Inventory Management System

Conor McGrath
Aaron Healy
Advised By: Daniel Cregg
B.Sc.(Hons) in Software Development
Department of Computer Science and Applied Physics
Galway-Mayo Institute of Technology (GMIT)

Contents

About this project

Abstract This paper aims to provide a technical understanding about computerized inventory management systems; the various implementations of, and the technologies used within these systems. Modern businesses can utilize the latest technologies for their warehouses to help:

- analyze trends or patterns through inventory flow
- reduce instances where stock is too low or high, and reduce instances where there is no stock
- increase in accuracy and fulfillment of orders with relation to picking, packing, and shipping a customer's order
- provide a deeper understanding of customer demand for your products

The significance of this study will redound to the benefit of consumers that utilize such systems, especially since businesses are faced with the problem of getting products and services to their costumers quickly and responsively. The demand for better customer experiences, justifies the purpose to evaluate the current technologies already available. Thus, businesses that integrate these technologies and solutions can create stronger customer experiences, through the tight integration of their web platforms and enterprise resource planning systems. For the researcher(s), this study will help advance our knowledge and evaluate the approach with which to take when determining which technologies are best integrated with one another when developing inventory management systems. Thus, a new implementation or integration for these systems may be established.

Authors
Conor McGrath - B.Sc. (Hons) Computing in Software Development
Aaron Healy - B.Sc. (Hons) Computing in Software Development

Chapter 1
Introduction

For the purpose of our final year project and dissertation, we wanted to build a solution that would be practical, scalable, and could easily integrate with most supply chain environments. We researched the uses for inventory management systems in modern day small to medium scale business' and decided that learning to build a solution like this would further build on the skills we had learned throughout the four years of this degree program.

Furthermore, we wanted to satisfy the standards for a software development honours level project, by evaluating the current trend for these systems, and building our own solution based off the knowledge we have gathered and the practical experience we have gained using various tools and technologies during this course.

To begin, the objectives outlined for the project were as such:

- Develop a web application that allows a user to login and based on the details given, that user can access information specifically relating to them and their customers.
- Build a scalable backend system that would be able to manage and maintain the data between the user's products, orders, and their customers.
- Build a user-friendly frontend web application that displays the correct data for the specific user based of their business' interactions, orders, stock and products.
- Secure a users' information using authentication and authorization protocols like JSON Web Tokens (JWT)
- Display generated table information based of a user's orders, customers, and their products, and use this data to build graphical charts so a user can better visualize their business.
- Gain experience in building scalable web applications and working collaboratively as a team of developers.
- Learn about new or upcoming trends or tools that might fit the solution we are developing or assess their potential in integrating such technologies with our solution.

Inventory management systems are a key part to any business. An effective inventory management system starts with analysis and design. The more thorough the analysis and design, the fewer problems you'll have running and managing a new inventory system. They allow the business to manage and control their customers, wholesalers, orders and stock. They are essential in giving up-to-date data on movement of stock throughout the company. We wanted to address some common problems that arise when using inventory systems and try to deliver on a possible solution for these. We looked at optimization in collecting and processing the data our application collects and how we can reuse this data for the benefit of the user.

The members of this group have taken it upon themselves to research and analyze existing Inventory Management Systems available, and with the knowledge gained create an application that will allow users to manage all their customers, wholesalers, orders and stock in an easy and conventional way.

One of the core concepts we wanted to focus on was to be able to develop a solution by following a process for completing projects in sections and iterations, and in addition, how we would be able to separate out the project's tasks, giving us further perspective for things like:

- Project planning
- Analysis
- Design
- Implementation
- Testing

We undertook the work as and separated out the tasks into individual prints cycles to mimic the process of working in an agile environment. This was best put to practice in the research and planning, designing, and implementing the solution throughout each increment. Agile was effective in keeping the progress for our project on track and provided us with quick resolutions to problems we encountered at any given iteration.

During each iterative cycle we managed to build components for each part of our application using version control software and hosting service to host these versions of our software online. We chose Git and Github for this task as it was the version control software that we were most familiar with and most comfortable in using.

We decided on a 3-tier architecture for our solution as it more suitable to modularizing the user interface, the business logic, and the data storage layer. The architecture for applications like these suit production environments as well as client-server communication systems. It would also be beneficial in decreasing development cycles as we could separately test our business logic before integrating it into our frontend application in our presentation layer. Furthermore, we would be free to work on our business logic without having to worry about any knock-on effect to the other parts of the application.

Moving onto reviewing the technologies we have used, we have listed and described the main functionalities of various tools, and languages we have used throughout the development process. We discuss our rationale for deciding on certain technologies and give a brief overview about each.

After this we discuss the overall architecture and the design of our application, discussing each feature in depth, such as the authentication and authorization for our users. We explain along with the use of code snippets, how the application works and communicates with the different architectural layers of our application.

We evaluate our design choices in the following chapter and discuss any limitations for our application. Something we both wanted to learn more about was how to test then develop and understanding more about test driven development for building web applications. This wasn't used as thoroughly throughout the course of the project, but we learned about new techniques for applying them in future projects.

Over the course of this project we wanted to learn new techniques for building a web application, introduce ourselves to new technologies and protocols. We wanted to gain better experience developing collaboratively, building on our teamwork skills and whilst focusing on the organizational aspects of a software project.

Chapter 2
Context

The general context of this project is an easy to use system that provides users online sales businesses a platform to manage their inventory. It will help them keep track of all stock and alert them when they are running low on certain products. The users will be able to track their customers and view who purchases what number of products and how regularly. This will allow a business to see what sort of customer their main target for future sales is. The products section of the application will allow users to view and manage the current stock they have. This will show them what products are selling good and the profit they are making from each product. This sort of information is vital for a business to know so that they can see the good and bad sellers and order more inventory accordingly. In terms of the order side of the application, the user will be able to input orders so that they can be tracked. When these orders are completed the products and customer information will all be updated accordingly. Each user will have their own home page which will contain a variety of tables and graphs that represent their current monthly information such as "Total Sales", "Top Selling Products" and "Top Spending Customers".

2.1 Objectives

The main objective of our application is to help businesses to manage their inventory easier. We also wanted to make it easier for businesses to be able to view all the statistics for their sales information easier in one place. The following is a list of the main pages in our application along with the objectives for each page.

1. **Login/Register:** The first page the user will see is a login and/or register page. If the user is new, they will be able to create an account using the register function. If the user has already created an account, they will be able to login using their credentials. Once logged in, the user has access to all the features of the application. All users of the application must be registered so that they can have their inventory data linked to them and only viewable by them.

2. **Home:** The objective of the homepage is that it is a base of navigation for the application while also providing the user with quick available statistics about their inventory. The homepage will display a variety of graphs and tables that will show information regarding the current month of sales information. The homepage will also provide links for the user to follow to the other pages of the application.

3. **Customers:** The customers page will provide a base for the user to view all information about their current customers. This will include information such as the products they have purchased and in what amount. Users will also be able to create, edit and delete customers on this page.

4. **Orders:** The orders page will be the location where the user will be able to manage their current orders, create new orders and look back on previously completed orders. The user will also be able to view orders from specific customers which will allow them to see what each customer is ordering and in what amount.

5. **Products:** The products page will provide a base for the user to view all information about the current products they have to offer. This will include information about the products such as cost price, sale price and the quantity in stock. This is vital information as it will allow the user to see what sort of margin they are making on products and when a re-order of a certain product is needed.

Chapter 3
Methodology

This is where will be discussing the methodologies we are using in our project. We researched the different methodologies we could use in our project. After seeing the positives and negatives of each methodology, we decided to use Agile as our main methodology. We felt this was a methodology that would suit our development and it is also a methodology that is used widely in organizations around the world

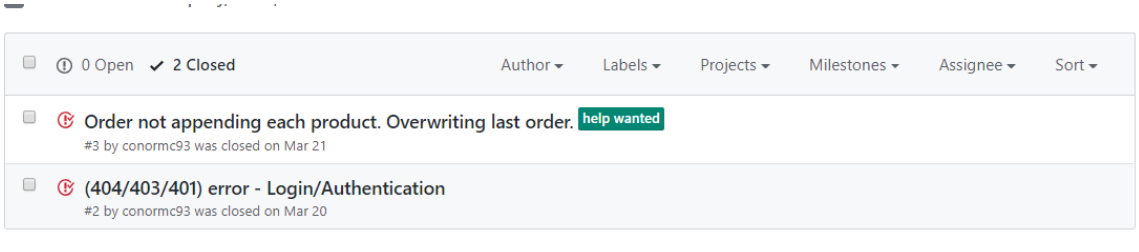
3.1 Agile Development

Our project contained three main stages, research, design and implementation. For each of these stages we applied an Agile like approach to complete them. Agile was suited to this project as it allows for flexibility and for us to deliver software incrementally. This is what made it stand out to us as a methodology that would work well for us. We used a Scrum like approach for our research, design and development process. Scrum is an agile framework for teams who members break their work into actions that can be completed within timed iterations, called sprints, usually around two weeks long but no longer than one month, they then track progress and re-plan in short meetings.

Our sprints involved certain parts of the development being completed. Our first sprint in the development phase was to create the database that would be used for storing our data. After this our sprints followed a pattern of developing a backend and frontend component for each part of the project.

This would mean one sprint would involve developing the users, another, the customers, the next, the products and so on.

We held weekly meetings with our project supervisor, this allowed us to have meetings before, after and during our sprints. This constant contact with our supervisor was key to making sure we were completing everything on time and to make sure the project was running smoothly. After each of these meetings we were able to note what was needed to be completed before the next one. This gave us a goal to work towards each week in our sprints. Any issues we noted at a meeting were logged into our Github repository under the issues tab. This allowed us to track problems we had throughout the problem. Below is an example of two of the issues we had at one stage of the project.



3.2 Version Control

Version control is key in any project development cycle. It is important as it always allows members of the team to work on the project at the same time and keep a working version of the project in a safe place. We decided to use Git and Github for our version control.

3.2.1 Git

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers and can be used to track changes in any set of files. Git's best features include its speed, data integrity and support for distributed, non-linear workflows.

As with most other distributed version-control systems, and unlike most client–server systems, every Git directory on every computer is a full-fledged repository with complete history and full version-tracking abilities, independent of network access or a central server. Some of the main features of Git are:

Strong support for non-linear development
Git supports rapid branching and merging. It includes specific tools for visualizing and navigating a non-linear development history. Git works on the basis that a change will be merged more often than it is written. Branches in Git are usually very lightweight: a branch is only a reference to one commit.

Distributed development
Git gives each developer a local copy of the full development history, and changes are copied from one such repository to another. These changes are imported as added development branches and can be merged in the same way as a locally developed branch.

Efficient handling of large projects
Git is very useful for use with large projects. It is very fast and scalable, and performance tests completed have showed that it was an order of magnitude faster than other version-control systems.

Pluggable merge strategies
As part of its toolkit design, Git has a well-defined model of an incomplete merge, and it has multiple algorithms for completing it, culminating in telling the user that it is unable to complete the merge automatically and that manual editing is needed.

Garbage accumulates until collected
Aborting operations can leave useless dangling objects in the database. Git will automatically perform garbage collection when enough loose objects have been created in the repository.

3.2.2 Github

GitHub is a web-based hosting service for version control using Git. It offers all the distributed version control and source code management functionality of Git as well as adding its own features. It provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project. These extra items were very useful for us developing this project.

Github is one of the most popular version control management services in the world. As of June 2018, GitHub had over 28 million users and 57 million repositories which included 28 million public repositories, making it the largest host of source code in the world.

3.3 Technology Choice

After researching Inventory systems and what type of application we would be designing, we made our decisions about the languages and software we would be using for our project. We decided on a MySQL database hosted on AWS for the database side of our project, Java Spring Boot for our backend and Angular 6 for our frontend. These technology choices were made based on our experience using each and the advantages they gave us over the other options. Each of these technologies will be explained in the Technology Review chapter.

Backend **Frontend**

Chapter 4
Technology Review

In this section, we will review all the different technologies used in our project. We will discuss what each technology is, and what it does in our project. We will use images and code snippets to show how the technology is implemented into our project. We will split the review into three smaller sections which will be the Database, Backend and Frontend sections. We will talk about the technologies used in each section and why we chose to use each for that section.

4.1 Database

4.1.1 MySQL

MySQL is a free-to-use, open-source database that facilitates effective management of databases by connecting them to the software. It is a stable, reliable and powerful solution. The choice of using MySQL as our database for our project was easy to make. Both members of the group have used MySQL for several modules over the course of our degree. We also used MySQL in our 3rd year project. This experience gave us the confidence to use it in our project. We also felt MySQL would suit our project for several reasons, which we mention below.

MySQL is globally renowned for being the most secure and reliable database management system and is used in popular web applications including WordPress, Facebook and Twitter. This data security is essential in our project as we need to protect our user's data. We will be storing information about the user's business such as orders and customers. This is important data that must be stored securely.

Although our project will only have a small number of users, MySQL allows it to be expanded to a huge size without putting any strain on the database. It is designed to meet even the most demanding applications while ensuring optimum speed, full-text indexes and unique memory caches for enhanced performance. MySQL also comes with the assurance of 24×7 uptime which makes sure our project will always be accessible.

4.2 Backend

When choosing what technologies, we were going to use for our backend we had to take a few things into consideration. We had to make sure that we would be able to create a secure backend capable of connecting to our database and frontend. After researching a few different solutions, we decided to go with the technologies mentioned below.

4.2.1 Java

Java is a general-purpose computer-programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. We have worked with Java in every year of our degree and we both feel like it is our strongest language. We felt our skills in the language would give us an opportunity to make a high standard backend. Having used Java as a backend before we both had experience in connecting it to databases and frontends. This knowledge would benefit us greatly when it comes to connecting the separate parts of our project.

After we decided on using Java for the backend, we researched a few ways we could design the backend. We looked at previous work we had done and decided to use the Spring framework. This is discussed in the next section.

4.2.2 Spring

The Spring Framework is an application framework and inversion of control container for the Java platform. Spring makes use of Inversion of Control and Dependency Injection to promote good software coding practices and speed up development time. We used Spring Boot which is a part of the Spring framework. Spring Boot is simply defined as a set of pre-configured frameworks and technologies which help to reduce boiler plate configuration. This provided us with a shorter way of getting a basic Spring web application up and running in the quickest amount of time. We used this as it allowed us to focus on other aspects of the project without being bogged down on setting up the configuration of the web application.

4.3 Frontend

When choosing what technologies, we were going to use for our frontend we had to take a few things into consideration. We had to make sure that we would be able to create a frontend that could connect to our backend where all the requests would be made. After researching a few different solutions, we narrowed it down to two choices. These were React and Angular. After a more in depth look at both, we decided to choose Angular for our frontend.

4.3.1 Angular

Angular is a TypeScript-based open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. Angular is a complete rewrite from the same team that built AngularJS. Angular allows us to build progressive web applications that use modern web platform capabilities to deliver app-like experiences. They allow our applications to be high performance and to run offline. Building native mobile applications is also possible through Angular with strategies from Cordova, Ionic, or NativeScript. Desktop applications can be created across Mac, Windows, and Linux using the same Angular methods that would be used for web applications.

Speed and performance are key for Angular. Angular can turn your templates into code that's highly optimized for today's JavaScript virtual machines. This gives you all the benefits of hand-written code with the productivity of a framework. Angular code can be rendered nearly instant once the first view of the application is served on Node.js®, .NET, PHP, and other servers. This also paves the way for sites that optimize for SEO.

Angular apps also load very quickly thanks to the new Component Router. This delivers automatic code-splitting, so users only load code required to render the view they request.

Angular also allows for great productivity. Templates can be used to quickly create UI views with simple and powerful template syntax. Angular command line tools allow us to start building fast, add components and tests, then instantly deploy. All popular IDEs and editors have intelligent code completion for Angular, provide instant errors and other feedback for Angular.

All these benefits mentioned above are the reasons we chose Angular to work with. It is in our opinion the best and easiest to pick-up front-end framework.

4.3.2 Typescript

TypeScript is an open-source programming language developed and maintained by Microsoft. It is a superset of JavaScript and adds optional static typing to the language. TypeScript is designed for development of large applications. As TypeScript is a superset of JavaScript, existing JavaScript programs are also valid TypeScript programs. TypeScript may be used to develop JavaScript applications for both client-side and server-side execution.

TypeScript supports definition files that can contain type information of existing JavaScript libraries. This enables other programs to use the values defined in the files as if they were statically typed TypeScript entities. There are third-party header files for popular libraries such as jQuery, MongoDB, and D3.js.Node.js programs can also be developed within TypeScript due to TypeScript headers for the Node.js basic modules being available, allowing development of Node.js programs within TypeScript.

Since TypeScript starts from the same syntax and semantics that all JavaScript developers know today, it is possible to use existing JavaScript code, incorporate popular JavaScript libraries, and call TypeScript code from JavaScript. TypeScript compiles to clean, simple JavaScript code which runs on any browser, in Node.js, or in any JavaScript engine that supports at least ECMAScript 3.

TypeScript has tools which enable large application development. Types enable JavaScript developers to use highly-productive development tools and practices. These include like static checking and code refactoring when developing JavaScript applications.

TypeScript is essentially state of the art JavaScript. It offers support for the latest and evolving JavaScript features. These features are available at development time for high-confidence app development but are compiled into simple JavaScript that targets ECMAScript 3 environments.

4.3.3 HTML

Hypertext Markup Language is the standard markup language for creating web pages and web applications. Combining HTML with Cascading Style Sheets and JavaScript, you get the triad of cornerstone technologies for the World Wide Web. Web browsers can receive HTML documents from a web server or local storage and render them into multimedia web pages. HTML documents are documents which describe the structure of a web page semantically by use of tags, with these tags representing different actions when the page is rendered.

HTML elements are the building blocks of HTML pages. They are what are used to describe the action taken when the page is rendered. HTML constructs, images and other objects such as interactive forms may be embedded into the rendered page. HTML provides a way to create structured documents by using tags to define structural semantics for text such as

headings, paragraphs, lists, links, quotes and other items. These tags are written using angle brackets. Tags such as <img /> and <input /> directly bring content into the page. The <img/> tag allows you to give the path to an image which will then be rendered into the web page. Other tags such as <h1> surround and provide information about document text and may include other tags as sub-elements. The <h1> tag shown represents a header. This will cause the text inside the tags to be displayed in a larger size than the text in the rest of the page. Web Browsers do not display the HTML tags, they just use them to interpret the content of the page.

HTML can also use programs written in a scripting language such as JavaScript or TypeScript, which affects the behavior and content of web pages. Cascading Style Sheets define the look and layout of content. HTML is used in our project through the Angular frontend. Angular uses HTML to build the web pages and the content they display. We use the TypeScript in Angular to control certain aspects of the HTML pages.

4.3.4 CSS

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language like HTML.As mentioned above CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript. HTML was never intended to contain tags for formatting a web page. HTML was created to describe the content of a web page, such as

<h1>This is a heading</h1>
This is a paragraph

Then tags like <font>, and color attributes were added to the HTML 3.2 specification, it became a nightmare for web developers. Large website development where fonts and color information were added to every single page, became a long and expensive process. Therefore, CSS was designed to allow for the separation of presentation and content, including layout, colors, and fonts. This separation was essential as it improved content accessibility, provided more flexibility and control in the specification of presentation characteristics and enabled multiple web pages to share formatting by specifying the relevant CSS in a separate .css file.

The separation of formatting and content also made it feasible to present the same markup page in different styles for different rendering methods, such as on-screen, in print or by voice. CSS also has rules for alternate formatting if the content is accessed on a mobile device. While CSS is mainly used with HTML, it is also supported by other markup languages including XHTML, plain XML, SVG, and XUL.

We used CSS in our project inside our Angular frontend to style our HTML elements. It allowed us to create items at certain areas on the page and style them in a way we wanted.

The cornerstone technology of the World Wide Web

4.4 Software Used

While developing this project we had to use software such as IDEs to write the code for our project. Below we will discuss the software we used while developing our project and for what did we use it for.

4.4.1 Eclipse IDE

Eclipse is an integrated development environment and is the most widely used Java IDE. Eclipse is written mostly in Java and its primary use is for developing Java applications, but it may also be used to develop applications in other programming languages via plug-ins.

The Eclipse software development kit, which includes the Java development tools, is meant for Java developers. It is possible to extend its abilities by installing plug-ins written for the Eclipse Platform, such as development toolkits for other programming languages.

We used Eclipse in our development for our backend Spring Boot development. We have used Eclipse for all Java development we have done in the past and this was our reason for using it for this project.

4.4.2 Visual Studio Code

Visual Studio Code is a source-code editor developed by Microsoft for Windows, Linux and macOS. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring. It is also customizable, so users can change the editor's theme, keyboard shortcuts, and preferences.

Visual Studio Code is based on Electron, a framework which is used to deploy Node.js applications for the desktop running on the Blink layout engine. Although it uses the Electron framework, the software does not use Atom and instead employs the same editor component used in Azure DevOps.

We used Visual Studio Code to develop the Angular frontend of our project. We used the TypeScript plugin for Visual Studio Code to give us the full tools available to us while developing. We both had past experiences using Visual Studio Code, therefore we decided it was the best code editor for us to use.

4.4.2 WAMP

Windows, Apache, MySQL, and PHP (WAMP) is a variation of LAMP for Windows systems and is often installed as a software bundle (Apache, MySQL, and PHP). It is often used for web development and internal testing; however, it can also be used to serve live websites.

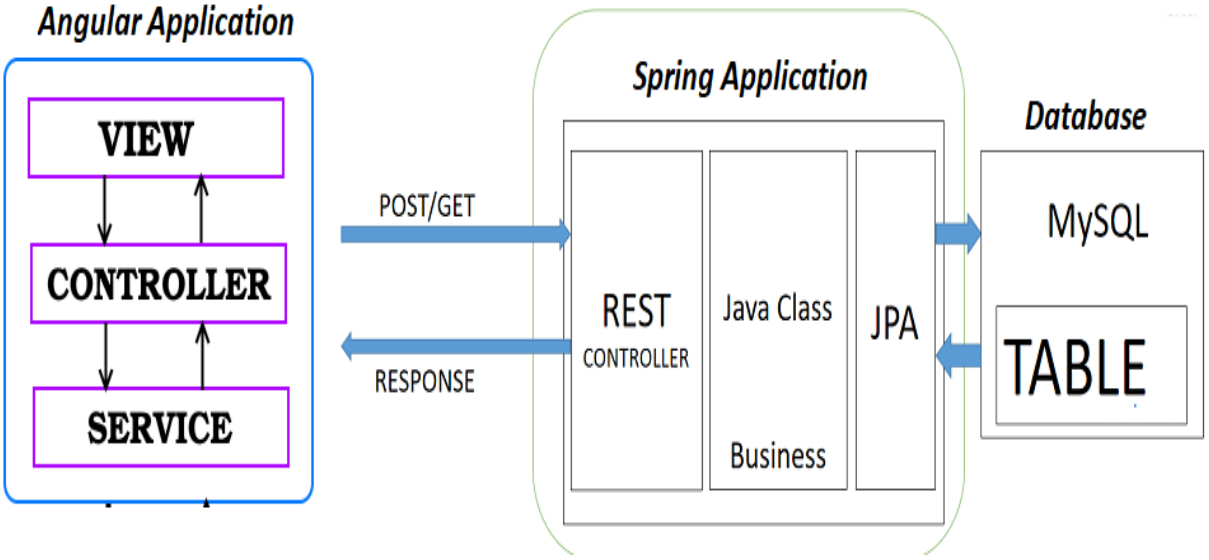
WAMP also includes MySQL and PHP. These are two of the most common technologies used in creating dynamic websites. MySQL is a high-speed database, which we use in our project, while PHP is a scripting language that can be used to access data from the database. By installing these two components locally, a developer can build and test a dynamic website before publishing it to a public web server.

We used WAMP to develop and control our MySQL database. Our database stores all information from our web application. Using WAMP it was possible for us to monitor the database and see when items were being added/deleted from it. This was useful in development as it allowed us to see when requests were being made correctly to and from the database.

Chapter 5

System Design

In this chapter, we will discuss the overall design and architecture of our Inventory Management application. We will look at code snippets and use visual aids to explain clearly the full design of the project and how it works. We have broken this chapter up into three different layers which we will discuss in depth. We will look at how they all connect to each other to make up the final application. The first of our three layers is our database layer which is where we will be describing how our MySQL database was created and how it is being hosted. We will then look at our backend layer which will be where we discuss our Java Spring Boot application and how it acts as a server for our application. We will talk about it is connected to both our database and the frontend layer. This frontend layer is what we will finish this chapter discussing. This frontend layer is our angular application which is the layer of the project that the users will be viewing and interacting with. The figure below shows the overview of our architecture.



5.1 Database Layer

The database layer is where we have our MySQL database setup to store all our application data. The database layer of the project is important as it stores important information for the application like user's information and login details such as passwords.

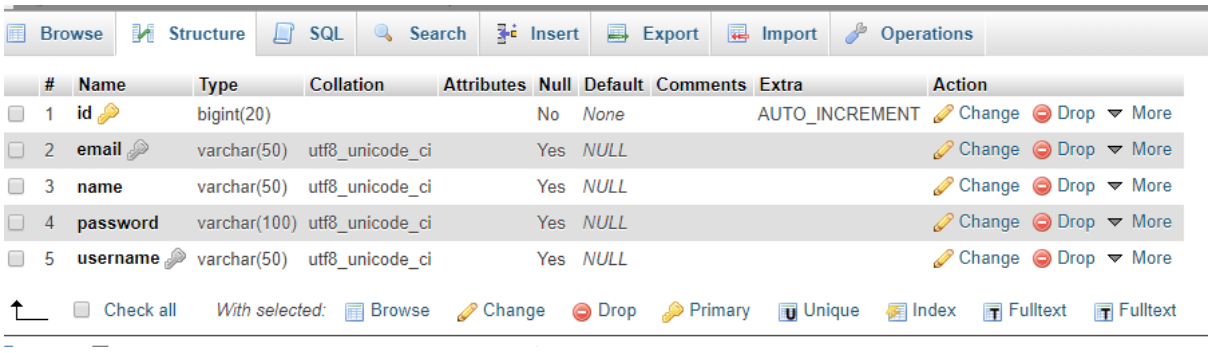
5.1.1 MySQL Database

In our research stage of the project we looked at various database technologies that we felt would be possible to use in our design. After discussing the different databases available we narrowed it down to MySQL and MongoDB. We had experience using both technologies in the past and were confident we would be able to work with either. We looked at possibly incorporating both in the project but in the end decided it would be better to focus on making sure we had one of them working correctly with the rest of our system. We decided MySQL would be the one we would we continue with.

We used W.A.M.P to develop our database. We didn't need to create any tables inside our database manually as this was being done for us automatically through the Spring Boot application. We are using Hibernate to create the tables from our entity classes. An example of one our entity classes is shown below.

```
@Entity
@Table(name = "users", uniqueConstraints = {
        @UniqueConstraint(columnNames = {
            "username"
        }),
        @UniqueConstraint(columnNames = {
            "email"
        })
})
public class User{
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank
    @Size(min=3, max = 50)
    private String name;

    @NotBlank
    @Size(min=3, max = 50)
    private String username;

    @NaturalId
    @NotBlank
    @Size(max = 50)
    @Email
    private String email;

    @NotBlank
    @Size(min=6, max = 100)
    private String password;
```

As you can see from the code snippet of the user class, our users have an ID, a username, a name, an email and a password. Using JPA annotations such as “Entity” we can then use Hibernate to create this table automatically in our database once the application is run. The image below shows what this table looks like in our MySQL database.



This shows that when the application was running, the table was created with all the rows that were contained in the class. We used Hibernate as it is database independent. It allowed us to work with any database we wanted. We already mentioned why we chose MySQL. Using hibernate we didn’t have to worry about writing MySQL queries and making sure we had the syntax correct. It also allowed us to make changes in the class and not have to go and write a new query again. Once the program was running, the database would be updated. Using Hibernate also meant mapping a database table with a java object called "Entity". This meant once we had these mapped, we got the advantage of object orientated programming concepts like inheritance and encapsulation.

We hosted on our MySQL online using a free hosting service. This service allowed us to use their server which is constantly running to make sure our database is always available. The user of the application will not know anything about the database and how the data is being stored. They will just run the Spring Boot and Angular parts of the project and get a working version of the application appear in their browser. A full architecture of our MySQL database is shown below.

5.2 Backend Layer

The backend layer of our system design is where all the logic of our application is contained. This is where all the main work is done in terms of reading and writing from the database and keeping everything up to date. It is the layer in which all communication from the database to the frontend of the application takes place. Everything flows through this section. As we did with the database layer, we researched possible backends we could use for our project. After looking at a few of the options we chose Java application using the Spring Boot framework. We chose these as we both feel our strongest language is Java and we wanted to improve our knowledge and skill in the language even further. We also had used Spring previously so felt it would be an ideal framework to use to help get our application up and running quickly.

5.2.1 Spring Boot Java Application

In the first steps of designing our backend we had to think about how we were going to connect to our database and how we would make calls to and from the database. To connect to our database, we declared its path In the "application.properties" file using the "spring.datasource.url" property. In this file we also declared the username and password for the database using the "spring.datasource.username" and "spring.datasource.password" properties. This allowed us to link our backend directly to our database being hosted on the online server. An image of this file is shown below.

```
spring.datasource.url=jdbc:mysql://remotemysql.com:3306/skUp97SeNY?useSSL=false
spring.datasource.username=skUp97SeNY
spring.datasource.password=AxcL6hhl6V
spring.jpa.generate-ddl=true

# App Properties
inventory.app.jwtSecret=jwtInventorySecretKey
inventory.app.jwtExpiration=86400
```

To make calls to and from the database we used a "CrudRepository" which is a Spring Data Interface. This allowed us to create, read, update and delete from the database. We added the "CrudRepository" ability to our application by creating interfaces for each table and extending them from CrudRepository as shown below.

```
import org.springframework.data.repository.CrudRepository;

public interface CustomerRepository extends CrudRepository<Customer, Long>{
    List<Customer> findByCname(String cname);
    Optional<Customer> findByCid(long cid);
    Integer deleteByCid(long cid);
    List<Customer> findByUid(long uid);

}
```

Spring provides “CrudRepository” implementation class automatically at runtime. It contains methods such as save, findById, delete, count and so on. We wanted to add extra methods for some of the tables, so we had to declare them in the interface that matched that table.

Our next step when we were developing the backend was to create each class for every table we wanted in the database. We started with our user class. The important part in the user class was making sure that each user had a unique email and username. This was key as otherwise the login system would not be secure as multiple accounts could have the same username and their data would overlap. We did this by declaring them as unique constraints as shown here.

```
@Entity
@Table(name = "users", uniqueConstraints = {
        @UniqueConstraint(columnNames = {
            "username"
        }),
        @UniqueConstraint(columnNames = {
            "email"
        })
})
```

This assured us that all users of our application would be unique, and all login details would be different. The rest of the user class was just generating getters and setters for each of the variables associated with the user. Some of these are shown below.

```
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getUsername() {
    return username;
}

public void setUsername(String username) {
    this.username = username;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}
```

After this we looked at how we could securely sign people in and out of our application and storing their password securely. We used Json Web Tokens to do this. A JSON Web Token (JWT) is a JSON object that can be defined as a safe way to represent a set of information between two parties. In our case this was the user's password and how we represented it in our database. The token is composed of a header, a payload, and a signature. Essentially it is a string with the following format "header.payload.signature". This is how we went about implementing JWT into our project.

When a HTTP request comes through to Spring, it will go through a chain of filters for authentication and authorization purposes. What we did was add our "JwtAuthTokenFilter" class to the chain of filters.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    ...

    http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
}
```

The "JwtAuthTokenFilter" then validates the Token using our "JwtProvider":

```
public class JwtAuthTokenFilter extends OncePerRequestFilter {

    @Autowired
    private JwtProvider tokenProvider;

    @Override
    protected void doFilterInternal(HttpServletRequest request, HttpSe
            throws ServletException, IOException {
        try {
            String jwt = getJwt(request);
            if (jwt != null && tokenProvider.validateJwtToken(jwt)) {
                ....
            }
```

We then use our "JwtAuthTokenFilter" to extract the username and password from the received token using the "JwtProvider". Based on the extracted data, "JwtAuthTokenFilter" then creates an AuthenticationToken which implements Authentication.

```
if (jwt != null && tokenProvider.validateJwtToken(jwt)) {
    //Extarct User Details
    String username = tokenProvider.getUserNameFromJwtToken(jwt);
    UserDetails userDetails = userDetailsService.loadUserByUsername(username);

    //create AuthenticationToken
    UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(
            userDetails, null, userDetails.getAuthorities());
    authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));
```

It then uses the AuthenticationToken as an Authentication object and stores it in the SecurityContext for future filter uses such as Authorization filters.

```
SecurityContextHolder.getContext().setAuthentication(authentication);
```

SecurityContextHolder is the most fundamental object where we store details of the present security context of the application. Spring Security uses an Authentication object to represent this information. It is the possible for us to query this Authentication object from anywhere in our application.

The next step in this process was to delegate the AuthenticationToken for the AuthenticationManager. The AuthenticationToken object that was created will be used as an input parameter for the authenticate() method in the AuthenticationManager which is in our AuthRestAPIs class. We used a DaoAuthenticationProvider which is provided with the Spring framework. We chose this as DaoAuthenticationProvider works well with form-based logins which submit a simple username and password authentication request. This is what we were having the user submit so it suited us to use this provider. It authenticates the User simply by comparing the password submitted in a UsernamePasswordAuthenticationToken against the one loaded by the UserDetailsService as a DAO.

```
public ResponseEntity<?> authenticateUser(@Valid @RequestBody LoginForm loginRequest) {

    Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(loginRequest.getUsername(), loginRequest.getPassword()));
```

Configuring this provider was simple for us with the use of AuthenticationManagerBuilder as shown below.

```
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {
    @Autowired
    UserDetailsServiceImpl userDetailsService;

    @Override
    public void configure(AuthenticationManagerBuilder authenticationManagerBuilder)
        authenticationManagerBuilder
                .userDetailsService(userDetailsService)
                .passwordEncoder(passwordEncoder());
    }

    @Bean
    @Override
    public AuthenticationManager authenticationManagerBean() throws Exception {
        return super.authenticationManagerBean();
    }
```

For retrieving our user details, we used our UserDetailsService class. We done this by obtaining a principal from the Authentication object. This principal is then casted into a UserDetails object to lookup the username, password and GrantedAuthoritys.

```
UserDetails userDetails = (UserDetails) authentication.getPrincipal();
```

It is easy for us to implement UserDetailsService and easy for us to retrieve authentication information using a persistence strategy

```
@Service
public class UserDetailsServiceImpl implements UserDetailsService {

    @Autowired
    UserRepository userRepository;

    @Override
    @Transactional
    public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {

        User user = userRepository.findByUsername(username).orElseThrow(
                () -> new UsernameNotFoundException("User Not Found with -> username or email : " + username));

        return UserPrinciple.build(user);
    }
}
```

We also had to protect our resources with HTTPSecurity & Method Security expressions. To help Spring Security know when we want to require all users to be authenticated, which Exception Handler to be chosen, which filter and when we want it to work, we implement WebSecurityConfigurerAdapter and provide a configuration in the configure method in the WebSecurityConfig class.

```
@Override
protected void configure(HttpSecurity http) throws Exception {
    http.cors().and().csrf().disable().
            authorizeRequests()
            .antMatchers("/**").permitAll()
            .anyRequest().authenticated()
            .and()
            .exceptionHandling().authenticationEntryPoint(unauthorizedHandler).and()
            .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);
}
```

Spring Security also provides some annotations for pre and post-invocation authorization checks, filtering of submitted collection arguments or return values. We enabled Method Security Expressions by using the @EnableGlobalMethodSecurity annotation.

Once we had all the JWT login and authentication complete it was time to focus on creating the models, controllers and interfaces for the items each user would have. We decided that each user would have customers, orders and products. We chose these three as we felt it was the basic things that would be involved with a user using the inventory management system.

Each user will have the products they provide, they will have the customers they sell these products too and they will have the orders which are a combination of the customers and products they buy. We started off creating the products for each user. We had to create a model class called Product. This would contain the Entity annotation which would allow the products table to be created in the database using the variables we declared.

```
@Entity
@Table(name = "products")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long pid;

    @Column(name = "pname")
    private String pname;

    @Column(name = "stock")
    private int stock;

    @Column(name = "cost_price")
    private float cost_price;

    @Column(name = "sale_price")
    private float sale_price;

    @Column(name = "uid")
    private long uid;
```

We gave each a product an ID, a name, a stock amount, a cost and sale price and a userID which links the, to their user. When the user is creating a product, they will be able to set all these values except for the ID which is auto incremented. We also created the getters and setters in this class. These will allow the user to set the values for the product and call them at a later point.

We then looked at the ProductRepository which is our interface that extends the CrudRepository. We had to add a few more methods that are not included in the CrudRepository off the bat. These methods will be explained when discussing the ProductController class next.

```
public interface ProductRepository extends CrudRepository<
    List <Product> findByPname(String pname);
    Optional <Product> findByPid(long pid);
    Integer deleteByPid(long uid);
    List<Product> findByUid(long uid);
}
```

Our ProductController class is where all our request methods are declared. These are the methods that are called from the frontend and return the data to the frontend while updating the database also. In our ProductController our first two methods are for getting all the products for the current logged in user and creating a product. The mapping annotation before the method declares the path that must be called form the frontend to reach this method. In both these methods we take in the username of the current logged in user. This is used to retrieve only their products from the database and to create a product that includes their userID. This is done to avoid any cross over of products between users.

```
@GetMapping("/products/{username}")
public List<Product> getProducts(@PathVariable("username")String username){

    List<Product> products = new ArrayList<>();
    repository.findByUid(userRepo.findAllByUsername(username).getId()).forEach(products::add);

    return products;
}

@PostMapping(value = "/products/{username}/create")
public Product postProduct(@RequestBody Product product, @PathVariable("username")String username) {

    product = repository.save(new Product(product.getPid(), product.getPname(), product.getStock(),
            product.getCost_price(), product.getSale_price(), userRepo.findAllByUsername(username).getId()));
    return product;
}
```

We can see in the getProducts method that the ProductRepository is called to find products that have a userID the same as the currently logged in user. These are then added to a list and returned. In the postProduct method we can see we use the save method in the repository to create a new product in the database. The rest of the methods in the ProductController are like these two. They all have a path that is called from the frontend and all either return a product, products or just update the database.

The Customer model, CustomerController and CustomerRepository are all very similar in their setup. The Customer model declares the variables that the customer table will have.

```
@Entity
@Table(name = "customer")
public class Customer {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long cid;

    @Column(name = "cname")
    private String cname;

    @Column(name = "address")
    private String address;

    @Column(name = "phone")
    private String phone;

    @Column(name = "uid")
    private long uid;

    @Column(name = "amount_purchased")
    private float amount_purchased;
```

Each customer will have an ID, a name, an address, a phone number, a UserID to link the logged in user and an AmountPurchased to keep track of the total amount that this customer has purchased off the user. In this class we also had a Comparator method which was used to compare customers. This was used in the controller when we wanted to order the customers based on the amount the had purchased.

```
public static Comparator<Customer> apComparator = new Comparator<Customer>() {
    @Override
    public int compare(Customer x, Customer y) {
      return (y.getAmount_purchased() < x.getAmount_purchased() ? -1 :
              (y.getAmount_purchased() == x.getAmount_purchased() ? 0 : 1));
    }
  };
```

In the CustomerRepository we had to add some of our own methods just like we did in ProductRepository. Just like ProductRepository, these methods will be used in the CustomerController class to help the request methods complete certain tasks.

```
public interface CustomerRepository extends CrudRepository<
    List<Customer> findByCname(String cname);
    Optional<Customer> findByCid(long cid);
    Integer deleteByCid(long cid);
    List<Customer> findByUid(long uid);

}
```

In CustomerController we declare all the request methods that will be used by the frontend to return details about the customers or to change data about the customers in our database.

```
@GetMapping("/customers/{username}")
public List<Customer> getCustomers(@PathVariable("username")String username){

    List<Customer> customers = new ArrayList<>();
    repository.findByUid(userRepo.findAllByUsername(username).getId()).forEach(customers::add);

    return customers;
}

@GetMapping("/customers/top/{username}")
public List<Customer> getTopCustomers(@PathVariable("username")String username){

    List<Customer> customers = new ArrayList<>();
    repository.findByUid(userRepo.findAllByUsername(username).getId()).forEach(customers::add);

    Collections.sort(customers, Customer.apComparator);
    return customers;
}
```

Here we can see two of the methods in the CustomerController. The first is used to return all the customers from the database that have the same userID as the user that is currently logged in. The second method is where we use the comparator method that we had previously mentioned. We use it the Collections import along with it to sort our customers from highest amount purchased to lowest amount purchased and return this list. The rest of the methods in the CustomerController are like what we had in ProductController. They are used to create, update and delete customers from the database.

We then set about creating our Order model, OrderRepository and OrderController. We had originally planned to have an order be able to take many products but unfortunately, we were not able to implement this as we could not get the data storing correctly into the database. Instead we went with the option of having a separate order for each product. This is what our order model variables look like and these are what the orders table in the database contains.

```
@Entity
@Table(name = "orders")
public class Order {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private long oid;

    @Column(name = "cname")
    private String cname;

    @Column(name = "uid")
    private long uid;

    @Column(name = "amount")
    private int amount;

    @Column(name = "pname")
    private String pname;

    @Column(name = "total")
    private float total;

    @Column(name = "profit")
    private float profit;
```

The order model contains an ID, a customer name, a userID, an amount, a product name, a total and a profit variable. The amount is the amount of the product in the order. The total is the total money value of the order and the profit is the profit of the order so the total sale price minus the total cost price.

Our OrderRepository is very similar to the previous two repositories. It contains a few methods we have added ourselves on top of the methods already provided by the CrudRepsoitory extension.

```
public interface OrderRepository extends CrudRepository
    List<Order> findByOid(long oid);
    Integer deleteByOid(long oid);
    List<Order> findByUid(long uid);
}
```

The OrderController is once again where all the request methods are declared. These methods are what the frontend requests when they want to manipulate data to do with an order or orders. One of these methods is used to return the most recent five orders that have been made. This method finds all the orders for the currently logged in user and creates a sub list of the last five that have been added to the database. These are then returned.

```
@GetMapping("/orders/recent/{username}")
public List<Order> getRecentOrders(@PathVariable("username")String username){

    List<Order> orders = new ArrayList<>();
    repository.findByUid(userRepo.findAllByUsername(username).getId()).forEach(orders::add);

    if(orders.size() > 5) {
        return orders.subList(orders.size() - 6, orders.size() - 1 );
    }
    return orders;
}
```

Another method in the OrderController that we have created is the method that allows the user to create an order. This method is important as when it is called it also updates the stock in the products table. This means that if an order of 20 was made for a product then its stock would decrease by 20 in the database.

```
//Update Stock
for(Product p: products) {
    if(order.getAmount() > p.getStock()) {
        throw new Exception();
    }
    total = (p.getSale_price() * order.getAmount());
    p.setStock(p.getStock()-order.getAmount());
}
//Get Profit
```

When creating the order we also have to calculate the profit for the order. This is done by getting the total sale price and taking away the total cost price from it. This is done as shown below.

```
//Get Profit
for(Product p: products) {
    float sale = (order.getAmount() * p.getSale_price());
    float cost = (order.getAmount() * p.getCost_price());
    profit = sale - cost;
}
```

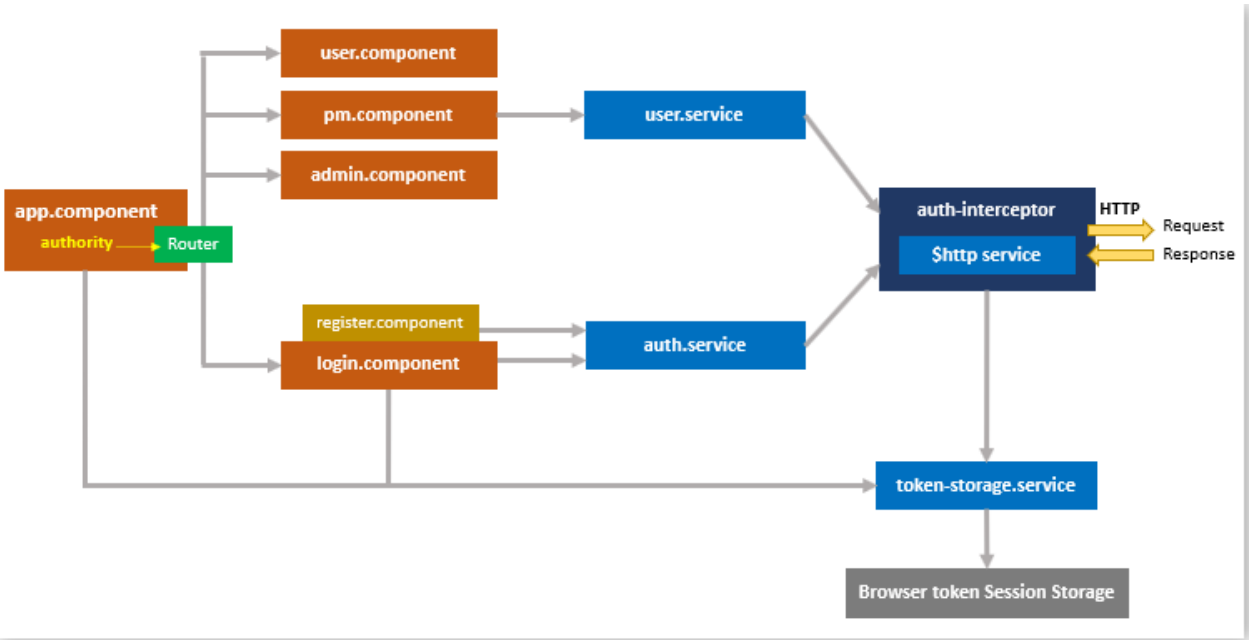
With all our models, controllers and repositories made for our users, customers, products and orders, it was now possible for us to fully test the backend section of our application. We did this using Postman, this allowed us to test all our request methods and make sure they were working correctly. It allowed us to see if the database was being updated or not. We had a few small fixes to make after testing through Postman and once these were complete it was time for us to move onto our frontend.

5.3 Frontend Layer

The frontend layer of our system design is where all the user interaction with our application is contained. This is where all the work is done in terms of how our application will be displayed to the user. It is the layer in which all communication from the user to the backend of the application takes place. This is where the user will be passing information the backend and where they will be viewing all their information. As we did with the previous two layers, we researched possible frontends we could use for our project. After looking at a few of the options, we were stuck to choose between Angular and React. We felt both technologies would be very suitable for our project but in the end decided to proceed with using Angular. We had a little experience using Angular and felt with a bit of research and learning, we would be able to implement it nicely into our project.

5.3.1 Angular

Our first step with our frontend was to get a basic Angular app up and running. This would give us a base to work off so that we could then add to it to connect it to our backend. Once we had this basic angular app up and running, we started work on getting our login and signup page created. This is where all our authorization would be taking place. This the architecture for this part of the frontend.



app.component is the parent component that contains routerLink and router-outlet for routing. It also has an authority variable as the condition for displaying items on navigation bar. This

means that depending on what authority the user has, the items on the navigation bar will be visible or not.
user.component, pm.component, admin.component correspond to Angular Components for User Board, PM Board, Admin Board. Each Board uses user.service to access authority data. At first, we planned on implementing different boards for different levels of users to access depending on their role, however as we ran a bit short on time, we did not implement this fully into our project.

register.component contains the User Registration form, when the user submits this form, the auth.service is called. login.component contains the User Login form; submission of the form will call auth.service and token-storage.service. The user.service gets access to authority data from the server using the Angular HttpClient.

```
this.authService.attemptAuth(this.loginInfo).subscribe(
  data => {
    this.tokenStorage.saveToken(data.accessToken);
    this.tokenStorage.saveUsername(data.username);
    this.tokenStorage.saveAuthorities(data.authorities);

    this.isLoginFailed = false;
    this.isLoggedIn = true;
    this.roles = this.tokenStorage.getAuthorities();
    this.reloadPage();
  },
```

The auth.service then handles the authentication and signup actions with the server using the Angular HttpClient.

```
attemptAuth(credentials: AuthLoginInfo): Observable<JwtResponse> {
  return this.http.post<JwtResponse>(this.loginUrl, credentials, httpOptions);
}
```

Every HTTP request by this Angular HttpClient will be inspected and transformed before being sent to the server by the auth-interceptor which implements HttpInterceptor. The auth-interceptor checks and gets a token from the token-storage.service to add to the Authorization Header of the HTTP Requests.

```
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
    constructor(private token: TokenStorageService) { }
    intercept(req: HttpRequest<any>, next: HttpHandler) {
        let authReq = req;
        const token = this.token.getToken();
        if (token != null) {
            authReq = req.clone({ headers: req.headers.set(TOKEN_HEADER_KEY, 'Bearer ' + token) });
        }
        return next.handle(authReq);
    }
}
```

The token-storage.service then manages the token inside the browser's session storage. When the user signs out the token is cleared from the session storage through the token-storage.service.

```
signOut() {
  window.sessionStorage.clear();
}
```

Now that we had working login and sign up front end working for our project, we were able to test it out with our database and spring boot server. We started up the project by running the Java Spring application and serving the Angular app in our localhost. We saw the login page appeared and were able to login successfully using the credentials we had created earlier in the Postman testing. We then tried the sign-up option, and this also worked correctly, updating our database as the user was created. Now that this was complete, we could start work on our three main pages. These were our customers, products and orders pages.

We started with our customers page. On this page we wanted to display all the customers the currently logged in user had and the option to create a new customer. We started off by creating a customer service and three components; create-customer, customer-list and customer-details. We also created a customer model which just declared the variables that each customer has.

```
export class Customer {
    cid: number;
    cname: string;
    address: string;
    phone: string;
    uid: number;
    amount_purchased: number;
}
```

Inside our customer-details component we updated the html file to contain an "ngIf" for the customer to go through and display its information. We also made a delete customer button which will be used for deleting the customer should the user wish to.

```
<div *ngIf="customer">
  <div>
    <label>Name: </label> {{customer.name}}
  </div>
  <div>
    <label>Address: </label> {{customer.address}}
  </div>
  <div>
    <label>Phone: </label> {{customer.phone}}
  </div>

 <div>
  <button type="button" class="button btn-danger" (click)='deleteCustomer()'>Delete</button>
</div>

  <hr/>
</div>
```

In the customer-list component we updated the html to use an “ngIf” to run through all the customers while displaying the customer details by calling the customer-details component.

```
<h1>Customers</h1>

<div *ngFor="let customer of customers | async" style="width: 300px;">
  <customer-details [customer]='customer'></customer-details>
</div>

<p> <a routerLink="../addCustomer" class="btn btn-primary active" role="button" routerLinkActive="active">Add Customer</a></p>


<div>
  <button type="button" class="button btn-danger" (click)='deleteCustomers()'>Delete All</button>
</div>
```

In the create-customer component we created a form in the html that allows the user to enter all the details for the customer they want to create. The save() method is then called to create the new customer.

```
save() {
  this.customerService.createCustomer(this.customer, this.tokenStorageService.getUsername())
    .subscribe(data => console.log(data), error => console.log(error));
  this.customer = new Customer();
}
```

The customer service controls all the requests to the backend concerning the customer. It contains several methods which all return different results. The base URL is declared which is then called inside each request. This URL matches the location of the CustomerController in the backend which is where the customer request methods are located.

```
private baseUrl = 'http://localhost:8080/api/customers';
```

We then moved onto our products page. This would be very similar to the customers page we created, and we used very similar steps as we did for creating that. We created a product service and three product components; create-product, product-list and product-details. We also created a product model which just declared the variables that each product has.

```
export class Product {
    pid: number;
    pname: string;
    stock: number;
    cost_price: number;
    sale_price: number;
    uid: number;
}
```

Inside our product-details component we updated the html file to contain an “ngIf” for the product to go through and display its information. We also made a delete product button which will be used for deleting the product from the products table in the database.

```
<div *ngIf="product">
  <div>
      <label>Product ID: </label> {{product.pid}}
    </div>
  <div>
    <label>Product name: </label> {{product.pname}}
  </div>
  <div>
    <label>Stock: </label> {{product.stock}}
  </div>
  <div>
    <label>Cost Price: </label> {{product.cost_price}}
  </div>
  <div>
    <label>Sale Price: </label> {{product.sale_price}}
  </div>
  <div>
    <label>User ID: </label> {{product.uid}}
  </div>

  <div>
  <button type="button" class="button btn-danger" (click)='deleteProduct()'>Delete</button>
  </div>
```

Once again, the product-list was very similar to what we had done previously in the customer-list component. We updated the html to use an “ngIf” to run through all the products while displaying the product details by calling the customer-details component.

```
<div *ngFor="let product of products | async" style="width: 300px;">
  <product-details [product]='product'></product-details>
</div>

<p> <a routerLink="../addProduct" class="btn btn-primary active" role="button" routerLinkActive="active">Add Product</a></p>


<div>
  <button type="button" class="button btn-danger" (click)='deleteProducts()'>Delete All</button>
</div>
```

In the create-product component, we created a form in the html that allows the user to enter all the details for the product they wish to create.

```
Create Product</h3>
 [hidden]="submitted" style="width: 300px;">
orm (ngSubmit)="onSubmit()">
<div class="form-group">
  <label for="name">Name</label>
  <input type="text" class="form-control" id="name" required [(ngModel)]="product.pname" name="pname">
</div>

<div class="form-group">
  <label for="stock">Stock</label>
  <input type="number" class="form-control" id="stock" required [(ngModel)]="product.stock" name="stock">
</div>

<div class="form-group">
    <label for="cost_price">Cost Price</label>
    <input type="number" step="0.01" class="form-control" id="cost_price" required [(ngModel)]="product.cost_price" name="cost_price">
  </div>

  <div class="form-group">
      <label for="sale_price">Sale Price</label>
      <input type="number" step="0.01" class="form-control" id="sale_price" required [(ngModel)]="product.sale_price" name="sale_price">
    </div>

<button type="submit" class="btn btn-success">Submit</button>
form>
v>

 [hidden]="!submitted">
4>You submitted successfully!</h4>
utton class="btn btn-success" (click)="newProduct()">Add</button>
v>
```

The product service controls all the requests to the backend concerning the product. It contains several methods which all return different results depending on the method the user has called. The base URL is declared which is then called inside each request. This URL matches the location of the ProductController in the backend which is where the product request methods are located.

```
private baseUrl = 'http://localhost:8080/api/products';
```

Finally, we had to complete our orders page. This would be like the previous pages but had a slight difference as in it would use products and customers when creating an order. We created an order service and three order components; create-order, order-list and order-details. We also created an order model which just declared the variables that each order would have.

```
export class Order {
    oid: number;
    uid: number;
    cname: string;
    pname: string;
    amount: number;
    total: number;
    profit: number;
}
```

Inside our order-details component we updated the html file to contain an “ngIf” for the order to go through and display its information. We also made a delete order button which will be used for deleting an order from the orders table in the database.

```
<div *ngIf="order">
    <div>
      <label>Order ID: </label> {{order.oid}}
    </div>
    <div>
      <label>Customer's Name: </label> {{order.cname}}
    </div>
    <div>
      <label>Products: </label> {{order.pname}}
    </div>
    <div>
      <label>Amount: </label> {{order.amount}}
    </div>
    <div>
      <label>Total: </label> {{order.total}}
    </div>
    <div>
      <label>Profit: </label> {{order.profit}}
    </div>

   <div>
    <button type="button" class="button btn-danger" (click)='deleteOrder()'>Delete</button>
  </div>
```

order-list was similar to what we had done previously in the list components. We had to update the html to use an “ngIf” to run through all the orders currently in the database for the current user and display the order details by calling the order-details component. There is also a button with the option to delete all the orders in the database.

```
<div *ngFor="let order of orders | async">
  <order-details [order]='order'></order-details>
</div>


<p> <a routerLink="../addOrder" class="btn btn-primary active" role="button" routerLinkActive="active">Add Order</a></p>


<div>
  <button type="button" class="button btn-danger" (click)='deleteAll()'>Delete All</button>
</div>
```

For creating an order, we used the create-order component. We made a form in the html that allows the user to enter all the details for the order they wished to create. There was a lot of detail in this form as we wanted the user to be able to pick from a dropdown of their current customers so that they could pick one for the order.

```
<div class="form-group">
  <label for="cname">Customer</label>
    <br/>
    <select name="customer" [(ngModel)]="order.cname" required>
      <option *ngFor="let customer of customers | async" [ngValue]="customer.name" > {{customer.name}} </option>
    </select>
</div>
```

We also had to create a dropdown option for what product was being put in this order. This was done in the same way as the customer dropdown list. We had to let the user enter an amount for the product also.

```
<div class="form-group">
  <label for="name">Products:</label>
  <select name="product" [(ngModel)]="order.pname" required>
    <option *ngFor="let product of products | async" [ngValue]="product.pname"> {{product.pname}} </option>
  </select>
  <input [(ngModel)]="order.amount" min=1 type="number" id="amount" class="amount" name="amount"/>
</div>
```

The order service controls all the requests to the backend concerning the orders. It contains several methods which all return different results depending on the method the user has called. The base URL is declared which is then called inside each request. This URL matches the location of the OrderController in the backend which is where the product request methods are located. Below shows where the base URL is being used in the method path.

```
private baseUrl = 'http://localhost:8080/api/orders';

constructor(private http: HttpClient) { }

getOrder(oid: number): Observable<Object> {
  return this.http.get(`${this.baseUrl}/${oid}`);
}
```

Once we had all these pages created, we could test them all out by creating various customers, products and orders and making sure they all appeared on the appropriate pages. With us happy that they were working in the intended way, we decided to focus on completing our home page. We wanted our homepage to display some information to the user as soon as they logged onto the site. We chose three separate pieces of information to display, a table containing the five most recent orders a user has made, a table containing the top customers a user has and a pie-chart displaying their customers and how much they have purchased from them. We done the two tables first as they were both very similar. The data passed to these tables are retrieved by calling the two methods shown below. These return the two lists that we want to then run through in our html.

```
reloadData() {
  this.orders = this.orderService.getRecentOrders(this.token.getUsername());
  this.customers = this.customerService.getTopCustomers(this.token.getUsername());
}
```

In our html we use a “ngFor” statement to run through the list of orders and customers and display the certain information we want in each table. We then use a “ngIf” statement to get the information for the current order or customer the “ngFor” is looping through.

```
<div *ngFor="let order of orders | async" >
    <div *ngIf="order" >
      <table >
      <tr>
        <th>ID</th>
        <th>Customer</th>
        <th>Product</th>
      </tr>
      <tr>
        <td>{{order.oid}}</td>
        <td>{{order.cname}}</td>
        <td>{{order.pname}}</td>
      </tr>
      </table>
```

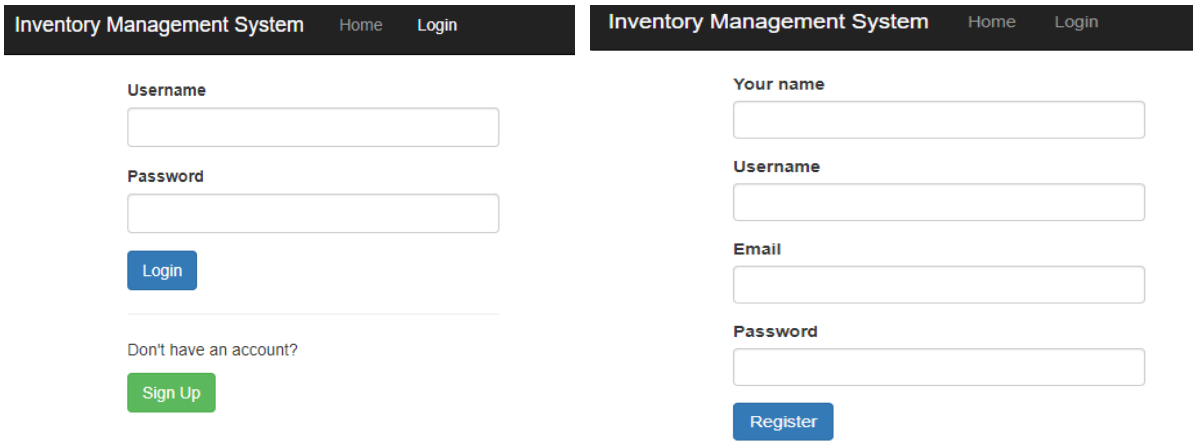
```
<div *ngFor="let customer of customers | async" >
  <div *ngIf="customer" >
      <table >
      <tr>
        <th>Customer</th>
        <th>Total Purchased</th>
      </tr>
      <tr>
        <td>{{customer.name}}</td>
        <td>€{{customer.amount_purchased}}</td>
      </tr>
      </table>
```

For the chart we used an Angular chart import called amCharts. This helped us to be able create a nicely designed pie chart to display our information. We had to create the chart and then add our data to it so it could be displayed correctly.

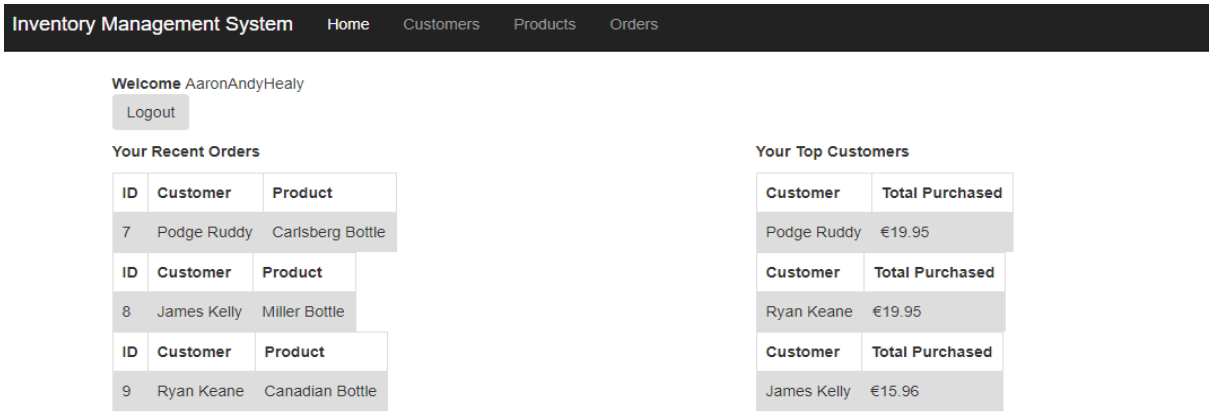
```
this.zone.runOutsideAngular(() => {
  let chart = am4core.create("chartdiv", am4charts.PieChart);
  chart.paddingRight = 20;
  let data = [];

  // Add data

 this.customers.subscribe(customer=>{
    customer.forEach(customer => {
      chart.data.push({
      "customer": customer.cid,
      "amount": customer.amount_purchased
      });
    });
})
  // Add and configure Series
  let pieSeries = chart.series.push(new am4charts.PieSeries());
  pieSeries.dataFields.value = "amount";
  pieSeries.dataFields.category = "customer";
  this.chart = chart;
});
```

After this was complete, we ran the project in full to make sure everything was correct and displaying properly. We have some screenshots from our application frontend running now.

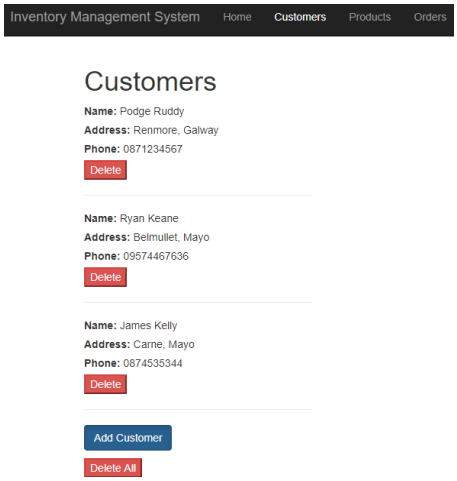
Login and Sign Up Page

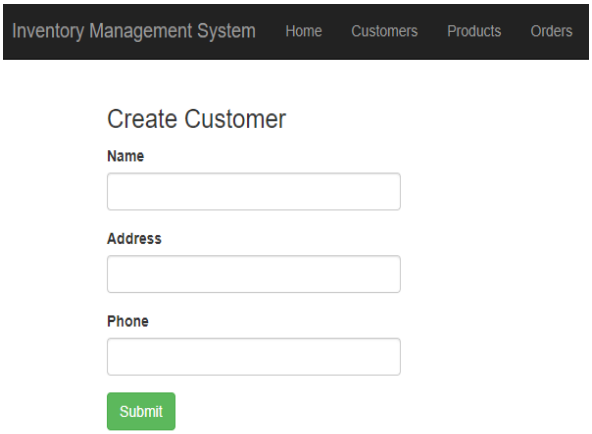


Home Page

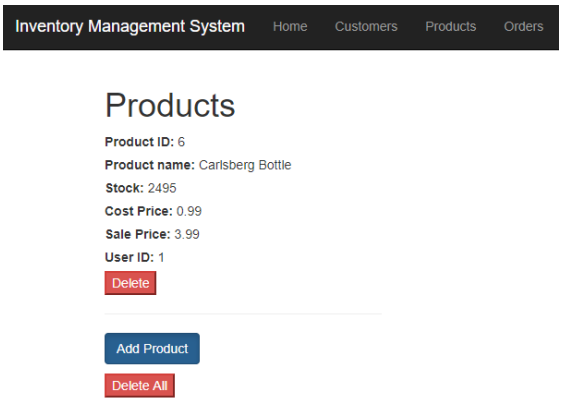


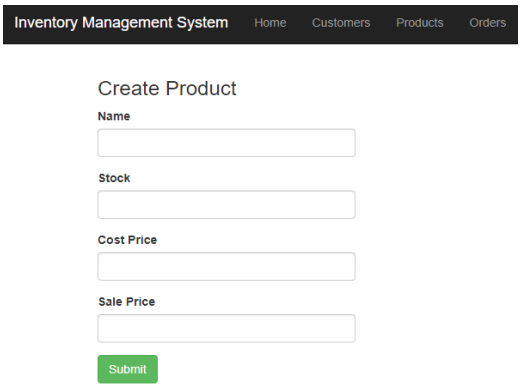
Customers Page



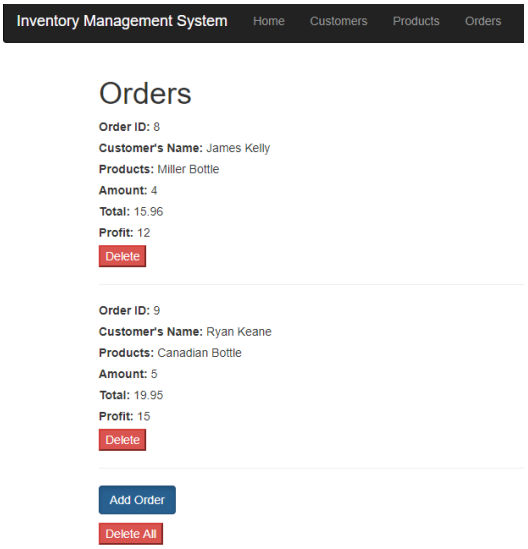


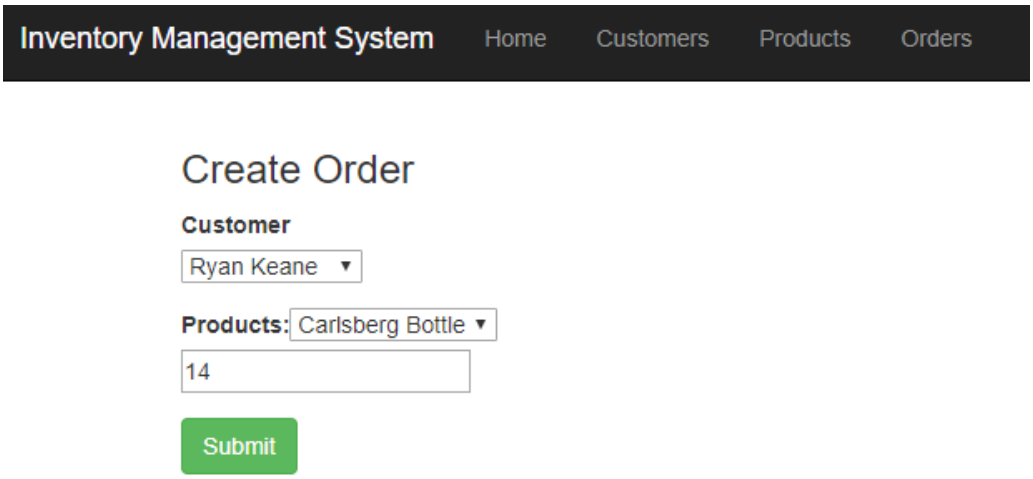
Products Page





Orders Page


Inventory Management System
Home
Customers
Products
Orders
Orders
Order ID: 8
Customer's Name: James Kelly
Products: Miller Bottle
Amount: 4
Total: 15.96
Profit: 12
Delete
Order ID: 9
Customer's Name: Ryan Keane
Products: Canadian Bottle
Amount: 5
Total: 19.95
Profit: 15
Delete
Add Order
Delete All


Inventory Management System
Home
Customers
Products
Orders
Create Order
Customer
Ryan Keane
Products:
Carlsberg Bottle
14
Submit

Chapter 6
System Evaluation

After competing the development of our application, we had to look at evaluating its performance. We aimed to evaluate our system in the following areas:

- Robustness
- Testing
- Limitations
- Results vs Objectives

6.1 Robustness

To measure the robustness of our application we agreed to focus on this part of the evaluation during the design and implementation process. We measured the usability of our software artifacts and API's through rigorous unit testing of our components, and in addition, we measured the results of our API and HTTP requests using tools such as Postman and Chrome Dev tools.

6.2 Testing

Through continuous white box testing and regular black box testing we believe our application has reached the level design and productivity that we envisioned when we set about designing this project. In order to achieve the level, we had set for this application, we carried out white box testing in the following ways:

Ng Serve
We used this method of testing in our browser as it allowed us to view the application as it was still being developed. It allowed us to check for bugs after a new addition to our code. To use this method of testing we simply had to navigate into the root of the Angular App folder in the command line and run the following command:
ng serve

We found this testing method very helpful to us during the development of our application as it provides a live-reload server. This meant once we made a change in our Angular code, it would re-compile and show if any errors had been created. It would also reload the browser showing the updated code. This would allow us to test the functionality of the code we had just added.

6.3 Limitations

The following are some of the limitations in our IMS application:

6.3.1 Order Limitations

In our application we process a single order at a time for a single product. This is mainly due to the schema for our database which doesn't account for user's placing an order of multiple

products. Instead our application allows the user to place an order for one product for each order.
We looked at this and determined that we could create a separate database using MongoDB which is a document-oriented database. This would allow us to load a number of products into an order and parse the result as a JSON object to a MongoDB document.
Another option would to be create another table in our MySQL database which would store BLOB's of the JSON data from the order placed. The former would have been the solution we would have considered the most but due to time constraints we shelved that feature and focused on other areas of the project.

6.3.2 User notifications of stockouts

We also looked at the possibility of adding the feature for a user to receive notifications in the event of inventory count running low or to manage the risk of stockouts for any product.

6.4 Results V Objectives

At the start of this project, we set certain objectives which we had planned to complete successfully throughout the development of our application. We feel that if you compare the objectives, we outlined at the start to the application that has been developed, then we have gone a long way to completing these objectives. Our application has met all the goals we set out for it to a certain level. Throughout the development we were met with obstacles, which sometimes impacted on how far we could go to meeting certain objectives we had planned to but overall, we feel like our results mirror the objectives we set out for ourselves.

Two of our main objectives that we set ourselves at the start of the project were to gain experience building scalable web applications and learn about the latest tools that could help in building these applications. We felt these were very important objectives as this being a college assignment, it is very important that we learn something from completing the project. Throughout the development we both learned an enormous amount about how scalable web applications operate and what is required to develop them to an industry standard.

Chapter 7
Conclusion

This report has discussed the development of an inventory management system (IMS) using some of the latest tools and technologies, whilst adopting an agile development approach to the software development lifecycle (SDLC). From the beginning we wanted to build an application that would be practical and user-friendly, but furthermore, we strived to apply the skills that we have learned - throughout our four years of software development – to build an application that would meet, the functional requirements needed for such a project, and also, the learning outcomes on completion of the module, and course.

In summary, the main objectives for this project were outlined as such:

- Develop a web application that allows a user to login and based on the details given, that user can access information specifically relating to them and their customers.
- Build a scalable backend system that would be able to manage and maintain the data between the user's products, orders, and their customers.
- Build a user-friendly frontend web application that displays the correct data for the specific user based of their business' interactions, orders, stock and products.
- Secure a users' information using authentication and authorization protocols like JSON Web Tokens (JWT)
- Display generated table information based of a user's orders, customers, and their products, and use this data to build graphical charts so a user can better visualize their business.
- Gain experience in building scalable web applications and working collaboratively as a team of developers.
- Learn about new or upcoming trends or tools that might fit the solution we are developing or assess their potential in integrating such technologies with our solution.

The application allows a user to sign up or log in while encrypting and managing access to that persons data using web and security protocols, and also provides details on products or orders they have, managing their stock and the relationship between those three and to which customer it applies to. This combined with expressing the data visually for each individual user.

This doesn't seem new and inventive, but most businesses and wholesalers still use these applications. Given the unyielding necessity to manage stock and orders is still as ever present as ever, we believed that a practical application, that focused on security, scalability and user-friendliness would help achieve the goals of this project and give us further experience in all aspects of the SDLC.

The project ran through several phases and as such, we both wanted to adopt an agile approach to software development whilst undertaking this project. Our initial step was to define some system requirements for the IMS. These would act as user features, where we would then provide details on the estimated amount of time for completion of said feature.

This step was necessary to give us both an overview of the tasks that needed to be completed and allowed us to define a timescale for the project.

We began this project as we meant to go on; determining critical aspects of the project that would allow us to do as much collaborative work as possible, thus allowing us to prioritize a certain feature for our application that maybe needed to be fixed, tested, or evaluated.

We dissected the project into tasks, milestones, sprints, and releases. Then began building and testing several components for our application using the skills and technologies we gained and most comfortably used throughout this course, therefore giving us the best possible outcome on completion of the application.

At the beginning of each sprint we outlined the tasks that needed to be completed during that iteration. This helped prioritize tasks and help us complete each as efficiently as expected. When a feature was undertaken during a sprint, we both divided up the work into smaller tasks. This process was similar for all features of the application and as such we felt that we adapted to working in iterations quickly. It also helped us to identify the work we needed to perform along with the commitment to delivering a product increment at the end of every iteration.

Adopting this methodology, we successfully:

- Identified and defined the user requirements for an IMS, therefore allowing us to highlight and prioritize certain features for our application.
- Planned our project accordingly by creating roadmaps or project timelines, giving us a high-level view of the requirements and allowing us to roughly estimate the time and effort needed to complete each feature. We made sure to revise this as much as possible.
- Created user flow and high-level UML diagrams to demonstrate certain features of the application at the beginning of our planning process before each iteration of our sprint cycle.
- Followed this project plan to produce a consistent workflow and provide an oversight of tasks to be completed before each release
- Worked collaboratively using a version control management system such as Git, which allowed us to communicate the work we were doing easier
- Detail any issues that we were having during each iteration of our development process and resolve them before the next sprint.

This project has given us the opportunity to gain valuable insight and experience into the process of the SDLC. Using the tools and methodologies that we have been taught, we were able to develop and deliver a solution that we believe is robust, scalable, and meets the requirements set out in the initial objectives for our project.

We learned more about the different stages of the SDLC, allowing us to better identify and define requirements through the process of documenting them in the form software requirement specification. Following the methods set out in the SDLC, we turned the specifications into a design plan that consisted of wireframes, UML diagrams, process/user flow diagrams among other things.

Building the solution quickly followed and after carefully following the previous steps, we ran into fewer issues than expected, and with the issues we did run into, we managed to resolve them with minimal disruption to the plan. Building happened during each sprint, and this was the most crucial period for us in terms of completing tasks, getting artefacts ready for testing, deployment, and maintaining those in future releases.

We both agreed that working with these stages and goals in place has helped us in eventually completing the project. We met many obstacles along the way - these were usually concerning other matters of our degree – which did take our focus away from the project for a few weeks or a month at a time. We feel that not having planned and carried out the tasks we defined throughout the course of the project, we would have been hard pressed to complete the overall project.

To conclude, we are both grateful to have had the opportunity to work on a project of this scale and to be asked to learn more about the core fundamentals, new technologies, and efficient methodologies in a software development project. We learned about new techniques for developing restful web applications, different tools for managing versions within a project, new technologies for packaging and distributing software applications, and web and security services or protocols. We feel that through more frontend design and the application could easily meet the needs of small to medium sized business' and wholesalers.

Chapter 8
Appendix

Project Source Code Link: https://github.com/conormc93/FinalYearProject

Installation instructions:

1. git clone https://github.com/conormc93/FinalYearProject.git
2. Open Java IDE and import existing Maven Project.
3. Open a command prompt in the "FrontEnd" sub-directory of the project folder.
4. Run the Spring application as a "Java Application"
5. In the command prompt type "Ng serve" after the Spring application starts.
6. After successful compilation open a web browser and navigate to "http://localhost:4200/"

Bibliography

[1] Dimitris Bertsimas, Inventory Management in the Era of Big Data, Production & Operations Management Vol 25 Issue 12, 2016

[2] G Ranganatham, Inventory Management Practices in Small Scale Enterprises, BVIMR Management Edge Vol 7 Issue 2, 2014

[3] B. Sai Subrahmanya Tejesh, Warehouse Inventory System using IoT and Open Source Framework, Alexandria Engineering Journal, 2018

[4] The Many Types of Inventory Management Technology, Veridian, 15-Jan-2019. [Online]. Available: https://veridian.info/inventory-management-technology/?fbclid=IwAR1-Rh9KMYE4ujrRs5DvLhQhD54wf3-jEKhayHwAHluPEDt28l8XY_KQTCw.

[5] Olivia Durden, How to Use Technology to Track Inventory, Chron, 06-Mar-2019. [Online]. Available: https://smallbusiness.chron.com/use-technology-track-inventory-64737.html?fbclid=IwAR2fIZh8fdKQJ_LKWjj50PPePsovRQM7gOGUsiLOURJeR5PYg3dPdknpcxg

[6] Utilizing WMS: Top Ways to Get the Most out of Your WMS Investment, Veridian, 10-Jul-2017. [Online]. Available: https://veridian.info/wms/?fbclid=IwAR3H9YybpS-Xkw3Gfe-Y7v8M_jgBT8ZAafk-veT9m-so85Vs4qrP8Kyy5rc

[7] Types of Inventory Systems, Acctivate, [Online]. Available: https://acctivate.com/types-of-inventory-systems/?fbclid=IwAR3RaQhQnwIBOvq-X7bmIh0FClVSRdu7xT5zDoNTa0VUZuSdGzDAB0y2oqI

[8] Nicole Pontius, 4 Types of Inventory Control Systems: Perpetual vs. Periodic Inventory Control and the Inventory Management Systems That Support Them, Camcode, 15-Jan-2019, [Online]. Available: https://www.camcode.com/asset-tags/inventory-control-systems-types/?fbclid=IwAR1-Rh9KMYE4ujrRs5DvLhQhD54wf3-jEKhayHwAHluPEDt28l8XY_KQTCw

[9] Johnny Marx, What is Inventory Control ?, Handshake,28-Mar-2018, [Online]. Available: https://www.handshake.com/blog/what-is-inventory-control?fbclid=IwAR1V5I4-y4GYU3w2CEteR3Zg0QXnAy8z5qbaqsKGM4lMKAeIsBrnk4g2k8Y

[10] Why Inventory Management is Critical for Your Operation, Datexcorp, [Online]. Available: https://www.datexcorp.com/why-inventory-management-is-critical-for-your-operation/?fbclid=IwAR3H9YybpS-Xkw3Gfe-Y7v8M_jgBT8ZAafk-veT9m-so85Vs4qrP8Kyy5rc

[11] Paul Trujillo, Inventory Management 101, WaspBarcode, 29-Jan-2015, [Online]. Available: http://www.waspbarcode.com/buzz/inventory-management-101/?fbclid=IwAR1vSPLZKNUL7SORWVuSY4O4lHsyyQ0MSit-bv4Zzr6sevx79sWEXZjiMhY

[12] Baeldung, Spring Security Login Page with Angular, 06-Nov-2018, [Online]. Available: https://www.baeldung.com/spring-security-login-angular?fbclid=IwAR3RaQhQnwIBOvq-X7bmIh0FClVSRdu7xT5zDoNTa0VUZuSdGzDAB0y2oqI

[13] Sebastian Chocke, How to Do JWT Authentication with an Angular 6 SPA, Toptal, [Online]. Available: https://www.toptal.com/angular/angular-6-jwt-authentication?fbclid=IwAR20Itfawl66W1H6KjKKrAEabOpQFWCSe8uSrSDmU-lY2yTFkrm0fdP1ZyE

[14] Zoltán Raffai, Getting Started with Java Spring, [Online]. Available: https://www.zoltanraffai.com/blog/page/2/?fbclid=IwAR3H9YybpS-Xkw3Gfe-Y7v8M_jgBT8ZAafk-veT9m-so85Vs4qrP8Kyy5rc

[15] Petri Kainulainen, Understanding Spring Web Application Architecture: The Classic Way, 19-Oct-2014, [Online]. Available: https://www.petrikainulainen.net/software-development/design/understanding-spring-web-application-architecture-the-classic-way/?fbclid=IwAR2vNTyhqE4rX-zkBy1XSBuMqxEhPGLynPgkL-KS5q8Cv_l6Br0oSo-oafQ