.getFarmerName(farmer_address).call();
    document.getElementById("farmer_name").innerHTML = farmer_name;
    document.getElementById("farmer_weight_harvest").innerHTML = farmHarvestEvent[0].returnValues.weight;
    document.getElementById("farmer_weight_sent").innerHTML = farmSendEvent[0].returnValues.weight;
    document.getElementById("farmer_time_harvest").innerHTML = new Date(farmHarvestEvent[0].returnValues.time*1000);
    document.getElementById("farmer_time_sent").innerHTML = new Date(farmSendEvent[0].returnValues.time*1000);
   }
  }
}



$(() => {
    App.load();
})
```

Figure 34: Tracking

Figure 34 reads a 'label_id' inserted into 'product history form' based on this 'label_id'. Using the 'label_id' the corresponding 'product_id', the 'label_id' is used to get the address of the retailer that owns the product. The retailer address is then used to get the retailers name. The 'label_id' is used to get the weight of the product that the retailer received. Finally, the event 'Product Received' is called and filtered using the 'label_id' to get the time that the product was received. This process is repeated for both processer and farmer except they both have time and weight of product sent.

4.2.3 entities.js

Entities.js gets all entities on the blockchain, which are then displayed in *figure 54.*

```
/*
* Description: This js file interacts with both the html files and
* solidity smart contracts to present all entities withing the supplychain
*/

App = {
  contracts: {},

  load: async () => {...},

  loadWeb3: async() => {...},

  loadAccount: async() => {...},

  loadAllContracts: async() => {...},

  loadAllEntities: async() => {...},

  loadFarmerContract: async() => {...},

  loadProcessorContract: async() => {...},

  loadRetailerContract: async() => {...},
```

Figure 35: Load Contracts Entities

Figure 35 loads all contracts.

```
getAllFarmers: async () => {
  const number_of_farmers = await App.farmContract.methods.getNumberOfFarmers().call();
  const all_farmers_template = $('#all_farmers'); //template that will hold individual farmers
  //iterates over all farmers within a look up table in the farm smart contract and appends them to all_farmers_template
  for(var i = 0; i < number_of_farmers; i++) {
    const farmer_address = await App.farmContract.methods.getFarmerAddress(i).call();
    const farmer_name = await App.farmContract.methods.getFarmerName(farmer_address).call();
    let output =
      `<p>` + farmer_name + `: ` + farmer_address + `</p>`;
     all_farmers_template.append(output);
  }
},

getAllProcessors: async () => {
  const number_of_processors = await App.processorContract.methods.getNumberOfProcessors().call();
  const all_processors_template = $('#all_processors'); //template that will hold individual processors
  //iterates over all processors within a look up table in the farm smart contract and appends them to all_processors_template
  for(var i = 0; i < number_of_processors; i++) {
    const processor_address = await App.processorContract.methods.getProcessorAddress(i).call();
    const processor_name = await App.processorContract.methods.getProcessorName(processor_address).call();
    let output =
      `<p>` + processor_name + `: ` + processor_address + `</p>`;
     all_processors_template.append(output);
  }
},

getAllRetailers: async () => {
  const number_of_retailers = await App.retailerContract.methods.getNumberOfRetailers().call();
  const all_retailers_template = $('#all_retailers');//template that will hold individual retailers
  //iterates over all retailers within a look up table in the farm smart contract and appends them to all_retailers_template
  for(var i = 0; i < number_of_retailers; i++) {
    const retailer_address = await App.retailerContract.methods.getRetailerAddress(i).call();
    const retailer_name = await App.retailerContract.methods.getRetailerName(retailer_address).call();
    let output =
      `<p>` + retailer_name + `: ` + retailer_address + `</p>`;
     all_retailers_template.append(output);
  }
 }
}

$(() => {
   App.load();

})
```

Figure 36: Get All Entities

Figure 36 interfaces with all three smart contracts getting every farmer, processor and retailer in each.

4.2.4 governing.js

Governing.js deals with the displaying of a governing bodies certifications and distribution of certificates.

```
/*
* Description: This js file interacts with both the html files and
* solidity smart contracts to pass data through the supply chain.
* Displays all certifications beloniging to a governing body allowing them
* to be distrubuted to farmers
*/
App = {
  contracts: {},

  load: async () => {…},

  loadWeb3: async() => {…},

  loadAccount: async() => {…},

  loadFarmerContract: async() => {…},

  getCertifications: async () => {
    const certficiationsTemplate = $('#certifications'); //template that will hold all certificates belonging to a governing body
    const governing_body_certificates = await App.farmContract.methods.getGoverningBodiesCertificates(App.account).call();
    for (var i = 0; i < governing_body_certificates.length; i++) {
      var i_int = parseInt(i, 10);
      const certificate_id = governing_body_certificates[i_int];
      var certificate_id_int = parseInt(certificate_id, 10);
      const certificate_name = await App.farmContract.methods.getCertificateName(certificate_id_int).call();
      //certificate values to be output to the screen
      let output =
          `<a href="#" class="list-group-item list-group-item-action" data-toggle="list" role="tab">
              <div class="d-flex w-100 justify-content-between">
                <h5 class="mb-1" name="product_id">ID: `+ governing_body_certificates[i_int] +`</h5>
              </div>
                <p class="mb-1" name="product_id">Name: `+ certificate_name +`</p>
           </a>`;

      certficiationsTemplate.append(output);

    }
  },
```

Figure 37: Get Certifications

Figure 37 loads farmer contract. Furthermore, it gets all certifications owned by the current account by calling 'getGoverningBodiesCertificates' which returns an array of certificate ids. The array of integers is iterated over returning the certificate name correlating to each id.

```
sendCertification: async () => {
  var items = document.getElementsByClassName("list-group-item active"); //get currently activated list item in governing_body_homepage.html
  var certificate_id = items[0].firstChild.nextElementSibling.innerText.slice(4); //slice first 4 characters giving the certificateId
  var certificate_id_int = parseInt(certificate_id, 10);
  //read reciever inputted by user in the governing_body_homepage.html form
  let reciever = $('[name="reciever"]').val();
  //validates input
  if (reciever == "") {
    alert('insufficient values');
  } else {
    const allocated = await App.farmContract.methods.getFarmerAllocated(reciever).call();
    if (allocated == '0') {
      alert('not allocated');
    } else if (allocated == '1') {
      const current_farmer_certifications = await App.farmContract.methods.getCertificationArray(reciever).call();
      var exists = 0;
      //checks if the reciever (farmer) already has the certificate
      for (var i = 0; i < current_farmer_certifications.length; i++) {
        if (current_farmer_certifications[i] == certificate_id_int) {
          var exists = 1;
        }
      }
      if (exists == 0) {
        await App.farmContract.methods.setFarmCertification(reciever, certificate_id).send({from: App.account, gasLimit: 4712388});
        alert('sent');
      } else {
        alert ('farmer already has this certification');
      }

    }
  }
 }
}


$(() => {
  App.load();
})
```

Figure 38: Send Certification

Figure 38 works the same as *figure 26* and *28* except a product is not being sent; it is a certification.

4.3 Front-end

Figure 39: Navbar

Figure 39 is a navbar that will appear at the top of each form. Allows for navigation between forms and displaying the address of the current Ethereum wallet. The address will not be displayed for all forms, as *figures 54* and *56* require that users can interact with them without needing an Ethereum wallet.

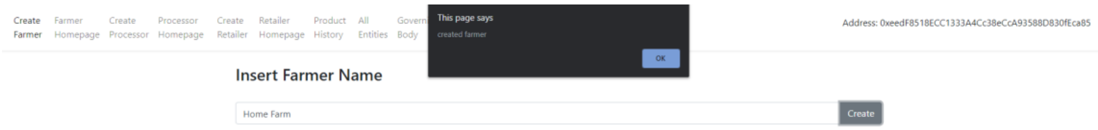


Figure 40: Create a Farmer Form

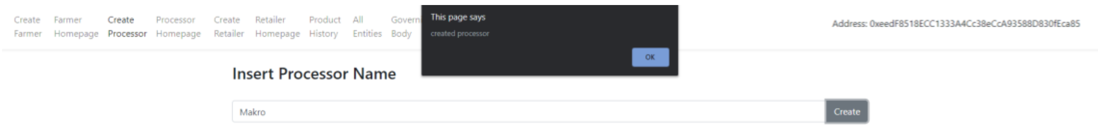


Figure 41: Create a Processor Form

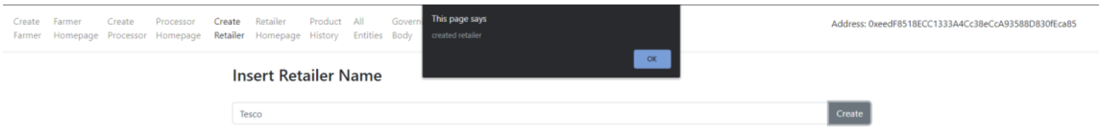


Figure 42: Create a Retailer Form

Figures 40, *41* and *42* are simple forms which create an entity at the current Ethereum wallet address with the name inserted into the text box.

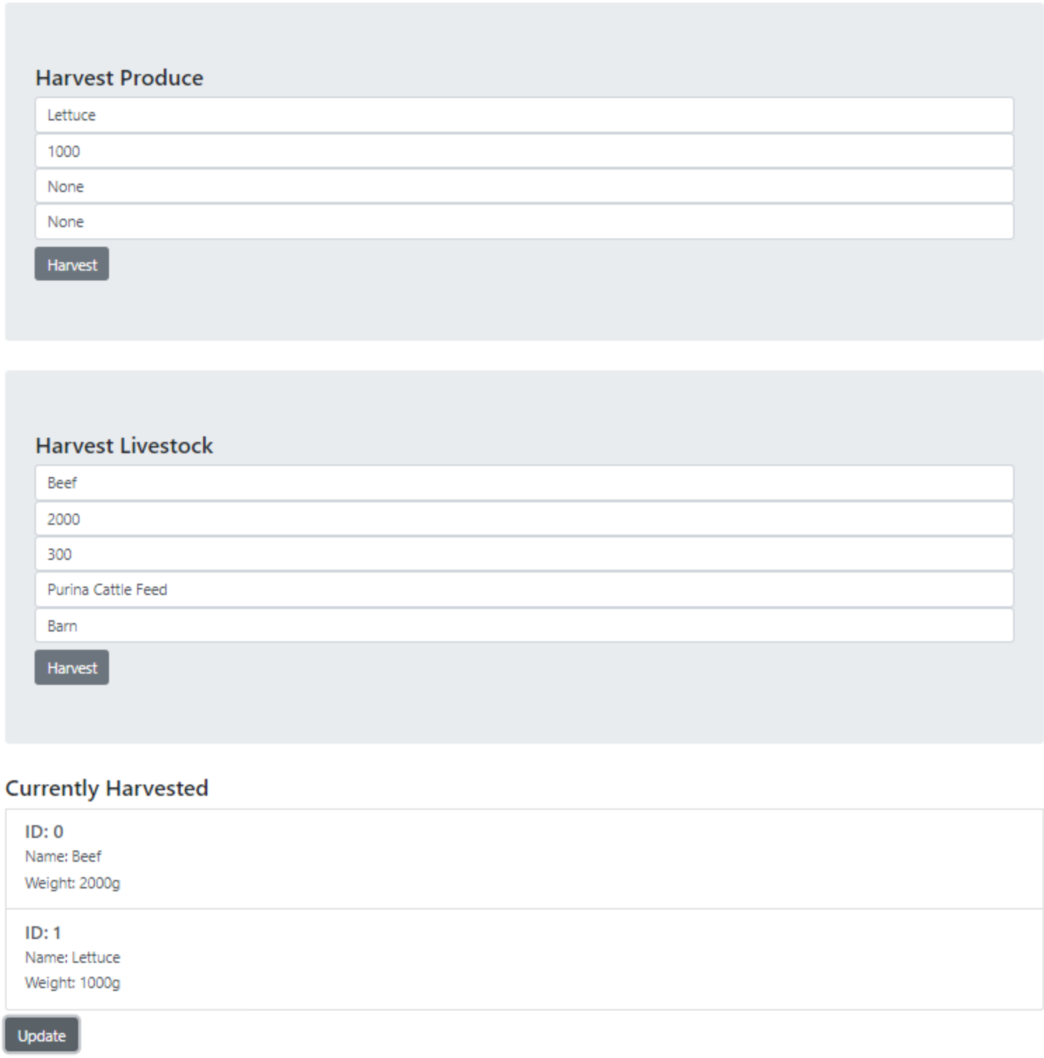


Figure 43: Farmer Homepage Form

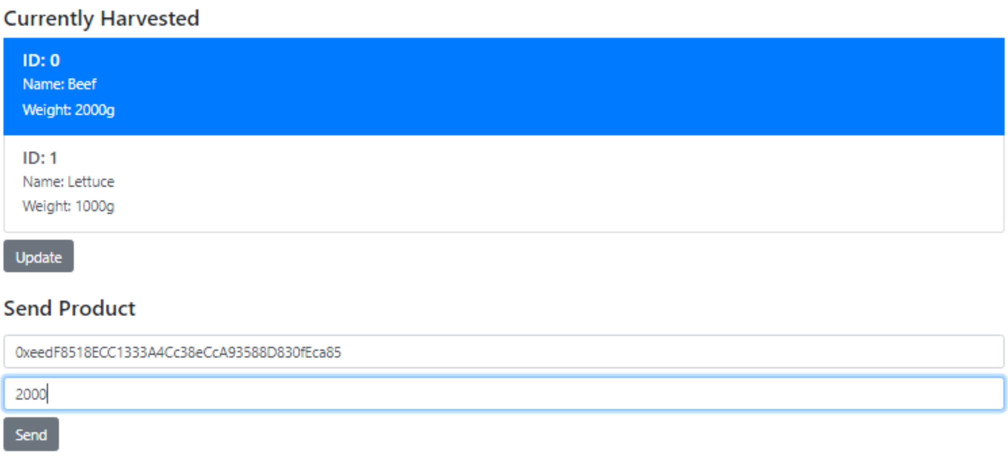


Figure 44: Send Livestock as Farmer

Figure 43 and 44 make up the farmer homepage. Dependant on their needs, users can harvest produce or livestock. This product will then appear in their inventory indexed by the product id. Each harvested product will be given a unique id. If the farmer wishes to see an updated view of their inventory, they should click on the update button. To send a product, a farmer will select the product they wish to send, insert a processor address which can be found in *figure 54* and insert a valid weight. If a product is successfully sent, an alert will appear (*figure 46*).

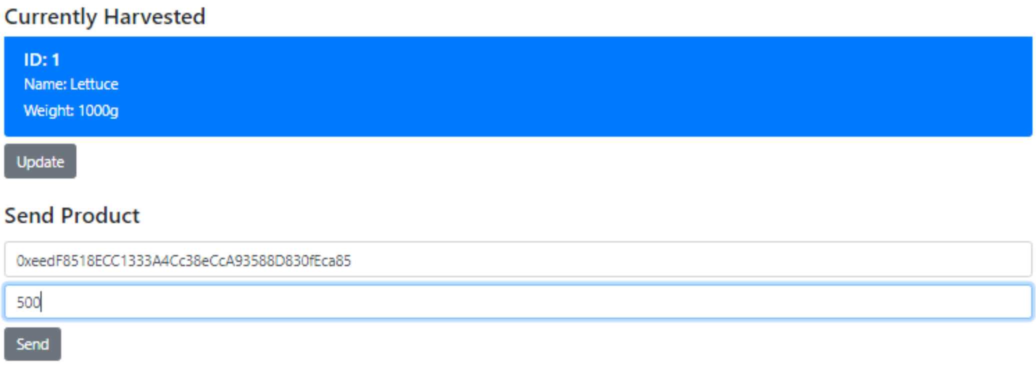


Figure 45: Send Produce as Farmer

Figure 46: Sent Alert

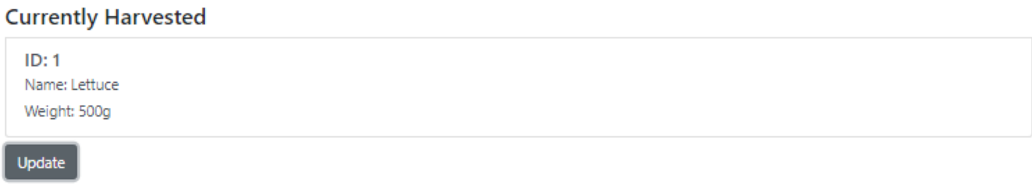


Figure 47: Weight of Lettuce After Send

Both *figures 45* and *47* show the farmers inventory after the products are sent.

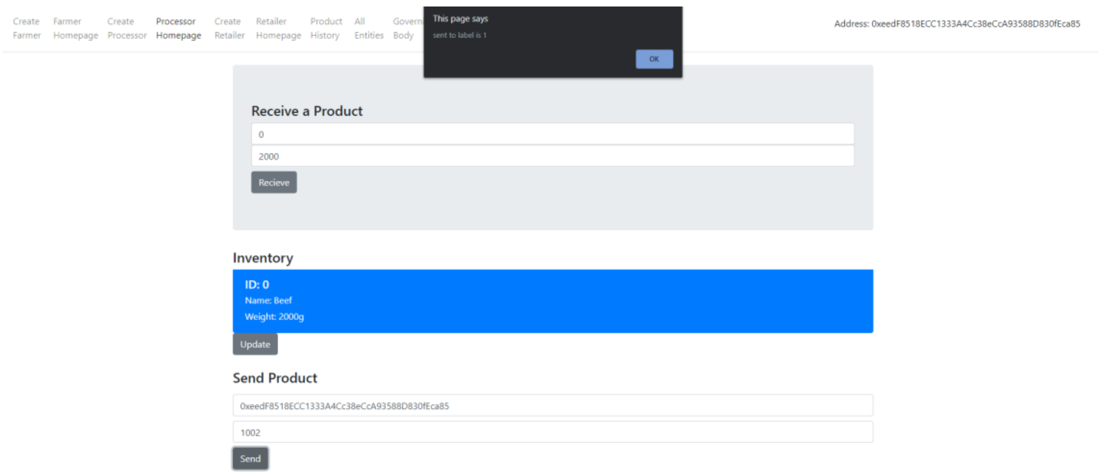


Figure 48: Processor Homepage Receive and Send Livestock

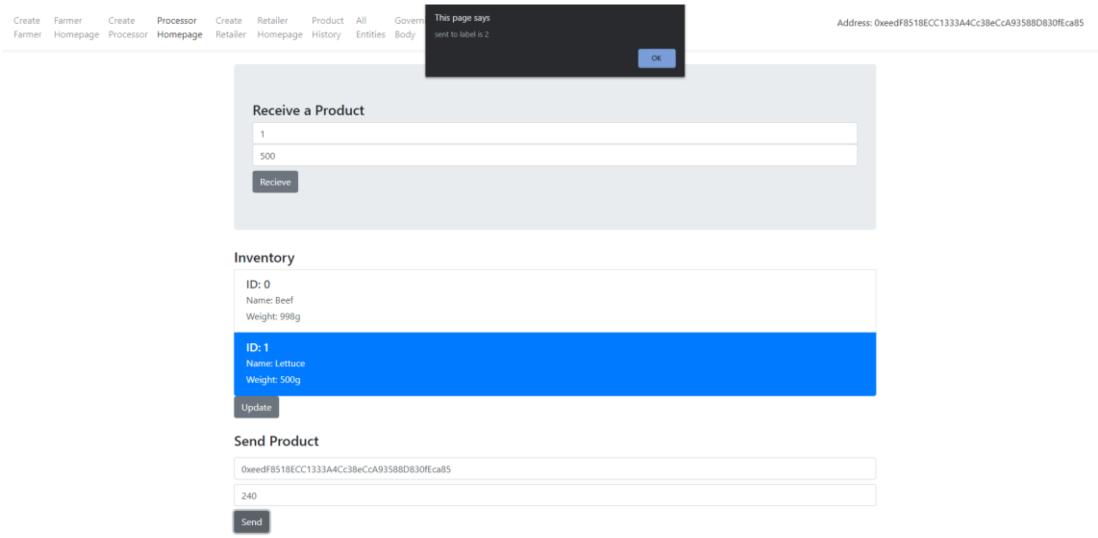


Figure 49: Processor Homepage Receive and Send Produce

Figures 48 and *49* display a processor receiving both produce and livestock by inserting the product id and weight of the product received. The received product can then be sent to a retailer in the same manner as in the farmer's homepage except a processor must insert a retailer's address. On send an alert with a label id will be given to the processor. Label id is separate from the initial product id and is necessary as many processors may have a product of the same product id. Therefore, it allows the retailer to differentiate which processor had sent the product.

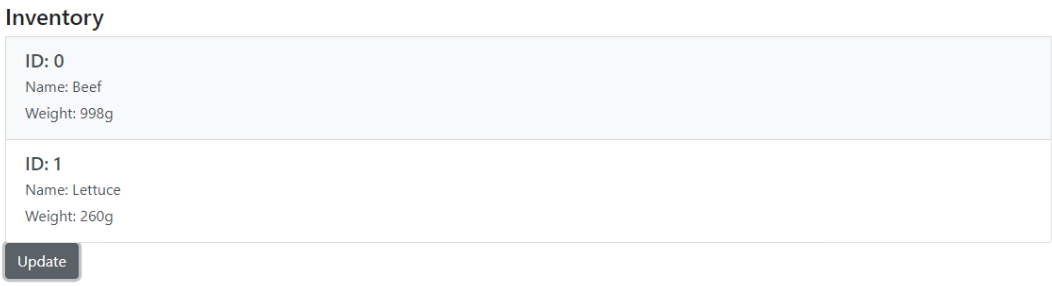


Figure 50: Processor Inventory After All Sends

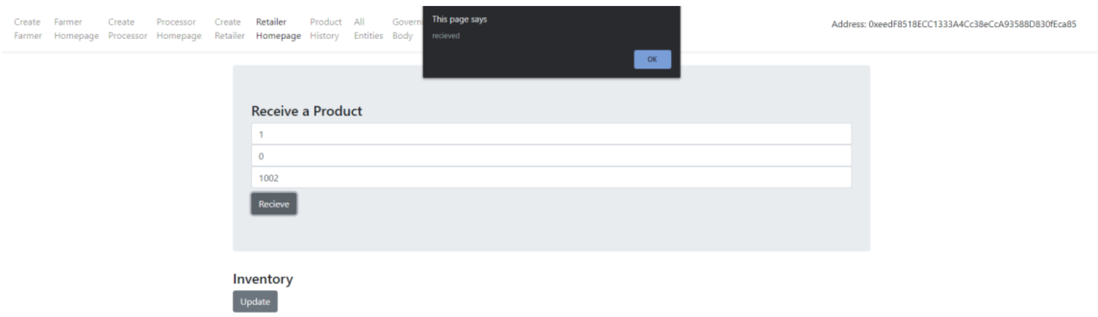


Figure 51: Retailer Receives Livestock

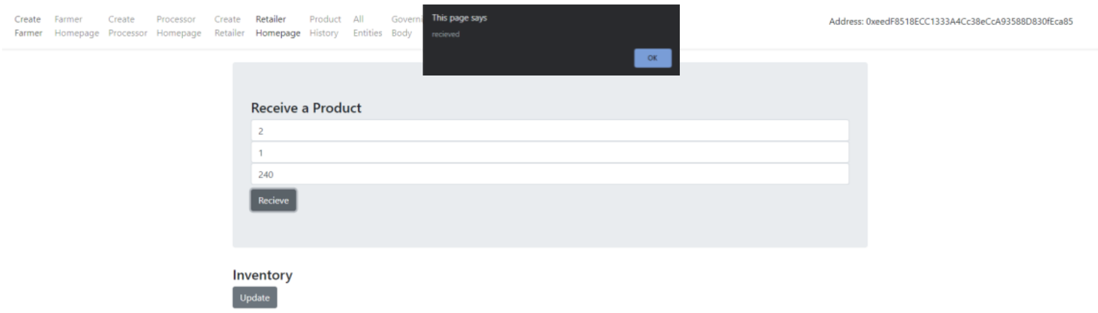


Figure 52: Retailer Receives Produce

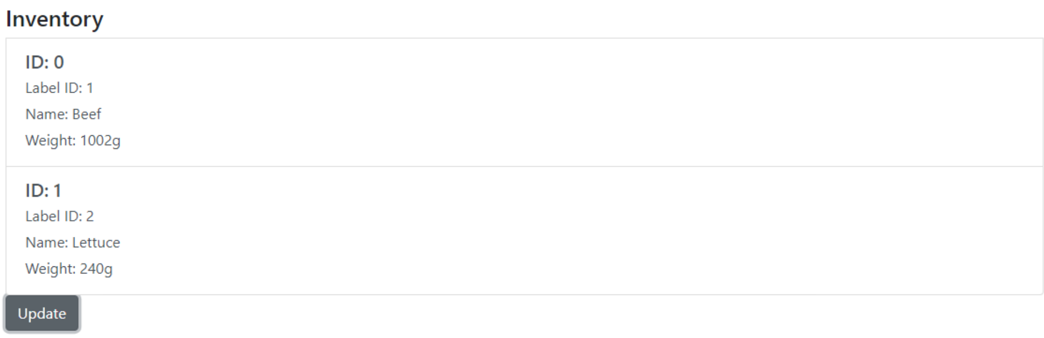


Figure 53: Retailer Inventory After All Receives

Figures 51, *52* and *53* display the retailer receiving livestock and produce. This is the final entity that a product will pass through once a product has reached this stage a user is able to track its path through the supply chain.

Figure 54: All Entities

Figure 54 lists all the names of all entities and their corresponding address. Acting as a lookup table, it simply allows users to find the address of the entity that they wish to send a product.

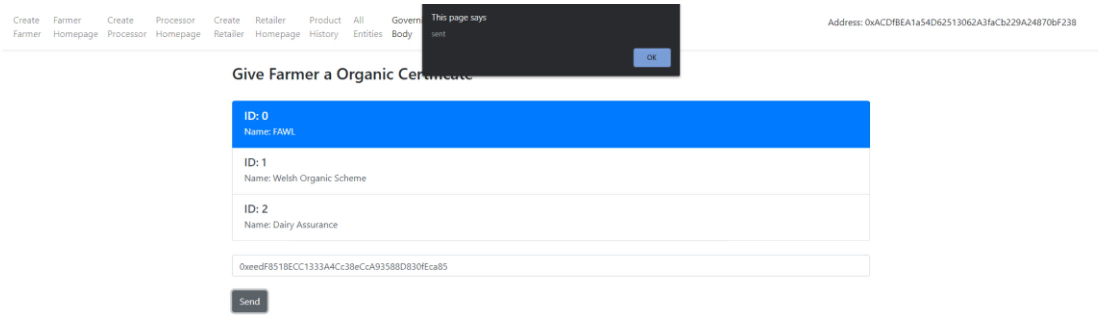


Figure 55: Give Organic Certification

Figure 55 displays a governing body distributing an organic certification to a farmer by selecting the certification they wish to send and finally inserting the farmer's address.

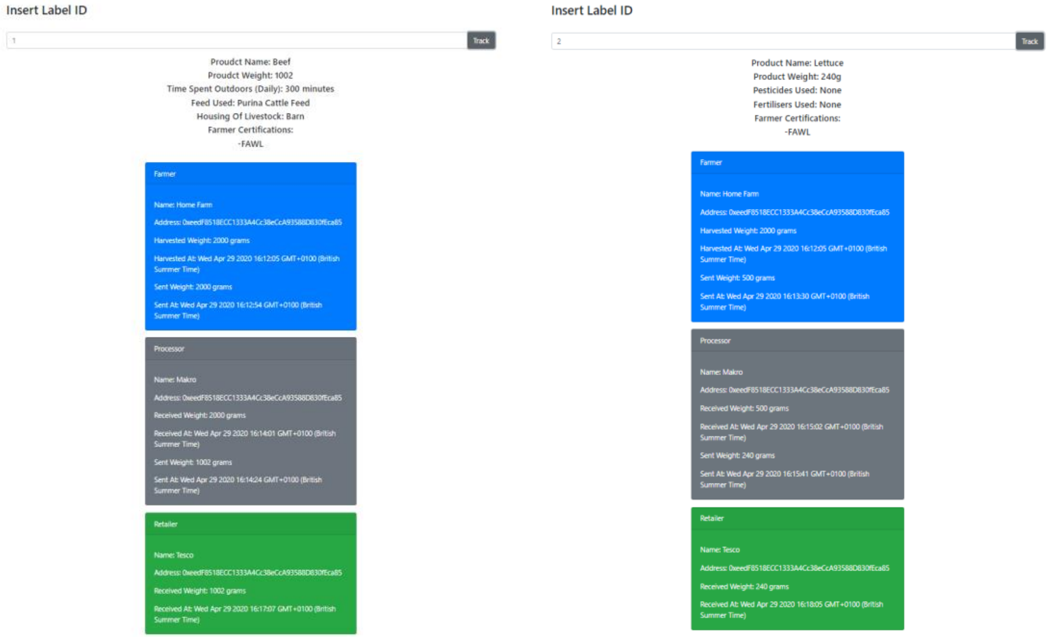


Figure 56: Tracking Produce and Livestock

Figure 56 shows organic properties and the path both products have travelled through the supply chain. The user inserts the label id of a product into the text box then presses the track button. Organic properties are displayed; these vary dependant on the product type. Tracking information is displayed in three bootstrap jumbotron boxes, one for each entity.

4.4 Testing

Testing was completed throughout the development, as functionality was completed tests would be run to ensure it was successfully implemented. All tests ran were compiled into multiple tables; these tables can be found in the Appendix A section of the document under the heading 'All Test Tables'. There are two tables, and each table has four columns these columns being:

1. Name of the test.
2. Test Data.
3. Expected behaviour.
4. Whether the expected behaviour matches the actual behaviour, this will be displayed as a Pass or Fail. On my supervisor's recommendation, I have included references to screenshots of the test passing or failing.

All A.1.1 basic functionality has passed as expected.
Issues with A.1.2 data validation exist, most prolific occur when data is received by both a processor and retailer. A processor and retailer can insert any product id and weight into their inventory, even if they have not received the product. The product will then appear in their inventory if the product received is not currently on the blockchain the name will not display.

Chapter 5

Evaluation

5.1 Summary

To summarise the project was a success, meeting a majority of the aims.

Below are the aims displayed and colour coded, Green implies a full implementation, Orange implies a partial implementation and Red implies no implementation.

- Create software to track produce as it is transported through three different parts of the supply chain.
  - Farmer
  - Processor
  - Retailer
- Create a distributed application for web browsers.
  - Create a portal for farmers, processors, retailers, governing bodies and customers.
- Store organic certification issued to farmers by governing bodies on the blockchain.
- Accommodate for produce.
  - Store type of pesticides and fertilisers used to grow produce on the blockchain.
- Accommodate for livestock.
  - Store type of feed, where the livestock is held and how long they spend outdoors on the blockchain.
- Give the user an understanding of how organic a compound product (ready meal) is.
  - Provide a percentage of the ingredients that are organic.
- Users should not be able to insert invalid data into the distributed web application.

Having the composite product feature implemented would add a great deal of value to the system allowing the system to cover a broadened range of products. However, it became evident that it would not be possible to complete during development due to time constraints. A start was made by beginning to write the smart contract code that store and send a composite product (*figure 57*).

The system provides partial validation and verification of user input. For this system to be implemented in the real world, it would need to be far more watertight. As stated in chapter 4, allowing users to receive a product when said product has not been sent to them will lead to security issues. Security issues may lead to a loss in the general publics trust in the system. Even when considering this, the system proves that it is possible to create transparency in the Ethereum blockchain using smart contracts.

5.2 Implications

If the system were to be widely implemented, it would have implications within social, economic and environmental sectors. In this section, the implications of the system are discussed.

5.2.1 Social

Eating more organic food may have a positive effect on the general public's health due to the reduction/removal of antimicrobials in livestock. A study chaired by Jim O'Neill found that the higher use of antimicrobials can lead to a rise in drug-resistant strains of microbes which can be passed on to humans (O'Neill, 2015), therefore leading to illnesses that cannot be treated by conventional drugs.

5.2.2 Economic

If the system is successfully integrated into the organic food supply chain, we could see organic companies gaining larger market shares within the food production industry. Consequently, companies who previously held these shares will suffer financially and may attempt to regain these shares by producing more organic products.

The increase in demand for organic products may cause fluctuations in market prices. Initially, supply may not be able to meet demand driving prices up. As large companies begin to implement the system and produce greater quantities of organic products; the demand will begin to be met driving prices down.

5.2.3 Environmental

This system may affect the way that the general public perceives organic produce. Some may be disappointed when they learn about the reality of organic products; for example, they expected livestock to be cared for to a higher standard. This could lead to a more significant push from the general public for farmers to improve livestock quality of life.

An increase in deforestation may occur if people begin to purchase more organic products due to the extensive amount of land that is typically required to grow organic produce. A study completed by the Chalmers University of Technology found that growing organic peas have around a 50% bigger climate impact (Searchinger, et al., 2018).

5.3 Future Work

In the future, I will go onto implement the composite product feature. A start on the composite product smart contract code was made, as seen in *figure 57.* In a fully implemented system, the code in *figure 57* would belong in the 'Processor.sol' smart contract. As it belongs to 'Processor.sol', only processors will be able to make composite products.

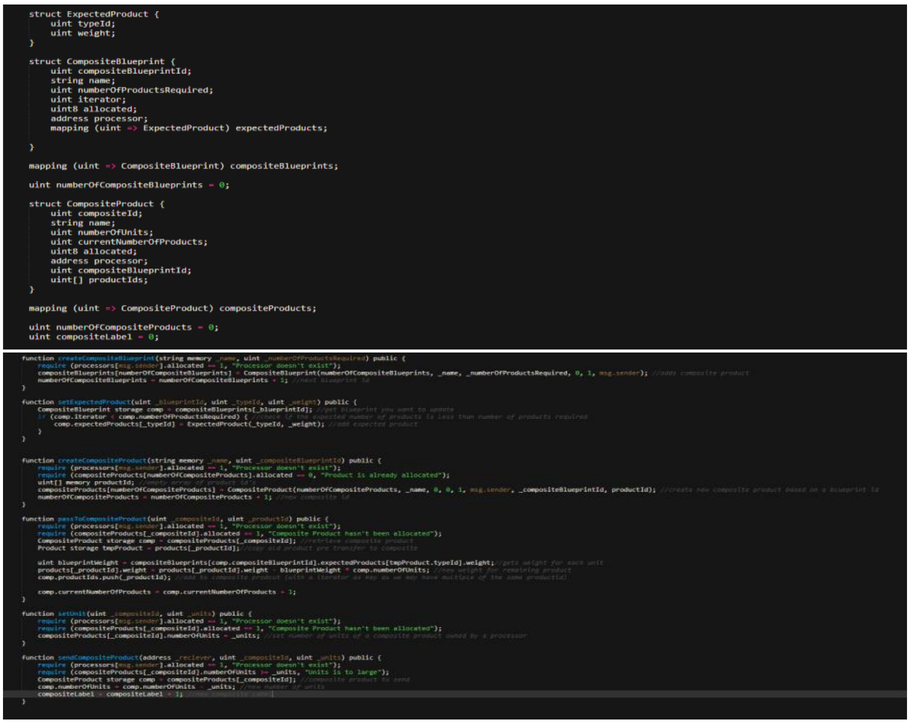
The process of creating a composite product is split in two:

- The creation of a blueprint
- The creation of a composite product.

A blueprint is made up of the mapping 'ExpectedProducts' indexed by a products id. Each expected product will have an expected type and weight.

A user can then create a composite product which will be defined by a 'compositeBlueprintId'. Products can be passed to the composite product, and the productId is stored in an array of unsigned integers. On passing to a composite product, the controller (not yet implemented) will check if the type of the product matches the type of an expected product.

Blueprints exist on the assumption that a processor will want to make multiple of the same composite product, saving time for the processor as they only have to define the composite product once.



```
struct ExpectedProduct {
    uint typeId;
    uint weight;
}

struct CompositeBlueprint {
    uint compositeBlueprintId;
    string name;
    uint numberOfProductsRequired;
    uint iterator;
    uint8 allocated;
    address processor;
    mapping (uint => ExpectedProduct) expectedProducts;

}

mapping (uint => CompositeBlueprint) compositeBlueprints;

uint numberOfCompositeBlueprints = 0;

struct CompositeProduct {
    uint compositeId;
    string name;
    uint numberOfUnits;
    uint currentNumberOfProducts;
    uint8 allocated;
    address processor;
    uint compositeBlueprintId;
    uint[] productIds;
}

mapping (uint => CompositeProduct) compositeProducts;

uint numberOfCompositeProducts = 0;
uint compositeLabel = 0;
```

```
function createCompositeBlueprint(string memory _name, uint _numberOfProductsRequired) public {
    require (processors[msg.sender].allocated == 1, "Processor doesn't exist");
    compositeBlueprints[numberOfCompositeBlueprints] = CompositeBlueprint(numberOfCompositeBlueprints, _name, _numberOfProductsRequired, 0, 1, msg.sender); //adds composite product
    numberOfCompositeBlueprints = numberOfCompositeBlueprints + 1; //next blueprint id
}

function setExpectedProduct(uint _blueprintId, uint _typeId, uint _weight) public {
    CompositeBlueprint storage comp = compositeBlueprints[_blueprintId]; //get blueprint you want to update
    if (comp.iterator < comp.numberOfProductsRequired) { //check if the expected number of products is less than number of products required
        comp.expectedProducts[_typeId] = ExpectedProduct(_typeId, _weight); //add expected product
    }
}


function createCompositeProduct(string memory _name, uint _compositeBlueprintId) public {
    require (processors[msg.sender].allocated == 1, "Processor doesn't exist");
    require (compositeProducts[numberOfCompositeProducts].allocated == 0, "Product is already allocated");
    uint[] memory productId; //empty array of product id's
    compositeProducts[numberOfCompositeProducts] = CompositeProduct(numberOfCompositeProducts, _name, 0, 0, 1, msg.sender, _compositeBlueprintId, productId); //create new composite product based on a blueprint id
    numberOfCompositeProducts = numberOfCompositeProducts + 1; //new composite id
}

function passToCompositeProduct(uint _compositeId, uint _productId) public {
    require (processors[msg.sender].allocated == 1, "Processor doesn't exist");
    require (compositeProducts[_compositeId].allocated == 1, "Composite Product hasn't been allocated");
    CompositeProduct storage comp = compositeProducts[_compositeId]; //retrieve composite product
    Product storage tmpProduct = products[_productId];//copy old product pre transfer to composite

    uint blueprintWeight = compositeBlueprints[comp.compositeBlueprintId].expectedProducts[tmpProduct.typeId].weight;//gets weight for each unit
    products[_productId].weight = products[_productId].weight - blueprintWeight * comp.numberOfUnits; //new weight for remaining product
    comp.productIds.push(_productId); //add to composite product (with a iterator as key as we may have multiple of the same productId)

    comp.currentNumberOfProducts = comp.currentNumberOfProducts + 1;
}

function setUnit(uint _compositeId, uint _units) public {
    require (processors[msg.sender].allocated == 1, "Processor doesn't exist");
    require (compositeProducts[_compositeId].allocated == 1, "Composite Product hasn't been allocated");
    compositeProducts[_compositeId].numberOfUnits = _units; //set number of units of a composite product owned by a processor
}

function sendCompositeProduct(address _reciever, uint _compositeId, uint _units) public {
    require (processors[msg.sender].allocated == 1, "Processor doesn't exist");
    require (compositeProducts[_compositeId].numberOfUnits >= _units, "Units is to large");
    CompositeProduct storage comp = compositeProducts[_compositeId]; //composite product to send
    comp.numberOfUnits = comp.numberOfUnits - _units; //new number of units
    compositeLabel = compositeLabel + 1; //new composite label
}
```

Figure 57: Composite Product Smart Contract Code

Finally, I would like to explore the possibility of reducing the number of smart contracts from three to one. Removing modularity may cause some unexpected issues moving forward with future updates to the system. On the contrary, it may reduce the amount of gas required to deploy the contracts onto the blockchain. A knock-on effect would be seen in the Javascript/JQuery controllers, with a reduction of the amount of code having to be written. For example, the loading of contracts (*figure 22)* would only require one function instead of three.

Bibliography

Atom, 2020. *Atom.* [Online]
Available at: https://atom.io/

BBC News, 2017. *Fipronil egg scandal: What we know.* [Online]
Available at: https://www.bbc.co.uk/news/world-europe-40878381

Bootstrap, 2020. *Bootstrap.* [Online]
Available at: https://getbootstrap.com/

Buterin, V., 2013. *A NEXT GENERATION SMART CONTRACT & DECENTRALISED APPLICATION PLATFORM.* s.l.:s.n.

Cohn, J. & Yiannas, F., 2017. *YouTube.* [Online]
Available at: https://www.youtube.com/watch?v=MMOF0G_2H0A

Ethereum, 2019. *Welcome to Remix documentation!.* [Online]
Available at: https://remix-ide.readthedocs.io/en/latest/

European Commission, 2019. *Organic farming in the EU.* [Online]
Available at: https://ec.europa.eu/info/sites/info/files/food-farming-fisheries/farming/documents/market-brief-organic-farming-in-the-eu_mar2019_en.pdf

Federal Register, 2010. *Electronic Code of Federal Regulations.* [Online]
Available at: https://www.ecfr.gov/cgi-bin/ECFR?page=browse

Foley, R., 2019. *Head of America's largest organic food fraud scheme scentenced to 10 years.* [Online]
Available at: https://globalnews.ca/news/5778147/organic-food-fraud-scheme/.

Github Inc, 2020. *GitHub.* [Online]
Available at: https://github.com/

Git, n.d. *git.* [Online]
Available at: https://git-scm.com/
[Accessed 2nd May 2020].

Hyperledger, 2016. *Hyperledger Blockchain Performance Metrics.* s.l.:s.n.

IBM, n.d. *IBM Food Trust.* [Online]
Available at: https://www.ibm.com/uk-en/blockchain/solutions/food-trust
[Accessed 3 May 2020].

JQuery, 2020. [Online]
Available at: https://jquery.com

MDN contributors, 2020. *JavaScript.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/JavaScript

MDN, 2020. *CSS: Cascading Style Sheets.* [Online]
Available at: https://developer.mozilla.org/en-US/docs/Web/CSS

Microsoft, 2020. *ASP.NET MVC Pattern.* [Online]
Available at: https://dotnet.microsoft.com/apps/aspnet/mvc

Mix, 2019. *The Next Web.* [Online]
Available at: https://thenextweb.com/hardfork/2019/05/24/javascript-programming-java-cryptocurrency/.

Nakamoto, S., 2008. *Bitcoin: A Peer-to-Peer Electronic Cash System.* s.l.:s.n.

npmjs, 2020. *npm.* [Online]
Available at: https://www.npmjs.com/

O'Neill, J., 2015. *ANTIMICROBIALS IN AGRICULTURE AND THE ENVIRONMENT: REDUCING UNNECESSARY USE AND WASTE,* s.l.: s.n.

Provenance, 2020. *Provenance.* [Online]
Available at: https://www.provenance.org/

Sainsbury's, n.d. *Sainsbury's Whole Chicken Free Range, Taste the Difference (approx. 900g-2.2kg).* [Online]
Available at: https://www.sainsburys.co.uk/gol-ui/product/sainsburys-british-free-range-whole-chicken--taste-the-difference-900g---22kg?catalogId=10241&productId=113060&storeId=10151&langId=44&krypto=o1InhZz3HZVNiUq%2Bb%2F8rwfvAD1lKAMFPjs0CsjlJK%2F9DJ1Z1AU%2BTxEQP3i%2B
[Accessed 3rd May 2020].

Searchinger, T., Wirsenius, S., Beringer, T. & Dumas, P., 2018. Assessing the efficiency of changes in land use for mitigating climate change. *nature.*

Solidity, 2019. *Solidity.readthedocs.io.* [Online]
Available at: https://solidity.readthedocs.io/en/v0.5.12/.

Truffle, 2020. *Truffle Suite.* [Online]
Available at: https://www.trufflesuite.com/truffle

w3schools, n.d. *HTML Introduction.* [Online]
Available at: https://www.w3schools.com/html/html_intro.asp
[Accessed 2 May 2020].

Web3js, 2020. *web3.js - Ethereum JavaScript API.* [Online]
Available at: https://web3js.readthedocs.io/en/v1.2.6/

Willer, H., Schaack, D. & Lernoud, J., 2017. *Organic Farming and Market Development in Europe and the European Union.* s.l.:Research Institute of Organic Agriculture FiBL and IFOAM.

Appendix A
Supplementary Data

A.1 All Test Tables

A.1.1 Basic Functionality

	Input	Expected Behaviour	Passed (P/F)
Create a Farmer	*Name*: Home Farm *Created at Address*: 0xeedF8518ECC1333A4Cc38eCcA93588D830fEca85	*Alert*: Created Farmer	P *figure 40*
Create a Processor	*Name*: Makro *Created at Address*: 0xeedF8518ECC1333A4Cc38eCcA93588D830fEca85	*Alert*: Created Processor	P *figure 41*
Create a Retailer	*Name*: Tesco *Created at Address*: 0xeedF8518ECC1333A4Cc38eCcA93588D830fEca85	*Alert*: Created Retailer	P *figure 42*
Give Farmer a Certificate	*Certification ID*: 0 *Farmer Address*: 0xeedF8518ECC1333A4Cc38eCcA93588D830fEca85	*Alert*: sent	P *figure 55 and 56*
Harvest Livestock	*Name*: Beef *Weight*: 2000g *Time Spent Outdoors*: 300 mins *Feed Used*: Purina Cattle Feed *Housing*: Barn	*Alert*: Harvested Displayed in farmer inventory: *Product ID:* 0 *Name:* Beef *Weight:* 2000g	P *figure 43*
Harvest Produce	*Name*: Lettuce *Weight*: 1000g	*Alert*: Harvested	P *figure 43*

	Pesticides Used: None *Fertilisers Used:* None	Displayed in farmer inventory: *Product ID:* 1 *Name:* Lettuce *Weight:* 1000g	
Send Livestock (Product) To Processor	*Product ID*: 0 *Weight:* 2000g *Processor Address*: 0xeedF8518ECC1333A4Cc38eCcA93588D830fEca85	*Alert*: Sent *No longer exist in farmer inventory*	P *figure 44*
Receive Livestock (Product) As Processor	*Product ID*: 0 *Weight*: 2000g	*Alert*: Received In Processor Inventory: *Product ID:* 0 *Name:* Beef *Weight:* 2000g	P *figure 48*
Send Livestock (Product) To Retailer	*Product ID*: 0 *Weight*: 1002g *Retailer Address*: 0xeedF8518ECC1333A4Cc38eCcA93588D830fEca85	*Alert*: Sent In Processor Inventory: *Product ID:* 0 *Name:* Beef *Weight:* 998g	P *figure 48*
Receive Livestock (Product) As Retailer	*Label ID*: 1 *Product ID*: 0 *Weight*: 1002g	*Alert*: Received In Retailer Inventory: *Product ID:*0 *Label ID:* 1 *Weight:* 1002g	P *figure 51 and 53*
Send Produce (Product) To Processor	*Product ID*: 1 *Processor Address:* 0xeedF8518ECC1333A4Cc38eCcA93588D830fEca85 *Weight:* 500g	*Alert*: Sent In Farmer Inventory: *Product ID:* 1 *Name:* Lettuce *Weight:* 500g	P *figure 45*
Receive Produce (Product) As Processor	*Product ID*: 1 *Weight*: 500g	*Alert*: Received In Processor Inventory: *Product ID:* 1 *Name:* Lettuce *Weight:* 500g	P *figure 49*
Send Produce	*Product ID*: 1 *Weight*: 240g	*Alert*: Sent In Processor Inventory:	P *figure 49 and 50*

(Product) To Retailer	Retailer Address: 0xeedF8518ECC1333A4Cc38eCcA93588D830fEca85	*Product ID:* 1 *Name:* Lettuce *Weight:* 260g	
Receive Produce (Product) As Retailer	*Label ID*: 2 *Product ID*: 1 *Weight*: 240g	*Alert*: Received In Retailer Inventory: *Product ID:*1 *Label ID:* 2 *Weight:* 240g	P *figure 52* *figure 53*
Display All Entities In The System	No Input	Home Farm Makro Tesco	P *figure 54*
View Produce (Product) History	*Label ID*: 2	All lettuce info (product id: 1) at each point of the supply chain including all organic properties	P *Figure 56*
View Livestock (Product) History	*Label ID: 1*	All beef (product id: 0) info at each point of the supply chain including all organic properties	P *Figure 56*

A.1.2 Data Validation

Test	Input	Expected Behaviour	Passed (P/F)
Two Farmers To One Address	*Farmer Address:* 0x408a56D80D9424f1C4C9B325cF981B9520002784 *Farmer Name:* Farm 1 *Farmer Name:* Farm 2	*Alert***:** farmer already exists *Farmer isn't created*	P *Figure 58* *Figure 61*
Two Processors To One Address	*Processor Address:* 0x408a56D80D9424f1C4C9B325cF981B9520002784 *Processor Name:* Processor 1 *Processor Name:* Processor 2	*Alert***:** processor already exists *Processor isn't created*	P *Figure 59* *Figure 61*
Two Retailers To One Address	*Retailer Address:* 0x408a56D80D9424f1C4C9B325cF981B9520002784 *Retailer Name:* Retailer 1 *Retailer Name:* Retailer 2	*Alert***:** retailer already exists *Retailer isn't created*	P *Figure 60* *Figure 61*
Ensure Entity	*Farmer Address:*	*Alert:* not allocated	P *Figure 62*

Exists (Send)	*Processor Address that does not exist:*	*Product still in farmer inventory*	
Ensure Product Sent	*Product ID of the product that has not been sent:* *Weight:*	*Alert:* product has not been sent to you	F *Figure 63*
Ensure Weight Is Valid (Send)	*Weight in farmer inventory:* 1000g *Weight to send:* 2000g	*Alert:* insert a valid weight *Product still in farmer inventory*	P *Figure 65*
Ensure Weight Is Valid (Received)	*Weight sent:* 1000g *Weight received:* 2000g	*Alert:* weight is not valid	F *Figure 64*
Insert Strings Into Integer Fields	*Weight***:** Eight	*Alert***:** insufficient value	P *Figure 66*
Leaving Fields Empty	*No input*	*Alert*: empty	P *Figure 67*
Not Connected to the Blockchain	*N/A*	*Alert*: Not connected to the blockchain	F *Figure 68*

A.2 Data Validation Results

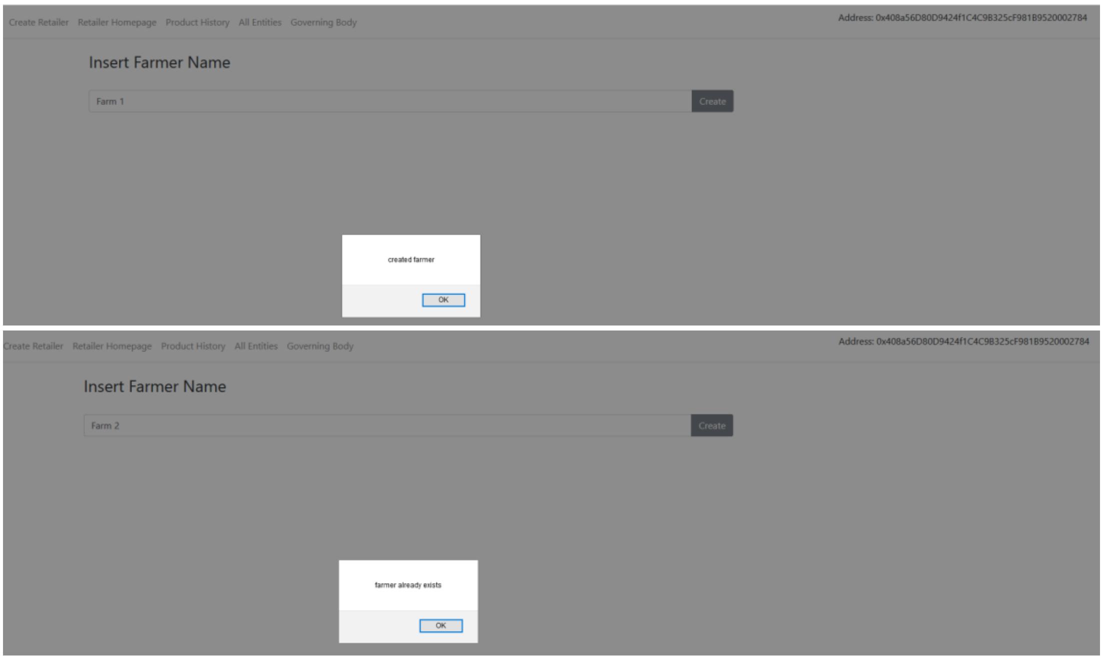


Figure 58: Multiple Farmers at the Same Address

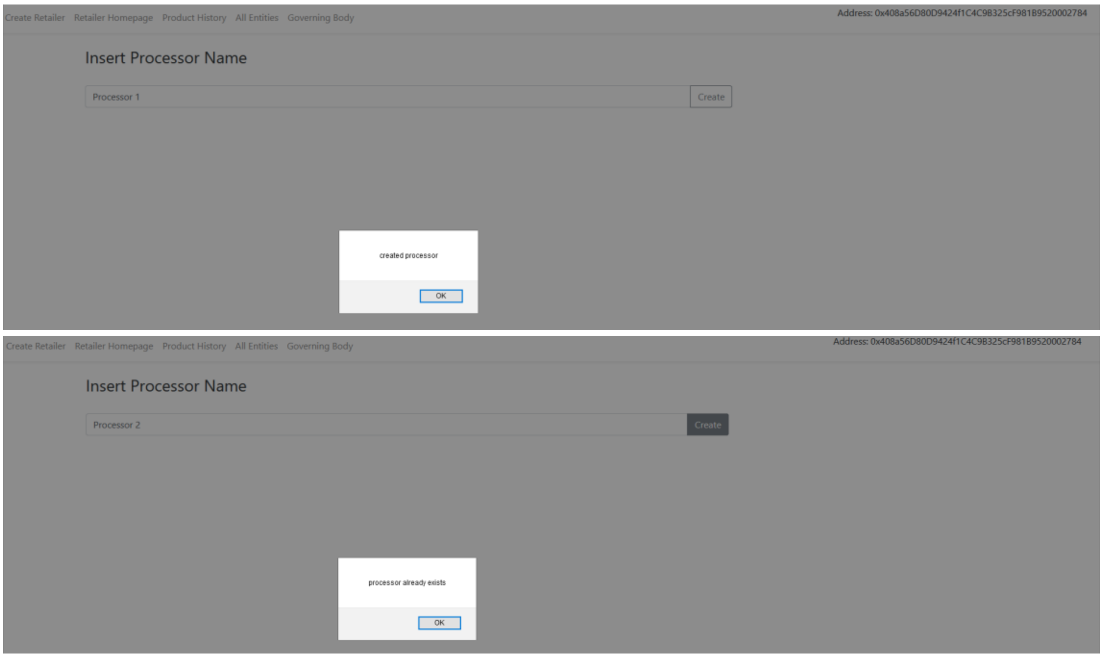


Figure 59: Multiple Processors at the Same Address

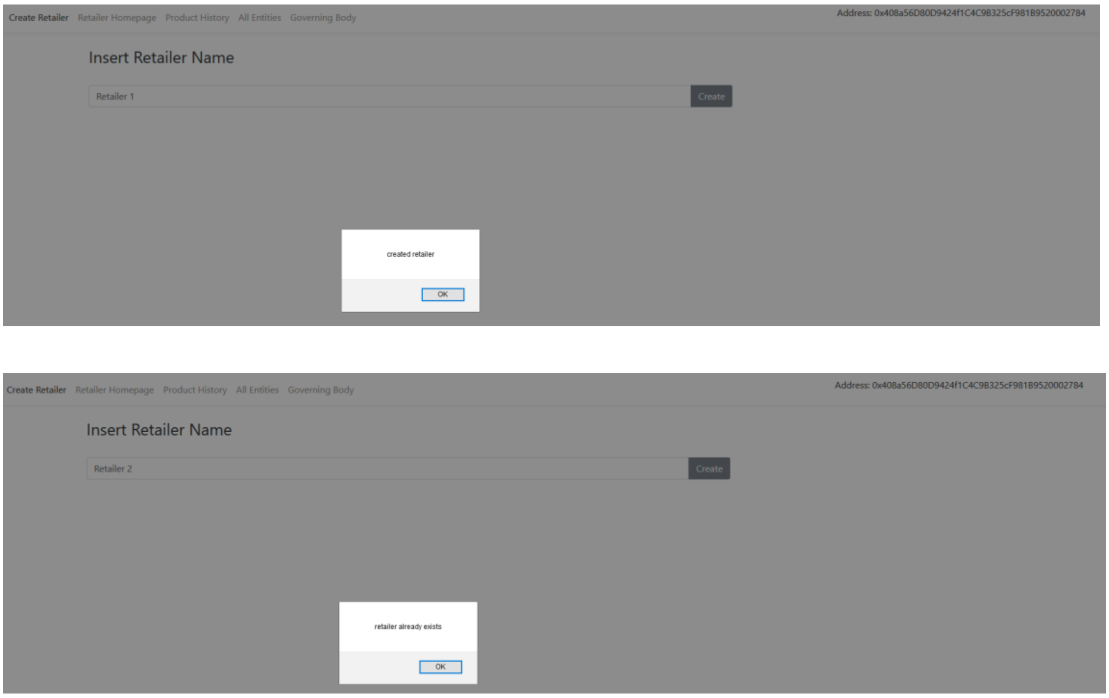


Figure 60: Multiple Retailers at the Same Address

Figure 61: Multiple Entities Output

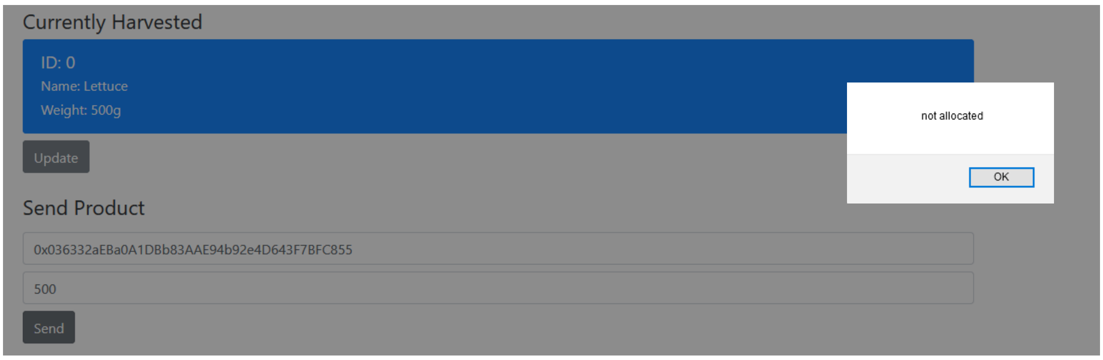


Figure 61: Sending To Unallocated Processor Address

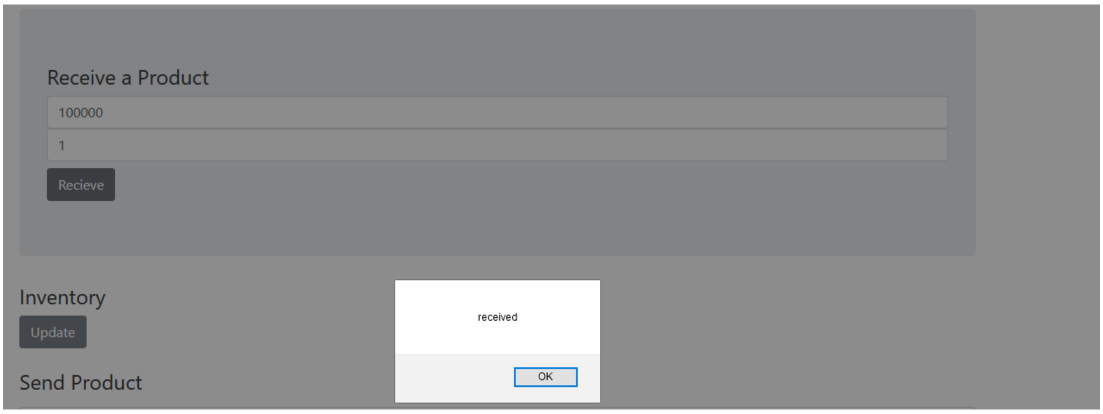


Figure 62: Receive Product That Does Not Exist

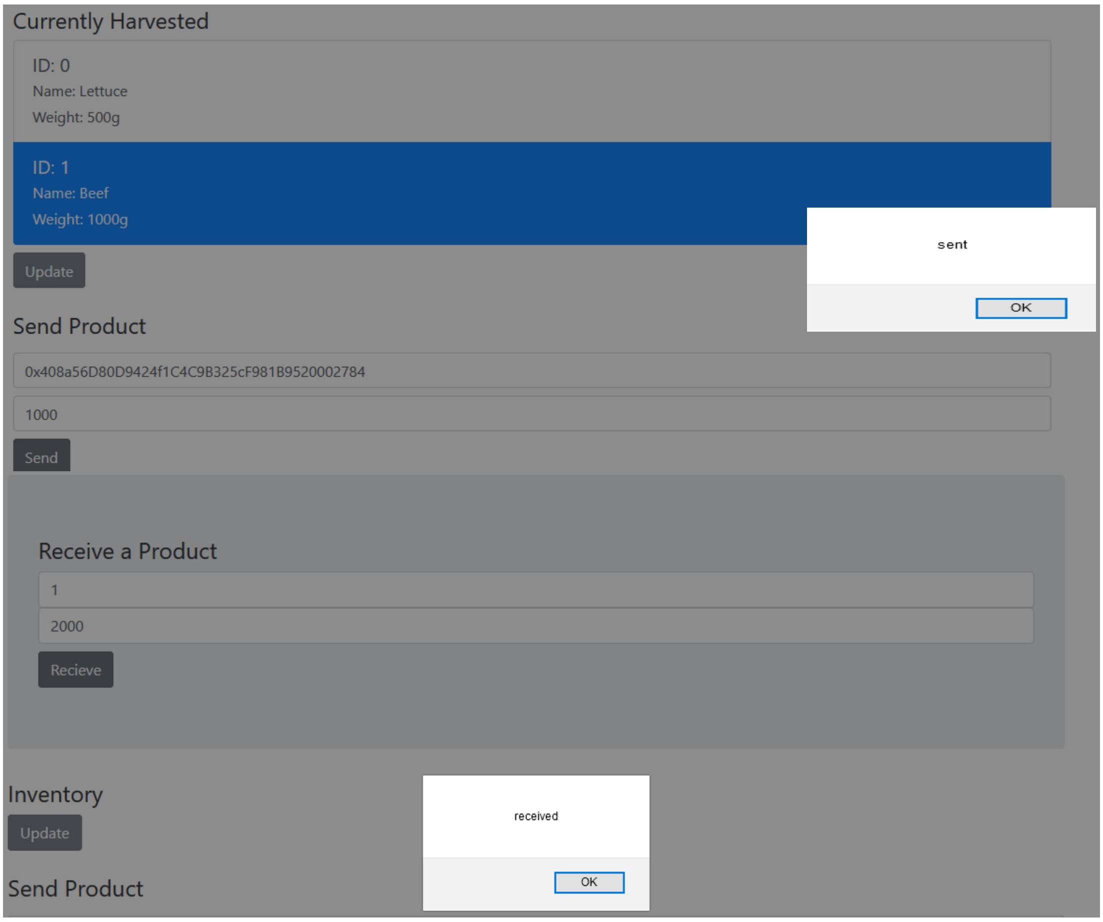


Figure 63: Receiving Product With Incorrect Weight

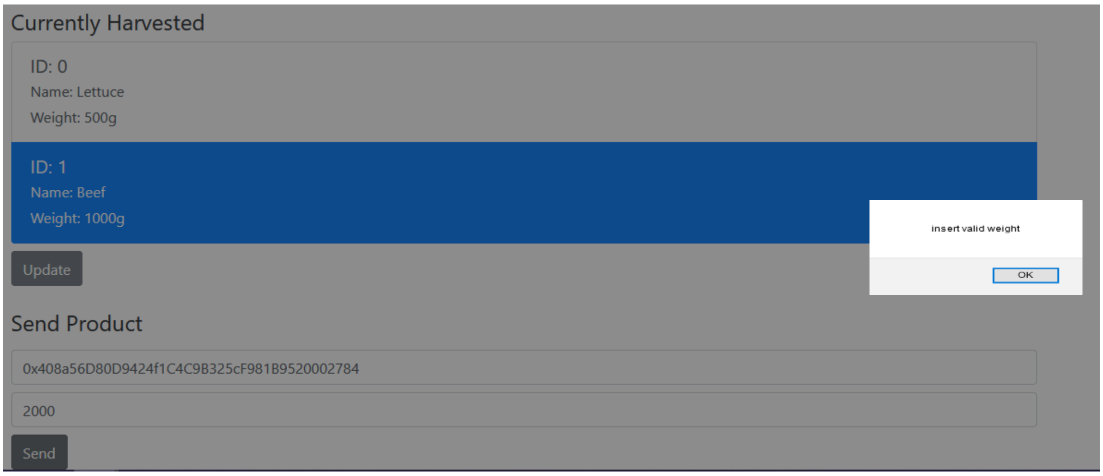


Figure 64: Invalid Weight

Figure 65: String in Integer Field

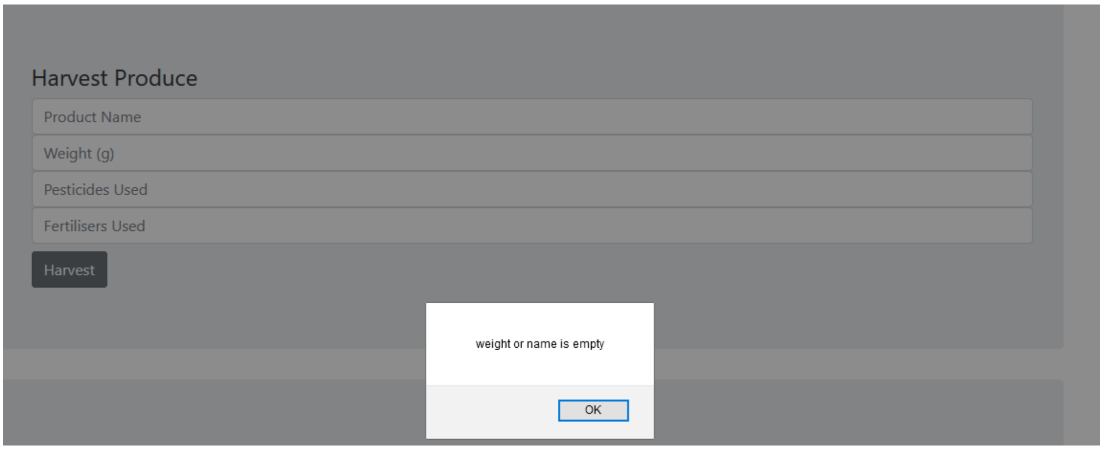


Figure 66: Empty Fields

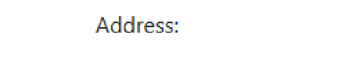


Figure 67: No Address Displayed When Not Connected To Blockchain