# System Implementation

## Architecture

For backend, the top layer is Spring Data JPA(Repository). It actually takes the advantage of the JPA specification, including the entity and association mappings, the entity lifecycle management, and JPA’s query capabilities. Besides, it has extra functions which do not require codes to implement with the repository pattern on a higher abstraction level. And it also provides the function which help generate MySQL queries automatically based on the method names.

The third layer is Hibernate, which is an object–relational mapping tool for the Java programming language. It provides a framework for mapping an object-oriented domain model to a relational database. Hibernate handles object–relational impedance mismatch problems by replacing direct, persistent database accesses with high-level object handling functions.

<https://en.wikipedia.org/wiki/Hibernate_(framework)>

The bottom layer is JDBC( Java Database Connectivity ). It is just an api for Java which defines the way clients access the database. The classes and interfaces of JDBC allow the application to send requests made by users to the specified database. It can be seen as a bridge between the Java application and relational databases(MySQL)

<https://en.wikipedia.org/wiki/Java_Database_Connectivity>

<https://www.geeksforgeeks.org/introduction-to-jdbc/>

### **Front-end**

The front-end architecture of our project mainly includes four parts, they are Vue Components, Router, Vuex Store, and Axios. Vue Components is mainly used for UI rendering. Router is used to manage page navigation and jump. Vuex Store is a centralized storage to manage the state of all components. Axios is a promise-based network request library. Each of them will be introduced in detail in the following sections. Figure 4-1-1-1 is the architecture diagram for the front-end.

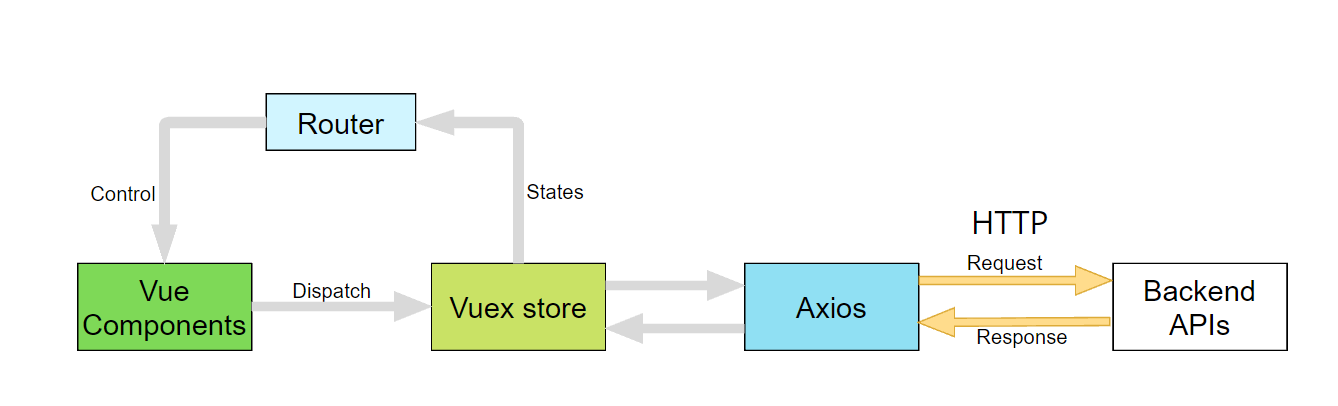


Figure 4-1-1-1: Front-end Architecture Diagram

For the front-end design, we use Vue.js 3.0 or Vue 3 which is a JavaScript framework for building the user interfaces. It builds on top of standard HTML, CSS, and JavaScript and provides a declarative and component-based programming model that make it more efficient to develop the UI. And we also use the Single Page Application [3] which is also the JavaScript framework for our front-end design. Compare with Vue 3, Vue 2 is also a very popular framework in recent years. But we choose to use Vue 3 not only because it is the current and latest major version of Vue, but also for many reasons. First, Vue 3 contains new features that are not present in Vue 2, such as SFC Composition API Syntax Sugar [4], Suspense, and multiple root elements per template. For instance, we use <script setup> to implement the Composition API inside the Single-File Components (SFC) [5]. Another reason is Vue 3 provides smaller bundle sizes, better performance, better scalability, and better TypeScript / IDE support. In general, Vue 3 makes the UI development of our project easier and more efficient, enabling us to better fulfill the functional requirements.

<https://vuejs.org/guide/introduction.html>

<https://www.spiceworks.com/tech/devops/articles/what-is-single-page-application/>

<https://vuejs.org/api/sfc-script-setup.html>

Because we use Vue.js to develop our mobile online shopping application, so it is not a native mobile application. If we just use links to navigate to the new pages. Every time when users jump to a new page, it will be reloaded, which will greatly affect the user experience. In order to make users feel that it is more like a mobile application. We used Single-Page Application (SPA) which is a web application or website supported by Vue.js. We want it to rewrite the page with new content fetched from a web server as the user interacts with it instead of loading a new page for every interaction. Thus, to achieve this function, we used Vue Router. Vue Router is the official routing library for Vue.js, which is great for handling the routing in Single-Page Applications. This means our application is only loaded once from the server to the browser and the result is that the browser does not need to reload when routing between pages and gives the user a smooth navigation experience between different pages. Here is an example of the router link in our project:

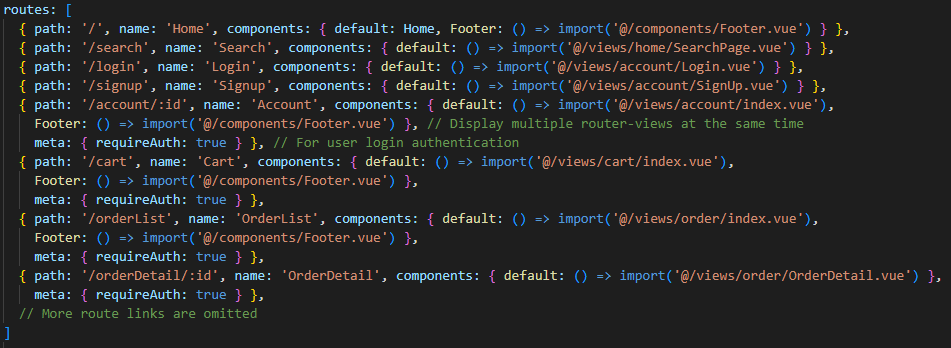


Figure 4-1-1-2: Router Links

<https://beginnersoftwaredeveloper.com/vue-router-why-use-it/>

In order to better manage component state, store and manage data obtained from the back-end, and provide users with a better experience, we adopted Vuex which is a state management pattern + library for Vue.js application. It serves as a centralized store for all the components and it has the ability to store and share reactive data across the app without trading off performance, testability or maintainability. If our system is not too complicated, we don’t need to use Vuex, but our web app involves multiple components that need to share data or need to transfer and synchronize data between different components. For example, from the order list page to the order detail page not only need to share order data (such as product and user data), but also need to synchronize the data between the two pages (such as order state). At this time Vuex is very useful. Because Vuex is a centralized store, any component in the application can access the data in Vuex, and the data in Vuex is reactive data, so if one component changes the value of a variable in Vuex, the value of the variable in other components will also change accordingly.

<https://vuex.vuejs.org/>

Figure 4-1-1-3 shows the structure of Vuex. In this structure diagram, if the components want to change the data, they can use the Dispatch method to call the Actions to process complex business logic or interact with the back-end. Then, the Actions will call the methods in the Mutations through the Commit method to change the State. Finally, the reactive data in the State will be rendered by components.

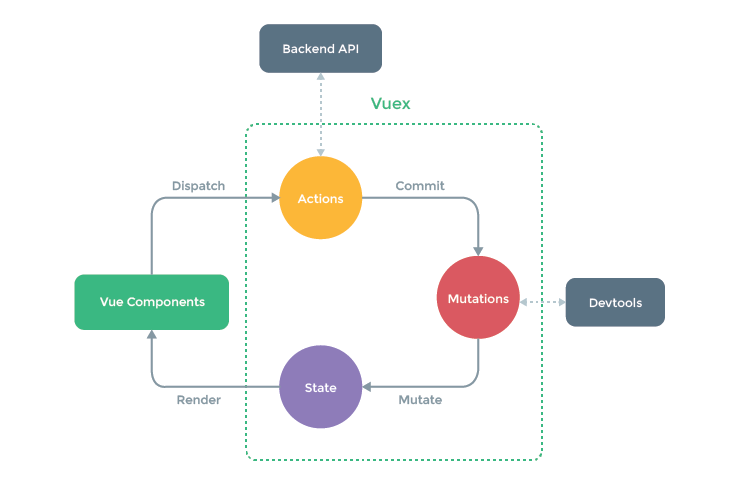


Figure 4-1-1-3: Vuex Structure

Figure 4-1-1-4 displays all the modules of the store in our system, such as User.js and Product.js. Each module can contain its own state, mutations, and actions.

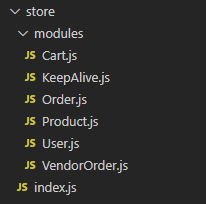


Figure 4-1-1-4: All the states in the store

Figure 4-1-1-5 exhibits a part of the User.js module. As you can see, the actions can get the user list from the back-end. After getting the response, it will call the mutations to change the “user” in the state.

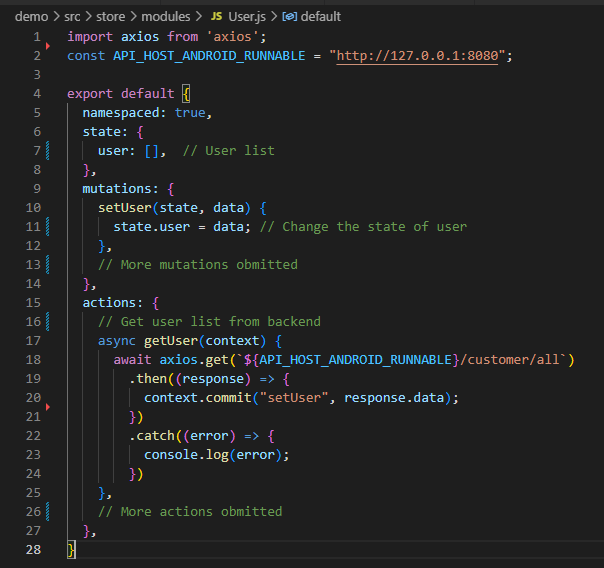


Figure 4-1-1-5: User module of the store

To connect to the back-end, we employed Axios in the front-end. Axios is a promise-based HTTP Client for node.js and the browser. Compare with Fetch, Axios is easier to use because of its built-in APIs. Axios also supports more functions such as built-in CSRF protection, cancelling requests, and request timeout.

<https://axios-http.com/docs/intro>

<https://inertiajs.com/csrf-protection>

For the UI design, we use Vant 4 which is a lightweight, customizable Vue UI library for mobile web apps. This component library is dedicated to providing mobile components for web apps. It provides more than 70 high-quality components, covering mainstream mobile scenarios, and has excellent performance. The average component size is less than 1KB. The most important thing is that we don't need to spend a lot of time writing very complicated CSS and it is very good for UI and UX design. Compare with Vant 4, there is also a popular web app component library called mint-ui. At first, we wanted to use it to help design the UI of our web app. but we found that it is actually a component library designed based on Vue 2, is not well compatible with Vue 3, and the maintenance and update speed of this component library is very slow, so we decided to use Vant 4.

For the file structure, we author Vue components using an HTML-like file format called Single-File Component (also known as \*.vue files, abbreviated as SFC). A Vue SFC, as the name suggests, encapsulates the component's logic (JavaScript), template (HTML), and styles (CSS) in a single file. Figure 4-1-1-1 is an example of SFC file format in our project.

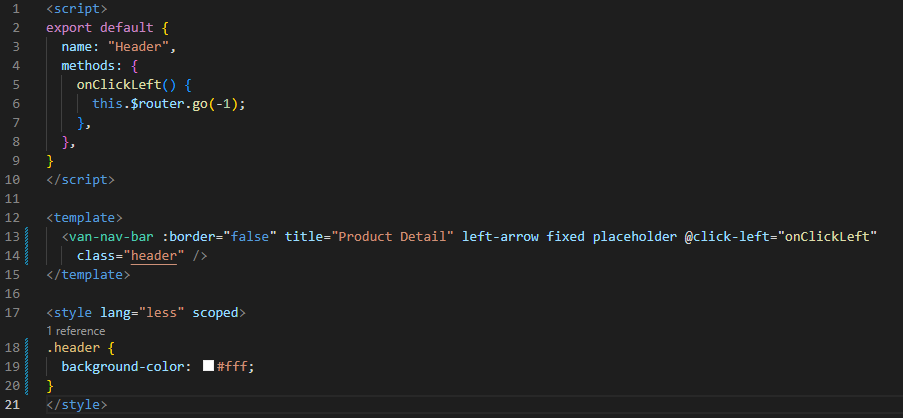


Figure 4-1-1-1: Single-File Components (SFC)

### **Back-end**

For backend, the implementation includes API server, Application server and MySQL server. After consideration, we choose Spring Boot as our API server, which is a popular, open source, enterprise-level framework based on Java[陈1]. For Application server, it is responsible for transmitting Vue. While MySQL server is mainly used for storing the related data of our mobile application.

<https://www.ibm.com/topics/java-spring-boot>

Spring Boot is an API server which is implemented with Java. It has high performance and is super popular among the developers, which means we could get more help and reference through resources on the Internet. Besides, it provides powerful tool named Spring Data JPA. It actually takes the advantage of the JPA specification, including the entity and association mappings, the entity lifecycle management, and JPA’s query capabilities. Besides, it has extra functions which do not require codes to implement with the repository pattern on a higher abstraction level. And it also provides the function which help generate MySQL queries automatically based on the method names.

MySQL server is a perfect choice to store all the related data to accompany with Spring Boot as API server. MySQL is an open-source relational database management system (RDBMS) and it has also been tested to be a "fast, stable and true multi-user, multi-threaded SQL database server", which means it has excellent performance. Since its first internal release on 23 May 1995, it has been developing for years. This means it has wide range of members applying it and we could get a variety of community support. Meanwhile, Spring Boot has already had many MySQL related tools and libraries. Developers could use MySQL more easily and quickly.

[https://en.wikipedia.org/wiki/MySQL](https://en.wikipedia.org/wiki/MySQL%20)

For Application server, it has the responsibility for sending static files to the client side. Here the static files are HTML files (Vue files).

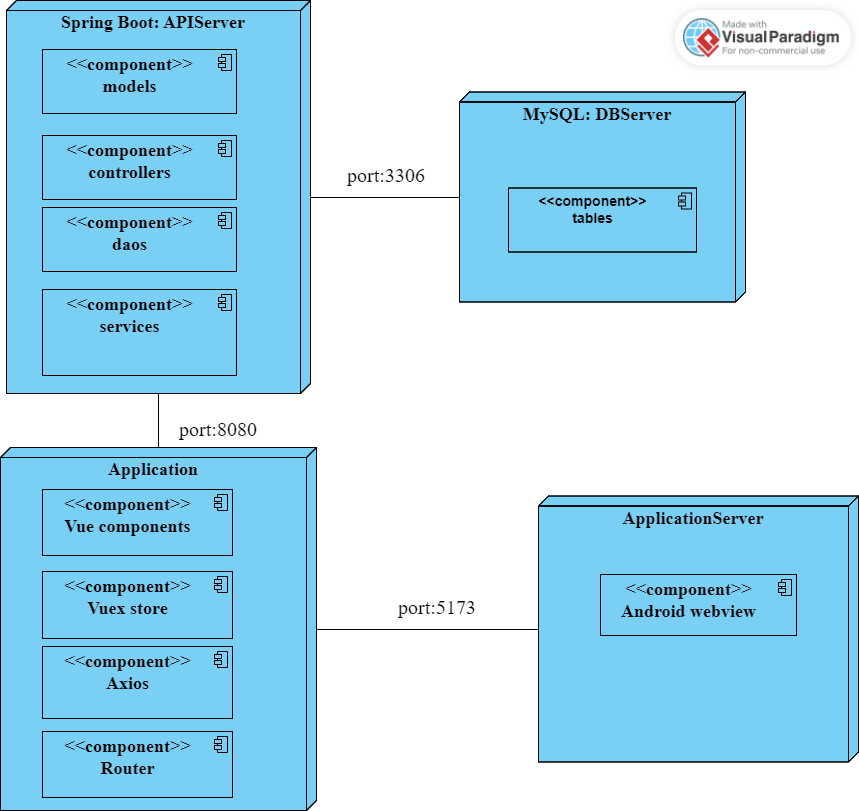


Figure 4-1-2-1: Development Diagram for Back-end

### **MVC Structure**

Our application adopts MVC structure. MVC consists of Model, View, Controller. Model layer represents data model and defines the storage of all the application’s data objects. Besides, it all contains the logic of data operation. In Niubility, Spring Boot JPA actually helps us deal with this layer. I will use Customer entity to illustrate. Fig. shows we simply create a class to define the specific entity and its related attributes. Spring Boot JPA will automatically generate corresponding tables in MySQL server. So how can we query the database server to get the desired data? Here Dao (Data Access Object) plays an important role, which could allow developers simply write the method name without requiring them to input the detailed query. And Dao actually provides the basic CRUD functions with some specific situations. Fig. X present the method in ShoppingCartDao which get the shopping cart records belonging to the specific customer.

View is the layer which is associated with User Interface. It provides the visual representation of MVC model. In other words, it displays the output to the user. In the application, Vue component actually in charge of this.

Controller takes care of request handler. It actually like a bridge between Model and View. It sends the request to get the data. After receiving desired data, it will send to the view layer. In our application, there are multiple controllers corresponding to various entities. If user want to browse the detail information of the specific product, he could visit the related url to get.

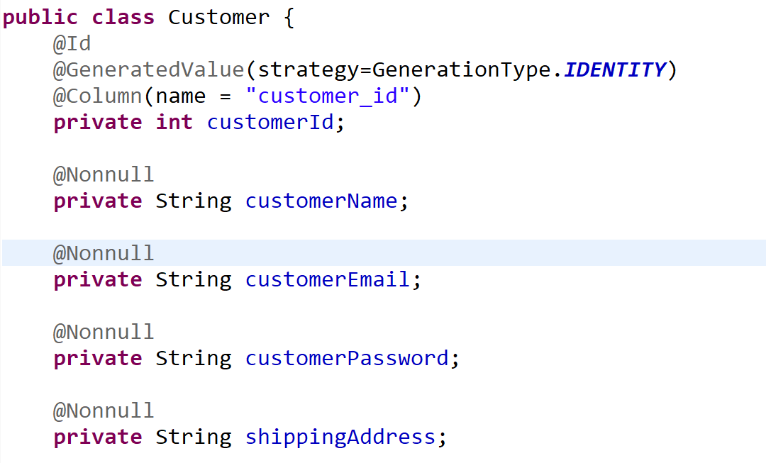


Figure 4-1-3-1:

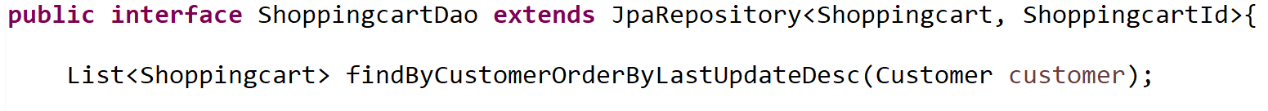


Figure 4-1-3-2:

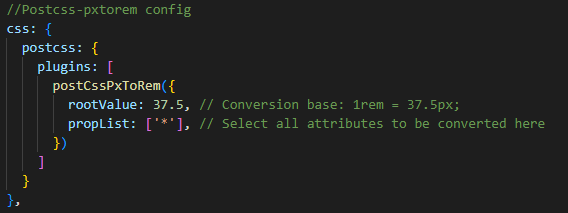
## REM Adaptation Solution for Mobile Screen

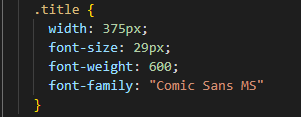
Nowadays, the brands of mobile phones are becoming more and more abundant, and the sizes of mobile phone screens are even more varied. Since our project is to develop a mobile web application, we must first consider the problem of displaying web content adaptively to the different screen sizes.

To solve this problem, we should first consider the unit that controls the page size. All lengths in the browser are in units of CSS pixels, which is what we often call "px", but px is absolute unit in CSS. Rem, on the other hand, is relative unit that is based on the document's root font size. So it can change with the root font size, thus realizing the adaptive function. To convert the CSS unit px to rem. We used Postcss-pxtorem which is a plugin for PostCSS that generates rem units from pixel units. Figure 4-2-1 shows how to configure postcss-pxtorem. In this file, we set the conversion base to 37.5 which is the “rootValue”. This means means that 1rem is equal to 37.5px. For example, in the second picture, I set the width of the title to 375px, and the width rendered in the browser is 10rem, just as shown in the third picture.

<https://github.com/cuth/postcss-pxtorem>

<https://github.com/amfe/lib-flexible>





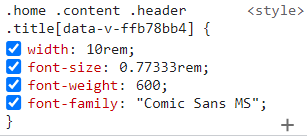
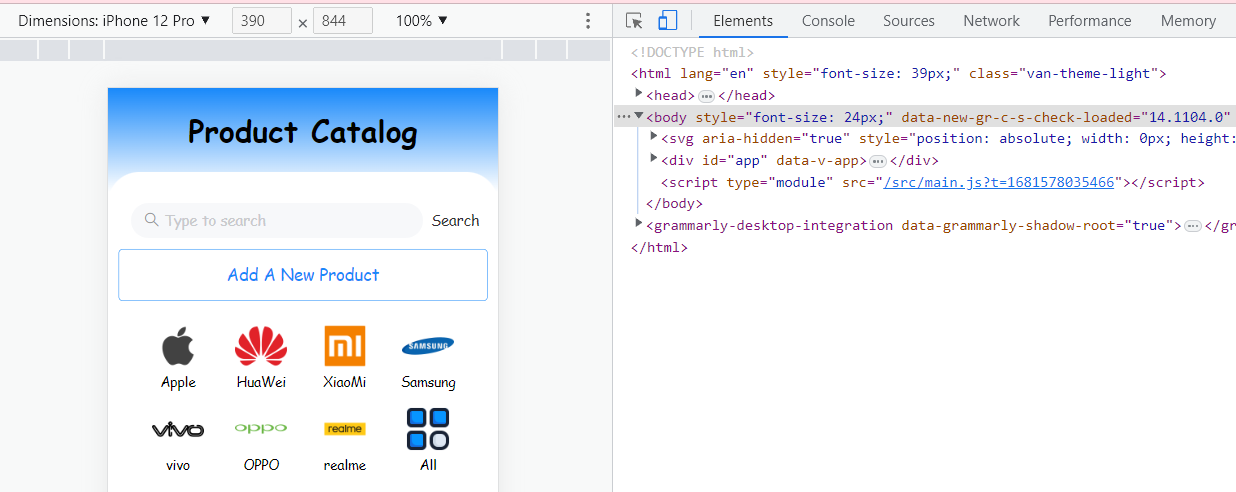
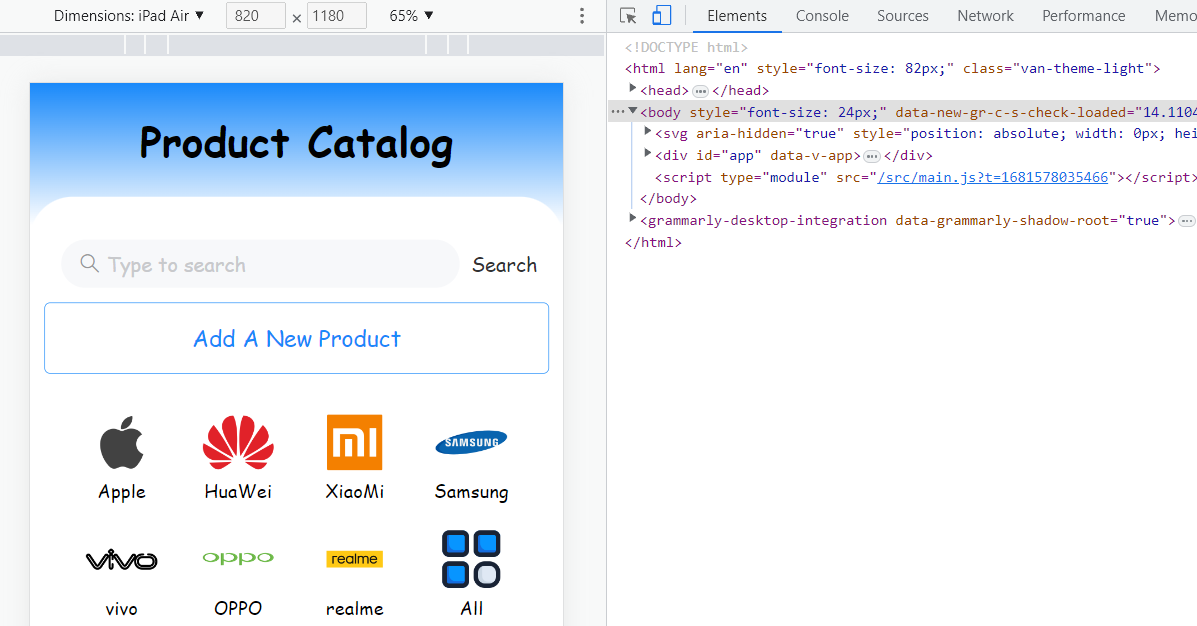


Figure 4-2-1: Postcss-pxtorem Configuration

If we just converted the CSS units, it would be the same as before changing the units. If we change the size of the screen, the size of the content of the web page will not change, which will cause the problem that the font is small even though the screen of the mobile phone is large. In order to solve this problem, we use postcss-pxtorem and amfe-flexible together to dynamically read the screen size of the user's mobile phone. As we mentioned before, postcss-pxtorem will convert px to rem, and amfe-flexible will set 1rem to viewWidth/10, which means that dynamically convert the font size to one-tenth of the screen width. In this way, the font size of our web app will change with the size of different mobile phone screens. The following two pictures show that in the case of different screens of different virtual machines. The font size will change with different screen sizes. For example, the font size in the iPhone 12 Pro is 39px, while the font size in the iPad Air The size is 82px.



Virtual Machine: iPhone 12 Pro



Virtual Machine: iPad Air

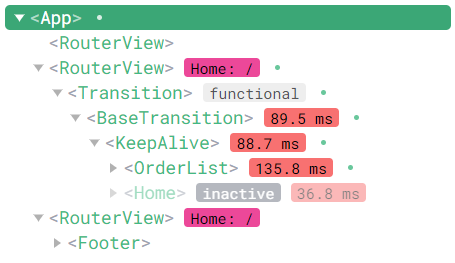
Figure 4-2-2: iPhone 12 Pro vs iPad Air

## <keep-alive> Cache Components Data

By default, an active component instance in Vue will be unmounted when switching away from it. This will cause any changed state it holds to be lost. When this component is displayed again, a new instance will be created with only the initial state. This means After the user jumps to other pages. The previous page will be destroyed, and when the user returns to the previous page again. All the operations performed by the user on the previous page will be refreshed.

Although this problem does not have a great impact on some pages, such as product detail page, order detail page, and account page. Because on these pages, there is not much data that users can manipulate. But for some pages, if the app refreshes the pages after the jumping, it will greatly affect the user experience. For instance, a user browses the product list on the home page. If the user opens the detailed page of a product on one of the product list pages. When the user returns to the home page again, the home page will reload and display the content of the first page.

To solve this problem, we adopted <keep-alive> in the front-end, which is a built-in component that allows us to conditionally cache component instances when dynamically switching between multiple components. When a component instance is removed from the DOM but is part of a component tree cached by <keep-alive>, it goes into an inactive state instead of being unmounted. When a component instance is inserted into the DOM as part of a cached tree, it will be activated. Figure 4-3-1 illustrates the two component that are inside the <keep-alive> component. This picture represents that the user is currently on the order list page, and the home page caches the data when the user left the page for the last time.



Inactive component

Activated component

Figure 4-3-1: Activated and inactive components

Figure 4-3-2 shows how to use <keep-alive> in our code. As you can see, <router-view> is used to render the component matched by a top-level route, which means that all main page components of our project will be rendered in it. In <keep-alive>, the "max" attribute limits the number of cached components. The purpose is to prevent the page from being stuck due to too much data cached by the browser. And the "include" attribute defines the components that need to be cached, such as home page, order list page, and search page.

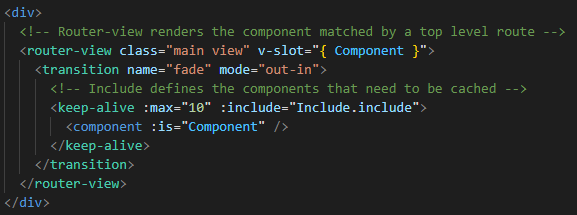


Figure 4-3-2: The use of <keep-alive>

Now, although the data of the page can be recorded and cached, <keep-alive> cannot cache the scrolling position of the page when it was left last time, so two <keep-alive> lifecycle hooks are used at this time. A <keep-alive> component can register lifecycle hooks for these two states using “activated” and “beforeRouteLeave” hooks. The “activated” hook function will be triggered every time the component is called on initial mount or it is re-inserted from the cache. On the other hand, the “beforeRouteLeave” hook function will be triggered whenever the component for the current location is about to be left. So we can record the scrolling position of the page every time before each component jumps, and record it in the “scrollTop” which will be cached in the data of the component. When returning to this page next time, it will directly read and position the page to this position. The code is shown below in Figure 4-3-3.

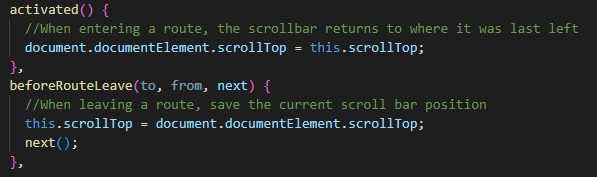
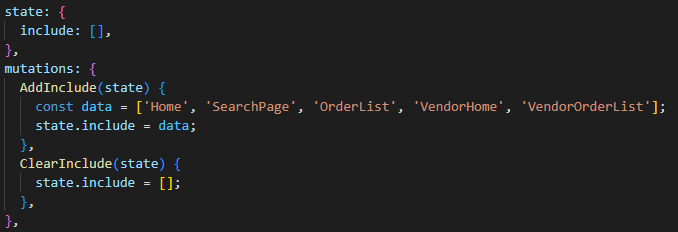


Figure 4-3-3: Recording scrollbar position

Now, although we have implemented the function of caching page data, but we will encounter another problem. Since <keep-alive> is used, the component will not be destroyed directly when leaving the page. So when users log out or switch accounts, the page still retains the data from the last time they left, which will confuses the users. So this time we need to clear the cache. But Vue.js only provides the method how to cache the data, it does not have a method like "clearKeepAliveCache" to clear the cache. So how to clear the cache becomes the new problem.

To solve this problem, we searched for information and found that the "include" attribute in <keep-alive> has a hidden feature. If we remove the component from the "include", the web app will not record and delete the cached data of the component, which achieves the purpose of clearing the cache. In order to take advantage of this feature, we used <keep-alive> in conjunction with the Vuex we mentioned earlier. We store all component names that need to be cached in the Vuex store, and whenever the user clicks the logout button, the "include" will be cleared. And whenever any component in "include" is mounted for the first time, the names of all components will be stored in the "include". The code is shown below in Figure 4-3-4.



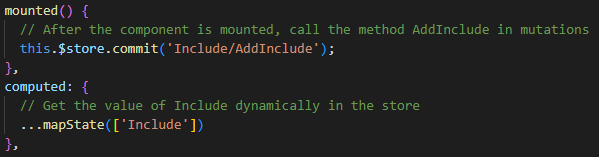


Figure 4-3-4: Implementation of “include”

## UI and UX Design

Nowadays, mobile phone applications are used in every aspect of our daily life, so user interface (UI) and user experience (UX) design is becoming more and more important. If an application is difficult to navigate or the layout of it is confusing, users will be gone in seconds. In this section, we will explain the UI or UX design used by our system.

### **Vant Component Library**

Vant 4 is a lightweight, customizable Vue UI library and it contains more than 70 Vue UI components for building mobile applications. The library is very lightweight with an average size of 1kb per component (min+gzip). This will greatly reduce the rendering time of the component and bring a good user experience to the customer. Vant also has the ability of theme customization, online theme preview tool, internationalization and so on. It can organize many different styles through rich CSS variables to suit many different types of customers. The styles of the various components of Vant also conform to the UI design of the mobile web app.

One example of the UI design of the Vant component library catering to the mobile user experience is "SwipeCell". As shown in the Figure 4-4-1-1 below, if the user wants to delete the item from the shopping cart, they can slide left and click the "Delete" button

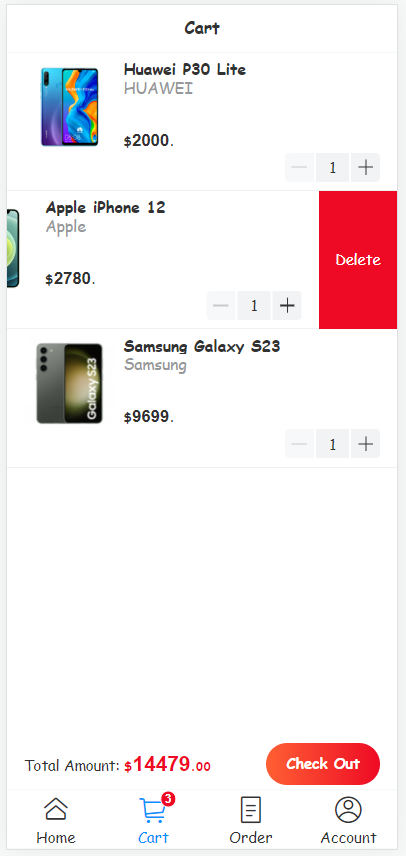


Figure 4-4-1-1: SwipeCell

### **Visual Hierarchy in UX**

Visual hierarchy is a clear visual hierarchy guides the eye to the most important elements on the page. It can be created through variations in color and contrast, scale, and grouping. The iOS design specification states that the current label should be highlighted. This is because when users shop on their mobile phones, basically all the information is obtained through their eyes. In the unconscious perception of visual depth, the user can judge the closest object to himself and know where he is. Therefore, in the design of the tab bar, the current tab usually needs to be highlighted, while other tabs should be appropriately weakened. In this way, the current page label and other labels can be visually separated from each other. This enables users to know their location more conveniently and intuitively.

Figure 4-4-2-1 displays the tab bar in our project, we not only use highlighting to show the current location of the user, but also add a badge to display the quantity in the upper right corner of the shopping cart icon.

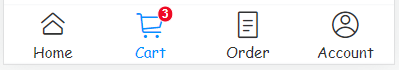


Figure 4-4-2-1: Tabbar

<https://www.nngroup.com/articles/visual-hierarchy-ux-definition/>

### **Gulf of Execution and Evaluation**

Gulf of execution is the degree of ease with which a user can understand the current state of a system. It is the difference between the intentions of the users and what the system allows them to do. For example, a person can easily determine the current state of the system (whether the light is on or off) and how to operate the switch by looking at a light switch.

Gulf of evaluation is the degree of ease with which a user can perceive and interpret whether or not the action they performed was successful. This gulf is small when the system provides information about its state in a form that is easy to receive, interpret, and matches the way the person thinks of the system.

In our project, we use blue icons with thumbs up to represent "Like", and conversely, red icons with thumbs down to represent "Dislike". At first, the user is in an unselected state. If he likes the item, he can click the blue thumbs up icon. After clicking, a green message prompt with "Like It!" will appear at the top of the screen, and the icon will change to a red icon with a thumbs down. At this time, it means that the user likes the product. If the user taps the red icon again, a red message with "Don't Like It!" will appear at the top, and it will change back to a blue icon. At this time, it is a state of dislike. As shown in the following two figures.

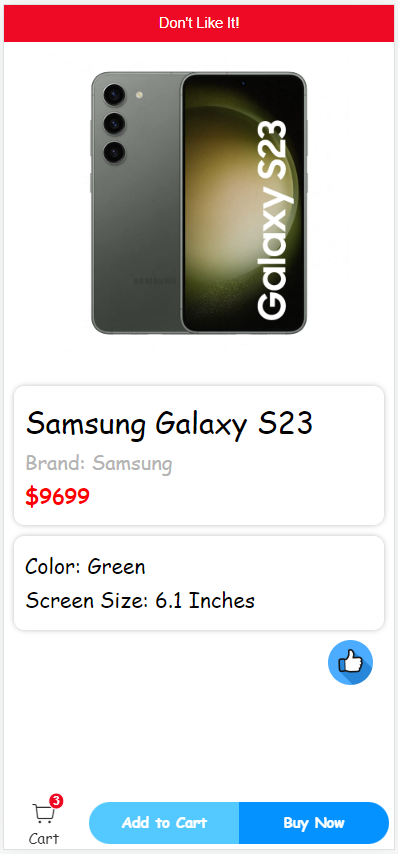
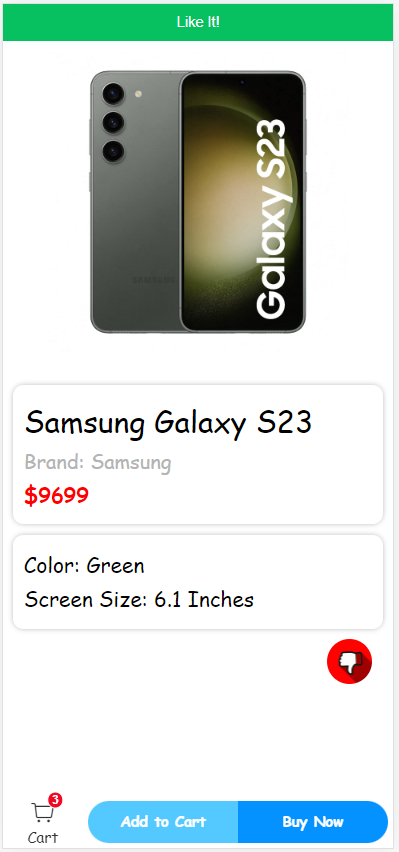


Figure 4-3-3-1: Like and Dislike

Therefore, our design can not only represent the user's status at this time through vivid icons, but also let users know what their operations mean and whether they are successful or not through message feedback in different colors.

## Product Search and Detail Display

## Image Storage and Handling

## Password Security

## Purchase Order Processing

## Customers’ Ratings and Reviews

## Concurrency Control

**Why We Choose Spring Boot for the Backend**

In school, we have learned two different programming languages: Python and Java. And we even study Django as backend framework which is based on Python. After looking through the Internet, we find out Spring Boot is also a good choice for us since it is a server-side Java framework.

Before implementation, we make comparison between these two frameworks in order to choose the most suitable one. Frist of all, in terms of performance, we found Spring Boot is faster in running the code as it is written in java comparing to Django. Since Google recommends fast websites and it also affects a website’s ranking, this is one of the most crucial factors we choose Spring Boot.

Secondly, we found that Django could only handle one request at once. However, Spring Boot is able to handle multiple requests at once. Using Spring Boot as backend will shorten the waiting time of users. In other words, it will promote the brilliant experience of users.

What’s more, as we know, Django and Spring Boot are both open-source technologies. According to statistics, Spring Boot has 25.8K GitHub forks and 39.8K GitHub stars, while Django has 42,000 ratings and 18,000 forks. From above, Spring Boot seems more popular than Django, which means we could get more reference and help benefiting from this. (Moreover, the responsible person is more familiar with Java.)

So, these are the reasons why we choose Spring Boot as the framework of our backend.

Spring Boot is ideal for building web applications due to its streamlined setup and auto-configuration features, which simplify the development process and enable developers to quickly build and deploy scalable and maintainable web applications.

For microservices, Spring Boot's modular architecture and lightweight design make it an excellent choice. It allows for the creation of independent microservices that can be easily scaled up or down to meet changing demands.

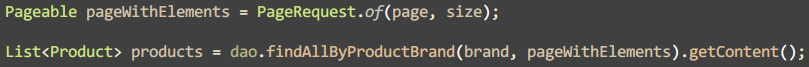
Finally, Spring Boot is also commonly used for batch processing, which involves processing large amounts of data at once. Its support for multiple data sources, job scheduling, and retry capabilities make it an ideal tool for this type of application.

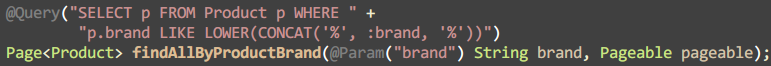
**Pagination**

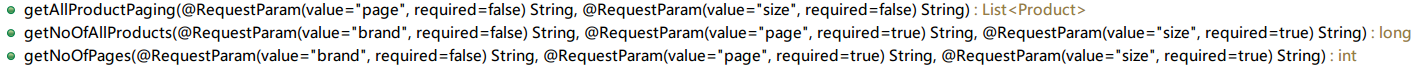
Pagination, also known as paging, is the process of dividing a web content into discrete pages. According to the requirement specification, products are displayed in multiple pages. Figure X and Figure Y are demonstrations.

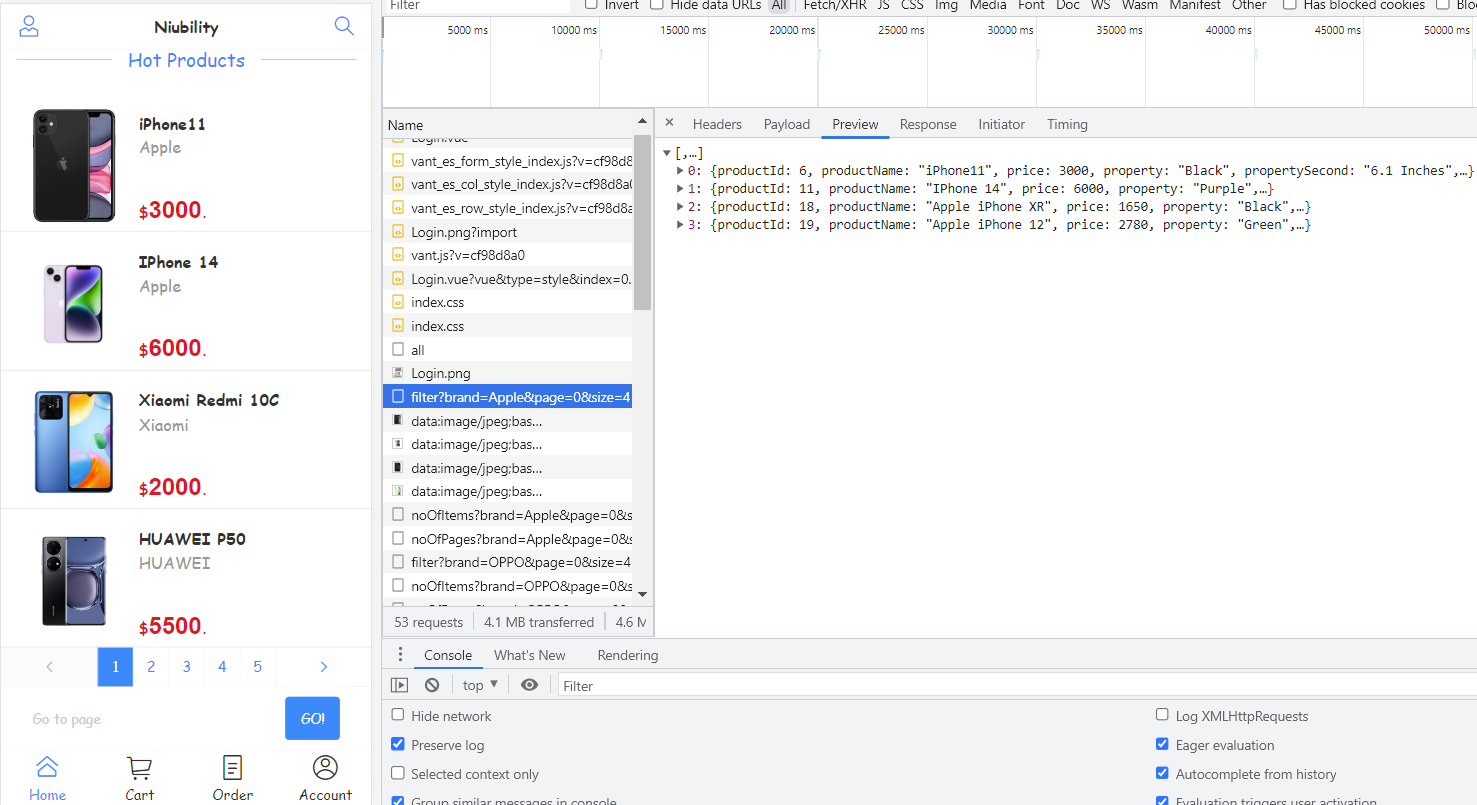
There are mainly two approaches of implementing pagination: client-side pagination and server-side pagination. They are simply slicing the whole product list either on the server or on the client. Our solution is the hybrid of both server-side and client-side pagination in the homepage and in the search page.

In the homepage, we use server-side pagination. Because the mass products may consume much bandwidth. Only few items are displayed at once in server-side pagination mode, and as the change of pages, the data of products are retrieved on demand. Figure XXX and YYY illustrates how paging functionality is utilized provided by JPA Repository and Spring framework.







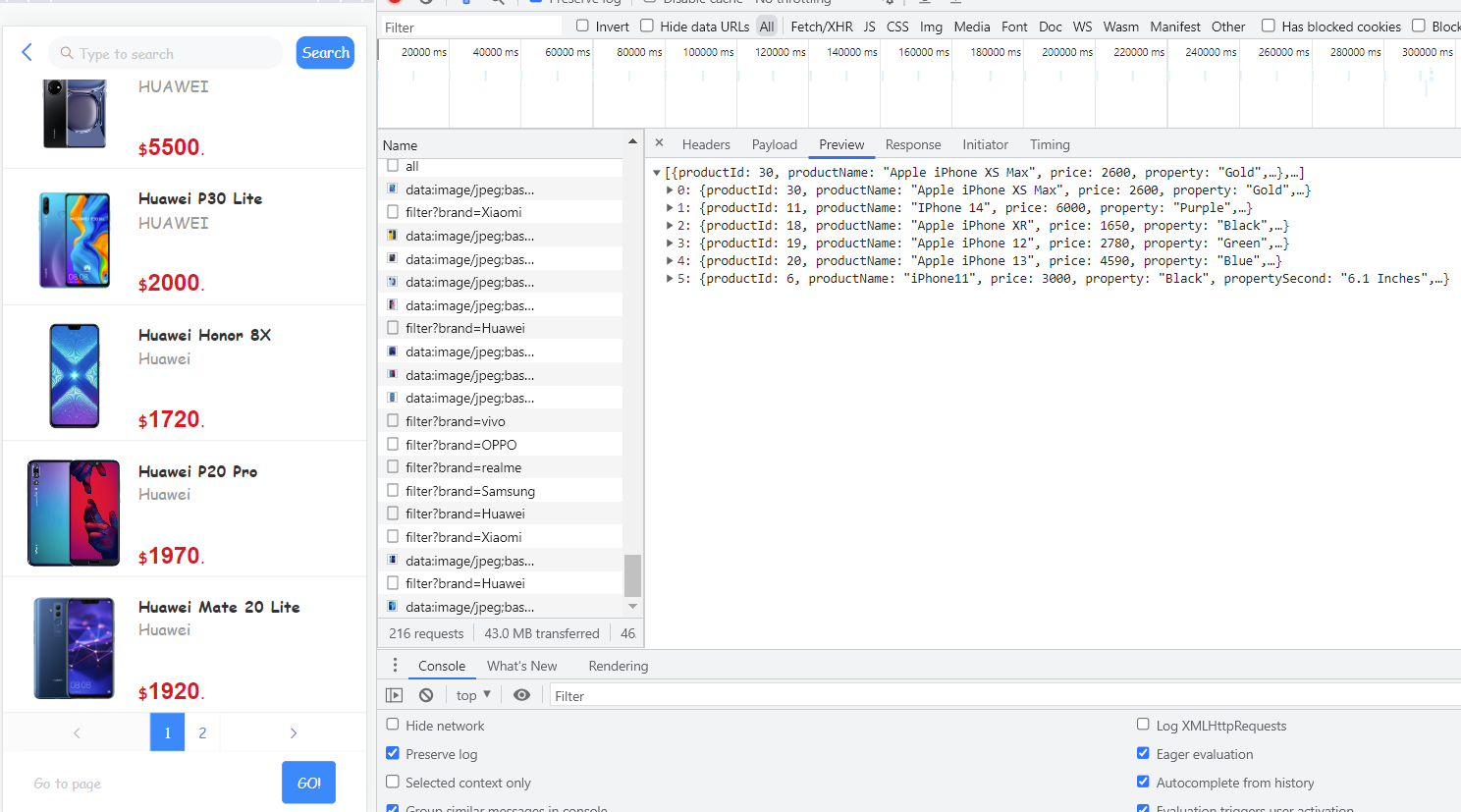


Page 1

Fetch data on demand

In the search page, we assume much smaller cardinality of data set is returned. Another assumption is switching speed among is highly required when the user is searching and filtering by brand simultaneously. Because the user tends to go through all products fast to look for the product that the user wants. Therefore, we decide to apply client-side pagination in the search page. The entire list of searching results is returned first, the paging and filtering are done by client-side program subsequently. It is believed that using client-side pagination here increases the responsiveness of the search page when the user is quickly switches among pages. In Figure XXX, products are sliced just before rendering.





Page 1

Client-side paging

The remaining of product list

is also retrieved

**CORS**

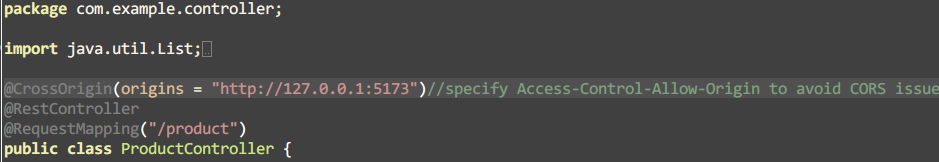
*Cross-origin resource sharing* (CORS) is a mechanism which restricts access of resources sent by a different domain. According to the *same-origin security* policy forbids cross-origin access, for instance, Ajax call, to another origin to resources.

It is where the problem emerged. In our architecture design (refer to Figure X: deployment diagram), our web server (running on port 5173) and API server (running on port 8080) are separate as loosely coupled modules. It allows us to develop View-controller and Model-controller concurrently and conveniently. In our development environment, they are running in different ports. However, the cross-origin Ajax are forbidden so that our front-end JavaScript cannot access the API.

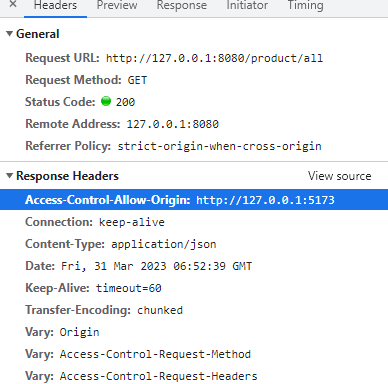


*Figure 1. without adding the control header*

To solve the problem, we utilized an annotation provided by Spring framework. It adds Access-Control-Allow-Origin HTTP header to HTTP responses requested by our web server.



*Figure 2. @CrossOrigin Annotation*

****

*Figure 3. Access-Control-Allow-Origin Header*

After configuring the API server, we successfully overcome this technical problem.

**Search**

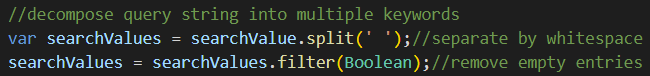
In consideration of user experience, the search function of the App should be efficient, accurate, and highly available.

Automatic Submission

Although typing one more word will send another request, which consumes a lot more data traffic than sending the request till the user tabs “Search” button, we still want to keep the function due to high performance and availability. We reckon when the user is searching instead of browsing, there must be some intention or aim. Therefore, the sooner the product is found or result is displayed, the sooner requirements or intention is satisfied or resolved. Such functionality greatly reduces the probability of causing anxiety when a customer cannot find a certain item in a short period of time.

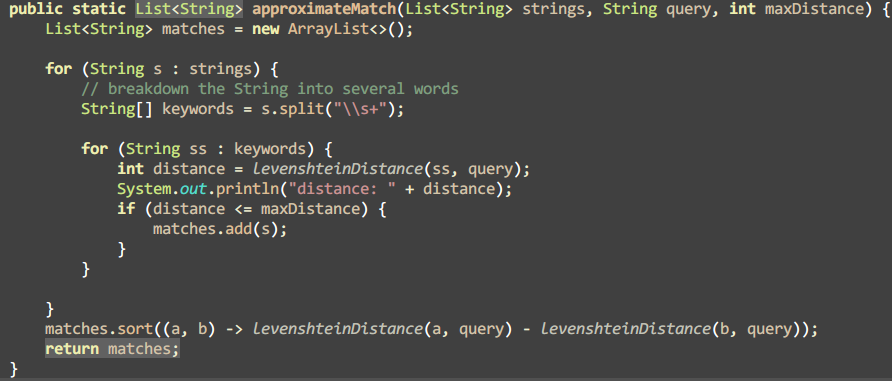
Keyword Processing

At the frontend, the input text is decomposed for searching in finer granularity.

****

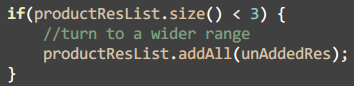
*Figure 1. Split the User Input Text*

At the backend, product names are decomposed for more exact matching through searching keywords.

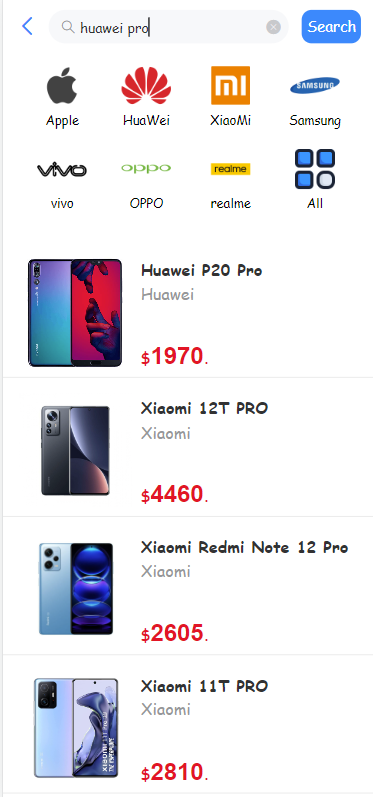


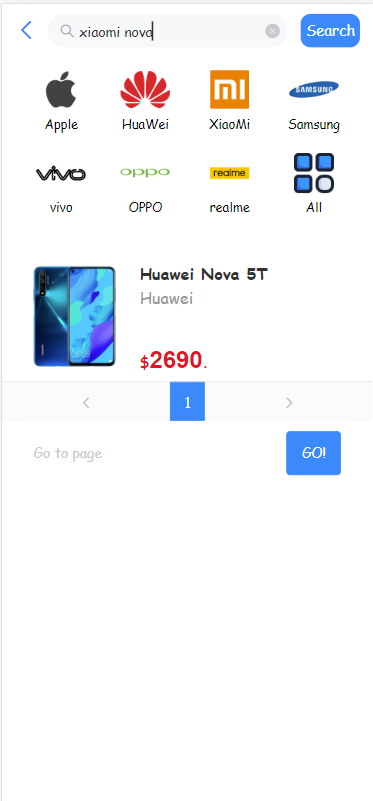
*Figure 2. Approximate Match Function*

In addition, we have eased the condition from satisfying all keywords to satisfying either one of them to ensure availability to a certain extent.



*Figure 3. Extend Result List when There are Few*



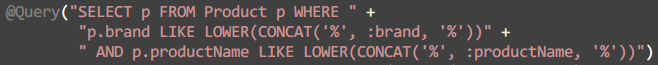


*Figure X. Complementary Results*

For instance, in Figure X. If the user types “huawei pro”. In our test dataset, there is only 1 product that simultaneously match “huawei” and “pro”. Besides, “Pro” model of “Xiaomi” also appears as the complementary of the result. Another example demonstrated in Figure X. Because there are only 1 result that match both 2 keywords (“huawei”, and “pro”), extra results are also returned even if they only match the keyword “pro”. Users are offered with flexibility so that they are encouraged to try to input everything they tend to obtain – even if there is no “Nova” model of brand “Xiaomi”.

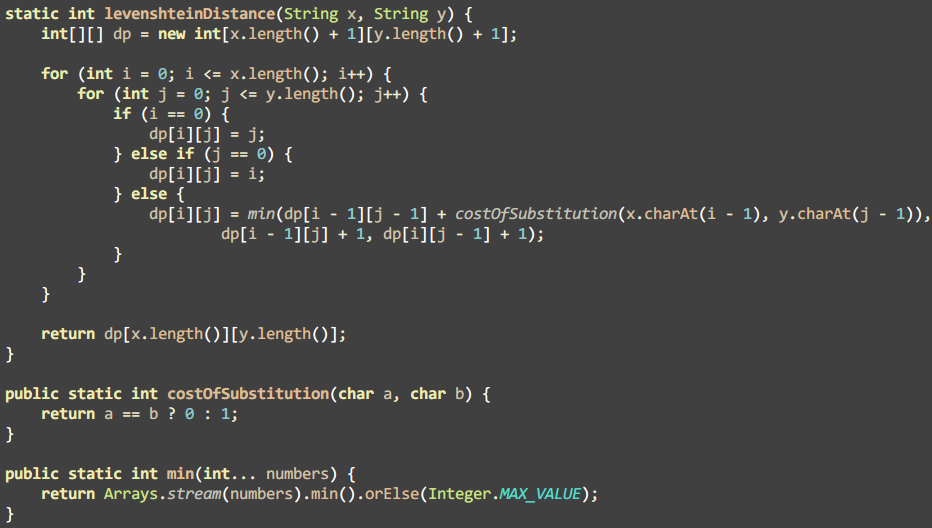
Fuzzy Match

Initially, our search function is simply implemented with a SQL statement to fetch matching records from MySQL database with the help of Query function provided by JPA Repository. The LIKE keyword and LOWER keyword enable searching by product name through a substring of it case-insensitively. For example, after the user typing “hua”, “Huawei P50” will be one record of the searching results provided to the user.



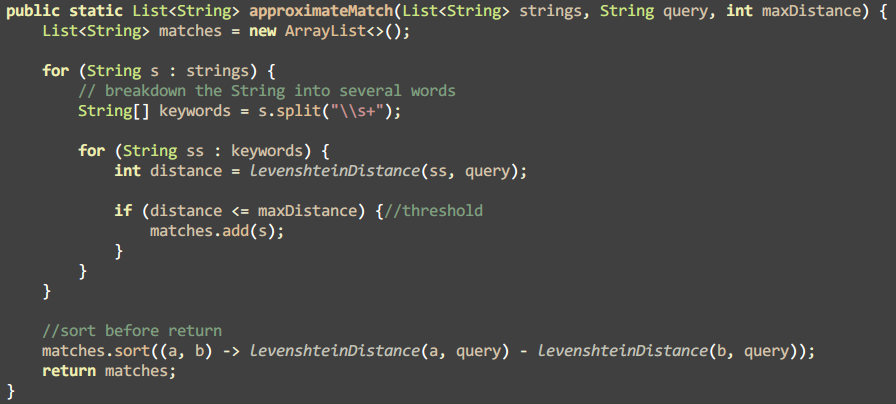
*Figure 1. @Query Annotation*

However, it’s a common situation that users sometimes type a misspelt word unconsciously. In order to resolve the requirements, we have developed a simple fuzzy match algorithm. Similar to Hamming distance, Levenshtein Distance is exploited for evaluating the similarity between two strings. In other word, it is the minimum number of single-character edits (insertions, deletions or substitutions) required to change one word into the other. The code of calculating Levenshtein Distance between two strings are showed in Figure X.

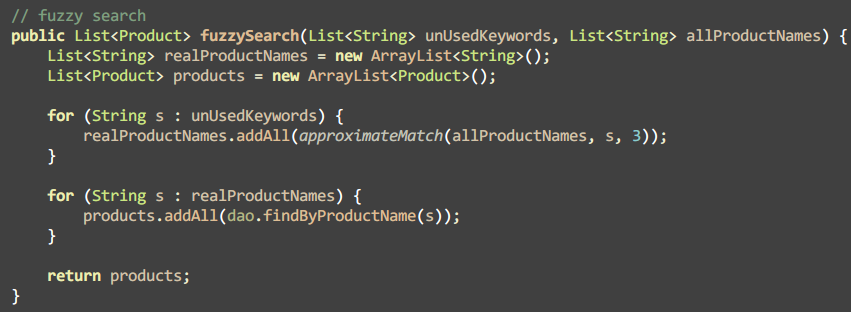


*Figure 2. Levenshtein Distance Algorithm*

The fuzzy match function is implemented by restricting the maximum Levenshtein Distance (we set it to 3) as the threshold, and return the results in ascending order of Levenshtein Distance between keywords that the user has input and keywords from product names. Finally, we successfully implemented a memory-based fuzzy search function.

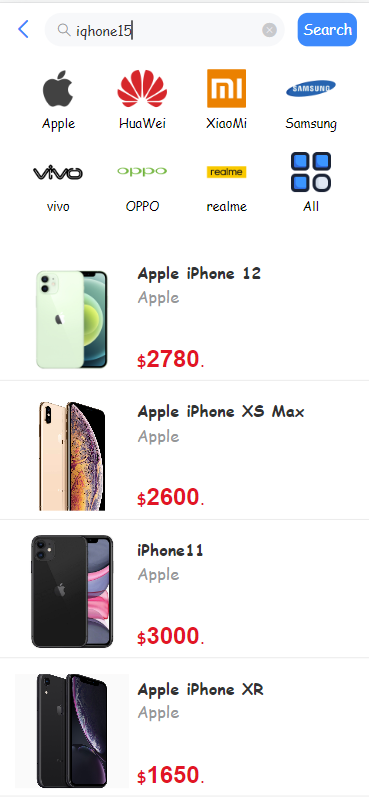


*Figure 2. Approximate Match Function*



*Figure 2. Approximate Match Function*

Because the fuzzy search algorithm is at least *O(n²)*, we try to reduce the number of accesses. The exact match function is applied at first for a keyword decomposed from user input. If there is no results found, the keyword will be added to an auxilliary array. After adding all the results retrieved from exact match function, fuzzy match will be applied accordingly for the unused keywords. By this mean, the correct or partially correct keywords (substrings of correct product name) are not fuzzily searched, thus the performance of our search function has increased.



*Figure X.*

It is illustrated in Figure X that “iqhone15” is a typical example of misspelt keyword and an advent of non-existing model (by far, the latest version of iPhone is iPhone 14). The outcome of returning all products related to iPhone satisfies the fuzzy match requirement.

**Personalized Recommendation**

Recommendation algorithm is a crucial functionality in nowadays e-commerce Apps. It is stated that 63% of smartphone users are more likely to purchase from mobile apps offer them relevant recommendations. We have applied a simple recommendation algorithm to generate 3 products that a customer may favor.

A like or dislike is a direct index which determines whether a customer likes a product or not. However, we still want to estimate how likely a customer would prefer a product without existing feedback from that customer. We are inclined to provide several recommendations to the customer … So that increase the likelihood for the customer to purchase a product.

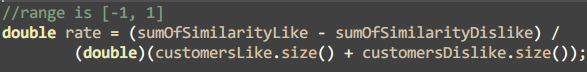
The algorithm is based on Jaccard index. Formula 1 calculates similarity between 2 users given the products they like (L1, L2) and dislike (D1, D2). Formula 2 derives the probability of a customer preferring a product from the similarity of the user and the users who have rated that product. For in-depth explanation for the algorithm, please refer to [Link]. In brief, it is assumed that a customer may like the product which is favored by similar customers.

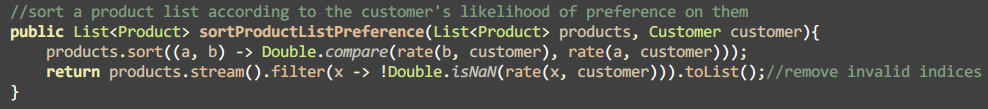
similarity index of two users

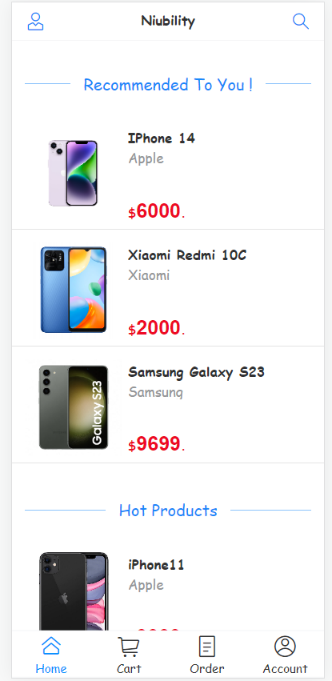
recommendation engine algorithm

*Figure 2. Approximate Match Function*

A part of the implementation in Java is demonstrated in Figure 1. In development, we noticed that if a product has never been liked or disliked by any user, the denominator will be zero, which contributes a *NaN* in Java. It is filtered out (Figure 2) in our consideration for valid and reasonable recommendation.





*Figure 2. Approximate Match Function*

Based on that, we push the top 3 products that are most likely to be favored to a customer and display them at the top of product list in the homepage.

*Figure 2. Approximate Match Function*

To evaluate the effectiveness of the recommendation algorithm, we have conducted a test case. For the details, please refer to [Chapter 5.2].

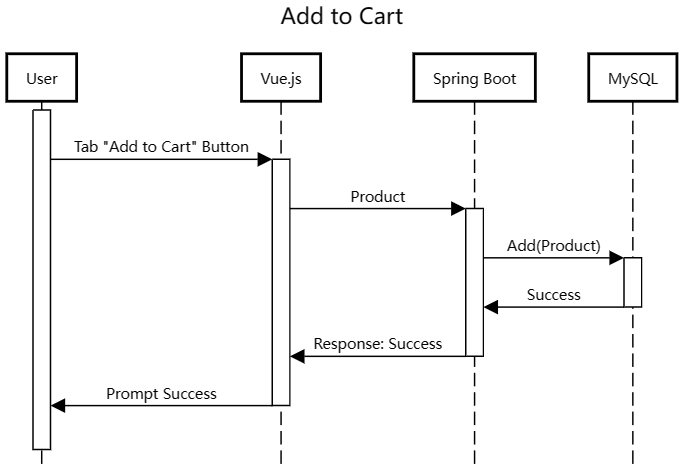
<https://developer.mozilla.org/en-US/docs/Glossary/CORS>

<https://spring.io/guides/gs/rest-service-cors/>

[Smartphone mobile app & site purchase data - Think with Google](https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/smartphone-mobile-app-and-site-purchase-data/)

[**https://www.toptal.com/algorithms/predicting-likes-inside-a-simple-recommendation-engine**](https://www.toptal.com/algorithms/predicting-likes-inside-a-simple-recommendation-engine)

<https://www.baeldung.com/java-levenshtein-distance#:~:text=What%20Is%20the%20Levenshtein%20Distance,to%20transform%20x%20into%20y>.

****