
# CS3219 Software Engineering Principles and Patterns

# Final Report

---

**Source code:** https://github.com/Stalkfish-Singapore/STALKFISH

**Deployed web app:** https://stalk.fish

**Team 13**


| |
|---|
| Chew Hong Wei Terence \| A0200399E \| chewterence@u.nus.edu |
| Joel Ho Eng Kiat \| A0200385M \| joel.ho@u.nus.edu |
| Aidil Fitrahadi Bin Kamsani \| A0181317J \| afk@u.nus.edu |
| Jason Lim Jian Shen \| A0200431E \| jasonlim@u.nus.edu |
| Lee Shi Jie Shawn \| A0190131W \| shawnlsj@u.nus.edu |

# Table of Contents

# 1 Background and Purpose of Project

## 1.1 Purpose & Overview

In the age of e-commerce, consumers are in constant search of better prices, offers and discounts. With how easy it is to find a better deal on a certain product, it becomes more important that companies offer competitive prices at the right timing and also the fastest timing possible in order to capture the market.

The client, “Snow Treasures Singapore”, is a family based business that sells frozen seafood products on various e-commerce platforms such as *Shopee*, *Qoo10* and *Lazada*. The client is a SME consisting of older staff who are not “tech-savvy”. Currently, in order to track competitor prices the staff would manually visit a URL link of the competitor’s products, and manually update an excel sheet that is shared through OneDrive. Consequently, doing so has proven to be very time consuming for the company.

To solve this issue, we proposed building a platform (StalkFish) to automatically monitor prices on selected e-commerce products across several client-specified competitors. On StalkFish, users are able to visualise the price fluctuations of client-specified products, with advanced features of suggesting pricing based on criteria provided by the user.

StalkFish works as follows: a user (a staff member from Snow Treasures Singapore e-commerce team) adds a URL to a competitor product that is to be monitored. The system would then periodically gather data such as product name, product price and number of products sold, which is then stored into a database. The user subsequently visits the dashboard to get an overview of all the details that have been gathered. To get an even more comprehensive outlook of each competitor product, the user could select and filter each product based on attributes they have defined.

## 1.2 Project Vision

Upon discussion with the client, the general business requirements are to automatically monitor the ever-fluctuating prices of competitor’s products. Doing so would enable the client to actively update the prices of their products at the right and fastest timing possible in order to capture the market. As the volume of sales for the client of the e-commerce platforms are not small, every

change in price matters in the long run. Ultimately, the vision and goal of StalkFish would be to maintain the client's competitive position in the e-commerce market.

## 1.3 Project Scope

To meet the business requirements and goals, the scope of StalkFish would encompass features which make up three main primary functions: 1) Automatically gather competitor data, 2) Process the gathered data 3) Visualize and display the processed data. The details of the product features are described in the subsequent sections of this document.

The intended users of StalkFish are hence defined as the employees from the client company's e-commerce team that is tasked to monitor and update product prices.

# 2 Individual Contributions

The following table summarizes our individual technical and non-technical contributions to the development of the application.

| | **Contributions** | |
|---|---|---|
| **Name** | **Technical** | **Non-Technical** |
| Chew Hong Wei Terence | Web Scraper, Notification Service, Deployment | Project management, Report writing Coordination with the client for delivery, testing and feedback |
| Joel Ho Eng Kiat | Web Scraper, Notification Service, Price Modelling, Deployment | Project management, Report writing |
| Aidil Fitrahadi Bin Kamsan | Backend infrastructure, Data aggregator, Database, API Gateway, Deployment | Project management, Report writing |
| Jason Lim Jian Shen | Frontend (Product history), Authentication, Notification Service Telegram Bot, Deployment | Project management, Report writing |
| Lee Shi Jie Shawn | Frontend (Dashboard), Price Modelling, Deployment | Project management, Report writing |

# 3 Requirements

## 3.1 Functional Requirements

Requirements of StalkFish have been gathered through an interview and active discussions between the software development team and the client. Upon finding the key pain points in the current process, we get a better understanding of the requirements of the system.

The functional requirements (FRs) of StalkFish have been filtered into the respective classifications:
category of feature, functional requirement of feature, priority of functional requirement (indicated by ‘H’ for the highest priority, ‘M’ for medium priority and ‘L’ for lowest priority), as well as the rationale behind prioritizing each requirement.

We have managed to implement all high and medium priority FRs, with the exception of one medium priority FR. We have left low priority FRs as future extensions for the project, and intend to implement them beyond the timeline of the module for the client. FRs that have not been implemented are highlighted in red in the table below.

| Category | Functional Requirement | Priority | Rationale |
|---|---|---|---|
| Scraping | **[FR1.1]** The application should support scraping data from a competitor page. (Refer to SRS for complete details) | H | The visualization of competitor data runs on the basis that there is existing product information stored in the database, following the web scraping. Hence, scraping and storing the data is one of the top priorities. |
| | **[FR1.2]** The application should support scraping data from an Item page. (Refer to SRS for complete details) | H | |
| | **[FR1.3]** The application should store data about the products (Rating, stock count, sales count, favorited count, price changes, current discount) (Refer to SRS for complete details) | H | |

| | | | |
|---|---|---|---|
| | **[FR1.4]** The application should support scraping from Shopee. | H | According to the client, the Shopee platform consists of the highest volume of sales, user traffic and competitor data. As a result, scraping data from Shopee has been prioritized as the highest. |
| | **[FR1.5]** The application should support scraping from Qoo10. | M | The Qoo10 platform has the second highest volume of sales, user traffic and competitor data. Hence, scraping data from Qoo10 has been prioritized as medium. |
| | **[FR1.6]** The application should support scraping from Lazada. | L | The Lazada platform has the lowest traffic as well as competitor data, hence the prioritization of scraping data from Lazada is at the lowest. |
| | **[FR1.7]** The application should periodically scrape the product information, minimally once every 30 minutes. | H | In order to get the latest updates on price changes, the application should periodically gather/scrape the data off the respective web pages every hour. |
| Data Processing | **[FR2.1]** The user should be able to search and filter competitors for the visualisation. | H | In order for users to efficiently visualize competitor data, they have to be able to search and filter for the desired product/competitor. Hence this requirement has been prioritized as one of the highest. |
| User Authentication | **[FR3.1]** The authorised user should be able to login securely. | H | Authentication is identified as one of the highest priorities for functional requirements, in order to prevent non-authorized users from accessing the StalkFish system. |
| | **[FR3.2]** The master user should be able to add accounts. | L | While there is no current requirement by the client for multiple users, including a master account for account management has been proposed. |
| Price Model | **[FR4.1]** The application should be able to predict price trends based on historical data in the form of a graph. | M | Price trend prediction has been proposed and discussed with the client as a secondary/advanced feature. |
| | **[FR4.2]** The application should be able to suggest a price based on user specified rules in the form of a graph. | M | The client has suggested additional means to enhance the price model through user specified rules. |
| | **[FR4.3]** The application should be able to detect anomalies in pricing. | L | This requirement is merely an enhancement for the price model. |
| Web App | **[FR5.1]** Users should be able to add an item page or | H | In order for the price scraping system to start gathering data, the user is required to add URLs of |

| | | | |
|---|---|---|---|
| | competitor page link for data scraping. | | either an item or a competitor page. Hence, this requirement has been prioritized. |
| | **[FR5.2]** The application allows users to visualize competitor data at one glance by plotting all related products on a graph. | H | Visualization of competitor data is one of the key requirements as discussed with the client. Hence, data visualization on a front-end application has been prioritized. |
| | **[FR5.3]** Users should be able to view historical sales and price for a particular product plotted on a graph. | H | Being able to keep track of the price changes and sales over a time period is important for the client to maximize their profit. |
| | **[FR5.4]** Users should be able to specify product associations between competitor and user products. | M | Each competitor may have different descriptions for the same product, and it may not be immediately obvious without human intervention. |
| Notifications | **[FR6.1]** The application should allow users to subscribe to notifications for each individual product. | M | The notification feature has been discussed with the client, and would be useful. However, the web application would be of higher priority as the notification would direct the user to the web app, hence the prioritization. |
| | **[FR6.2]** The application should send out timely alerts subscribed by the user, minimally once every 30 minutes. | M | |
| | **[FR6.3]** The application should support sending notifications through Telegram. | M | |
| Document Export | **[FR7.1]** The application should allow users to export certain data to csv format. | L | While this feature would prove useful to the client, it has been identified as one of the lowest priorities, as the primary objective of StalkFish is to monitor competitor data. |

## 3.2 Non-Functional Requirements

As with the Functional Requirements, the Non-Functional Requirements (NFRs) have been gathered through an interview. Similarly, they have been prioritised with rationale given.

We have fulfilled all NFRs in this project.

| **Category** | **Requirement** | **Priority** | **Rationale** |
|---|---|---|---|
| Performance | **[NFR1.1]** Search and filter operations should update the visualisation quickly (<10s) | M | The application should be reasonably responsive, but there is some slack on how responsive the client requires it to be |
| Scalability | **[NFR1.2]** Able to scale up new services if required. | H | Web Scraping and processing of data could potentially be a computationally intensive process. As the user adds more URLs to be scrapped, computation needs to scale up |
| Portability | **[NFR1.3]** Accessible via desktop. | H | The pricing has to be adjusted on the desktop interface, and a larger device screen would allow for easier visualisation |
| | **[NFR1.4]** Accessible via mobile. | L | For ease of looking at data while not in front of a desktop/laptop |
| Security | **[NFR1.5]** Only authorized users from the company should be able to access the platform. | H | Security in terms of authentication has been requested as a priority from the client. In order to prevent unauthorized users from using the system. |
| Usability | **[NFR1.6]** Web-application should be user friendly for non-tech savvy staff. | M | The users are not very tech-savvy, but can learn to use a mildly complex system |

# 4 Development Process

This section describes how our team chose to allocate our time and manpower for the overall project.

## 4.1 Roles and Responsibilities

We decided to split the project into 3 key parts - 1) Authentication and notifications, 2) Scraping, 3) Price modeling and data visualization

| Authentication and Notifications | Scraping | Price Modeling and Data Visualization |
|---|---|---|
| ● Jason<br>● Terence | ● Terence<br>● Joel<br>● Aidil | ● Joel<br>● Shawn |

### Team Roles

- Team Lead
- Tech Lead
- Test/QA IC
- Integration IC
- Documentation IC

We allocated roles to each member to ensure that a direction was set for each major component of the project. This contributed to our project's success as there was always a member verifying that the team met the task requirements. The tech lead had the strongest technical skill and provided assistance to all other members throughout the project development. Refer to Figure 4.1a below for our role and part allocation.

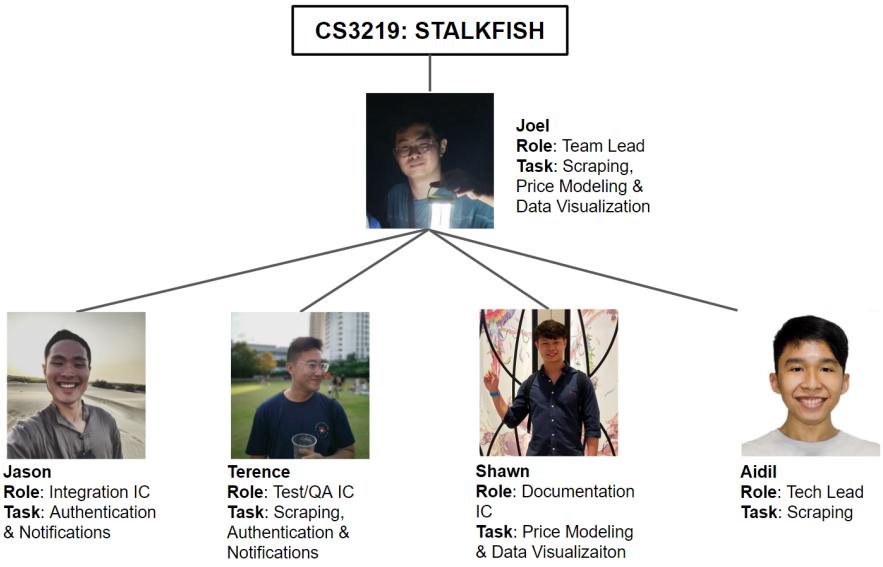



*Figure 4.1a: Team Structure*

## 4.2 Project Timeline

In developing StalkFish, we adopted the iterative software development process model, where the system was developed incrementally through repeated cycles. The iterative model allowed us to continuously gather feedback, review our performance and make improvements in each subsequent cycle. We gradually learnt more about the problem and tools after each cycle, and used our newly acquired knowledge to improve our solution.

More specifically, we adopted the breadth-first iteration model, in which we worked on all components simultaneously in order to develop one functionality at a time. This way, we could conduct frequent integration and system testing at the end of each mini-iteration to ensure that we had a working prototype to show to our client for feedback. Furthermore, this approach allowed us to be able to identify bugs and design issues in an early stage, which allowed us to rectify our errors ahead of deadlines.

We have divided 3 main iterations into 6 mini-iterations. Each mini-iteration is conducted within 1-2 weeks, with weekly team meetings. Refer to Figure 4.2a below for an illustration of our overall development timeframe.

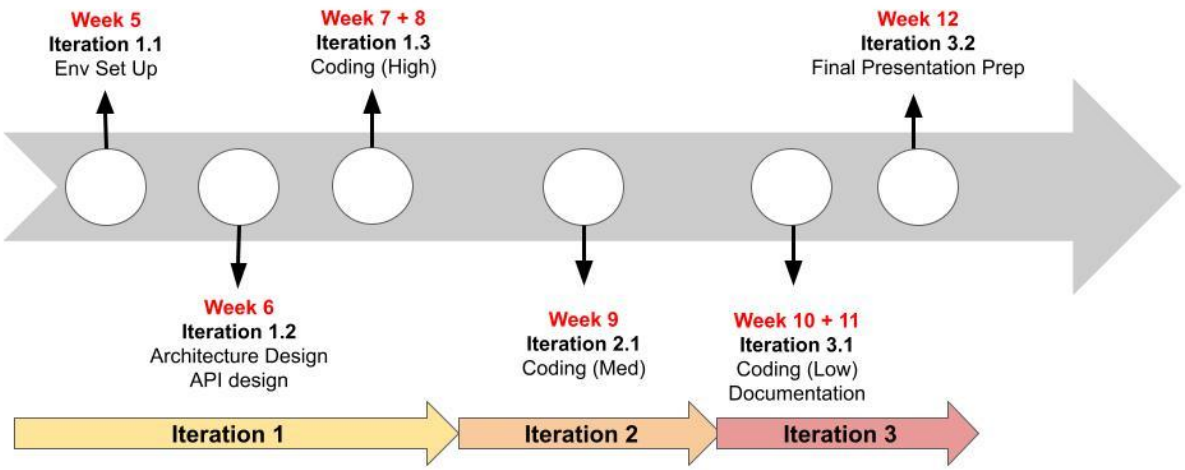



*Figure 4.2a: Overall Development Timeline*

We have divided our coding development based on our requirements mentioned above in Section 3, with high priority requirements being implemented first, followed by medium priority then low priority.

## 4.3 Project Management Tools

The GitHub issue tracker was used as a project management tool to keep track of tasks, enhancements, and bugs. We have created individual labels for each of the services, which made it easier for the relevant team members involved in each service to keep track of its related issues. The screenshot below illustrates a snippet of our GitHub issue tracker.

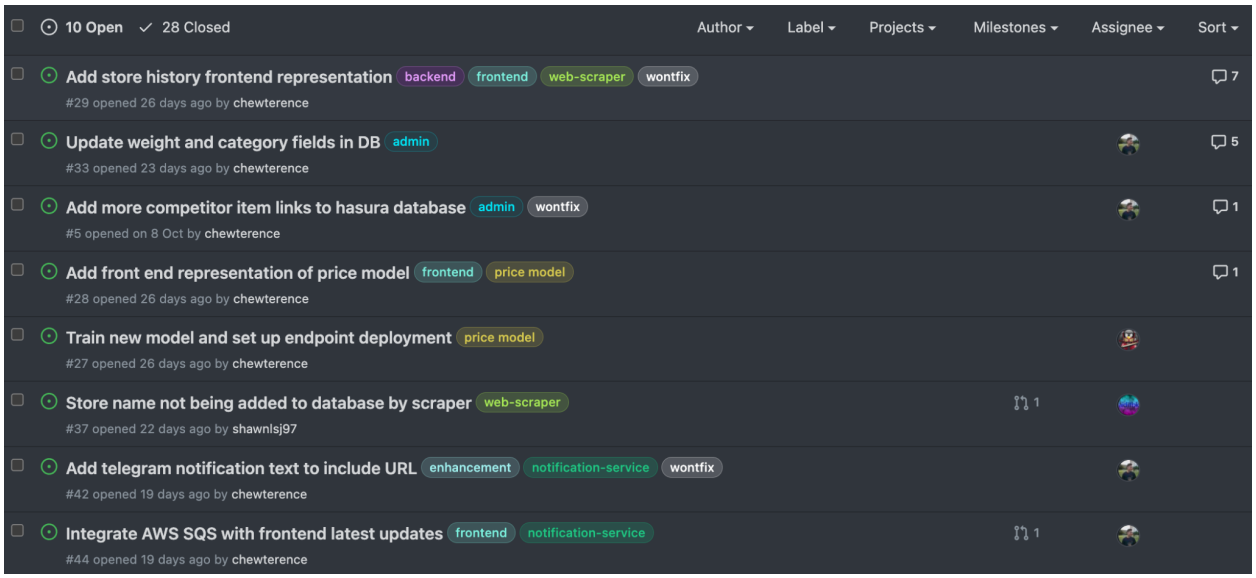



*Figure 4.3a: Issue Tracking and Assignment on GitHub*

Another tool we utilized for project management was Discord. Similar to GitHub issues, we have created multiple voice and text channels to organize information by our application's services. For instance, weekly meetings were held in the 'Meeting' channel, while there were also several other voice/text channels to hold smaller service-related discussions. The screenshot below shows how we have utilized Discord as a project management tool.

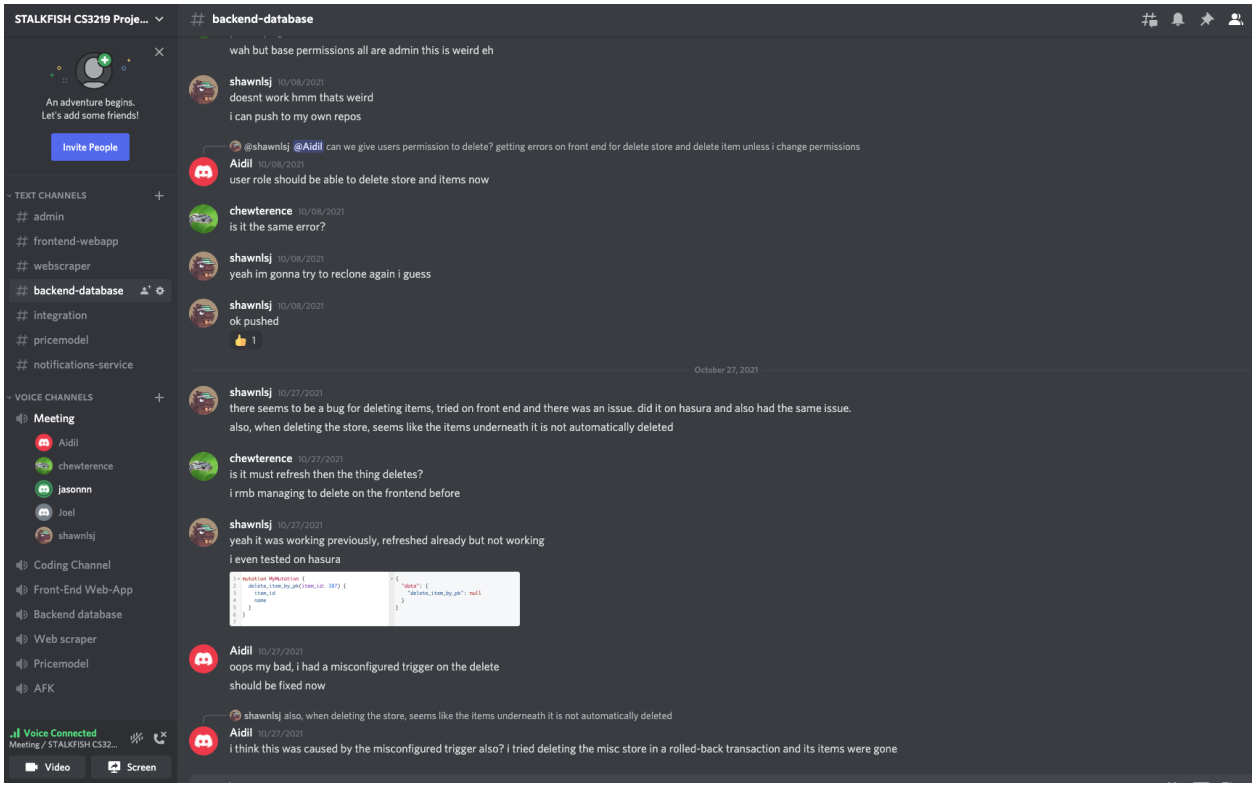



*Figure 4.3b: Project Management on Discord using Channels*

The use of these project management tools have simplified team collaboration, by organizing communications in a unified place, and ensuring that important alerts are automatically sent to the relevant parties. Moreover, it helps facilitate effective task delegation as well as mitigate potential bottlenecks when it comes to technical issues.

## 4.4 Project Tasks

Project tasks and timeline are summarized in diagrams in this section.

### 4.4.1 Task Timeline

| Tasks / Components | Mini-Iteration 1 | Mini-Iteration 2 | Mini-Iteration 3 | | Mini-Iteration 4 | Mini-Iteration 5 | | Mini-Iteration 6 |
|---|---|---|---|---|---|---|---|---|
| | Week 5 | Week 6 | Week 7 | Week 8 | Week 9 | Week 10 | Week 11 | Week 12 |
| Env | Set up & installation | | | | | | | |
| Front-End | | Design (wireframe + mock-ups) | Set up React front-end | **[FR5.1]** Implement CRUD operations for competitors / products | **[FR2.1]** Implement filtering | **[FR4.2]** Integrate price modelling with front-end | | |
| | | | **[FR3.1]** Implement login page | **[FR5.3]** Implement product history | **[FR5.4]** Implement product association | Implement price prediction | | |
| | | | **[FR5.2]** Implement dashboard | Integration of microservices and front-end | | **[FR6.1]** Integrate notifications with front-end | | |
| | | | | Unit testing | | | | |
| Back-End | | | Set up NGINX API gateway | Set up Hasura data aggregator | | | | |
| | | | Set up AWS EC2 VM instance | **[FR1.3]** Set up postgres database | | | | |
| Scraping | | | **[FR1.2] [FR1.4]** Implement scraping of products | Integrate scraping with Redis queue manager | | Stress test scrapers | | |
| | | | **[FR1.1] [FR1.4]** Implement scraping of competitors | Unit testing | **[FR1.7]** Deploy AWS Lambda for scraping | | | |
| Price Modeling | | | | | | **[FR4.1]** Implement price modelling service | | |
| Authentication | | | Implement auth | Integrate auth with front end | | | | |
| | | | Set up auth db | | | | | |
| Notifications | | | | | **[FR6.3]** Set up telegram bot | **[FR6.1]** Integrate notifications with front-end | | |
| | | | | | **[FR6.2]** Implement notifications for Telegram | | | |
| Overall System | | | | System testing | | | | Final product testing |
| Documentation | | Architecture design | | Documentation of mini-iteration and tests | Gather feedback from client | | Documentation of mini-iteration and tests | |
| | | Presentation and SRS submission | | | Refinement of requirements | | | Work on final documentation + presentation |

Figure 4.4.1a. Gantt Chart Summarizing Our Team's Task Timeline.

### 4.4.2 Task Assignment

The following table consolidates an overview of tasks assigned to each team member.

| | Mini-Iteration | Task | Jason | Terence | Shawn | Joel | Aidil |
|---|---|---|---|---|---|---|---|
| **Iteration 1** | 1 | Set up & installation | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 2 | Design (wireframe + mock-ups) | ✓ | ✓ | ✓ | | |
| | | Architecture design | ✓ | ✓ | ✓ | ✓ | ✓ |
| | | Presentation and SRS submission | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 3 | Set up React front-end | ✓ | | ✓ | | |
| | | **[FR3.1]** Implement login page | ✓ | | ✓ | | |
| | | **[FR5.2]** Implement dashboard | ✓ | | ✓ | | |
| | | **[FR5.1]** Implement CRUD operations for competitors/products | | | ✓ | | |
| | | **[FR5.3]** Implement product history | ✓ | | ✓ | | |
| | | **[FR1.1] [FR1.4]** Implement scraping of competitors | | ✓ | | ✓ | |
| | | **[FR1.2] [FR1.4]** Implement scraping of products | | ✓ | | ✓ | |
| | | Set up NGINX API gateway | | | | | ✓ |
| | | Set up Hasura data aggregator | | | | | ✓ |
| | | Set up AWS EC2 VM instance | | | | ✓ | ✓ |
| | | **[FR1.3]** Set up postgres database | | | | | ✓ |
| | | Integration of microservices and front-end | ✓ | | ✓ | | ✓ |
| | | Unit testing of front-end | ✓ | | ✓ | | |
| | | Unit testing of microservices | | ✓ | | ✓ | ✓ |
| | | System testing | ✓ | ✓ | ✓ | ✓ | ✓ |
| | | Documentation of mini-iterations | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Iteration 2** | 4 | **[FR2.1]** Implement filtering of products | | | ✓ | | |
| | | **[FR5.4]** Implement product association | ✓ | | | | |
| | | **[FR6.3]** Set up Telegram bot | ✓ | ✓ | | | |
| | | **[FR6.2]** Implement notifications for Telegram | ✓ | ✓ | | | |
| | | **[FR1.7]** Deploy AWS Lambda for scraping | | ✓ | | ✓ | ✓ |
| | | Implement scraper queue manager | | | | | ✓ |
| | | Gather feedback from client | | ✓ | | | |
| | | Refinement of requirements | ✓ | ✓ | ✓ | ✓ | ✓ |
| **Iteration 3** | 5 | **[FR6.1]** Integrate notifications with front-end | ✓ | ✓ | | | |
| | | Cypress testing for front-end | ✓ | ✓ | ✓ | | |
| | | **[FR4.1]** Implement price modelling service | | | | ✓ | |
| | | **[FR4.2]** Integrate price modelling with front-end | | | ✓ | | |
| | | Connect price modelling to API gateway | | | | | ✓ |
| | | Stress test scrapers | | | | | ✓ |
| | | Documentation of mini-iterations and tests | ✓ | ✓ | ✓ | ✓ | ✓ |
| | 6 | Final product testing | ✓ | ✓ | ✓ | ✓ | ✓ |
| | | Presentation and final report submission | ✓ | ✓ | ✓ | ✓ | ✓ |

## 4.5 Rationale for Development Plan

The rationale behind our plan was to ensure that we were able to implement the requirements listed in Section 3 by the project deadline. On top of that, our plan ensures that we have sufficient time to test our solution extensively.

### Coding

We decided as a team to have 2 people work on the authentication and notifications, 3 people on the web scraping and 2 people on the price modeling and data visualization. We decided on this division of labour based on how complex we felt each component would be.

We also made sure that each component should have at least 2 members to facilitate closer collaboration and better problem solving.

### Testing

Testing was conducted concurrently with coding in order to enforce correctness of the program incrementally. At the end of each iteration, we delivered a prototype to the client to elicit feedback and identify any issues with the system.

Refer to Section 8 for our detailed test plan, which includes unit, integration, system and acceptance testing.

### Continuous Integration and Deployment

As the solution was to be delivered to the client at regular intervals, it was important to set up a continuous integration and continuous deployment workflow, to ensure that changes to the code base were correct, and then subsequently deployed. The decision was made here to use continuous deployment instead of continuous delivery, especially in the development stage, to rapidly prototype features, and enable the client to view changes as they were being made.

Refer to Section 7 for more information on how StalkFish was deployed.

### Documentation

Our documentation lead, Shawn, decided on the structure of the report to ensure that the documentation for the different sections is consistent. Each team was responsible for the documentation of their respective components following this structure.

We also made sure to update the documentation during each mini-iteration instead of leaving it to the end of a major iteration.

# 5 StalkFish Features

The feature tree below illustrates an overview of all the key features of StalkFish.

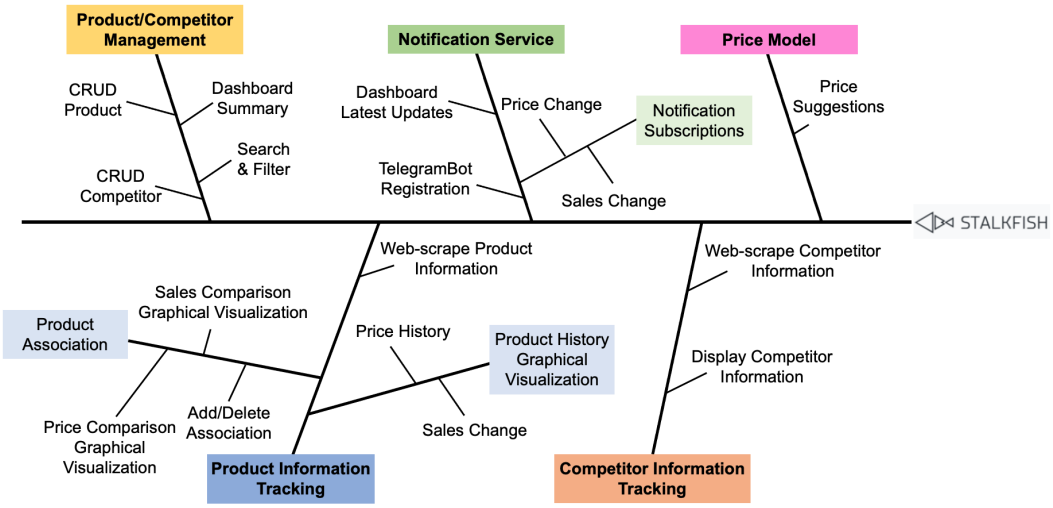



*Figure 5a: Feature Tree of StalkFish*

## 5.1 Login via Google Auth

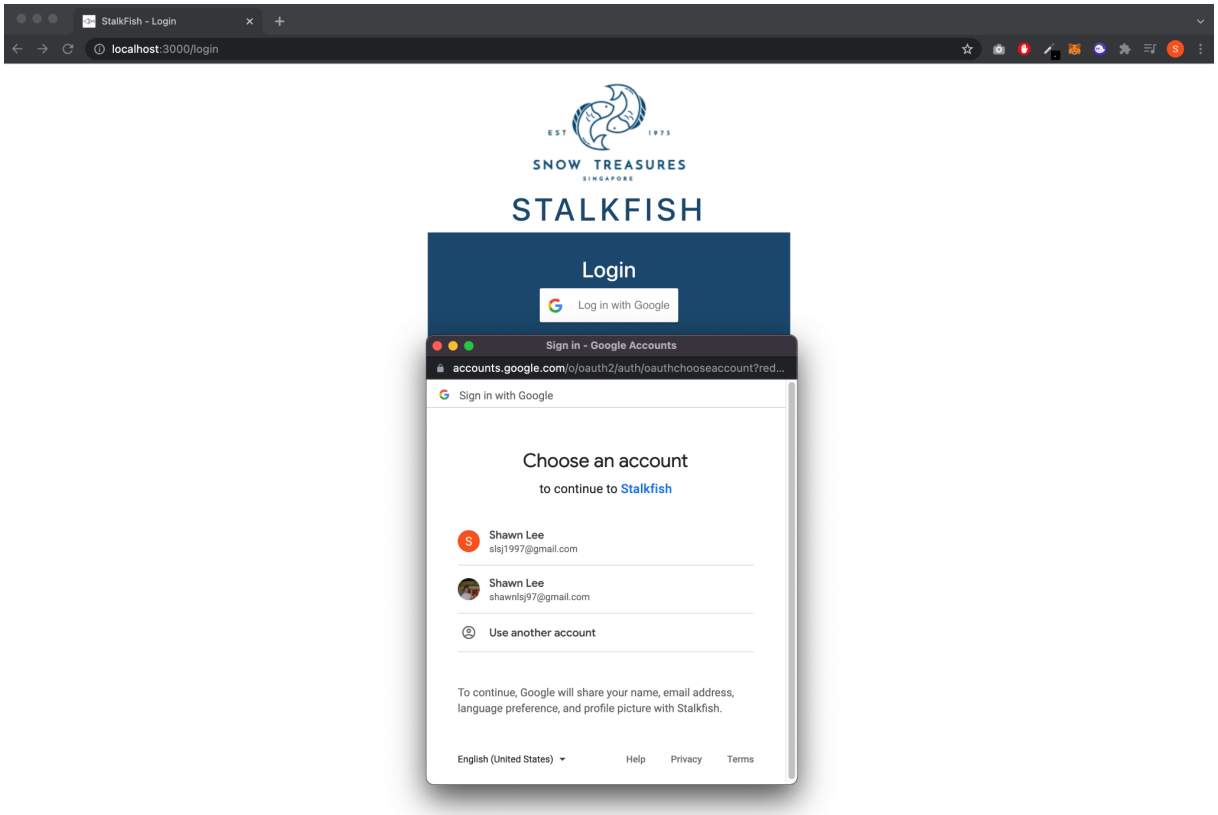



*Figure 5.1a: Login Screen*

When users first access the web app, they are brought to the login screen. We have made user authentication convenient for users by integrating Google Authentication. Clients can simply login with their existing Google account provided it has been granted access, fulfilling **[FR3.1]**.

# 5.2 Store and Products Dashboard

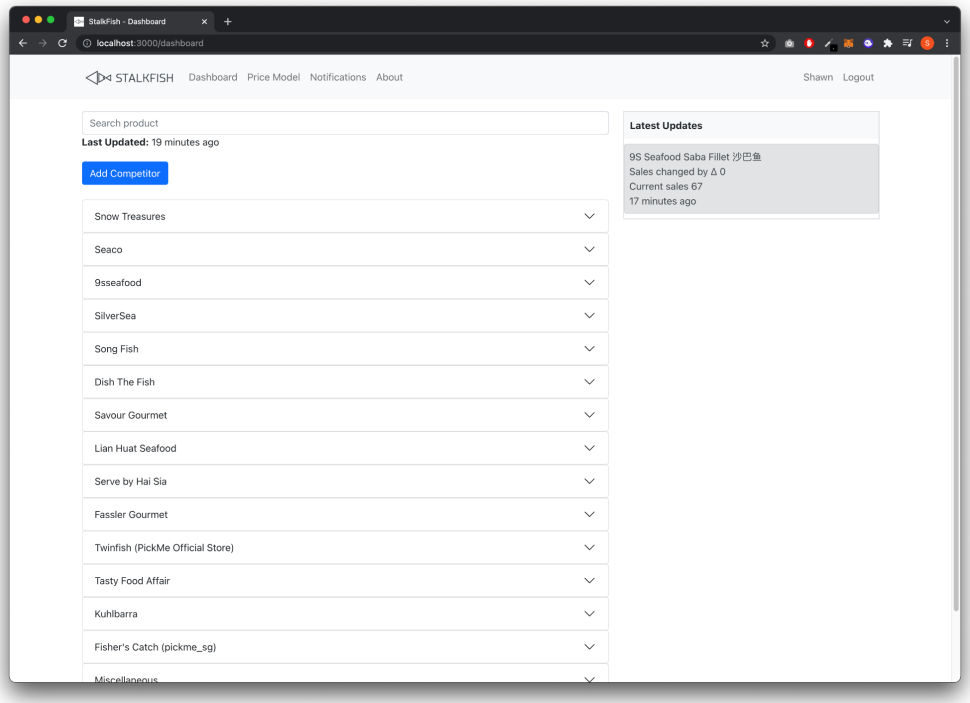



*Figure 5.2a: Dashboard*

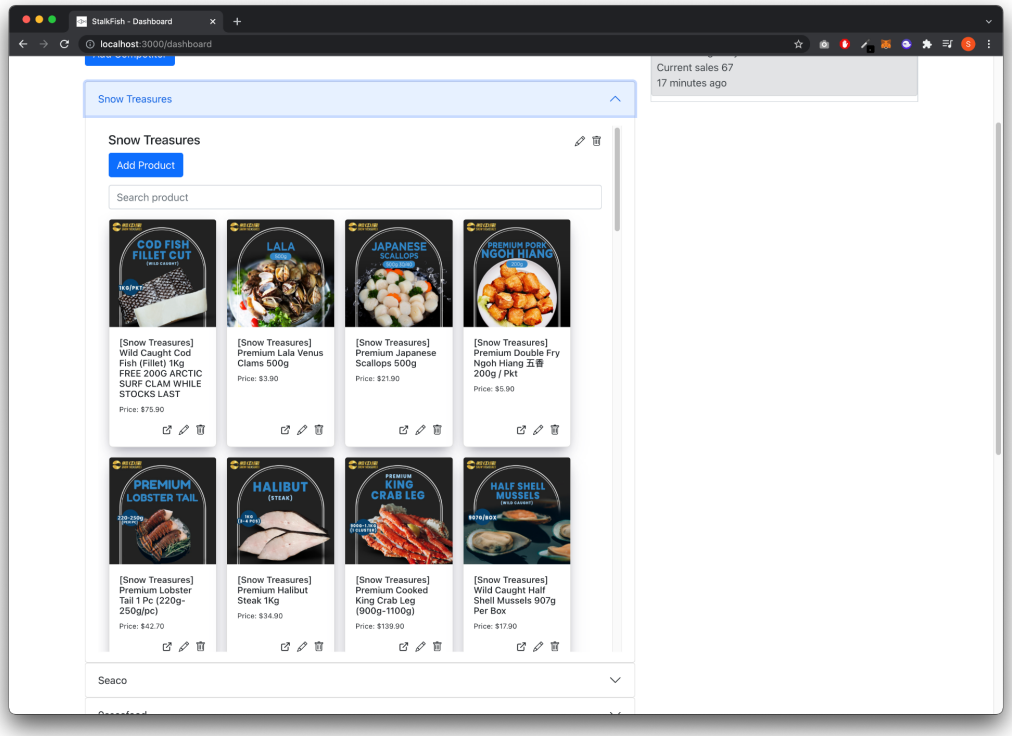



*Figure 5.2b: Expanded Store Accordion Tab*

Upon successful login, users are brought to the main dashboard. On this page, users are able to see a list of all stores in an accordion on the left and latest notifications on the right. When users expand the accordion of a particular store, they can view all the products sold, fulfilling **[FR5.2]**.

## 5.2.1 Search and Filter Products

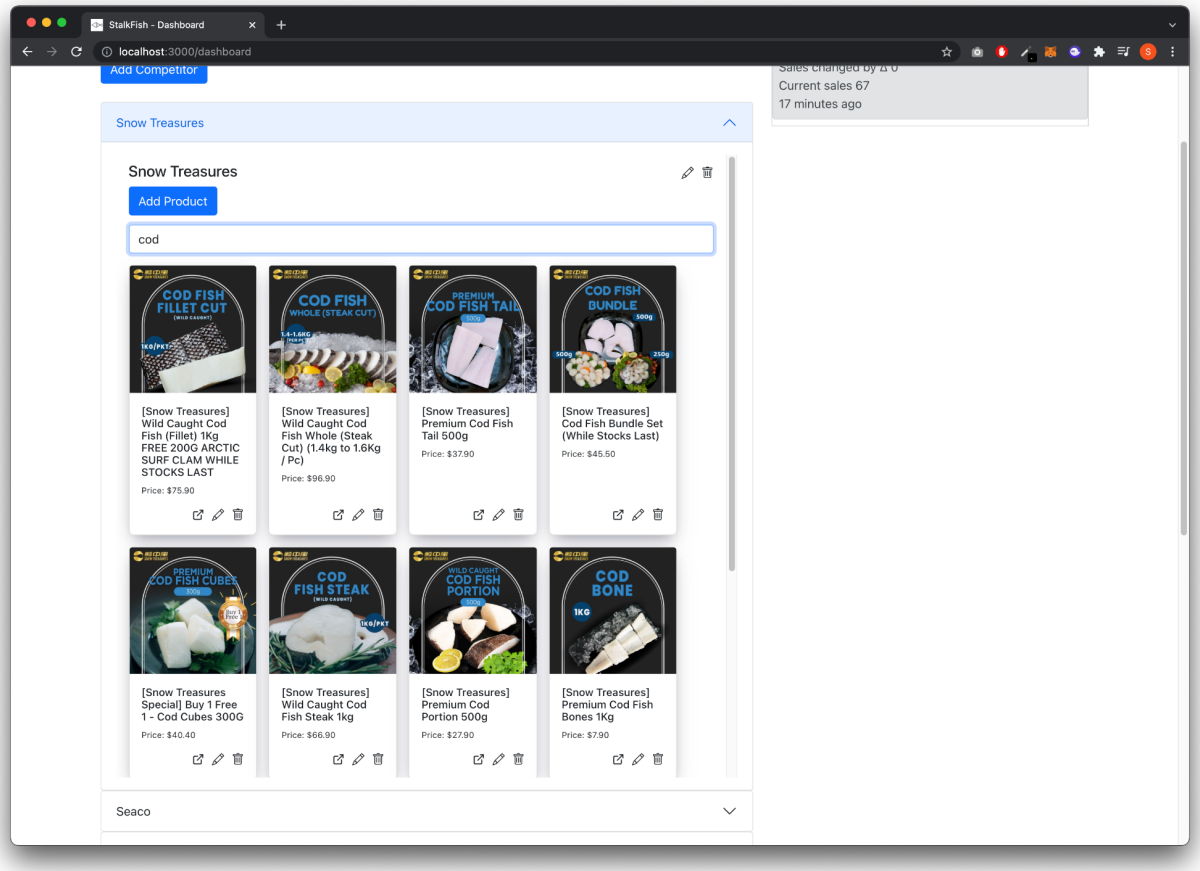



*Figure 5.2.1a: Search and Filter Products within Store using Product Name*

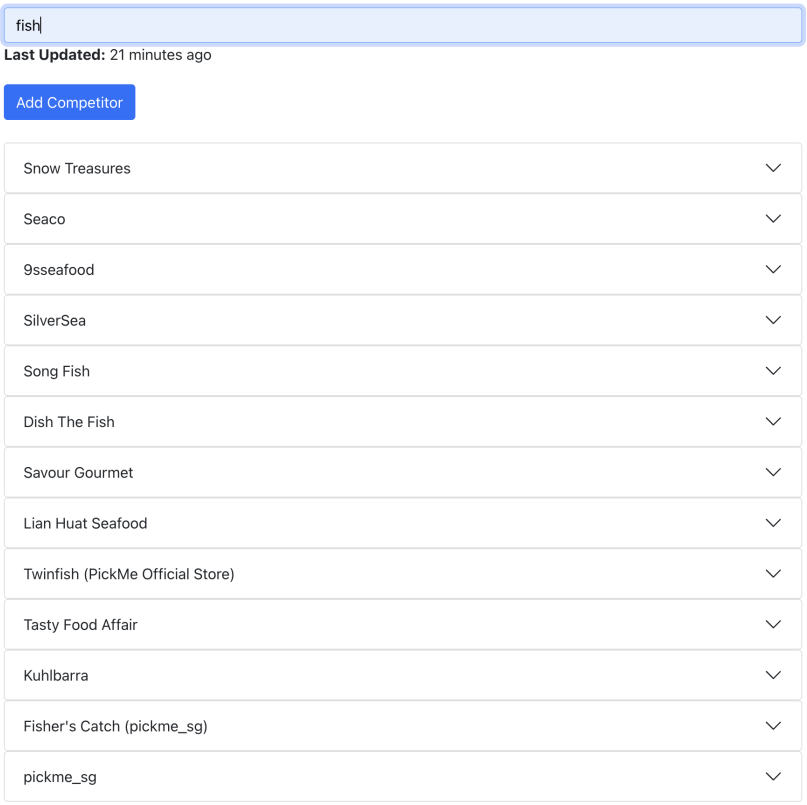



*Figure 5.2.1b: Search and Filter Stores by Product Name*

We have provided 2 ways for users to search and filter within the app, fulfilling **[FR2.1]**. Figure 5.2.1a shows how a user may use the search bar within a store to search for a specific product. Figure 5.2.1b shows how a user may use the search bar at the top of the dashboard to filter all stores that carry a specific product.

## 5.2.2 CRUD Operations for Stores and Products

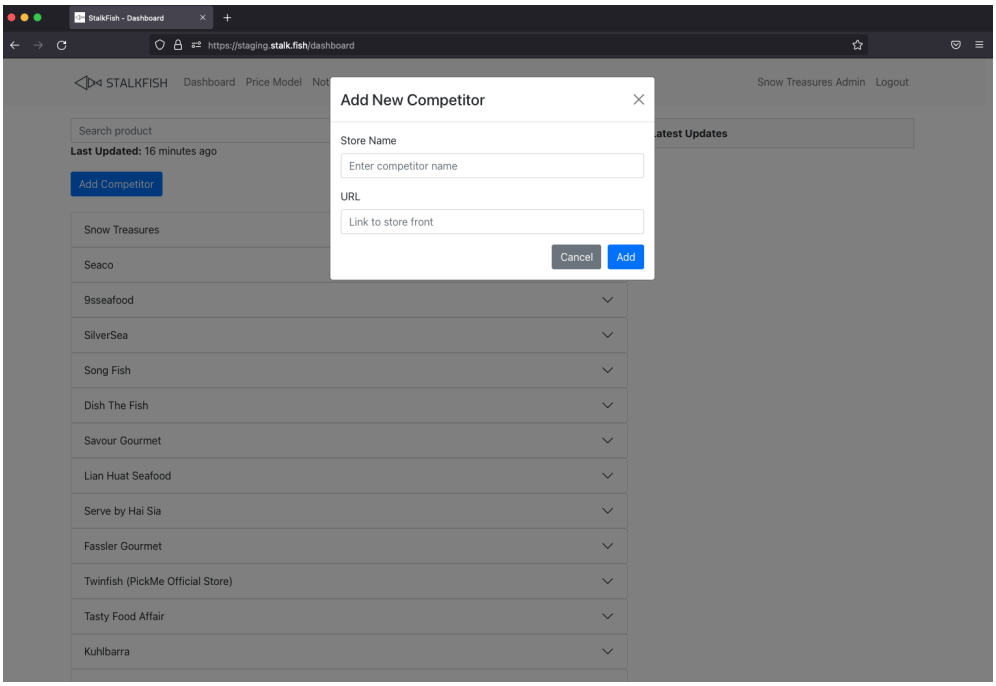



*Figure 5.2.2a: Adding a New Competitor Store*

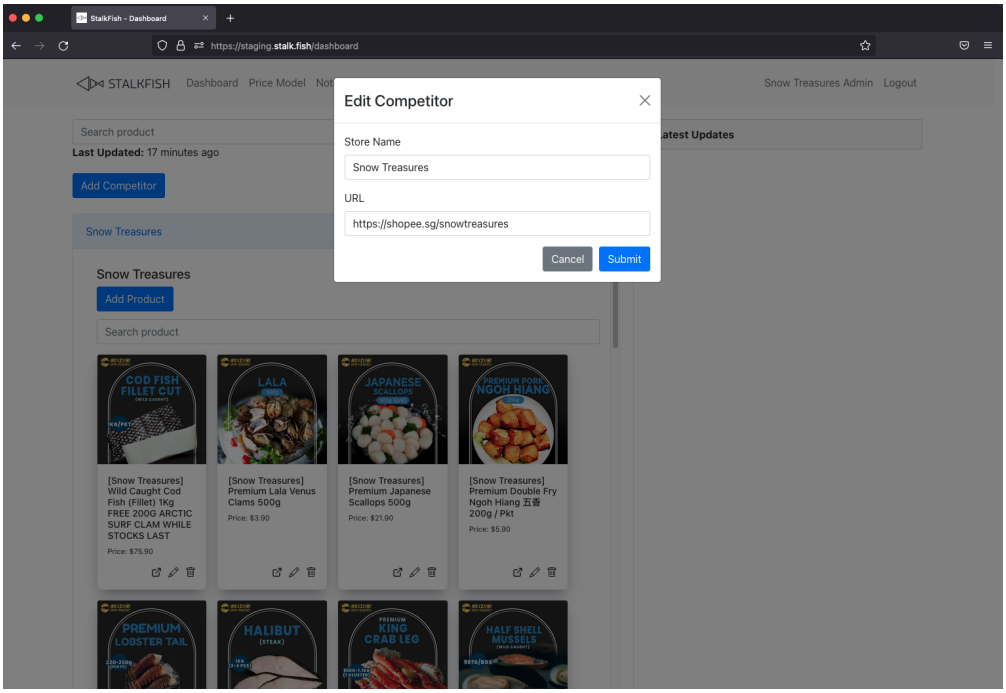



*Figure 5.2.2b: Editing an Existing Competitor Store*

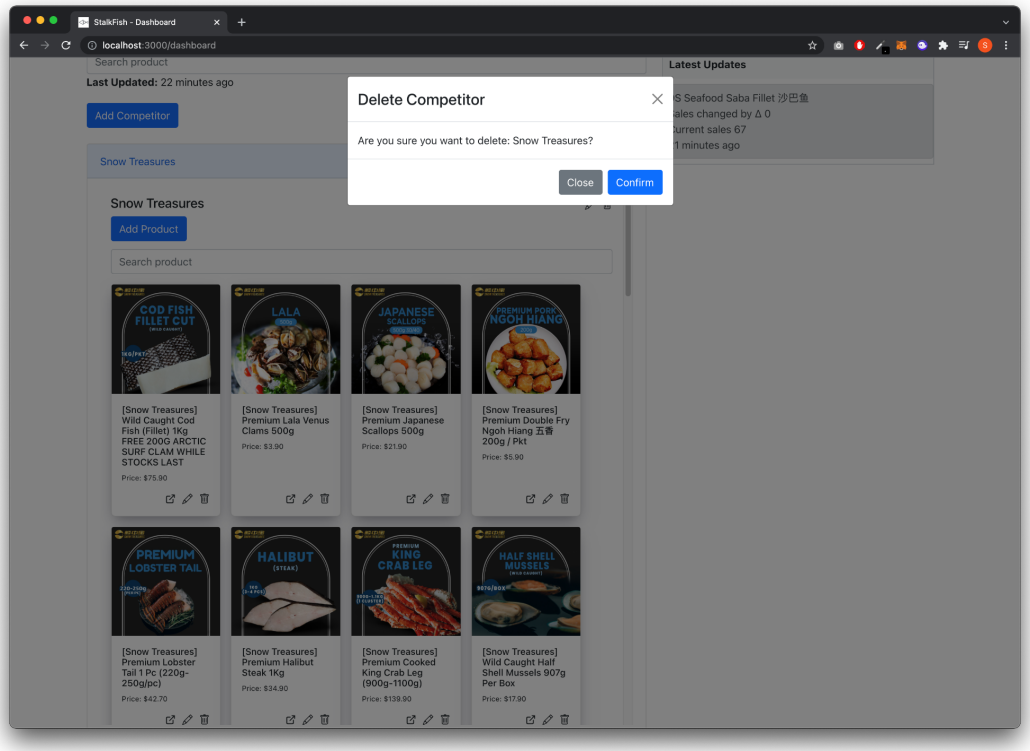



*Figure 5.2.2c: Deleting Existing Competitor Store*

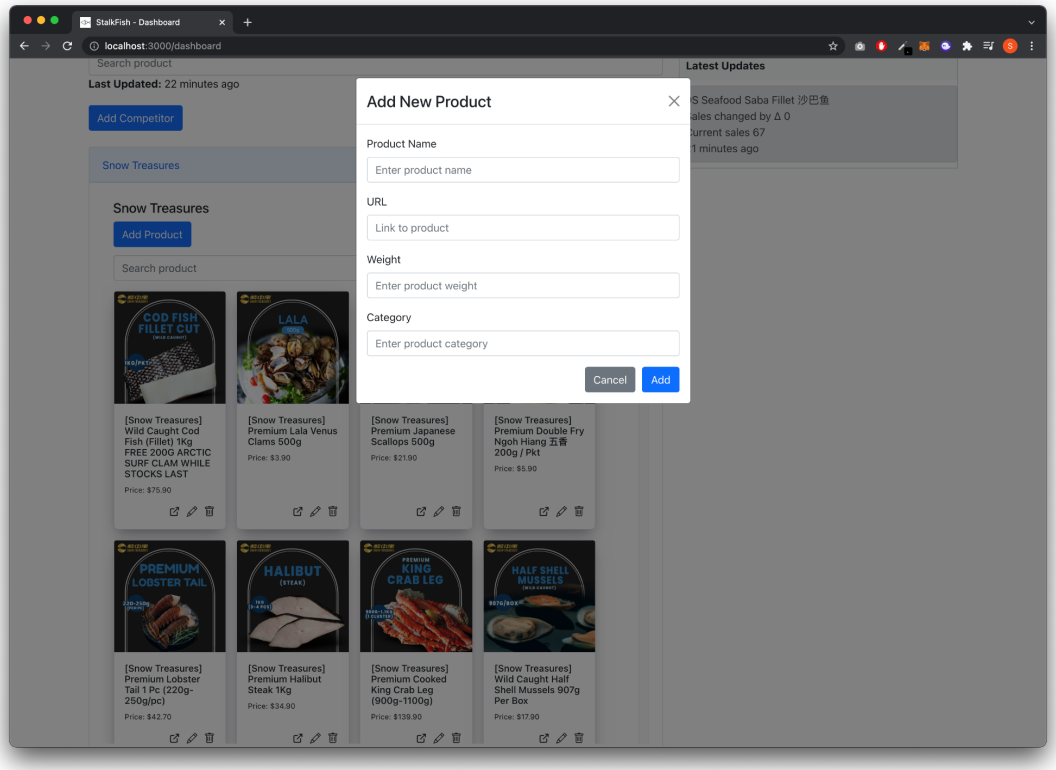



*Figure 5.2.2d: Adding a New Product*

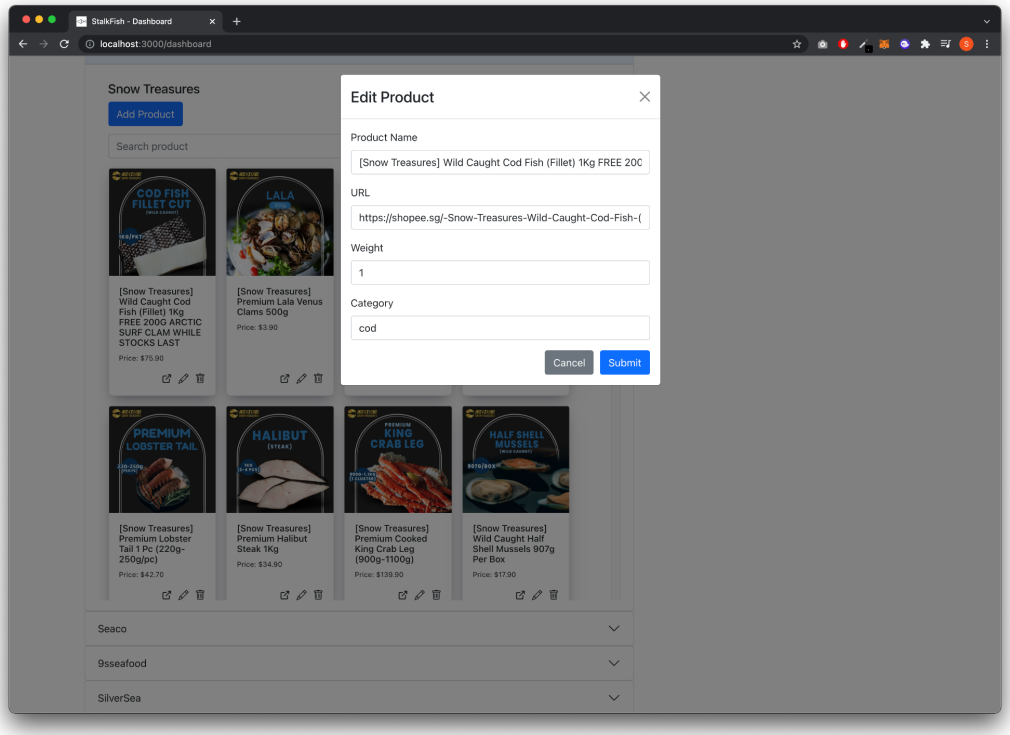



*Figure 5.2.2e: Editing an Existing Product's Details*

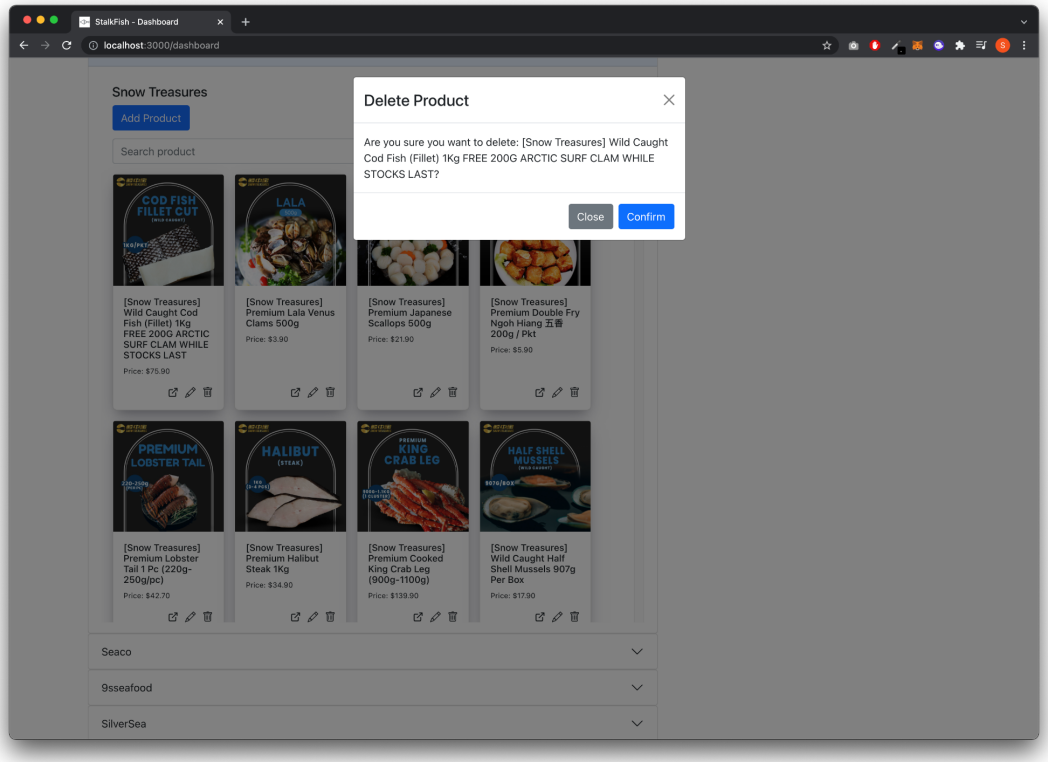



*Figure 5.2.2f: Deleting an Existing Product*

We have implemented CRUD operations for stores and products, fulfilling **[FR5.1]**. When users add a product or edit an existing product's URL, scraping is performed on the backend to retrieve the product's information automatically, fulfilling **[FR1.2]**. Once scraping is completed, the product will be displayed on the dashboard.

### 5.2.3 Notifications

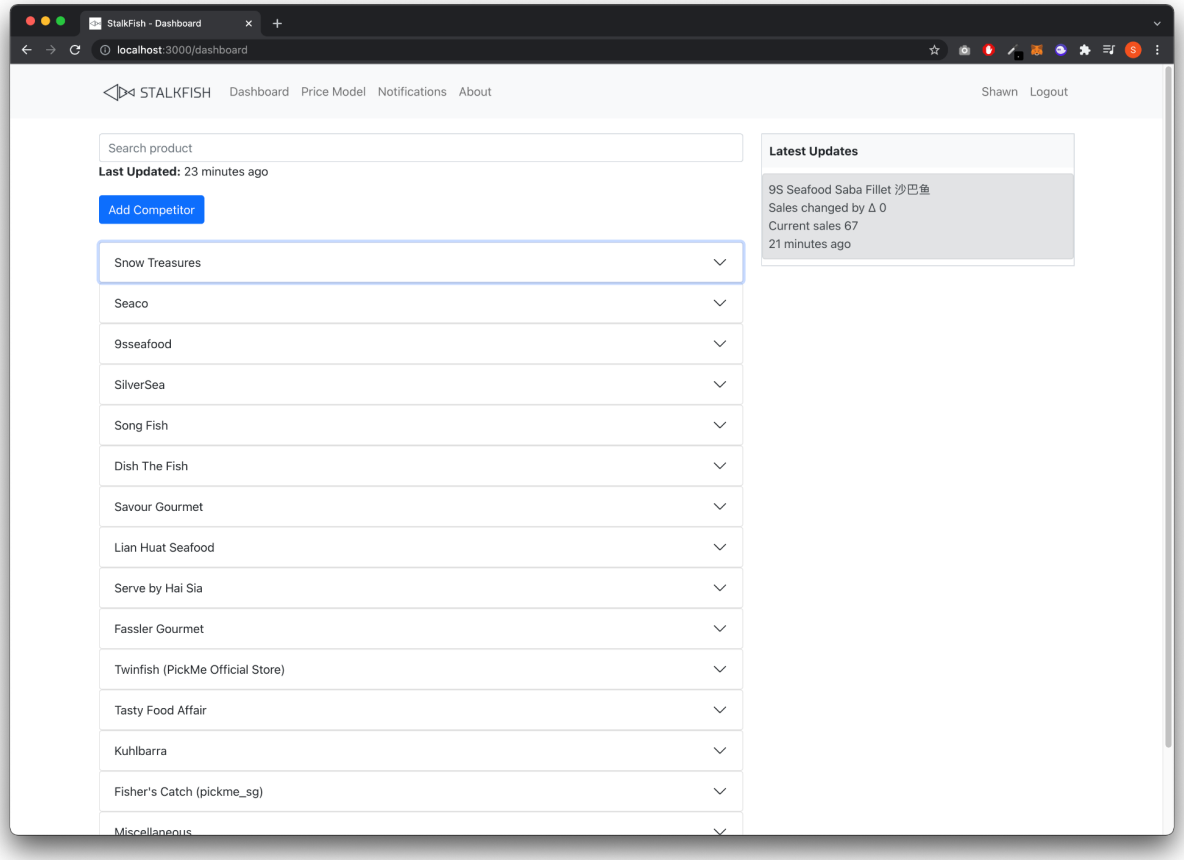



*Figure 5.2.3a: Notifications on Dashboard*

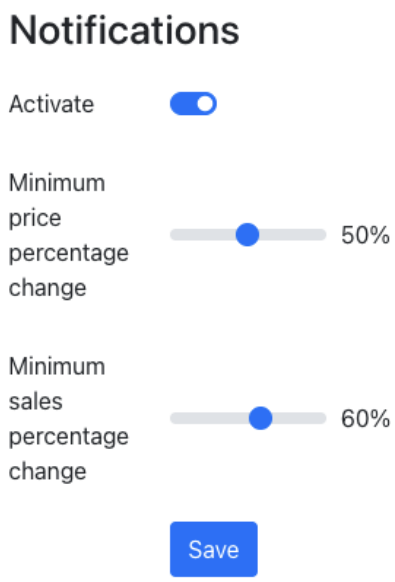



*Figure 5.2.3b: Configuring Notifications for a Product*

We have provided the ability for users to subscribe to price and sale percentage changes for products, fulfilling **[FR6.1]**. Figure 5.2.3b shows the configuration panel found on the product details page when users click on a product on the dashboard.

Notifications will be sent out to users via a Telegram bot whenever the threshold they set for price and sale change percentage is hit, fulfilling **[FR6.2]** and **[FR6.3]**. We also display the latest notifications on the right on the main dashboard as seen in Figure 5.2.3a.

Users can link their Telegram account via the ‘notifications’ tab in the navbar as seen in Figure 5.2.3c below.

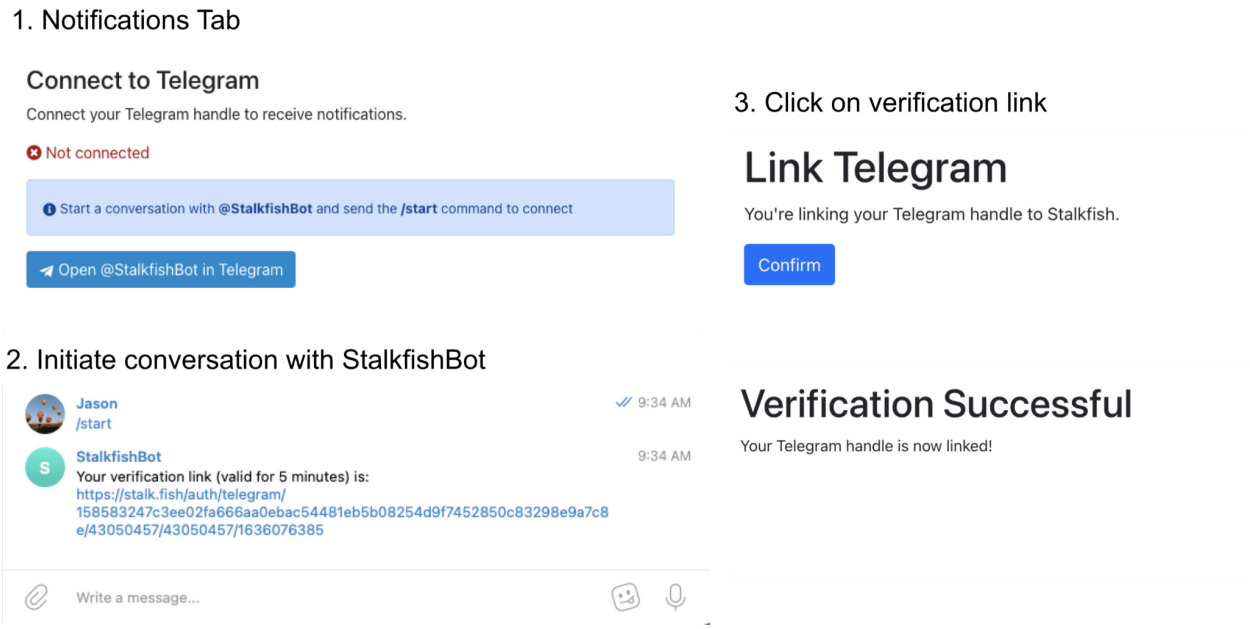



*Figure 5.2.3c: Telegram authentication flow*

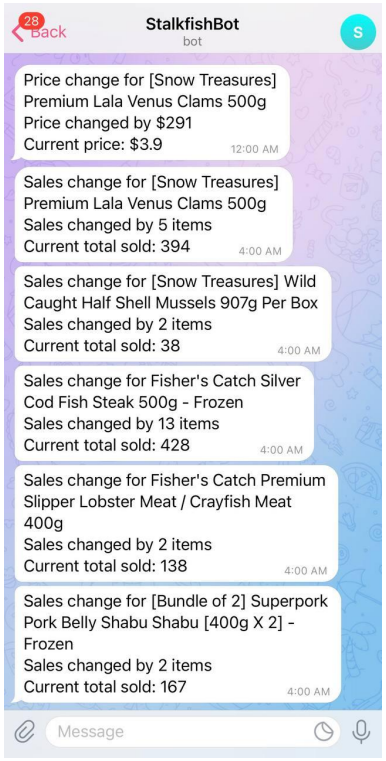



*Figure 5.2.3d: Telegram notifications*

Users will receive notifications in the form of Telegram messages as seen in Figure 5.2.3d. Notifications are sent to individual Telegram accounts subscribed to the product, and each user can choose their own threshold to trigger notifications, as seen in Figure *5.2.3b* earlier.

# 5.3 Product Details and History

## 5.3.1 Product Details

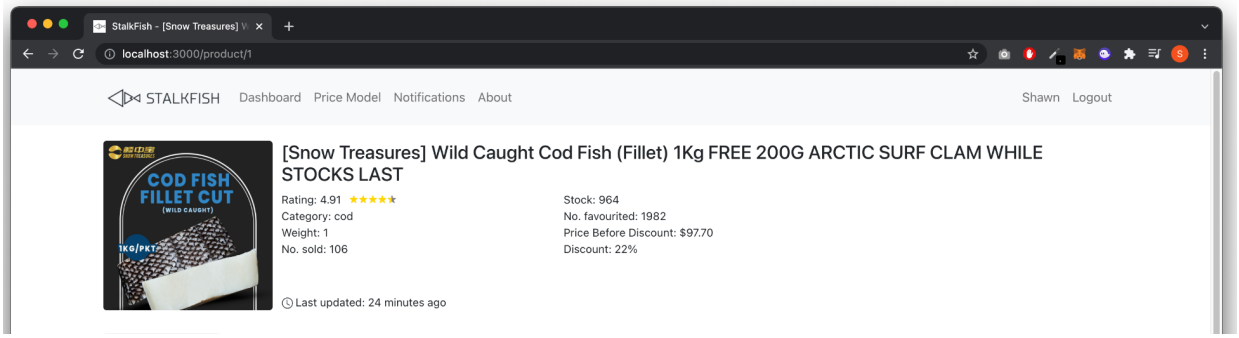



*Figure 5.3.1a: Product Details*

When users click on a particular product on the dashboard, they are brought to the product details page. Here, users are able to see more details about the product such as the rating on Shopee, number of products sold, number of stock available and more.

## 5.3.2 Product History

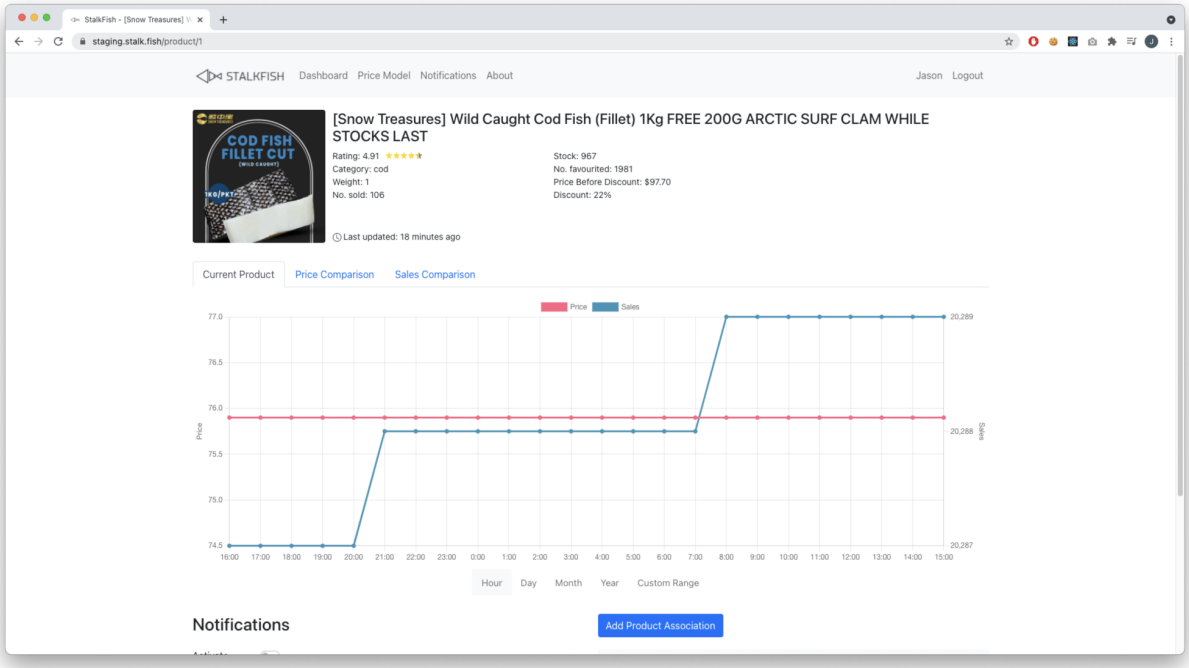



*Figure 5.3.2a: Product History*

The product details page also shows the price and sales history in the form of a graph, fulfilling **[FR5.3]**. By default, it shows data over the past 24 hours, and users are able to control the time range they wish to view.

The other tabs of the graph allow the user to compare associated products, which is elaborated upon in the sections below.

### 5.3.3 Product Association


*Figure 5.3.3a: Product Association list*

To allow for more useful comparison among competitor products, the product history page lets users add ‘Product Associations’, fulfilling **[FR5.4]**. Product associations are products of the same type that the user would like to compare with and they have a transitive closure property.

For example, if you add the following product associations for [Snow Treasures] Wild Caught Cod Fish Fillet:

- Silversea - Cod Fish Steak 1KG [Wild Caught Sea Frozen]
- Silversea - Cod Fish Belly Cube 200G

Visiting the product history page for ‘Silversea - Cod Fish Belly Cube 200G’ will show the following product associations:

- [Snow Treasures] Wild Caught Cod Fish Fillet
- Silversea - Cod Fish Steak 1KG [Wild Caught Sea Frozen]

The team decided on this behaviour for the convenience of the users - so the user does not need to individually add associations that are one-way. Instead, the associated products will automatically appear on the corresponding products as a result of the transitive closure property.

When a product has at least 1 product association(s), the ‘Price Comparison’ and ‘Sales Comparison’ tab will be enabled. When a user views the comparison tab, all associated products are plotted on the same graph as seen in Figure 5.3.3b. This gives the user an overview of the price and sales changes among all products.

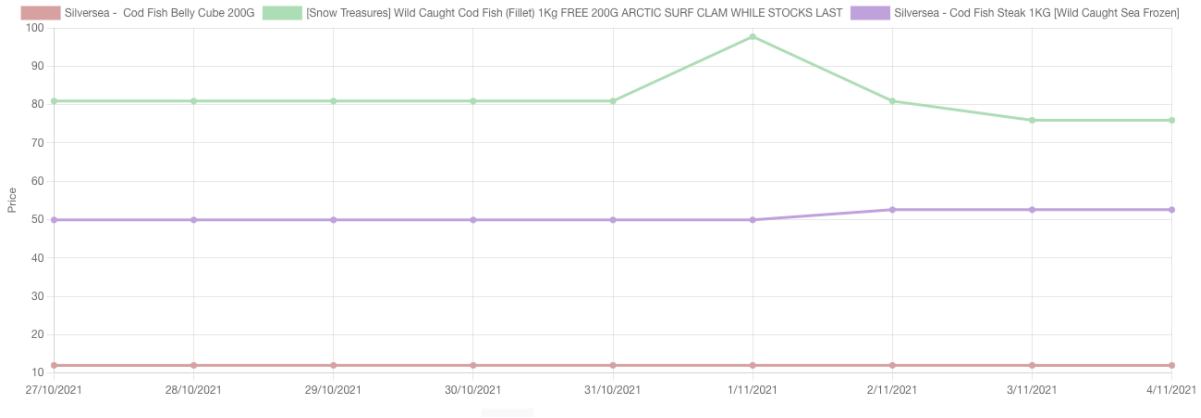



*Figure 5.3.3b: Comparison graph*

# 5.4 Store Details

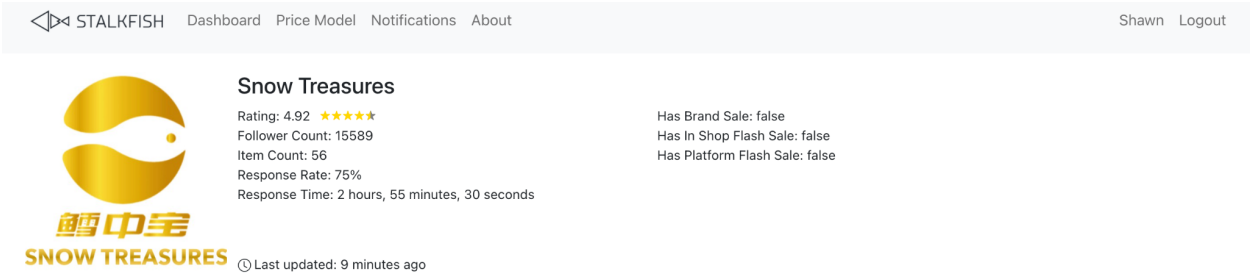



*Figure 5.4a Store Details*

A higher level of customer service helps to set our client apart from their competitors. Metrics such as response rate, response time and discounts are therefore important to keep track of. When users add a store and include a URL, StalkFish will scrape these details from the Shopee storefront, and update them periodically with half hourly scrapes. Users can click on a specific store's name under the accordion on the dashboard in order to view its details, fulfilling **[FR1.1]**.

# 5.5 Price Modelling

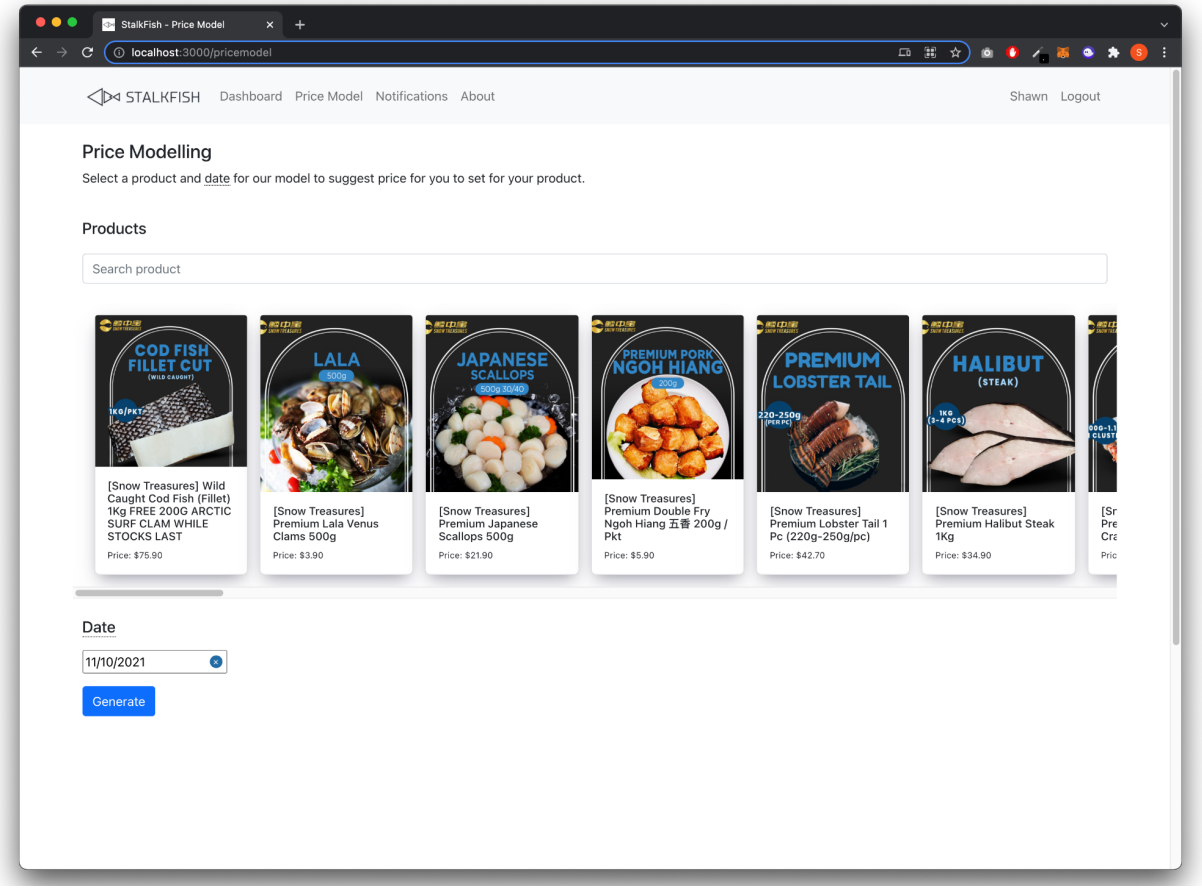



*Figure 5.5a: Price Model Page*

Using the price model, users are able to generate a forecast for future sales. Users can access the price model using “price model” in the navigation bar as shown in Figure 5.5a.

From there, users specify a product and date for the model to use to forecast the number of sales, then click on the “Generate” button to generate the graph as seen in Figure 5.5b below.


*Figure 5.5b: Generated Price Model Graph*

The number of sales can be found on the y-axis, and the x-axis shows the time from the selected date (0 being the selected date, -50 being 50 hours before the selected date and 20 being 20 hours in the future from the selected date). Observed data can be ignored if we select a date that is more than 2 days in the future since we do not have data from the past 50 hours.

This was a key feature requested by the client, to better inform the sales and procurement teams about future demand.

For more details on the price modelling algorithm, refer to Section 6.2.7 below.

# 6 StalkFish Design

## 6.1 Overview

### 6.1.1 Chosen Design

The diagram below illustrates the overall architecture of StalkFish.

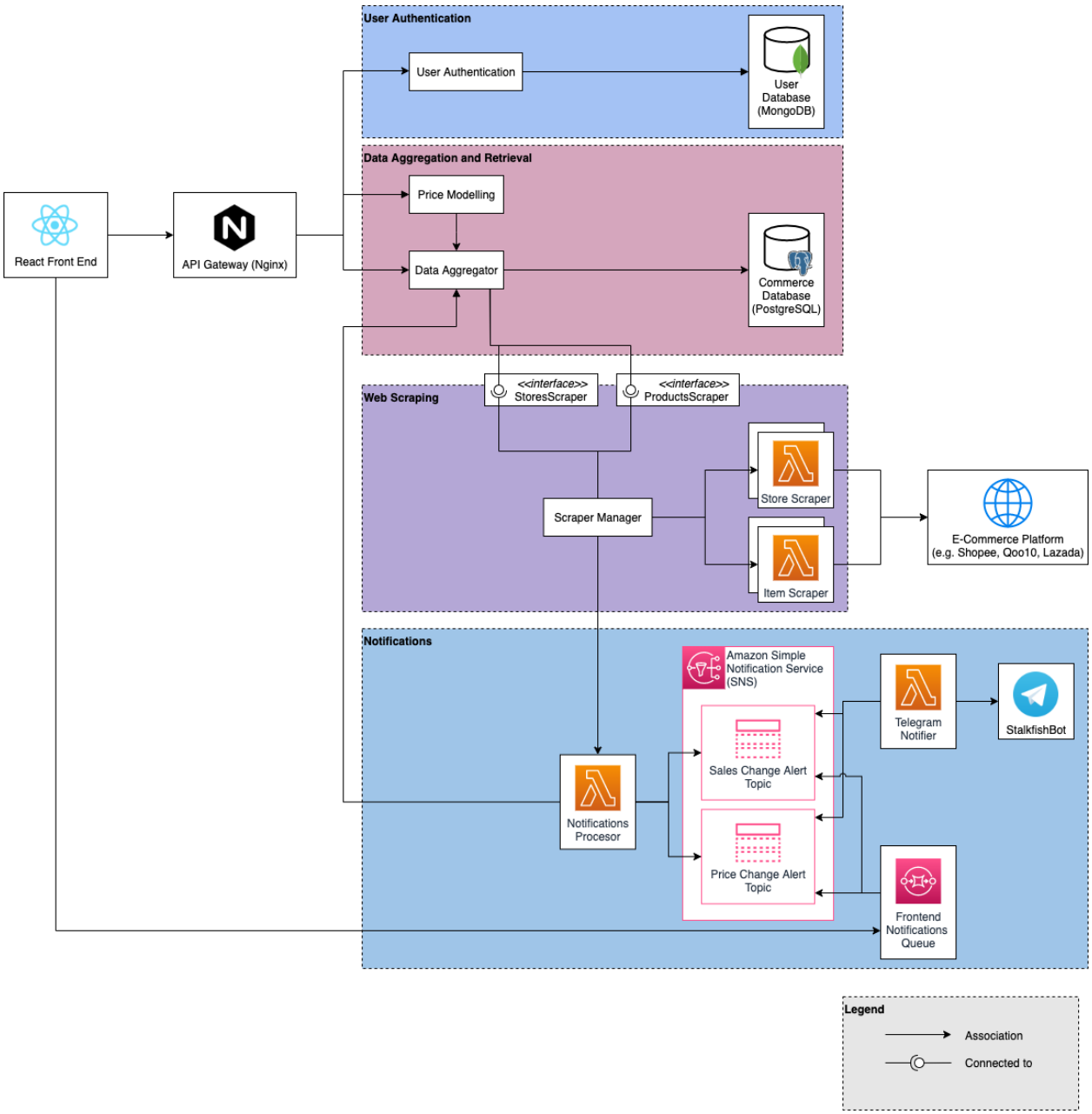



*Figure 6.1.1a. Overall Architecture Diagram of StalkFish.*

### 6.1.2 Alternative Design

While the microservices architecture was used for StalkFish, a monolithic architecture might have worked in the client's use case. At the moment, there are only approximately 730 items added by the client, which a monolithic architecture could have easily handled. Such a stack could potentially run on a singular backend and singular database, responsible for all the features implemented.
However, there are a few considerations between the two which are as detailed below.

| | **Microservices Architecture** | **Monolithic Architecture** |
|---|---|---|
| **Ease of implementation** | As the chosen microservices architecture consists of many different components, it is harder to implement them all and discover APIs for all of them | With a singular codebase, APIs need not span multiple protocols and different languages. |
| **Ease of collaboration** | As each component has clear boundaries, and are loosely coupled, it is easy to work in parallel as a team. It is also less risky to implement new features in a component as it only breaks that one component. | As the components are very tightly coupled, it would require more effort to ensure unit testing is done well, and to ensure merge conflicts are minimised. Further, as the codebase grows, it would get too complex to understand. |
| **Ease of deployment** | Much harder to deploy across different services and providers. More effort required to implement monitoring/logging. | Simpler to manage a singular deployment. |
| **Scalability/Performance** | With the chosen architecture, it is capable of handling more users, and many more items for the future when more companies decide to open an online storefront. | More users may slow down processes, and given that all the requests are fired from a single machine, it may exceed Shopee's rate limit. Furthermore, it might be impossible to deploy the application across multiple replications to improve load balancing. |
| **Reliability** | With multiple components, there are multiple points of failure. However, this can be somewhat mitigated with the monitoring put in place. | With a singular component, it is less likely that the server would be down, as opposed to multiple services on different machines. However, as testing is harder, it is more likely to run into issues with the codebase. |

| | | |
|---|---|---|
| **Security** | With multiple services, there is clear separation of concerns, and components that need not be responsible for authentication do not have access to that pipeline. | With a singular application and database, there is a higher possibility of accidentally leaking authentication information out. |

## 6.2 Components and Services

### 6.2.1 Frontend

The frontend was developed using the React Javascript library - which allows for fast and scalable development.

The frontend displays the information about the stores and their products on the main dashboard page. The dashboard allows users to create/edit/delete stores and products, as well as view the latest notifications sent out by the application.

From the dashboard, users can navigate to the item profiles by clicking on their respective cards. From the item profile, users can view more details of the item such as its rating and number sold. Users are also able to view the price and sales history of the product and configure the time frame displayed. On top of this, users can also view and add product associations, linking the product to other products. From here, users are able to compare the price and sales of different products on a single graph. Finally, users can register to receive notifications for the particular product on this page.

Additionally, from the dashboard, users can navigate to the price model page. On this page, users are able to select a product for us to generate price and sales predictions in the form of a graph using historical data. We have also provided users with fine control over the date range for the data to be used.

Figure 6.2.1a below depicts the user flow for navigating the frontend:

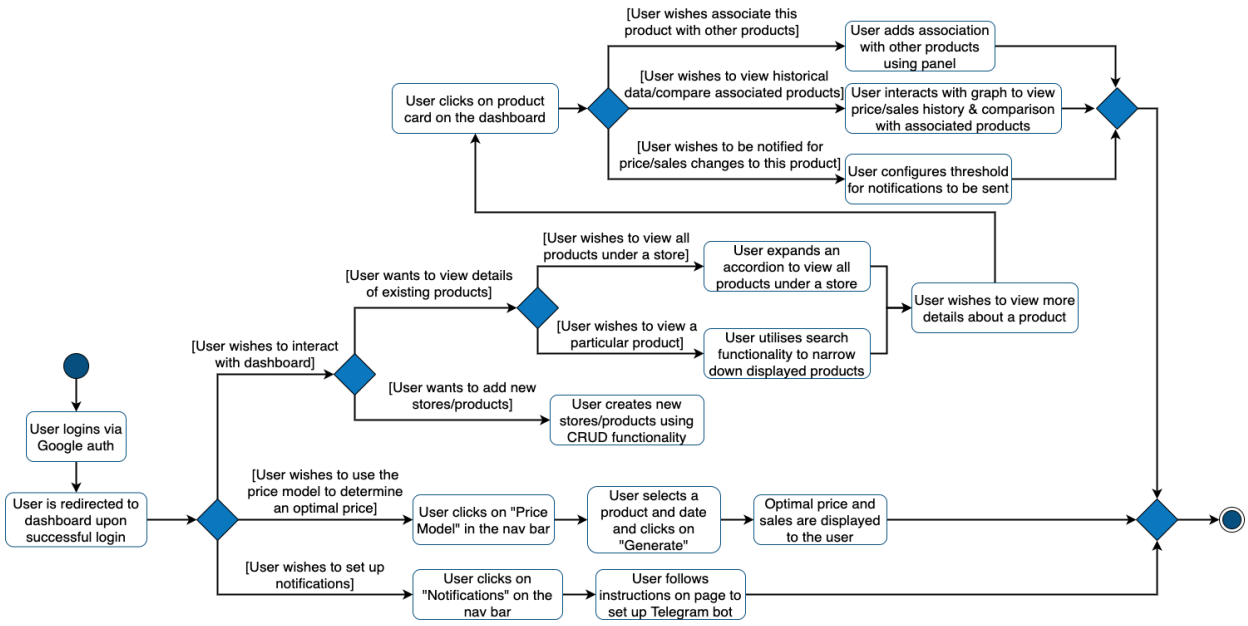



*Figure 6.2.1a: Activity Diagram for Frontend User Flow*

Refer to Section 5 for more details on StalkFish's features along with screenshots.

## 6.2.2 API Gateway

In order for the frontend to access the various microservices, StalkFish uses an API Gateway to provide a unified entry point to the backend. This API Gateway behaves as a façade over multiple microservices by taking care of the routing, load balancing and rate limiting.

To pick the right tool, we evaluated a few options with the following criteria:

- High availability, as the API Gateway is critical for the frontend to get the data it needs.
- Reliable under many concurrent connections. This ensures that the API Gateway does not become a bottleneck in handling requests.
- Integrates with Kubernetes primitives for configuration, scaling, logging, and Service Discovery.

Ultimately, Nginx was chosen for the API Gateway as it fulfils the above criteria, in addition to being supported by the Kubernetes project.

With the Kubernetes integration, the API Gateway uses the built-in Service Registry to route and load balance to the appropriate microservice instance. Furthermore, this abstraction provides scalability by allowing each microservice to scale independently using multiple instances, without requiring any client knowledge. This isolates the frontend from the operational complexities such as failover and service upgrades.

### 6.2.3 Authentication

#### 6.2.3.1 User Authentication

StalkFish's authentication is handled via Google's OAuth 2.0, which was also a recommendation by the project team. After consideration, the team decided to link our authentication to Google's OAuth instead of setting up our own from scratch. The following were our key considerations:

- Google OAuth is secure and managed by Google
- On the user's end, they do not need to remember an additional set of credentials
- External email addresses e.g. Yahoo etc. can also sign up for a Google Account and login via Google
- Time is also saved on having to develop our own suite of authentication flow for logging in, signing up, and password reset

The team also wanted to explore various database systems and opted for a non-relational database system for our simple User database. We have chosen MongoDB as our choice due to its ease of use and popularity.

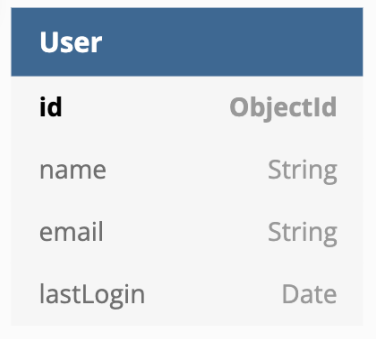



*Figure 6.2.3.1a: User model*

As we are using the microservices architecture, a token based authentication as opposed to session based would be more ideal. Hence, the team decided to make use of JSON Web Tokens (JWTs) which is generated upon successful authentication via Google.

API requests to our API Gateway are authenticated using JWTs and used to verify whether a user is authenticated and/or authorised to access certain API endpoints.

For requests made to the Hasura GraphQL Engine, JWT is used to authenticate the user. Additionally, the 'x-hasura-allowed-roles' and 'x-hasura-default-role claims' are used to authorise requests e.g. users are only allowed to query their own registered Telegram handles.

A sample of the JWT payload is shown in Figure 6.2.3.1b below.


*Figure 6.2.3.1b: JWT payload*

### 6.2.3.2 Telegram Authentication

To support our Telegram notification service, users are required to connect their StalkFish account to their desired Telegram account. Telegram requires that the user initiates contact with the bot in order to obtain the user's Telegram and Chat id.

We make use of Telegram Webhooks to detect when a user sends a verification initiation message (/start) and generate a verification link. The webhook endpoint points to an AWS Lambda function. Since webhooks serve the purpose, using a serverless function is the best approach as we do not have to keep a server running specially to detect/poll for new messages

- the webhook endpoint is simply accessed by Telegram whenever a message is received by the bot.

In summary, the user first uses the **/start** command on the **@Stalkfishbot** Telegram Bot and after which, the bot will respond with a unique verification link valid for 5 minutes. Clicking on the link sends the user to the verification page which is only accessible to authenticated users. If the Telegram authentication link provided is valid, the user's Telegram account will be successfully linked.

The Telegram connection/authentication flow is seen in Figure 6.2.3.2a below.

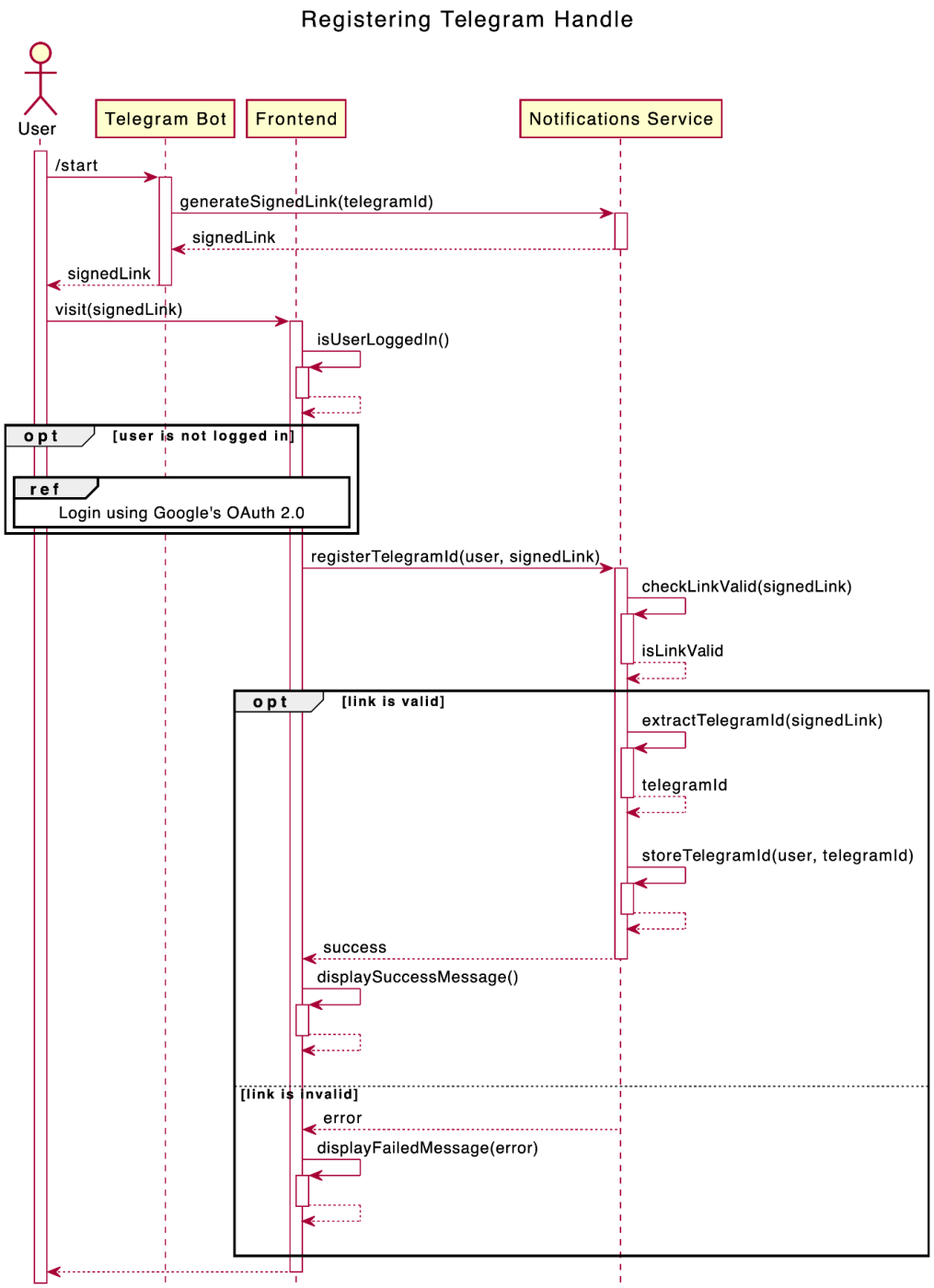



*Figure 6.2.3.2a: Sequence Diagram of Telegram Authentication*

#### 6.2.3.3 Telegram Verification

The conventional way for generating verification links, password reset links etc. would be to randomly generate a unique verification code and add it to the database. However, the team thought of an alternative to workaround having to create additional tables for verification by using simple cryptography concepts like HMAC (keyed-hash message authentication code).

When a user requests for the verification link, a HMAC is created with the following: <user's telegram id>-<user's chat id>-<expiry>. The hash function used is SHA256.

When verifying a particular verification link, we will verify that the message digest was not altered and check that the link has not expired. After which, we store the telegram_id and chat_id in the telegram_handle table in the notifications database.

There are also drawbacks to using this approach, as compared to generating links and storing them in the database, such as the private key being leaked. However, because the contents we are working with are not as sensitive in nature, and the private key is solely used for generating verification links, we found this to not be a security risk. The key can also be refreshed periodically, if required.

### 6.2.4 Data Aggregation and Retrieval

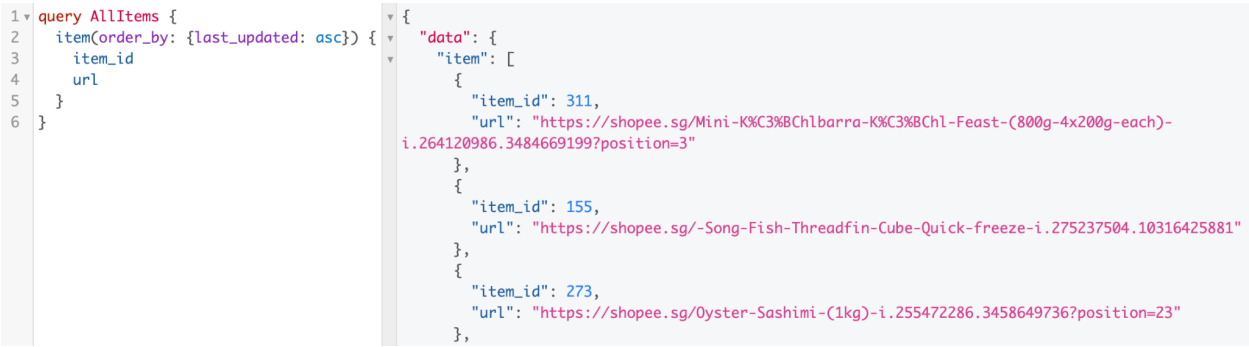
The Data Aggregation and Retrieval service is the custodian of all Stores and Products data. As such, other services that require such data must interface with the service via an API. Since StalkFish is primarily an analytical workload, the API must be designed to support access to large amounts of data at once. In addition, we restricted our API transport protocol to HTTP as we wanted frontend compatibility. These restrictions gave us two options in how we could design the API, either using REST or GraphQL. These options were evaluated against the following criteria:

- Performance in terms of latency.
  The API will be used by users through the frontend and a non-responsive API directly results in a poor user experience.
- Minimizes turnaround time for new features or improvements.
  As other services depend on this service to do their work, not providing the necessary data may block the development progress of other services.

In terms of performance, while REST is fast for manipulating a single resource or a series of resources with the same type, fetching nested resources requires multiple round trips which increases latency. Due to the hierarchical nature of the Stores and Products data, this would occur frequently. Caching could be employed to reduce response latency, but it cannot eliminate the network latency caused by multiple roundtrips. Another option would be to embed nested resources, however this means services will receive some data that it does not need.

With regards to turnaround time, REST requires the endpoints to be implemented in order to manipulate a resource. This means that developer resources must be spent on writing and maintaining boilerplate code, especially if we chose to embed nested resources.

Based on the reasons discussed in the two paragraphs above, a GraphQL API was implemented using Hasura. Choosing GraphQL instead of REST significantly sped up the development of our other services. We could provide a GraphQL schema of the resources available and other services could query for exactly the data they needed without having to wait for a REST endpoint to be added or modified. For example, the two figures below show how two different services, the Scraper Manager and the Notification Service, can specify only the data needed.



```
query AllItems {
  item(order_by: {last_updated: asc}) {
    item_id
    url
  }
}
```


```
{
  "data": {
    "item": [
      {
        "item_id": 311,
        "url": "https://shopee.sg/Mini-K%C3%BChlbarra-K%C3%BChl-Feast-(800g-4x200g-each)-i.264120986.3484669199?position=3"
      },
      {
        "item_id": 155,
        "url": "https://shopee.sg/-Song-Fish-Threadfin-Cube-Quick-freeze-i.275237504.10316425881"
      },
      {
        "item_id": 273,
        "url": "https://shopee.sg/Oyster-Sashimi-(1kg)-i.255472286.3458649736?position=23"
      },
```


*Figure 6.2.4a: Scraper Manager Querying for Data*

```
query ($item_id: Int!) {
  item_by_pk(item_id: $item_id) {
      item_histories (limit: 2, order_by: {time_accessed: desc}) {
          name
          price
          sold
          time_accessed
      }
  }
}
```


```
{
  "data": {
    "item_by_pk": {
      "item_histories": [
        {
          "name": "Mini Kühlbarra Kühl Feast (800g - 4x200g each)",
          "price": 4122,
          "sold": 41,
          "time_accessed": "2021-11-03T16:30:02.108+08:00"
        },
        {
          "name": "Mini Kühlbarra Kühl Feast (800g - 4x200g each)",
          "price": 4122,
          "sold": 41,
          "time_accessed": "2021-11-03T16:00:01.654+08:00"
        }
      ]
    }
  }
}
```


*Figure 6.2.4b: Notification Service Querying for Data*

To handle the persistence of the Stores and Products, we knew that a relational database was the best choice, due to the hierarchical nature of the data. We chose the PostgreSQL database as it is fully featured, widely deployed and familiar to the team. The Stores and Products are represented using the following database schema.

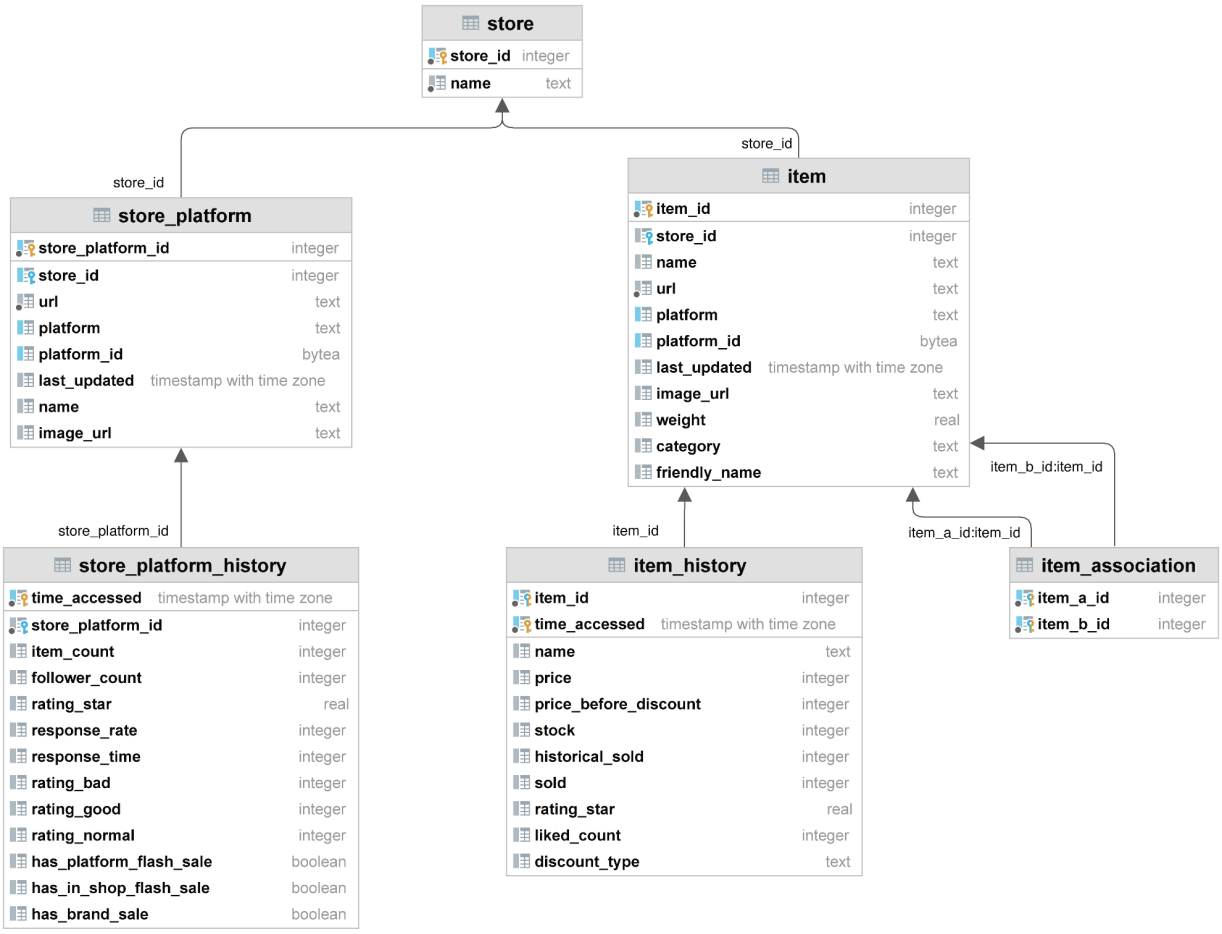



*Figure 6.2.4c: Database Schema*

The store table represents a single Store, which can be thought of as a competitor. Each store can have many store_platforms and these represent the various e-commerce platform storefronts that a competitor can have. For example, one company with accounts on both Shopee and Qoo10 would be represented as a single entry in the store table with two linked entries in the store_platform table. This design offers two advantages. The first is user simplicity, as our users would not think of these different accounts by the same company as different Stores. The second is of application extensibility, allowing StalkFish to support other e-commerce platforms by writing new Scrapers and updating the Frontend. The historical record of a particular storefront's metrics is kept in the store_platform_history table.

The item table represents a single Product that a Store has. It has a pair of attributes called friendly_name and name which may be confusing at first. The friendly_name represents a user-assigned name for the Product whereas the name represents the Product name as it appears on the e-commerce storefront. The latter tends to be longer, as it includes additional information that would entice potential customers. Having the friendly_name attribute allows our StalkFish users to specify a succinct name for their own internal use, while still being able to see how their competitors are labelling their Products. Like the Stores, a Product's historical

records are stored in the item_history table. But, unique to the Product is the item_association table. This table supports the Product Association feature as previously discussed in Section 5.3.3.

Additionally, the store_platform and item tables have the two attributes, platform and platform_id. These act as unique, somewhat stable identifiers to a Store and Product as provided by the e-commerce platform and are only used by the Scrapers to update the correct records. These identifiers were intentionally not used as part of our own identifiers as there was no guarantee that they would not change.

### 6.2.5 Web Scraping

Web scraping was implemented as an AWS Lambda. Each Lambda is responsible for a single product, as opposed to processing products in batches.

This was chosen as it easily enables scalability, as each Lambda is it's own instance, allowing for quick asynchronous fetches of product data. As such, it is not limited to the performance of a singular server instance, which could potentially bottleneck and hence miss a data point on the hour.

This however, could potentially lead to issues with pricing and performance, if there were many products, and hence many requests being fired off by the Scraper Manager. However, since each request is resolved in a matter of seconds, a dedicated instance would sit idle for the rest of the time, potentially costing more. Further, since all requests are fired at the same time, less the e-commerce site being down, it is unlikely to take longer than the scraping interval, despite the latency associated with using the serverless architecture.

### 6.2.6 Notifications

The diagram below illustrates an overview of the components that make up the StalkFish Notifications service.

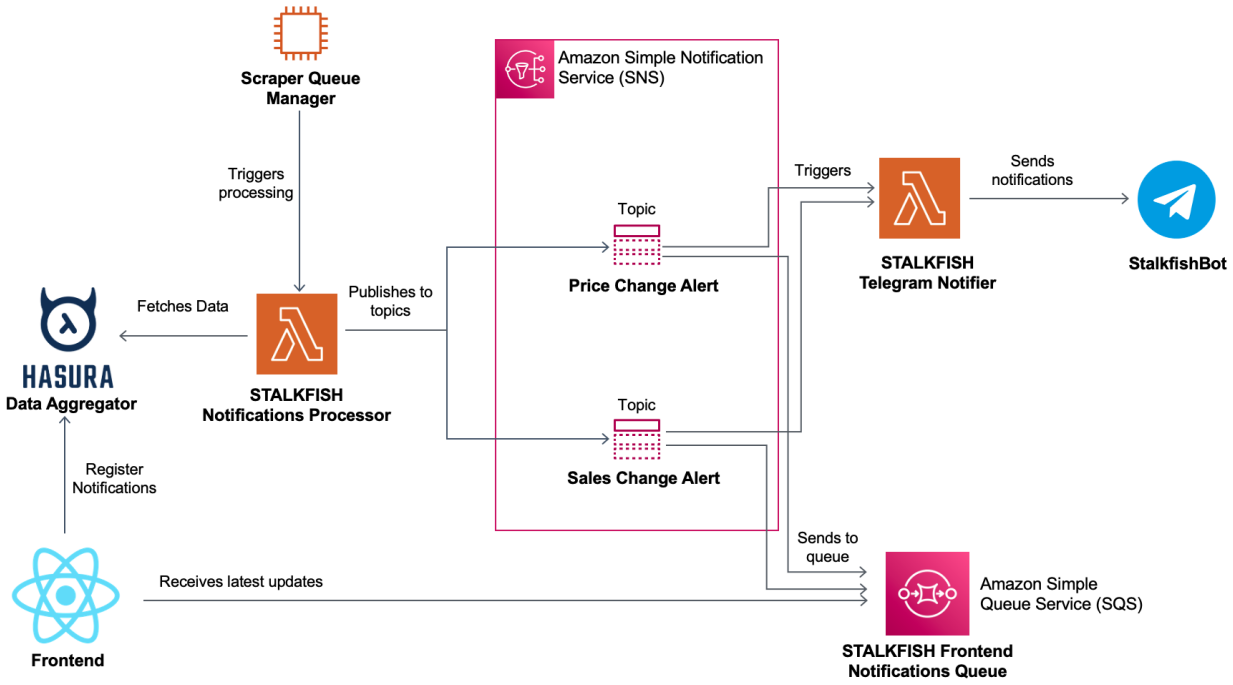



*Figure 6.2.6a: Diagram of notifications service*

Notifications are first registered through the React frontend user interface, which adds to the database in the following schema:

| registered_notification | |
|---|---|
| user_id | text |
| item_id | integer |
| price_percent_change | integer |
| sales_percent_change | integer |

The Scraper Manager then periodically triggers the StalkFish Notifications Processor after every scraping (approximately half an hour per run). This is when the StalkFish Notifications Processor fetches data from the Hasura Data Aggregator, where registered notifications are fetched along with the latest scraped data. During the processing, registered notifications which have exceeded the defined threshold (i.e. price_percent_change or sales_percent_change) would be published to the Amazon's Simple Notification Service (SNS) topics.

We utilized Amazon's SNS, which provides a high-throughput, pushed-based messaging service between event-driven serverless applications and other microservices. It provides a Publish-Subscribe model that reduces complexity by removing point-to-point connections by replacing it with connections to a topic, that manages subscriptions that decide which notifications are delivered to which endpoints. This design provides us the ability to easily extend our design to include new notification destinations, such as email, mobile text messaging or web push notifications.

Amazon’s SNS works especially well with Amazon’s other services such as its Simple Queue Service (SQS) and Lambda. We have purposed an AWS Lambda “StalkFish Telegram Notifier” as a subscriber to the SNS notification topics. The Lambda provides the functionality of sending out the actual notifications to the respective users over the Telegram bot.

Another subscriber to the SNS topics is the SQS “StalkFish Frontend Notifications Queue”, a fully managed message queue which has the responsibility of storing the notifications that are published to the SNS topics. This enables a user to view the notifications when visiting the React frontend web application, where the notification messages are retrieved and displayed on the latest updates section on the main dashboard.


*Figure 6.2.6b: Notifications on Frontend Dashboard*

On top of the reliable delivery of messages, the utilization of AWS SQS also allows for the decoupling between the frontend and the SNS topics, increasing the overall fault tolerance of the system in the event one of these components fail.

### 6.2.7 Price Model

The data collected was used to train a Temporal Fusion Transformer (TFT), a type of Long Short-Term Memory Neural Network (LSTM), to enable predictions of sales numbers.

While the exact details are out of the scope of this report, a Neural Network was chosen over a simple regression as the input data is highly multi-dimensional, and therefore unsuitable for modelling, even with nonlinear regression. In previous attempts, non linear regression and simple feedforward Neural Networks led to deviations between 100-10, which was completely unacceptable as the sales volumes for some products was in the single digits for most days.

Even with basic LSTM models, predicted sales numbers were occasionally wrong, as in the case of big sales (e.g. when the day and month coincide), as the basic architecture is very expensive to train and run when needing to account for very long sequences of data (as are the sales every month). Therefore, a TFT, as described by Bryan et al. [1] was chosen to perform this analysis.

This TFT model was implemented with PyTorch, and took around 6 hours to train on a single Nvidia GTX 1060, and therefore does not require expensive computing resources. This is important for potential future iterations as it allows for potential continuous deployment at a low cost.

A sample output of the model at the current state is as shown below.

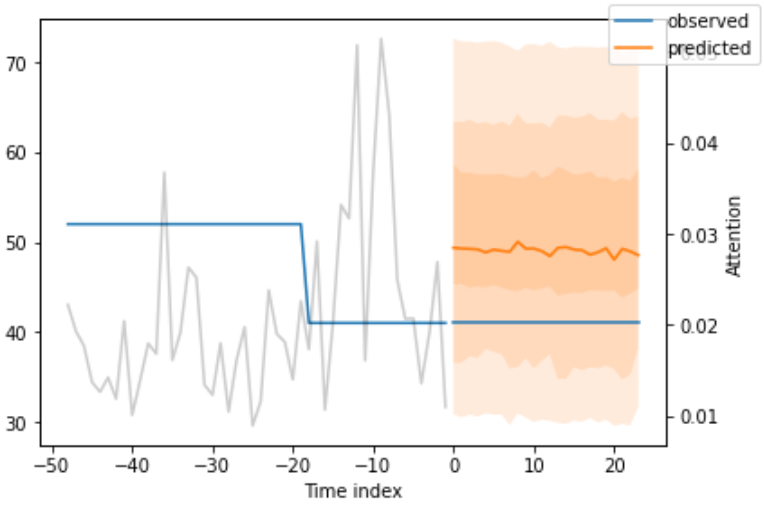



Figure 6.2.7a Output of Price model

While the graph is not easy to read, time constraints due to the different iterations of the model and training led to this MVP. Each time index corresponds to one hour, with the sales on the left axis. The grey plot is mostly used for describing the graph, and is irrelevant to the end user. Observations are inserted in the 1 day before and after index 0, or the user's selected date, and do not reflect actual data.

The TFT is then deployed on a Flask server for ease of integration, and is managed by Kubernetes.

[1] Bryan Lim, Sercan Ö. Arık, Nicolas Loeff, Tomas Pfister, *Temporal Fusion Transformers for interpretable multi-horizon time series forecasting*, International Journal of Forecasting, Volume 37, Issue 4, 2021, Pages 1748-1764,ISSN 0169-2070

# 7 Deployment and Monitoring

## 7.1 Installation and Deployment

The Amazon Web Services (AWS) cloud provider was used for the deployment of StalkFish. In order to properly utilize AWS services, AWS Identity and Access Management (IAM) roles had to be first configured. IAM roles allow for the custom configuration of specific permissions to respective AWS resources, these configurations could be found in the AWS IAM console.

### 7.1.1 Frontend

For the React frontend, there are two branches continuously deployed on AWS Amplify from the GitHub repository.

**Master** branch: The production environment which would be delivered to the client. Accessible at https://www.stalk.fish
**Staging** branch: The staging environment which we would use to preview the changes in a production-like environment. Accessible at https://staging.stalk.fish

Our workflow thus followed one where frontend changes are first merged into our **staging** branch for testing, and thereafter merged into **master** when ready for production. The image below illustrates the two deployed branches on the AWS Amplify console.


Build specifications are defined in AWS Amplify's build settings YML file "amplify.yml", where preBuild phases and test phases are configured. Apart from the installation of packages and dependencies, the preBuild phase includes the copying of environment variables, such as secret keys for third party API, which is stored in AWS Amplify console's environment variable settings. The test phase includes the installation of the MochaAwesome report-generator that is

used to automatically validate our frontend Cypress tests. More information on the testing phase is included in the integration testing section of this report at Section 8.2.

Moreover, it is worthwhile to note that the custom domain “stalk.fish” was purchased from the domain name registrar “namecheap”. The HTTPS certificate is automatically enabled by AWS Amplify, from the certificate issuer “Amazon Certificate Manager” (ACM). We have additionally enabled a redirect from https://stalk.fish to https://www.stalk.fish on the AWS Amplify console, for the purpose of disambiguation.

**Custom domain: stalk.fish**

| Domain | Status |
|---|---|
| stalk.fish | Available |

| URL | Branch | Redirects to |
|---|---|---|
| https://stalk.fish | master | https://www.stalk.fish |
| https://www.stalk.fish | master | - |
| https://staging.stalk.fish | staging | - |

This required some configuration on the domain name registrar platform, where DNS resource records are pointed towards the AWS Amplify domain management service. Furthermore, we have configured the DNS name to be used in other parts of the StalkFish application. The image below illustrates some of the configurations done for DNS resource record management, done on the domain name registrar.

| Domains | Status | Auto-Renew | Expiration ↑ |
|---|---|---|---|
| stalk.fish<br>Domain Privacy protection is ON | ACTIVE | | Oct 1, 2022 |

| Type | Host | Value | TTL |
|---|---|---|---|
| A Record | api | 18.138.102.38 | Automatic |
| A Record | k8s-cluster | 18.138.102.38 | Automatic |
| ALIAS Record | @ | d2z2wdgwcre1fd.cloudfront.net. | 5 min |
| CNAME Record | _4dcce26070ac9f... | _244f2bfbc96372be14eda7bcce0877a8.bnpptgxfyj.acm-validati... | Automatic |
| CNAME Record | staging | d2z2wdgwcre1fd.cloudfront.net. | Automatic |

### 7.1.2 Backend

Our backend services are built as Docker images using GitHub Actions and pushed to the GitHub Packages registry.

API Gateway, Data Aggregator and  are deployed on an AWS EC2 instance shown below.

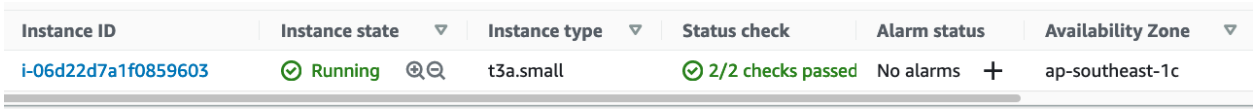



*Figure 7.1.2a. AWS EC2 Console t3a.small Instance Deployment.*

ssh via ec2-user@k8s-cluster.stalk.fish

```
       __|  __|_  )
       _|  (     /   Amazon Linux 2 AMI
      ___|\___|___|

https://aws.amazon.com/amazon-linux-2/
8 package(s) needed for security, out of 26 available
Run "sudo yum update" to apply all updates.
-bash: warning: setlocale: LC_CTYPE: cannot change locale (UTF-8): No such file
or directory
[ec2-user@ip-172-31-14-69 ~]$
```

*Figure 7.1.2b. Accessing EC2 instance*

```
NAME                                          READY   STATUS      RESTARTS   AGE
authentication-express-78fbc4585b-fchwk       1/1     Running     0          20d
commerce-postgresql-0                         1/1     Running     0          38d
authentication-db-0                           1/1     Running     0          38d
hasura-68f47d9bb-sf4ck                        1/1     Running     0          13d
scraper-manager-b5cf5c887-7l4mw               1/1     Running     0          12d
price-model-export-db-to-s3-27270240-nsz85    0/1     Completed   0          2d22
h
price-model-export-db-to-s3-27271680-gf47l    0/1     Completed   0          46h
price-model-export-db-to-s3-27273120-85nn5    0/1     Completed   0          22h
```

*Figure 7.1.2c. Pods*

```
[redacted] % kubectl port-forward svc/ingress-nginx-controller 3
000:80 --namespace ingress-nginx

Forwarding from 127.0.0.1:3000 -> 80
Forwarding from [::1]:3000 -> 80
Handling connection for 3000
Handling connection for 3000
```

*Figure 7.1.2d. Connecting to API gateway locally*

The Notification service and web scraper has components that are deployed on AWS Lambda.

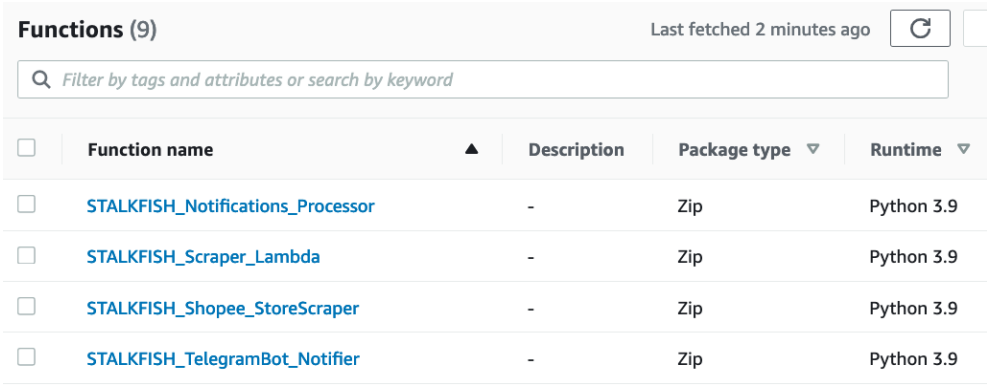



*Figure 7.1.2e AWS Lambda Console Deployed Lambda Functions.*

## 7.2 Monitoring

To obtain visibility into the state of StalkFish, we rely on both our infrastructure-level and service-level logs. The infrastructure-level logs inform us about the health of the Kubernetes cluster, providing information such as CPU and memory usage, as well as any pod allocation issues. Our service-level logs allow us to monitor performance and debug issues, especially ones that span across multiple services.


*Figure 7.2a: Average Execution Time for Lambda Functions*

To simplify the logging integration, our services write their logs and events to the standard output. Each service's standard output was then forwarded to AWS CloudWatch at the node-level by a Fluent Bit daemon. This design allows services to be abstracted from the details

of routing and storing logs, shifting the complexity to the infrastructure instead of placing it on the individual service.

With the logs from all services stored in CloudWatch, we can perform queries to get an overview of what a service has been doing. For instance, Figure 7.2b below shows a query to ensure that the Scraper Manager service has been communicating with the Notifications service on time.

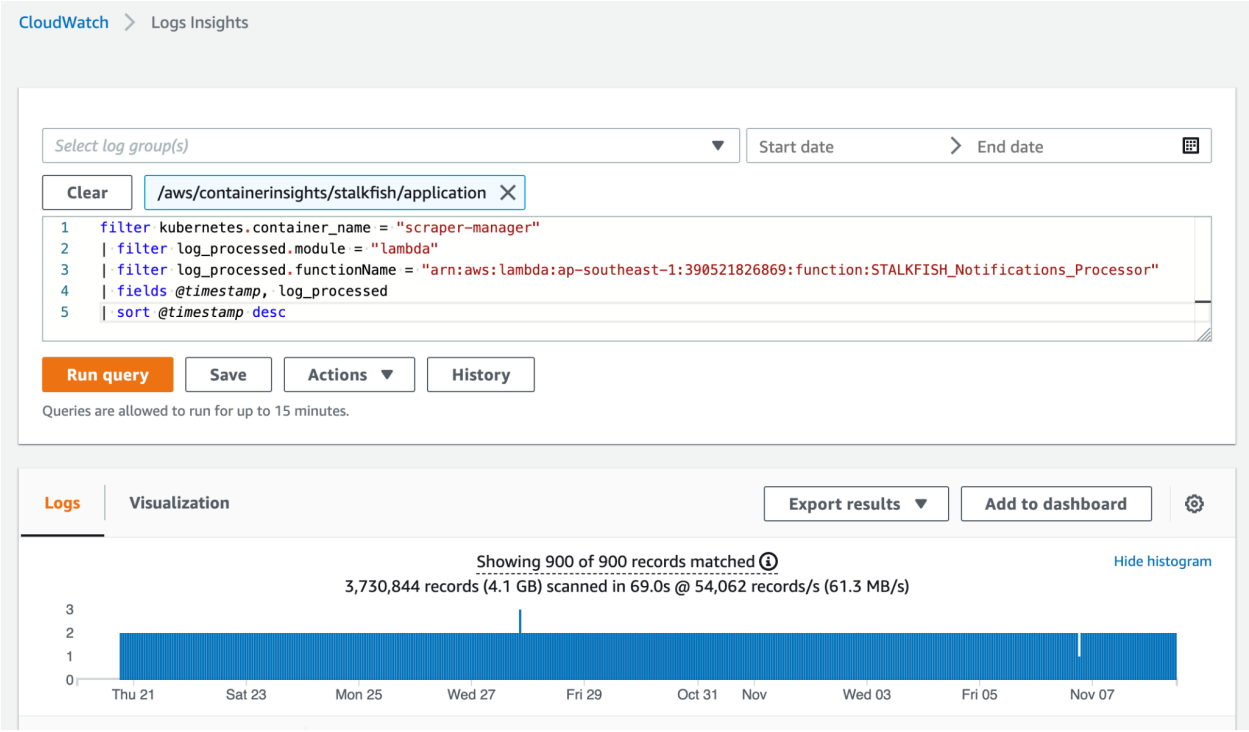



*Figure 7.2b: Example of Cross-Service Logs Discovery*

# 8 Testing

## 8.1 Unit Testing

For the frontend, testing was performed using Cypress, a Javascript framework for end-to-end testing. Tests were written for the login/logout flow, page navigation, as well as the CRUD operations for stores and products. This allowed us to quickly perform regression testing whenever we developed new features on the frontend.

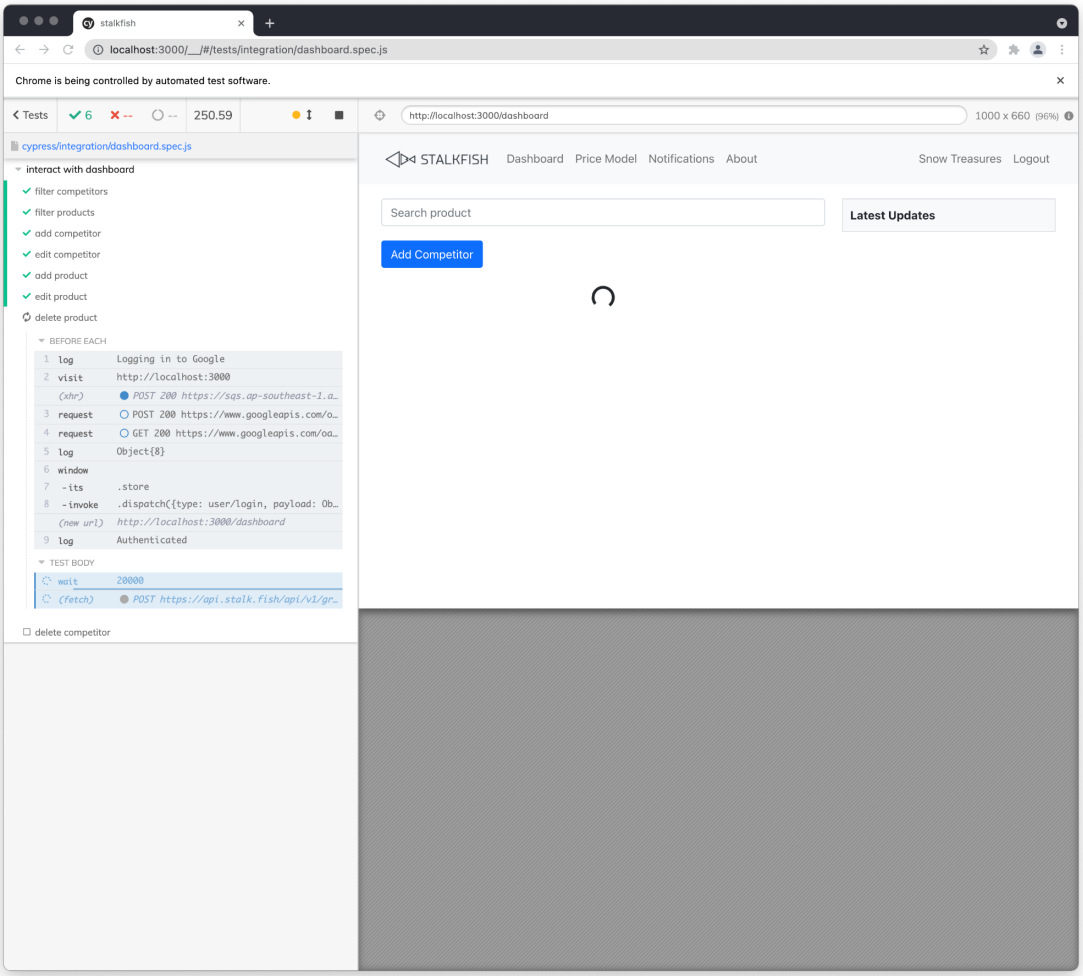



*Figure 8.1a: Automated Testing For Frontend Using Cypress*

Unit testing for the web scraper and notification service lambdas were done using the Python unittest framework. Since these were implemented as AWS Lambda functions, there was no requirement to mock the AWS Lambda environment, and instead the handler could be called from unittest similar to a regular Python method.

## 8.2 Integration Testing

AWS Amplify provides the ability for Cypress integration tests to be written and be used for continuous integration. This is configured in the amplify.yml file on the Amplify console, where the MochaAwesome report-generator is used for verification of the tests. This was exploited for the CI workflow, ensuring that the frontend was working as intended  before deploying any changes. The image below illustrates an example of some test cases that are automatically run on the AWS Amplify console.

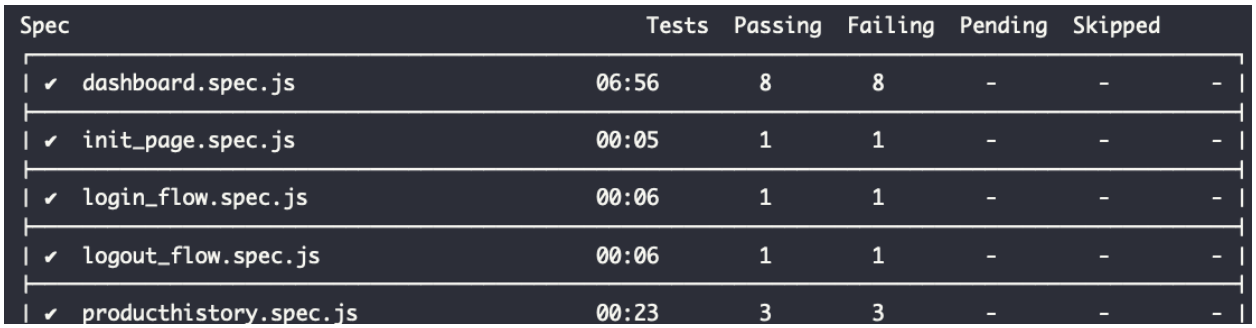



*Figure 8.2a: AWS Amplify Testing Console Cypress Tests.*

Moreover, AWS Amplify provides the ability to download test artifacts for both tests that are passing and failing. Artifacts come in the form of a video or a screenshot, depending on the test case, providing greater clarity for any failing tests.


*Figure 8.2b: AWS Amplify Console Cypress Artifact Download Page.*

Integration tests between other components are carried manually. The Notifications Service was tested using the AWS Lambda ‘Test’ feature illustrated below, to trigger the test instead of the Scraper Queue Manager.

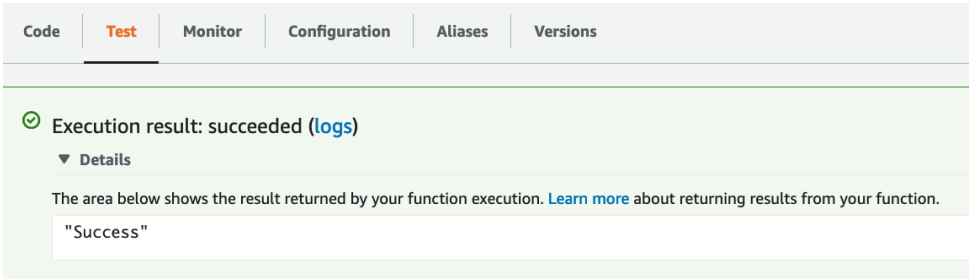



*Figure 8.2c: AWS Lambda Console Test Feature for StalkFish Notifications Processor Lambda.*

The notifications processor was tested on how it integrates with the other components of the notifications service. For instance, if a registered notification was able to be retrieved, processed and sent over to the AWS SNS topic. Moreover, it was manually tested if the StalkFishBot was able to correctly output the notification published to the SNS topic. There are several other internal AWS tools built into the services that help in the process of integration testing. To name a few, messages could be manually published on AWS SNS, while SQS provides the ability to manually poll for messages.

## 8.3 System Testing

After conducting unit and integration tests, system testing was done manually. This is to validate all the integrated components of StalkFish, such that we are able to do an end-to-end evaluation of the system. The diagram below illustrates the steps we have taken to carry out system testing, while the table after summarizes the details of each testing step.

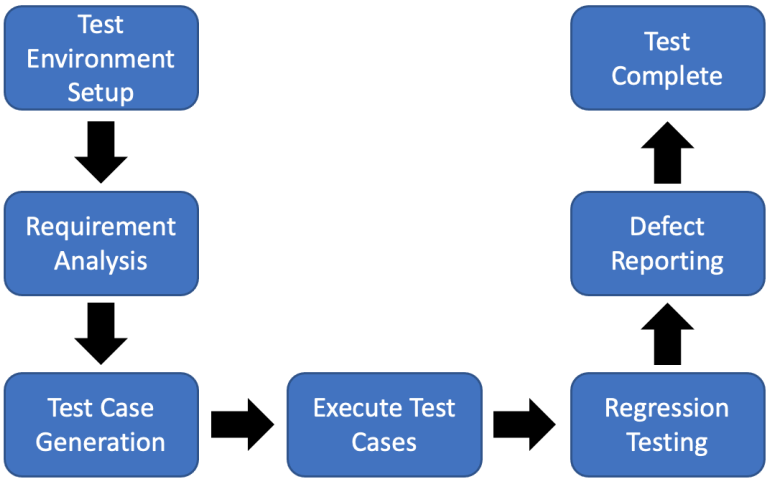



*Figure 8.3a: System Testing Process*

| Testing Steps | Process Details |
|---|---|
| 1. Test Environment Setup | New features for the front end are pushed to the staging branch, while backend features are directly deployed onto the k8s cluster. System testing is thus carried out on https://staging.stalk.fish by the software development team. |
| 2. Requirement Analysis | New features added are analysed to figure out what they offer and what should be tested. |
| 3. Test Case Generation | Test cases and scenarios are engineered such that they are a form of black-box testing. |
| 4. Execute Test Cases | Software development team carries out the system testing on the staging site. |

| | |
|---|---|
| 5. Regression Testing | Regression tests are run. |
| 6. Defect Reporting | Defects are recorded, and GitHub issues are opened to report bugs. Moreover, respective component I/Cs are also informed over discord on the bug present. |

## 8.4 Acceptance Testing

Finally, we performed user acceptance testing upon completion of unit, integration and system testing. This was done prior to moving StalkFish to its production environment. In order to do so, our testing lead has liaised with the client to allocate some resources and staff members to participate in the testing process. Acceptance testing was also carried out on https://staging.stalk.fish

The table below is an overview of the acceptance testing process that was carried out.

| Testing Steps | Process Details |
|---|---|
| 1. Creation of Test Plan | Testing lead outlines a test plan and schedules meetings with the client. |
| 2. Analysis of Business Requirements | Meeting with the client's e-commerce manager to go through the Software Requirements Specification SRS document, to identify business requirements. |
| 3. Creation of Test Cases and Scenarios | Meeting with the client e-commerce manager to formulate test cases and scenarios with respect to high-level business processes. Includes the preparation of test data, such as the list of competitors. |
| 4. Run the Test Cases | Test case instructions are then handed over to the two staff members who were designated to participate in the testing process. |
| 5. Record Results | Test participants then record down the results of the test, along with any other additional feedback and suggestions which would work towards improving StalkFish. |
| 6. Test Review | Meeting with the client's e-commerce manager and test participants, to review the test results, defects log and confirm if business requirements have been met. |

Feedback from these acceptable testing runs were discussed at our weekly standups, as per the agile workflow, to potentially flesh out further features/changes, and delegate them to the respective team member.

# 9 Documentation and Coding Standards

## 9.1 Frontend

### 9.1.1 File Structure and Organisation

The team followed a standard structure for organising our frontend project - this ensured that features and components are easy to find and manage. Our approach was to group files by their feature or route - one of the recommended ways in the official [React documentation].

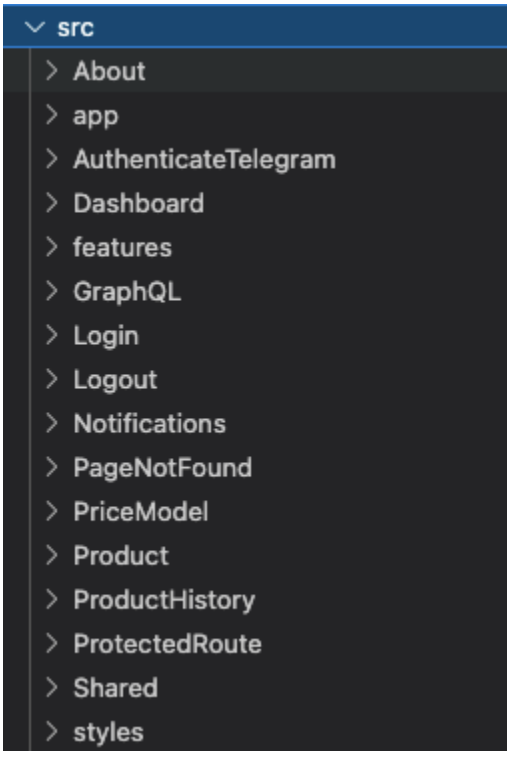



*Figure 9.1.1a: File structure for frontend application*

As part of the suggested requirements was that the project was on a monorepo, individual components were organised into their own folders for ease of organisation.


*Figure 9.1.1b Structure of monorepo*

Moreover, while many languages were used in the development of StalkFish, the following standards were followed for each language:

| Language | Standard |
|---|---|
| Python | PEP 8 |
| Javascript | Google Javascript Style |

### 9.1.2 Decomposing Components

In the React frontend, the team decomposed larger components into smaller components for easy reuse and maintainability.

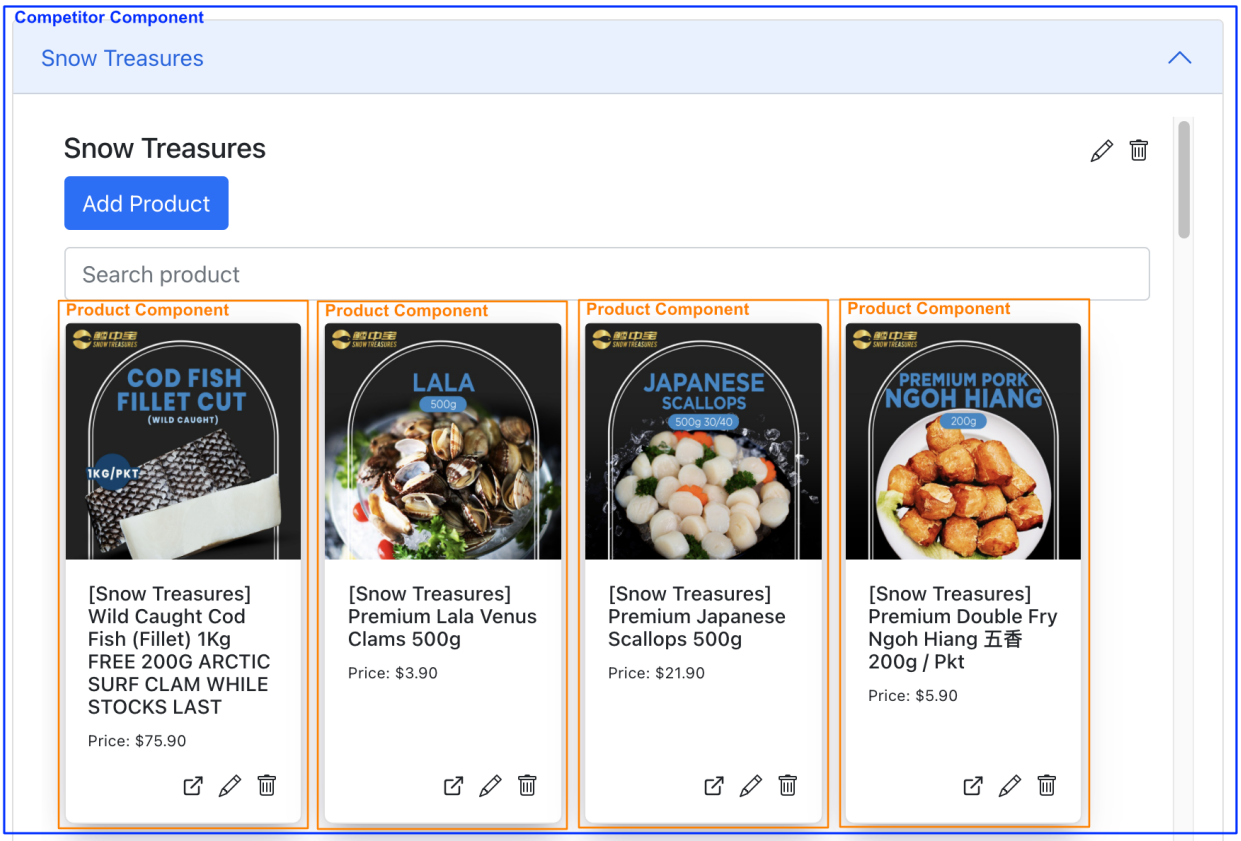



*Figure 9.1.2a: Dashboard components*

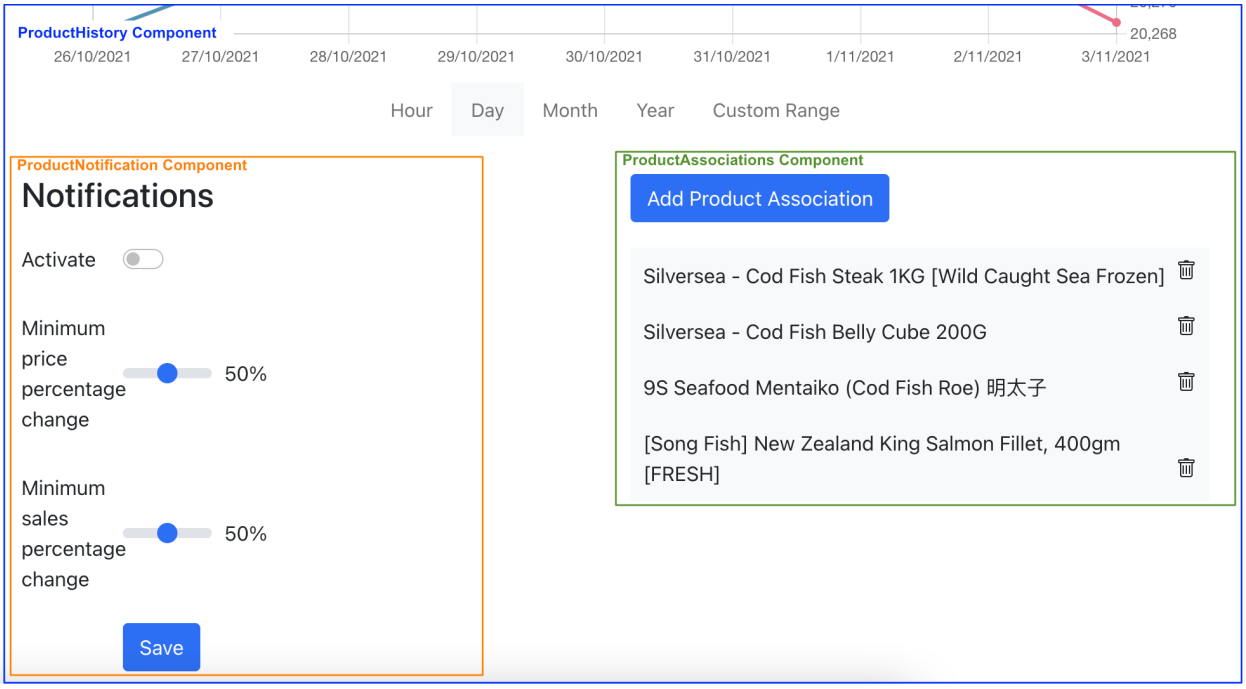



*Figure 9.1.2b: ProductHistory components*

As seen in Figure 9.1.2a, the Product component is reused multiple times. In the ProductHistory component in Figure 9.1.2b, the notification and product associations sections are split into their individual components to greatly reduce the amount of code in a single component.

### 9.1.3 Redux

The frontend uses Redux, a popular state management tool for React. Redux is a pattern and library that helps with state management using actions (or events).

In StalkFish, a user store is created to store the authenticated user's information - containing the user's name, email address, and JSON Web Token. When a user logs in, the login reducer is called and passes the authenticated user's credentials to the login reducer which then updates the state with the user's credentials, signifying that the user is logged in.

Several components that require access to the user's information (such as the Navbar and ProtectedRoutes) will then subscribe to the user store and can access the always up-to-date user information - the concept is similar to the Observer and pub-sub patterns.

There are several alternatives to using Redux such as passing the login state through props - however, when working with multiple components, this leads to prop-drilling, when data is required to be passed through multiple components even when they may not require direct access to it. React Context was also considered, however, Redux offers several useful features out of the box such as 'persist state' which stores the state locally using local storage and populates it on load. Choosing React greatly reduced the amount of time needed to get started.

# 10 Budgeting Considerations

Instead of using Amazon Elastic Kubernetes Services (EKS) to set up and manage a Kubernetes cluster, we decided to deploy K3s instead. This was due to the relatively high cost of EKS which only provided a managed Kubernetes control plane and no worker nodes to run our application. By running K3s, we could efficiently host a fully certified Kubernetes cluster with fewer resources required.

To properly account for the needs of StalkFish, we experimented with a few different EC2 instance types with varying CPU performance and memory available. We found that the t3a.medium instance type provided the best value for our application. In future, further savings could be achieved by using a mix of Reserved Instances for long-running stateful services and Spot Instances for stateless services, or choosing ARM-based instances. These options were not explored at this juncture as we were prioritizing the application's development.

Event based (Notifications Service, Scraper) components were utilised, AWS Lambda was used to reduce the reliance on server instances, and therefore costs. This enabled us to easily implement components that would otherwise only be used briefly each day, without needing to reserve a whole instance for that component, or potentially sacrificing performance on the EC2/Hasura instance.

# 11 Suggestions for Improvements and Enhancements

Upon checking in with the client, one limitation of StalkFish is that the “sold/historical_sold” field taken from the Shopee API does not exactly tally with the client’s actual sold value on the Shopee seller’s console. This implies that the same attribute that appears on competitor’s StalkFish item history page would be equivalently inaccurate. Hence, this “sold/historical_sold” could merely be used as a coarse estimate and not a ‘real-time metric’ for them to update their prices. Moreover, this “sold/historical_sold” attribute when taken into account for price modelling may cause the price model to be potentially less meaningful. We speculate that this “sold/historical_sold” attribute represents a product that a Shopee customer has purchased and already received, only after they indicated that they have received the product. In order to obtain a more meaningful price model, the client should clarify with the respective e-commerce platforms what the exact meaning of each metric is.

Furthermore, the price modelling component does not actually model price changes at the moment. While this is a useful feature for the client, without more data and more time, it is simply unable to produce meaningful results. Perhaps a future iteration could more meaningfully provide this data. Currently, the graph produced by the model is also quite hard to interpret as it is meant more to inform training. However, due to time constraints, this was the best representation possible. A future iteration could also properly extract the data and plot it natively inside the front end, rather than as a static image. Another option is to use AWS Sagemaker to train and deploy the model.

Finally, the model is currently static and is only trained on a few weeks worth of data. Since the training is relatively inexpensive, perhaps a future iteration could implement continuous training and deployment of data.

Ultimately, some of the above requirements were not selected for development in the final product, detailed below is some of the rationale for doing so:

| Category | Functional Requirement | Rationale |
|---|---|---|
| Scraping | **[FR1.5]** The application should support scraping from Qoo10. | While Qoo10 is the client’s second largest market, the server side rendering of Qoo10’s product pages was deemed too time consuming to attempt to scrape. However, provisions (as described in Section 6.2.4 Data Aggregation and Retrieval) have been made to the components to allow for ease of extension in a future iteration. |
| | **[FR1.6]** The application should support scraping from Lazada. | Lazada is too small a market for the client for it to be a priority. However, as with Qoo10, provisions have been made for ease of extension. |

| | | |
|---|---|---|
| User Authentication | **[FR3.2]** The master user should be able to add accounts | The client has said that this is not a high priority due to the fact that they are mainly using a single account for logging in. However, this is definitely a feature to be implemented in future iterations, when the application is more widely used and when additional features are implemented and requires role-based authentication. |
| Price Model | The application should be able to predict price trends based on historical data. | While such a feature would be useful, it was found that regression was not able to produce meaningful results, as there is not enough data to reliably draw a correlation. |

# 12 Reflections and Learning Points

Throughout the course of this project, we have learnt the following points:

## Identifying the right tools and technologies

Identifying the right tools and technologies is an important part of the SDLC. In the case of StalkFish, the use of Hasura for our commerce DB has helped us:

### Hasura GraphQL Engine

As StalkFish uses the microservices architecture, the Hasura GraphQL Engine was used to help stitch several separate databases into one database endpoint.

Hasura easily turns our PostgreSQL database into a GraphQL endpoint. In addition, Hasura also has a fully featured API console which helps the team visualise and work with the API endpoint quickly.

Benefits of having a GraphQL endpoint:

- Reduce the need to build backend API endpoints for different resources
- More flexible to future changes, when additional database fields are added, there is no need to alter our API endpoints
- Highly customisable for our frontend needs: the flexibility of the query parameters enabled faster development of our frontend, especially when configuring our graphs

### AWS Lambda Functions

Further, the extensive use of AWS Lambda has also helped us enable:

- Ease of scalability
    - As more calls are abstracted to the AWS Lambda, the computation for the components are able to be done asynchronously, and without concern for autoscaling
- Ease of development
    - These Lambdas are relatively easy to implement, as opposed to a stack where an API is required to explicitly be exposed
    - Lambdas are also useful in the case of a microservices architecture were we need to access different services not on the same server
- Reduce number of services required
    - With Lambda functions, we can avoid having to spin up additional services e.g. a Node server to run certain services. For example, the need for a Notification server to continuously poll for new messages to deal with Telegram authentication requests (when a user sends the /start message to the bot)

## Devops

Ultimately, we got a glimpse of the benefits of adopting the DevOps approach. By utilizing continuous integration tools, we managed to automate some processes of testing code which reduced some amount of manual work. As a result, we could focus our attention on implementing features which is a process that is unable to be automated.

Moreover, through the combination of development and operations, we had a client-centric approach with shortened iterative feedback loops. This enabled us to focus on continuously improving StalkFish based on our client's constant feedback, eventually creating a final product that fulfils their business requirements.

# 13 Appendix

## 13.1 Frontend Mock-Up

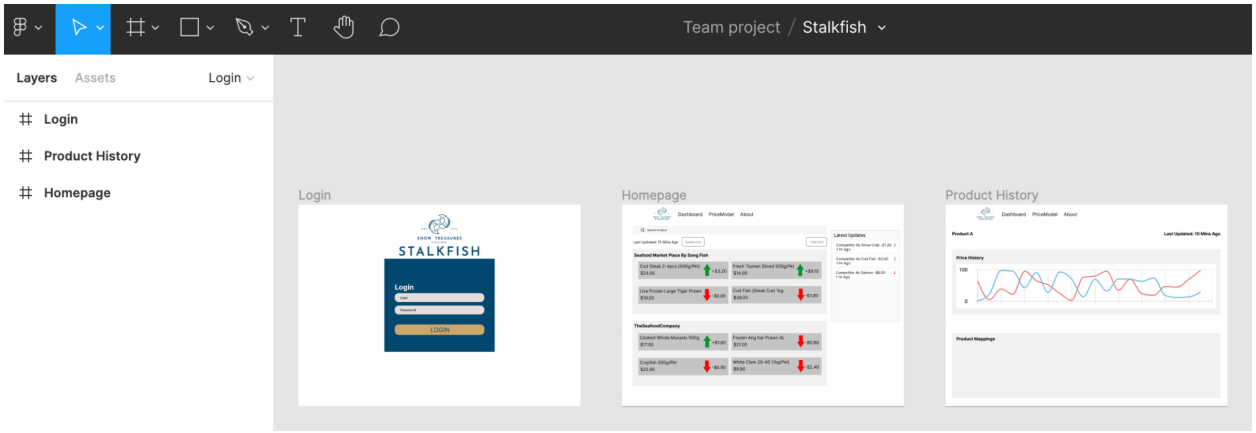



*Figure 13.1a: Mock-up on Figma*

Before the development of the frontend, we designed the frontend using Figma. Figma allowed us to collaborate easily and make changes concurrently.

## 13.2 Microservices API

As mentioned in Section 12, Hasura GraphQL Engine provided us with a convenient GraphQL endpoint that is flexible and customisable. Figure 13.2a below shows the Hasura console, where queries and mutations can be easily constructed by clicking the checkboxes on the left. This way, we did not have to create separate endpoints for different queries and mutations.

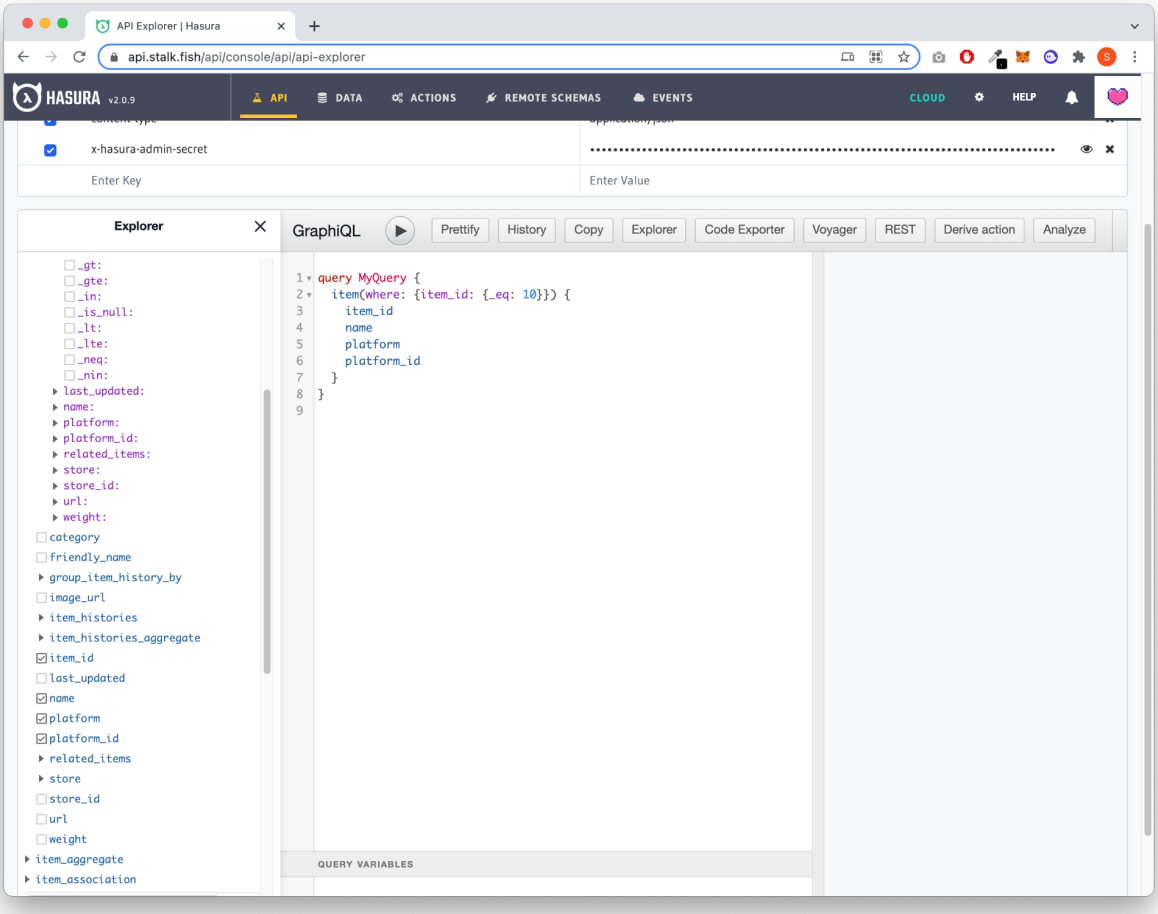



*Figure 13.2a: Hasura Console*

## 13.3 Use Cases

For all the following use cases, the system is StalkFish.

| **UC 1** | User login using Google account |
|---|---|
| Actor | User |
| MSS | 1. User navigates to login page<br>2. User attempts to login<br>3. User is directed to Google authentication site<br>4. User enters username and password<br>5. User is redirected back to StalkFish<br>6. StalkFish logs user in using a generated JWT given by Google OAuth 2.0 |

| | |
|---|---|
| | Use case ends |
| Extensions | 1a. User is already logged in<br>1a1. StalkFish redirects user to dashboard page<br><br>4a. User enters invalid username and password<br>4a1. StalkFish displays invalid access error message<br>Use case resumes at step 4. |

| **UC 2** | View stores and products |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands the accordions of the different stores<br>3. StalkFish displays all the products under a store<br>Use case ends |

| **UC 3** | Filter stores using product name |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User enters product name into search bar<br>3. StalkFish filters stores using product name<br>Use case ends. |

| **UC 4** | Filter products under a store using product name |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands the accordions of a particular store |

| | |
|---|---|
| | 3. User enters product name into search bar under the expanded accordion<br>4. StalkFish displays all products with specified name under the particular store<br>Use case ends |

| **UC 5** | Add store |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User clicks on “Add Competitor” button<br>3. User enters store name and store’s Shopee URL<br>4. StalkFish creates new store accordion at the bottom of the page<br>Use case ends |

| **UC 6** | Edit store |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to edit<br>3. User clicks on pencil icon to edit store<br>4. User enters new store name and Shopee URL<br>5. StalkFish updates store’s name and details<br>Use case ends |

| **UC 7** | Delete store |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to edit<br>3. User clicks on trash can icon to delete store |

| | |
|---|---|
| | 4. User confirms deletion of store<br>4. StalkFish removes store from list of stores<br>Use case ends |

| **UC 8** | View store details |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to view<br>3. User clicks on store name<br>4. StalkFish redirects user to store details page<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when adding/editing store<br>4a1. StalkFish displays page not found<br>Use case ends |

| **UC 9** | Visit store on Shopee |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes view<br>3. User clicks on external link icon beside pencil icon<br>4. StalkFish opens store Shopee url in a new tab<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when adding/editing store<br>4a1. StalkFish displays page not found<br>Use case ends |

| **UC 10** | Add product |
|---|---|

| | |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to add product for<br>3. User clicks on “Add Product” button<br>4. User enters product name, product’s Shopee URL, weight and category<br>5. StalkFish creates new product at the bottom of the store accordion<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when adding product<br>4a1. StalkFish displays that product URL is invalid<br>Use case ends |

| | |
|---|---|
| **UC 11** | Edit product |
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to edit product for<br>3. User clicks on pencil icon<br>4. User enters new product name, product’s Shopee URL, weight and category<br>5. StalkFish updates product<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when editing product<br>4a1. StalkFish displays that product URL is invalid<br>Use case ends |

| | |
|---|---|
| **UC 12** | Delete product |
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar |

| | |
|---|---|
| | 2. User expands store accordion for store he wishes to edit product for<br>3. User clicks on trash can icon<br>4. StalkFish deletes product<br>Use case ends |

| **UC 13** | View product details/history |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to view product<br>3. User clicks on product name/image<br>4. StalkFish redirects user to product details/history page<br>5. StalkFish displays product details and price/sales history<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when adding/editing product<br>4a1. Product name is not clickable<br>Use case ends |

| **UC 14** | Visit product on Shopee |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to view product<br>3. User clicks on external link icon beside the pencil icon<br>4. StalkFish opens product Shopee url in a new tab<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when adding/editing product<br>4a1. StalkFish displays page not found<br>Use case ends |

| **UC 15** | Associate product with other products |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to view product<br>3. User clicks on product name/image<br>4. StalkFish redirects user to product details/history page<br>5. User clicks on “Add Product Association Button”<br>6. User selects other product to add<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when adding/editing product<br>4a1. Product name is not clickable<br>Use case ends |

| **UC 16** | Compare product with associated products |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to view product<br>3. User clicks on product name/image<br>4. StalkFish redirects user to product details/history page<br>5. User clicks on price comparison/sales comparison tabs above the graph<br>6. StalkFish displays price/sales data for product and its associated products in the graph<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when adding/editing product<br>4a1. Product name is not clickable<br>Use case ends |

| **UC 17** | Delete product association |
|---|---|

| | |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to view product<br>3. User clicks on product name/image<br>4. StalkFish redirects user to product details/history page<br>5. User clicks on trash can icon beside associated product<br>6. StalkFish deletes product association<br>Use case ends |
| Extensions | 4a. User entered invalid Shopee URL when adding/editing product<br>4a1. Product name is not clickable<br>Use case ends |

| | |
|---|---|
| **UC 18** | Register notifications for a product |
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to notifications page using navigation bar<br>2. User clicks on “Open StalkFish in Telegram” button<br>3. User clicks on verification link in Telegram and is redirected back to StalkFish<br>4. User clicks “Confirm” button on StalkFish<br>4. StalkFish notifies user that verification is successful<br>5. User navigates to dashboard page using navigation bar<br>6. User expands store accordion for store with product he wishes to add notifications for<br>7. User clicks on product name/image of product he wishes to add notifications for<br>8. StalkFish redirects user to product details/history page<br>9. User toggles “Activate”<br>10. User sets threshold for price and sales change to be notified<br>11. User clicks “Save” button |

| | |
|---|---|
| | 12. User will receive notifications on Telegram when threshold is hit on next scrape<br>Use case ends |
| Extensions | 1a. User already registered with Telegram<br>1a1. StalkFish shows that Telegram is connect<br>Use case resumes at step 5. |

| **UC 19** | Edit notifications threshold for a product |
|---|---|
| Actor | User |
| Precondition | User is logged in |
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store with product he wishes to edit notifications for<br>3. User clicks on product name/image of product he wishes to edit notifications for<br>4. StalkFish redirects user to product details/history page<br>5. User sets new threshold for price and sales change to be notified<br>7. User clicks “Save” button<br>8. StalkFish updates threshold for sending notifications<br>Use case ends |
| Extensions | 5a. User has not set up notifications<br>5a1. StalkFish prompts user to set up notifications in Telegram<br>5a2. User proceeds to set up notifications in Telegram (refer to **UC 18** for instructions)<br>Use case resumes at step 1. |

| **UC 20** | Remove notifications for a product |
|---|---|
| Actor | User |
| Precondition | User is logged in |

| | |
|---|---|
| MSS | 1. User navigates to dashboard page using navigation bar<br>2. User expands store accordion for store he wishes to view product<br>3. User clicks on product name/image<br>4. StalkFish redirects user to product details/history page<br>5. User toggles “Activate”<br>6. User clicks on “Save” button<br>7. StalkFish no longer sends notifications for the product<br>Use case ends |