1. Introduction

1.1. Overview of Information System

Soup Street is an ecommerce business designed to offer a variety of soups that can be purchased at a click of a button. These soups are then delivered to each individual ordered.

The Soup Street information system is a user-centric e-commerce platform developed on the NextJS framework, incorporating MongoDB for flexible and scalable database management. With a focus on secure user authentication using Next-Auth and Google login, the system provides a good user experience for both administrators and users.

The admin panel allows efficient product, category, order, and user management, while the responsive design ensures accessibility across various devices. The product catalogue, shopping cart, and secure payment gateway contribute to a user-friendly shopping experience.

Extensive unit testing and user feedback have validated the functionality and usability of the system, highlighting positive responses to navigation, security, and overall design. The LocalShop information system stands as a well-designed and efficient solution for successful e-commerce operations.

1.2. Technologies Used

NextJS

The application is developed using the NextJS framework. NextJS is an open source web development framework used to create react-based web applications with server-side rendering and static website generation. Using a framework such as NextJS allows for fast and secure web app development.

Tailwind CSS

Tailwind css is an open source css framework, allowing for quick css styling using predefined classes for changing attributes such as height and width.

For the admin panel tailwind css was used to make styling individual components quick and efficient.

Styled-Components

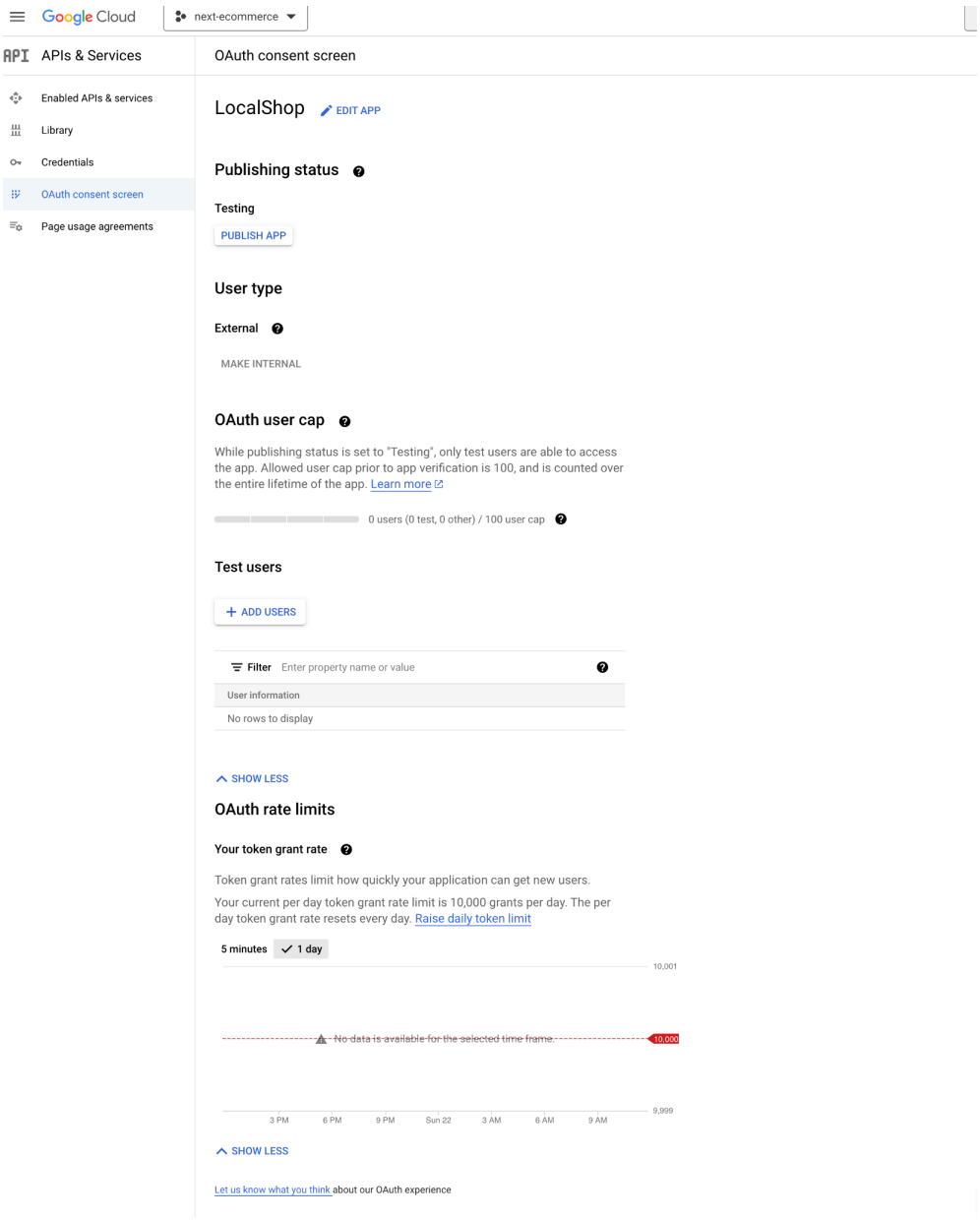
Styled-Components allow you to write actual CSS code within your JavaScript or JSX files, and they create unique class names to scope the style of a specific component. Styled-Components is used to style all shop front pages, the components directory contains all the styled components created for front end use.

MongoDB

The Database for the application is mongoDB. MongoDB is a noSQL, document oriented database program. MongoDB is easy to use and widely supported across a variety of frameworks.

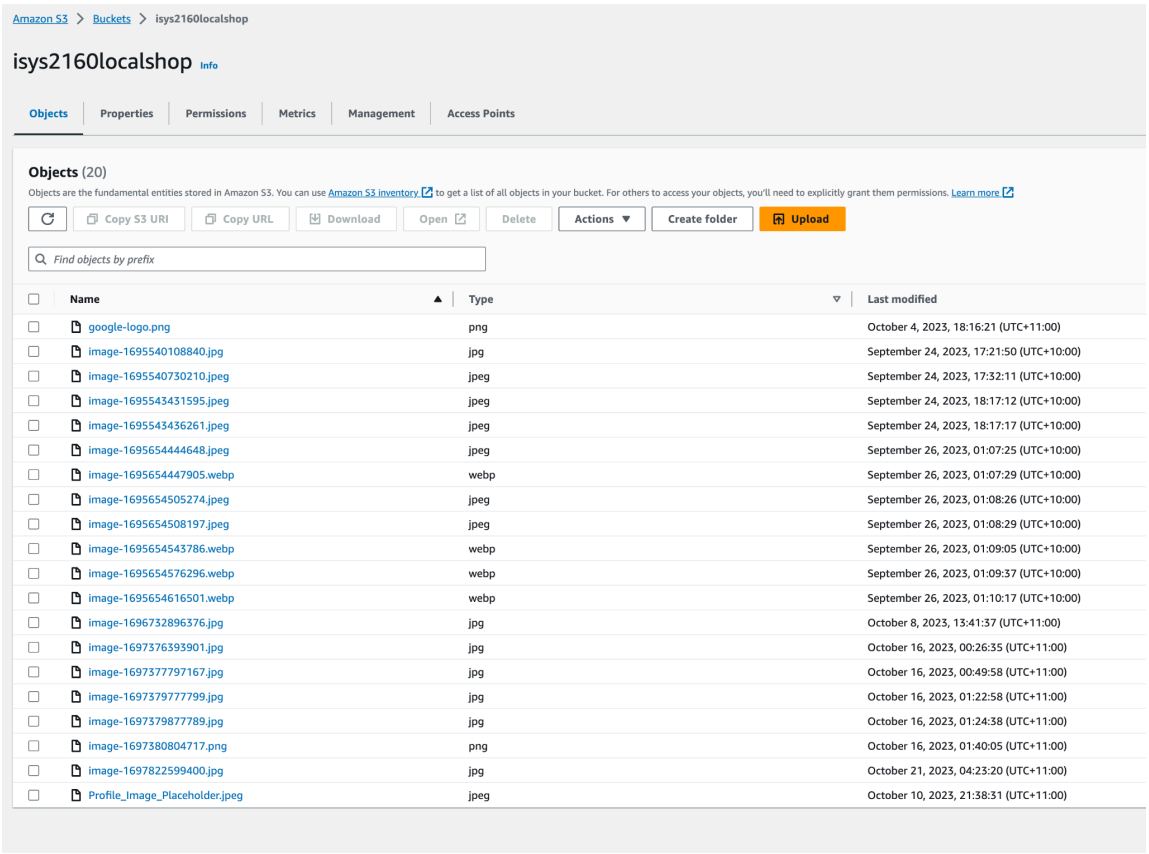
Google Developer Console

To allow third party sign in using google via Next Auth, a google developer account is required to authorise Oauth2 transactions.



AWS' S3

S3 is an object storage service that allows developers to store a variety of objects-mostly images. In this project S3 was used to store images, which are then set to public viewing. By storing the unique image link, this allows for fast and efficient displaying of images.

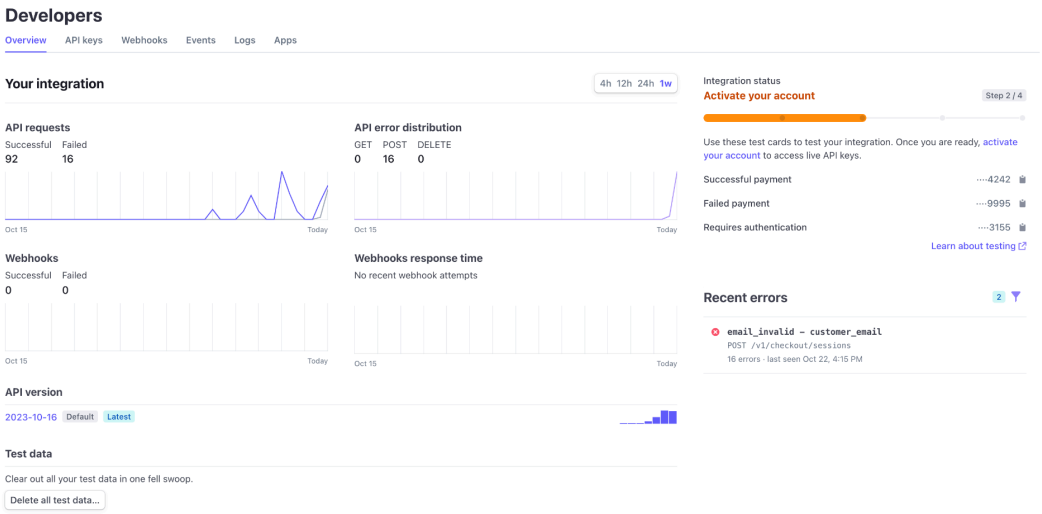


Next-Auth

NextAuth is an authentication service for NextJS that allows for easy integration of third party authentication such as with google. NextJS seamlessly connects to mongoDB using the mongoose library, allowing for secure and simple user account management.

Stripe

The Stripe javascript library was installed to use the stripe platform to successfully complete payment through a secure payment gateway. A stripe business account was created for soup street, where as developers we used the test payment feature to correctly test the payment success or failure based on the functional requirements.



2. User Manual

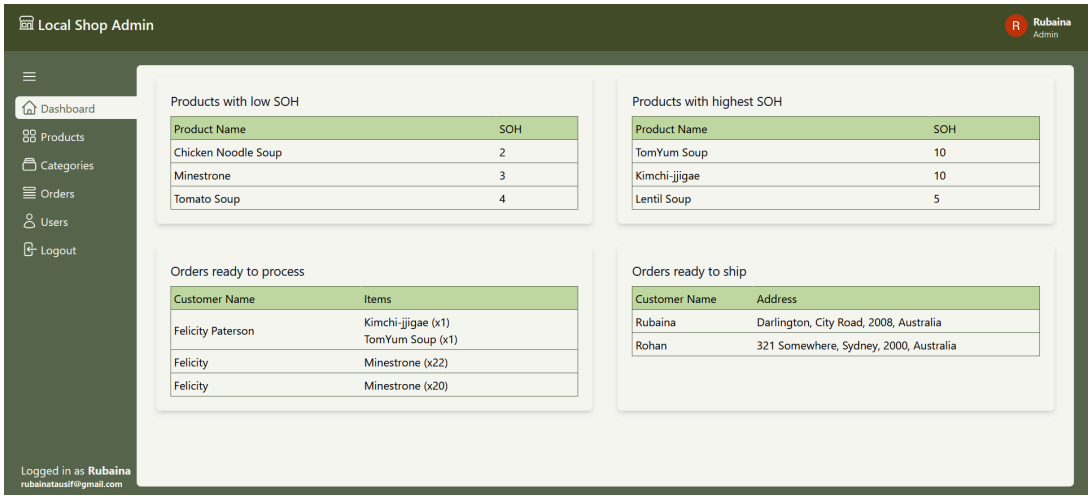
To use the app first navigate to the correct directories, the admin website is launched using the `localShop` directory whereas the shop front is launched through the `LocalShopFront` directory. Open the terminal in the localShop for. Enter the command `npm install` or `yarn install`, this will download the necessary libraries needed to run the application. After the installation is complete enter the command `npm run dev` or `yarn dev`. Then, whilst the backend is operating, navigate to the front-end folder, `LocalShopFront`, and follow the same process. This should start the backend on localHost 3000 and the frontend on localHost 3001. If this is not correct, try again as the login authorisation will be rejected if this is not reflected.

2.1. Admin

Please note that all path names are relative to the localShop folder.

To add your account as an admin, access the `nextAuth` file under `pages>api>auth>...[nextAuth].js` and add your google account email to the admin emails list. All authorised admin emails are allowed to log in to the backend using google authentication. If you are redirected, it means your email is not in this list.

Once you have access to the backend, as an admin you have a selection of features available to you. Firstly, you will see the dashboard page upon logging in. The dashboard currently only has 2 information displays on it, allowing for you to see the products with highest and lowest stock on hand (SOH) as well as orders awaiting your action.



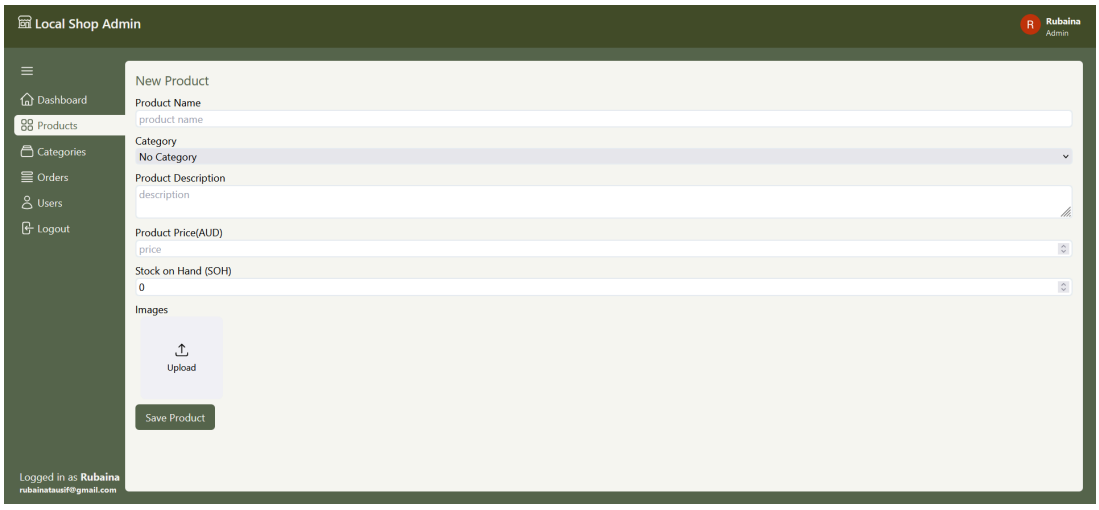
Using the navbar on the left of the screen, you can expand by clicking on the menu icon (☰), you can access pages for, products, categories, orders and user management.

Products

When opening the product page you are presented with a list of products currently in the system, their SOH, the ability to edit and delete each product as well as the ability to add a new product.

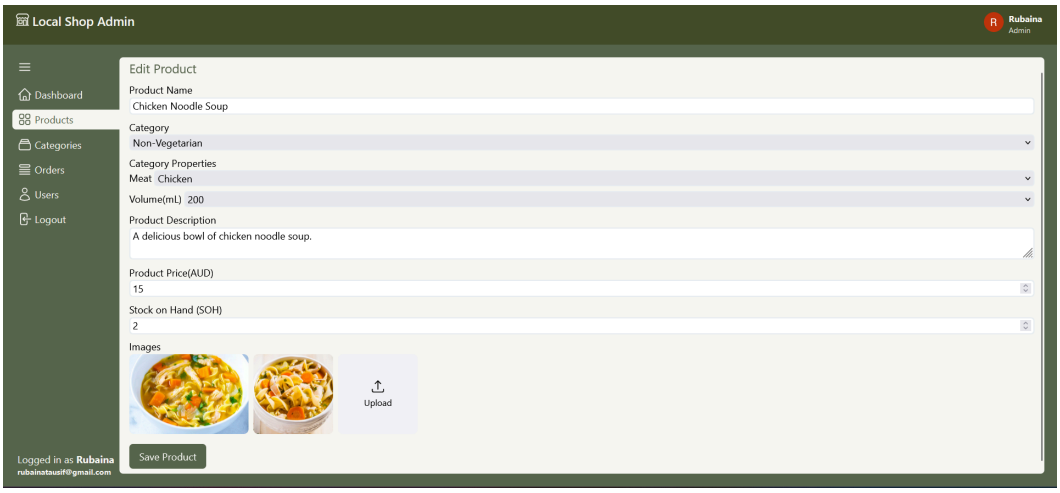
Add New Product:

To add a new product, first navigate down and click the “add new product” button at the bottom of the page. Clicking on this will take you to a form where you can enter all appropriate information such as Name, Description, SOH, Category as well as upload any photos. Once you are done, click “save” to save the product to the database, this will take you back to the products page.



Edit a Product:

To edit a product, first navigate to the product in the list you want to edit and select the edit button in the appropriate row. This will take you to the same form that appeared when creating the product, prefilled with the correct information for the product. You can edit this however you wish and press “save” to save it to the database. To cancel any changes, simply click on any icon in the navbar and it will navigate away without saving.

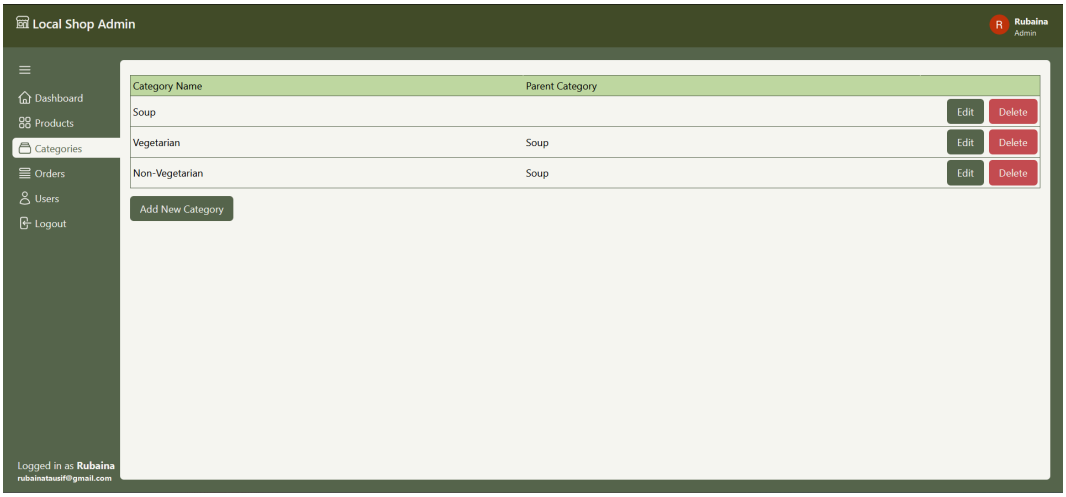


Delete a product:

To delete a product, first navigate to the product in the list you want to delete and select the delete button in the appropriate row. This will take you to a separate page to confirm whether you want to delete or not. Click yes to delete and no to cancel. Both options navigate you back to the products page.

Categories

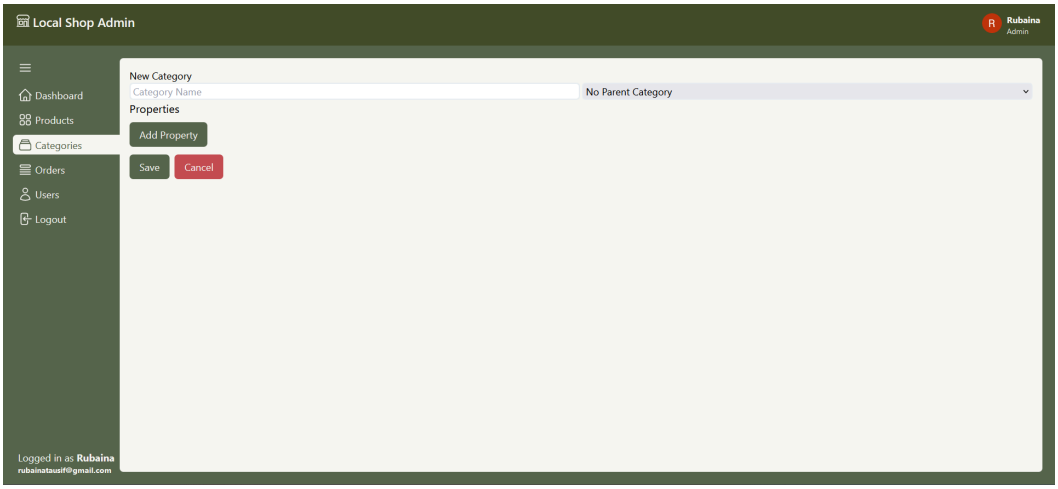
When opening the categories page, you are presented with a table containing every category, its parent category and buttons to edit/delete said category as well as a button to add a new category.

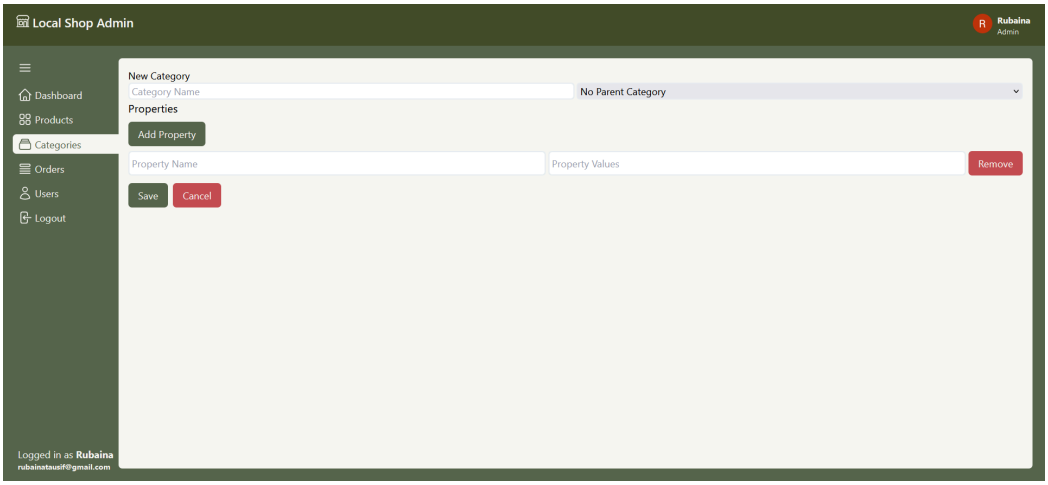


Add new Category:

To add a new category, first navigate down to the bottom of the page and click the "add new category" button. This will take you to a form where you can give the category a name, a parent category and any relevant properties. When adding properties, add the name on the left and the potential values on the right, comma-separated. Eg. "Count per box" would go on the left and "1,2,3,4,5" Would go on the right. When adding categories to products, the product also gets the properties of any category that is above the specified category in the parent chain. For example, if you had 3 categories with the relationship grandparent, parent, and child, the product with the child category would get all the properties

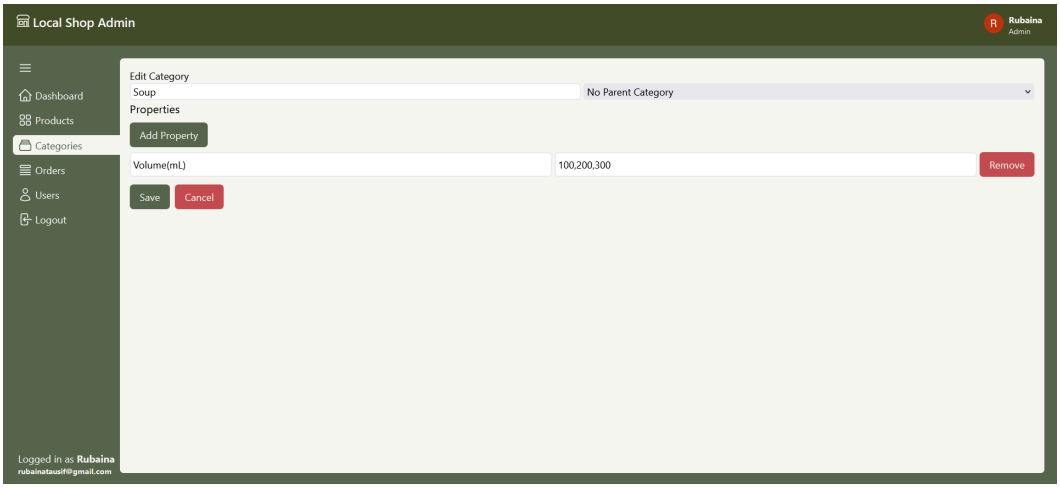
from, grandparent, parent and child. Once you have finished creating a category, press “save” to save it to the database or “cancel” to cancel.





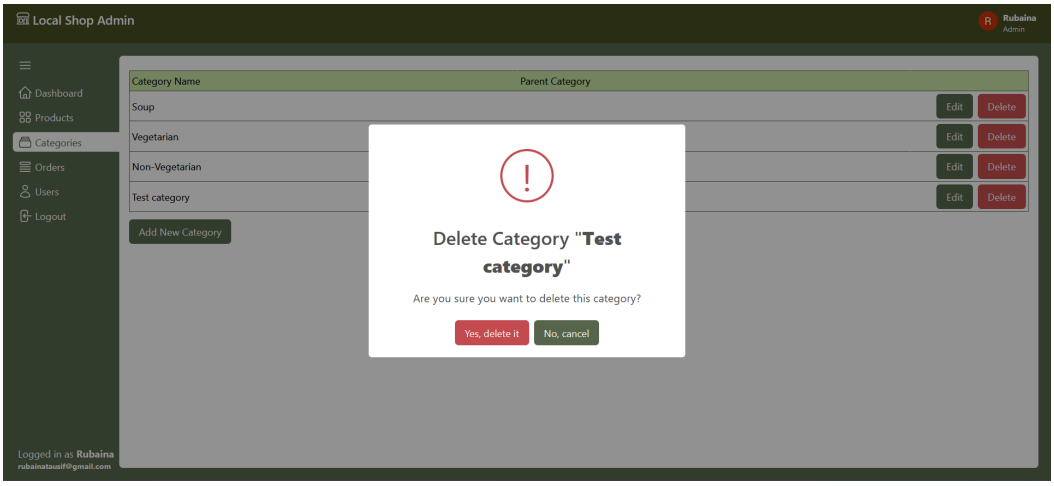
Edit a Category:

To edit a category, first navigate to the category in the list you want to edit and select the edit button in the appropriate row. This will take you to the same form that appeared when creating the category, prefilled with the correct information for the product. You can edit this however you wish and press “save” to save it to the database. To cancel any changes, simply click on the “cancel” button.



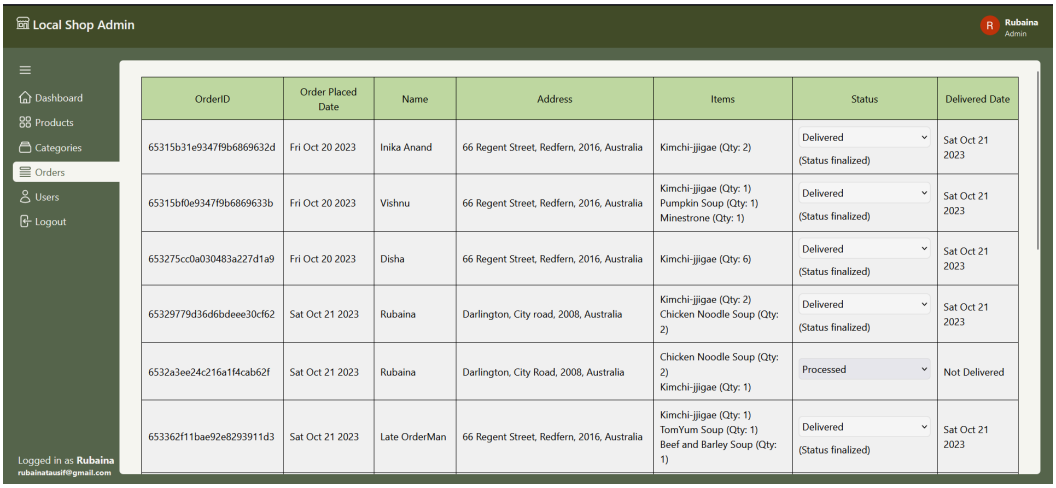
Delete a Category:

To delete a category, first navigate to the category in the list you want to delete and select the delete button in the appropriate row. This will trigger a popup to confirm whether you want to delete or not. Click yes to delete and no to cancel. Both options navigate you back to the categories page.



Orders

When opening the orders page, you are presented with a table presenting all orders, by date created. If you are on a phone, tapping on a row will display all information, this is done by default on the computer. When processing orders, you have the ability to change the status of an order between, paid, processed and shipped. Once an order is selected as delivered you are no longer able to edit the status and it will appear as delivered on that date.

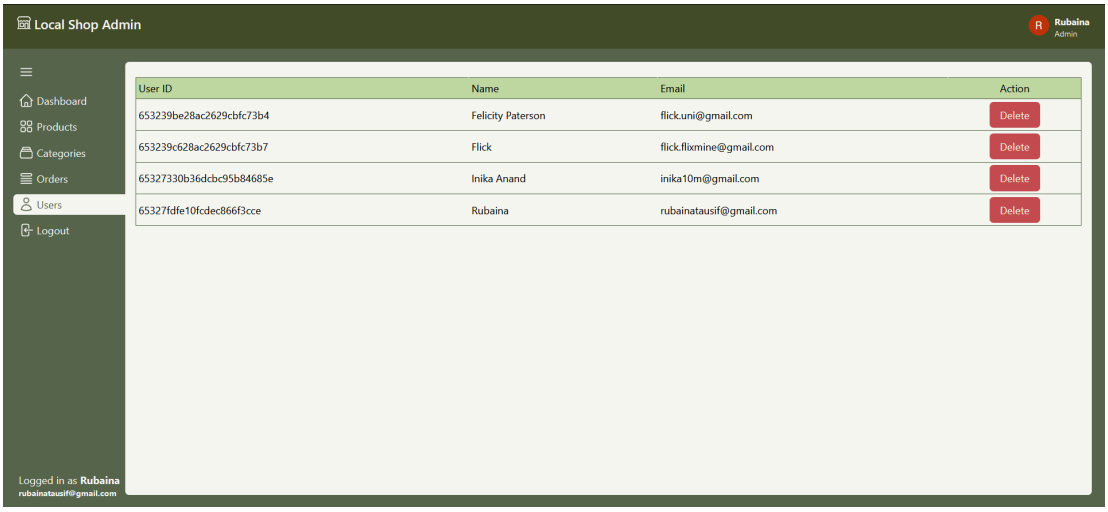


OrderID	Order Placed Date	Name	Address	Items	Status	Delivered Date
65315b31e9347f9b6869632d	Fri Oct 20 2023	Inika Anand	66 Regent Street, Redfern, 2016, Australia	Kimchi-jjigae (Qty: 2)	Delivered (Status finalized)	Sat Oct 21 2023
65315bf0e9347f9b6869633b	Fri Oct 20 2023	Vishnu	66 Regent Street, Redfern, 2016, Australia	Kimchi-jjigae (Qty: 1) Pumpkin Soup (Qty: 1) Minestrone (Qty: 1)	Delivered (Status finalized)	Sat Oct 21 2023
653275cc0a030483a227d1a9	Fri Oct 20 2023	Disha	66 Regent Street, Redfern, 2016, Australia	Kimchi-jjigae (Qty: 6)	Delivered (Status finalized)	Sat Oct 21 2023
65329779d36d6bdeee30cf62	Sat Oct 21 2023	Rubaina	Darlington, City road, 2008, Australia	Kimchi-jjigae (Qty: 2) Chicken Noodle Soup (Qty: 2)	Delivered (Status finalized)	Sat Oct 21 2023
6532a3ee24c216a1f4cab62f	Sat Oct 21 2023	Rubaina	Darlington, City Road, 2008, Australia	Chicken Noodle Soup (Qty: 2) Kimchi-jjigae (Qty: 1)	Processed	Not Delivered
653362f11bae92e8293911d3	Sat Oct 21 2023	Late OrderMan	66 Regent Street, Redfern, 2016, Australia	Kimchi-jjigae (Qty: 1) TomYum Soup (Qty: 1) Beef and Barley Soup (Qty: 1)	Delivered (Status finalized)	Sat Oct 21 2023

65337348cb9111b713f74726	Sat Oct 21 2023	Inika 2.0	66 Regent Street, Redfern, 2016, Australia	Chicken Noodle Soup (Qty: 2)	Delivered (Status finalized)	Sat Oct 21 2023
6533734bd9d06e49fd5421e3	Sat Oct 21 2023	Felicity Paterson	42 Wallaby Way, Sydney, 2000, Australia	Kimchi-jjigae (Qty: 1) TomYum Soup (Qty: 1)	Paid	Not Delivered
65337a45cb9111b713f7475a	Sat Oct 21 2023	Inika Anand	66 Regent Street, Redfern, 2016, Australia	Minestrone (Qty: 1) TomYum Soup (Qty: 1)	Delivered (Status finalized)	Sat Oct 21 2023
65337be9cb9111b713f74776	Sat Oct 21 2023	Inika Anand	66 Regent Street, Redfern, 2016, Australia	TomYum Soup (Qty: 1) Beef and Barley Soup (Qty: 1)	Delivered (Status finalized)	Sat Oct 21 2023
653387bdd9d06e49fd54224b	Sat Oct 21 2023	Felicity	42 Wallaby Way, Sydney, 2000, Australia	Minestrone (Qty: 22)	Paid	Not Delivered
6533891e3b64fb898e5ea820	Sat Oct 21 2023	Ryan Pada	66 Regent Street, Redfern, 2016, Australia	Minestrone (Qty: 1) TomYum Soup (Qty: 1)		Sat Oct 21 2023
65338d58d9d06e49fd542261	Sat Oct 21 2023	Felicity	123 NAH, Sydney, 2000, Australia	Minestrone (Qty: 20)		Not Delivered
65338fa8d9d06e49fd5422a8	Sat Oct 21 2023	Rohan	321 Somewhere, Sydney, 2000, Australia	Kimchi-jjigae (Qty: 5)	Processed	Not Delivered

Users

When opening the users page, you are presented with a list of all users who have created an account. If you are on a phone, tapping on a row will present all the information. If you click delete, this will delete the user's account.



User ID	Name	Email	Action
653239be28ac2629cbfc73b4	Felicity Paterson	flick.uni@gmail.com	Delete
653239c628ac2629cbfc73b7	Flick	flick.flixmine@gmail.com	Delete
65327330b36dcbc95b84685e	Inika Anand	inika10m@gmail.com	Delete
65327fdfe10fcdec866f3cce	Rubaina	rubainatausif@gmail.com	Delete

Login/Logout

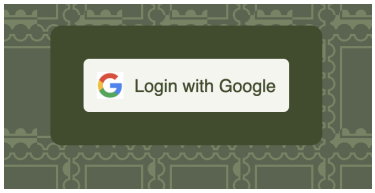
If you select the bottom button on the navbar, this will log you out and take you to the logged out screen (the first screen before you logged in)

2.2. Shop Front

Note: All path names are relative to the localShopFront folder.

Landing Page:

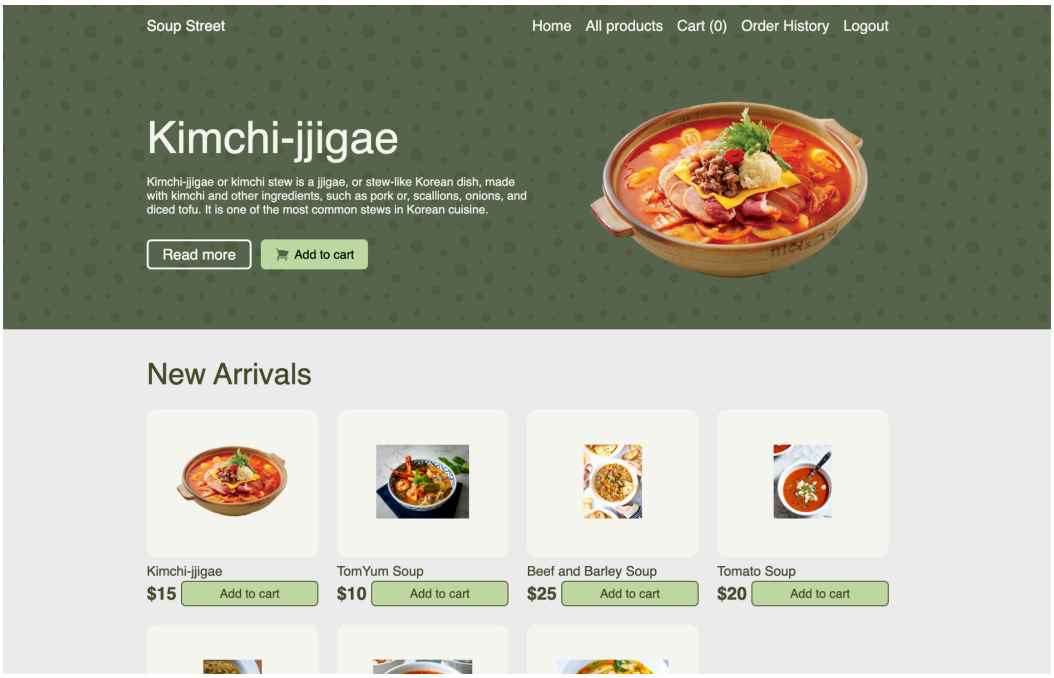
When you correctly enter localHost3001 in your browser, the landing page will show up with a login button and an about us section. Click on login to log in with your google account. The application will take you to a new page where you will be able to enter your login details,



followed by taking you to the homepage upon completion of the login.

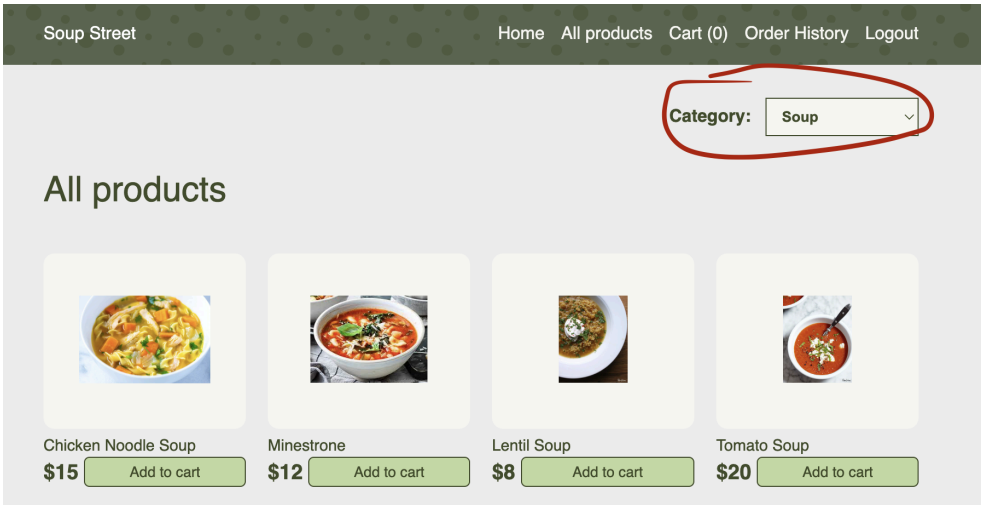
Homepage:

The homepage comprises a features section where you can see our features project. There is also a top navigation. The add-to-cart buttons allow you to add a product to your cart. Each product has an image and a price being displayed. Use the top navigation bar to move to different pages.



All Products:

The All Products page allows you to view all the products. Similar to the homepage you can add an item to your cart by clicking the add to cart button.

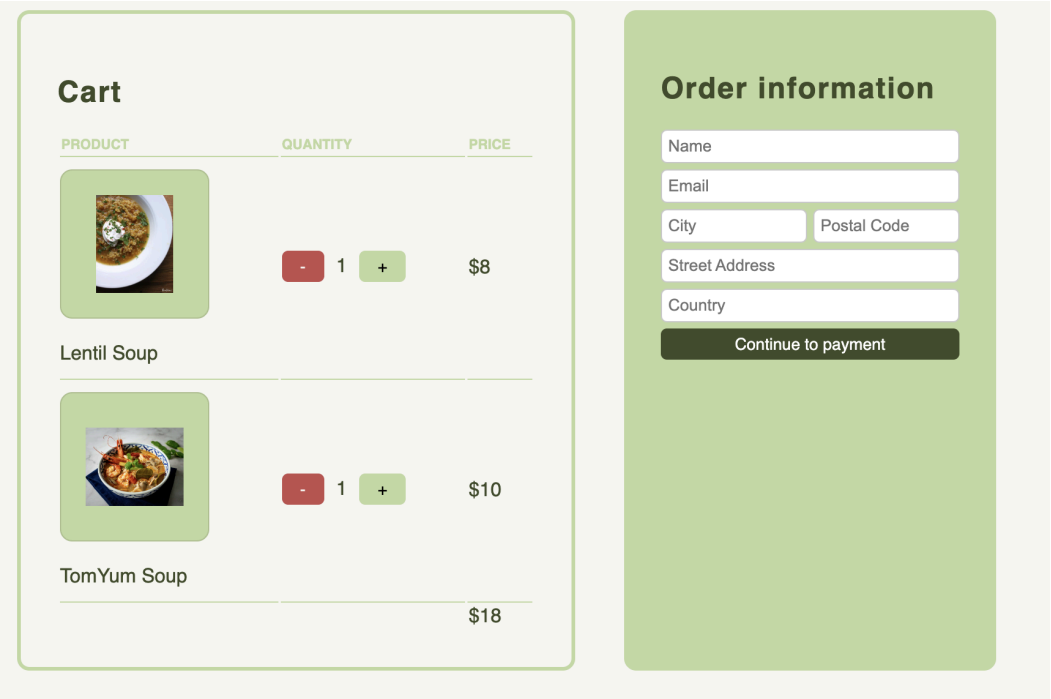


The categories button at the top allows you to filter the products based on categories. Click the button and select a category from the dropdown.
Additionally, you can click on a product to view the product description on a separate page.

This also allows the user to view other images of the product. Simply click the back button or the links in the top navigation bar to navigate off this page.

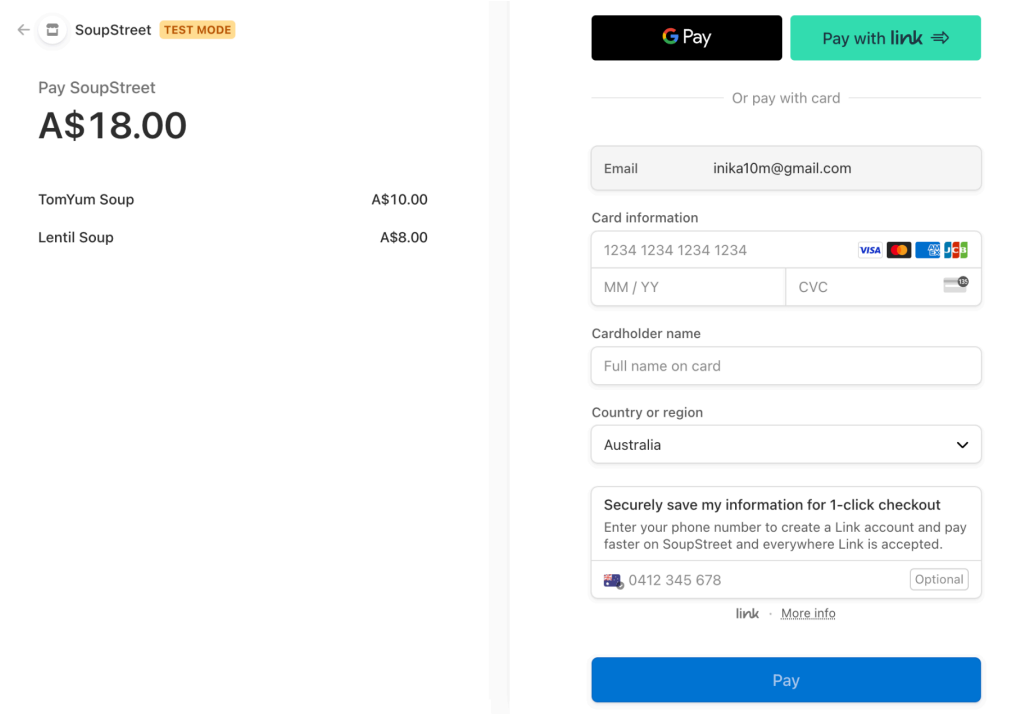
Cart

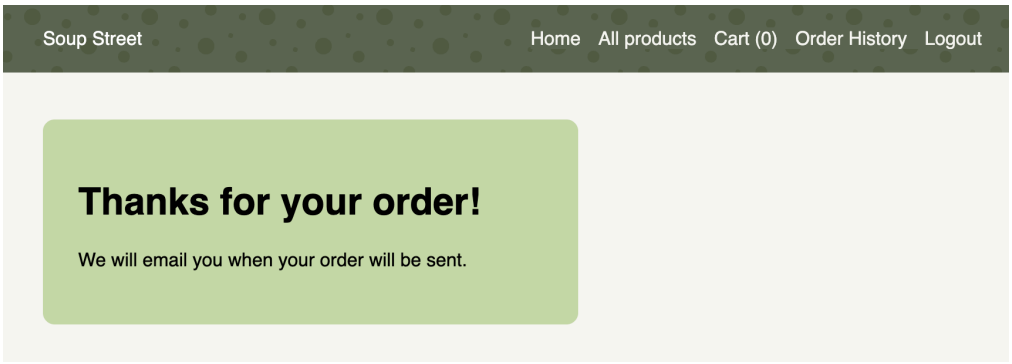
Once you click the cart button at the top of the navigation, you will be able to view the cart page. The cart page displays your order information along with the ability to increase or decrease the quantity of the items you have purchased. Click the add or subtract buttons to change the quantity of a particular item. The cart also displays the total price at the bottom which is the price the user will be charged for. Enter the order information in the order information section and click the continue to payment button.



Payment:

This will take you to a new payment page where you will be able to see the total amount that you are paying and also be able to enter your payment details. Enter your correct payment details or click the buttons Gpay or pay with the link based on your payment method. After entering the correct details respectively click the pay button at the bottom of the page.





Upon successful payment, you will be redirected to a new page that will thank you for your purchase. Click the top navigation to navigate out of that page.

Order History:

In order to view your order history click on the order history button in the top navigation. You will be able to view your orders and your order status. Your most recent orders will be added at the bottom of the page. You can view all your past orders on this page along with the status and the delivery date. When you have

completed the payment you can successfully change whether or not the order has appeared on this page with the status set as paid.

Orders

OrderID	Order Placed Date	Name	Address	Items	Status	Delivered Date
65315b31e9347f9b6869632d	Fri Oct 20 2023	Inika Anand	66 Regent Street, Redfern, 2016, Australia	• Kimchi-jjigae (Qty: 2)	delivered	Sat Oct 21 2023
653362f11bae92e8293911d3	Sat Oct 21 2023	Late OrderMan	66 Regent Street, Redfern, 2016, Australia	• Kimchi-jjigae (Qty: 1) • TomYum Soup (Qty: 1) • Beef and Barley Soup (Qty: 1)	delivered	Sat Oct 21 2023
65337348cb9111b713f74726	Sat Oct 21 2023	Inika 2.0	66 Regent Street, Redfern, 2016, Australia	• Chicken Noodle Soup (Qty: 2)	delivered	Sat Oct 21 2023
65337a45cb9111b713f7475a	Sat Oct 21 2023	Inika Anand	66 Regent Street, Redfern, 2016, Australia	• Minestrone (Qty: 1) • TomYum Soup (Qty: 1)	delivered	Sat Oct 21 2023
65337be9cb9111b713f74776	Sat Oct 21 2023	Inika Anand	66 Regent Street, Redfern, 2016, Australia	• TomYum Soup (Qty: 1) • Beef and Barley Soup (Qty: 1)	delivered	Sat Oct 21 2023
6533891e3b64fb898e5ea820	Sat Oct 21 2023	Ryan Pada	66 Regent Street, Redfern, 2016, Australia	• Minestrone (Qty: 1) • TomYum Soup (Qty: 1)	delivered	Sat Oct 21 2023
65346c07795e283562b7fef3	Sun Oct 22 2023	Inika Anand	66 Regent Street, Redfern, 2016, Australia	• TomYum Soup (Qty: 1) • Lentil Soup (Qty: 1)	paid	Not Delivered

Logout:

Click the logout button in the top navigation to logout of the page. This will take you back to the landing page. In order to access the homepage again you will have to login once again.

3. Design Documents

3.1. System Architecture

The system architecture of LocalShop is organised around two primary actors: the customer and the admin. Notably, the admin and user websites operate as distinct entities, communicating directly with the database independently. This design ensures that a potential crash or malfunction in one website does not impact the other, improving system reliability.

On the admin side, interaction occurs through a visual interface, where the admin can manage orders, products, and users. This interface initiates requests for data retrieval from the database based on admin actions. Messages are exchanged between the admin interface and the database to facilitate data management.
These messages create the flow of information, enabling the admin to efficiently interact with and manipulate data through the website. The separation of concerns between the admin and user websites minimises the risk of cascading failures.

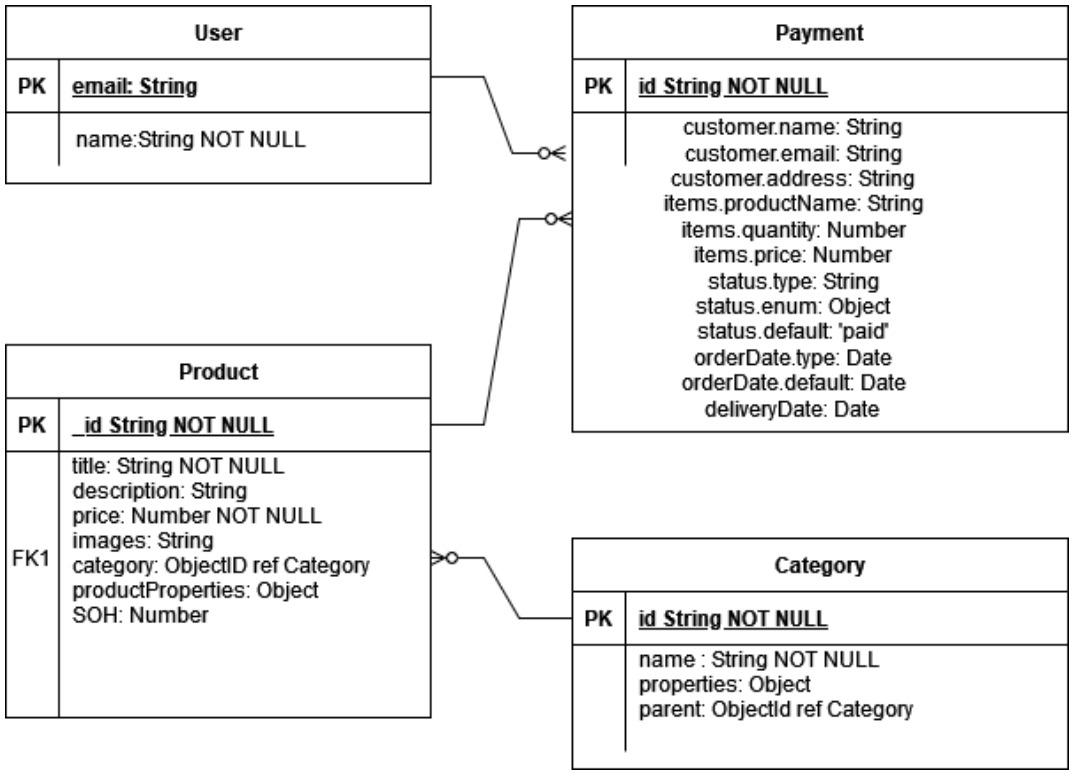
On the user side, data retrieval is direct from the database, allowing users to access and view product information. The user interface fetches information from the database to display an array of products. Followed by posting the products

related to an order and additional order details. The payment gateway, integrated using Stripe, operates as an external component. When users initiate payments, the architecture involves messages being sent to the Stripe platform for processing. Payment transactions are securely stored on the Stripe business account, ensuring a high level of security.

In summary, the system architecture has efficient communication between the admin and user interfaces and the database. This separation of concerns and the integration of external components contribute to a flexible and secure e-commerce platform.

3.2. Database Management Design

The following diagram is our database schema that encapsulates the design thinking behind the creation of the database.



Description

1. User:
    - Represents individual customers who sign in using their Google email.

- Each user can place orders, view products and select categories

2. Payment:
   - Represents orders placed by the users that have been paid for.
   - It contains information related to their Order and Payment such as their transaction details which have been implemented using Stripe, amount paid, date of payment, items purchased, etc.
3. Product:
   - Represents the different soups available in the shop.
   - Each product has details such as name, price, image, and Stock On Hand (SOH).
4. Category:
   - Represents the classification of products. For instance, products can be categorised as “Vegetarian”, “Non-Vegetarian”, etc.
   - Helps users to filter and choose products based on their preferences.

The connection between the entities suggests that a User can have multiple Payments (orders) which means that they can place multiple orders over time. The Products table that is related to Payments is designed in a way so that each order can have one or more products. The products of the Products table can belong to a specific Category. Overall, this schema enables the local soup shop to manage its user base, track orders, organise its product offerings, and categorise products for easier browsing by the customers

3.3. Security Design

Our decision to implement Google login within our Next.js application was driven by focusing on both security and user-centric design. We chose Google login for its proven security infrastructure. This not only improves our application against potential threats but also provides a seamless user-friendly login experience.

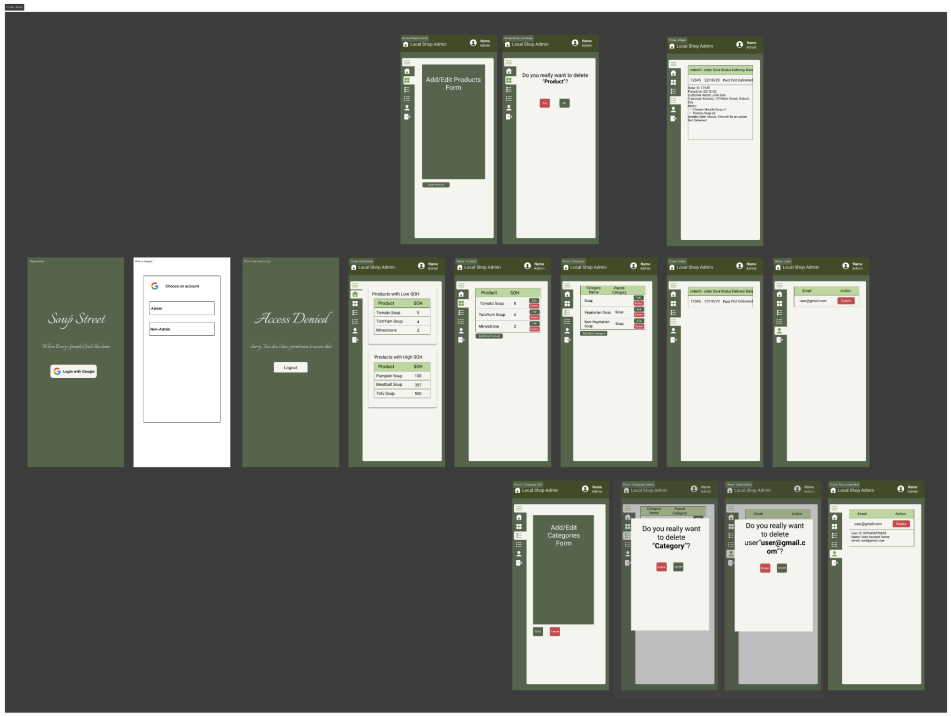
Google login is a trusted authentication platform, which also helps instil confidence within our users when logging in. This design decision not only helps with authentication but also allows the user to onboard successfully and contribute to a positive user experience.

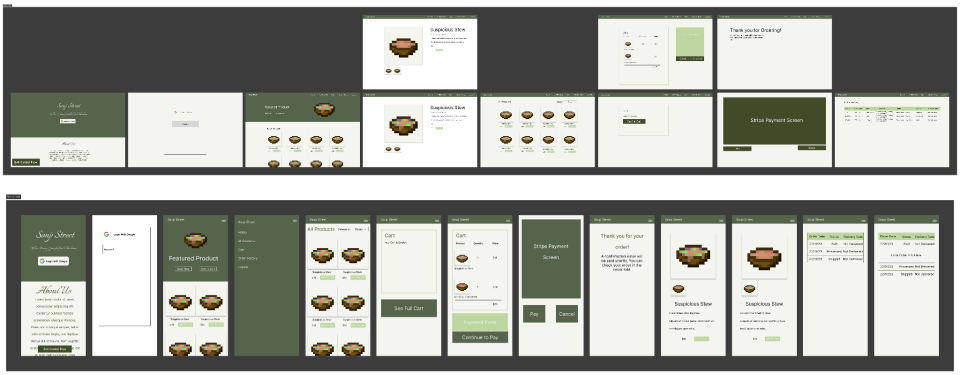
3.4. Usability Design

In accordance with usability design, we designed the UX/UI features on Figma. We designed the top navigation and the connections between web pages based on minimising clicks and providing a quick and easy buy online experience.

The following is a link to the wireframes designed.
https://www.figma.com/file/cnLtnlEw3Qagpxh5am7kxn/Mockup?type=design&node-id=0%3A1&mode=design&t=BZyHi6Lx0jdjaCdr-1
To View wireframes; Follow the link- Click Present in the top right corner and follow the buttons to the desired control flow. However, please note that the initial control flow selection is not part of the design.

We designed wireframes for the website on different devices which included a desktop computer and a mobile phone. This helped us plan our responsive design by following the wireframes and control flow and replicating it in our application.





4. Implementation Steps

4.1. User Interface Implementation

To implement the user interface, the front-end code leverages the MERN (MongoDB, Express.js, React, Node.js) stack, with a focus on server-side rendering facilitated by the use of Next.js. API routes handle product and category data retrieval, demonstrating a RESTful architecture. The React frontend utilises state management through hooks like useState and useEffect, providing dynamic updates and seamless user interactions.

The codebase was divided into components to increase the usability of code and maintainability. Each component designed was used at least twice. For example, the buttons component in the code base was used when calling buttons on different pages. This removed the possibility of redundant code and improved brand association with the type and colour of the button being standard across different pages.

A certain colour scheme was also applied and used throughout the entire website including the admin side. In terms of front-end user interface design, the design principles of simplicity and consistency were thoroughly applied throughout the website keeping the layout minimalistic yet consistent throughout all webpages.

Lastly, the implementation of the admin side and the shop front was made responsive in order to increase the user experience regardless of the device. This was an important feature in order to ensure that admins and users can access the application regardless of the device.

4.2. Database Implementation

The database implementation was done through MongoDB. MongoDB's flexible, schema-less data model was well-suited for the dynamic and evolving nature of our e-commerce data. This flexibility allows for easy modification and adaptation as the structure of your product and user data evolves over time.

MongoDB was also implemented when considering scalability. In an e-commerce website, scalability is an essential aspect. MongoDB is scalable in terms of the distribution of data across multiple servers. This ensures that the database can handle increasing amounts of traffic and data without sacrificing the performance of the website.

The choice of using MongoDB is further supported by the wide array of resources available to support implementation both in libraries as well as information on how to implement it. It is a widely supported database that supports many languages and frameworks. Combined with nextJS, there are provided adapters to connect the database and the authentication service.

4.3. Feature Development

4.3.1. Product Catalogue

The user-friendly product catalogue implemented in the website incorporates essential features to enhance the overall browsing experience. Each product is displayed, featuring its name, description, price, and accompanying images. The product details are fetched from a MongoDB database using the Mongoose library. All products are also displayed in a catalogue format that allows the user to quickly browse through all the products.

To allow for easy navigation and discovery, a categorization system has been implemented. Categories are retrieved from the backend and presented as options in a dropdown menu, allowing users to filter products based on their interests. This categorization is implemented using React state management, ensuring real-time updates as users interact with the dropdown menu.

The filtering functionality also integrates the HTML select input styled with styled-components, providing an aesthetically pleasing user interface. When a category is selected, the product grid updates to display only the relevant products, enhancing the efficiency of the browsing experience.

The implementation of the product management feature in the admin panel provides administrators with the ability to add, edit, and delete products. Using the Next.js framework and React library, the code establishes a connection to the backend and fetches the product data from the designated API endpoint. The fetched product information is then displayed in a tabular format on the admin panel, offering a clear and organised overview that includes essential details such as product name and stock on hand (SOH).

The users can interact with the application through intuitive buttons allowing them to navigate to a specific product when editing and deleting. The `handleEditRouting` and `handleDeleteRouting` functions use Next.js's `useRouter` hook to route administrators to the respective product editing or deletion pages when the corresponding buttons are clicked. The implementation is responsive, ensuring a consistent user experience across different screen sizes.

4.3.2. Shopping Cart

The implemented shopping cart in this codebase offers a set of features for users to interact with products. Using React, styled-components for a visually appealing design, and Next.js for server-side rendering, the shopping cart provides users with the ability to add, update quantities, and remove items. The CartContext, implemented as a React context, plays a crucial role in managing the state of the shopping cart across different components. Local storage (localStorage) is utilised to persist the cart data, ensuring a consistent user experience even when users navigate away or refresh the page.

The CartPage component fetches product details from the backend based on the products stored in the cart. Users can increment or decrement the quantity of items with the buttons added, and the total price is automatically calculated based on product prices. Additionally, the page includes an Order Information section where users can input their details, and upon proceeding to checkout, the user is redirected to the payment page through a call to the /api/checkout endpoint.

The shopping cart is both aesthetic and functional, with styled-components ensuring a visually cohesive user interface. To enhance user experience,

the checkout process includes a success page, and the cart is automatically cleared after a successful transaction. The CartContextProvider efficiently manages cart state and persistence.

4.3.3. Secure Payment Gateway

The payment functionality is underpinned by a model named Payment, defined using Mongoose. This model schema comprises fields capturing order details (like line items and address) and a paid field, indicating if the payment has been made. With timestamps enabled, it helps track when a payment record was created or modified.

Within the CartPage component, there is an async function named goToPayment. When a user decides to proceed with the payment after filling out their order information, this function is triggered. An API call is made to /api/checkout endpoint, sending over the user's order and address details.

By using services like Stripe, the application benefits from a robust, secure online payment system. Such services handle the complex tasks of encrypting and processing payment details, allowing us to integrate payment features without directly dealing with sensitive information. Stripe's platform is replete with validation checks, ensuring that details like card expiration dates and input formats are properly verified. It's also worth noting that this platform offers a sandbox environment, enabling us to test the payment process without actual financial transactions. To verify our payment implementation, we conducted tests using a dummy card provided by Stripe. By entering valid card details, we were able to simulate a transaction and ensure that the payment process was functioning correctly. By offloading payment processing to trusted third-party services, the application ensures that users' payment information is encrypted, protected, and processed securely.

4.3.4. Order Management System

We've built an Order Management System to help both customers and admins. This system uses MongoDB as our database to keep track of all orders. In this database, every order has its own ID, customer details like name and address, a list of the items bought, and the current status of the order.

Customers have the convenience of accessing their order history, providing insights into their past purchases and the current status of their orders. They can determine whether an order is still being processed, has been shipped out, or has already been delivered.

For admins, a specialised view is available that displays orders from all customers in a table format, offering a comprehensive overview at a glance. We make sure our users always know what's happening with their orders. They can quickly check if their order is 'paid', 'processed', 'shipped', or 'delivered' for up-to-date information, granting them real-time updates. Moreover, admins possess the capability to modify the order status via a dropdown box, enabling swift status updates in the system when orders are shipped or delivered.

4.3.5. User Authentication and Authorisation

The application uses the Next.js framework along with the Next Auth library for user authentication. The primary method of authentication provided is through Google, allowing users to sign in using their Google accounts. This feature can be found on the main page, clearly labelled as "Login with Google."

Beyond just authentication, the system also implements user authorisation. This is done by maintaining a specific list of email addresses identified as administrators. When a user logs in, their email address is cross-referenced with this list. If there's a match, they are assigned the 'admin' role and are redirected to the admin dashboard, which provides access to a range of administrative functionalities. The entire implementation is wrapped in a higher-order component (`withAdminProtection`) to enforce security measures, ensuring that only authorised administrators can access and perform actions within the admin panel. However, if the email doesn't match any in the list, they are treated as a regular user and will not be able to access the admin panel.

In summary, the application ensures that users can easily authenticate using familiar methods, and it further categorises them based on predefined roles. This ensures that only those designated as administrators can access the admin panel, maintaining system security and role-based access.

5. Testing Reports and Results

5.1. Unit Testing

For each feature we added, we ran unit tests to ensure they functioned properly.

Testing dashboard page: For the dashboard rendering, we ran tests to ensure that it gets rendered without crashing and displays the loading page properly. We also tested if it rendered the login state when the user was not logged in.

```
import { render } from "@testing-library/react";
import Dashboard from "../pages/dashboard";
import { useSession } from "next-auth/react";
import axios from 'axios';

jest.mock('axios');
jest.mock('next-auth/react');

jest.mock('../__mocks__/localShop/hoc/withAdminProtection');

describe('Dashboard'), () => {

  test('renders without crashing', () => {
    render(<Dashboard />);
  });

  test('renders loading state', () => {
    useSession.mockReturnValueOnce({
        status: 'loading'
    });

    render(<Dashboard />);
    expect(screen.getByText("Loading...")).toBeInTheDocument();
});

  test('renders login state when not logged in', () => {
    useSession.mockReturnValueOnce({
        status: 'unauthenticated'
    });

    render(<Dashboard />);
    expect(screen.getByText("Login with Google")).toBeInTheDocument();
  });

}
```

```
PASS tests/dashboard.test.js (19.325 s)
  ✓ renders without crashing (13 ms)
  ✓ renders loading state (1 ms)
  ✓ renders login state when not logged in (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        20.144 s
```

Testing orders page: For the orders page in the admin and the user side, we tested if they can be rendered without crashing, showing all the orders. Additionally, we created dummy data to see if our order history page can render that properly on both the client and admin side as well. We have also created a toggle feature that will expand the rows on small screens to see if our display is scalable and readable as well.

```
Run | Debug
14  describe('Orders Component', () => {
15    afterEach(() => {
16      jest.clearAllMocks();
17    });
18
      Run | Debug
19    test('it renders without crashing', () => {
20      render(<Orders />);
21      expect(screen.getByText(/OrderID/i)).toBeInTheDocument();
22    });
23
24
```

```
Run | Debug
25  test('displays orders correctly', async () => {
26    // Mock an example order
27    const exampleOrder = {
28      _id: '12345',
29      orderDate: new Date().toISOString(),
30      customer: {
31        name: 'John Doe',
32        address: '123 Street',
33      },
34      items: [
35        { productName: 'Product A', quantity: 2 },
36        { productName: 'Product B', quantity: 1 },
37      ],
38      status: 'paid',
39    };
40
41    global.fetch.mockImplementationOnce(() =>
42      Promise.resolve({
43        json: () => Promise.resolve([exampleOrder]),
44      })
45    );
46
47    render(<Orders />);
48    await waitFor(() => expect(screen.getByText(exampleOrder._id)).toBeInTheDocument());
49  });
50
```

```
Run | Debug
51  test('toggles expanded rows in small screens', async () => {
52    global.innerWidth = 500;
53
54    const exampleOrder = {
55        _id: '12345',
56        orderDate: new Date().toISOString(),
57        customer: {
58          name: 'John Doe',
59          address: '123 Street',
60        },
61        items: [
62          { productName: 'Product A', quantity: 2 },
63          { productName: 'Product B', quantity: 1 },
64        ],
65        status: 'paid',
66      };
67
68    render(<Orders />);
69    await waitFor(() => expect(screen.getByText('Order ID: 12345')).toBeInTheDocument());
70
71    // Check that the expanded row is hidden by default
72    expect(screen.queryByText(/Customer Name:/i)).not.toBeInTheDocument();
73
74    // Simulate a click on the row to expand it
75    const row = screen.getByText(exampleOrder._id);
76    userEvent.click(row);
77    // Check that the expanded row is now visible
78    expect(screen.getByText(/Customer Name:/i)).toBeInTheDocument();
79  });
80  });
```

```
PASS tests/orders.test.js (14.914 s)
  ✓ it renders without crashing (11 ms)
  ✓ displays orders correctly
  ✓ toggles expanded rows in small screens

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        15.694 s, estimated 24 s
```

Testing product page: In the testing products page, we have tested if we can fetch and display all the products we have in the database. For that we have created a mockData containing some products and its related information and used that to render the page. Our second test was to test the edit route of the products page and to check if it handles the ID in its request appropriately. We did the same for the delete route where we were testing if we could delete a product with itsgh8 id given in the request. The last test was to test if we were rendering the Add New Product button correctly.

```
Run | Debug
describe('Products component', () => {
  const mockPush = jest.fn();
  useRouter.mockImplementation(() => ({
    push: mockPush,
  }));

  beforeEach(() => {
    jest.clearAllMocks();
  });

  Run | Debug
  test('fetches and displays products', async () => {
    const mockData = [
      { _id: '123', title: 'Product 1', SOH: 10 },
      { _id: '456', title: 'Product 2', SOH: 5 },
    ];

    axios.get.mockResolvedValueOnce({ data: mockData });

    render(<Products />);

    await waitFor(() => expect(screen.getByText('Product 1')).toBeInTheDocument());
    expect(screen.getByText('Product 2')).toBeInTheDocument();
  });
```

```
37    test('handles edit routing correctly', async () => {
38      const mockData = [{ _id: '123', title: 'Product 1', SOH: 10 }];
39
40      axios.get.mockResolvedValueOnce({ data: mockData });
41
42      render(<Products />);
43
44      await waitFor(() => expect(screen.getByText('Product 1')).toBeInTheDocument());
45
46      fireEvent.click(screen.getByText('Edit'));
47      expect(mockPush).toHaveBeenCalledWith('/products/edit/123');
48    });
49
      Run | Debug
50    test('handles delete routing correctly', async () => {
51      const mockData = [{ _id: '123', title: 'Product 1', SOH: 10 }];
52
53      axios.get.mockResolvedValueOnce({ data: mockData });
54
55      render(<Products />);
56
57      await waitFor(() => expect(screen.getByText('Product 1')).toBeInTheDocument());
58
59      fireEvent.click(screen.getByText('Delete'));
60      expect(mockPush).toHaveBeenCalledWith('/products/delete/123');
61    });
62
```

```
63    test('renders Add New Product button correctly', () => {
64      render(<Products />);
65      const addButton = screen.getByText('Add New Product');
66      expect(addButton).toBeInTheDocument();
67      expect(addButton.closest('a')).toHaveAttribute('href', '/products/new');
68    });
69  });
70
```

```
PASS tests/products.test.js (16.891 s)
  ✓ fetches and displays products (12 ms)
  ✓ handles edit routing correctly
  ✓ handles delete routing correctly
  ✓ renders Add New Product button correctly (1 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:   0 total
Time:        17.628 s
```

Testing users page: The last component we tested was the users page. Here we ran tests that checked if we were able to fetch and display the users in our database. On the admin side, we have also implemented the feature of being able to delete a user and we ran a test on that too to see if we were successfully able to delete a user using the delete button. Lastly, we tested if we could render the user information on a smaller screen.

```
describe('Users component', () => {

  beforeEach(() => {
    fetch.mockClear();
  });

  Run | Debug
  test('fetches and displays users', async () => {
    const mockData = [
      { _id: '1', name: 'John Doe', email: 'john.doe@example.com' },
      { _id: '2', name: 'Jane Smith', email: 'jane.smith@example.com' }
    ];

    fetch.mockResolvedValueOnce({
      ok: true,
      json: async () => mockData
    });

    render(<Users />);

    await waitFor(() => expect(screen.getByText('John Doe')).toBeInTheDocument())
    expect(screen.getByText('Jane Smith')).toBeInTheDocument();
  });
```

```
  test('deletes a user on delete button click', async () => {
    const mockData = [
      { _id: '1', name: 'John Doe', email: 'john.doe@example.com' },
    ];

    fetch.mockResolvedValueOnce({
      ok: true,
      json: async () => mockData
    });

    fetch.mockResolvedValueOnce({
      ok: true,
    });

    render(<Users />);

    await waitFor(() => expect(screen.getByText('John Doe')).toBeInTheDocument());

    fireEvent.click(screen.getByText('Delete'));

    await waitFor(() => expect(screen.queryByText('John Doe')).not.toBeInTheDocument());
  });
```

```
54    test('expands row content in smaller screen widths', async () => {
55      global.innerWidth = 800; // Mocking stage1 condition
56      const mockData = [
57        { _id: '1', name: 'John Doe', email: 'john.doe@example.com' },
58      ];
59
60      fetch.mockResolvedValueOnce({
61        ok: true,
62        json: async () => mockData
63      });
64
65      render(<Users />);
66
67      await waitFor(() => expect(screen.getByText('John Doe')).toBeInTheDocument());
68
69      fireEvent.click(screen.getByText('john.doe@example.com'));  // clicking the row
70
71      expect(screen.getByText('User ID: 1')).toBeInTheDocument();
72      expect(screen.getByText('Name: John Doe')).toBeInTheDocument();
73    });
74
75  });
```

```
PASS tests/users.test.js (43.967 s)
  ✓ fetches and displays users (35 ms)
  ✓ deletes a user on delete button click (1 ms)
  ✓ expands row content in smaller screen widths (1 ms)

Test Suites: 1 passed, 1 total
Tests:       3 passed, 3 total
Snapshots:   0 total
Time:        45.963 s
```

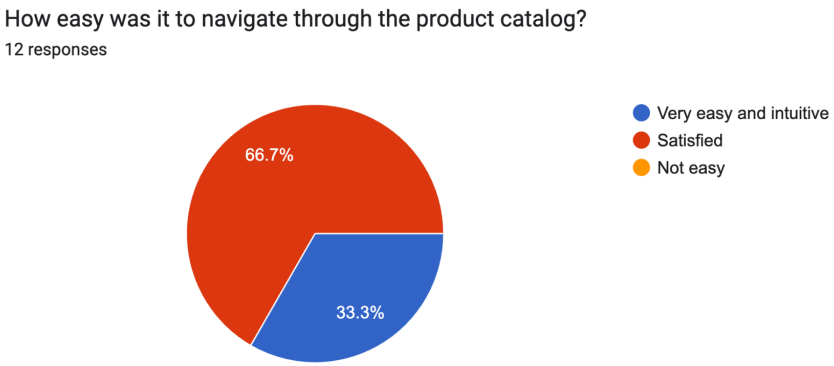
5.2. User Testing

We conducted a comprehensive user testing session with 12 participants. Initially, each user was provided an opportunity to interact with and test the implemented features of our frontend interface. Following this, we invited them to participate in a survey, which aimed to gather insights and feedback on their user experience.

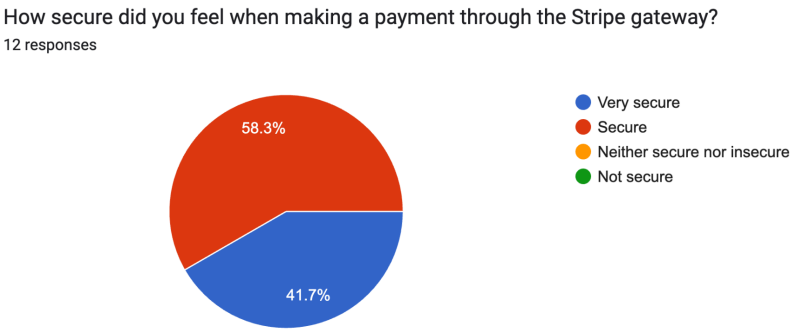
The survey encompassed several questions directly related to the features they had just interacted with. You can view the exact survey questions in the screenshots provided below or directly access the Google Form through this link.

The insights derived from the survey responses (response sheet) revealed:

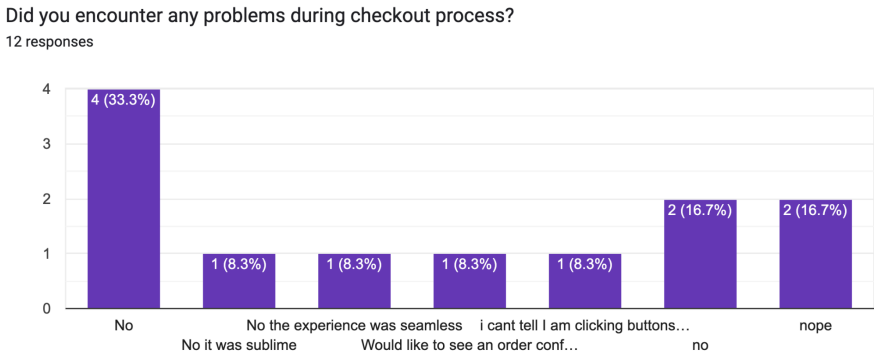
- Most users appreciated the ease of navigating the product catalogue and filtering products.



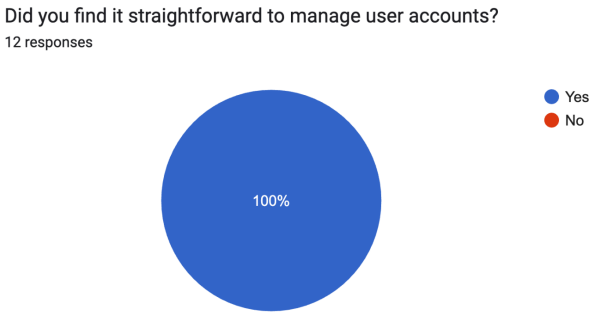
- Checkout through Stripe was perceived as secure by a majority of users.



- There's a recommendation to enhance order placement with direct notifications detailing ordered products and estimated arrival dates.



- Admins found the addition of categories, products, and user account management to be straightforward.



From this we can conclude that overall, the platform is well-received, but some refinements could further enhance user satisfaction.