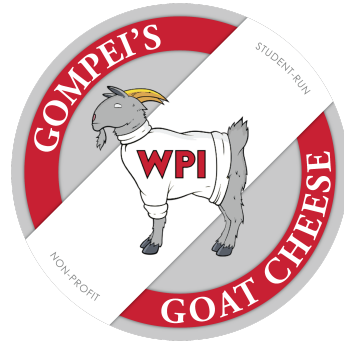




WPI



Continuing Development of a Cloud-Based Enterprise Resource Planning System for Gompei's Goat Cheese

A Major Qualifying Project Submitted to the Faculty of WORCESTER POLYTECHNIC INSTITUTE in partial fulfillment of the requirements for the Degree of Bachelor of Science

By:

Gabrielle Acquista
BS in Computer Science

Victoria Buyck
BS in User Experience Design

Benjamin Sakac
BS in Management Information Systems

Sohrob Yaghouti
BS in Management Information Systems

Date: April 26, 2023

Submitted to:

Gompei's Goat Cheese, Unofficial Sponsor
James E. Ryan, MIS Advisor
Michael Engling, CS Advisor

This report represents the work of WPI undergraduate students submitted to the faculty as evidence of completion of a degree requirement. WPI routinely publishes these reports on its website without editorial or peer review.

Abstract

Gompei's Goat Cheese is a non-profit, student-run business with a mission of supporting global scholarships, promoting entrepreneurial skills, and allowing members to be part of something that has real impact. However, their operations would be improved by an enterprise resource planning (ERP) system. A previous MQP designed and began development on such a system. The goal of this project was to continue development of a cloud-based ERP system to improve the efficiency of GGC's operations. After gaining user feedback on a high-fidelity prototype, the team iterated on the previous design and continued development work. A functional prototype brings GGC one step closer to a system that provides them the benefits of a cloud based platform that will centralize and unify their data.

Acknowledgements

We would first like to thank Gompei's Goat Cheese for sponsoring this MQP. They have continued to build on this organization for years since the previous MQP groups that started GGC in 2013 and 2014. We are also excited to congratulate their founders and the current GGC team as GGC has reached its ten year anniversary on April 18, 2023.

We are incredibly grateful for Jeremy Berman, Danielle Payne, Joseph Botelho and Rodrigo Calles; these dedicated students created the foundation for this thriving student-run business and for the future of student-run entrepreneurial endeavors at WPI. Without them, none of this would have been possible. We would also like to thank the current GGC executive board for their time and participation in our user study. Their feedback shaped the way we designed our iteration of this system so we couldn't have done it without their help.

We would also like to thank our advisors Professor Jim Ryan and Professor Michael Engling for their continued support throughout the course of this project. We couldn't have made it through without their guidance, kindness, and understanding.

And finally, thank you to the goats. All of them. Without them, we have no goat cheese. At that point it's just cheese which definitely isn't as cool. Also thanks Gompei, our mascot, we love you!

Executive Summary

Gompei's Goat Cheese (GGC) is a non-profit student-run business that sells award winning cheese from Westfield Farm in Hubbardston, Massachusetts. The stated mission of GGC is "to support Worcester Polytechnic Institute's (WPI) global scholarships, learn entrepreneurial skills, and interact with the WPI and Worcester community by selling award-winning goat cheese so that we can be a part of something meaningful that impacts our community" (Gompei's, 2022). However, GGC is in need of a new system for order and payment tracking, as their current system is prone to inconsistencies.

An enterprise resource planning system (ERP) centralizes all the information used for organizing and directing an organization. While large corporations use ERP systems that integrate accounting, finance, production, inventory, order entry, and logistic operations and more, a smaller-scale system custom built to address deficiencies with GGC's current record keeping system could help the company to

grow and have a more meaningful impact in the community. Figure E.1 displays several characteristics that GGC shares with small businesses and startup companies that were considered when deciding how the system should function.

Last year, two students and members of GGC proposed a Major Qualifying Project with the goal of designing a cloud-based ERP system for order tracking. They began their project by interviewing several GGC members and the owner of Westfield Farm in order to evaluate the current operations and create a set of requirements for the new system. This Phase One team then began to design and then develop the system they proposed. They started their design phase by creating a set of use cases for the new system and then developing data flow diagrams as well as an entity relationship diagram (ERD). Then, they created a user flow diagram depicting a few paths that a typical user interaction with the system might follow. Their last step before beginning development was to design the system mock-ups on Figma.

The Phase One team used Figma's flow features to convert the mockups into a high

	GGC	Small Businesses	Startups
Gives Back To Local Economy	✓	✓	✗
Employees From Local Community	✓	✓	✗
High Employee Turnover Rate	✓	✗	✓

Figure E.1: Comparing GGC to Small Businesses and Startups

fidelity prototype of the system. Finally, they began development of a functional prototype on the AWS Amplify Studio platform using React.js for the front-end and DynamoDB for the database. This NoSQL database was the only configuration that could interface directly into Amplify Studio. It just required a GraphQL API to connect the front-end with the database.

Goal and Objectives

The goal of this project was to continue development of the cloud-based enterprise resource planning system meant to improve the efficiency of GGC’s operations. In order to successfully continue the work of the Phase One team, it made sense for the project to continue following the same system development life cycle (SDLC) methodology of system prototyping. The system prototyping methodology aims to quickly develop a simplified version of the final product and then continuously refine it until stakeholders agree it is functional to be implemented.



Figure E.2: System Prototyping Methodology and Project Objectives

1. Evaluate Existing Prototype

The team conducted user testing using the think-aloud methodology to gain feedback on the Phase One prototype in order to evaluate it. The think-aloud method was chosen because it is user centered, easy to implement, and allowed the team to directly observe the user’s reactions. In addition to the think-aloud testing, the team also asked participants to fill out a System Usability Scale (SUS) survey providing quantitative analysis on the usability of the system prototype.

2. Iterate on Phase One Design

In order to iterate on the design of Phase One, our team did the following: updated the entity relationship diagram, list of use cases, and data flow diagrams; incorporated the feedback from user testing into the high-fidelity prototype; and overhauled the software architecture to create a centralized, predictable, and scalable system.

3. Develop a Functional Prototype

The team developed a functional prototype using a MySQL database, a Node Express.js server, a React.js

front-end with Redux for state management, and the Axios library to handle HTTP requests. Additionally, the team designed a cloud architecture in Amazon Web Services establishing the foundation for an implementation in the future.

4. Create Support Documentation

To encourage the continuation of the project, the team developed further support documentation with information supplemental to the report that will aid in the continued development and implementation of the new system.

User Testing

In order to conduct useful user testing sessions, the team first modified the high-fidelity prototype from Phase One. These modifications were made so that participants could complete ten tasks in the prototype. Appendix B contains the full user testing protocol including the list of tasks. One of the suggestions for future work from Phase One was to look at ways to reduce redundancy by automating order entry for GGC operations. Through user testing, the team learned that one of GGC’s standard operating procedures is to delay sending orders to the farm in case a customer wants

to make an alteration to their order. Not wanting to disrupt this process, the team instead focused on making user experience improvements to the order form used by customers.

The team then split observations from user-testing and the participants’ comments into three categories: confusing points, improvement suggestions, and additional features. The confusing points and some suggestions were considered when iterating on the design, but the additional features suggested could not be included due to time constraints. Quantitative analysis revealed

Confusing Points	Improvement Suggestions
<ul style="list-style-type: none"> • Difference between orders and invoices • Button placement • Too many fields 	<ul style="list-style-type: none"> • More info on order card <ul style="list-style-type: none"> ◦ Date and last edited • Permissions for roles <ul style="list-style-type: none"> ◦ Finance ◦ Accounting ◦ View Only ◦ Farm Owner ◦ Data Person ◦ Admin
Additional Features	
<ul style="list-style-type: none"> • Data visualization • Notification center • Sticker label tracking • In person orders • Account statements • Holiday ordering • Tooltips 	<ul style="list-style-type: none"> • Search Bar updates • Add more relevant stats for dash • No max cheese log in an order • Refine bulk ordering

Figure E.3: Summary of User Feedback

that the majority of participants found the system easy to use. An important functionality that was missing was the ability to have users with different roles or permission levels. Overall, feedback was that the system would be an improvement over GGC’s current process, but that it could be further enhanced with some

modifications and additions.

Design Iteration

With this feedback in mind, the team began to iterate on the design developed in Phase One. First the process models were updated to accurately reflect how data flows through the system. These include slight alterations to the context and level zero diagrams, as

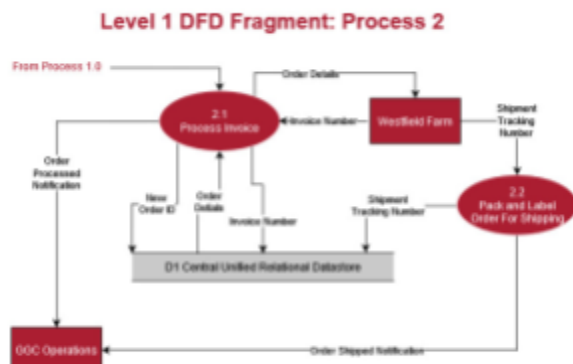


Figure E.4: Process 2 Level 0 DFD Fragment

well as the addition of a level one diagram that further breaks down the second process in the level zero diagram.

Also based on the feedback, an additional use case focused on accessibility was added to the design. Further, the logical ERD was reworked to more accurately represent the system and the adjustments made based on user feedback.

The team also updated the high-fidelity prototype to incorporate changes based on the feedback as a design reference. The team made changes to the dashboard, reduced the number of pages, and increased the clarity of

the distinction between the “Order” and

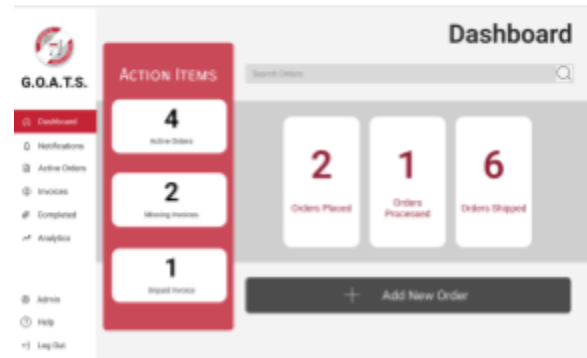


Figure E.5: Updated Mockup of New Dashboard

“Invoice” pages. Additionally, the team developed a design for how the farm users perspective of the application would look.

Development

The team developed the database, server, and front-end within one parent directory hosted in a Github repository named GGCPortal. The GGCPortal/tables folder contains scripts to create the database tables, a Python script to parse the current order spreadsheet, and the outputted CSV files from the script. The GGCPortal/app folder contains the code for the server, and GGCPortal/client contains the code for the front-end client. This simple but logical hierarchy will help future collaborators to more easily contribute to the project.

The team first created the data model in a local instance of MySQL in order to test the proposed relational model. After confirming the model was acceptable, the team

generated a script that would enable future developers to easily recreate the database in a new instance. In order to test the system with real data, and in the future migrate historical records into the new system, the team created a Python script that converts the current GGC order spreadsheet into separate CSV files for insertion into the database.

The team created an Express.js server running in a Node environment in order to create a backend to access the data on the frontend client. For each table in the database the team created an Object-Relational Map and accompanying controller. A data service was then mapped to each ORM's create, read, update, & delete (CRUD) functions for use on the front-end. This allows the frontend code to interact with the database as objects and make changes to the database with the data service.

The team started frontend development with refactoring of the components that the Phase I team created in order to make them scalable. The team also utilized the Redux library to create a frontend with centralized state management and better code organization. The centralized state management that Redux provides is

especially beneficial in providing users access to the same data but with different views based on permission levels. The centralized state also makes it clear how user actions affect the state of the application. In order to achieve the desired layout and styling, the team utilized vanilla

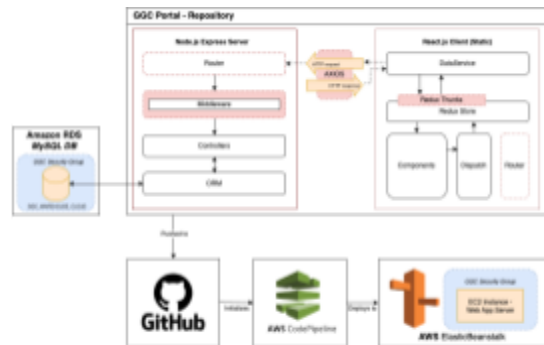


Figure E.6: Software Architecture and Code Pipeline

CSS in combination with the Bootstrap CSS framework.

The team also explored a design configuration for a cloud deployment of the system. The team continued working within the AWS environment, albeit with different services than the Phase One iteration. In a future implementation, the database would be hosted in Amazon Relational Database Services, the Express server and React front end would be hosted using Amazon Elastic Beanstalk.

Future Work

The team developed the functional prototype to meet the minimum system requirements, but do not recommend that the system is

ready for implementation. First, future developers should conduct further user testing to re-evaluate the most needed additional functionalities. Based on the work and research the team did in this iteration, these are the most important steps for future iterations to complete:

1. Refine database schema based on updates within GGC's business processes.
2. Refactor the Express.js API to better utilize the relational nature of the database.
3. Zapier integration with Formsite (automate entry of custom orders into database).
4. Iterate on delivery time and company communication management.
5. Encryption for login.
6. Update Node.js and dependencies and get solid test code coverage.

Additionally, the Westfield Farm role has not been fully implemented. While the role exists within the functional prototype, neither iteration of this project has conducted user-testing with anyone from the farm. This will be a crucial step in implementing the system into both organizations.

Gompei's Goat Cheese has always been a pillar of the WPI Business School, but its recent CEO has indicated they want to increase GGC's impact campus wide. One way in which it can do this is to incorporate other departments in the hiring process for new positions within GGC such as software engineers or data scientists. Such students may also contribute to the continued development of the system. In this way, GGC can expand and also provide a real-business learning experience for even more WPI students.

Table of Contents

Abstract	2
Acknowledgements	3
Executive Summary	4
Table of Contents	10
List of Tables	12
List of Figures	13
List of Appendices	15
1.0 Introduction	1
1.1 Gompei's Goat Cheese	1
1.2 Enterprise Resource Planning Systems	2
2.0 Background	4
2.1 Similarities Between GGC, Small Businesses & Startups	4
2.2 Review of Phase 1	5
Interviews & System Requirements	5
Design & Development	7
Next Steps	13
3.0 Methodology	15
Objective 1: Evaluate Existing Prototype	16
Objective 2: Iterate on Phase One Design	17
Objective 3: Develop a Functional Prototype	17
Objective 4: Create Support Documentation	18
4.0 User Testing	19
4.1 Formsite	19
4.2 Prototype Modifications	19
4.3 Study Protocol	20
4.4 Response Analysis	21
Qualitative Analysis	21
Quantitative Analysis	23
4.5 Feedback	24
4.6 System Improvement Suggestions from User Testing	25
4.7 Key Take-Aways	26
5.0 Iterating on the Phase One Design	27
5.1 Process Model Refinement	27
5.3 Additional Use Case	30
5.4 Database Schema Refinement	30

Initial Database Comparisons	30
Logical Entity Relationship Diagram and Data Dictionary	32
5.5 High-Fidelity Prototype Improvements	34
Westfield Farm Perspective	36
6.0 Development	37
6.1 Software Architecture	37
6.2 Back-End Development	40
Schema Generation and DB Configuration	40
Express.js Object Relational Mapping	41
6.3 Front-End Development	42
State Management with React Redux and Redux Toolkit	43
Layout and Styling	45
6.4 Cloud Deployment	45
6.5 Persisting Issues	47
6.6 Functional Prototype	48
7.0 Future Work	54
7.1 Next Steps	54
General Development	55
Westfield Farm Perspective	57
7.2 Final Thoughts	58
References	59
Appendices	61

List of Tables

Table 4.1 <i>Displays which interview questions are mapped to the six learning objectives</i>	22
Table 5.1 <i>Tooltip Use Case</i>	30
Table 6.1 <i>GGC Portal Technical Stack</i>	39
Table 6.2 <i>Tools and Technologies Used</i>	39
Table 7.1 <i>Aspects of the functional system that are complete, in progress, future implementation, and special features to be added</i>	56

List of Figures

Figure 1.1: <i>Context Diagram of Data Flows in as- is system</i>	2
Figure 2.1: <i>System Requirements Developed by Phase One Team</i>	6
Figure 2.2: <i>Use Cases Developed by Phase One Team</i>	7
Figure 2.3: <i>Phase One Context Diagram</i>	7
Figure 2.4: <i>Phase One Level 0 DFD</i>	9
Figure 2.5: <i>Phase One User Flow Diagram</i>	9
Figure 2.6: <i>Dashboard Mockup</i>	10
Figure 2.7: <i>Active Orders Mockup</i>	11
Figure 2.8: <i>Active Invoices Mockup</i>	11
Figure 2.9: <i>Order Example Mockup</i>	12
Figure 2.10: <i>Order and Invoice Lookup Mockup</i>	12
Figure 3.1: <i>System Prototyping Methodology and Project Objectives</i>	16
Figure 4.1: <i>Add New Order Page</i>	20
Figure 4.2: <i>A bar chart of instances of participant responses in reference to acceptability of the system</i>	23
Figure 4.3: <i>A bar chart of instances of participant improvement suggestions for the system</i>	23
Figure 4.4: <i>The figure shows where the G.O.A.T.S. prototype lands on the SUS based on the user data collected. The acceptability ranges and grade scale are provided for reference.</i>	24
Figure 4.5: <i>A Summary of user feedback highlighting confusing points, improvement suggestions, and additional features</i>	25
Figure 5.1: <i>Refined Context Diagram</i>	27
Figure 5.2: <i>Update Level 0 DFD</i>	28
Figure 5.3: <i>Process 2 Level 1 DFD Fragment</i>	29
Figure 5.4: <i>MySQL Workbench visual representation of the database</i>	32
Figure 5.5: <i>GGC ERP - Entity Relationship Diagram</i>	33
Figure 5.6: <i>GGC ERP - Entity Relationship diagram with Referential Integrity</i>	34
Figure 5.7: <i>Proposed Mockup of new dashboard</i>	35
Figure 5.8: <i>Figma Mockups for Orders and Invoices</i>	35
Figure 5.9: <i>Mockup of the farm facing side</i>	36
Figure 6.1: <i>Project Directory</i>	37
Figure 6.2: <i>Diagram of the software architecture and code pipeline</i>	38
Figure 6.3: <i>Tables Directory</i>	40
Figure 6.4: <i>Express Directory</i>	41
Figure 6.5: <i>Axios Data Services</i>	42
Figure 6.6: <i>React Client Directory</i>	42
Figure 6.7: <i>Visual example of React Redux data flow</i>	44
Figure 6.8: <i>Hex numbers of GGC colors</i>	45

Figure 6.9: <i>Software Version Deployed to Elastic Beanstalk Cloud Environment</i>	47
Figure 6.10: <i>Login Page</i>	48
Figure 6.11: <i>Dashboard Page</i>	49
Figure 6.12: <i>Orders Page</i>	49
Figure 6.13: <i>Order page</i>	50
Figure 6.14: <i>Editing Order Page</i>	50
Figure 6.15: <i>Active Orders Tab</i>	51
Figure 6.16: <i>New Order</i>	51
Figure 6.17: <i>Search Orders Page</i>	52
Figure 6.18: <i>Searching in Search Bar</i>	52
Figure 6.19: <i>Delete Order</i>	53
Figure 6.20: <i>Navigation Bars by Role from Left to Right (Logout page, GGC role, Farm role, and Admin role)</i>	53

List of Appendices

Appendix A: User Testing Consent	62
Appendix B: User Testing Protocol	65
Appendix C: User Testing Data Analysis Sheet	70
Appendix D: Functional Prototype Images	76
Appendix E: Context Diagram	77
Appendix F: Data Flow Diagrams	78
Appendix G: Entity Relationship Diagrams	80
Appendix H: Use Cases	82
Appendix I: Data Dictionary	89
Appendix J: Package.json	96
Appendix K: MySQL Script	101
Appendix L: Python Script	105
Appendix M: Support Documentation	112

1.0 Introduction

1.1 Gompei's Goat Cheese

Gompei's Goat Cheese (GGC) is a non-profit, student-run goat cheese business at Worcester Polytechnic Institute (WPI). The company itself started as a Major Qualifying Project (MQP) for the WPI Business School in 2012, in which students created a brand to sell local goat cheese. The MQP is the culmination of WPI's project based curriculum, similar to a capstone project, giving students the opportunity to work as part of a team to solve a real world problem. Often sponsored by a company or other external organization, the MQP also allows students to demonstrate their major specific knowledge. (Major Qualifying Project, n.d.). The founders of GGC established a partnership with Westfield Farms in Hubbardston, Massachusetts, where the cheese would be produced and shipped to customers. GGC donates all profits to the WPI Global Scholarship Program, supporting student Interactive Qualifying Projects (IQP) scholarships. The IQP is a project before the MQP that gives every WPI student the experience of working in interdisciplinary teams with students not in their major, to tackle an issue that relates science, engineering, and technology to society. Sustainability is a common theme among IQPs, many of which address problems related to energy, environment, sustainable development, education, cultural preservation, and technology policy (Interactive Qualifying Project, n.d.). These IQPs can be done locally or through one of WPI's project centers around the globe, allowing for students to make a difference in communities across the globe.

Ultimately, GGC's mission is "to support WPI global scholarships, learn entrepreneurial skills, and interact with the WPI and Worcester community by selling award-winning goat cheese so that we can be a part of something meaningful that impacts our community" (Gompei's, 2022). Unfortunately, GGC's current operations are not well kept. All communications are relayed through Google Sheets or email, and the typical flow of events is as follows:

1. Collect order details and payment.
2. Forward order information to the farm.
3. Farm confirms order through email.

4. Farm Generates an invoice to send to GGC to request payment for the order.
5. Farm ships the order to the customer's home

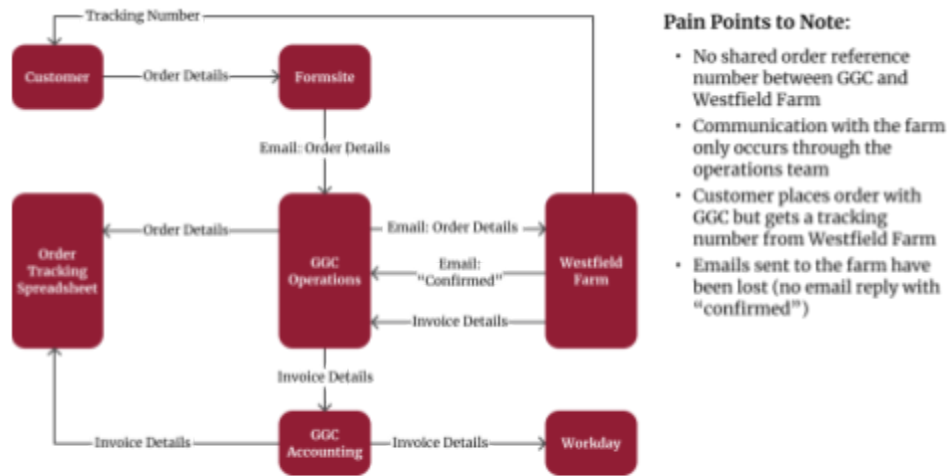


Figure 1.1: Context Diagram of Data Flows in as-is system

Figure 1.1, created by last year's team, illustrates this flow of information as well as detailing some of the issues with the current system. The existing operations structure creates harmful inconsistencies between GGC's records versus the farm's records. These issues impact the fulfillment of orders, further limiting the growth of GGC. Thus, it was clear that the company "was in need of a new order and invoice management system, along with improved communication with the farm" (Guerrette & Mohn, 2022).

1.2 Enterprise Resource Planning Systems

Enterprise resource planning (ERP) systems are used to organize and direct the processes within a company such as accounting, finance, production, inventory, order entry, and logistic operations. An ERP can integrate all departments into one information system or into a set of integrated systems that organizations use to make business decisions (Baltzan, 2020). A cloud-based ERP system performs the same functions as a traditional ERP system but the hardware and software is managed by a third-party vendor. One of the main aspects of an ERP system is that information is shared across the organization via a centralized database. This allows decision makers to have more information about the whole business to utilize. Cloud-based ERP systems also have the added benefit of being accessible from mobile

computing devices. Development and implementation are two critical phases for any business deploying a new information system. An ERP system consists of core and extended components (Baltzan, 2020). Core components are focused on internal operations such as accounting or finance, whereas extended components are “add-ons”, such as customer relationship management, that meet organizational needs not covered by the core components. The implementation of a new ERP system requires careful planning as well as support from key stakeholders. In this project, key stakeholders include both members of GGC as well as operators of the partner farm.

2.0 Background

2.1 Similarities Between GGC, Small Businesses & Startups

The Small Business Administration (SBA) states that a small business is defined as “an independent business having fewer than 500 employees.” Yet with over 32 million small businesses all over the United States, these small businesses account for 99.9% of all U.S. employer firms (Small Business Administration, 2019). Specifically in Massachusetts, there are 718,467 small businesses that make up 99.5% of business within the state, making small businesses the backbone of their local economies (Small Business Administration, 2022). When spending money at a small business, a lot of the money spent goes right back into the local community. The Better Business Bureau stated that “if [a person] spends \$100 at a local business, roughly \$68 stays within [the] local economy” whereas \$100 spent at a non-local business would only retain \$43 within the local economy (Better Business Bureau, 2019). Among these businesses, Gompei’s Goat Cheese exemplifies what a small business is like since everything is supplied locally, and the employees are members of the WPI community. However, GGC is also much like most startup businesses because of the company’s nature as a student-run business.

Most startup companies have a higher employee turnover rate, or attrition rate, on average than the business industry as a whole. The attrition rate for startup companies is 25%, which is roughly double the overall average attrition rate of 13%. This means that employees only hold their position for an average of 2 years at a startup company (Sharma, 2022). Since the GGC workforce is solely made up of students, the attrition rate at the company is very high because employees often graduate from WPI then leave the company to move on with their careers. This means as a student-run business, Gompei’s Goat Cheese will always stay in a startup, small business state since the company is meant as a learning environment for running a real business. To cater to this style of business, the ERP system must be designed to be intuitive to use and simple to learn as to avoid confusion to new members of the GGC team. This also allows the business to spend less time training new members and more time focusing on

operating and improving the business. Designing the system to be simple also reduces employee mistakes as more redundant operations will be automated.

2.2 Review of Phase 1

In the previous academic year, an MQP proposal was submitted by Chris Guerrette and Natalie Mohn to design a cloud based ERP system that would address many of the pain points with GGC's current operations. In order to have a successful Phase Two of the project, the team first needed to review what the Phase One team had accomplished. The following subsections examine the work done in Phase One of the Project.

Interviews & System Requirements

After performing their own background research, last year's team started by conducting a series of interviews with different GCC members, as well as the owner of the partner farm. Based on these interviews, they constructed a list of benefits and challenges with the current system, as well as other relevant information:

- Interaction with the current GGC operations system ranges from a few times a semester to nearly every day.
- Team members prefer to use their computer, rather than their phone, when using the current operations system.
- Westfield Farm and GGC use different order numbering systems.
- Inputting information into the current system takes too long, and there's a lot of redundancy in the work that the operations team does.
- Westfield Farm handles Gompei's Goat Cheese order's differently than its normal orders.

From these key findings, as well as their own experience as GGC members, the Phase One team developed a set of system requirements. Figure 2.1 on the following page contains all the requirements. Some key requirements include: both GGC members and partner farm employees can access the system simultaneously; users can view, create, update, and delete orders based on

their accounts permission level; the system is both intuitive and accessible; and users can securely and quickly access the data stored in the system.

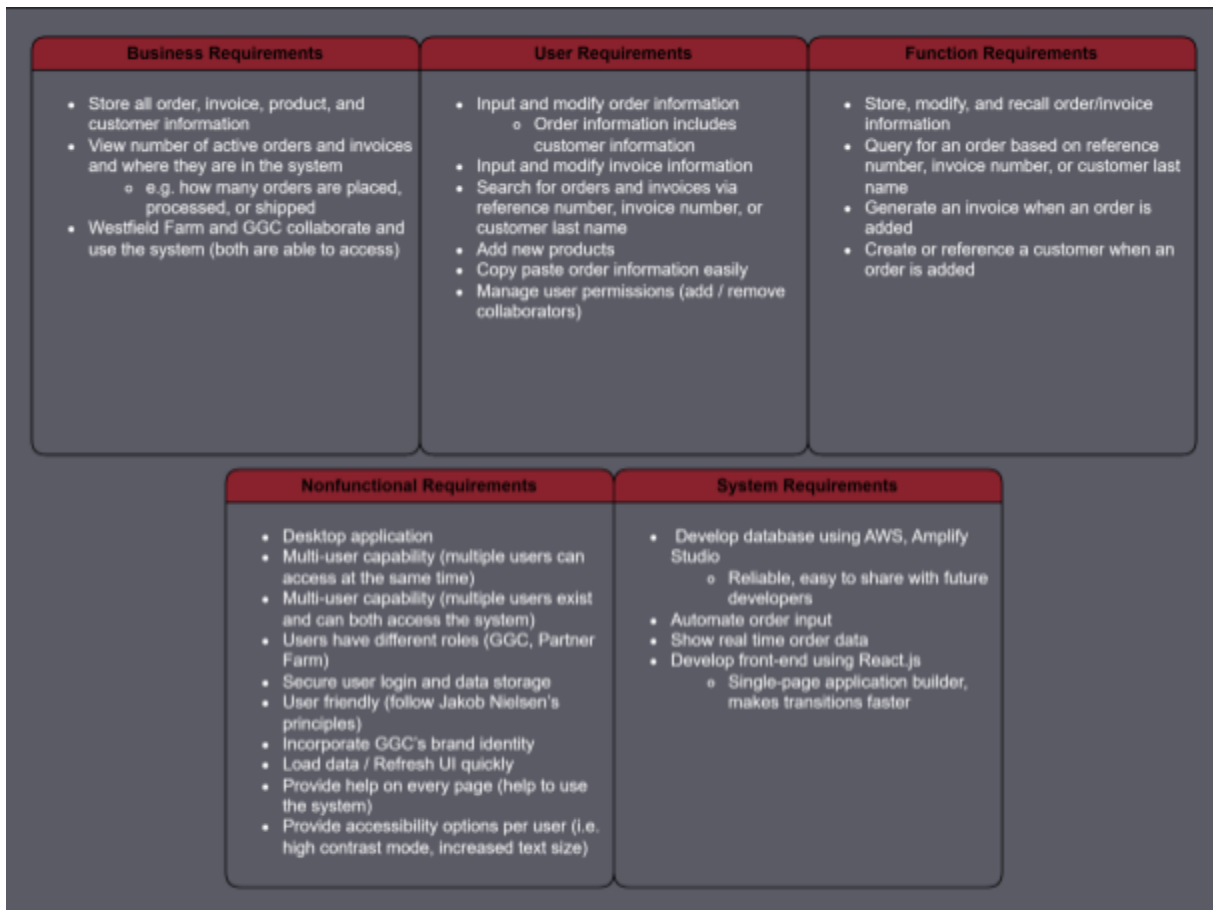


Figure 2.1: System Requirements Developed by Phase One Team

Next, using the use cases they developed as guidelines, the Phase One Team created process models, also known as data flow diagrams (DFD), for the planned system. Process modeling depicts business processes and how data flows between them. The team began with the context diagram, the top level of any business process model. The context diagram depicts the entire system within its surrounding environment. Figure 2.2 shows the system, Gompei's Operations & Accounting Tracking System (G.O.A.T.S), and the data flowing between it and external entities such as the partner firm of GGC Operations.

The Phase One team then constructed a level zero DFD of the planned system. The level zero DFD provides more detail than the context diagram, breaking the system down into its high level processes. The four high level processes are:

1. Add New Order
2. Process & Ship Order
3. Send Invoice
4. Process Invoice

Additionally, information is stored in three separate data stores.

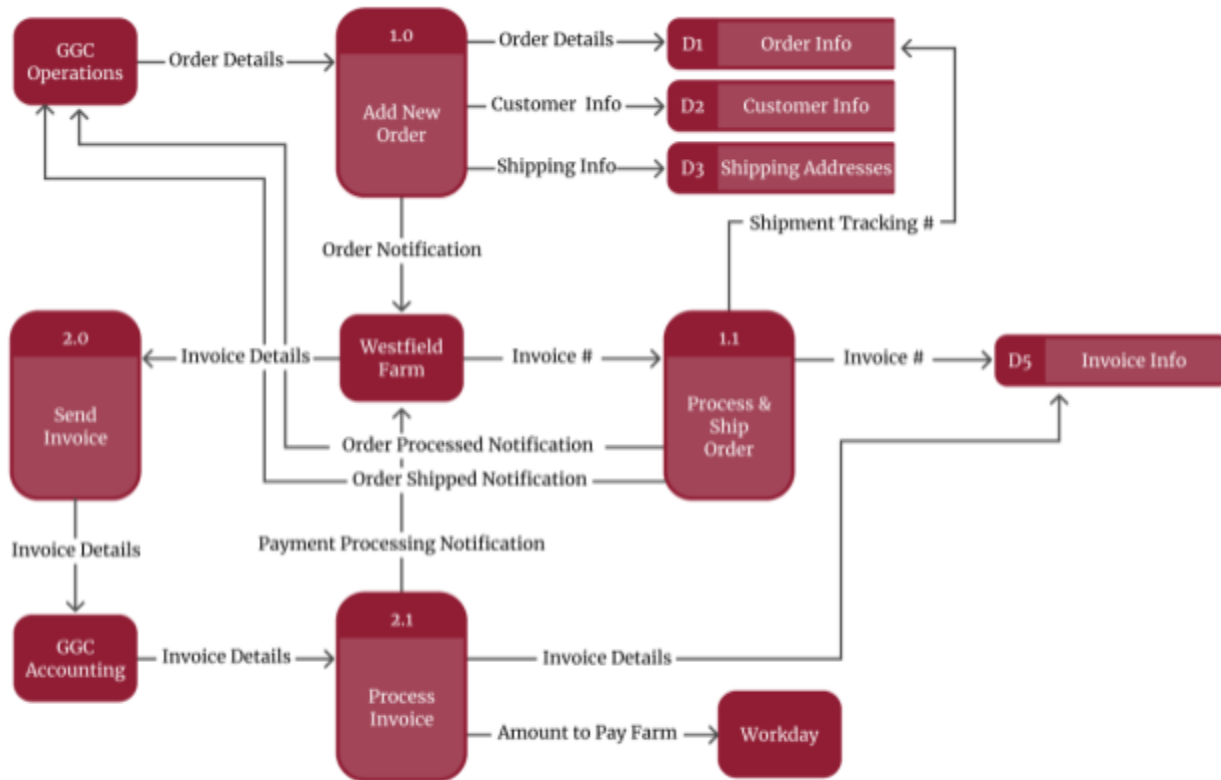


Figure 2.4: Phase One Level 0 DFD

After completing the process models, the Phase One team designed a user flow diagram depicting some of the common pathways the user, a member of the GCC operations team, might take through the system. Figure 2.3 depicts a user logging in and first viewing the dashboard. From there, the user can go to invoices, orders, or the order lookup page. From each page, there are a series of actions that the user can take, reversing to a previous step if necessary.

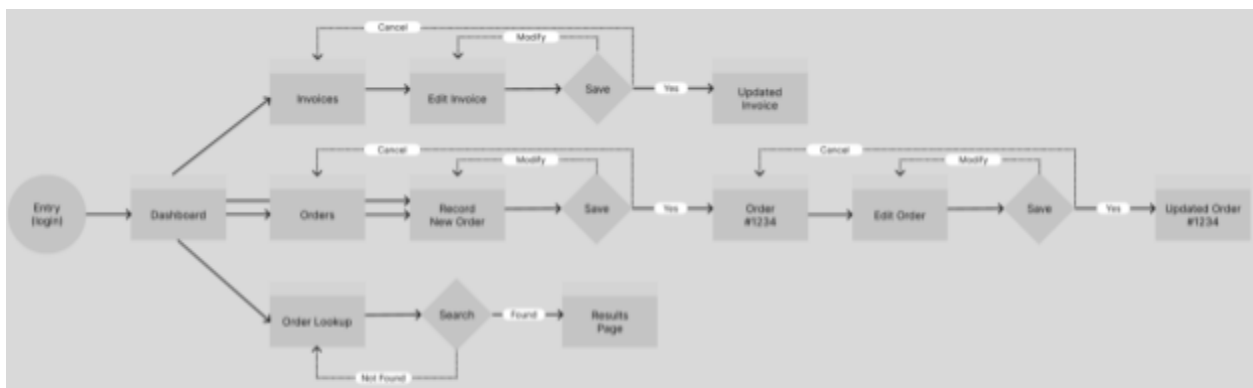


Figure 2.5: Phase One User Flow Diagram

With the use cases, process models, and user flow diagram as references, the Phase One team developed a set of system mockups. The team chose to develop them on Figma because it had the tools to turn high-quality mockups into a high-fidelity prototype. Figma is a tool where one can make high quality mockups of an application. The user lands on the dashboard page where they can view quick stats on the current orders, or add an order. The user then continues to the active orders page where they can view orders that have been placed, processed, and shipped and invoices are in a similar format. They then move to order details where they can view all of the information fields for the customer and the order. If they need, they may also edit the order by clicking the edit button which turns the fields editable. A user may also search for orders by invoice number, reference number, or name. Images of the proposed prototype can be seen in the figures below.

Figma Screenshots:

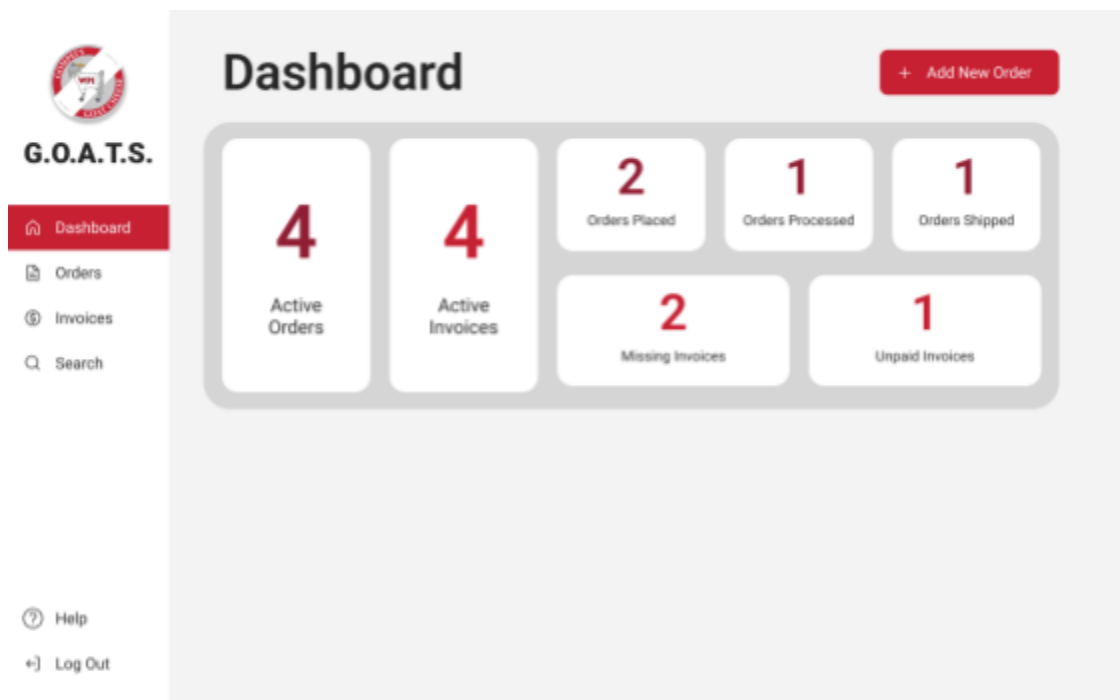


Figure 2.6: Dashboard Mockup

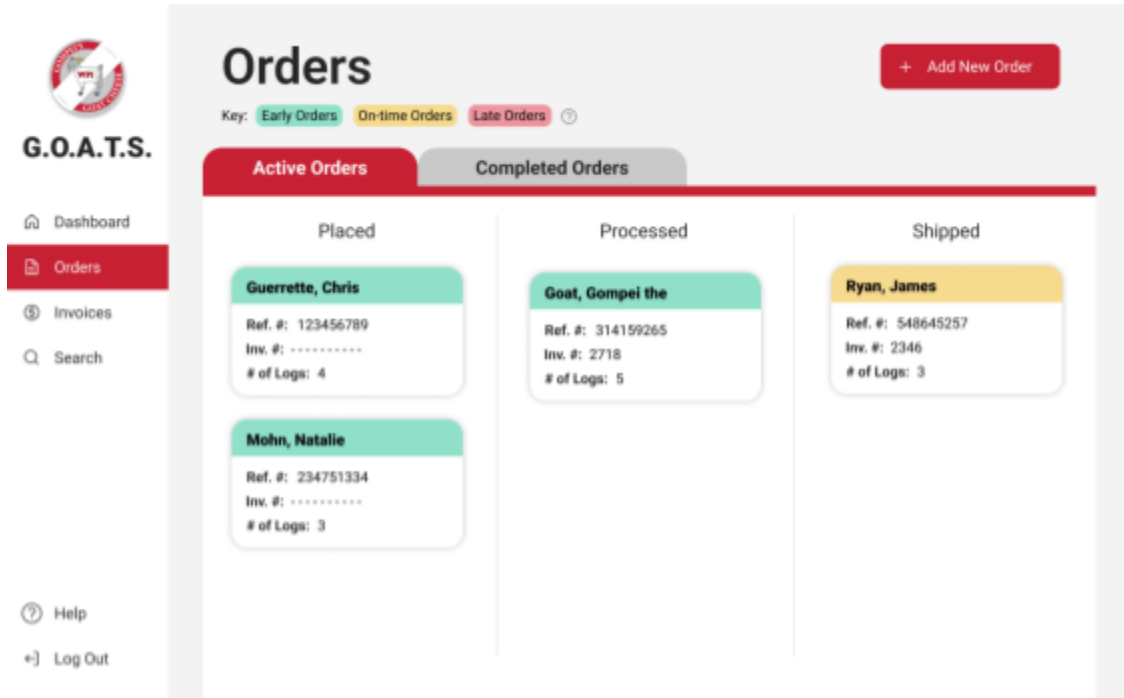


Figure 2.7: Active Orders Mockup

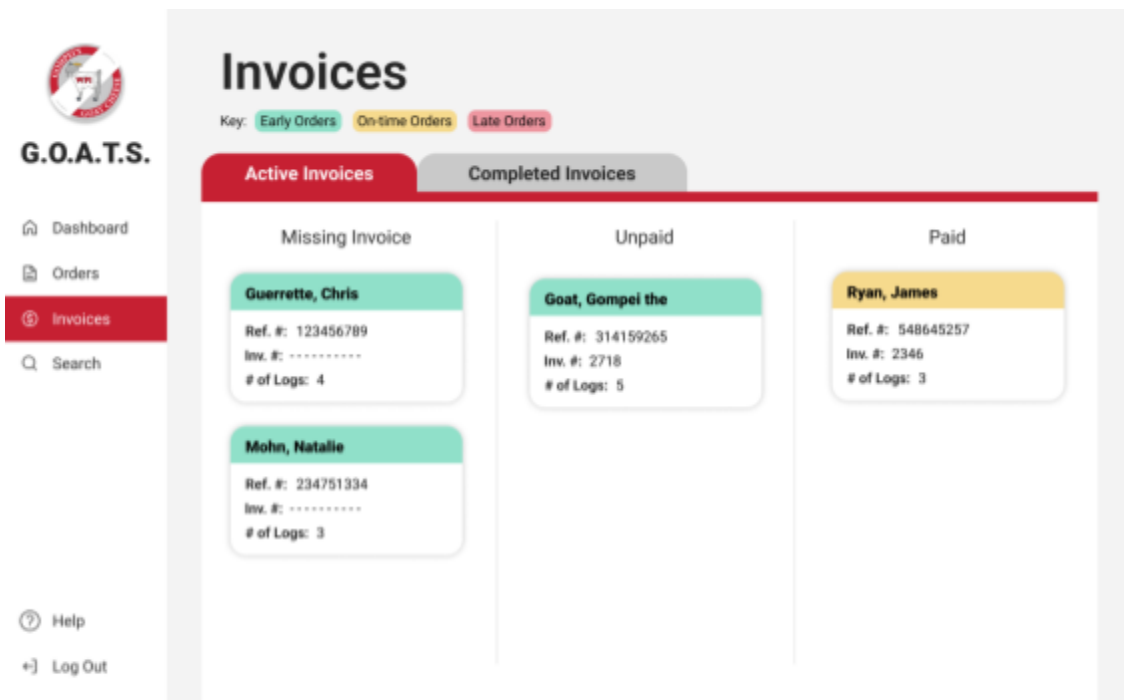



Figure 2.8: Active Invoices Mockup



G.O.A.T.S.

- Dashboard
- Orders**
- Invoices
- Search
- Help
- Log Out

Orders

Key: Early Orders On-time Orders Late Orders

Active Orders | Completed Orders | **Guerrette, Chris**

Order Status: Placed

Save | Cancel

Reference # 123456789

Invoice # [Redacted]

Date Placed 04/07/2022

Recipient Natalie Mohn

Address
100 Institute Road
WPI Box #5886
Worcester, MA 01609

Tracking # [Redacted]

Gift Message Happy Graduation!

Name Chris Guerrette

Email cguerrette@wpi.edu

Phone 2037256633

Flavor Information

Name	SKU	Quantity	
Plain	JPL5	2	X
Cranberry Orange	JCRA	1	X
Chocolate	JCH05	1	X


At a Glance:

Total Number of Logs: 4

Total Number of Flavors: 3

...Add New Flavor + Add

Figure 2.9: Order Example Mockup



G.O.A.T.S.

- Dashboard
- Orders
- Invoices
- Search**
- Help
- Log Out

Order & Invoice Lookup

LAST NAME Search

DATE	LAST NAME, FIRST NAME	REFERENCE #	INVOICE #	ORDER STATUS
02/01/22	Januszewski, Beverly	6006594	6248	Completed
06/03/21	Januszewski, Lyndsey	5283436	3568	Completed
02/08/21	Januszewski, Zoe	4888565	1256	Completed

Figure 2.10: Order and Invoice Lookup Mockup

The team began their development by starting the creation of a database using Amazon Web Services (AWS). They decided to use AWS to host the database and code because of how easy it would be for future teams to access and make changes to the system. Within AWS, Amplify Studio is used to hold all the project's contents (e.g. code and data) all in one place. Amplify Studio is also what makes it easy to add new users to the project. For the database, the team in Phase One created a key value store (NoSQL) database using DynamoDB. At the time, this was the only type of database in AWS that could interface directly into Amplify Studio, however it does offer its benefits; Key value store databases can read and write data very quickly and are very flexible to use, which benefited the Phase One team's original plan to automate order processes and use real-time order data within the application. To develop the user interface, the team used React.js to build the front end of the system. React is a highly powerful Javascript library for building user interfaces, and is considered one of the leading tools in web development. Yet with the intention of building an ERP system, our Phase Two team must give the React code a more maintainable structure. To glue the system together, the original team proposed using a GraphQL API to connect the front-end site to the database. The reasoning behind this decision is that when building a GraphQL schema, Amplify Studio creates a DynamoDB table automatically for any object with the *@model* directive tag, which makes it very easy to create the database. It also automatically creates a fully functioning API complete with auto generated resolvers for basic interaction with the database.

Next Steps

The Phase One team concluded their report by outlining what the next steps in the project might look like if it were to be continued. Their first recommendation was to gather user feedback on the prototype that they had developed. They then outlined use cases from which their design had been developed in the functional prototype they began creating. Following that, the Phase One team proposed one way in which a future team might be able to further reduce the redundancy of the system: a system that could process order data from the files output by Formsite. Although the Phase One team's prototyping focused mainly on the GGC side of the system, they also considered ways in which the farm's perspective of the system would have to differ due to differing requirements. Certain information, like customer info, should not be accessible to the farm. Finally, the Phase One Team provided several recommendations for

improving the accessibility of the system such as adding a high contrast mode or an option to increase the text size.

3.0 Methodology

The goal of this project was to continue development of a cloud-based enterprise resource planning system to improve the efficiency of GGC's operations. We accomplished this goal by fulfilling the following objectives:

1. Evaluate the existing prototype by conducting user testing sessions with members of GGC.
2. Iterate on Phase One design by applying feedback from user testing.
3. Develop a functional prototype of the ERP system.
4. Create support documentation for future development.

In order to develop a high quality system, it is important to follow a methodology that implements the four phases of the system development life cycle (SDLC): planning, analysis, design, and implementation. Each of these phases have steps that produce deliverables and help to gradually refine the system (Dennis et al., 2019). Since the Phase One Team chose to follow a system prototyping methodology, it made sense for the project to continue to use the same methodology. This methodology, “performs the analysis, design, and implementation phases concurrently in order to quickly develop a simplified version of the proposed system and give it to the users for evaluation and feedback” (Dennis et al., 2019). Figure 3.1 shows how each objective fits into each of the phases of the SDLC.

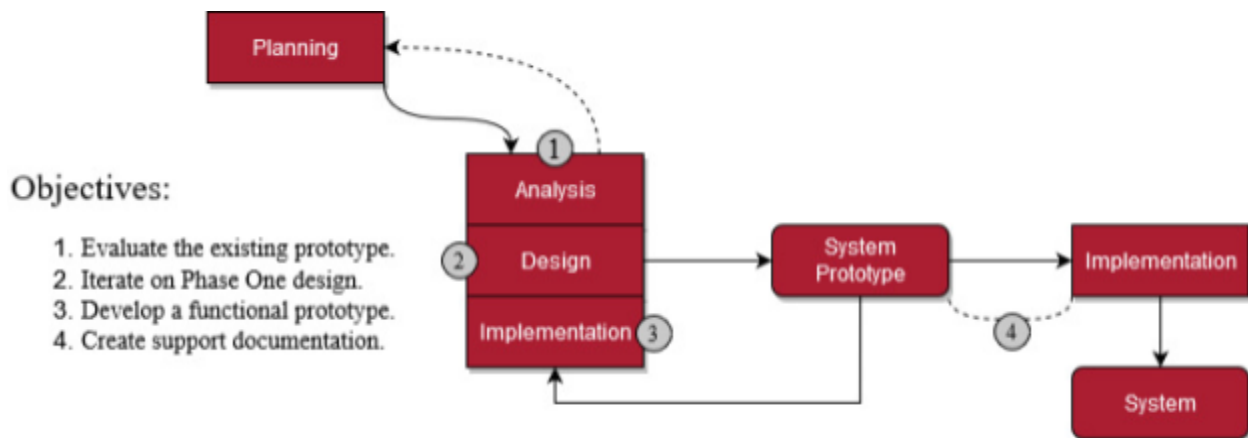


Figure 3.1: System Prototyping Methodology and Project Objectives

Objective 1: Evaluate Existing Prototype

Our team's first objective was to evaluate the prototype designed by the Phase One team. To accomplish this the team conducted user testing with current GGC members and analyzed the system's usability as well as how well it met the needs of GGC.

The team chose to use the think-aloud method, "... the single most valuable usability engineering method" according to Jakob Nielsen, cofounder of the usability consulting company Nielsen Norman Group (Nielsen, 2012). The think-aloud method entails giving users a set of tasks to complete on the system being tested, and asking them to speak aloud their thoughts as they complete each task. The think-aloud method has many advantages that made it an ideal choice for achieving the team's objective. Some of the most important are that it enforces user-centered design, allows the team to directly observe the user's reactions, and that it is easy to implement.

We also chose to use a System Usability Scale (SUS) survey to evaluate the prototype using quantitative analysis. SUS is a quick and simple non-diagnostic tool to determine if a system is usable or unusable. Although the scale is 0-100, the SUS score is not synonymous with letter grading scores. If a system scores above a 70 it is considered an acceptable system.

Evaluating G.O.A.T.S. with SUS is a valid way to assess a system on a small sample size, and is a good way to establish a baseline of the usability (U.S. General Services Administration, 2023).

During each session with a test user, one member of the team would take notes whenever a participant had trouble performing a task, noting the task and the problem encountered. The team then generated a list of confusing points, general improvements, and ideas for new features.

Objective 2: Iterate on Phase One Design

The second objective of the team was to iterate on the design from the Phase One team by incorporating the feedback from the user testing the team conducted in the first objective. The team first updated the entity relationship diagram, list of use cases and DFDs. The team then incorporated the feedback from the testing into the high fidelity prototype to use as a reference when developing the system. Finally, the planned software architecture of the system was overhauled in order to create a centralized and predictable system that would be easy to expand in the future.

Objective 3: Develop a Functional Prototype

After iterating on the design of the system, the third objective was to develop the system into a functional prototype incorporating the feedback from user testing. The team went through the following steps to complete this objective:

1. Create a software architecture with sensible organization
2. Develop a backend:
 - a. Design a relational database schema and configuration
 - b. Create a an Express.js server to interface with the database
3. Develop a front-end portal using the React.js and Redux frameworks
4. Connect the portal with the backend using the Axios HTTP client
5. Design a cloud implementation

The team developed the code repository on Github so that it can be easily shared with future collaborators. The repository contains all the code necessary for cloud deployment. The team chose to revert to a relational database over a key-value store database because it is better suited to address the issues GGC currently faces such as duplicate records. The MySQL database was first created in a local environment in order to test the schema. Next, an Expressjs application was developed to interface with the database. The team created an Object-Relational Mapper (ORM) for each table in the database. Along with a controller for each table, this allows interaction with the database through JS objects instead of SQL queries. The application runs in a Node.js environment and also handles routes requested by the front end. The team developed the front-end using React.js and Redux for state management. React.js is one of the most popular front-end JS frameworks and allows for the creation of reusable UI components. Redux is a popular addition to React.js and is used for managing state within a React application. The front-end connects to the backend API using the Axios library which manages HTTP requests from the client and the responses from the server. Finally, the cloud implementation was designed using Amazon Web Services (AWS). The team created the cloud implementation design utilizing the following services: Amazon Relational Database Services (RDS), AWS CodePipeline, and AWS Elastic Beanstalk, When changes are made to the Github repository, CodePipeline deploys the changes to Elastic Beanstalk which is what hosts the frontend a Node application, and is connected within a security group to the database in RDS.

Objective 4: Create Support Documentation

In order to encourage its continuation, the fourth objective of this project was to create support documentation so that future collaborators will be able to quickly start contributing to the project. In order to complete this objective the team created a support document with important login credentials and links to helpful resources. Additionally, the team created a document in the code base, README.md, that includes very technical information regarding the next steps to take to continue developing the functional prototype.

4.0 User Testing

4.1 Formsite

The Phase One team of the GGC MQP gave recommendations to reduce redundancy in employee operations by automating order entry from Formsite instead of having to manually input order data into the database. However, the Phase Two team decided not to follow this recommendation after gaining more information from the user-testing sessions we conducted. It was found that customers tend to make mistakes when ordering, so the GGC usually waits one to three days before sending orders to the farm so there's a grace period to change the order. The team decided not to alter this business process because an alternative option couldn't be found in time. Instead, the group investigated the pitfalls of the current customer order form that leads customers to make mistakes. To solve this issue, the customer order form was updated to increase accessibility and reduce customer mistakes. Although a change was submitted, the site has not been updated since the conclusion of this project.

4.2 Prototype Modifications

Although the prototype was mostly completed by the Phase One team, There were some minor changes that needed to be added in order to complete user testing and meet all use cases. These changes included non-visual aspects such as prototype mapping, or linking buttons to certain pages to ensure the participants were able to complete the tasks in the user study we conducted. This allows the participants to go through the system as if it were a functional website, to get them as close to the functional prototype as possible to identify any changes we would need to reduce the use of resources. The developed prototype was only able to modify an existing order, not create a new order, so the team added that capability, while still following the same layout as the others. Additionally, the team expanded more with the search page to highlight the different options you can search by.

G.O.A.T.S.

Dashboard
Orders
 Invoices
 Search
 Help
 Log Out

Orders

Key: Early Orders On-time Orders Late Orders

Active Orders | Completed Orders | **New Order**

Save Cancel

Reference # 333429210
 Invoice #
 Date Placed 04/02/2022

Recipient: Victoria Buycck
 Address: 100 Institute Road, WPI Box #6886, Worcester, MA 01609
 Tracking #
 Gift Message

Name: Victoria Buycck
 Email: vebuycck@wpi.edu
 Phone: 5088881234

Flavor Information

Name	Quantity
Plain	2
Herb Garlic	1
Hickory Smoked	0
Blueberry Lemon	0
Cranberry Orange	0
Firey Fig	2
Chives	2
Pink Peppercorn	0
Calabrine	0
Chocolate	2

At a Glance:
 Total Number of Logs: 9
 Total Number of Flavors: 5

Figure 4.1 Add New Order Page

4.3 Study Protocol

The team developed the prototype to meet all user requirements for the system. A user study was conducted with four participants from GGC executives to test the prototype system and gain an understanding of its strengths and weaknesses. All of the interviewees followed the same Prototype Testing Protocol where they were asked to complete a number of tasks that covered the major use cases of this system. This protocol can be found in Appendix B. We performed a think aloud followed by an interview as well as a System Usability Scale (SUS) survey, then recorded all of their responses to each question to identify any common themes

across responses. This method best allows the team to understand what the user is thinking during the process and enables participants to speak freely about their thoughts of the system. Additionally, the team marked whether or not the participant successfully completed each task to further develop the analysis.

4.4 Response Analysis

After gathering all participant responses, the team transferred the raw data into an Excel spreadsheet to maintain organization. The responses were then analyzed with respect to qualitative and qualitative analysis based on the data collected. More information about this sheet can be found in Appendix C.

Qualitative Analysis

To analyze the feedback qualitatively, the team recorded each participant's answers, and identified any common themes amongst them. Each of the questions on the interviews were intended to touch upon six different target points that the team was interested in learning about for this user study. The team wanted to gather first impressions, identify if the system is perceived to be useful, identify any comments about the design or layout, highlight any improvements that needed to be made, and explore what is already acceptable in the system. Based on these learning objectives, the team developed appropriate questions to ask during the study. These question responses were then matched up against the target points to identify the participants' subjective thoughts about the system.

Interview Questions

Q1: What stands out most to you?

Q2: What did you LIKE about the new system?

Q3: What did you DISLIKE about the new system?

Q4: How would you describe the navigation of the system?

Q5: What are your thoughts about the design of the system?

Q6: Do you think the information was displayed in an effective way that's easily readable?
(why/why not)

Q7: What do you think about the “dashboard” feature?

Q8: Which system do you prefer? Old (email, google sheets) vs New (cloud-based ERP)

Q9: Are there any features you would like to see implemented?

Q10: Are there any other suggestions for improvement?

Learning Objectives	Questions
Impressions: What is the main takeaway from this system?	Q1, Q2, Q3, Q8
Useful: Does the system meet all of the user's needs?	Q3, Q7, Q8
Design: Is the design clear and effective or does it distract the user?	Q5
Layout: Is the flow of the system and displaying of information intuitive for the user?	Q4, Q6, Q7
Improvements: What additions or features can be added to the system?	Q3, Q9, Q10
Acceptable: What does the system already do well?	Q2

Table 4.1: Displays which interview questions are mapped to the six learning objectives

The responses display that what is most acceptable is the ease of use of the system. Additionally, participants stated the proposed system provides less error than the current, and that it is convenient. Conversely, all participants made suggestions for improvement of the system. The most improvement suggestion that was given was in aspects of accounting such as account statements and invoice clarification.



Figure 4.2: A bar chart of instances of participant responses in reference to acceptability of the system



Figure 4.3: A bar chart of instances of participant improvement suggestions for the system

Quantitative Analysis

To analyze the feedback quantitatively, we used a System Usability Scale to identify if the system is usable. The average SUS score of this group was a 95, which, according to the SUS, is in the “best imaginable” range in terms of usability. We used a SUS scoring template to properly calculate the scores while minimizing human error. It is important to note that this score does not give any insight on the functionality of the system, it is merely a tool used to detect the perceived usability of the system by the user (U.S. General Services Administration, 2023).

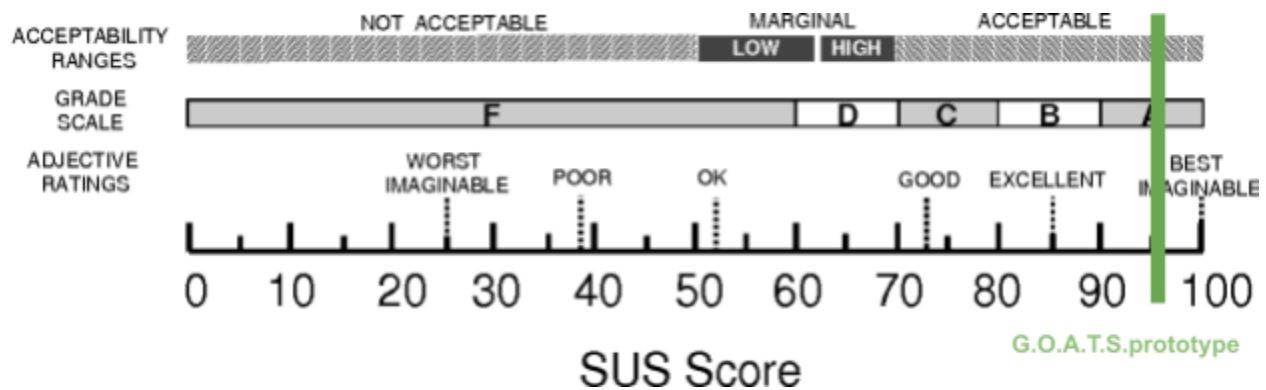


Figure 4.4: The figure shows where the G.O.A.T.S. prototype lands on the SUS based on the user data collected. The acceptability ranges and grade scale are provided for reference.

4.5 Feedback

All participants stated that the system is easy to use, however, they all gave suggestions for improvement as well. For each prototype system test, all four of the participants’ first impressions were about the system UI and ease of use. The participants all commented that the system UI is very clean and professional, as well as the system itself being very easy to use and intuitive. When asked for improvements, every participant gave a suggestion for improvements and additions to be made. There was some confusion in some participants between the ‘invoices’ and ‘orders’ tabs, so some suggested bringing more clarity between these tabs. Other recommendations were to add different information in the order cards, add permissions for different roles in the company, decrease the amount of information on a given page, and add an ability for in person orders.

The Chief Operations Officer (COO) of Gompei’s Goat Cheese found the system very intuitive, and only had a minor concern about the interface. The COO found the interface clean, organized and convenient. Their only concern was that they didn’t know the information had to be saved, and so the save function should become more prominent. The COO said they would prefer this system over the current system as long as it can be integrated with the farm well. The feedback of the COO is weighed heavier than other participants’ feedback because they use the current system the most and have the most well-informed insight for a new system.

The COO gave valuable ideas and suggested additions that would help both the GGC and the supplier farm. The suggestion that would help GGC would be to add a section for in-person orders where pickup details can be inputted as well as a section for sticker label tracking. One problem currently is that there can be miscommunications between the farm owner and the student-run company, so a way to help that is to give the farm owner a role to view account statements, invoice numbers, and order numbers. This allows for the farm owner to see what orders haven't been paid for yet directly. Another addition that was suggested was to have a function for the farm owner to notify GGC missing payments, and for the company to notify the farm owner of missing invoices. This would make communication easier between GGC and the farm. The current system also has limits on goat cheese quantity that can be ordered, so it would be helpful to remove that and allow any amount of cheese logs to be ordered.

4.6 System Improvement Suggestions from User Testing

Confusing Points	Improvement Suggestions
<ul style="list-style-type: none"> • Difference between orders and invoices • Button placement • Too many fields 	<ul style="list-style-type: none"> • More info on order card <ul style="list-style-type: none"> ◦ Date and last edited • Permissions for roles <ul style="list-style-type: none"> ◦ Finance ◦ Accounting ◦ View Only ◦ Farm Owner ◦ Data Person ◦ Admin • Search Bar updates • Add more relevant stats for dash • No max cheese log in an order • Refine bulk ordering
Additional Features	
<ul style="list-style-type: none"> • Data visualization • Notification center • Sticker label tracking • In person orders • Account statements • Holiday ordering • Tooltips 	

Figure 4.5: A Summary of user feedback highlighting confusing points, improvement suggestions, and additional features

4.7 Key Take-Aways

Although the system is clean, simple, and easy to use, there are still many improvements that need to be made with the main functionality and usability of the system. Having different roles for various GGC positions as well as a role for the farm is an essential part of the system that was overlooked. The differences between invoices and orders also needed to be more defined to avoid confusion and optimize functionality and processes. The COO made minor layout and UI suggestions to the design that would aid in the fluidity and intuitiveness of the system. With this feedback, The design process starts again and the team brainstormed ideas to solve these challenges.

5.0 Iterating on the Phase One Design

5.1 Process Model Refinement

After getting user feedback, the next step in the system prototyping methodology is to redesign. One element of the design that the team refined was the process models. These models were updated to more accurately reflect how the data would flow through the system. The

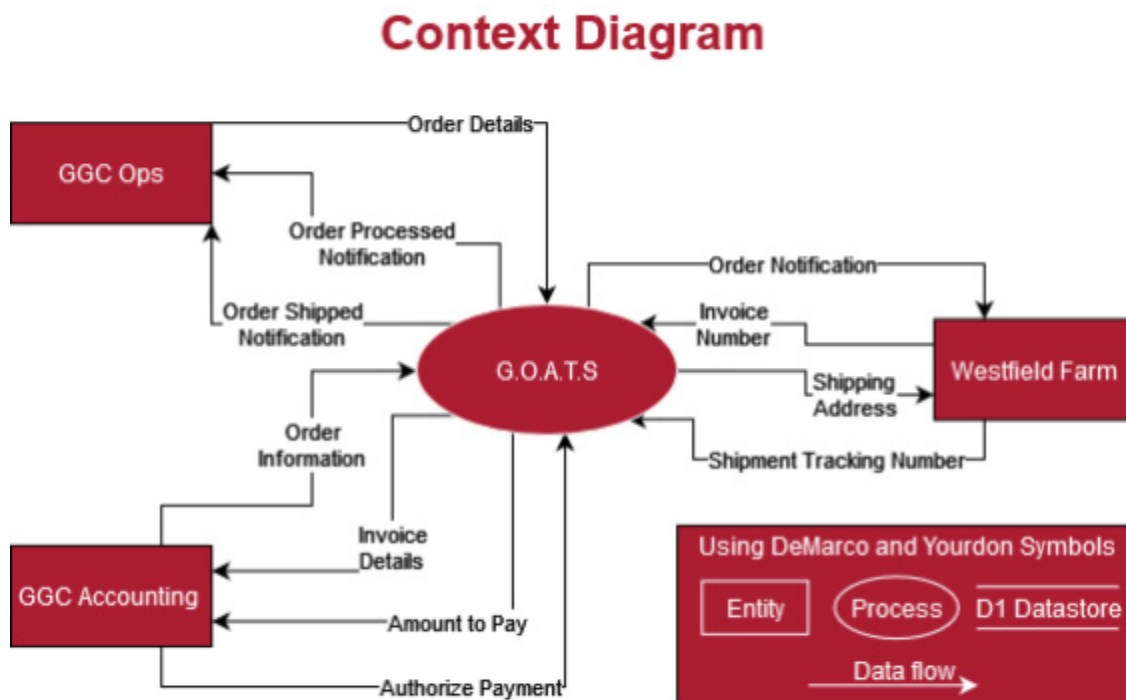


Figure 5.1: Refined Context Diagram

context diagram in Figure 5.1 shows the overall system and how data flows in and out of it to three external entities. Although some of the data may flow from an external entity to another one, such as a customer, this interaction is not part of the system and is not included in the diagram.

Following the context diagram, a level zero DFD shows all the high level processes of a system as well as including data stores. The level zero diagram shows how the processes are related to each other and to the data that is stored. The second iteration of the level zero diagram

retains the same number of high level processes as the first, although some have been modified. The first process of creating a new order in the system remains the same with little change. The second process, where the farm processes the order, now also contains what was the third process (the farm sending an invoice to GGC) in the first iteration. The third and fourth processes in the second iteration are GGC Accounting processing and paying invoices respectively.

Level 0 Data Flow Diagram

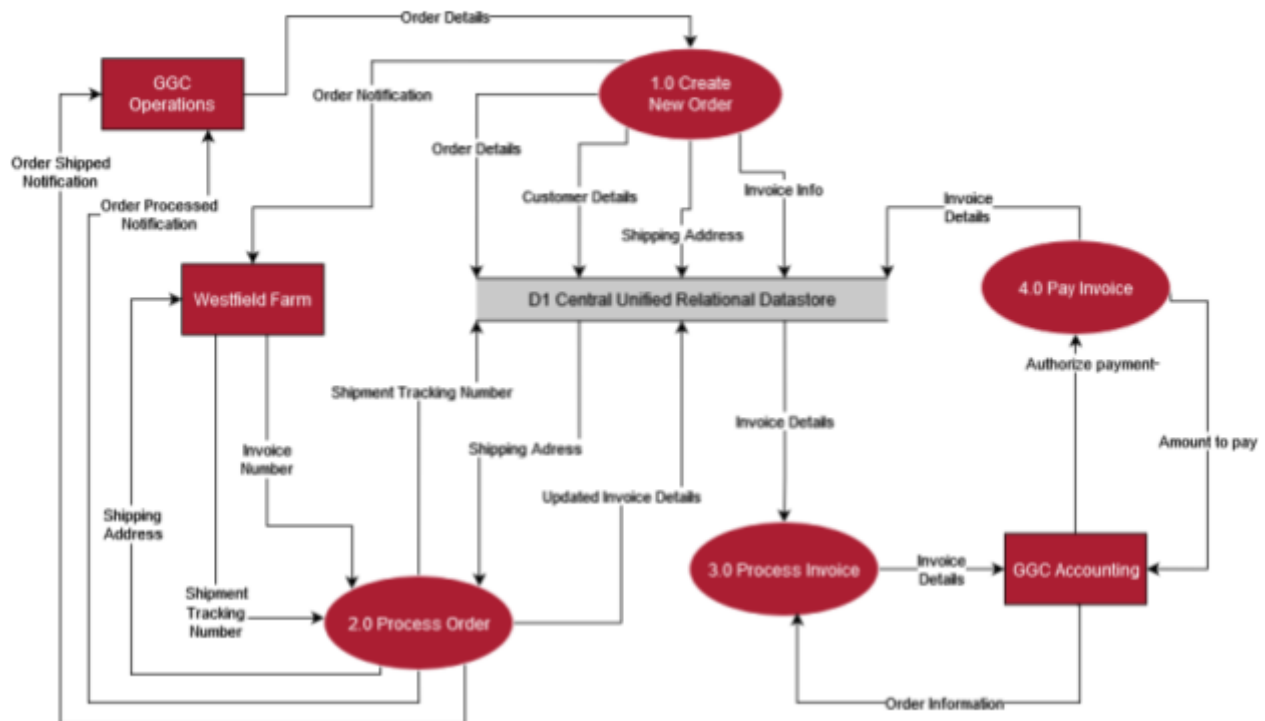


Figure 5.2: Updated Level 0 DFD

Another key change displayed in Figure 5.2 is the change in how data is stored. The second iteration condensed the four separate data stores from iteration one into a single datastore, nicknamed “Central Unified Relational Database” or CURD for short. This is representative of how data is stored in separate tables in one database as opposed to separate storage entities. The reasoning for this change is further discussed in the Development chapter.

Every system will have a context and level zero diagram, and additional levels can further break the details in each high-level process. The team developed a level one DFD

fragment to provide further detail on Process 2.0 (Process Order) in Figure 5.2. The level one fragment expands the process into two sub-processes: Process Invoice and Pack and Label Order for Shipping.

Shown in Figure 5.3, the process begins from order notification that flows out of Process 1.0. The farm then retrieves the order info from the datastore and enters that information in their accounting system which generates an invoice. As part of the first sub process, this invoice number is then added to the datastore and notification is sent to GGC Operations that the order is being processed. Following that the farm packs and then labels the order for shipment. The farm uses an online tool for managing their shipments that generates shipping labels with tracking numbers. The second sub process involves adding the tracking number to the datastore with a link to the associated order and sending an order shipped notification to GGC Operations.

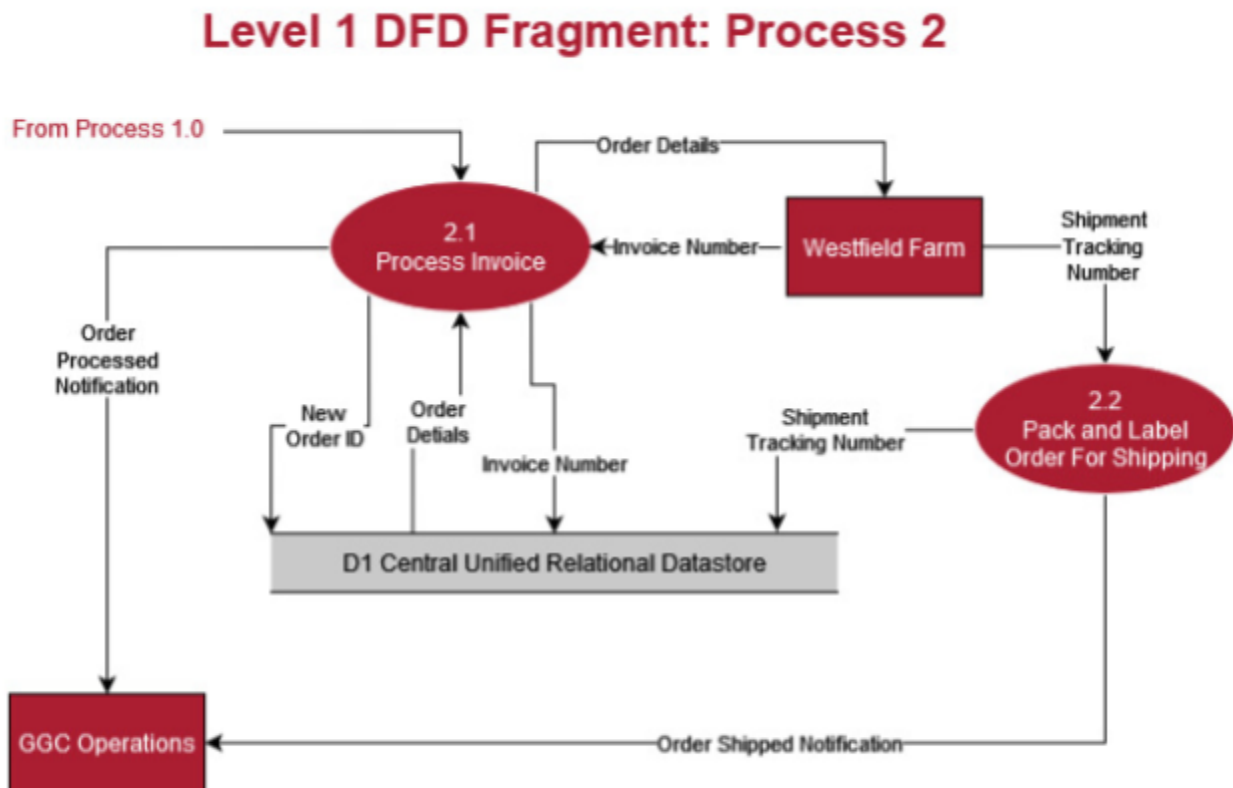


Figure 5.3: Process 2 Level 1 DFD Fragment

5.3 Additional Use Case

Use cases define events that trigger actions within the system. The list of use cases developed by the Phase One team covered the core functionality that the system needs in order to be beneficial to both GGC and the farm. These use cases can be found in Appendix G along with the additional use case below. After user testing, the Phase Two team created one additional use case with a focus on accessibility:

Information Icon	
Participating Actors	User
Entry Conditions	1. Hover mouse over information icon next to site function
Exit Criteria	Brief explanation of function and its uses
Flow of Events	<ol style="list-style-type: none"> 1. User hovers over or clicks information icon 2. Small popup shows explanation of site function and its uses

Table 5.1: Tooltip Use Case

5.4 Database Schema Refinement

As mentioned in the background section of this paper, the Phase One team chose to implement a NoSQL MongoDB for simplicity. However, they described an entity relationship diagram and data dictionary that “reflects the tables that would be expected in a relational database” (Guerrette & Mohn, 2022). In refining their system, the team took the opportunity to determine what kind of data management system would best suit the GOATs portal.

Initial Database Comparisons

Typically NoSQL databases are very fast, easy to develop, and straightforward to understand. They allow developers to control the data structure and quickly create a scalable, high-performing backend. They were specifically designed to handle high levels of data traffic with apps that are constantly running. NoSQL became increasingly more popular with the rise in Big Data analysis because the exponentially larger size of the dataset requires high performance and flexibility. However, because of the non-relational nature of the data, NoSQL databases do

not support ACID: atomicity, consistency, isolation, durability (Bourgeois). This means that duplicate entries of data are often overlooked, an issue that the GGC operations team currently struggles with by using Google Sheets.

On the other hand, relational databases were actually created to store transactional data and support multi-record ACID transactions. Additionally, they have rigid schemas and allow for the use of joins, making it easy to access data from multiple tables that have relationships between them. MySQL relational databases are also the most widely used and early taught database, developed back in the 1970's for the purpose of reducing data duplication (mongodb.com). Using a relational database does, however, require setting up an ORM, or object-relational-map, which requires more development and testing time (Bourgeois).

Ultimately, we determined that despite the ease of NoSQL, GGC's proposed data warehouse is most effectively modeled by a relational database. As stated in the previous MQP, "One of the goals that the project hoped to accomplish was to find a way to reduce redundancy" (Guerrette & Mohn, 2022). While the speed, scalability, and ease of cloud integration with NoSQL databases is tempting, GGC's data is most effectively described as relational. And, with the help of Amazon Relational Database Services (RDS), creating a cloud instance of a relational database is not nearly as complicated as it once was.

Although we chose to transition to a relation database, we still had two relational database providers to compare: PostgreSQL and MySQL. PostgreSQL emphasizes the extensibility of SQL and was developed to handle transactions at the *enterprise* level, which would ultimately make sense to use for an Enterprise Resource Planning system. However, MySQL is often the first querying language a data designer learns. With the help of MySQL Workbench, a tool that visualizes the data modeling process, identifies syntax errors in queries, and performs operations all in one platform, it is clear that managing a MySQL database is very convenient and easy to learn for beginners. Syntactically, the two are very similar, but MySQL Workbench is an integrated environment that only supports a MySQL database system.

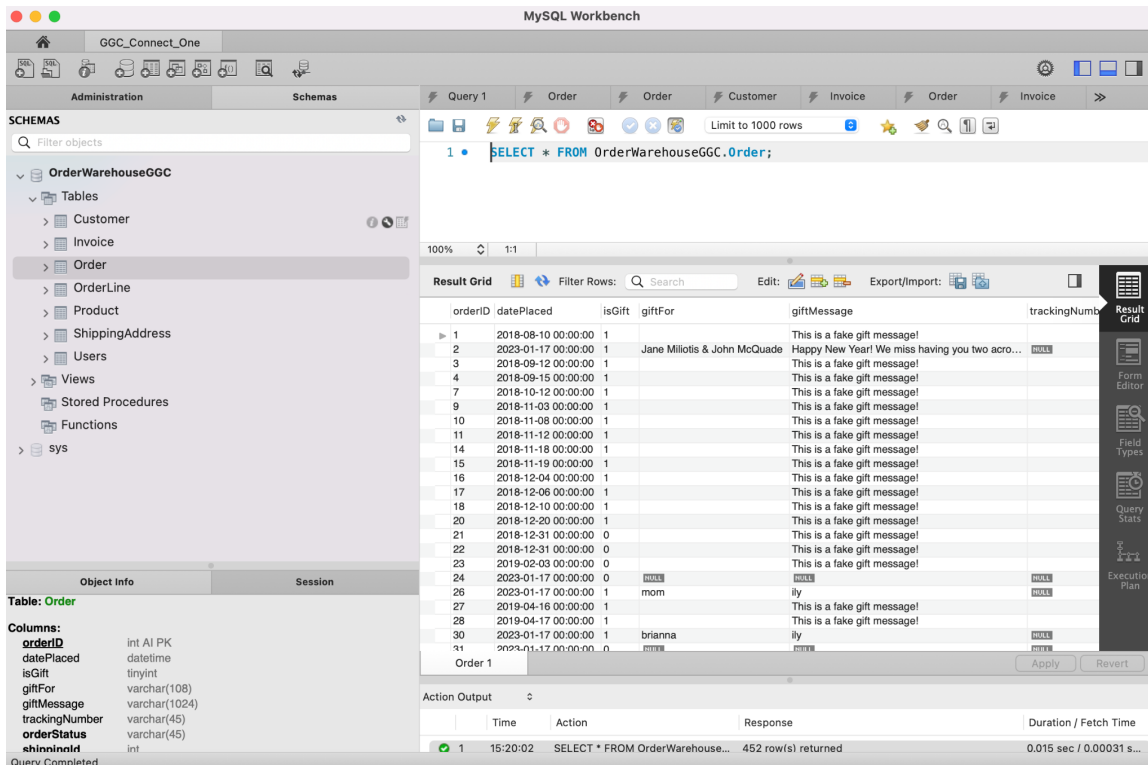


Figure 5.4: MySQL Workbench visual representation of the database

Keeping in mind the system’s unique target audience, we know that a small, student-run business with constant turnover will not likely reach the magnitude needed to warrant an enterprise level database. Additionally, GGC volunteers are driven to join by the learning opportunity, and ultimately, the management of this database will be in the hands of student volunteers with varying levels of technical experience. Providing the option to use MySQL workbench is a huge learning advantage for future users, or developers maintaining the system. Next, we explain the refined data model that would be represented by a MySQL database.

Logical Entity Relationship Diagram and Data Dictionary

First, the team reworked the logical entity relationship diagram (ERD), or visual representation of the relational database, to more accurately represent the system needs and adjustments.

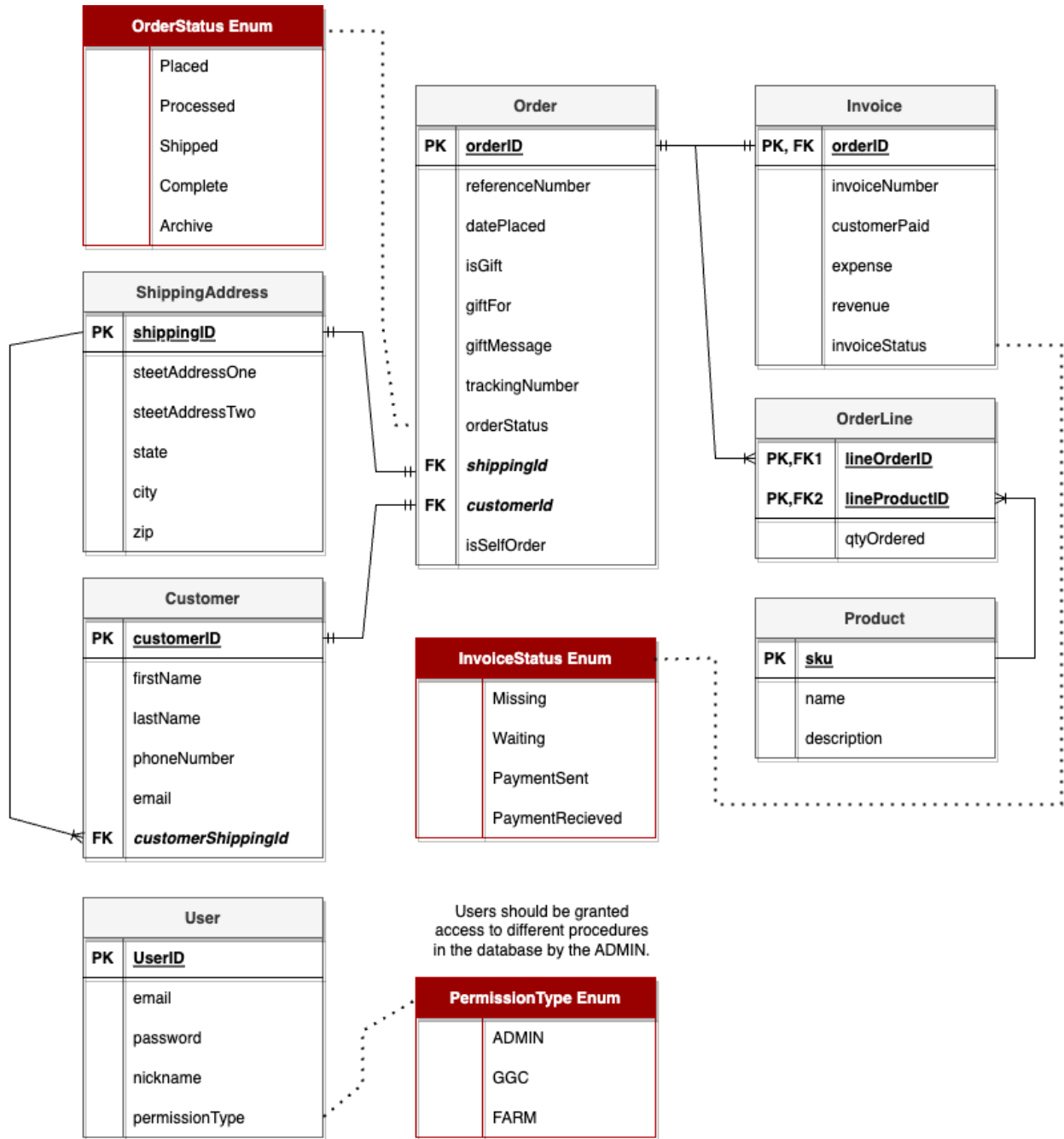


Figure 5.5: GGC ERP - Entity Relationship Diagram

It is important to note that the best practice for relational databases is for all information to be related to each other. While the User table does not directly relate to any of the other information in the warehouse, this entity could be used to grant permissions and roles to different

future users. The Phase Two team did not implement any procedures, roles, or groups within this iteration, however, this idea will be reiterated in the Future Works section of the paper.



Figure 5.6: GGC ERP - Entity Relationship Diagram with Referential Integrity

Additionally, a data dictionary describing the tables reflected in the relational database, and defines all of the details for the attributes of every table is located in Appendix I.

5.5 High-Fidelity Prototype Improvements

The team updated the dashboard, changed the flow, and defined differences between the invoice and dashboard. The user flow of the figma file was cleaned up to increase efficiency and minimize the amount of pages used. It is divided into three core sections: admin settings, invoices, and orders.

The dashboard was updated to highlight action items and include the ability to search orders and add a new order right after login, which can be seen in (Figure 5.7). This reduces the amount of screens and clicks the user would need to go through to get the information they need. The “quick stats” were kept from the Phase One Team, because it breaks down what the orders/invoices pages show in a concise way. These also link to the appropriate pages if clicked.

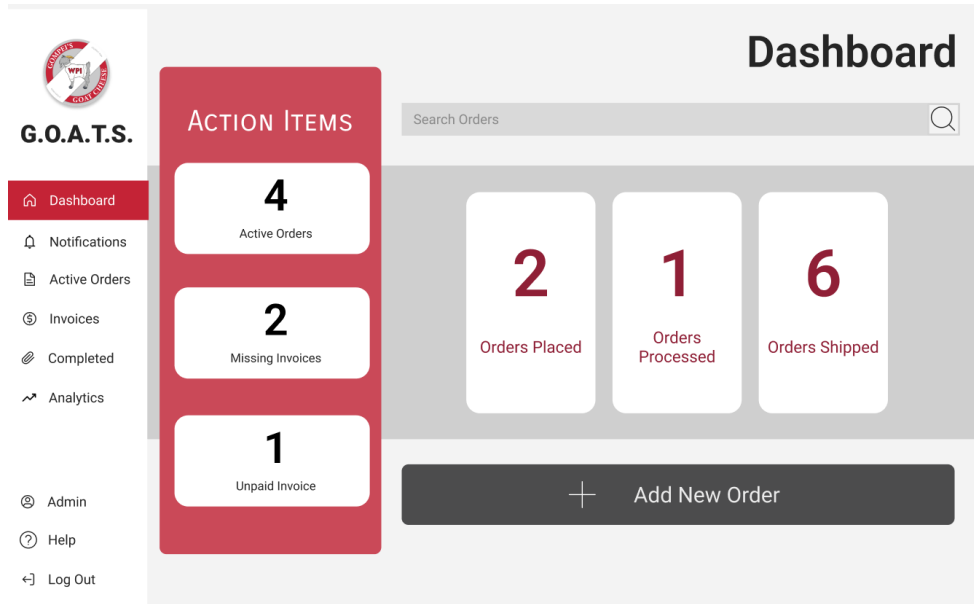
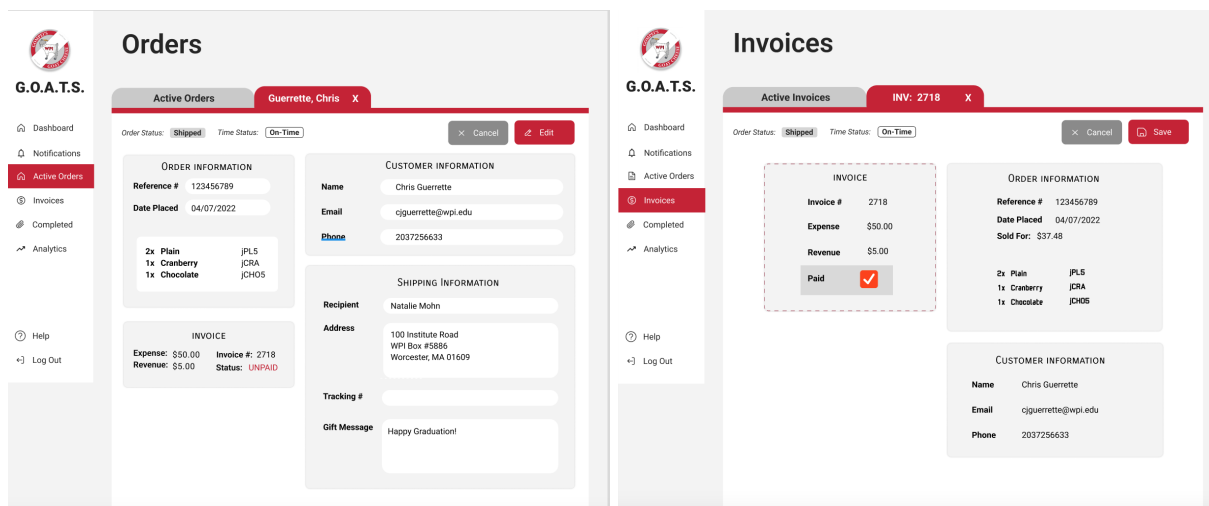


Figure 5.7: Proposed mockup of new dashboard

After meeting with the COO they expressed the need of distinguishing the difference between invoices and orders. The orders tab allows the COO to edit any of the customer/payment information. Invoices on the other hand are used by the accountants to track the flow of money easily, and can only edit whether an order is paid or not. The paid checkmark will be password protected so that not just anyone can change an order status.



Figures 5.8a & 5.8b: Figma Mockups for Orders and Invoices respectfully.

Westfield Farm Perspective

The team also created a mockup of how the farm side would look. Bob would only need to edit the invoice number and expenses owed. We wanted to ensure the order information was not accidentally editable by Bob, since he is older and unfamiliar with information systems. He is only able to view and copy customer information. He then inputs the invoice information and the COO gets a notification when saves the updated information.

G.O.A.T.S.

Dashboard
Orders
Search
Help
Log Out

Orders

Active Orders **Guerrette, Chris** X

Order Status: Unpaid Time Status: On-Time Cancel Save

ORDER INFORMATION		CUSTOMER INFORMATION	
Reference #	123456789	Name	Chris Guerrette
Date Placed	04/07/2022	Email	cjguerrette@wpi.edu
2x Plain	JPL5	Phone	2037256633
1x Cranberry	JCRA		
1x Chocolate	JCH05		
Sold For:	\$37.48		

INVOICE		SHIPPING INFORMATION	
Invoice #	<input type="text"/>	Recipient	Natalie Mohn
Expense	<input type="text"/>	Address	100 Institute Road WPI Box #5886 Worcester, MA 01609
		Gift Message	Happy Graduation!

Figure 5.9: Mockup of the farm facing side.

6.0 Development

Good software is described as “maintainable, dependable and usable” (Bittner & Pureur). With good software principles in mind, we chose to give the legacy code a major makeover in our project iteration. What existed as a small React project with a few components grew into a crafted codebase and technical stack that we found to best suit GGC’s needs. Thus, establishing a proper software architecture in the GOATS Portal was a key factor in continuing its development.

6.1 Software Architecture

The Portal’s main codebase is one parent directory (GGCPORTAL) containing the active project directory in addition to the MySQL scripts and backup CSV tables to easily recreate an instance of the database. Originally, this directory held the archived code for reference, stylistic inspiration, and ownership reference to the original developers. However, it was removed from the main branch to reduce deprecated dependency issues once making the transition to a cloud environment. Many of the components in the client directory are similarly structured to their counterparts in the archive. This client folder is its own React project containing the static files for the frontend components and React Redux store configuration. The table directory contains the python script created to traverse through the existing GGC order spreadsheet, SQL scripts for each database table, and the starting CSV files for each table. It’s important to also note the .ebextensions, .elasticbeanstalk, server.compiled and Procfile in figure

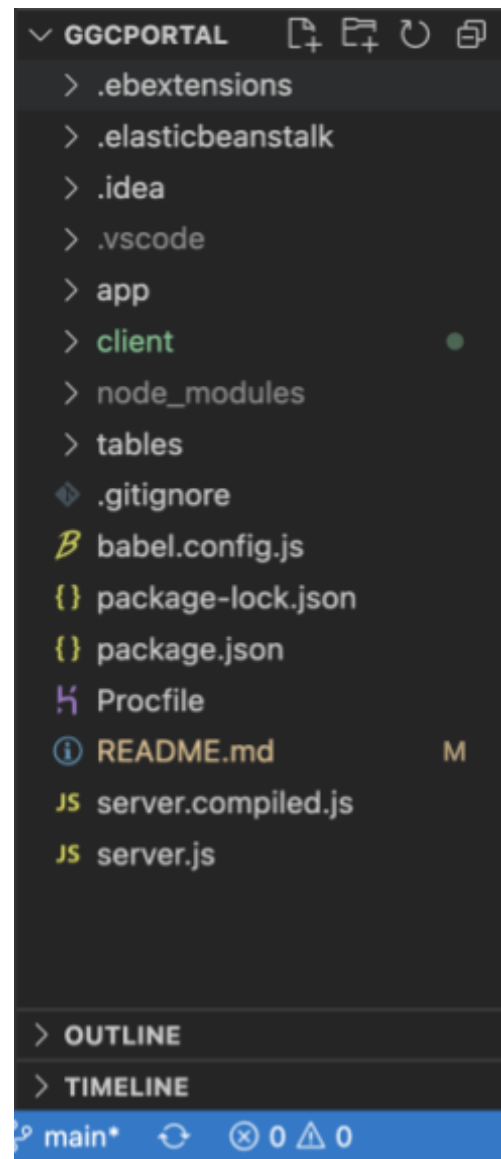


Figure 6.1: Project Directory

6.1 of the folder hierarchy. These are necessary for deployment on AWS Elastic Beanstalk.

Working under the impression that developers will continue to iterate on this codebase in the future, we aimed to establish a sensible hierarchy so that future collaborators will have an easy time navigating the codebase and avoiding potential coding conflicts. As seen in figure 6.2, there are three core components to the overall architecture: the database, the server, and the pipeline. The MySQL database (Amazon RDS) stores all of the GGC data. The code repository has an Express.js root server with a React.js frontend (client). The client is managed by a Redux store and uses the Axios library to form an http connection with the backend data service set up with Express.

The external softwares used to set up the pipeline were GitHub, AWS CodePipeline, and Amazon Elastic Beanstalk. Github allows for version control, storage of the codebase, and other code sharing benefits. The AWS CodePipeline gets the code ready for deployment as it directly connects the Github repository to the Elastic Beanstalk environment. Upon the most recent push to the main branch of the repository, AWS Elastic Beanstalk deploys the most recently healthy version of the software to a URL. One benefit of the pipeline is that it protects the deployed application by reverting to a healthy build if something goes wrong.

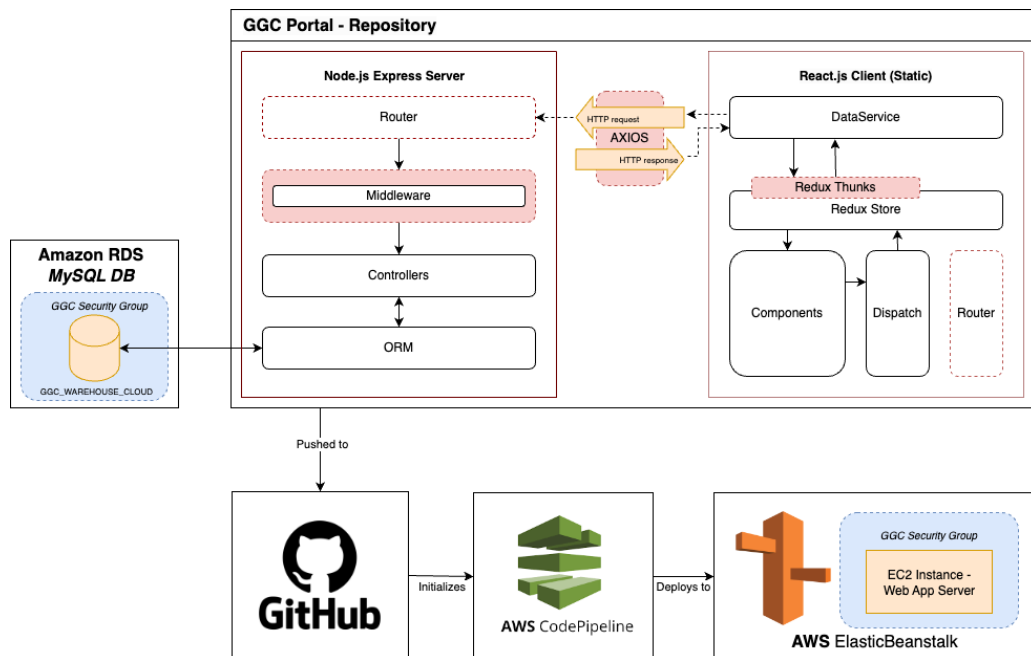


Figure 6.2. Diagram of the software architecture and code pipeline.

It is important to note that the figure above provides the overarching architecture and proposed pipeline, however this was not used for the entirety of development. Due to frequent testing, it was more sustainable to connect to a local database endpoint with an identical schema and deploy changes locally.

The table below summarizes the technical stack used the team used to develop the cloud ERP system:

Software/Technology	Version	Notes
Express	^4.18.2	Server and routing
React	^17.0.0	Client code framework
Redux	^7.1.0	Redux Toolkit (1.9.1)
Axios	^1.2.2	Forms http connection
Node.js (Elastic Beanstalk)	16.16.0	Runs the Node.js environment.
MySQL	5.7.41	Hosted through Amazon RDS

Table 6.1: GGC Portal Technical Stack (See Appendix J for more)

Tool:	Usage:
Visual Studio Code	Major IDE used for development
Intellij IDEA	Front end development
MySQL Workbench	Data modeling and database development
Amazon Web Services (AWS)	Hosted the bulk of the project and used for deployment

Table 6.2: Tools and Technologies Used

For more information on the software versions used in the implementation, please see the package.json file located in Appendix I.

6.2 Back-End Development

The backend development was done in two parts; create the database server and schema for testing and create an Express.js ORM to allow data to pass to the frontend.

Schema Generation and DB Configuration

In order to test the proposed relational database, the team set up a local server and created the tables. Though tedious, the actual schema setup was not especially intricate once a server connection was created. Using MySQL Workbench and the data dictionary described earlier, the team developed an active MySQL data model with scripts generated such that it can be easily recreated in a new database instance. This should help future developers avoid starting from scratch.

GGC's current order tracking system resides in a Google Sheets document. In order to test the system being developed as well as maintain continuity, the team developed a Python script that reads the data from a CSV file downloaded from the Sheet and outputs separate CSV files with the data necessary to populate the tables in the database. Detailed in the script which is located in Appendix K, some orders were manually transcribed to these tables because they were few in number and too complex for the script to easily digest.

Additionally, within the code repository is a subfolder "GGCPortal/tables/sql scripts" which contains the SQL script files with every table's Data Definition Language (DDL) - a set of SQL commands used to create, modify, and delete tables and other database structures. There is also a script that defines a trigger in the database which creates a new invoice record with matching order ID. If future iterations of the project ever need to recreate the database, they create the tables in the following order: User,

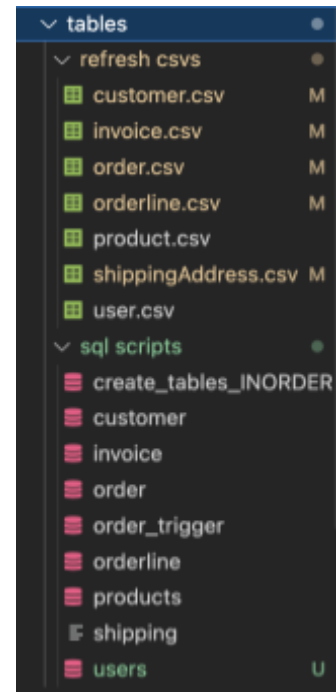


Figure 6.3 Tables Directory

Product, ShippingAddress, Customer, Order, OrderLine, Invoice. At this point, import any historical records by importing the appropriate CSV files, which are also located in this subfolder of the repository. Finally, enable the trigger in the database environment. It is necessary to import the data prior to enabling the trigger in order to avoid incorrect mapping of the order-invoice relationship.

Express.js Object Relational Mapping

The Express.js server has been referred to as the “root” of the project repository. This framework is what helps translate information between the MySQL database and Javascript. Express servers are a very popular choice for building web applications using Node.js as it allows developers to easily create robust and scalable software. They have the ability to handle

HTTP requests and responses, which makes it ideal for building RESTful APIs. This was exactly the next step needed to continue development of the backend from the Phase One team’s prototype. These API’s allowed access to the database fetching to occur from the frontend. As a reminder, the seven tables in the schema are for address, customer, invoice, order, orderline, product, and user.

As seen in the figure to the left, the parent app directory holds all of the backend code. There are directories for configurations, controllers, middleware, models, and routes.

Models define the structure and behavior of the data, controllers define the behavior of the application, and routes define how the incoming requests are handled by the server. In order to develop an API for each table in the database schema, the team first established a database connection (db.js) using an endpoint specified in one of the configuration files (config). The models are JavaScript classes that define the schema of each database table. These classes are responsible for interacting with and querying from the database through an ORM. The models developed in this

project iteration include create, read, update, and delete queries for each table, with additional

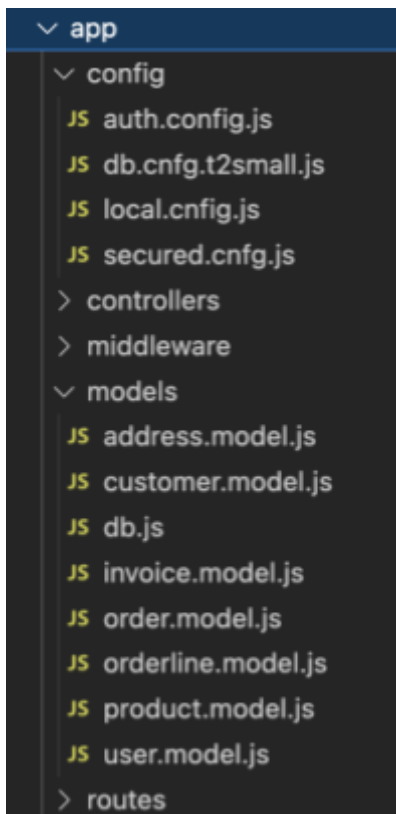


Figure 6.4: Express Directory

queries for certain tables, depending on what was needed in the frontend (i.e., readAll, deleteAll, etc...). On the other hand, the controllers are JavaScript functions that handle incoming requests, process the data, and return the response to the client. The team set up one controller for every model as controllers make use of one (or more) model(s) to retrieve or update data in the database. Lastly, the routes are a set of rules defining how incoming requests are handled by the server, using the HTTP methods GET, POST, PUT, and DELETE. Each of the routes are associated with a specific controller that handles the actual request and response data.

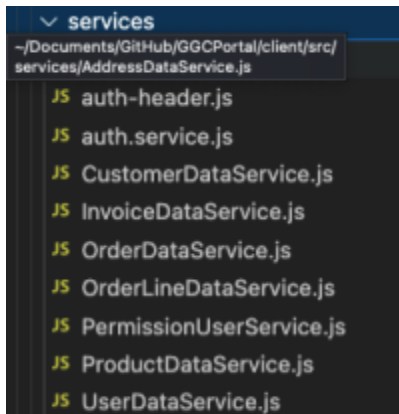


Figure 6.5 Axios Data Services

Although the Axios library is used in the frontend directory, the DataService purpose should be discussed with the backend development. Classes are again separated by database table, but the data services use an Axios connection to make http requests from the frontend. These requests are mapped to their specified ORM's route, thus allowing CRUD operations to happen across the stack.

6.3 Front-End Development

To iterate on the existing React prototype, we first refactored the way screen components were used and how data is managed on the frontend. Even though the archive contained usable React code, the way the components managed frontend data was not reasonably scalable. React typically manages information on the frontend through the React State, but a project with so much potential for growth requires a structured pattern. React Redux is a state container for Javascript applications that provides predictability and optimization when accessing data in the state. Using the React Redux modern toolkit, state management becomes a lot simpler.

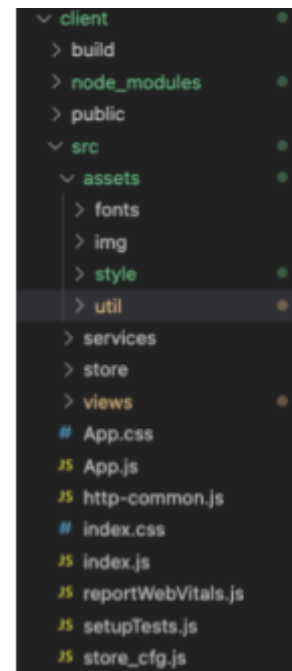


Figure 6.6 React Client Directory

The figure on the right shows an overview of the frontend (client) directory within the codebase. Within the source folder (src), there are assets, services (discussed in Chapter 6.2),

store, views, and other important files detailed in Appendix M. The store is the directory holding the Redux Toolkit reducers, and the views directory contains all of the React classes and components.

State Management with React Redux and Redux Toolkit

React Redux is a popular library for managing state in a React application. The benefits of using React Redux are centralized state management, predictable state updates, improved performance, better code organization, and easy integration with React.

Centralized state management means there is a single place that manages the state of the application, making it much easier to keep track of state with multiple types of users interacting with it. This also allows for more efficient state updates and gives access to the state from any component in the application, which is important when different users need access to the same data, but the software is rendering a completely different component depending on the permission. State changes are easier to follow, especially since we used Redux Toolkit. This framework provided the ability to create slices - or pieces of the reducer - for each unique part of the state. Each of these slices were actually just a representation of the backend tables in the frontend state. Despite having seven different slices setup, Redux allows developers to get access to some parts, but handles the change together so nothing is out of sync.

With central state management comes predictable state updates. There is only one way to update the state and that is to submit changes through the UI. These actions are dispatched directly to the store, making it easier to understand how the state changes in response to user actions. Considering accessibility and the desire for a simple experience, this is a significant benefit for the app. Take a look at the following visual to better understand how the Redux flow improves the software design (Figure 6.7):

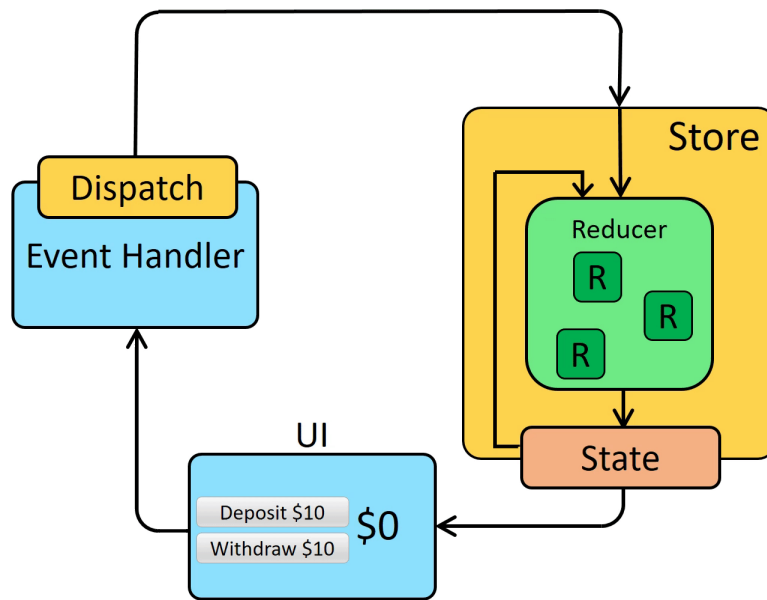


Figure 6.7: Visual example of React Redux data flow.

Here, the Store is shown as the container for a single reducer, with three small “R’s” inside, representing the slices that are managed within its container. The state is populated by the reducer, the UI loads the state data, and different events dispatch unique actions to the store. This pattern has a strict unidirectional data flow, only allowing the state to be modified through predefined actions, thus ensuring that state changes are predictable and easy to debug. This flow also intends for users to feel more control over their actions.

The Redux solution also reduces the number of times the page re-renders, which helps avoid unnecessary recalculations and ultimately improves the overall performance.

Lastly, it is important to highlight the benefits Redux and Redux Toolkit provided for code organization and developer experience. Redux Toolkit encourages the separation of the presentation logic from the business logic which makes it easier to maintain and test the code. This made the developer’s on the project much easier as the application state was actually separate from the components local state, making the code more modular, reusable, and understandable. Additionally, the set of utilities provided by Redux Toolkit allows for faster development by reducing boilerplate code typically needed to create a centralized store, which was critical on the short MQP timeline. Specifically, the team made use of preconfigured middleware for handling async actions as it allowed them to make API calls to the backend within the async function of a database table’s respective slice.

Because both React and Redux are popular tools, the online developer communities are vast and provide access to many tools and resources. Redux has incredibly easy integration with React as it is designed to work seamlessly with it. Overall, React Redux helps simplify the state management of a React application and improve the overall performance and maintainability of the project code.

Layout and Styling



Figure 6.8: Hex numbers of GGC colors

In addition to implementing a solid structure for the frontend with React Redux, the team also utilized Bootstrap and CSS to create a cohesive style across each user interface. We kept the same color scheme and layout that was made by the Phase One team. The GGC marketing team identifies these four colors as part of branding so we continued to design the interface to follow these patterns. We generally used flexbox to arrange the objects on screen so that it is adaptable to all screen sizes.

6.4 Cloud Deployment

Part of this iteration’s development involved a deeper investigation into the “cloud” piece to “Cloud-Based ERP.” This meant finding an appropriate solution for hosting the web server and database for multiple users across many networks. The team decided to stick with Amazon AWS for the cloud services like the Phase One MQP team, but focused on Amazon Relational Database Services (RDS) for the database and Elastic Beanstalk (EB) to host the Node.js project itself.

Over the course of the project, the team went through trial and error when setting up the database with RDS. The main takeaways were to set up the appropriate instance configuration, set up a security group, set up a parameter group, and ultimately just create the database through the EB environment directly. The team chose a small.t2.instance for the database because it seemed to be the cheapest option to couple with the EB environment security allowance; the free tier eligible databases don't work sustainably with EB because they lack proper configurations and security tolerance. The security group is used to control inbound and outbound traffic, so it is important to only specific IP addresses access the database, such as the address from the EB environment. To allow the correct traffic to the instances without having to reconfigure the settings every time, the team created a GGC-MQP security group that specified the inbound rules to allow access to the database instance from the developer's IP addresses and the EB environment. The parameter group defines a collection of database engine configuration settings. Adding this to the RDS instance allowed the team certain actions on the database that were previously prevented or hidden. Provided there is already a security and parameter group set up, it is really simple to just create the database instance through the EB environment directly.

Setting up an EB environment was quite simple to do after using the AWS CodePipeline and configuring the RDS instance within the environment. This process is detailed in Appendix M.

The following figure shows an earlier version of the software that was deployed to the Cloud before terminating the environment due to budget cuts.

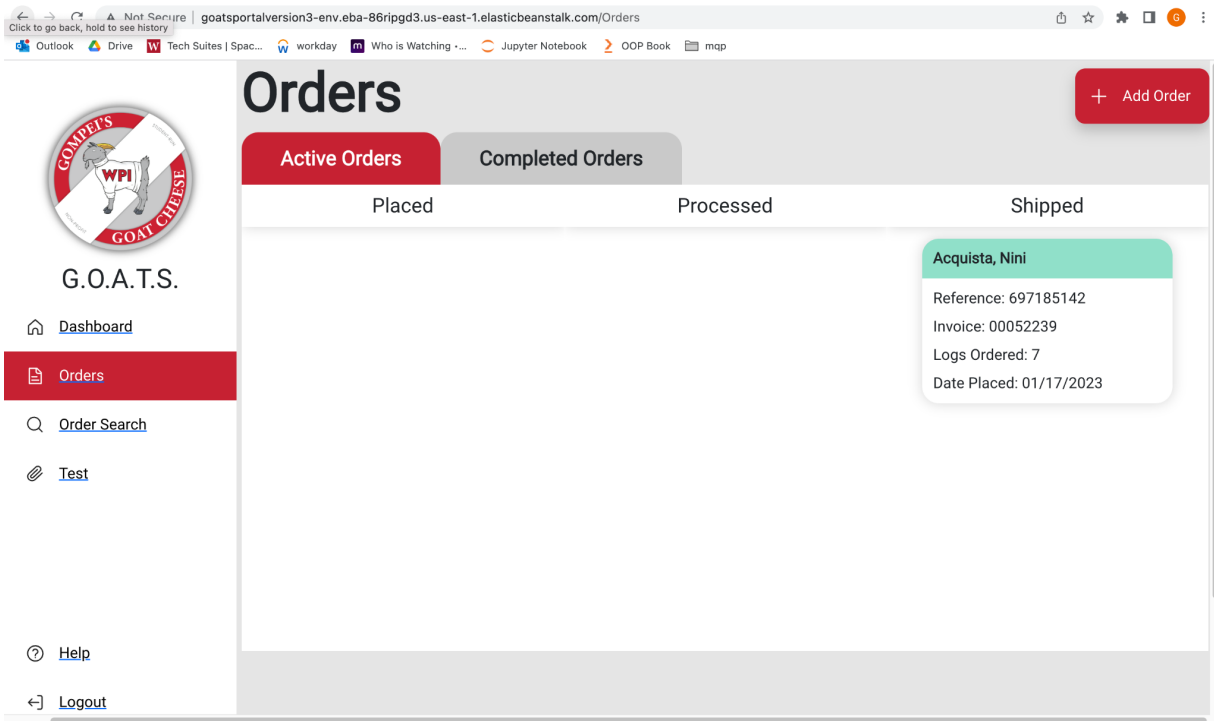


Figure 6.9 Software Version Deployed to Elastic Beanstalk Cloud Environment.

6.5 Persisting Issues

Having very few developers working on a full stack application caused issues such as the agile methodology not being maintainable and sufficient progress being stunted by backtracking. Some frequent trouble spots were in the backend maintenance. There was a big learning curve with maintaining a healthy connection with a backend while working to refactor a consistently buggy frontend.

Given the scope of the project and varying skill focuses in our group, it was difficult to work with the front-end database connection while simultaneously trying to define a properly structured database instance. For example, whenever the data needed to be reset or uploaded into a new or existing database instance, it was imperative to remove the triggers that had been set or else the existing orders would not be mapped to their respective invoice. Another improvement to the database would be including shipping cost and unit values for each product so values can dynamically update rather than having to update invoice values manually.

Due to time and lack of bandwidth, the main priority of this project iteration was to establish a modular software for future expansion by students. Despite positive intentions using a 1:1:1 ratio of routes/models/controllers for each database entity, this is not a best practice and prevented the team from completing CRUD actions for all tables. The decision was made for ease, time constraints, and compatibility with the redux store. Through the progression of this project, the team found that refactoring the ORM was out of scope for the project time. However, the ORM should be adjusted to efficiently utilize joins between tables, grant table permissions to users of the database, and incorporate procedures to reduce logic on the frontend.

6.6 Functional Prototype

This section includes images of the final prototype submitted. More information regarding its full-fledged functionality are mentioned in Chapter 7.1, further shown in Appendix D, and detailed in Appendix M. Outlined below is one particular flow that is fully functioning in the prototype (edit screen not included but viable):

1. Login to the system as a regular GGC User

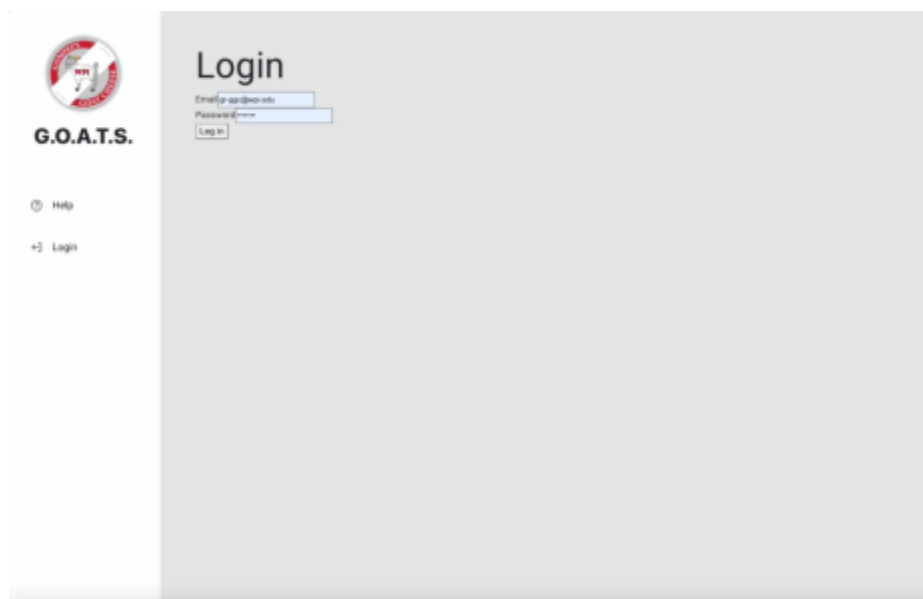


Figure 6.10 Login Page

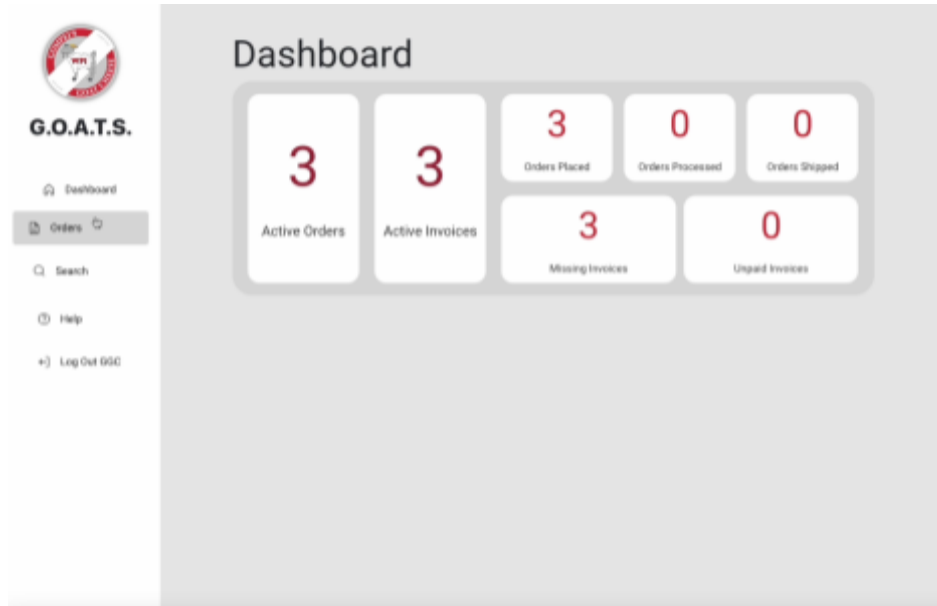


Figure 6.11 Dashboard Page

2. Navigate to the Orders tab

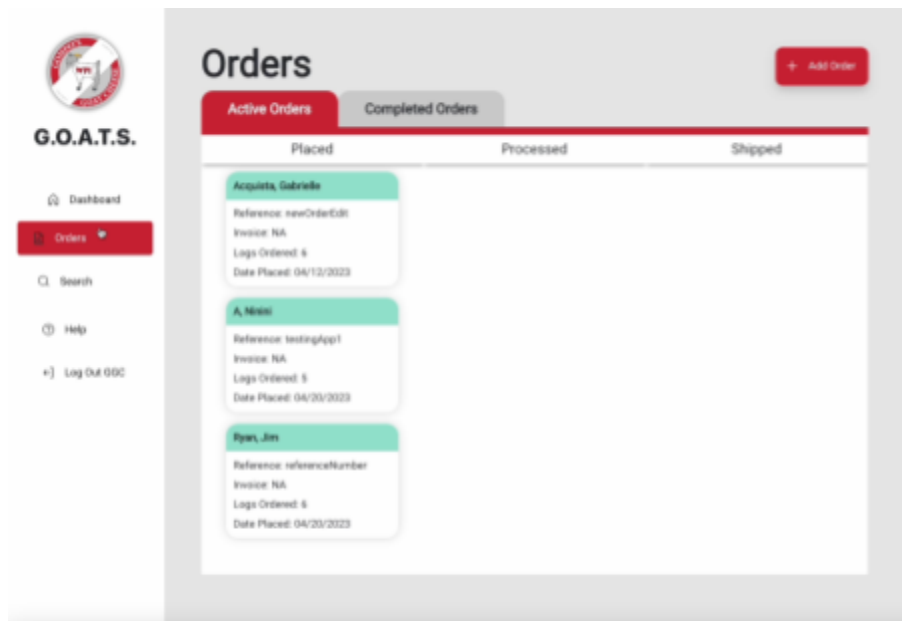


Figure 6.12 Orders Page

3. Click add order

The screenshot shows the G.O.A.T.S. Orders page. The page title is "Orders" and the user is logged in as "Ryan". The "New Order" tab is active. The form includes the following sections:

- ORDER INFORMATION:** Reference #, Date Placed (mm/dd/yyyy), Tracking #.
- CUSTOMER INFORMATION:** First, Last, Email, Phone.
- ORDER DETAILS:** Flavor (Plain), Add Flavor.
- SHIPPING INFORMATION:** Street Address One, Street Address Two, City, State (dropdown), Zip.
- INVOICE:** Invoice Generated Upon Save, Internal Use Only: Customer Paid.
- GIFT INFORMATION:** (Empty)

Buttons: + Add Order, Status Placed, Cancel, Save Order.

Figure 6.13 Order Page

4. Fill in the order information

The screenshot shows the G.O.A.T.S. Orders page with the "New Order" form filled out. The user is logged in as "Ryan". The form includes the following sections:

- ORDER INFORMATION:** Reference # (newOrderTest), Date Placed (04/20/2023), Tracking #.
- CUSTOMER INFORMATION:** First (Mia), Last (A.), Email (mia@wpf.edu), Phone (413-232-1234).
- ORDER DETAILS:** Flavor (Pink Peppercorn), Flavor (Chocolate), Add Flavor.
- SHIPPING INFORMATION:** Street Address One (22 Highland St), Street Address Two (Apt 1), City (Worcester), State (MA), Zip (01529).
- INVOICE:** Invoice Generated Upon Save, Internal Use Only: Customer Paid (27).
- GIFT INFORMATION:** Gift For (Mia).

Buttons: + Add Order, Status Placed, Cancel, Save Order.

Figure 6.14 Editing New Order

5. Save the new order and see it in the Active Orders tab.

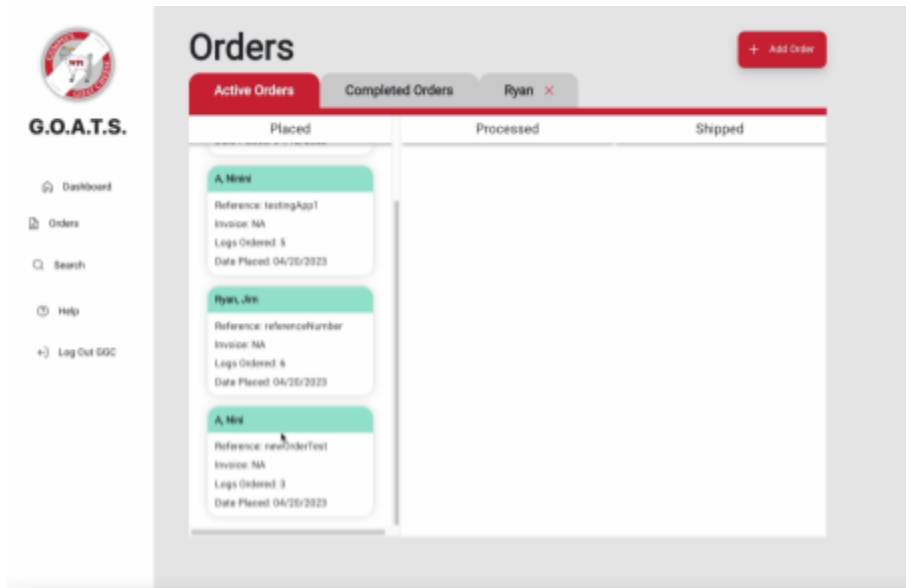


Figure 6.15 Active Orders Tab

6. Click the newly created order.

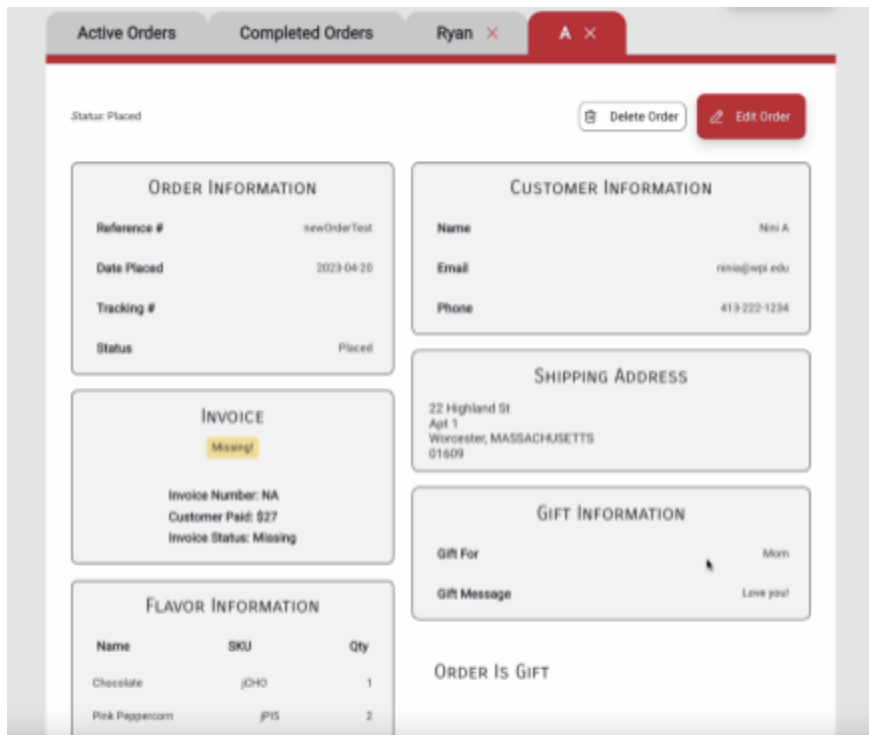


Figure 6.16 New Order

7. Navigate to the Search page and type the reference number in the search bar to watch as the orders filter.

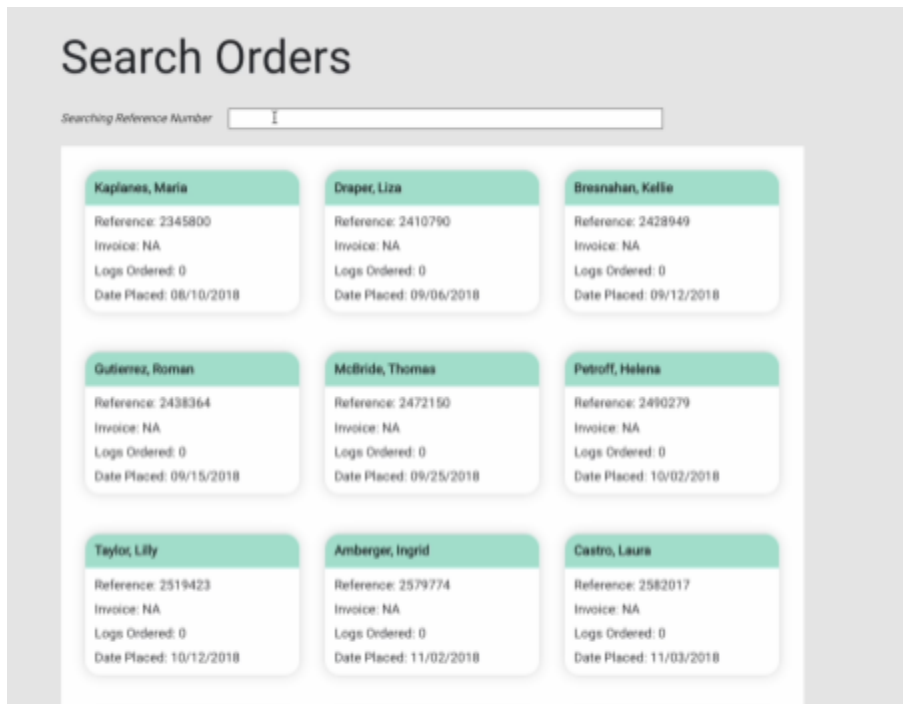


Figure 6.17 Search Orders Page

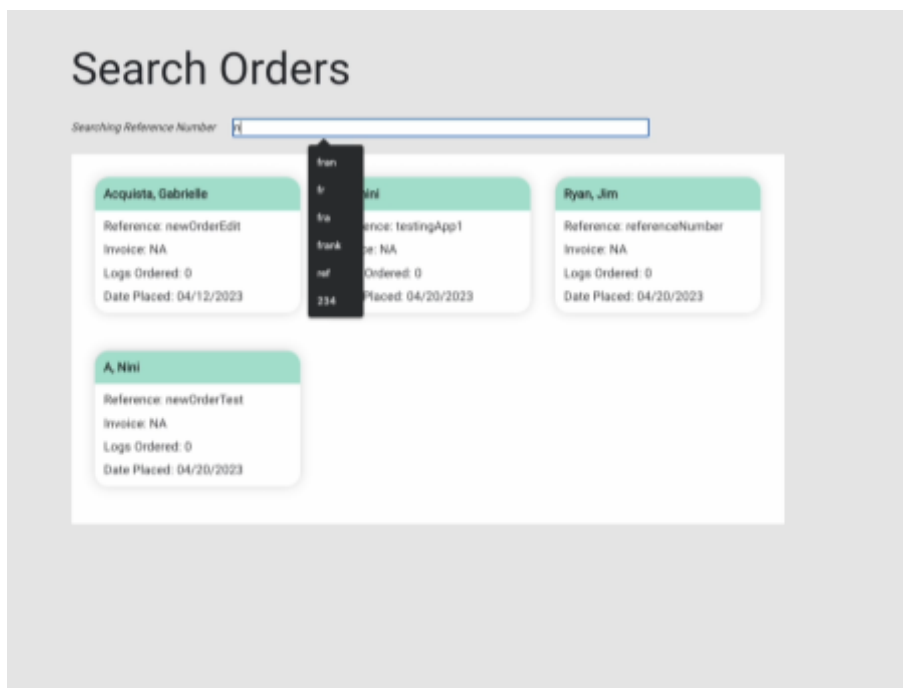


Figure 6.18 Searching in Search Bar

8. Go to your order and delete it.

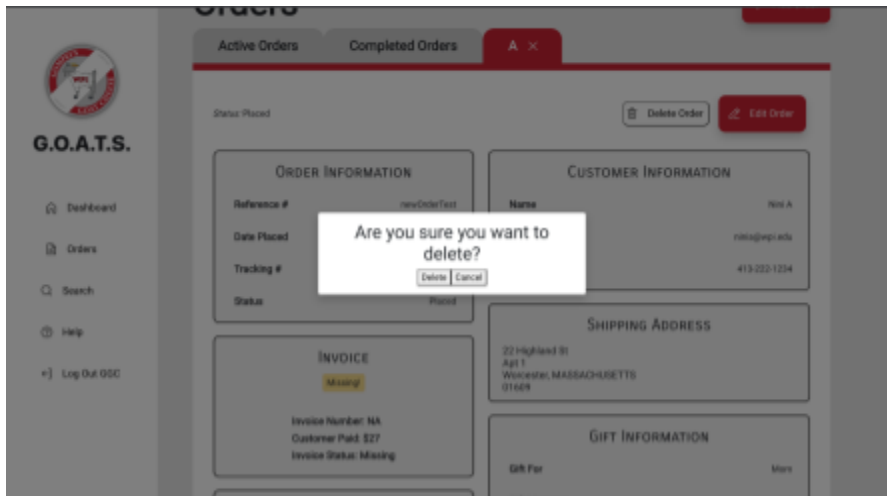


Figure 6.19 Delete Order

9. Confirm delete and verify it was removed from the Active Orders page.

Additionally, because the framework for user permissions was implemented, the following figures highlight the dynamic navigation bar for different views.

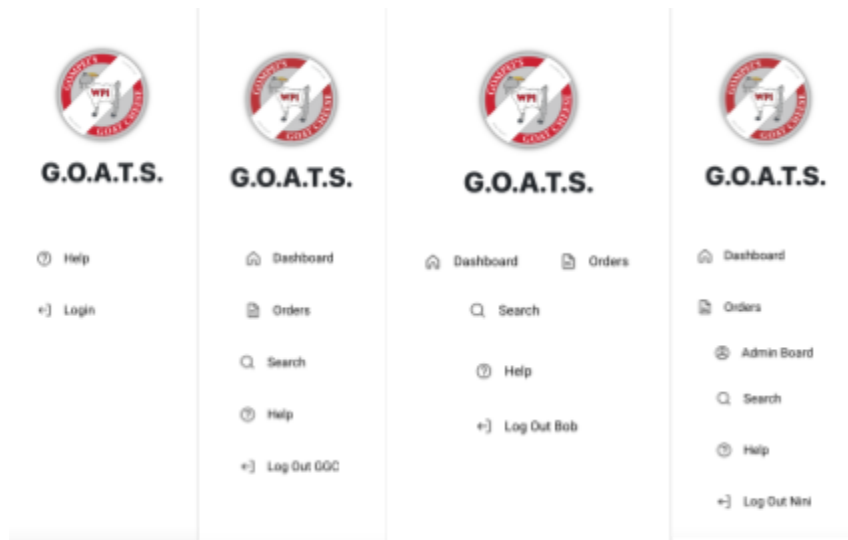


Figure 6.20: Navigation Bars by Role from Left to Right (Logout page, GGC role, Farm role, and Admin role)

7.0 Future Work

Although this is a functional prototype, it is not recommended the software is integrated into the company until proper user testing is conducted.

7.1 Next Steps

The most critical next step is to conduct a UX evaluation of the software and determine the most important next steps for future functionality. Several important steps based on the findings in this paper are as follow:

1. Refine the database schema based on updates within GGC's business processes.

GGC is a very dynamic, growing company, recently under new leadership. It may be necessary to modify the schema model defined in this project iteration, or even design a data warehouse for further development.

2. Refactor the Express.js API to better utilize the relational nature of the database.

While the team put a lot of research into choosing the most efficient database for the system, the time constraint of the project prevented the ability to implement joins. This fix would allow future developers to take better advantage of querying on the backend, and state management on the frontend. Redux Toolkit would allow developers to work with an entire order in one slice by fetching data from a joined table, rather than joining the customer, shipping, and order information through their respective slices in each component that an order is needed.

3. Zapier integration with Formsite.

Cheese orders are placed by customers directly through Formsite and sent to the GGC operations team, where it is looked over before being sent to the farm. Upon investigation, Zapier is a tool that can be used to scrape order data from Formsite and insert it into a specified database. Upon investigation, the team found that Formsite occasionally writes order data to the site's spreadsheet before there is a payment filled,

thus it was decided to forego the Zapier route until a more formal investigation takes place. As long as the payment is fulfilled, Formsite orders should be written to the database schema with default status “Placed.”

4. Iterate on delivery time and company communication management.

Despite including On Time/Early/Late tags in both mockup phases of this project, this feature is strictly a proof of concept and was not informed by any specific findings. However, given the inconsistencies and lack of proper communication between the farm, GGC, and customers about order placement and arrivals, this is an important feature to iterate on.

5. Encryption for login.

The functional login for this prototype is strictly proof of concept that user permissions should affect the app navigation. User passwords are stored as plain text and there are no tokens involved in authorization. This is an extreme security vulnerability and any future collaborators must correct it before implementation.

6. Update Node.js version and dependencies and get solid test code coverage.

Like many prototypes, this code is prone to mysterious bugs and behaviors, so it would be beneficial to investigate the quality before adding additional functionality.

In addition to the steps outlined above, the README.md included in Appendix M further details bugs that need attention and future steps to take in order to integrate this software for GGC’s use.

General Development

The development of the platform is just beginning. The list below outlines what functionality was completed, the functionality currently in progress, and what is left to be implemented for increased usage for GGC and partners. Additionally, there is a list of potential features noted in the user study that were outside the scope of the team’s prototype implementation, but would incredibly increase the value of the portal. Please note that many of

the unfinished features should be documented as additional use cases in future iterations, as the modularity of the software presented benefits that were previously overlooked.

Functionality Completed	Functionality in Progress	Functionality to Implement	Special Features
<ul style="list-style-type: none"> ● View Completed Orders ● View Active Orders ● View Order Information ● Update Order Information ● Add User ● Remove User ● Record New Order ● Delete Order ● Log In 	<ul style="list-style-type: none"> ● View Customers ● View Shipping Addresses ● Update Products Ordered ● Tooltip ● Update OrderLines 	<ul style="list-style-type: none"> ● Copy Text Information ● Automate Status Updates with Early/Late/On-Time Tracking ● Automate emails between Customers, Farm, GGC ● Autofill for Existing Information ● Encryption on Login ● Autofill Existing DB Information in New Orders ● Cross Component Search 	<ul style="list-style-type: none"> ● Data visualization ● Specialized roles ● Inventory tracking ● Inventory machine ● Upload Invoice as PDF

<ul style="list-style-type: none"> ● Log Out ● Update Invoice Information ● Search Orders by Reference Number ● View Invoice 			
--	--	--	--

Table 7.1: Aspects of the functional system that are complete, in progress, future implementation, and special features to be added.

Westfield Farm Perspective

The Westfield Farm role has not been fully implemented and will need further investigation and development to be complete. Currently, the design for the farm perspective is set in place, however, a prototype of this design has not been user-tested with the farm owner. Without user-testing, there wouldn't be an understanding of the usability of the design and the specific needs of the user. The farm owner is also elderly, so it is important to have a mistake-proof, simple, and accessible design. The farm owner has expressed a willingness to try new technology as long as it integrates with his own system well, however, it isn't known how well he can interact with a new system. To aid this, a future team would need to investigate what he is comfortable handling electronically to aid in forming the farm role requirements.

7.2 Final Thoughts

Gompei's Goat Cheese is a great pillar of the WPI Business School as it is the first student-run, non-profit business at WPI. However, the new CEO of GGC in 2023 wants to make some big changes to help grow Gompei's Goat Cheese to a new level. As of now, GGC remains mainly known to people surrounding the WPI Business School and not the rest of the community, so the CEO wants to expand GGC to be a pillar of the whole WPI community. This can be done by reaching out to the other schools of science and engineering to hire students for new positions like software engineers, data analysts, and project managers. This can introduce a learning experience for STEM students to work in teams within a real business, as well as help GGC grow to become a great asset to the science and engineering community at WPI. A goal for a future software development team would be to use an agile methodology to set up the software architecture to allow incremental changes without disturbing any part of the system. This would allow Gompei's Goat Cheese to progress and grow progressively larger over time.

References

- Baltzan, P. (2020). Integrating the Organization from End to End--Enterprise Resource Planning. *In Business Driven Technology* (8th ed., pp. 208–224). chapter, McGraw-Hill Education.
- Bangor, A., Kortum, P.T., & Miller, J.T. (2009). Determining what individual SUS scores mean: adding an adjective rating scale. *Journal of Usability Studies archive*, 4, 114-123.
- Better Business Bureau (2019). *10 Ways Small Businesses Benefit Their Local Communities*. Medium.com.
<https://medium.com/@BBBNWP/10-ways-small-businesses-benefit-their-local-communities-7273380c90a9>
- BezKoder. (2021, July 12). *Redux-Toolkit example with CRUD Application - BezKoder*. BezKoder; <https://www.facebook.com/bezkoder>.
<https://www.bezkoder.com/redux-toolkit-example-crud/>
- Bittner, K., & Pureur, P. (2022, June 8). *A Minimum Viable Product Needs a Minimum Viable Architecture*. InfoQ; InfoQ. <https://www.infoq.com/articles/minimum-viable-architecture/>
- Bourgeois, Dave. “Chapter 4: Data and Databases – Information Systems for Business and Beyond.” *Pressbooks Create – Your Partner in Open Publishing*, Published through the Open Textbook Challenge by the Saylor Academy, 28 Feb. 2014,
<https://pressbooks.pub/bus206/chapter/chapter-4-data-and-databases/#:~:text=All%20information%20in%20a%20database,created%20to%20manage%20unrelated%20information.>
- Dennis, A., Wixom, B. H., & Roth, R. M. (2019). *Systems analysis and design* / Alan Dennis, Indiana University, Barbara Haley Wixom, Massachusetts Institute of Technology, Roberta M. Roth, University of Northern Iowa. (7th ed.). John Wiley & Sons, Inc.
- Fisher, K. (2022, February 8). What is Cloud ERP and How Does It Work? Oracle NetSuite. <https://www.netsuite.com/portal/resource/articles/erp/cloud-erp.shtm>
- Guerrette C.& Mohn, N. (2022). *Development of Cloud-Based Enterprise Resource Planning Software for Gompei's Goat Cheese* [Major Qualifying Project]. Worcester Polytechnic Institute https://digital.wpi.edu/concern/student_works/jq085p23d?locale=en
- Interactive Qualifying Project*, (n.d.). Retrieved December 11, 2022, from <https://www.wpi.edu/academics/undergraduate/interactive-qualifying-project>

- Nielsen, J. (2012, January 15). *Thinking aloud: The #1 usability tool*. Nielsen Norman Group. Retrieved February 27, 2023, from [https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/Major Qualifying Project](https://www.nngroup.com/articles/thinking-aloud-the-1-usability-tool/Major_Qualifying_Project), (n.d.). Retrieved December 11, 2022, from <https://www.wpi.edu/academics/undergraduate/major-qualifying-project>
- Kamthan, P. (n.d.). Using Patterns for Engineering High-Quality Web Applications. In *Software Engineering for Modern Web Applications* (pp. 100–122). IGI Global. <http://dx.doi.org/10.4018/978-1-59904-492-7.ch006>
- mongodb.com. (n.d.). NoSQL vs SQL databases. MongoDB. Retrieved April 26, 2023, from <https://www.mongodb.com/nosql-explained/nosql-vs-sql#:~:text=and%20fewer%20bugs.,What%20are%20the%20drawbacks%20of%20NoSQL%20databases%3F,acceptable%20for%20lots%20of%20applications>
- Moore, L. (2018, July 31). *What is MySQL? | Definition from TechTarget*. SearchOracle; TechTarget. <https://www.techtarget.com/searchoracle/definition/MySQL>
- React Redux | React Redux*. (n.d.). React Redux | React Redux. Retrieved February 21, 2023, from <https://react-redux.js.org/>
- Style Guide | Redux*. (n.d.). Redux - A Predictable State Container for JavaScript Apps. | Redux. Retrieved February 21, 2023, from <https://redux.js.org/style-guide/>
- Small Business Administration. (2021) *Frequently Asked Questions*. <https://cdn.advocacy.sba.gov/wp-content/uploads/2021/12/06095731/Small-Business-FAQ-Revised-December-2021.pdf>
- Small Business Administration. (2022) *Massachusetts Small Business Profile*. <https://cdn.advocacy.sba.gov/wp-content/uploads/2022/08/30121319/Small-Business-Economic-Profile-MA.pdf>
- Sharma, A. (2022, September 8). *The challenge of high turnover at Startups: Founders Circle*. Founders Circle Capital. Retrieved March 3, 2023, from <https://www.founderscircle.com/high-startup-turnover-rate/>
- U.S. General Services Administration. (2023). *System Usability Scale (SUS) | Usability.gov*. Usability.gov. <https://www.usability.gov/how-to-and-tools/methods/system-usability-scale.html>
- Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), Victoria, BC, Canada, 2013, pp. 15-19, doi: 10.1109/PACRIM.2013.6625441.

Appendices

Appendices	73
Appendix A: User Testing Consent	62
Appendix B: User Testing Protocol	65
Appendix C: User Testing Data Analysis Sheet	70
Appendix D: Functional Prototype Images	76
Appendix E: Context Diagram	77
Appendix F: Data Flow Diagrams	78
Appendix G: Entity Relationship Diagrams	80
Appendix H: Use Cases	82
Appendix I: Data Dictionary	89
Appendix J: Package.json	96
Appendix K: MySQL Script	101
Appendix L: Python Script	105
Appendix M: Support Documentation	112

Appendix A: User Testing Consent

Informed Consent Agreement

For Participation in a Research Study: Software Testing

Project Team: Gabrielle Acquista, Victoria Buyck, Benjamin Sakac, Sohrob Yaghouti

Contact Information: gr-ggcmqp-team@wpi.edu

Title of Research Study

Implementation of Cloud-Based Enterprise Resource Planning Software for Gompei's Goat Cheese

Introduction

You are being asked to participate in a study researching the effectiveness of a prototype ERP system for Gompei's Goat Cheese. Before you agree, however, you must be fully informed about the purpose of the study, the procedures to be followed, and any benefits, risks or discomfort that you may experience as a result of your participation. This form presents information about the study so that you may make a fully informed decision regarding your participation.

Study Purpose

The purpose of the study is to assess the usability and accessibility of a prototype ERP system for Gompei's Goat Cheese, how well it handles needed functionality, and if there are any missing features. The information gained from this study will inform decisions made during the implementation of the ERP system.

Study Procedure

You will participate in a 20-30 minute talk-aloud session in which you will interact with a prototype system with the goal of completing a list of tasks.. As you interact with the system we ask that you try to speak out loud your thoughts as you navigate through the application. Once all the tasks have been completed, a member of the team will ask you a few additional questions about your thoughts on the prototype.

Risks to Study Participants

There are no foreseeable risks or discomfort to you.

Benefits to Research Participants and Others

This study will inform the development of a system that will greatly benefit the operations of Gompei's Goat Cheese. This system will improve the internal operations of GGC and communication with its supplier, Westfield Farm.

Confidentiality

No participants will be identified by name in this study, only by title in relation to GGC if we have your permission. Your comments will not be shared with anyone other than our project team. Constructive criticism is encouraged and will not negatively affect your relationship with GGC or Westfield Farm.

Voluntary Nature of Study

Your refusal to participate will not result in any penalty to you. If you feel uncomfortable once you begin the study, you may stop participating at any time.

By signing below, you acknowledge that you have been informed about and consent to be a participant in the study described above. Make sure that your questions are answered to your satisfaction before signing. You are entitled to retain a copy of this consent agreement.

Study Participant Signature

Date

Study Participant Name (Please print)

Signature of Person who explained this study

Appendix B: User Testing Protocol

Introduction

Hi, my name is [name]. I will be walking you through the test of our proposed operations system for GGC. This testing session will be in a think-aloud format today, which you will learn more about shortly. My group members [names] are also on the line and will be taking notes during our session.

Study Purpose

The purpose of this study is to identify the areas where the proposed operations system for GGC performs well, and where there is room for improvement. This information will be used to inform the revisions of the cloud-based enterprise resource planning system.

Consent Form and Incentive

Before we start, we would like to go through important parts of the interview information document ([Software Test Consent](#)) that was sent to you via email earlier.

This session will be recorded and transcribed for the purpose of analysis only. We would like to have the option to use your responses in our final report. Again, this would mean that we would identify you by your title in relation to Gompei's Goat Cheese. Are you comfortable with this?

(Participant gives oral consent)

Do you have any questions about the testing session today?

(Participant asks questions if they have any)

Are you comfortable with the study procedures and ready to move forward?

(Participant gives oral consent)

Interview Time

The testing and follow up questions will last about 30 minutes. Despite us calling this a testing session, there are no right or wrong answers. We are testing the system, not you. In this think-aloud session, you will be asked to complete a series of tasks within this new system. These tasks will be given to you one by one for you to complete in your own time. We recommend you voice your thought process verbally as you complete the tasks. We would like to see how you naturally navigate and use the system and will not give any guidance unless asked. We ask that you please give your best attempt at the task before asking for help. If at any time you feel uncomfortable and wish to stop the interview, please let us know. After the testing session, you will be asked a few questions and to fill out a quick post-testing form.

Are you ready to start?

(Participant gives oral consent)

Great. Let's get started. We will start screen and voice recording now.

remind participant to keep talking aloud as needed during the tasks

Tasks:

1. Log into the portal
2. View orders
3. View order information for customer Chris
4. Edit Chris's order information by updating the gift message
5. View invoices
6. Notify farm of orders missing invoice number
7. Search Orders using one of the available search fields
8. Add a new order
9. Delete the order you just created

10. Logout

*Section I will be asked verbally to the participant after their interaction with the new system.

All other sections will be recorded using a Google Form*

Section I: System Reflection

1. What stands out most to you?
2. What did you LIKE about the new system?
3. What did you DISLIKE about the new system?
4. How would you describe the navigation of the system?
5. What are your thoughts about the design of the system?
6. Do you think the information was displayed in an effective way that's easily readable?
(why/why not)
7. What do you think about the “dashboard” feature?
8. Which system do you prefer?
 - a. Old (email, google sheets)
 - b. New (cloud-based ERP)
2. Are there any features you would like to see implemented?
3. Are there any other suggestions for improvement?

Section II: Demographics / Other info (our use only)

1. Name, age
2. Title in relationship to GGC

3. On a scale of 1-5 how comfortable are you with using technology? (1 being not comfortable at all and 5 being very comfortable)
4. On a scale of 1-5 how comfortable are you with adapting to new systems? (1 being not comfortable at all and 5 being very comfortable)

Section III: System Usability Scale (SUS) Statements About The New System

Respond to each statement below selecting from a range of Strongly Disagree to Strongly Agree (with 5 = Strongly Agree and 1 = Strongly Disagree).

1. I think that I would like to use G.O.A.T.S. frequently.
2. I found G.O.A.T.S. unnecessarily complex.
3. I thought G.O.A.T.S. was easy to use.
4. I think that I would need the support of a technical person to be able to use G.O.A.T.S.
5. I found the various functions in G.O.A.T.S. were well integrated.
6. I thought there was too much inconsistency in G.O.A.T.S.
7. I imagine that most people would learn to use G.O.A.T.S. very quickly.
8. I found G.O.A.T.S. very awkward to use.
9. I felt very confident using G.O.A.T.S.
10. I needed to learn a lot of things before I could get going with G.O.A.T.S.

Section IV: Conclusion

1. Please write any other comments you have (about the systems, the study, or anything on your mind from this testing process!)
 - a. Long answer response

Appendix C: User Testing Data Analysis Sheet

For our user testing analysis, the team gathered all of the participant data in an Excel sheet to analyze any common themes and trends both quantitatively and qualitatively.

The ReadMe file of this document outlines the purpose of this study, and identifies the codes we used in the subsequent sheets of this analysis. The SUS questions are standard system usability questions to identify the usability of a general system. This alone does not give us information about whether the system is functional or not.

Objective:			
Identify the areas where the proposed operations system for GGC performs well, and where there is room for improvement.			
Researchers: Nini Acquista, Victoria Buyck, Ben Sakac, Sohrob Yaghouti			
Conducted via Zoom from 11/10/22 - 11/20/22			
Description for Raw Quantatative Data			
SUS (System Usability Score, 0-100, obtained from the following items on a 5-point scale)			
5 point likert scale, 1= strongly disagree, 5 =strongly agree			
Bangor, A., Kortum, P., & Miller, J. (2009). Determining what individual SUS scores mean: Adding an adjective rating scale. <i>Journal of usability studies</i> , 4(3), 114-123.			
https://uxpajournal.org/determining-what-individual-sus-scores-mean-adding-an-adjective-rating-scale/			
SUS1	I think that I would like to use G.O.A.T.S.	ODD # questions:	Positively Phrased
SUS2	I found G.O.A.T.S. unnecessarily complex.	EVEN # questions:	Negatively Phrased
SUS3	I thought G.O.A.T.S. was easy to use.		
SUS4	I think that I would need the support of a technical person to be able to use G.O.A.T.S.		
SUS5	I found the various functions in G.O.A.T.S. were well integrated.		
SUS6	I thought there was too much inconsistency in G.O.A.T.S.		
SUS7	I would image that most people would learn to use G.O.A.T.S. very quickly.		
SUS8	I found G.O.A.T.S. very awkward to use.		
SUS9	I felt very confident using the G.O.A.T.S.		
SUS10	I needed to learn a lot of things before I could get going with G.O.A.T.S.		

The ReadMe also identifies the questions asked in the interview as well as learning objectives we hoped to reach by asking these questions. The individual questions are mapped to one or more target points.

Description for Raw Qualitative Data

PID	Participants ID						
TP	Target Points						
Task ID	Task						
T1	Log into the portal						
T2	View orders						
T3	View order information for customer Chris						
T4	Edit Chris's order information by updating the gift message						
T5	View invoices						
T6	Notify farm of orders missing invoice number						
T7	Search Orders using one of the available search fields						
T8	Add a new order						
T9	Delete the order you just created						
T10	Logout						
Question ID	Question					TP	
Q1	What stands out most to you?					TP1S	
Q2	What did you LIKE about the new system?					TP1S, TP6A	
Q3	What did you DISLIKE about the new system?					TP1S, TP2U, TP5V	
Q4	How would you describe the navigation of the system?					TP4L	
Q5	What are your thoughts about the design of the system?					TP3D	
Q6	Do you think the information was displayed in an effective way that's easily readable? (why/why not)					TP4L	
Q7	What do you think about the "dashboard" feature?					TP2U, TP4L	
Q8	Which system do you prefer? Old (email, google sheets) vs New (cloud-based ERP)					TP1S TP2U	
Q9	Are there any features you would like to see implemented?					TP5V	
Q10	Are there any other suggestions for improvement?					TP5V	
TP ID	Learning Objectives					Questions	
TP1S	Impressions: What is the main takeaway from this system?					Q1, Q2, Q3, Q8	
TP2U	Useful: Does the system meet all of the user's needs?					Q3, Q7, Q8	
TP3D	Design: Is the design clear and effective or does it distract the user?					Q5	
TP4L	Layout: Is the flow of the system and displaying of information intuitive for the user?					Q4, Q6, Q7	
TP5V	Improvements: What additions or features can be added to the system?					Q3, Q9, Q10	
TP6A	Acceptable: What does the system already do well?					Q2	

This sheet shows the calculation of SUS scores by participant as well as the SUS score average of all participants. Additionally we included the average score by each of the SUS questions. To calculate this score we used a SUS data template to ensure no mistakes were made.

Participant ID	SUS1	SUS2	SUS3	SUS4	SUS5	SUS6	SUS7	SUS8	SUS9	SUS10	SUS Score
p1	5	1	4	1	4	1	5	1	5	1	95.0
p2	5	1	5	1	5	1	4	1	5	1	97.5
p3	4	1	5	1	5	1	4	1	5	1	95.0
p4	4	1	5	1	5	1	4	1	5	2	92.5
Mean	4.50	1.00	4.75	1.00	4.75	1.00	4.25	1.00	5.00	1.25	95.00
SD	0.58	0.00	0.50	0.00	0.50	0.00	0.50	0.00	0.00	0.50	2.04
SUS ID	question	mean by question									
SUS1	I think th	4.5									
SUS3	I thought	4.8									
SUS5	I found tl	4.8									
SUS7	I would ii	4.3									
SUS9	I felt ver	5.0									
SUS2	I found G	1.0									
SUS4	I think th	1.0									
SUS6	I thought	1.0									
SUS8	I found G	1.0									
SUS10	I needed	1.3									

For each participant during the completion of each task, the participant identified whether the task was completed successfully, if the participant had any trouble with the tasks, and jotted down any comments the participant made during each task. The green boxes identify successful completion of the task with no issues or redirections while the yellow boxes identify successfully completed tasks with minor mistakes or if the participant got stuck. The team also recorded participant responses to the interview questions in this sheet and shows the mapping of target points to questions.

Task Notes										
Participant ID	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
p1		Used dash Confused ;			What's the first went t Confused a	Confused a		Used dash! Save should	No date of	
p2				trouble fin		"looks like	searched b			
p3		used dash		trouble fin						
p4		used dash		trouble fin				Trouble fin -make butt		
Question Responses										
Participant ID	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
p1	Visually ap	Allows for	More info It doesnt w	Straightfor	Simple, att	Easily read	I like it, it v	The new sy	Accounting	Different p
p2	Simplicity	Ease of use	References	Pretty goo	Its very flu	Yes, the qu	Pretty goo	NEWWVV	Data visual	Statistics a
p3	The interfa	clear, orga	---	easy, self e	liked the d	yes, easy to	non-memc	Its hard to	-Sticker lab	- Notify for
p4	clean feel,	good visua	Wish the c	easy, flowe	Likes it, like	-Once you	Nice and b	New - Easy	-in person	- no max lc
p5									- bob View	- Invoice ai
									- Account s	- bob conti
									- Holiday C	- bulk orde
TP1S	Q1, Q2, Q3, Q8									
TP2U	Q3, Q7, Q8									
TP3D	Q5									
TP4L	Q4, Q6, Q7									
TP5V	Q3, Q9, Q10									
TP6A	Q2									

From there, the team identified the general thoughts and feelings of each participant in regards to the target points. These conclusions were made based on the responses to the respective questions. For example, to understand participant one’s impressions of the system (target point 1), the team looked at the participants responses for questions 1, 2, 3, and 8. We also identified any positive or negative comments that the user made on the system.

Participant ID	TP1S	TP2U	TP3D	TP4L	TP5V	TP6A	OTHER
	Impressions	Useful	Design	Layout	Improvements	Acceptable	
p1	UI Design/Positive	No	Positive	Neutral	Yes; Format, Accounting, Statistics	Easy; Less Error	
p2	UI Design	No	Positive	Neutral	Yes; Accounting, Statistics	Easy; Convenient	
p3	UI Design/Positive	Somewhat	Neutral	Neutral	Yes; Orders, Format, Accounting	Easy; Convenient; Less Error	
p4	UI Design	Yes	Positive	Positive	Yes; Orders	Easy; Convenient	
	100% first impression UI Design				100% suggested improvements	100% stated the current system is easy to use	
TP1S	Q1, Q2, Q3, Q8						
TP2U	Q3, Q7, Q8						
TP3D	Q5						
TP4L	Q4, Q6, Q7						
TP5V	Q3, Q9, Q10						
TP6A	Q2						
Participant ID	Positive	Negative					
p1	Easy, Simple, Attractive	info displayed					
p2	Clean, Easy, fluid, useful	Lacking order stats/ data visualization					
p3	Clean, Easy, convenient, useful	Info displayed, lacking features					
p4	Clean, Easy, Convenient						

Finally, we identified the insights we got from this analysis. The SUS Items by Average Score chart shows the average score by question. Since the odd numbered questions were positively phrased (best being 5), and the even numbered questions were negatively phrased (best being 1), by the participant responses we can conclude the system is acceptable in terms of usability. Additionally the responses to what the system does well (acceptability) and what needs improvement are shown in a bar chart.



This study had 5 participants

SUS:

- Positively phrased questions received average scores ranging from 4 - 5
- Negatively phrased questions received average scores ranging from 1 - 1.3

Positive Feedback:

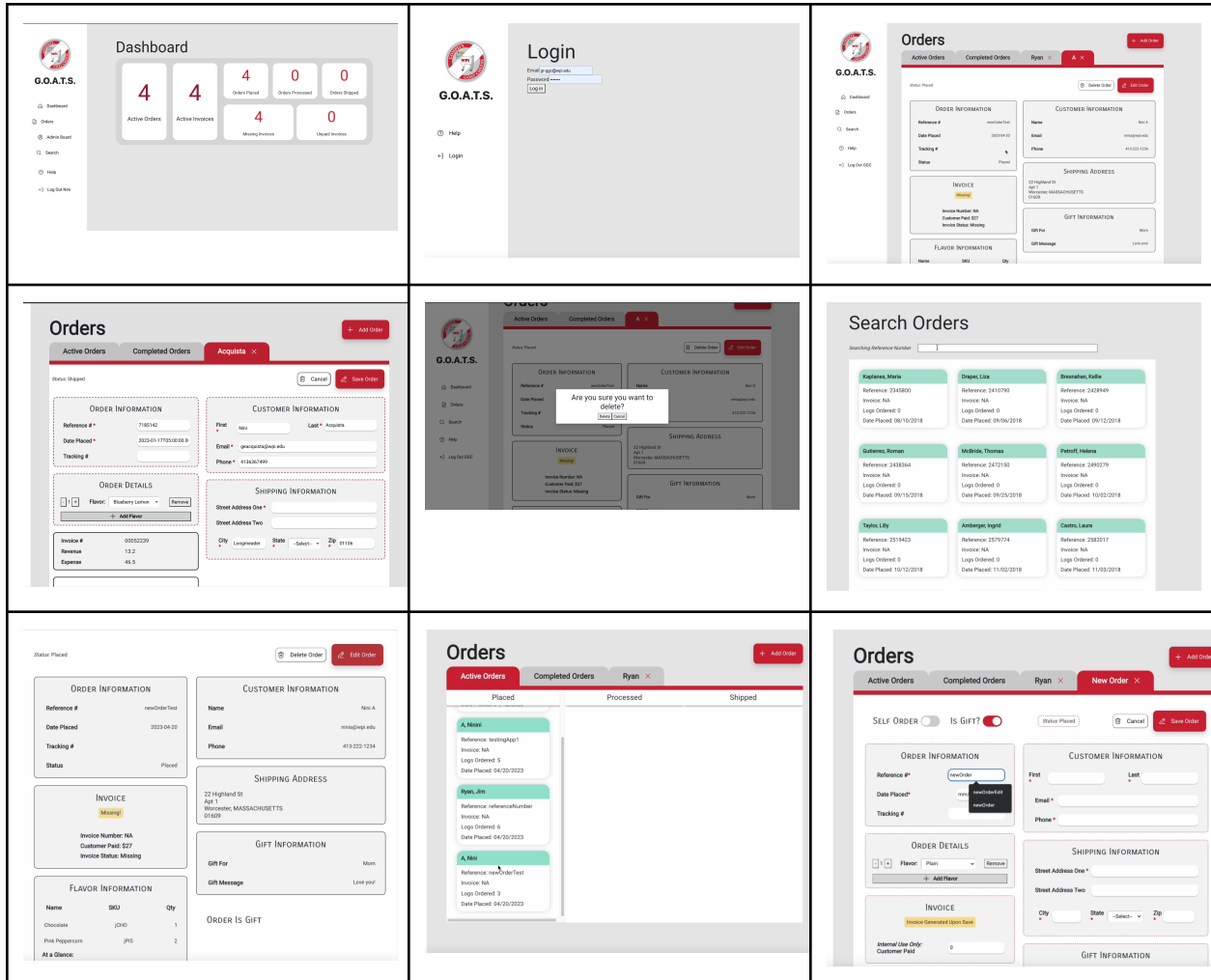
- 100% of participants stated this system is easy to use
- 100% of participants first impressions were about the systems UI

Improvements:

- 100% of participants gave improvement suggestions
- There was confusion about the difference between invoices and orders
- Order cards need more information
- Add permissions for different roles (Bob, GGC, admin, read-only)
- Add ability for in person orders

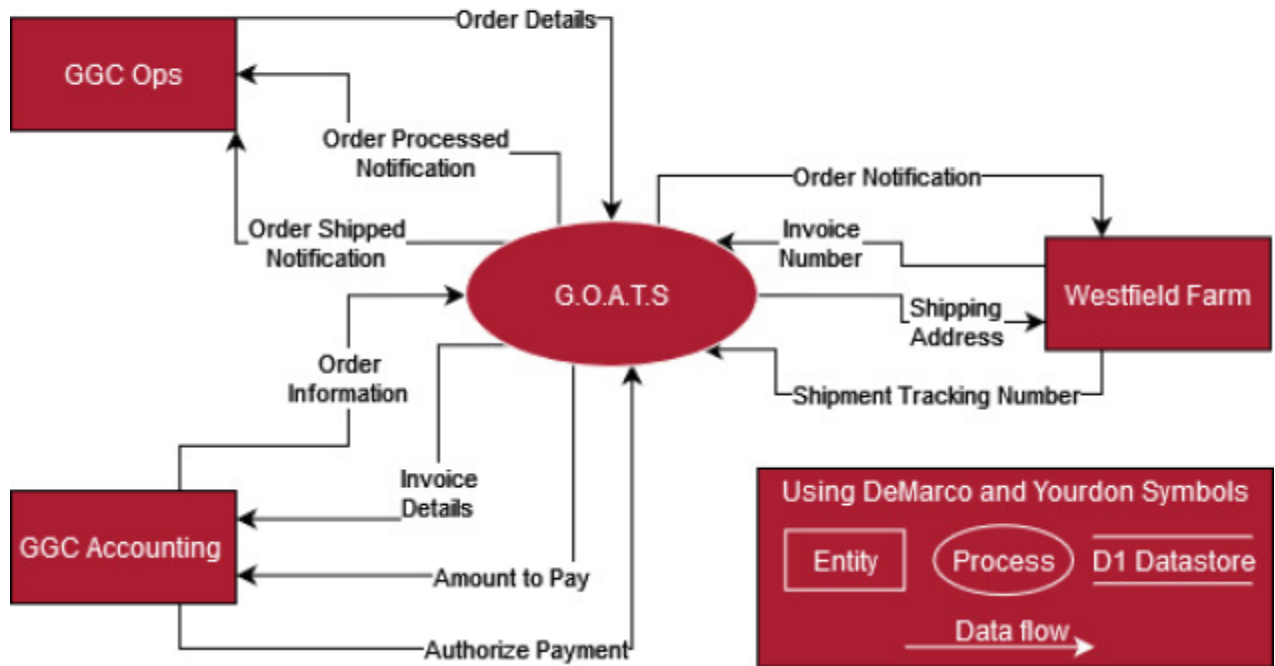


Appendix D: Functional Prototype Images



Appendix E: Context Diagram

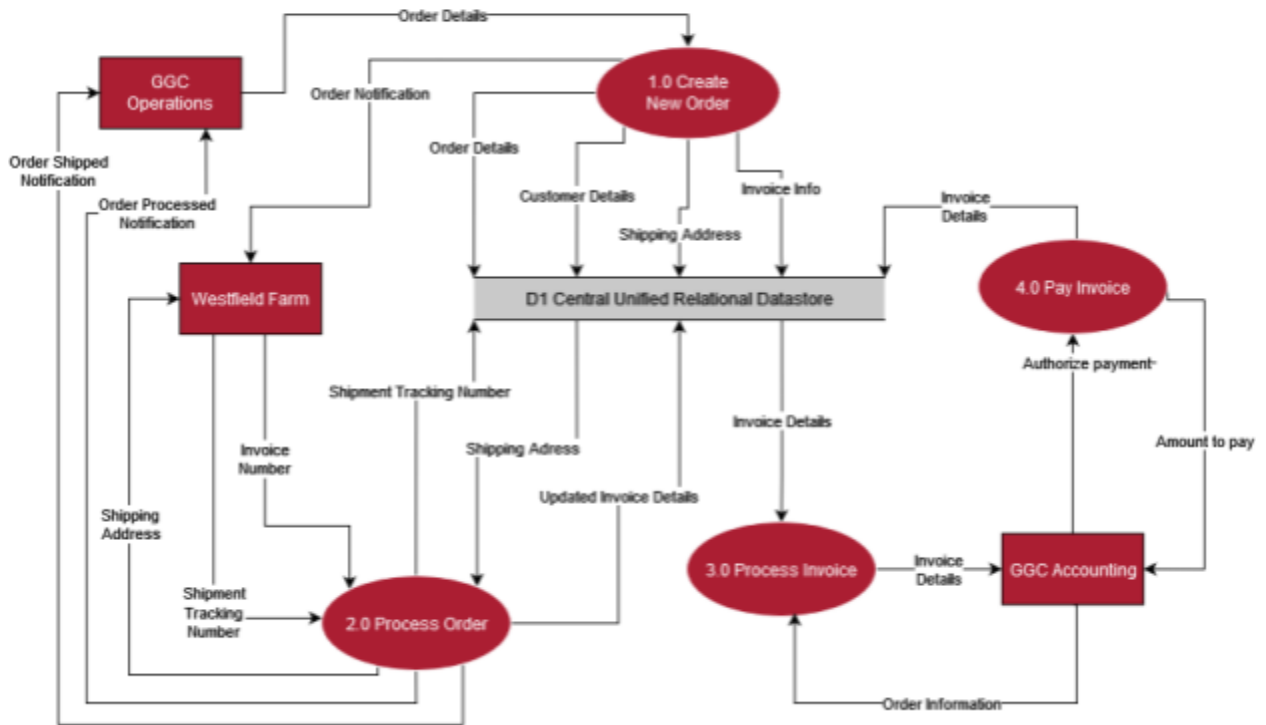
Context Diagram



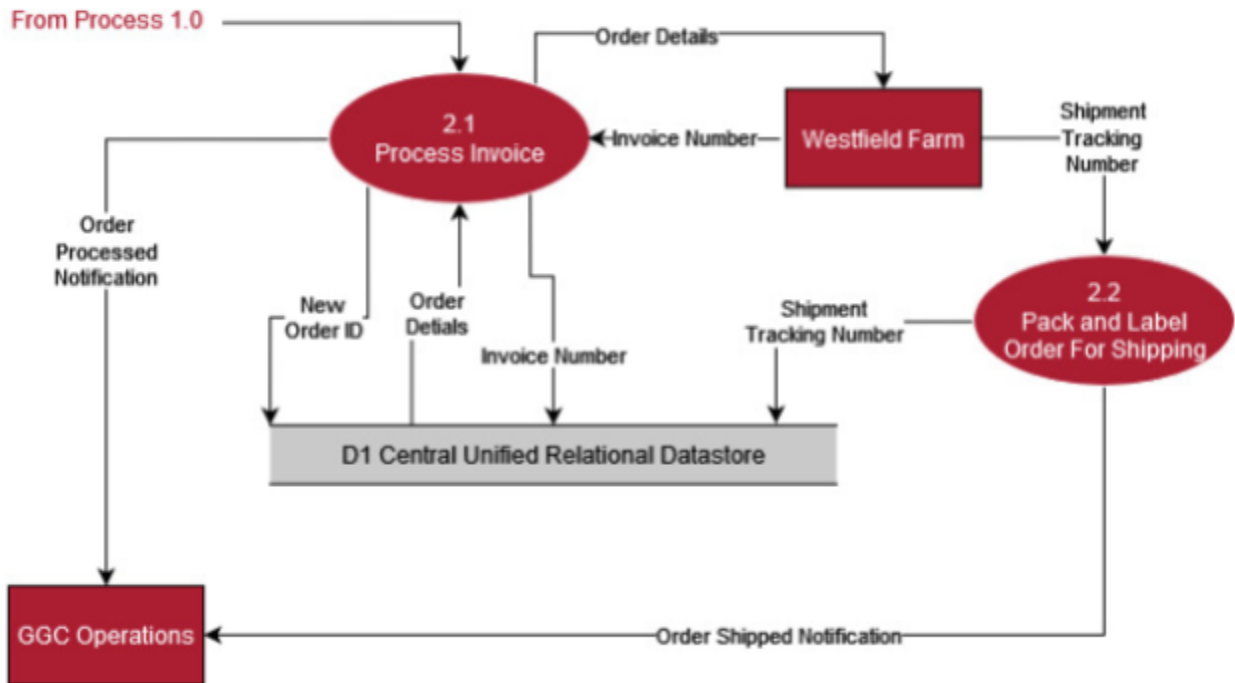
Appendix F: Data Flow Diagrams

Each of these diagrams are the updated data flow diagrams our team developed

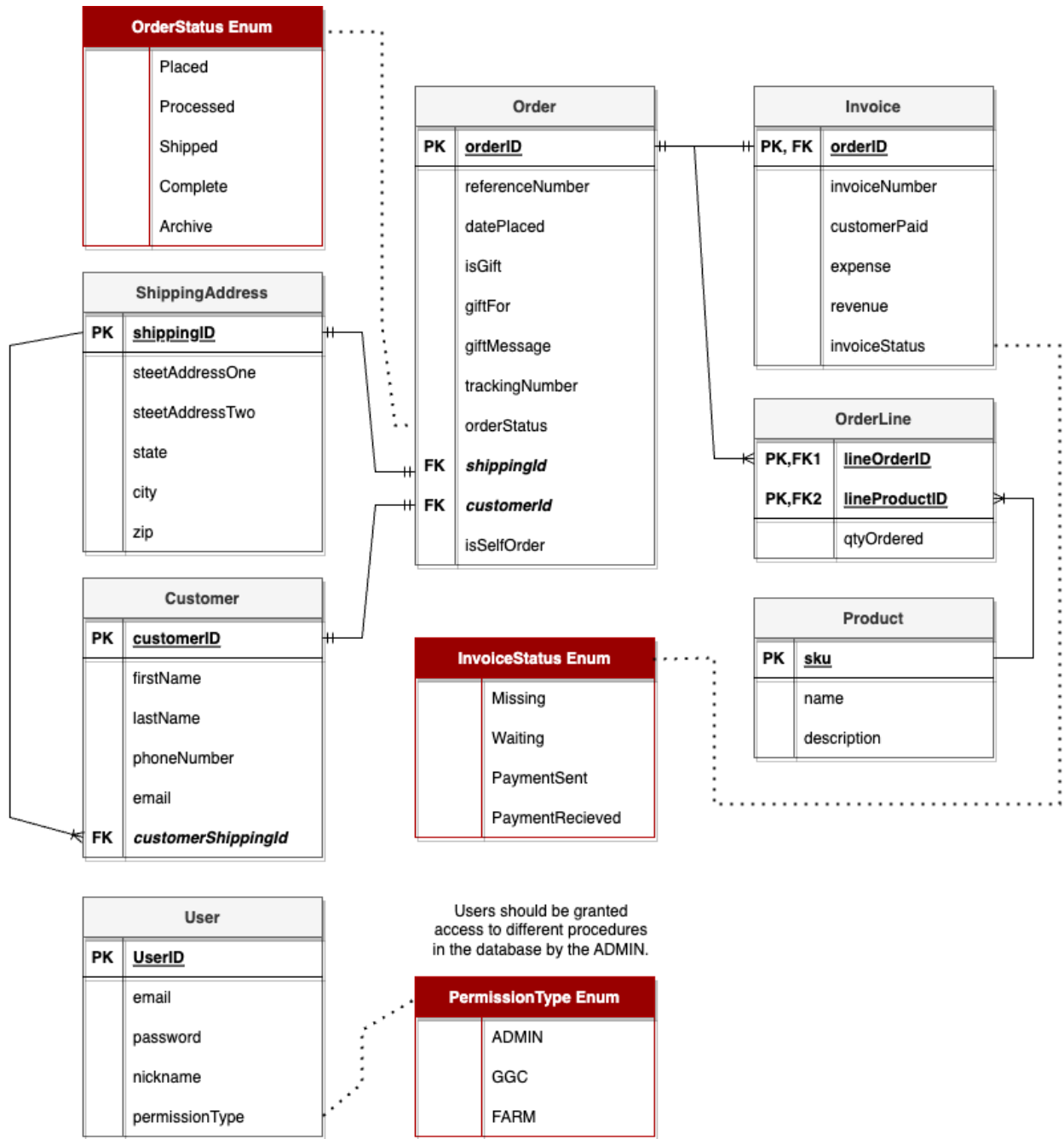
Level 0 Data Flow Diagram

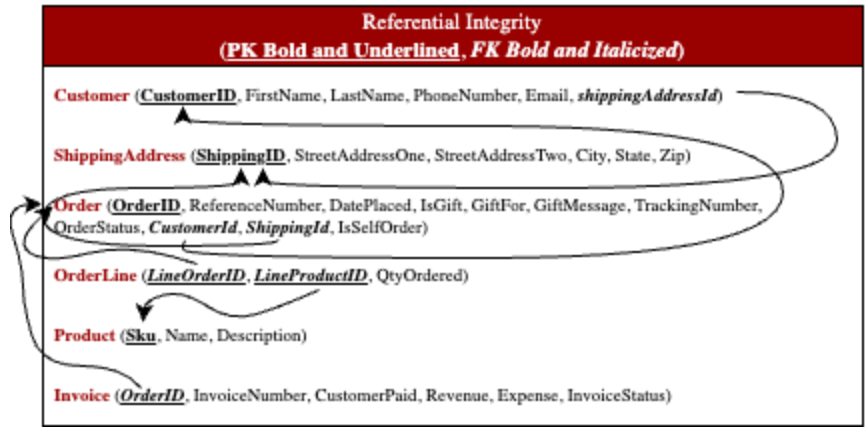
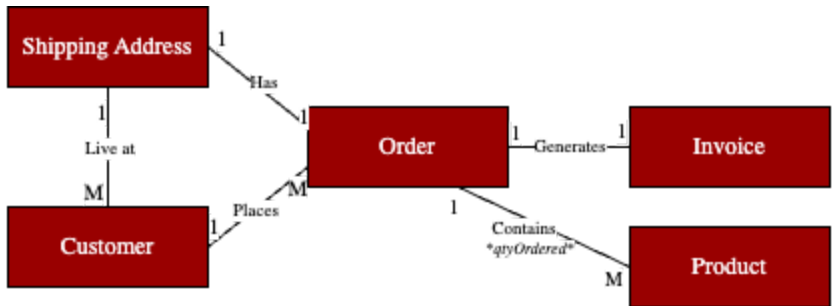


Level 1 DFD Fragment: Process 2



Appendix G: Entity Relationship Diagrams





Appendix H: Use Cases

Add User	
Participating Actors	GGC User
Entry Conditions	<ol style="list-style-type: none"> 1. User is a GGC User. 2. Email is a real email.
Exit Criteria	The email is verified for system login. User has a role (GGC or Westfield Farm)
Flow of Events	<ol style="list-style-type: none"> 1. GGC User requests to add a user. 2. GGC User types in a new User's email and submits a request. 3. System adds email to a list of verified login emails, refreshes display.

Remove User	
Participating Actors	GGC User
Entry Conditions	<ul style="list-style-type: none"> • User is selected.
Exit Criteria	The email is no longer verified for system login.
Flow of Events	<ol style="list-style-type: none"> 1. GGC User requests to remove a verified email. 2. System prompts GGC User to confirm removal. 3. GGC User confirms removal. 4. System removed email from the list of verified email logins, refreshes display.

Log In	
Participating Actors	User
Entry Conditions	User's email is in the system.
Exit Criteria	User is logged in.
Flow of Events	<ol style="list-style-type: none"> 1. User requests to log in. 2. System verifies user email and displays GGC's dashboard.

Log Out	
Participating Actors	User
Entry Conditions	User is logged in.
Exit Criteria	User is logged out.
Flow of Events	<ol style="list-style-type: none"> 1. User requests to log out. 2. System logs out User and shows the login screen.

Record New Order	
Participating Actors	GGC User
Entry Conditions	All required order information has been put in.
Exit Criteria	Order has been recorded and an email is sent about the new order.

Flow of Events	<ol style="list-style-type: none"> 1. GGC requests to add an order. 2. System adds order to the system and refreshes the display. 3. System sends an email to GGC and the Partner Farm that a new order has been added.
----------------	--

Delete Order	
Participating Actors	GGC User
Entry Conditions	An order is selected.
Exit Criteria	The selected order and associated invoice is deleted.
Flow of Events	<ol style="list-style-type: none"> 1. GGC requests to delete an order. 2. System prompts GGC to confirm that they want to delete the order. 3. GGC responds to prompt on screen. 4. System deletes the order if GGC confirms the prompt and refreshes the screen.

Update Order Information	
Participating Actors	User
Entry Conditions	Order Information is being viewed and new information has been put in.
Exit Criteria	Order Information is updated.

Flow of Events	<ol style="list-style-type: none"> 1. User requests to update order information. 2. The system updates the order information and refreshes the screen.
----------------	--

Update Invoice Information	
Participating Actors	User
Entry Conditions	Invoice Information is being viewed and new information has been put in.
Exit Criteria	Order Information is updated.
Flow of Events	<ol style="list-style-type: none"> 3. User requests to update order information. 4. The system updates the order information and refreshes the screen.

View Active Orders	
Participating Actors	User
Entry Conditions	None
Exit Criteria	The current orders are displayed.
Flow of Events	<ol style="list-style-type: none"> 1. User requests to view the current active orders. 2. The system displays the current active orders.

View Completed Orders

Participating Actors	User
Entry Conditions	None
Exit Criteria	All the orders are displayed.
Flow of Events	<ol style="list-style-type: none"> 1. User requests to view completed orders. 2. The system displays the completed orders.

View Order Information	
Participating Actors	User
Entry Conditions	Order is selected.
Exit Criteria	The order information is displayed.
Flow of Events	<ol style="list-style-type: none"> 1. User requests that the order information be displayed. 2. The system displays the order information.

View Invoice	
Participating Actors	User
Entry Conditions	Order is selected and has an invoice.
Exit Criteria	Invoice PDF is displayed.
Flow of Events	<ol style="list-style-type: none"> 1. User requests to view invoice.

	2. System displays Invoice PDF in a new window.
--	---

Search Orders	
Participating Actors	User
Entry Conditions	Search parameters are specified.
Exit Criteria	Orders matching search parameters are displayed.
Flow of Events	<ol style="list-style-type: none"> 1. User requests to search for orders matching a set of parameters. 2. The system displays all orders matching the search parameters.

View Customers	
Participating Actors	GGC User
Entry Conditions	None
Exit Criteria	List of customers is displayed
Flow of Events	<ol style="list-style-type: none"> 1. GGC requests to view the list of customers 2. The system displays all orders matching the search parameters.

Copy Text Information

Participating Actors	User
Entry Conditions	None
Exit Criteria	Text is copied to clipboard
Flow of Events	<ol style="list-style-type: none"> 1. User requests to copy nearby text. 2. The system copies that text to the users clipboard.

Information Icon	
Participating Actors	User
Entry Conditions	<ol style="list-style-type: none"> 1. Hover mouse over information icon next to site function
Exit Criteria	Brief explanation of function and its uses
Flow of Events	<ol style="list-style-type: none"> 1. User hovers over or clicks information icon 2. Small popup shows explanation of site function and its uses

Appendix I: Data Dictionary

CUSTOMER

Attribute	Type	Constraint	Description
CustomerID	ID	Primary Key	Automatically generated ID
FirstName	String	Not Null	Customer's first name
LastName	String	Not Null	Customer's last name
PhoneNumber	String		Customer's primary phone number
Email	String	Not Null	Customer's primary email
ShippingID		Foreign Key, References SHIPPING ADDRESS	Associated ShippingID if the order is not a gift, default NULL

SHIPPING ADDRESS

Attribute	Type	Constraint	Description
ShippingID	ID	Primary Key	Automatically generated ID
StreetAddressOne	String	Not Null	

StreetAddressTwo	String		
City	String	Not Null	
State	String	Not Null	
Zip	String	Not Null	

ORDER

Attribute	Type	Constraint	Description
OrderID	ID	Primary Key	Automatically generated ID
ReferenceNumber	String		Reference Number (generated by GGC Formsite)
DatePlaced	Datetime	Not Null	Date the order was placed on.
IsGift	TinyInt		True if the order is a gift.
GiftFor	String		Name of the recipient of the gift (Full name / title to address the order)
GiftMessage	String		Optional gift message that customers can choose to send with their order.

TrackingNumber	String		When shipping the order, the farm receives a shipment tracking number from Stamps.com. This attribute tracks that number. When filled out, the order status changes to shipped.
OrderStatus	Enum		Current order status [Placed, Processed, Shipped, Complete, Archive]
ShippingID		Foreign Key, References SHIPPING ADDRESS	Associated shipping address (where to send the order)
CustomerID		Foreign Key, References CUSTOMER	ID of the customer who placed an order.
IsSelfOrder	TinyInt		True if the order is placed by GGC for pickup at the farm.

ORDER_STATUS ENUM

String	Context
Placed	Order is placed on Formsite or by GGC.

Processed	Order is sent to the farm.
Shipped	Order is shipped from the farm.
Complete	Order is received by the customer.
Archive	Order is over 3 months old.

ORDERLINE

Attribute	Type	Constraint	Description
LineOrderID	ID	Primary Key, References ORDER OrderID	Associated Order's ID
LineProductID	ID	Foreign Key, References PRODUCT Sku	Associated ID of one product in an order.
QtyOrdered	Int		Quantity ordered of the associated product.

PRODUCT

Attribute	Type	Constraint	Description
Sku	ID	Primary Key	Westfield Farm SKU Value or determined by ADMIN.
Name	String		Name of the cheese flavor or

			item.
Description	String		Description of the item.

INVOICE

Attribute	Type	Constraint	Description
OrderID	ID	Primary Key, References ORDER	Associated Order ID
InvoiceNumber	String		Westfield Farm invoice number.
Revenue	Float		Amount GGC made from the order (Revenue = CustomerPaid - Expense) *Note: this should be changed to "Profit"*
Expense	Float		Amount GGC owes the farm.
CustomerPaid	Float		Amount customer paid for the order (0 if its a selfOrder)
InvoiceStatus	Enum		Status of the invoice.

INVOICE_STATUS ENUM

String	Context
Missing	Invoice is missing an invoice number (default).
Waiting	Invoice number and expense is sent to GGC. This is the status when an InvoiceNumber is input to the database.
PaymentSent	Invoice payment has been sent by GGC Accounting.
PaymentRecieved	Invoice payment has been received by the farm.

USER

Attribute	Type	Constraint	Description
UserID	ID		Associated User ID, automatically incremented.
Nickname	String		Nickname for welcome message.
Email	String		User email
Password	String		User password
PermissionType	Enum		Users' permission for accessing/updating data.

PERMISSION_TYPE ENUM

String	Context
ADMIN	Users have administrator privileges.
GGC	Users can read/add/edit/delete orders and update invoice status to PaymentSent.
FARM	Users can read order information but ONLY edit the InvoiceNumber, Expense, and change InvoiceStatus to PaymentRecieved.

Appendix J: Package.json

(root)

```
{  
  
  "name": "ggcportal",  
  
  "version": "1.0.0",  
  
  "description": "",  
  
  "main": "server.js",  
  
  "scripts": {  
  
    "test": "echo \"Error: no test specified\" && exit 1",  
  
    "server": "nodemon --quiet server",  
  
    "start": "nodemon --exec babel-node server.js",  
  
    "client": "npm start --prefix client",  
  
    "build": "babel server.js --out-file server.compiled.js"  
  
  },  
  
  "repository": {  
  
    "type": "git",  
  
    "url": "git+https://github.com/geacquista/GGCPortal.git"  
  
  },  
  
  "keywords": [  
  
    "nodejs",  
  
    "express",  
  
    "mysql",  
  
    "restapi"  
  
  ],  
  
  "author": "niniacquista",  
  
}
```

```
"license": "ISC",

"dependencies": {

  "@babel/preset-react": "^7.18.6",

  "aws-chime": "git+ssh://git@github.com:aws/amazon-chime-sdk-js.git",

  "aws-sdk": "https://github.com/aws/aws-sdk-js.git",

  "bootstrap": "^5.2.3",

  "cors": "^2.8.5",

  "express": "^4.18.2",

  "mysql": "latest",

  "mysql2": "^3.2.0"

},

"bugs": {

  "url": "https://github.com/geacquista/GGCPortal/issues"

},

"homepage": "https://github.com/geacquista/GGCPortal",

"devDependencies": {

  "@babel/cli": "^7.21.0",

  "@babel/core": "^7.21.3",

  "@babel/node": "^7.20.7",

  "@babel/plugin-syntax-jsx": "^7.21.4",

  "@babel/preset-env": "^7.20.2",

  "nodemon": "^2.0.22"

},

"engines": {

  "node": "16.17.0"
```



```
}  
}
```

(client)

```
{  
  "name": "frontend_goats_portal",  
  "version": "0.1.0",  
  "private": true,  
  "dependencies": {  
    "@reduxjs/toolkit": "^1.9.1",  
    "@testing-library/jest-dom": "^5.16.5",  
    "@testing-library/react": "^13.4.0",  
    "@testing-library/user-event": "^14.4.3",  
    "axios": "^1.2.2",  
    "bootstrap": "^5.2.3",  
    "connected-react-router": "^6.9.3",  
    "date-fns": "^2.29.3",  
    "formik": "^2.2.9",  
    "history": "^5.3.0",  
    "moment": "^2.29.4",  
    "react": "^17.0.0",  
    "react-datepicker": "^4.8.0",  
    "react-dom": "^17.0.0",  
    "react-redux": "^7.1.0",  
    "react-router-dom": "^6.4.5",
```

```
"react-scripts": "^2.1.3",

"react-switch": "^7.0.0",

"redux": "^4.2.0",

"redux-thunk": "^2.4.2",

"web-vitals": "^2.1.4",

"yup": "^1.0.2"

},

"scripts": {

  "start": "react-scripts start",

  "build": "react-scripts build",

  "test": "react-scripts test",

  "eject": "react-scripts eject"

},

"eslintConfig": {

  "extends": [

    "react-app",

    "react-app/jest"

  ]

},

"browserslist": {

  "production": [

    ">0.2%",

    "not dead",

    "not op_mini all"

  ],

  "development": [
```

```
"last 1 chrome version",  
  
"last 1 firefox version",  
  
"last 1 safari version"  
  
]  
  
}  
  
}
```

Appendix K: MySQL Script

```
CREATE TABLE `GGC_WAREHOUSE_CLOUD`.`User` (  
  
  `userID` int NOT NULL AUTO_INCREMENT,  
  
  `email` varchar(45) NOT NULL,  
  
  `password` varchar(45) NOT NULL DEFAULT 'ggc@wpi',  
  
  `nickname` varchar(45) DEFAULT NULL,  
  
  `permissionType` enum('ADMIN','GGC','FARM') NOT NULL,  
  
  PRIMARY KEY (`userID`)  
  
) ;  
  
CREATE TABLE `GGC_WAREHOUSE_CLOUD`.`Product` (  
  
  `sku` varchar(15) NOT NULL,  
  
  `name` varchar(45) NOT NULL,  
  
  `description` varchar(1024) DEFAULT NULL,  
  
  PRIMARY KEY (`sku`),  
  
  UNIQUE KEY `sku_UNIQUE` (`sku`)  
  
) ;  
  
CREATE TABLE `GGC_WAREHOUSE_CLOUD`.`ShippingAddress` (  
  
  `shippingID` int(11) NOT NULL AUTO_INCREMENT,  
  
  `streetAddressOne` varchar(255) NOT NULL,  
  
  `streetAddressTwo` varchar(45) DEFAULT NULL,  
  
  `city` varchar(255) NOT NULL,  
  
  `state` varchar(45) NOT NULL,  
  
  `zip` char(5) NOT NULL,  
  
  PRIMARY KEY (`shippingID`)
```

```

) ;

CREATE TABLE `GGC_WAREHOUSE_CLOUD`.`Customer` (
  `customerID` int NOT NULL AUTO_INCREMENT,
  `firstName` varchar(45) NOT NULL,
  `lastName` varchar(45) NOT NULL,
  `phoneNumber` varchar(45) DEFAULT 'NA',
  `email` varchar(255) NOT NULL,
  `customerShippingId` int DEFAULT NULL,
  PRIMARY KEY (`customerID`),
  UNIQUE KEY `customerID_UNIQUE` (`customerID`),
  KEY `ShippingID_idx` (`customerShippingId`),
  CONSTRAINT `CustomerAddressID` FOREIGN KEY (`customerShippingId`) REFERENCES
`ShippingAddress` (`shippingID`)
);

CREATE TABLE `GGC_WAREHOUSE_CLOUD`.`Order` (
  `orderID` int NOT NULL AUTO_INCREMENT,
  `referenceNumber` varchar(45) DEFAULT NULL,
  `datePlaced` datetime NOT NULL,
  `isGift` tinyint NOT NULL DEFAULT '0',
  `giftFor` varchar(108) DEFAULT NULL,
  `giftMessage` varchar(1024) DEFAULT NULL,
  `trackingNumber` varchar(45) DEFAULT NULL,
  `orderStatus` enum('Placed','Processed','Shipped','Complete','Archive') NOT NULL
  DEFAULT 'Placed',
  `shippingId` int NOT NULL,

```

```

`customerID` int NOT NULL,

`isSelfOrder` tinyint DEFAULT '0',

PRIMARY KEY (`orderID`),

KEY `ShippingID_idx` (`shippingId`),

KEY `CustomerID_idx` (`customerID`),

KEY `OrderStatusID_idx` (`orderStatus`),

CONSTRAINT `CustomerID_Order_FK` FOREIGN KEY (`customerID`) REFERENCES `Customer`
(`customerID`) ON DELETE CASCADE ON UPDATE CASCADE,

CONSTRAINT `ShippingID_Order_FK` FOREIGN KEY (`shippingId`) REFERENCES
`ShippingAddress` (`shippingID`) ON DELETE CASCADE ON UPDATE CASCADE
) ;

CREATE TABLE `GGC_WAREHOUSE_CLOUD`.`Invoice` (

`orderID` int NOT NULL,

`invoiceNumber` varchar(45) DEFAULT NULL,

`customerPaid` float DEFAULT '0',

`revenue` float DEFAULT '0',

`expense` float DEFAULT '0',

`invoiceStatus` enum('Missing','Waiting','PaymentSent','PaymentRecieved') DEFAULT
'Missing',

PRIMARY KEY (`orderID`),

UNIQUE KEY `OrderID_UNIQUE` (`orderID`),

CONSTRAINT `ORDER_ID_INVOICE_FK` FOREIGN KEY (`orderID`) REFERENCES `Order`
(`orderID`) ON DELETE CASCADE ON UPDATE CASCADE
) ;

CREATE TABLE `GGC_WAREHOUSE_CLOUD`.`OrderLine` (

```

```

`lineOrderID` int NOT NULL,

`lineProductID` varchar(15) NOT NULL,

`qtyOrdered` int DEFAULT '0',

PRIMARY KEY (`lineOrderID`,`lineProductID`),

KEY `sku_idx` (`lineProductID`),

CONSTRAINT `LineOrderID_FK` FOREIGN KEY (`lineOrderID`) REFERENCES `Order`
(`orderID`) ON DELETE CASCADE ON UPDATE CASCADE,

CONSTRAINT `SKU_FK_LINE` FOREIGN KEY (`lineProductID`) REFERENCES `Product` (`sku`)
ON UPDATE CASCADE

) ;

-- AFTER IMPORTING ANY DATA IN CSV FILES, ADD THE TRIGGER TO AFTER INSERT ORDER

CREATE DEFINER = CURRENT_USER TRIGGER
`GGC_WAREHOUSE_CLOUD`.`Order_AFTER_INSERT_GENERATE_INVOICE` AFTER INSERT ON `Order`
FOR EACH ROW

BEGIN

insert into `GGC_WAREHOUSE_CLOUD`.`Invoice`(`orderID`) values (new.orderID);

END

```

Appendix L: Python Script

```
# Script to transform sample GGC order data from one csv into several
csv files
# representing tables in a database
# Author: Ben Sakac

# Field names of sample data:
# Date Received,Reference Number,"Name (last, first)",Email,Phone,
# Town/City,State,Zip,Quantity (logs),Gift?,Gift For,Sold For,
# Bought For,Flavor 1,Flavor 2,Flavor 3,Flavor 4,Flavor 5,Flavor
6,Flavor 7,
# Flavor 8,Flavor 9,Flavor 10,Flavor 11,Flavor 12,Flavor 13,Flavor
14,
# Flavor 15,Flavor 16,Flavor 17,Flavor 18,Flavor 19,Flavor 20,Flavor
21,
# Flavor 22,Flavor 23,Flavor 24,Flavor 25,Flavor 26,Flavor 27,Flavor
28,
# Flavor 29,Flavor 30,Flavor 31,Flavor 32

# Self Orders/Orders made by campus orgs or other external
organizations were removed from
# input csv and added into output CSVs manually with exception of
order reference # 5838657
# which needed first and last name to be manually corrected post
script running
# Gift? column also needs manual cleaning to ensure script will run
# Changing Gift? column to ensure Ys and Ns are all capitalized

# Module imports
import random
import pandas as pd

# final db tables column names
SHIPPING_FIELDS = ["shippingID", "streetAddressOne",
                  "streetAddressTwo", "city", "state", "zip"]
CUSTOMER_FIELDS = ["customerID", "firstName", "lastName",
                  "phoneNumber", "email", "customerShippingID"]
ORDER_FIELDS = ["orderID", "datePlaced", "isGift", "giftFor",
```



```

"giftMessage", "trackingNumber",
                "orderStatus", "shippingID", "customerID",
"referenceNumber"]
INVOICE_FIELDS = ["orderID", "invoiceNumber", "customerPaid",
"expense", "revenue", "invoiceStatus"]
ORDERLINE_FIELDS = ["lineOrderID", "lineProductID", "qtyOrdered"]

# products dictionary {Name: ID}
PRODUCTS = {
    "Plain": "jPL5",
    "Herb Garlic": "jHG5",
    "Hickory Smoked": "jSM4",
    "Blueberry Lemon": "jBL6",
    "Cranberry Orange": "jCRA",
    "Pink Peppercorn": "jPI5",
    "Fiery Fig": "jFF6",
    "Chive": "jCHI",
    "Calabrini": "jCA6",
    "Chocolate": "jCHO",
    "3LB Calabrini": "jCA3",
    "3LB Herb Garlic": "jHGB",
    "3LB Plain": "jPLB3",
    "Chive Capri": "jCHI",
    "Herb Capri": "jHE5",
    "Pepper Capri": "jPE5",
    "Wasabi": "jWA6",
    "Herb Garlic (8oz)": "jHG8"
}

# Input filename
raw_data = "ggc_sample.csv"
# Number of rows in csv
SRC_ROWS = 559

# Filenames for output files
SHIPPING = "output/shipping.csv"
CUSTOMER = "output/customer.csv"
ORDER = "output/order.csv"
INVOICE = "output/invoice.csv"

```

```

ORDERLINE = "output/orderline.csv"

# Create data frame from sample data
df_sample_data = pd.read_csv(raw_data)
# rename columns to match what they will be in DB tables
df_sample_data = df_sample_data.rename(columns={'Town/City': 'city',
                                                'State': 'state',
                                                'Zip': 'zip',
                                                'Email': 'email',
                                                'Phone':
'phoneNumber',
                                                'Gift?': 'isGift',
                                                "Gift For":
"giftFor",
                                                "Date Received":
"datePlaced",
                                                "Reference Number":
"referenceNumber",
                                                "Bought For":
"expense",
                                                "Sold For":
"revenue"})

# Create shipping table with default of address as WPI,
# All historical customers will have this as their address,
# A new order by that customer entered into the system will update
# their address
shipping_cols = ["city", "state", "zip",
                 "streetAddressOne", "streetAddressTwo",
                 "shippingID"]
df_shipping_table = pd.DataFrame(columns=shipping_cols)
default_address = {"city": "Worcester",
                  "state": "MA",
                  "zip": "01609",
                  "streetAddressOne": "100 Institute Rd.",
                  "streetAddressTwo": "",
                  "shippingID": "1"}
df_shipping_table.loc[len(df_shipping_table)] = default_address

```

```

# Split Last and First name into separate columns
customer_cols = ["Name (last, first)", "phone", "email"]
name_field = "Name (last, first)"
df_name_split = pd.DataFrame(df_sample_data[name_field].str.split(
    ", ").to_list(), columns=["Last", "First"])
df_sample_data['firstName'] = df_name_split['First']
df_sample_data['lastName'] = df_name_split['Last']
df_sample_data = df_sample_data.drop(name_field, axis=1)

CUST_FIELDS_NO_IDS = ["firstName", "lastName", "phoneNumber",
"email"]
# Create DF of only rows where gift is no
df_not_gift_orders = df_sample_data[df_sample_data['isGift'] == "N"]
# Drop duplicate based on email, keep last
df_not_gift_orders_customers = df_not_gift_orders.drop_duplicates(
    ['email'], ignore_index=True, keep="last")
# Left join to get shippingID for each customer
df_NGOC_shipID = df_not_gift_orders_customers.loc[:,
CUST_FIELDS_NO_IDS].copy()
df_NGOC_shipID['customerShippingID'] = 1

# Create DF of only rows where gift is yes
df_gift_orders = df_sample_data[df_sample_data['isGift'] == "Y"]
# Drop duplicate based on email, keep last
df_gift_orders_customers = df_gift_orders.drop_duplicates(
    ['email'], ignore_index=True, keep="last")
# Left join to get shippingID for each customer
df_GO_shipID = df_gift_orders.loc[:, CUST_FIELDS_NO_IDS].copy()
df_GO_shipID["customerShippingID"] = ""

# Isolate customers who have only placed gift orders
df_gift_only_customers =
df_gift_orders_customers[~df_gift_orders_customers['email'].isin(
    df_NGOC_shipID['email'])]

# Combine non gift with gift orders removing duplicates and keeping
shipping info from non gift
df_combine_NGOC_SID_GOC =
df_NGOC_shipID.combine_first(df_gift_orders_customers)

```

```

# Concat that with gift only customers to get final unique customer
list
df_uniq_cust = pd.concat(
    [df_combine_NGOC SID_GOC, df_gift_only_customers],
    ignore_index=True)
# Strip phone numbers of spaces and dashes
df_uniq_cust['phoneNumber'] =
df_uniq_cust["phoneNumber"].str.replace(
    r'\D', '')
# Create customerID
df_uniq_cust['customerID'] = pd.RangeIndex(1, len(df_uniq_cust) + 1)
# Swap isGift Y to 1 and N to 0
df_uniq_cust["isGift"] = df_uniq_cust["isGift"].replace({"Y": 1, "N":
0})
# Final unique customers data frame
df_unique_customers = df_uniq_cust.loc[:, CUSTOMER_FIELDS].copy()

# Order
order_cols = ["orderID", "datePlaced", "isGift", "giftFor",
"giftMessage",
    "trackingNumber", "orderStatus", "shippingID",
"customerID", "referenceNumber"]
# Merge sample data with unique customers on email
merged_df_orders_shipping_customers = df_sample_data.merge(
    df_unique_customers, on=["email"], how='left')
# Create orderID
merged_df_orders_shipping_customers["orderID"] = pd.RangeIndex(
    1, len(merged_df_orders_shipping_customers) + 1)
merged_df_orders_shipping_customers["trackingNumber"] = ""
merged_df_orders_shipping_customers["orderStatus"] = ""
merged_df_orders_shipping_customers["shippingID"] = 1
df_orders = merged_df_orders_shipping_customers
# Add gift message if order was gift
df_orders.loc[df_orders["isGift"] == "Y",
    "giftMessage"] = "This is a fake gift message!"
# Convert dates to datetime
df_orders["datePlaced"] = pd.to_datetime(df_orders['datePlaced'])
# Invoice
df_invoice = df_orders.copy()

```

```

df_invoice['invoiceNumber'] = ""
df_invoice['isPaid'] = 0
df_invoice['customerPaid'] = "PaymentRecieved"
df_invoice["invoiceStatus"] = "PaymentSent"

# Orderline
LIST_FLAVOR_FIELDS = ["Flavor 1", "Flavor 2", "Flavor 3", "Flavor 4",
"Flavor 5", "Flavor 6", "Flavor 7",
                        "Flavor 8", "Flavor 9", "Flavor 10", "Flavor
11", "Flavor 12", "Flavor 13", "Flavor 14",
                        "Flavor 15", "Flavor 16", "Flavor 17", "Flavor
18", "Flavor 19", "Flavor 20", "Flavor 21",
                        "Flavor 22", "Flavor 23", "Flavor 24", "Flavor
25", "Flavor 26", "Flavor 27", "Flavor 28",
                        "Flavor 29", "Flavor 30", "Flavor 31", "Flavor
32"]

# Create dictionary where key is orderID and value is list of
products ordered
dict_orders: dict[int, list[str]] = {k: [] for k in
df_orders["orderID"]}
for field in LIST_FLAVOR_FIELDS:
    for index, row in df_sample_data.iterrows():
        if type(row[field]) != float:
            dict_orders[index +
1].append(PRODUCTS[row[field].strip().title().replace("Lb", "LB")])
            # Also mess around with sample data and make sure there
are no typos

dict_product_qtys = {}
dict_order_line = {}

# Looping through orders
for (orderID, products_ordered) in dict_orders.items():
    # Looping through each item in list of products ordered
    for item in products_ordered:
        # Create dictionary where key is item and val is quantity
ordered
        dict_product_qtys[item] = products_ordered.count(item)

```

```
# Set orderline key orderID to value dict product keys
dict_order_line[orderID] = dict_product_qtys

df_orderline = pd.DataFrame(columns=ORDERLINE_FIELDS)
# Create final dataframe from orderline dictionary
for (orderID, prod_qtys) in dict_order_line.items():
    for prodID, prodQty in prod_qtys.items():
        df_orderline.loc[len(df_orderline)] = [orderID, prodID,
        prodQty]

# Output files
df_shipping_table[SHIPPING_FIELDS].to_csv(SHIPPING, index=False)
df_unique_customers[CUSTOMER_FIELDS].to_csv(CUSTOMER, index=False)
df_orders[ORDER_FIELDS].to_csv(ORDER, index=False)
df_invoice[INVOICE_FIELDS].to_csv(INVOICE, index=False)
df_orderline[ORDERLINE_FIELDS].to_csv(ORDERLINE, index=False)
print("DONE")
```

Appendix M: Support Documentation

Where can everything be found?

We created a Github project setup with an AWS CodePipeline used to build and deploy the app in an Elastic Beanstalk environment. The backend of the environment was configured to an Amazon RDS MySQL instance so the app could make use of environment variables. There is a cloned version of the codebase in Gompei's GitHub Organization and a new Amazon AWS account set up for the future team. There will be instructions and links to resources on how to set up the AWS CodePipeline and Elastic Beanstalk Environment in the **README.md**.

How to access information and accounts?



This section is removed for security reasons.

What now? Running the Codebase

In order to access the codebase, a developer needs a computer that can access GitHub and any preferred IDE to run the code. A user is able to access the codebase by following these steps. Further information can be found in the **README.md**.

1. Add a personal Github account to Gompei's Github (Organization)
 - a. Log into the GGC Admin Github account and add the account as a collaborator on the organization.
2. Clone the GGCPortal repository to the local machine.
3. Delete the .node_modules folders and package-lock.json files in root and client directories.
4. Run 'npm install' in the terminal of the client AND root directories
5. Run 'npm run build' in the terminal of the client AND root directories
 - a. This command creates the production build directory of the app, which is the code actually being deployed.
 - b. Running 'npm run build' in the root directory updates the server.compiled.js file, which creates a build of the app that can run on the Node.js version expected by Elastic Beanstalk.
6. Run 'npm run start' on root directory
7. Open browser to local <http://localhost:3001/>
 - a. Note: If you run 'npm run start' in the client directory, localhost:3000 will open the client app, but the root (Express server) needs to be running in order to see

information from the database. This is a good way to see your frontend changes automatically deployed, because you need to run 'npm run build' every time you want to see frontend changes running the root directory.

8. Success! You should see the G.O.A.T.S. login page.

What has been done?

Paper Documentation

In our paper, there are a lot of helpful diagrams and other documentation, including: System Use-Cases, Data Flow Diagrams, an Entity Relationship Diagram, a Data Dictionary, a User Flow Diagram, System Mockups, and a Software Architecture Overview.

Database Schema: GGC_WAREHOUSE_CLOUD

The database that this project uses is Amazon RDS MySQL database. This was chosen for various reasons outlined earlier in the paper.

To continue with using this type of database and API, a new instance must be created, connect to that endpoint, create schema 'GGC_WAREHOUSE_CLOUD' and run the MySQL script to create the backend tables. Import refresh csv files to their respective tables and then add the trigger into the order table.

The IAM account should have access to read the following database snapshot in the Phase 2 AWS account:

Snapshot Name:
[ggc-mqp-2-snapshot](#)

DB Instance:
ID removed for security reasons

Front-end Development:

The status of front-end development can be found in the README.

Looking for resources to begin understanding the frameworks used in this software?

There are many great resources out there but here's a list of some of the things that helped the team get started. If you're interested in an academic reading that involves Redux and React, I recommend [this](#) one :)

Figma:

[Figma Tutorial: Prototyping](#)

[Figma Tutorial: Interactive Components](#)

Front End:

[React.js Crash Course](#)

[Crash Course Github](#)

[Intro to React Tutorial](#)

[React Functional Components](#)

[React Class Components](#)

[Conditional Rendering](#)

[Conditional Rendering with Enum](#)

[Redux Toolkit](#)

[Toolkit Fundamentals Tutorial](#)

[CRUD with Redux Toolkit](#)

Back End:

[What is Express.js?](#)

[Express Routing](#)

[Axios Documentation](#)

[Rest API with Express and MySQL](#)

Version Control:

[Beginner's Guide to npm](#)

[Gitting Started](#)

[Git Cheat Sheet](#)

Amazon AWS:

[Getting Started](#)

[MySQL to Amazon RDS](#)

[AWS RDS Setup Example](#)

[Full Stack with Elastic Beanstalk](#)

[Elastic Beanstalk Deployment Tutorial](#)

The README.md

```
Hello! You are likely feeling incredibly overwhelmed looking at so much code. Take a deep breath!
```

There are quite a few bugs in the code that just weren't resolved when the time was up, so this README will try and point you where to start.

To login, use any of the email/password combinations in the user.csv file (or however you've configured the database).

1. Overview of the mess, (and what I learned throughout MQP):

- There are a mix of React classes and React functional components in the codebase, but going forward I would consider making everything components (Chat GPT is your friend).

- The reason I didn't do this was because of using mapStateToProps with Redux Toolkit, to investigate the alternative with functional components.

- The tab system in MainOrderCards_Tabs is buggy. It's essentially the remnants of the Phase One Prototype, so consider revamping the code or scratching it altogether with a new tab/windowing solution.

- Check the rendering of the OrderCards, it doesn't seem to be working quite right for the Search component

- You can probably see this in the screenshots in the final paper, but the Search Page shows 0 logs for every order card, which is obviously wrong.

- Also... the onclick function for those cards was never fixed, but don't focus on that until the tab system works

- Check on OrderLines, in ExistingOrder_Tab I disabled editing an orderline because it was deemed to be a bit more complicated than anticipated.

- An orderline has a dual primary key (the order id, and the product id), but we only create an order line for the products ordered.

- When a user then goes to edit the products ordered on an existing order, the front-end arraylist of ProductsOrdered doesn't care that a primary key changed.

- So, going to update the orderliness with the new list of products doesn't work.

- The solution was to delete each orderline of that order, and add the new ones, but it's getting a duplicate primary key error.

- All in all, consider just creating an integer PK to avoid any more issues with the dual key.

- The AdminPanel was the first thing that worked. I swear. It was the first thing that connected the front and backend. But at some point... one day... the users stopped appearing. I didn't look into it... Presentation day passed and I have quite literally run out of time.

```
- I think it'll be a good place to start and debug to understand:  
  1. how the components get information from the store -->  
  2. how the store gets information from the API -->  
  3. how the API calls the backend.  
  
- App.js contains the Navbar actually being used: Navbar.js has the correct styling  
-- it's from Phase One, but App.js provides the structure for user permissions and  
login authorization.  
  - It does not currently handle access tokens or password security but that  
should be fixed before put into production.  
  - The Routes are the possible pages of the application with their respective  
components.  
  - Above the list of routes is the navbar rendering. Depending on the type of  
user logged in will control which buttons are shown. The farm side routes are set up  
but the components are not made (literally copy and paste once you've made your edits  
to the order state)  
  - An Admin user has access to the Admin panel but they should also have their  
own order page to be able to update EVERY field.  
  - Finalizing the admin flow will allow the software to be integrated for  
company use.  
  
- Rework the Redux Toolkit slices, they were inherently based on the tables in the  
backend but that's not necessarily the best approach. The order slice should hold ALL  
of the information on an order, rather than holding the shippingId, customerId, etc  
(like the backend table would). The local state in the OrderCard and ExistingOrder_Tab  
should be populated much more simply than it is now -- fixing this will probably save  
you the trouble I went through and solve the frontend issues explained earlier (I  
started with populating the order cards, then add order, then existing, and... I could  
have avoided a lot of problems, hopefully this is something that can be fixed quickly  
if you are at all familiar with React)  
  - Keep the other slices available (customers, shipping addresses, etc. even  
keep a copy of the order_slice around while you work a new solution, it's gonna be  
tricky manipulating the data as its passed directly from the backend, but that's where  
most of the work needs to be done), but it's not necessary to be joining arrays and  
objects in the frontend business logic -- just create the frontend state EXACTLY  
how it'll be useful for you populating the UI... then go back and rework your queries or  
data services.  
  - Also, research best practices for having business logic in the slices, I  
think you can do more with them than I am currently doing. That might help with a lot  
of the frontend calculations.  
  
- Speaking of queries and data services, the API's need quite a bit of work.
```

- If you find it to be too much work, you can consider Amazon AWS REST API Gateway, but developing the Express server and APIs yourself is a great way to strengthen backend software engineering skills, and will help you feel a greater amount of control over how data is passed across the server.

- You may need to wrap the order form in a form element, because right now it functions as a form but the required conditions aren't enforced by anything.

- One flow that does work and should be used to conduct a user test is as follows: (it doesn't really matter that not all the information shows up/can be edited yet)

1. Login to the system as a regular GGC User
 - email: gr-ggc@wpi.edu
 - password: company
2. Navigate to the Orders tab
3. Click add order
4. Fill in the order information
 - The user has the option to clear the fields if they mess up
5. Save the new order.
6. Navigate to the Active Orders tab and click the newly created order.
7. In the existing order tab, click edit and change the reference number to something easy to remember.
8. Navigate to the Search page.
9. Type the reference number in the search bar and watch as your order appears.
10. Go back to the Orders screen (it would be convenient if the onclick worked for the card)
11. In the Active Orders tab, click your order and delete it.
12. Confirm delete and verify it was removed from the active orders page (this might not work, it might look like a different order was deleted... but if you refresh the screen the correct order was deleted... sorry but that's probably another bug to go fix :)

2. How do I navigate the codebase? (GGCPortal is the root directory) It's gonna take some time to get comfortable. Here's a cheat sheet:

- Configurations:
 1. (.ebextensions/) & (.elasticbeanstalk/) are directories for elastic beanstalk environment configurations. They probably won't need to change unless you're modifying how the app is being deployed to eb.
 2. (babel.config.js) has the babel configuration for the server. Look at Babel documentation for more clarification on what it does.
 3. (Procfile) is similar to a makefile, it allows the web app know which file to run the server on (which is server.compiled.js because it is the compiled version of our app)

```

    4. (package.json) manages root project dependencies, configuration, and scripts
- includes express and mysql configurations
    4. (client/package.json) manages client project dependencies, configuration,
and scripts - includes all of the React stuff irrelevant to server itself

- Notable "Entry" Files:
    1. (server.js) The main entry point of the Express.js server that defines and
configures the server, middleware, routes, and any other setup.
    - (server.compiled.js) is generated by running 'npm run build', this is
what the Procfile points to
    2. (client/src/index.js) this is the main entry point for the React app
    - imports necessary dependencies: react, react-dom, the main app component,
and provides the *store* to the component!
    - renders the app component into the DOM using the ReactDOM.render()
method.
    3. (client/src/App.js) START HERE!!!
    - the main app component! This is where I have login authorization
happening and create the routes for the navigation bar.
    4. (app/models/db.js) gets path to db configuration and exports the database
connection
    5. (client/src/http-common.js) this defines the Axios connection to fetch from
API
    6. (client/src/store_cfg.js) this is where the slices (from app/src/store) and
any middleware are combined into one reducer.

- (app/): home of the Express API stuff
    1. (config/) each file in here is configured its own db endpoint
    - for best practice with EB, use environment variables (secured.cnfg.js)
    2. (controllers/)
    - handle incoming requests, process the data, and return the response to
the client
    - one controller for each db model in addition to an auth controller
(verifying user) and a board controller (this is unused right now but you should play
with the routing based on user permission)
    3. (middleware/)
    - I don't use any of the middleware but this is where it should be defined.
    4. (models/)
    - These define the schema of the database table
    - They are the ORM representation in JS
    - This is where the queries are -- look into creating join queries (you'd
probably put it in the order model, but look into how to handle that)
    5. (routes/) Map the API requests to the correct place

```

- (client/src/): home of the React App
 1. (services/) Think of them as your routes from the frontend.
 - These are the Axios calls to your APIs depending on which service
 2. (assets/) fonts, images, style, util
 3. (store/) is where all of our slices are configured.
 - Go here to learn about the state of the application and how the framework works.
 4. (views/) All of the JS components and classes for the frontend

- (tables/):
 1. (sql scripts) all of the table creation scripts in MySQL including one master script and an order trigger after add.
 2. (csv files) backup from CSV files to use if necessary and to reference the data model in a real order.

- 3. You might have a few thoughts. One will be to completely start over, another will be to try and fix each thing at a time. Hopefully you don't start over, I tried to make this as modular as possible. My advice:
 - Please before anything else update the dependencies and versions. It might be beneficial to do more research into what is compatible with what but the big ones to look at are MySQL (used 5.7.41 in RDS) Node.js, React, Redux, etc (see the package.json file)

- 4. What the heck do I do about Elastic Beanstalk? Once you are able to use the AWS account created for you (and hopefully be a permanent account), set up a codepipeline from Github to an Elastic Beanstalk environment.
 - Create the EB Environment
 - Setup the CodePipeline from Github to the EB Environment

- Follow the steps here to guide you with EB and CodePipeline:
<https://www.honeybadger.io/blog/node-elastic-beanstalk/>

- Setup the database through the environment

- Follow the steps here to guide you with setting up a database in EB:
<https://docs.aws.amazon.com/elasticbeanstalk/latest/dg/using-features.managing.db.html>

- Follow the steps here

Author: Nini Acquista

Still confused and neeeded to bug an alumni with questions? geacquista@wpi.edu