MIMS

Mobile Inventory Management System

Design Document

Patrick Richeal, Brandon Ostasewski,
Dominic Marandino, John Donahue, Matthew Finnegan,
Paul Sigloch, Sagarika Kumar, and Steven Bruman

Last updated March 8th, 2019

Table of Contents

1. High Level Design

The purpose of the MIMS product is to provide businesses that already have product, inventory, and sales databases in place a modern and easy to use frontend system to get the information that employees need during the day to day of their jobs. This consists of a mobile application for employees to search and find information on products, and an admin web application for managers to use to manage the users of the mobile app.

1.1 - System Architecture

The major components that need to be built for the MIMS product are the two front end applications (the mobile app and the web app), an API for both front end applications to communicate with, and a database to store information on things like the users of our system and the businesses using our system. As specified above, this product is meant to hook into an existing database of products, inventory, and sales, so such a database is not actually apart of the MIMS product. To model the functionality of the MIMS product, however, one of these databases will be constructed and implemented into our API. This architecture can be seen in figure 1.1, where arrows indicate communication between components.

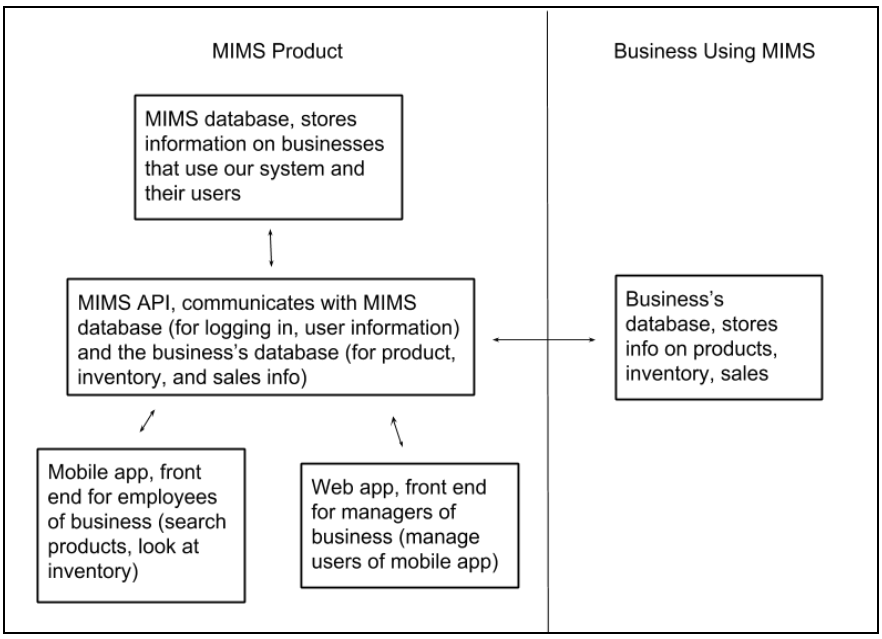

Figure 1.1, system architecture

1.2 - Business Adoption

A typical adoption process for a business like a grocery store would first involve us implementing their databases into our API, which would allow our mobile and admin application to function properly with their data. Then, a higher level day to day manager of the grocery store would be given a login for the admin web application to login and create user accounts for his or her employees. The employees could then use that information to login to the mobile application and easily search products in their database and get the information they need.

2. Problem Solving Approaches

2.1 - Multiple Businesses

After deciding upon the separation of the MIMS product and the business databases as described in section 1.1, the next major issue was to design an API that could allow for a single format of data for products and inventory to be returned from central endpoints for all businesses. This allows for both front end applications to remain completely unchanged in the event of a new business adopting the MIMS system. The only changes that would need to be made would be to add the new business to our database of businesses, and implement a few functions that return their product and inventory data in the correct format for our system.

To model how this will work in the actual API, let us look at an example for the *GET /products/* endpoint, an endpoint meant to return a list of products based on input queries. The job of the *GET /products/* endpoint will essentially be to figure out who is making the request, figure out what business that user is a part of, and utilize the appropriate function (to get products) that we have implemented for that specific business. It would then be our job to ensure that each of these functions implemented for each business return the product data in the exact same format for consumption by our frontend applications.

This solution for this problem of multiple businesses was ultimately chosen so we could easily add new businesses to our system. It is inevitable that we would need to actually program everytime we want to incorporate a new business into our system, but this design allows that process to happen entirely on the API so we won't have to worry about updating old versions of applications or distributing different applications for every business. The API will always be the software within our control entirely, so it made the most sense to keep this logic at that level.

2.2 - Populating the Sample Business Database

As described in section 1.1, we will be building a sample business database that resembles a database that could potentially be eventually integrated into our system from an actual business. In the scenario of a business adopting the MIMS product, the business would already have a database of products and sales that is

changing everyday and updating live as sales are made and inventory is brought in. This is something that even a database we create as a sample wouldn't have, live day to day updates and real information. To combat this problem, we will manually add a small number of sample products to our fake business database and then create a simulation that models the daily transactions that may take place on the database in the real world. This simulation will simply run and fill up the fake business database with sample data so we will be able to show off the MIMS product in actual use.

3. Design Choices

3.1 - Separation of MIMS and Business Database

One of the initial design choices made as a team was to create a clear distinction between the database for storing MIMS product related information such as user accounts and the businesses that use the MIMS product and the databases that are owned by the businesses that the MIMS product needs to interface with. This distinction is essential to the way that the MIMS product allows multiple businesses to utilize the same API endpoints and front end clients.

The need for this distinction was discovered when we started to think about what type of information we would need to store in a database to have this product work. When we agreed upon the fact that the MIMS product would work by having other businesses integrate their data with our API, the separation of data became very clear. There is data owned and operated by the business, which includes products, inventory, and sales information. On the other end, data needs to be stored for the MIMS product to function properly. This data, as said previously, includes user accounts and storing the actual businesses that use MIMS for our records and purposes of identifying what business a user is apart of. The separation of data follows the principle that the MIMS product simply provides a system to sit on top of existing business databases.

4. Frontend Mockups, Navigation, and Functionality

4.1 - Mobile App

Login Page

The login screen can be seen in figure 4.1. It allows the user to authenticate themselves and login to the app. It will use the *POST /login/* endpoint of the API. It can only navigate forward to the product search page.

Product Search Page

The product search page can be seen in figure 4.2. It allows the user to search for products. It will use the *GET /products/* endpoint of the API. It can navigate back to the login page if the user logs out or forward to a product details page if a product it selected.

Product Details Page

The product details page can be seen in figure 4.3. It allows the user to look at information regarding a specific product like its various identifiers and current inventory. It will use the *GET /products/<item_code>/* endpoint of the API. It can navigate back to the product search page or forward to the product inventory movement or product inventory forecast page.

Product Inventory Movement Page

The product inventory movement page can be seen in figure 4.4. It allows the user to look at how the inventory of the product has changed over time. It will use the *GET /products/<item_code>/movement/* endpoint of the API. It can only navigate back to the product details page.

Product Inventory Forecast Page

The product inventory forecast page can be seen in figure 4.5. It allows the user to look at predicted future inventory transactions for the product. It will use the *GET /products/<item_code>/forecast/* endpoint of the API. It can only navigate back to the product details page.

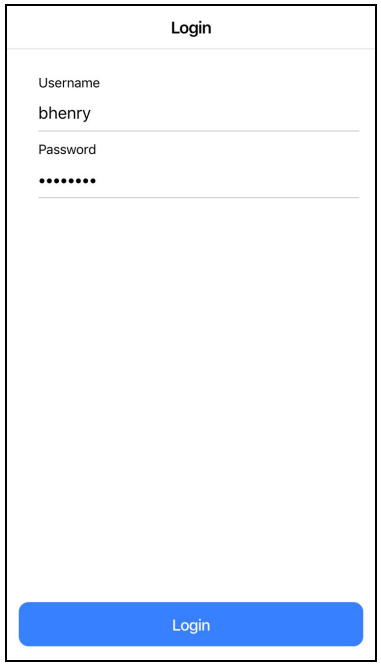

Figure 4.1

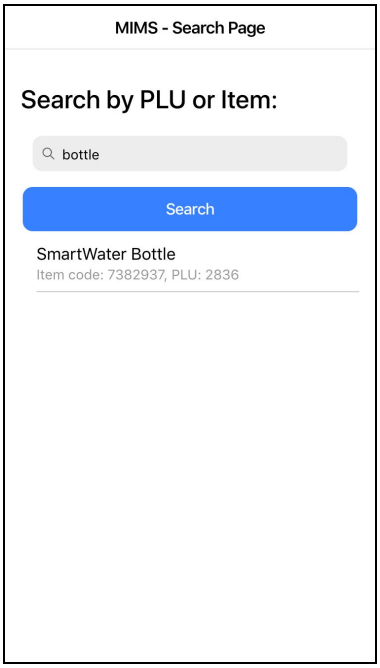

Figure 4.2

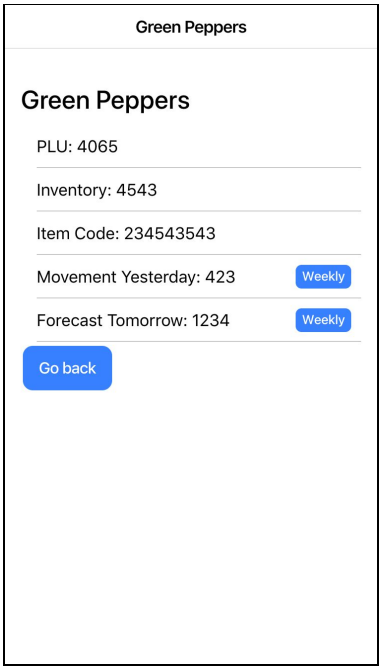

Figure 4.3

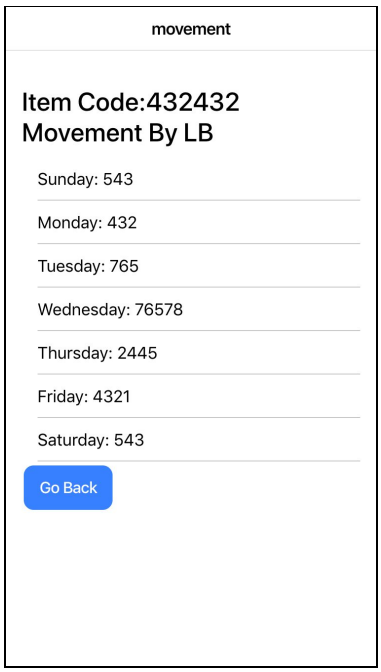

Figure 4.4

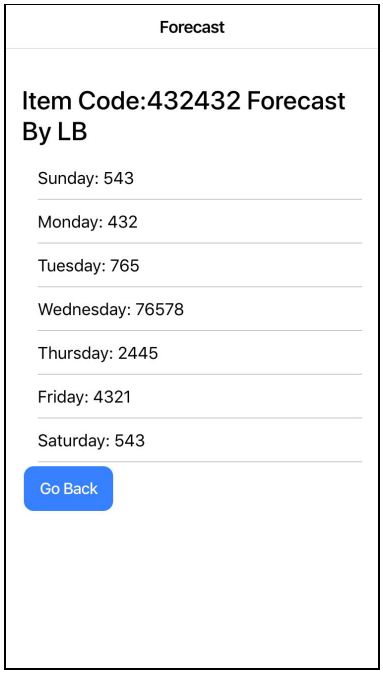

Figure 4.5

4.2 - Admin Web App

Login Page

The login screen can be seen in figure 4.6. It allows the user to authenticate themselves and login to the admin web app. It will use the *POST /login/* endpoint of the API. It can only navigate forward to the user management page.

User Management Page

The user management screen can be seen in figure 4.7 It allows the user to create, edit, and delete current users from their business. It will use the *GET /users/* endpoint of the API. It can navigate back to the login screen if the user logs out, or forward to the create or edit user modal.

Create User Modal

The create user modal can be seen in figure 4.8. It allows the user to create

new users for their business. It will use the *POST /users/* endpoint of the API. It can only navigate back to the user management page.

Edit User Modal

The edit user modal can be seen in figure 4.9. It allows the user to edit existing users from their business. It will use the *PUT /users/<user_id>/* endpoint of the API. It can only navigate back to the user management page.

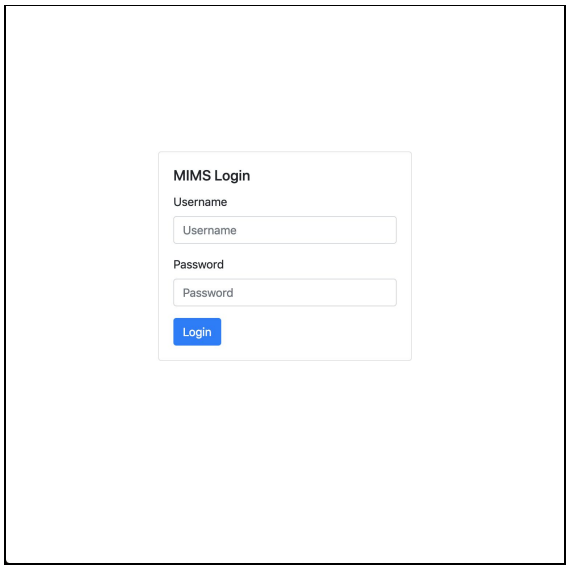

Figure 4.6

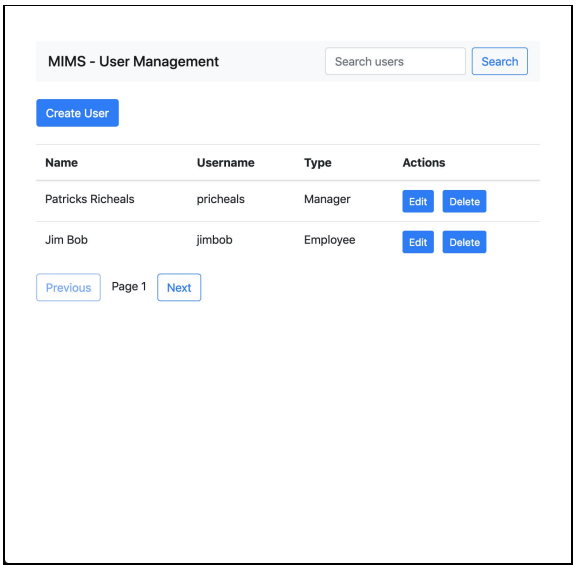

Figure 4.7

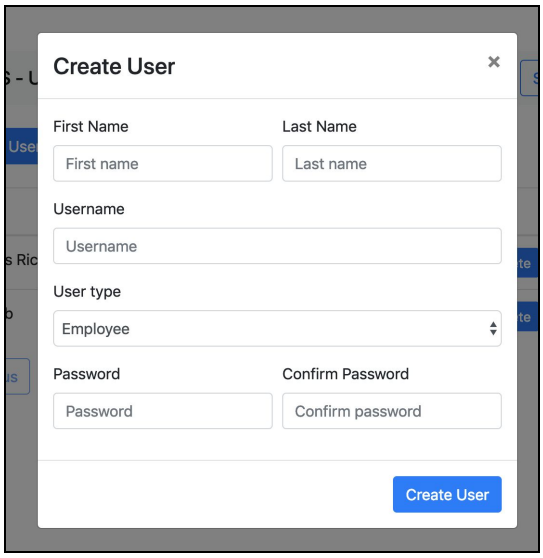

Figure 4.8

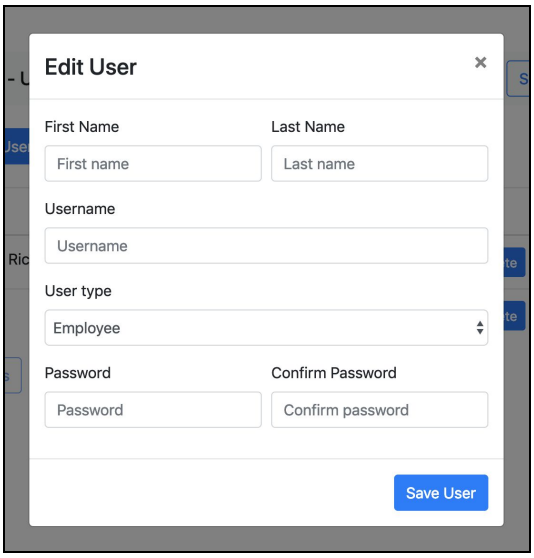

Figure 4.9

5. Technology Stack

5.1 - Frontend Mobile Application

The frontend mobile application will be developed using the hybrid mobile application framework, Ionic 4. Ionic is a hybrid framework, meaning the actual application is programmed in web languages, namely TypeScript, HTML, and CSS. This allows the application to be easily packaged to run on the web, iOS, or Android. Programming in these web languages allows for fast frontend development and high portability of a truly singular codebase. Other frameworks we considered were Flutter and React Native. Ultimately, Ionic was chosen over these alternatives due to its maturity and active community for help and open source modules that can both aid in development.

5.2 - Frontend Admin Web Application

The frontend admin web application will be developed using Angular 7 and Bootstrap 4. Ionic, the framework that will be used to develop the frontend mobile application, is actually built on top of Angular. Building our admin web application on top of Angular will allow us to really get familiar with Angular and improve our development speed. Angular itself provides a system to organize and control components within a web application, but does not provide any visual elements. To fill this need, Bootstrap will be used to provide a clean user interface.

5.3 - Backend API

The backend API will be implemented as a RESTful API using Python Flask. Flask was chosen due to its simplicity and ease of use for our purposes. It will allow us to quickly implement API endpoints and provide no clutter or features that we don't need. For interfacing with the database, SQLAlchemy will be used to easily connect to different databases and tables and provide a functional way of constructing queries based on endpoint input data. The API will be interfaced with via a combination of URL query parameters for GET endpoints and JSON body data for POST and PUT endpoints. Java Spring Boot was considered as an alternative to

develop the API, but Python Flask was ultimately chosen due to its simplicity and ease of development.

5.4 - Backend Database

The databases required for this project will be implemented in MySQL. MySQL was ultimately chosen due to the relational nature of our data and the team's familiarity with the database. MySQL was also chosen specifically to implement the sample business database because MySQL is one of the most popular databases used today and could potentially make it easier for our API to integrate with other MySQL databases in the future.

6. Database Design

Two databases will be implemented for this project. The first database, which is described in section 6.1, is the actual database for the MIMS product. The second database, which is described in section 6.2, is a database for a fake business that we are building simply to show off what the MIMS product can actually do.

6.1 - MIMS Database

The MIMS database is the database that stores all relevant information for the MIMS product itself. This includes information on the actual businesses that use the MIMS product and all the users that are created and can login to the different applications under the MIMS system (mobile and web app). The schema diagram for this database can be seen in figure 6.1.

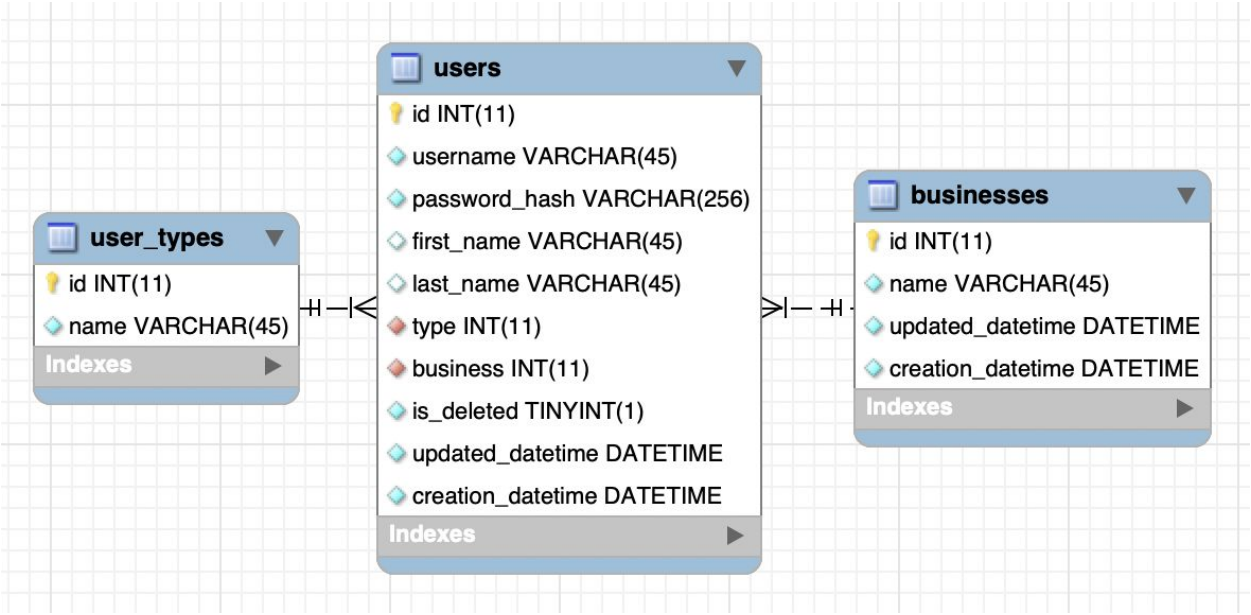

Figure 6.1, database schema diagram for MIMS database

As seen in the diagram above, there are only three tables in the MIMS database. The *users* table stores information on a single user of the MIMS product. A user has a username, a password hash, a name, a type, and the business that the user belongs to. The *type* column is a foreign key to the *user_types* table, which only has two rows, one for "Manager" and one for "Employee." The difference between a

manager and employee is that a manager can login to the admin web app and has access to a few endpoints on the API regarding retrieval and creation of users that an employee doesn't have.

The *businesses* table stores a record for each business that uses the MIMS system. The table exists simply to ensure that we know what business each user works for, so we know what information to serve them from the various endpoints.

5.2 - Business Database

The business database, again, is not apart of the actual MIMS product. It is simply a database that we will design to act as a fake business using the MIMS product. It stores information on products, departments, sales, and inventory transactions. The schema diagram for this database can be seen in figure 6.2.

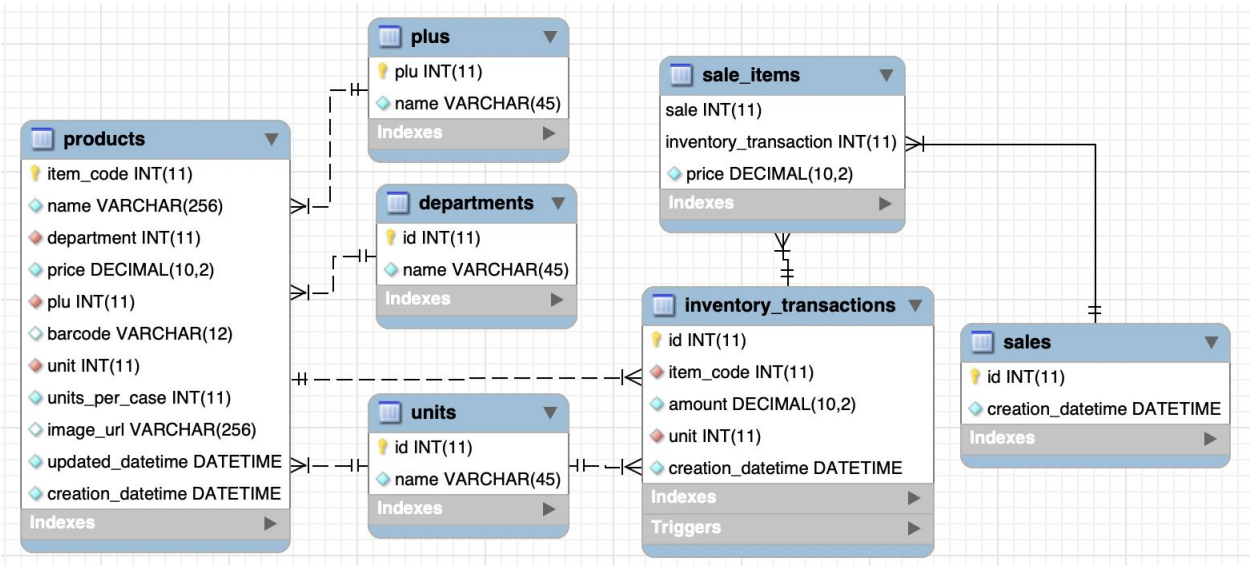

Figure 6.2, database schema diagram for business database

The *products* table stores information on each product in this business's database. This includes information such as the item code, name, what department it's a part of, the current price, the PLU code, the barcode number, the unit that the product is sold in, and the amount of that unit that are in a single case (all items get shipped to the business in cases). The *department* column is a foreign key to the *departments* table, the *plu* column is a foreign key to the *plus* table, and the *unit* column is a foreign key to the *units* table.

The *units* table only has three rows, one for “Unit,” one for “Pound,” and one for “Case.” These units are used by the *products* table to store the unit that the product is sold in. It is also used by the *inventory_transactions* table to specify the unit of the amount of inventory that has been either gained or lossed in an inventory transaction.

The *inventory_transactions* table, then, stores inventory transactions for different products. A negative number in *amount* represents a loss of that amount of *unit* of product while a positive number represents a gain of that *amount* of *unit* of product.

The *sales* and *sales_items* table work in conjunction to store sales that are made in the business. A single sale has many sale items which represent an amount of a specific product that is purchased. Each row in the *sale_items* table references the sale that it is a part of and the inventory transaction for the specific product within the sale, which stores the amount of the product that was purchased. Each sale of an amount of a specific product will generate a loss of however many of that product, so the *sales_items* table just stores this reference to the inventory transaction instead of storing the information twice. The price of the product at time of purchase is also stored on the sale item row as the price can change, and we would want to be able to have a record of the price at time of purchase.

7. API Endpoints

The following endpoints are described in the form of *METHOD /endpoint/url/{input_param}/*. Authorization refers to how the endpoint is protected, input refers to any data that goes into the endpoint, and returns refers to the data that is returned by the endpoint. All input for GET endpoints are taken as query parameters while all input for POST and PUT endpoints are taken as JSON body data. All endpoints return a JSON response.

POST /login/

- Purpose: Validates a username and password for a user and returns an access token for other endpoints on the API
- Authorization: No access token required
- Required input: *username, password*
- Optional input: None
- Returns: An access token upon successful authentication, an error otherwise

GET /users/

- Purpose: Retrieves a list of users
- Authorization: Access token required, must be a manager
- Required input: None
- Optional input: *username, first_name, last_name, search_term, type, page_size, page*
- Returns: A list of users

POST /users/

- Purpose: Create a user
- Authorization: Access token required, must be a manager
- Required input: *username, first_name, last_name, password, type*
- Optional input: None
- Returns: The id of the newly created user, an error otherwise

PUT /users/{user_id}/

- Purpose: Edit a user

- Authorization: Access token required, can be employee or manager. Managers can edit any user of their business, while employees can edit only themselves and require *current_password* to be sent up to update their password. Only managers can edit *type* and *is_deleted*.
- Required input: *user_id*
- Optional input: *username, first_name, last_name, new_password, current_password, type, is_deleted*
- Returns: The id of the updated user, an error otherwise

GET /products/

- Purpose: Retrieves a list of products
- Authorization: Access token required, can be employee or manager
- Required input: None
- Optional input: *name, item_code, plu, barcode, department, search_term, page_size, page*
- Returns: A list of products

GET /products/{item_code}/

- Purpose: Retrieve a specific product
- Authorization: Access token required, can be employee or manager
- Required input: *item_code*
- Optional input: None
- Returns: The requested product

GET /products/{item_code}/movement/

- Purpose: Retrieve a specific product's inventory movement history
- Authorization: Access token required, can be employee or manager
- Required input: *item_code*
- Optional input: None
- Returns: The request product's inventory movement history

GET /products/{item_code}/forecast/

- Purpose: Retrieve a specific product's forecasted future inventory transactions
- Authorization: Access token required, can be employee or manager

- Required input: *item_code*
- Optional input: None
- Returns: The requested product's forecasted future inventory transactions

POST /inventory/

- Purpose: Create an inventory transaction on a business's database
- Authorization: Access token required, can be employee or manager
- Required input: *item_code, amount, unit, datetime (same format for datetime as in figure 7.1)*
- Optional input: None
- Returns: None

PUT /products/{item_code}/

- Purpose: Edit a product (not required for the MIMS product to function, just used by the simulation)
- Authorization: Access token required, can be employee or manager
- Required input: *item_code*
- Optional input: *price*
- Returns: The id of the updated product, an error otherwise

POST /sales/

- Purpose: Simulate a sale on our fake business database (not required for the MIMS product to function, just used by the simulation)
- Authorization: Access token required, can be employee or manager
- Required input: A JSON body in the format as specified in figure 7.1
- Optional input: None
- Returns: None

```
{
  "items": [
    {
      "item_code": 7382937,
      "amount": 2,
      "datetime": "2019-01-23 17:59:59"
    },
```

```
    {
      “item_code”: 8293841,
      “amount”: 1,
      “datetime”: “2019-01-23 17:59:59”
    }
  ]
}
```

Figure 7.1, example of JSON body format for POST /sales/

8. Authentication and Security

Authentication of the API and security of user accounts are critical for this project, as the API contains a direct connection to potentially private data for many businesses. Similarly, user account safety is critical as they are what provide safe access to the API and the information that it returns.

8.1 - User Account Safety

User accounts will be secured by storing a hashed password in the database. When a user account is created, their new password will be hashed by an algorithm designed specifically for hashing passwords, specifically the Argon2 password hash implemented by the Python library Passlib. That password hash will be the only thing stored about the user's password, so the original password can not be recovered even by someone with direct access to the database. When we need to authenticate that the password is accurate, we will simply hash the given password in the same manner that the original was and then check equality. This will ensure that user's passwords are safe even in the event of data breach.

8.2 - API Authentication

All endpoints of the API, except for the login endpoint, will require an access token. To obtain an access token, the user must have successfully authenticated their username and password on the login endpoint. The access token that is returned is a JSON Web Token that stores the user's id, so we will always know exactly what user is making the request. This is very important when making requests to any of the product endpoints, as the business of the user making the request is needed to determine what products to actually return. This also ensures that it is impossible for any user to obtain data from a business other than theirs.

9. Goals for Mid Assessment

9.1 - Goals

The team's goals for the mid assessment are to complete all product features as described in the project specification, excluding the future inventory prediction system. The following features that are planned to be completed for the mid assessment include the following.

- Fully featured admin web portal allowing a manger to search, create, and edit users of their business
- Fully featured app allowing a manager or employee to search products, look at product details, adjust product inventory, and look at a product's past inventory movement

9.2 - Team Roles

The team has decided to create smaller teams that will focus on key aspects of each feature set. These teams will still work together to integrate what they are working on with each other, but each team will be responsible for completing the required product functionality for each feature set. These teams and their responsibilities of what to complete by the mid assessment are outlined below.

API Team

- Members: Patrick Richeal
- Responsibility: Complete any remaining API endpoints that are needed by the simulation, app, and admin. Add functionality to api to calculate product inventory amounts at specific points in time.

Simulation and Forecast Team

- Members: Steven Bruman, Dominic Marandino, Brandon Ostasewski
- Responsibility: Complete the simulation program to fill the fake product business with fake sales and inventory transactions. Begin researching methods of predicting future inventory based on previous inventory transactions, and ideally begin to implement an algorithm to accomplish this.

App Team

- Members: Matthew Finnegan, Paul Sigloch
- Responsibility: Complete all remaining pages and functionality on the mobile app as described in the mid assessment goals in section 9.1.

Admin Team

- Members: John Donahue, Sagarika Kumar
- Responsibility: Complete all remaining pages and functionality on the admin web portal.

9.3 - Changes to Product Specification

Due to certain discoveries and design decisions made in the process of creating this document (namely the decision to separate the mims and business databases), certain features described in the project specification no longer make sense in the MIMS product. The first feature under the product features section of the project specification states that the admin web portal will allow managers to add products to their product database. This no longer makes sense, due to the nature of the separation between the MIMS and business database, so that feature is being removed from the project specification.