Software Design: Management System (GotoGro-MRM) for Goto Grocery Inc.

Team: MSP_CL4_T1	
Name	**ID**
Enzo Peperkamp	*102895415*
Nelchael Kenshi Turija	*103057559*
Julian Codespoti	*102997816*
Alex Kyriacou	*103059830*
Marella Morad	*103076428*
Lachlan Martin	*103067448*

Introduction

As we conclude Sprint 2, this document serves as a continuation and evolution of our software design journey, which began with the "14P Software Design" report from Sprint 1. Over the course of this sprint, our team has made significant strides in advancing the software's design, building upon the foundational principles and structures established in Sprint 1. Recognizing the importance of consistency and clarity, we have maintained several conventions from our initial sprint, while also introducing enhancements in various areas to accommodate the new features and functionalities.

Throughout this report, references to our Sprint 1 decisions and designs will be made to highlight the progression and rationale behind the changes and improvements implemented in Sprint 2. Our commitment to creating a robust and scalable software system remains unwavering, and this document aims to provide a comprehensive overview of our design choices, their justifications, and their alignment with established software design principles.

Design of the Software Components

Updated Software Diagram

In our journey from Sprint 1 to Sprint 2, the software architecture has seen a natural evolution. While the foundational MVC structure remains consistent, several new components and interactions have been integrated to accommodate the advanced features introduced in this sprint.

Components and Their Roles

The updated software diagram introduces new components that play specific roles within our system:

Model Layer:

The foundational setup of Supabase/GCP from Sprint 1 has been retained in Sprint 2. This decision was grounded in the realization that our existing database and cloud infrastructure were already robust and versatile, seamlessly accommodating the influx of new items introduced this sprint.

Controller/View Layer:

The enhancements in this layer are driven by a focus on user experience (UX). New visual components and user interfaces have been seamlessly integrated, providing users with a richer, more intuitive experience. The website's overall structure has been reimagined to improve navigability and coherence. A testament to these changes is the users' newfound ability to directly access the following pages from the home page:

- Dashboards for Products and Sales
- Reports on Inventory and Sales

- Dedicated home sections for Products, Sales, and Members

CI/CD pipeline:

With the recent addition of a team member specialising in Quality Assurance (QA), our CI/CD pipeline has undergone significant refinements. While our cornerstone tools – GitHub for version control and Jira for sprint task management – remain essential to our development cycle, Sprint 2 saw an evolution in our approach to synchronising Jira with GitHub. The initial goal of this synchronisation was to automate several transitions, such as moving a card to "done" upon merging a pull request or marking an issue as "in progress" upon branch creation.

However, our experiences during this sprint highlighted that while such synchronisation can automate certain transitions, it does not fully capture the intricate nuances of our development process. For example, the mere act of creating a branch or a pull request doesn't necessarily reflect the comprehensive status of a Jira card. Tasks often involve phases like research, analysis, and discussions before coding even begins. Consequently, instead of maintaining strict synchronisation, we've chosen a more adaptable approach. We primarily use the integration to link pull requests directly to their respective Jira cards, ensuring a truer representation of a card's status, which in turn facilitates improved project management and clarity.

A notable enhancement in our CI/CD pipeline is the "Testing/Quality Assurance" segment. Utilising the Selenium WebDriver IDE, comprehensive usability tests have become standard. These tests mimic user behaviours on our platform, ensuring the system offers an intuitive and error-free user experience. The automation capabilities of this tool allow for consistent testing across various user paths, thereby validating functionality in multiple scenarios.

Furthermore, the Selenium WebDriver IDE is also adept at functional testing. We've crafted a series of test cases that assess specific functionalities, ensuring they operate as intended. These evaluations include monitoring data transactions, user input processes, and interactions between system modules.

The integration of this testing tool into our CI/CD pipeline has established a continuous feedback loop. Any code modification by developers triggers automated tests, reinforcing the software's ongoing readiness.

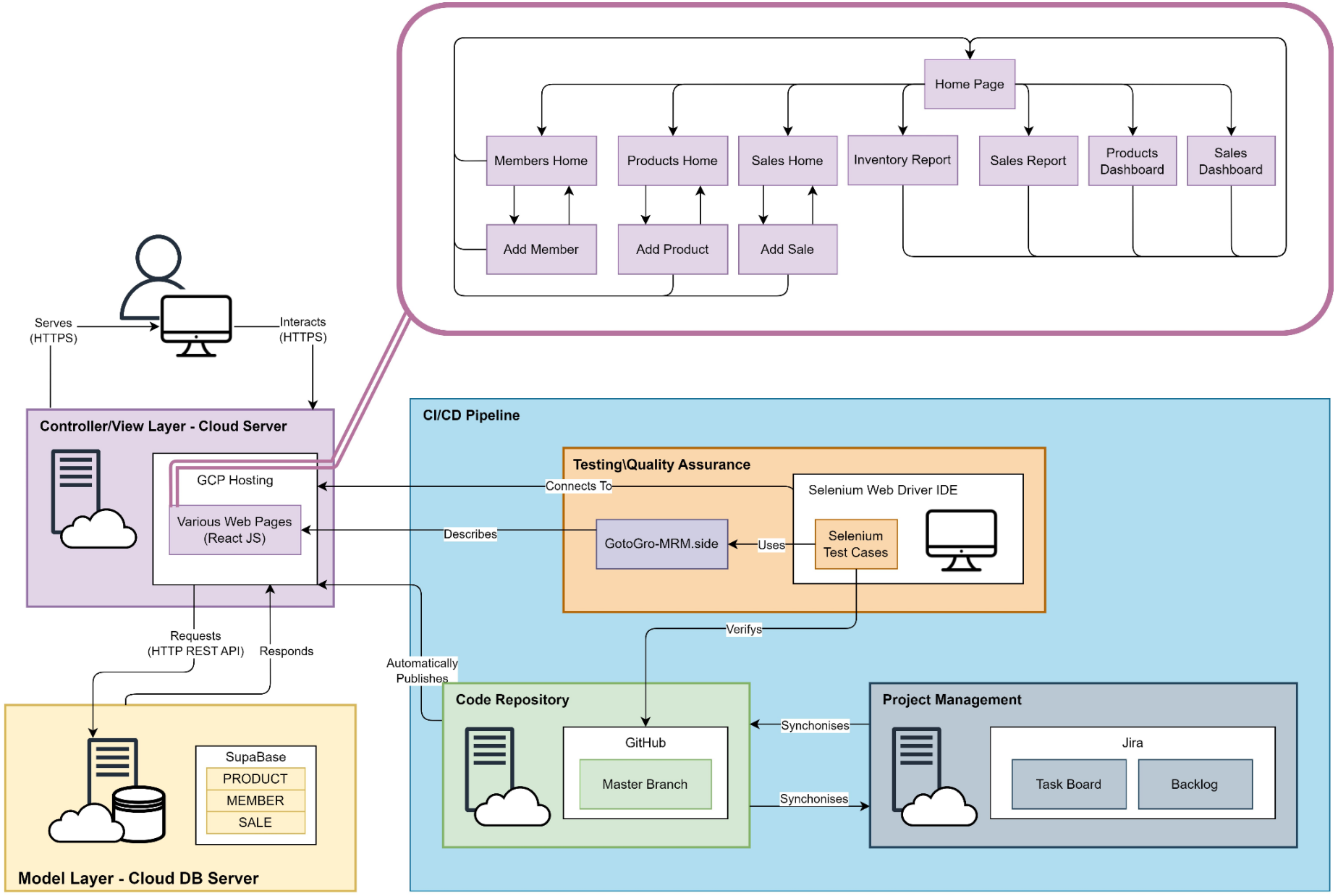
Interactions and Flow

The interactions in Sprint 2 showcase a system with broader capabilities:

- **User Interactions**: Users, depicted by monitor and person icons, connect with the system via HTTPS to access a range of web pages hosted on GCP Hosting.
- **UI Navigation**: From the "Home Page", users can explore the Members, Products, or Sales sections. Each of these has further detailed navigation options, such as adding a member or viewing the sales dashboard.
- **Server & Data Interactions**: Hosted web pages, crafted with React JS, reside on the cloud server. Any user data input or retrieval involves the cloud server

communicating with the Cloud DB Server using HTTP REST API, ensuring efficient data storage and retrieval.

- **Data Management**: Data entities like "PRODUCT", "MEMBER", and "SALE" are stored on the Cloud DB Server, which interacts with the Controller/View Layer based on user actions.
- **CI/CD Workflow**:
    - **Code Management**: Developers commit code to GitHub's "Master Branch", set up for automated updates.
    - **Task Synchronization**: GitHub syncs with Jira, a project management tool, which organizes tasks on a board and backlog, aligning code changes with tasks.
    - **Quality Assurance**: Upon code push, it's linked to the "Selenium WebDriver IDE" for testing. This tool uses test cases to ensure that all new additions function correctly before deployment.

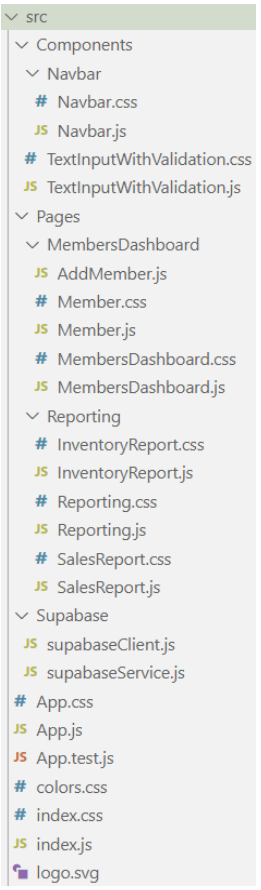

Home Page
Members Home
Products Home
Sales Home
Inventory Report
Sales Report
Products Dashboard
Sales Dashboard
Add Member
Add Product
Add Sale
Serves (HTTPS)
Interacts (HTTPS)
Controller/View Layer - Cloud Server
GCP Hosting
Various Web Pages (React JS)
CI/CD Pipeline
Testing\Quality Assurance
Connects To
Selenium Web Driver IDE
Describes
GotoGro-MRM.side
Uses
Selenium Test Cases
Verifys
Requests (HTTP REST API)
Responds
Automatically Publishes
Code Repository
GitHub
Master Branch
Synchonises
Project Management
Jira
Task Board
Backlog
Synchonises
SupaBase
PRODUCT
MEMBER
SALE
Model Layer - Cloud DB Server

Directory and Naming conventions

Building upon the conventions established in Sprint 1, our directory structure and naming conventions have remained consistent. The decision to maintain these conventions was influenced by our team's collective industry experience, which vouched for the clarity and efficiency of the chosen structure and naming protocols.

Here's a brief reminder of the conventions we've established:

- **Naming Conventions**:
  - For code readability and maintainability, consistent and descriptive names are used for variables, functions, and components.
  - React components are named in PascalCase, reflecting their function or content. For example, "Navbar" for a navigation bar.
  - Functions and variables use camelCase. Descriptive function names like `fetchSalesByDateRange` clearly indicate their purpose.
  - Function parameters are descriptive, e.g., `updatedMember` for a member update.
  - Asynchronous functions, particularly those interacting with the backend, are prefixed with their operations, like `signIn` or `searchMembersByName`.
- **Directory Structure**:
  - Components are systematically organized, with each stored in its own directory like `./Components/Navbar/Navbar`.
  - Pages or views are kept in a "Pages" directory, following the same naming consistency as components.
- **File Naming**:
  - Filenames, including component files, follow PascalCase, e.g., "MembersDashboard".
  - Styles associated with specific components or pages use the component name with a `.css` extension, like `App.css`.
- **Comments and Documentation**:
  - Functions include JSDoc-style comments detailing their purpose, parameters, and return values.
  - Inline comments clarify specific lines of code or operations that might be non-intuitive.
- **Database Operations**:
  - Database tables are referenced in PascalCase, like "SaleRecords" or "Members".
  - Queries are crafted to clearly indicate their purpose.



References to our Sprint 1 directory and naming decisions can be found throughout our codebase, emphasizing our dedication to a systematic approach. Our experienced team recognizes the value of clear and intuitive file structures, component naming, and the overall organization, ensuring that as our project grows, any developer—new or returning—can navigate and understand the system with ease.

Jira Workflow Enhancements

Our experiences in Sprint 1 prompted us to refine our task management approach for Sprint 2, leading to several improvements in our Jira workflow:

1. At the sprint's commencement, backlog items are transitioned into the "To do" column.
2. Developers, upon selecting a task, move it to the “In progress” column. They then create a branch from the main one, naming it after the User Story ID/Code.
3. Post-development and once changes are merged into the production server, the task is relocated to the “Ready to be tested” column.
4. Testers take on items from here and transition them to the “Testing” column.
5. Post-testing, they shift the items to the “To be confirmed” column, appending all collected test evidence.
6. The Product Owner or Scrum Master reviews the evidence. If satisfied, they transition the item to the "Done" column, signifying its successful completion.

This enhanced Jira workflow ensures clarity, transparency, and accountability at every stage, fostering a seamless progression from task inception to completion.

Design Justification using Design Principles

Object-Oriented Principles

In continuation of our commitment to object-oriented design from sprint 1, we have ensured that:

- Each React component still represents a cohesive unit of functionality. For instance:

```
import React from 'react';
import {Link} from 'react-router-dom';
import './Reporting.css';

function Reporting() {
    return (
        <div className='reporting-content'>
            ...
        </div>
    );
}
```

- We've maintained the encapsulation of behaviour and rendering logic in components like `ProductsDashboard` and `InventoryReport`:

```
const ProductsDashboard = () => {
    const [products, setProducts] = useState([]);
    ...
    return (
        ...
    );
}
```

```
function InventoryReport() {
    const [products, setProducts] = useState([]);
    ...
    return (
        ...
    );
}
```

- State variables and hooks continue to be the backbone of data flow and component interactions. This consistency ensures predictability and ease of understanding:

```
const [products, setProducts] = useState([]);
useEffect(() => {
    getProducts();
}, []);
```

Cohesion and Coupling

In our ongoing commitment to strong cohesion and weak coupling, the components discussed below are merely illustrative examples, chosen at random from the myriad of cohesive and loosely coupled components in our codebase. They demonstrate the consistent design philosophy we've adhered to throughout our development:

- **ProductsDashboard Component:**
  - **Cohesion:** This component is a testament to our commitment to strong cohesion, with a clear-cut responsibility of managing and displaying product information.
  - **Coupling:** Consistent with our previous components, its interactions with external services and libraries, like supabaseService and chart.js, are cleanly defined and minimal.
- **InventoryReport Component:**
  - **Cohesion:** It remains laser-focused on its core responsibility of managing and displaying inventory data.
  - **Coupling:** Its interactions with supabaseService, papaparse, fuse.js, and react-toastify are clear cut and minimal, ensuring weak coupling.

By consistently applying these principles across all components, not just the ones highlighted, we ensure a modular, understandable, and adaptable system.

Design Patterns and Architectural Styles

Building upon our foundational design principles, our codebase embraces various design patterns and architectural styles. These patterns and styles, coupled with our commitment to Object-Oriented Principles, fortify our system's maintainability, modularity, and extensibility.

1. **Module Pattern**
   We've encapsulated functionality within ES6 modules, ensuring each module serves a singular, cohesive purpose. For instance, the SalesDashboard, ProductsDashboard, and InventoryReport components are each contained within their respective modules.

```
import React, { useState, useEffect } from 'react';
...

const SalesDashboard = () => {
    ...
    return (
        ...
    );
}

export default SalesDashboard;
```

2. **Component-based Architecture**
   Embracing the power of React, our system thrives on a component-based architecture. Each component, like `ProductsDashboard`, represents a unit of functionality:

```
const ProductsDashboard = () => {
    const [products, setProducts] = useState([]);
    ...
    return (
        ...
    );
}
```

3. **Hooks**
   React hooks, especially `useState` and `useEffect`, have become the backbone of our components, driving state management and side effects.

```
const [products, setProducts] = useState([]);
useEffect(() => {
    fetchProducts().then(data => setProducts(data));
}, []);
```

4. **Service Layer Pattern**
   Our components remain clean and focused by offloading data-fetching logic to a service layer, as demonstrated with the `fetchProducts` function from the `supabaseService` module.

```
const fetchedProducts = await fetchProducts();
setProducts(fetchedProducts);
```

5. **Observer Pattern**
   Implicit in our usage of hooks, the observer pattern enables our components to respond to state and prop changes.

```
useEffect(() => {
    if (selectedProductId) {
        getSalesData();
    }
}, [selectedProductId]);
```

6. **Factory Pattern**
   Our system dynamically constructs data objects based on input data, much like how chart data is built from the products array.

```
const dataBar = {
    labels: products.map(p => p.product_name),
    ...
};
```

7. **Strategy Pattern**
   We provide flexibility in data visualization by adopting different charting strategies, such as Bar, Doughnut, and Pie charts.

```
<Bar data={dataBar} />
<Pie data={dataPie} />
```

8. **Facade Pattern**
   We've employed libraries like `papaparse` and `fuse.js` to offer simplified interfaces, masking more complex underlying operations.

```
Papa.parse(data, {...});
const fuse = new Fuse(products, options);
```

9. **Decorator Pattern**
   Enhancing user feedback, we've integrated toast notifications without altering the core functionality of our components.

```
toast.success('Product updated successfully!');
```

By integrating these patterns and styles, we've ensured a robust, scalable, and highly maintainable system. Each component, while part of a larger ensemble, is designed to be independently updated and changed, championing a modular approach.

Testing rationale

Tool Selection - Why Selenium WebDriver IDE?

Our choice of Selenium WebDriver IDE is rooted in its versatility and broad industry recognition. Its capabilities extend beyond just automating web application tests—it ensures that our software consistently meets expectations, and any potential regressions are pinpointed in real-time.

Immediate Feedback Mechanism

The power of Selenium isn't merely in automation but in the immediacy of its feedback. Each time developers introduce modifications to the code, automated tests spring into action, providing instantaneous insights. This real-time feedback mechanism ensures that our codebase remains perpetually deployable, bolstering our confidence in each release.

Fostering Collaborative Excellence

One of the standout benefits of our testing approach is the enhanced synergy between developers and testers. By clearly defining the testing phase, testers are unburdened from routine validations, directing their focus on more nuanced, scenario-specific evaluations. This not only ensures thorough testing coverage but also cultivates an environment where quality assurance is a shared responsibility.

Transparent Documentation & Comprehensive Evidence Collection

Beyond just test execution, the realm of quality assurance emphasizes the importance of transparency and traceability. Selenium WebDriver IDE champions this cause by meticulously logging test outcomes. This not only aids in immediate debugging but also serves as a repository of evidence, ensuring accountability and facilitating future audits.

Conclusion

As we wrap up Sprint 2, it is an opportune moment to reflect on our progress, understand our growth, and chart our future course. The advancements made during this sprint are a testament to our team's relentless dedication, innovative spirit, and unwavering commitment to excellence.

Comparing our journey from Sprint 1, it is evident that our software's design and functionality have matured considerably. While the foundational principles set in Sprint 1 provided the bedrock for our development, Sprint 2 saw the seamless integration of advanced features, enhanced user interfaces, and a fortified CI/CD pipeline. The adoption of Selenium WebDriver IDE has not only automated our testing processes but also fortified the quality of our deliverables, ensuring that every feature aligns with our high standards. The consistency in our directory and naming conventions, coupled with the refined Jira workflow, underscores our dedication to systematic and transparent development processes.

However, it's essential to recognize that while we have achieved significant milestones, the world of software is ever-evolving. As we look ahead, our focus remains on scaling our software system further, refining our user experiences, and continuously adapting to the dynamic needs of our stakeholders. We foresee a hypothetical Sprint 3 bringing forth newer challenges, and we remain poised to tackle them head-on. Our journey from Sprint 1 to Sprint 2 has been transformative, and we are confident that with each subsequent sprint, we will continue to break boundaries, set higher benchmarks, and redefine excellence.