Automating Inventory Management

Routing through Sensor Networks

Design Document

Team Number: sdmay19-29
Client: Jimmy Paul (Crafty)
Adviser: Dr. Goce Trajcevski

Team Members:
David Bis — Meeting Facilitator, Back-End Developer
Hanna Moser — Meeting Scribe, Front-End Developer
Adam Hauge — Report Manager, Computer Network Architect
Ben Gruman — Resource Acquisition, Hardware Architect
Sam Guenette — Public Relations, Back-End Developer
Noah Bix — Documentation Manager, Hardware Architect

Team Email: sdmay19-29@iastate.edu
Team Website: http://sdmay19-29.sd.ece.iastate.edu/

Revised: December 2, 2018 / Version 2.0

Table of Contents

Table of Contents i

List of Figures ii

List of Tables ii

List of Definitions ii

1 Introductory Material 1

1.1 Acknowledgment 1

1.2 Problem and Project Statement 1

1.3 Operational Environment 1

1.4 Intended Users and Intended Uses 2

1.5 Assumptions and Limitations 2

1.6 Expected End Product and Other Deliverables 3

2 Specifications and Analysis 4

2.1 Proposed Design 4

2.1.1 Sensor Modules 4

2.1.2 Sensor Network 6

2.1.3 Database 8

2.1.4 Project Back-End 9

2.1.5 Front-End User Interface 10

2.2 Design Analysis 11

3 Testing and Implementation 14

3.1 Interface Specifications 14

3.2 Hardware and Software 14

3.3 Functional Testing 14

3.4 Non-Functional Testing 16

3.5 Process 17

3.6 Results 18

4 Closing Material 19

4.1 Conclusion 19

4.2 References 19

4.3 Appendices 21

List of Figures

Figure 1: Weight Sensor Circuit Diagram
Figure 2: Load Cell Wiring Diagram via Arduino (will change to ESP chip)
Figure 3: Sonar Sensor Wiring Diagram via ESP8266
Figure 4: 3-Way-Handshake Algorithm
Figure 5: Block Diagram of Solution Components
Figure 6: Load Cell Scale
Figure 7: Raspberry Pi Model 3B Pinout
Figure 8: Database Schema
Figure 9a: Data Display Page
Figure 9b: Update Threshold Modal
Figure 9c: Successful Update Notification
Figure 9d: Failed Update Notification
Figure 10a: Device Setup Page
Figure 10b: Update Product to Monitor Modal
Figure 10c: Successful Update Notification
Figure 10d: Failed Update Notification
Figure 11a: Gantt Chart for Prototype 1 Phase
Figure 11b: Gantt Chart for Prototype 2 Phase
Figure 11c: Gantt Chart for Prototype 3 (Minimal Viable Product) Phase
Figure 11d: Gantt Chart for Final Product Phase

List of Tables

(no tables at this time)

List of Definitions

Master-slave System: A type of software design where multiple smaller *slave* components carry out work, communicate, and are controlled by a single *master* component.
Sensory Device: A collection of sensors assigned to a specific product for the sensor network.
ADC: Analog to Digital Converter
SYN: Synchronize Sequence Number. A type of networking packet used to request or establish a connection.
ACK: A type of networking packet used to acknowledge a request for connection.
RFID: Radio Frequency Identification. Uses electromagnetic waves to identify and monitor the location of tags attached to certain objects.
UPC: Universal Product Code

1 Introductory Material

The objective of this project is to implement a system that can effectively monitor goods in office pantries and create efficient orders based off their current status. Using these orders, an optimal delivery route will be calculated to each office location.

1.1 Acknowledgment

The Inventory Automation Team would like to thank the Iowa State University Department of Electrical and Computer Engineering for providing this team with a professional experience, quality resources, and consultation with experts. The team appreciates the willingness of the Electronics and Technology Group (ETG) to help provide hardware and server components for the project. Special thanks also go to Iowa State's Dr. Goce Trajcevski for weekly consultation with regards to dealing with technical issues and moving forward throughout the development process. Appreciation also goes towards our client, Crafty LLC., for the given project. Special thanks go to CTO and co-founder of Crafty, Jimmy Paul, for making time to meet with the development team to gain more information and feedback throughout the project's development.

1.2 Problem and Project Statement

Crafty, LLC is a warehouse company that delivers food to office pantries. Their current infrastructure is based on having an employee at each of their client offices handling shipment orders and physically monitoring when the company's pantry needs to reorder certain products, which is prone to human error. Furthermore, warehouse truck routing becomes severely inefficient and expensive from restocking at individual offices based on separate orders.

Our objective is to provide an integrated solution that will enable more effective management of office pantries. Specifically, we aim to balance local (i.e. at the level of individual offices) and global (i.e. at the level of a geographic region) management in terms of both customer satisfaction and overall efficiency for Crafty. Towards that, our proposed solution consists of two main collaborative modules that we plan to develop: (1) a microcontroller device that will monitor the quantities of different items in an individual office pantry, coupled with other existing monitoring tools (e.g. weight scales, sonar sensors); (2) software tools that will be able to combine the multiple restocking notifications from different offices, and plan effective shipment (i.e. routes) and delivery.

1.3 Operational Environment

The microcontroller used to monitor office shipment levels is expected to be stored in a dry pantry area with a reliable power supply and WiFi connection. The current device consists of a microcontroller-connected "Smart-Bin" that is expected to fit on shelves. Any software setup with the device will be done through a web application. The web application will be available for Crafty employees stationed at office locations for setting up the physical device and in the warehouse for overseeing and handling shipment orders.

1.4 Intended Users and Intended Uses

There are three main users of our sensor network. Each user will interact with the sensor network through use of the web application component. The following are brief overviews of the three users:

1. *Pantry Employee*: Actor based in a given company office in charge of handling shipments sent from the warehouse. This user is in charge of setting up the microcontroller device and registering items for the device to monitor.

2. *Warehouse Employee*: Actor based in the company warehouse in charge of overseeing shipment orders and filling trucks with the proper items. This user will go into the application to view orders and routes. This user is expected to possess moderate technical experience.

3. *Warehouse Truck Driver*: Actor in charge of the physical delivery of items between the Crafty warehouse and client office pantries. This user will employ the online application to view their assigned shipping route for the day.

1.5 Assumptions and Limitations

Assumptions:

- Pantry sensor network has access to a reliable power source
- Pantry sensor network can connect to internet
- Pantry sensor network is not at risk of water damage
- Pantry Employee properly sets up sensor network and associates each monitoring device with the corresponding product being stored through the online application

Limitations:

- Accuracy of automatic orders is completely reliant on the accuracy of the sensors used in the sensor network
- Sensor network monitors product availability in bulk, not individually
- Sensor network can only make correct orders if the Crafty user properly establishes what type of products are being monitored

1.6 Expected End Product and Other Deliverables

The product deliverables will be split into (1) a proof-of-concept, (2) minimum viable product, and (3) a final product. The proof-of-concept will be delivered by the end of the fall 2018 semester. The minimum viable product will be delivered approximately two months into the Spring 2019 semester. Lastly, the finalized product will be delivered at the end of the Spring 2019 semester. An overall design manual of the product and its architecture will also be provided for any future development.

1. <u>Proof-Of-Concept (October 25th, 2018)</u>
    - The Proof-of-Concept prototype will prove the product's capabilities and potential to both monitor physical storage units and communicate that information to monitor and analyze. Users will be able to store a product in a single "Smart-Bin" which will send that information to an SQL Database. Users can log into a web application to register the device and monitor storage levels of the product in the Smart-Bin.

2. <u>Minimum Viable Product (March 15th, 2019)</u>
    - The Minimum Viable Product will be deployed for beta-testing. Multiple cheaper Smart-Bin devices will communicate with a single WiFi component that sends information to the database. The web application component will monitor information and build an appropriate and optimized shipment orders.

3. <u>Finalized Product (April 15th, 2019)</u>
    - The Finalized Product is ready to integrate for use with the client's existing infrastructure. The smart-bin device is expected accurately monitor multiple products. The user interface will also provide easy setup, handling, monitoring, and registering of smart bin devices. The database is connected to a Cloud Management System to analyze data and learn to better optimize shipping routes and customer orders.

2 Specifications and Analysis

In this section we will discuss our design decisions and approaches in detail.

2.1 Proposed Design

The design for this solution consists of five major components:

1. Sensor Modules
2. Sensor Network
3. Database
4. Project Back-End
5. Front-End User Interface

The sensor modules are used to approximate the quantity of any given item in the stockroom. Each sensor module will be registered by the barcode scanner so that it is assigned to a specific item in the stockroom. The sensor network is controlled by a master Raspberry Pi using socket communication with the sensor modules to continuously collect inventory data throughout the day to be sent to the database. The front-end of the web application allows users to login, view inventory levels, and edit monitoring device information. The back-end handles automatic reordering of products once they have reached their threshold level. Sections 2.1.1 through 2.1.5 describes the functionality of each component in more detail.

2.1.1 Sensor Modules

The sensor modules consist of a combination of weight sensors and sonar sensors. These sensors work together to calculate the weight and depth of the inventory used to output an estimated calculation on the total inventory. Each monitoring device has a weight sensor as a primary component to monitor product levels. The sonar sensor is intended to act as a backup to monitor when a package has been placed on or removed from the device.

The weight sensor (Figure 1) utilizes a bridge circuit which alters in total resistance as the load cell bends under the weight of items placed on top of it. The total resistance of the bridge circuit directly relates to a given output voltage. This voltage is converted to a digital value through an ADC and is then communicated back to the Raspberry Pi via an ESP8266 chip with the wiring specifications below (Figure 2). These values can then be quickly calibrated for each specific product on the weight scale. Calibration is done by taking a voltage reading for one item on a single scale and equating that value as one unit. As items are added and removed from the sensor, the value output by the sensor automatically adjusts with respect to the unit value. Once a reference exists for a given product, future sensors can be calibrated by leaving the sensor empty, assigning the resulting value to zero, and shifting the reference accordingly.

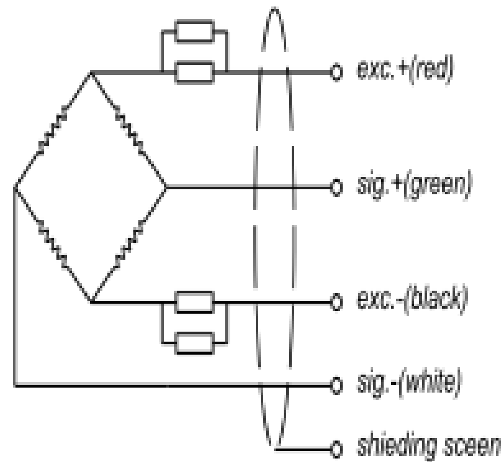

Figure 1: Weight Sensor Circuit Diagram

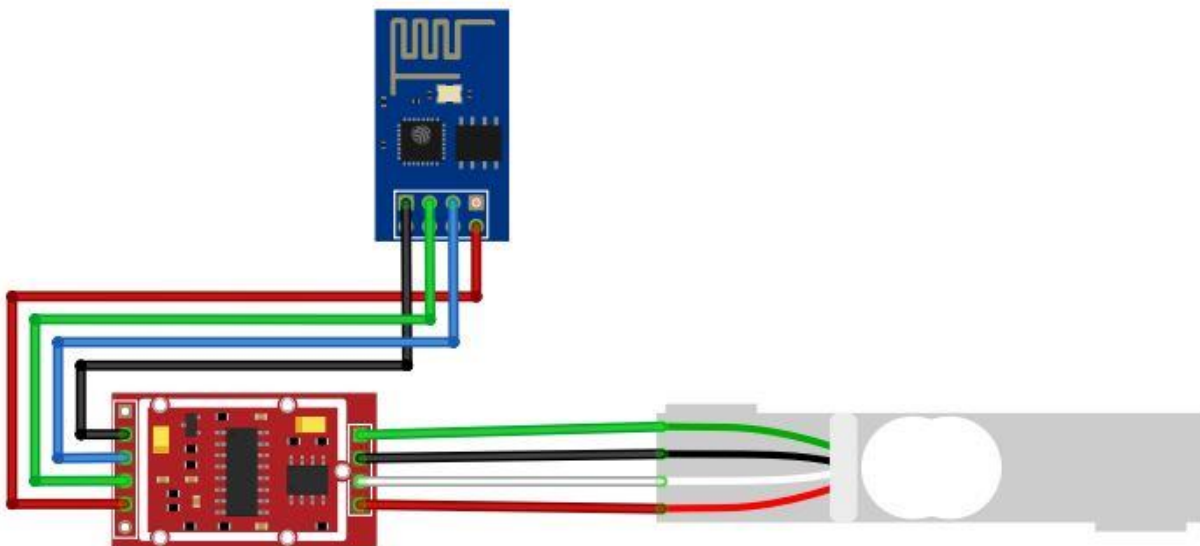

Figure 2: Load Cell Wiring Diagram via ESP8266

The sonar sensor sends a pulse signal out and detects the time it takes for the signal to reach an object and bounce back. The amount of time required for the pulse signal to return to the sensor is then reported as a value to the master Raspberry Pi. This sensor will use an ESP chip as well with the wiring specification in Figure 3. To make sense of these values, the sensors are calibrated to display distance using regression analysis. Once the sensor is calibrated, it then is able to track the depth of each stack of products.

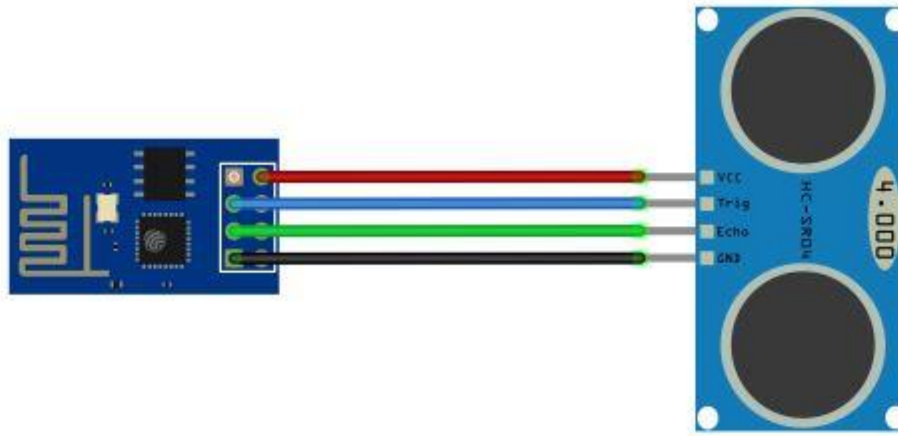

Figure 3: Sonar Sensor Wiring Diagram via ESP8266

A barcode scanner will be used in our design to register or change what products are being monitored in the pantry. Every product has its own Universal Product Code (UPC), which is represented by its barcode, which can be stored into our database. Therefore, when a new item is being brought into the pantry, the Crafty employee can simply scan the item before putting it on the weight scale. This way the employee will not have to browse through all of Crafty's products to find what they are looking for. This is particularly useful when selecting a product that has many different varieties of the same brand.

2.1.2 Sensor Network

The sensor network, controlled by a Raspberry Pi, is responsible for gathering all inventory data throughout the day. The Raspberry Pi is programmed to act as a server continuously waiting for sensor module clients to request a connection and transmit data. Connections are established through the use of a 3-way-handshake algorithm (as shown below and through figure 2) using the Raspberry Pi as the server and the sensor modules as clients. Upon receiving information from any of the stockroom's sensor modules, the Raspberry Pi packages the data to be sent over to the database before closing the connection.

The 3-way-handshake employs the following algorithm:

1. Sensor Module sends a SYN packet to Raspberry Pi to initiate connection.
    - SYN packet contains all necessary sensor module identification data.
2. Raspberry Pi receives SYN packet and sends a SYN-ACK packet to Sensor Module.
    - SYN-ACK packet acknowledges that the Raspberry Pi is ready to receive data.
3. Sensor Module receives SYN-ACK packet and sends ACK packet to Raspberry Pi.
    - ACK packet contains all necessary sensor data needed to calculate total inventory data.

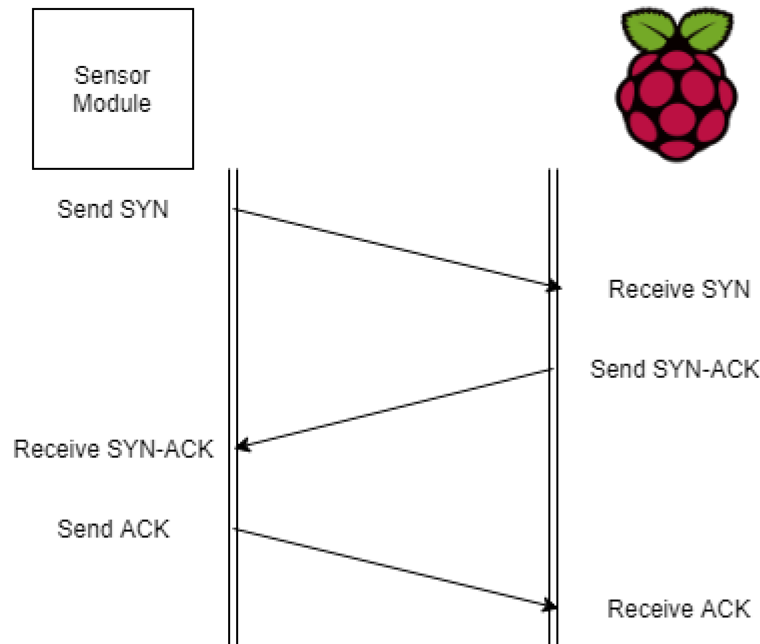

Figure 4: 3-Way-Handshake Algorithm for Network Communication

All communication between the Raspberry Pi and the Sensor Modules is handled via socket communication. Due to this, it is important to ensure that the server program is working consistently. Furthermore, ensuring a consistent server for handling raw data is required to meet the non-functional requirement for data integrity. One possible issue that could arise is that a client could request to connect to the server by sending a SYN packet, but never sends the proceeding ACK packet. This could be caused by a faulty sensor module or possibly as an attack on the sensor network. In order to mitigate this, all connections will be given a time constraint. All transactions that are not completed within the given time constraint are discarded. This ensures that the server is never busy for too long and that all sensor modules will have opportunities to connect to the server. An issue that arises with the 3-way-handshake connection is the possibility of a SYN flood attack on the server. SYN flood attacks occur when a client repeatedly sends SYN packets in an attempt to occupy all the memory on the server and prevent legitimate clients from making a connection. In order to mitigate this, the server only accepts packets from registered sensors and does not send any ACK packets to clients not identified in the sensor network. Any SYN packets from clients not recognized by the Raspberry Pi are discarded immediately.

The sensor network is set up through a series of preloading all necessary programs onto the ESP8266 wireless chips and the Raspberry Pi before connecting to power. All WLAN data needs to be loaded onto the Raspberry Pi's micro SD card in the file "/etc/wpa_supplicant/wpa_supplicant.conf". The Raspberry Pi is also pre-assigned a static IP address which the ESP8266 wireless chips are programmed to connect with. Because of this, the sensor network will be automatically setup upon each module powering on.

In order to preload WLAN connection information onto the Raspberry Pi, the following must be added to /etc/wpa_supplicant/wpa_supplicant.conf

```
network={ ssid="my-network-name"
          psk="my-network-pass"
          key_mgmt=WPA-PSK
        }
```

When it comes to setting up the sensor network, the Raspberry Pi can automatically detect what sensors modules are connected to the local network. Since the ESP8266 modules on each sensor module will have an open port, they will be discoverable to the Raspberry Pi which will store each module's identification locally after a successful connection. A unique ID will be assigned to each sensor module and forwarded to the database. If a sensor module loses connection with the Raspberry Pi, the database will not be updated during the regularly scheduled inventory measurement. Both the Raspberry Pi and the back-end will flag the sensor network as disconnected.

2.1.3 Database

The database will be implemented as a relational database with MySQL connected with primary and foreign keys between tables. The database schema is shown in the appendix 4.3.2. The database contains a single repository of certified devices as a security precaution. Whenever a new monitoring device is registered, the device must send an ID matching a unique ID in the *ActiveDevice* table that is not currently in use.

The *company* table used to keep track of *clients*, including their password, and address. This is used for both login security and as a key reference when storing orders. The product table stores information on every product offered by Crafty that clients can order from. This stores information like name, weight, serving size, etc.

The *monitoringDevice* table stores every monitoring device currently sending information to the database. Each monitoring device contains foreign key reference to the client that owns the device (*clients foreign key*), the current product being monitored (*product foreign key*), the product threshold, and the current inventory level.

The *Preferences* table keeps track of product thresholds specified by the user. Sometime multiple monitoring devices monitor the same product, which is why a central repository of data is needed to set overall threshold of a product. When a user specifies a threshold for a specific product, this table is updated and compared with the sum inventory level of all devices monitoring the corresponding product. This stores a reference to the product being monitored (*product* foreign key), the company the device belongs to (*company* foreign key), and threshold.

PantryData stores historical data of inventory levels over time. This data will be later used and analyzed to optimize shipping routes. This table stores the product, the company the data belongs to, date, and the total product weight measured on the corresponding date.

The *orderLog* table stores shipping orders and historical data of shipping orders. There is only one active shipping order at all time, which is specified by by a boolean value “newOrder”. Each entry also contains a shipping ID, and the date the shipping order was made.

The *orderItem* table represents the contents of each order from the *orderLog* table. Whenever a client is running low on an item, an entry containing the company id, the product being ordered, the quantity of the, and the order ID corresponding to the most active order ID in the *orderLog* table.

2.1.4 Project Back-End

The back-end will be implemented in a NodeJS environment using the ExpressJS framework. This component will interface directly with the database and process data for further analysis. The Express API will be made to manage data flow between the front-end and the back-end. There will be four types of HTTPS requests used: GET, POST, PUT, DELETE. GET calls will be used for retrieving data from the database. POST calls will be for only uploading new data. PUT will be used for updating existing entries in the database. DELETE will be used for removing entries from the database. Each endpoint will access the database using the Sequelize library. This is to take advantage of the object-relational mapping (ORM) properties of the database. Sequelize will pull information from the database and convert it into classes with the proper inheritance of information between datatables. This allows developers to make queries and access database information more efficiently. The Sequelize library is capable of being paired with the *sequelize-auto* library, which includes a command line interface that can take an existing database schema and generate the necessary data access files for each table. It is important to add the additional field *timestamps* with value *false* into each table data access file because the default value is true and would throw an exception otherwise.

The back-end will need to process the data on a regular basis. To do this, the *node-scheduler* library will be used to run a series of processes at of end of each weekday to determine and create orders for each client based on their inventory levels. This is done by comparing the current levels of each product with their specified threshold. If the current level is below the threshold, it is added to the order list. The items ordered are stored the *orderItem* table the entire order tracking information in the *orderLog* table. The orderLog table is used to track of Crafty’s order history. One orderLog contains multiple orderItems.

After the orders are determined, an optimal delivery route will be planned. An optimal route is determined by the route with the shortest transit time with the least amount of trucks. Additional parameters we must consider is that each truck should not be out for more than 8 hours, and that each truck has a limited capacity. We also assume that every stop along the route will take a constant amount of time. The *products* table will store information regarding packaging information, such as weight and size. A dynamic programming algorithm is used to determine the load contents of each truck, taking the weight and size of each product into consideration. Order contents will be entirely on the same truck in order to reduce the number of stops by the drivers.

Based on the contents of each truck, an optimal order of locations to stop must be determined. However, trying to find the most efficient path is similar to the Travelling Salesman Problem, which is an NP-Hard problem. Despite this, this is not expected to be a problem for performance because each truck is only able to carry no more than eight (8) orders, and therefore won't take more than eight stops. As a result, the time to determine the optimal route per truck is expected to be acceptable. The algorithm will have an undirected weighted graph data structure as one of the inputs, with vertices being delivery locations and Crafty warehouse locations, and edges being the segment of the road network connecting them. The weight assigned to the edges is the travel-time necessary to transport between two locations and is assumed to be the same in both directions, thus why we can use an undirected graph rather than a directed graph. The Google Maps API will be used to calculate transit times and routes. The Google Maps API is an ideal choice since it accounts for traffic and road closures in its time analysis. The 2-Opt algorithm is used to calculate the shortest route in the most optimal way. The 2-Opt algorithm finds an optimized route by searching through nodes and their edge connections and divide route as paths when 2 pairs of node connections overlap. Once the optimal delivery order is determined, the routes will then be divided amongst trucks, each filled with client orders according to the corresponding optimal route.

2.1.5 Front-End User Interface

The front-end of the web application is implemented using the ReactJS framework.The purpose of the UI is to provide the user with visualization of the pantry data, as well as provide them with an interface for configuring the sensor network. ReactJS allows for breaking down each web component into independent exportable objects, leading to easier implementation and testing. There are two main pages for the front-end: a data display and a device setup page. The user can navigate between the data display page and device setup page via the navigation bar provided at the top of the page. There is also a login page which provides for a secure login for the user using a unique username and hashed password. All front-end screen sketches can be found for reference in Appendix 4.3.3.

Upon login, the user is immediately taken to the data display page (Figure 9a). The data display page provides the user with data visualization of the inventory levels for a pantry.On this page a real-time filter is provided for the user so they can filter products within the pantry by multiple product names as well as their threshold status (whether the threshold of a given product has been reached or not). Each time the filter is changed, an HTTPS GET request is sent to the back-end to get all of the products to which the filter applies. All of the product in the pantry displayed after the filter is applied are in a React datatable created using ag-Grid, where each row represents a single product. In addition, each row is colored according to the product's inventory level. For instance, there will be two colors highlighting rows: green and red. Red indicates that a product's inventory level has reached its threshold value, while green indicate that a product's inventory level has not reached its threshold value. A user can change the threshold of a product on the data display page by clicking on a row, which will bring up a modal (Figure 9b). This modal will provide a text box where the user can enter a new threshold value. Once the user enters a valid threshold value, the apply button will become active. If the user no longer wishes to update the threshold value, they can select the CANCEL button in the upper

right-hand corner of the modal. If the user has entered a valid threshold value and still wishes to update, they can select the APPLY button and the modal will close followed by an HTTPS PUT request will being sent to the back-end. Upon the modal closing, a toast popup will indicate whether the update was successful or not (Figures 9c & 9d).

The device setup page (Figure 10a) consists of a real-time filter similar to the one on the data display page. However, the fields that the user can filter by on the device setup page are the device ID and the product name. Once the filter has been applied, each of the devices for which the filter applies will be displayed in individual rows. If the user would like to change the product that that device is monitoring, they can click on the corresponding device row, which will cause a modal to open (Figure 10b). The modal will contain a drop-down box of all of the products available in the database from which the user can select. Once the user has selected a product, then they can hit the APPLY button which will cause the modal to close and send an HTTPS PUT request to the database. Once the PUT request has been processed, a toast popup will display at the bottom of the page indicating whether the update was successful or not (Figures 10c & 10d). If the user has opened the modal for a device but no longer wishes to update the product being monitored, they can select the CANCEL button in the upper right-hand corner of the modal and no changes will be applied.

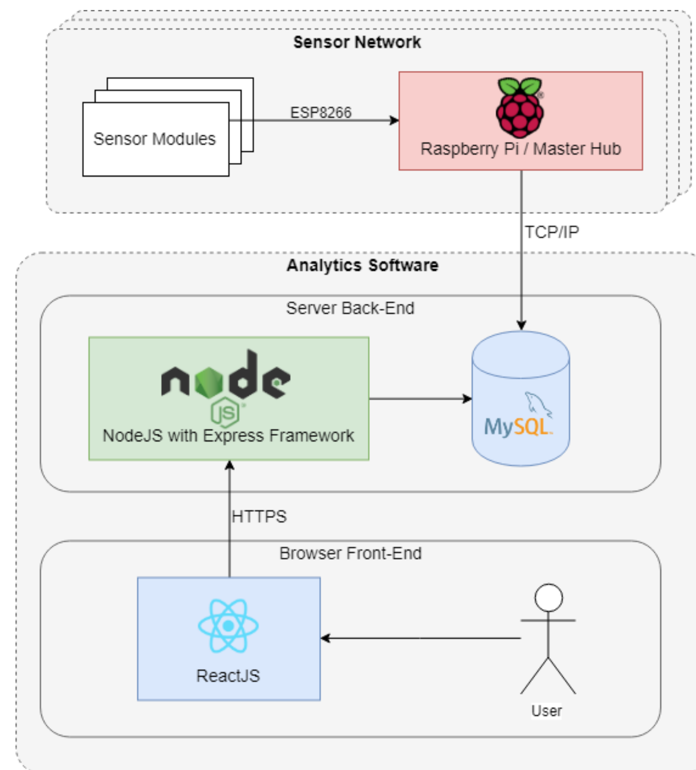

Figure 5: Deployment Diagram of Solution Components

2.2 Design Analysis

There are many pros and cons to this architectural design. The three-tier setup allows for easy maintenance and updates to software components without negatively affecting the rest of the

project. This also presents an ease-of-use for the user since they simply have to plug in their device, and then finish the setup through a simple form on the web-application.

In terms of physical sensors, RFID sensors were thought of as an alternative solution to the sensor network. However, the other sensors were selected for their simplicity and cost factor. While RFID sensors would complement our design well, the initial cost of RFID sensors and the capitalized cost of RFID sensor tags were concluded to be too expensive for our project. Labor is also a factor with the RFID sensor, Crafty employees would have to spend time and labor placing tags on every item that is shipped in.

Weight sensors will be much more effective as the primary sensor in our network. The weight sensor eliminates the capital cost that comes with an RFID sensor. Weight sensors also reduce the labor factor of applying RFID tags to each Crafty product. With this solution the error factor should be minimized as well. This is due to the fact that the weight of each product has to be within a certain range instead of an exact value.

An alternative considered for the master computer for the sensor network was an Arduino microcontroller, as opposed to the Raspberry Pi. While both options provide sufficient features to satisfy the requirements, the Raspberry Pi was selected due to its ease of use for development purposes. An important difference is the Raspberry Pi's on-board operating system (OS). A non-functional requirement for the project was scalability. Since Crafty services multiple pantries, it is important that the sensor network is modularly configurable. Since the OS is on-board the Raspberry Pi, it enables the user to easily replace and alter sensor array configurations. The OS enables changes at run-time through a program written for the device. One last difference is that the Raspberry Pi has an on-board wireless capability, whereas the Arduino does not. Since the master computer must make calls to upload data to the database, the feature is convenient.

The use of wireless socket communication throughout the sensor network allows for easy sensor module organization: the sensor modules will be able to be placed anywhere in the stockroom in any possible configuration. This means that our solution will be able to be implemented in any possible stockroom given that the assumptions of constant power and network connection hold true. Unfortunately, using the socket communication also means that each sensor module and the Raspberry Pi must always be connected to the WLAN in the stockroom. However, connecting each sensor module and the Raspberry Pi to the network requires an extra step of connecting each piece of hardware to the Wi-Fi network.

By using one-way communication from monitoring device to the database, the accuracy of the web-component analysis will rely completely upon the data being sent. Accuracy is also completely reliant on the weight sensors' accuracy.

Using an SQL-Database can also make things a little more difficult for long-term analysis. Unlike databases like MongoDB, which has its data nested to easily obtain association in information; SQL adds a layer of difficulty through its key-value architecture, making it difficult to obtain and group information in the proper format to analyze. However, since our primary

objective is inventory and order tracking, a row-column database setup was an ideal choice for storing and retrieving information easily and optimally.

The NodeJS back-end is beneficial in that it is asynchronous by nature. This makes it easy to manage multiple simultaneous processes. Although we do not expect our back-end to experience heavy traffic, the environment will help out mitigate any potential concurrency issues. The NodeJS environment also offers up a number of easily integrated external libraries, or node modules. This reduces the amount of necessary code to write and simplify new code. However, this runs risk due to the increasing number of dependencies. If any of the libraries used stops being maintained or has changes that affect our application's performance, we will have to be reactive to the changes of our dependencies.

The route optimization is challenging to optimize as it is equivalent to solving the "travelling salesman problem", which is an NP-hard problem. However, we do not expect on having the trucks to travel to a large number of locations per day, so we can assume the algorithm run-time will be negligible. To avoid an exponential computing time for calculate truck routes, the 2-Opt algorithm is determined to be the most optimal implementation. Though harder to implement, it is the better choice compared to brute-forcing the best route plan. This also makes our software more scalable as Crafty gets more clients and therefore, more delivery locations.

ReactJS will help break down the front-end into separate components, allowing for each to be independently implemented and tested. This helps ensure that the front-end will appropriately satisfy each use-case.

3 Testing and Implementation

This section will discuss our step by step testing process for each individual component in detail.

3.1 Interface Specifications

In any project of this size, testing is necessary in order to ensure that all components work properly both individually and together as a unit. Our sensor modules will be tested using python scripts designed to continuously output data from the ADCs connected to the sensors. Different amounts of inventory will be used with the sensor modules in order to ensure that the values returned by the ADCs are making realistic approximations to the true inventory values. The master Raspberry Pi will run test scripts that takes manual input through a socket connection and pushes the data into the database. This will ensure that the Raspberry Pi is able to accept data and send it to the database seamlessly.

3.2 Hardware and Software

The sensor modules are tested through the use of python scripts designed to continuously output readings from the connected ADCs to be displayed on a monitor. These scripts will be used to ensure that the data output from the sensor modules accurately depicts the amount of inventory being measured. Furthermore, when the amount of inventory is changed, these scripts will show whether or not the sensor modules are able to detect the change.

In order to test the sensor network, a team laptop will act as a client by sending premade packets to the Raspberry Pi via a socket connection. This will ensure that the Raspberry Pi is able to make viable connections with the sensor modules. Furthermore, incorrectly formatted data will be sent to the Raspberry Pi to ensure that bad packets are discarded from the network to ensure that the network will not be kept busy by mis-formatted packets. Upon successful testing of the communication sockets, the Raspberry Pi will send test data to the database in order to ensure a successful database connection.

To automate the testing on the back-end, the Mocha and Chai testing libraries will be used. These libraries will be responsible for testing all data processing modules, such as the truck loading algorithm and route planning algorithm. Postman, a GUI software used to simulate REST API calls, will be used to test the API. Postman provides various features that help automate the test process, such as preparing test suites for calling the different endpoints, as well as feeding test data into the system.

3.3 Functional Testing

To test the validity of the sensors, various products of different shapes and weight will be placed on the scales. A snapshot of the data will be taken before and after a product is added or removed. The two snapshots will then be compared to determine the accuracy of the sensors. This applies to both the load cell and the sonar sensor.

The act of adding or removing new sensor arrays in the system will be tested by a simple metric of success. Success is measured by the master Raspberry Pi's ability to detect and receive accurate data from any sensor array. If the master Raspberry Pi is able to detect the new sensor array and receive accurate data from it, the test is deemed successful, and a failure if otherwise. The same applies for removing a sensor array.

A testing suite will be used to test all of the written project software. This testing suite will include test cases using:

- Expected Data
- Fringe Data
- Failure Data

The tests will be run in a variety of stages starting with unit tests. The following stage will consist of testing after integration with the system. The purpose of this stage is to test not only the newly integrated module, but to test existing modules to ensure they are still satisfying their requirements. The final stage will be user testing, which will focus on validation of the user experience. The purpose of this stage is to test and verify the practicality of the solution. This stage will include testing front-end usage and sensor network deployment.

The objective of these tests is to verify that the solution saves the user time in the inventory management process. Through the tests, both function and non-functional, sufficient evidence will be collected to prove a time save.

3.4 Non-Functional Testing

The project will be tested for satisfaction of the following non-functional requirements: scalability, data integrity, availability, deployment, usability, resilience. Procedures and success and failure criteria will be provided.

Scalability

Procedure
1. Begin with *k* sensory devices connected to the sensor network
2. Set-up a new sensor device to connect to the sensor network
3. After completion, open front-end and check if *k+1* sensory devices are connected

Success Criteria: *k+1* sensory devices are identified on the sensor network
Failure Criteria: < *k+1* sensory devices are identified on the sensor network

Data Integrity

Procedure
1. Begin monitoring sensor data from Raspberry Pi. Save sensor data in a *.csv* file.
2. Transmit sensor data to database. Store data into *.csv* on back-end.
3. At the end of the observation period, compare the two *.csv* files to determine equality

Success Criteria: The two files are equivalent
Failure Criteria: The two files are not equivalent

Availability

Procedure
1. Schedule using the *node-scheduler* library an inventory process to happen at the end of every weekday
2. At the start of a new day, check if a new order and delivery route has been generated

Success Criteria: A new order and delivery route has been generated
Failure Criteria: A new order and delivery route has not been generated

Deployment

Procedure
1. Set-up sensory devices in desired location
2. Run Raspberry Pi sensory device discovery protocol
3. Check if three-way handshake is successful for each new sensory device

Success Criteria: Each new sensory device is discovered by the master Raspberry Pi
Failure Criteria: A sensory device is not discovered by the master Raspberry Pi

Usability

Procedure

1. Track the time it takes for one individual to manually count the inventory
2. Based on the inventory counts, determine an order for the day, end timing of process
3. Track the time it takes for the sensor network to log inventory and produce order
4. Compare manual and automatic times

Success Criteria: Automatic time is less than manual time
Failure Criteria: Manual time is less than automatic time

Resilience

Procedure

1. Once per day, add or remove product from the sensory device
2. Repeat process for a month
3. After a month of moving product, check to make sure the sensory device is not damaged

Success Criteria: The sensory device is not damaged
Failure Criteria: The sensory device is damaged

3.5 Process

All sensors are being tested by comparing the quantity measured with the true quantity of different products. This will ensure that the combination of sensors being used are sending accurate data visible to our users.

The analytics software will face unit and user testing to assure accuracy. This involves the use of mock data and simulations. The application will also be tested by actual users in order to gather feedback and observe areas to refine and improve.

The primary features will be implemented independently of the other features. Testing features that require the products of other features will be supplied with dummy data to simulate the entire functioning system in order to isolate specific features. This will be beneficial when testing the analytics software, which as mentioned in section 3.2, requires inventory data to be sent from the sensor network in order to be successful. Each feature will have an automated testing suite associated with it, so that it can be tested as an isolated feature, and retested after itself and other features have been implemented into the system in order to prevent regression errors.

An Agile development process will be followed, with weekly meetings and constant communication, including weekly meetings with the client. Phase 1 of the development process begins with consulting with the Crafty client on their initial problem and desired solution. From there, the team discussed and refined the solution and architectural design plan approved by the client. The development team now goes through a set of sprints lasting one or two weeks. We

then demo what is currently developed to the client to get feedback, likes, dislikes, requested changes, and issues to both be sure we are developing the best product while keeping the client updated on our progress. Phase 2 involves developing a Proof-of-Concept to prove our solution acceptable and possible for the client. Phase 3 of the development process involves building the minimal viable product that adequately solves the client's issues and can be beta-tested for further refinement. Phase 4 involves finalizing the product, fixing all existing issues and finalizing all promised features before handing the product off to the client. For further information on the development phases, refer to Figure 11 in the appendix.

3.6 Results

We are only halfway into this two-semester project. As a result, no sufficient results have been concluded. However, tests to support our design decisions have been done and are included in both the Design Analysis (section 2.2) and Testing and Implementation (section 3) sections. An image of our prototype load cell scale is shown below.

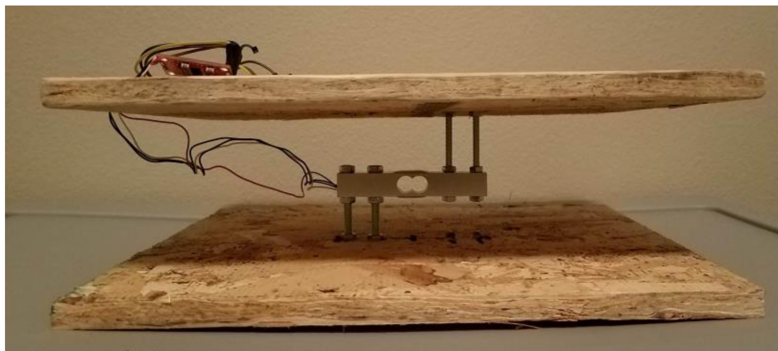

Figure 6: Load Cell Scale

4 Closing Material

The conclusion section will reflect upon the current status of the solution and future goals for the team.

4.1 Conclusion

Crafty, LLC's current infrastructure is based on a middleman, creating an unnecessary expense that is prone to human error. This causes warehouse truck routing to become severely inefficient and expensive from restocking at individual offices based on separate orders.
Our solution consists of a microcontroller device that automatically monitors the amount of items in an office pantry and places orders to the Crafty warehouse when the office has reached its minimum threshold value for certain pantry items. We will then develop software that processes the current orders from all office orders to optimize shipment routes.

At this point in the project, our team has implemented the basis for the major components of our project: a Raspberry Pi sensor network and an analytics software application. We have been able to create a prototype so far that can take a weight measurement from the load cell and send it to through the Raspberry Pi to transmit it out to the database. The analytics software is able to interpret the data and determine whether or not an item has to be reordered. It is also capable of displaying the inventory data on a ReactJS front-end through a web browser.

We plan to continue development based on the content of this document. We believe that the design presented will be able to assist Crafty and its clients to a more effective and efficient inventory management system. There will be an emphasis moving forward on testing the implementations to ensure the accuracy of the data and resulting outputs from processing that data. The end product will be capable of tracking the inventory of a pantry and using the inventory data from multiple pantries to determine an efficient route to resupply pantries with products that are at a low quantity.

4.2 References

Intellectual Reference

1. Impinj.com. (2018). Automated Inventory Management with RAIN RFID | Impinj. [online] Available at: https://www.impinj.com/solutions/healthcare/inventory-management/.
2. Wind.cs.purdue.edu. (2018). [online] Available at: http://wind.cs.purdue.edu/doc/adhoc.html.
3. GeeksforGeeks. (2018). Computer Network | TCP 3-Way Handshake Process - GeeksforGeeks. [online] Available at: https://www.geeksforgeeks.org/computer-network-tcp-3-way-handshake-process/.
4. Interserver Tips. (2018). What is SYN Flood attack and how to prevent it? - Interserver Tips. [online] Available at: https://www.interserver.net/tips/kb/syn-flood-attack-prevent/.

5. Python, R. (2018). Socket Programming in Python (Guide) – Real Python. [online] Realpython.com. Available at: https://realpython.com/python-sockets/.
6. Raspberrypi.org. (2018). Networking Lessons | Raspberry Pi Learning Resources. [online] Available at: https://www.raspberrypi.org/learning/networking-lessons/rpi-static-ip-address/.

Pricing Reference Websites:

1. Electronics, S. (2018). SEN-13329 SparkFun Electronics | Sensors, Transducers | DigiKey. [online] Digikey.com. Available at: https://www.digikey.com/product-detail/en/SEN-13329/1568-1852-ND/7393715/?itemSeq=272691527.
2. Electronics, S. (2018). SEN-13879 SparkFun Electronics | Sensors, Transducers | DigiKey. [online] Digikey.com. Available at: https://www.digikey.com/product-detail/en/SEN-13879/1568-1436-ND/6202732/?itemSeq=272691544.
3. Industries, A. (2018). ADS1015 12-Bit ADC - 4 Channel with Programmable Gain Amplifier. [online] Adafruit.com. Available at: https://www.adafruit.com/product/1083.

4.3 Appendices

4.3.1 Raspberry Pi Model 3b Datasheet

- Industries, A. (2018). ADS1015 12-Bit ADC - 4 Channel with Programmable Gain Amplifier. [online] Adafruit.com. Available at: https://www.adafruit.com/product/1083.

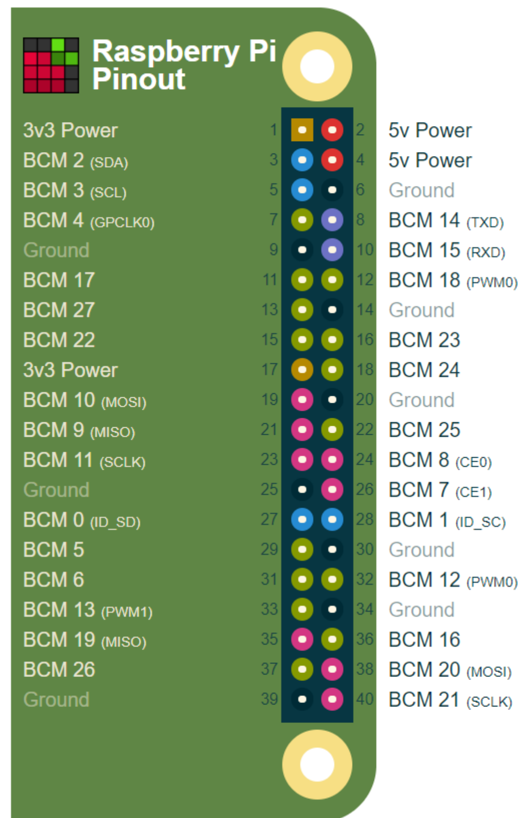

Figure 7: Raspberry Pi Model 3B Pinout

Pinout.xyz. (2018). Raspberry Pi GPIO Pinout. [online] Available at: https://pinout.xyz/.

4.3.2 Database Schema

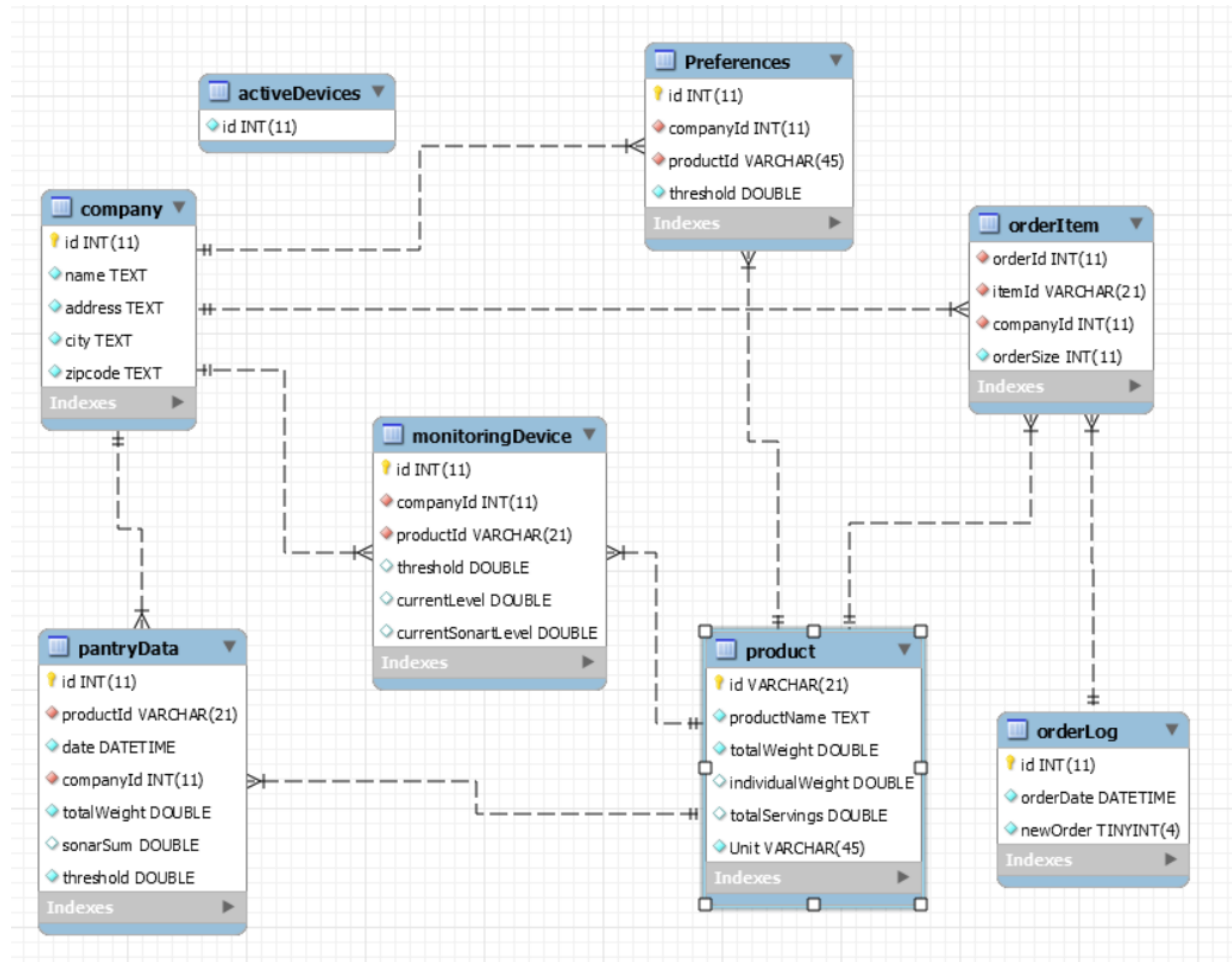

Figure 8: Database Schema

TAL220 Load Cell Datasheet:

Cdn.sparkfun.com. (2018). [online] Available at: https://cdn.sparkfun.com/datasheets/Sensors/ForceFlex/TAL220M4M5Update.pdf [Accessed 26 Nov. 2018].

4.3.3 Front-end Screen Sketches:

Crafty

Devices Inventory

Product Name(s)

<name>

ProductName1
ProductName2
ProductName3
ProductName4

Clear All

Reorder Status

Reordered

Product Name	Product ID	Inventory Level	Reorder Status	Threshold
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>

Figure 9a: Data Display Page

Crafty

Devices Inventory

Product Name(s)

<name>

ProductName1
ProductName2
ProductName3
ProductName4

Clear All

Reorder Status

Reordered

Product Name	Product ID	Inventory Level	Reorder Status	Threshold
<name>	<ID>	<level>	<status>	<threshold>
<name>				<threshold>
<name>				<threshold>
<name>				<threshold>
<name>				<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>

Update Threshold: <Product Name> CANCEL

Threshold <New Threshold> <units>

Update

Figure 9b: Update Threshold Modal

Crafty

Devices

Inventory

Product Name(s)

<name>

ProductName1
ProductName2
ProductName3
ProductName4

Clear All

Reorder Status

Reordered

Product Name	Product ID	Inventory Level	Reorder Status	Threshold
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	[illegible]	[illegible]	[illegible]	<threshold>
<name>	[illegible]	[illegible]	[illegible]	<threshold>

Product <Product Name> Updated Successfully

Figure 9c: Successful Update Notification

Crafty

Devices Inventory

Product Name(s)

<name>

ProductName1
ProductName2
ProductName3
ProductName4

Clear All

Reorder Status

Reordered

Product Name	Product ID	Inventory Level	Reorder Status	Threshold
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>
<name>	<ID>	<level>	<status>	<threshold>

Product <Product Name> Update Failed

Figure 9d: Failed Update Notification

Crafty

Devices Inventory

Device ID

<ID>

Product(s) to Monitor

<name>

ProductName 1
ProductName 2
ProductName 3
ProductName 4

Clear All

Device ID	Product Name
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>

Figure 10a: Device Setup Page

Crafty
Devices
Inventory
Device ID
<ID>
Product(s) to Monitor
<name>
ProductName 1
ProductName 2
ProductName 3
ProductName 4
Clear All

Device ID	Product Name
<ID>	<name>
<ID>	
<ID>	
<ID>	
<ID>	
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>

Update Product to Monitor: <Device ID> CANCEL
Product to Monitor <Product Name>
Update

Figure 10b: Update Product to Monitor Modal

Crafty

Devices

Inventory

Device ID

<ID>

Product(s) to Monitor

<name>

ProductName 1
ProductName 2
ProductName 3
ProductName 4

Clear All

Device ID	Product Name
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	

Device <Device ID> Updated Successfully

Figure 10c: Successful Update Notification

Crafty

Devices

Inventory

Device ID

<ID>

Product(s) to Monitor

<name>

ProductName 1
ProductName 2
ProductName 3
ProductName 4

Clear All

Device ID	Product Name
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	<name>
<ID>	

Device <Device ID> Update Failed

Figure 10d: Failed Update Notification

4.3.4 Gantt Charts

	Task Title	9/3/18	9/10/18	9/17/18	9/24/18	10/1/18	10/8/18	10/15/18	10/22/18
Prototype 1	Determine Team Values and Norms	■							
	Gather Requirements		■	■					
	Design Initial Solution Architecture		■	■	■				
	Implement Initial Solution Architecture			■	■	■	■		
	Complete Prototype					■	■	■	■
	Testing							■	■
	Documentation							■	■

Figure 11a: Gantt Chart for Prototype 1 Phase

	Task Title	10/8/18	10/15/18	10/22/18	10/29/18	11/5/18	11/12/18	11/19/18	11/26/18	12/3/18
Prototype 2	Provide Database Information	■								
	Register Device		■							
	Implement Sensors		■	■	■					
	Register Items			■	■					
	Database-Model Entity			■	■	■	■	■		
	Create Order					■	■			
	Testing								■	■
	Documentation								■	■

Figure 11b: Gantt Chart for Prototype 2 Phase

	Task Title	1/14/19	1/21/19	1/28/19	2/4/19	2/11/19	2/18/19	2/25/19	3/4/19	3/11/19
Prototype 3 (Minimal Viable Product)	Handle Multiple Sensors (Master-Slave)	■	■	■	■	■				
	Handle Multiple Orders	■	■	■						
	Database Refinement		■	■						
	Route Optimization				■	■	■	■	■	■
	Testing							■	■	■
	Documentation								■	■

Figure 11c: Gantt Chart for Prototype 3 (Minimal Viable Product) Phase

	Task Title	3/18/19	3/25/19	4/1/19	4/8/19	4/15/19	4/22/19	4/29/19
Final Product	Cloud Integration	■	■	■	■	■		
	Predictive Analytics Component		■	■	■	■		
	Final Testing and Troubleshooting			■	■	■		
	Final Documentation					■	■	■

Figure 11d: Gantt Chart for Final Product Phase