Project Report

# Vision

Our team developed “SuperPrice”, a website that provides customers with a convenient and efficient way to search, compare and purchase products from different groceries. It provides a range of benefits:

* **Effortless Price Comparison**: Customers can easily compare product prices across various grocery stores, saving time and effort in finding the best deals.
* **Cost Savings**: "SuperPrice" allows customers to identify the most budget-friendly options, potentially saving money on their purchases.
* **Expanded Product Choice**: The platform offers a wide array of products, enabling customers to explore and choose based on their preferences.
* **Convenient Shopping**: Customers can conveniently browse and compare prices from anywhere, making their shopping experience more accessible.
* **Transparent Information**: Clear product details and pricing information empower customers to make informed purchasing decisions.
* **Promotions and Discounts**: Customers can easily find ongoing promotions and discounts, further enhancing their savings.
* **Efficiency and Simplicity**: The platform provides an efficient and organized shopping experience, making it simple for customers to find what they need.

In summary, "SuperPrice" delivers a streamlined shopping experience by offering effortless price comparison, cost savings, a wide product selection, convenient browsing, transparent information, promotions, and efficiency - collectively providing customers with a comprehensive and user-friendly platform for their shopping needs.

# System architecture

|  |
| --- |
| **Front-end:** React |
| **Middle end:** REST API |
| **Back-end:** MySQL (formerly H2 database) |

*Figure 1.0: System architecture*

**Architecture pattern used:** We implemented a three-tier architecture, where there are distinct layers for the front, middle and back ends.

**Front-end**

* The front-end is the user interface of the system, where users interact with the application. It’s responsible for presenting information and enabling actions related to searching for products, viewing product details, managing the cart, choosing delivery options, and completing the checkout process.
* Components:
  + User Interface (UI): The graphical interface where users interact with the system. This includes search bars, product listing views, cart displays, and checkout interfaces.
  + Client-side Logic: JavaScript code that enables functionalities like keyword-based search, item viewing, cart management, and delivery option selection.
  + Front-end Frameworks/Libraries: React was used to create a user-friendly interface design.

**Middle end**

* The middle-end serves as an intermediary layer, facilitating communication between the front-end and back-end. It processes requests related to product searching, viewing, cart management, and delivery options, while also handling requests for the notification feature.
* Components:
  + API Endpoints: URLs such as "/search" and "/product" to handle requests for searching and viewing products.
  + Request Handlers: Components that process user requests for actions like adding/removing items to/from the cart, managing delivery options, and handling notifications.
  + Business Logic: The core logic that operates on the received requests and returns appropriate responses based on the request type.

**Back end (MySQL)**

* The back end is responsible for CRUD of the data related to products, cart and cart products.

This three-tier architecture, with its distinct components and layers, effectively supports the system's features, enabling users to search for products, view item details, manage their cart, choose delivery options, and complete purchases while also incorporating the enhancement feature for notifications.

# Refactoring

**Scenario 1:** Refactoring Repository Classes (Backend)

**Design patterns used:**

1. Single Responsibility Principle (SRP): Classes should only be allocated a specific task only, rather than trying to do everything (Madasu et al., 2015).
2. Façade Pattern: This structural pattern helps create a simplified interface for a complex set of classes (Refactoring Guru 2023).

**Description:**

In the initial state of the API, the 'SuperpriceRepository' class violated the Single Responsibility Principle, as it had multiple responsibilities. To address this, the class was refactored by breaking it down into three separate repository classes: 'ProductRepository,' 'NotiRepository,' and 'CartRepository.' Each of these new classes was designed to have a single responsibility, adhering to the principles of the Single Responsibility Principle. Furthermore, by creating distinct repository classes, a Facade pattern was implicitly followed, as each repository provided a unified interface to a subsystem, simplifying the overall structure.

**Scenario 2:** Mitigating Database Timeout issues (Backend)

**Design patterns used:**

1. Builder Pattern (indirectly): A creational pattern that creates objects step by step (Refactoring Guru 2023).
2. Mock Object Pattern: This involves testing the behavior of objects using simulated objects that imitate the real versions (DisciplinedAgile n.d).

**Description:**

As the feature set expanded, the number of tests increased, impacting the backend testing process with potential database timeouts. To tackle this issue, the team utilized mocking in tests that involved database manipulation. Although not a direct application, the use of mocking in this scenario indirectly aligns with the Builder pattern by allowing the construction of complex objects (mocks) step by step. Additionally, the usage of mock objects conforms to the Mock Object pattern, enabling the creation of simulated behavior for the database. This approach ensured that the tests remained independent of each other and focused more on behavior rather than the actual impact on the database.

**Scenario 3:** Improving front-end code.

**Description:**

Initially, the front-end code was a bit messy. To make it better, we followed a few simple rules.

* **Elucidating Comments for Clarity**: We introduced informative comments within the code, explaining the functionality of each segment.
* **Code Refinement**: Unnecessary code sections were removed, decluttering the structure by removing redundant or excess lines.
* **Precise Variable Typing for Clarity**: Diligence was exercised in clearly defining data types for every variable, thus improving the clarity of the code regarding the fundamental components it manages.

By following these simple rules, we made the front-end code easier to understand and work with, ensuring it runs smoothly with Docker and leaving no room for confusion or unexpected errors.

# Git Organization

**Branching strategy:**

We began the project by creating the skeleton code in the Main branch, including the Maven and SpringBoot setup, folder directories and basic code. We then created branches for specific features and major implementations such as implementing H2 database. This helps enable us to test the new code with the existing one, without affecting the already-existing code. If an issue were to arise, we can also rollback to the version that was working previously.

**Commit frequency and practices:**

Throughout this project, we tried to commit based on tasks completed. For example, a commit would entail a functional implementation of a method and its respective test. We also ensure that before pushing, the code is working properly and does not affect other parts of the code.

**Collaboration and code review:**

Once a feature is considered complete, our team reviews everything that is on the branch: checking if the code works as intended, if it passes all tests and if the test cases cover all possible scenarios. After reviewing, we then push the code to the main branch and mark it “Completed” on our project board.

# Scrum Process

**Meeting frequency**

We conducted weekly stand-up meetings with the Product Owner in-class, showcasing our progress, clarifying questions, discussing changes that could be implemented and discussing what needs to be done moving forward.

We also conducted 2-3 scrum meetings throughout each week, where our team came together to provide progress updates, discuss questions, identify obstacles that may affect our progress, and our action.

# Deployment Pipeline

# References

Madasu, V.K., Venna, T.V.S.N. and Eltaeib, T., 2015. Solid principles in software architecture and introduction to resm concept in oop. Journal of Multidisciplinary Engineering Science and Technology (JMEST), 2(2), pp.3159-0040.

‘Facade’, Refactoring Guru (2023), accessed October 8, 2023, <https://refactoring.guru/design-patterns/facade>

‘Builder, Refactoring Guru (2023), accessed October 8, 2023, <https://refactoring.guru/design-patterns/builder>

‘The mock object pattern’, Disciplined Agile (n.d), accessed October 8, 2023, <https://www.pmi.org/disciplined-agile/the-design-patterns-repository/the-mock-object-pattern>