# 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/>

## 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.

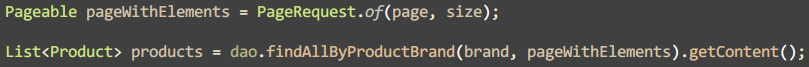
Finally, Spring Boot is also commonly used for batch processing, which involves processing large amounts of data at once. Its support for multiple data sources, job scheduling, and retry capabilities make it an ideal tool for this type of application.

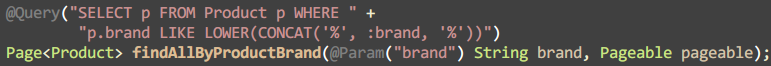
**Pagination**

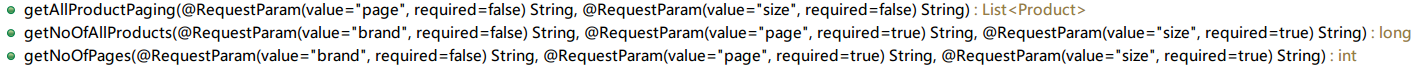
Pagination, also known as paging, is the process of dividing a web content into discrete pages. According to the requirement specification, products are displayed in multiple pages. Figure X and Figure Y are demonstrations.

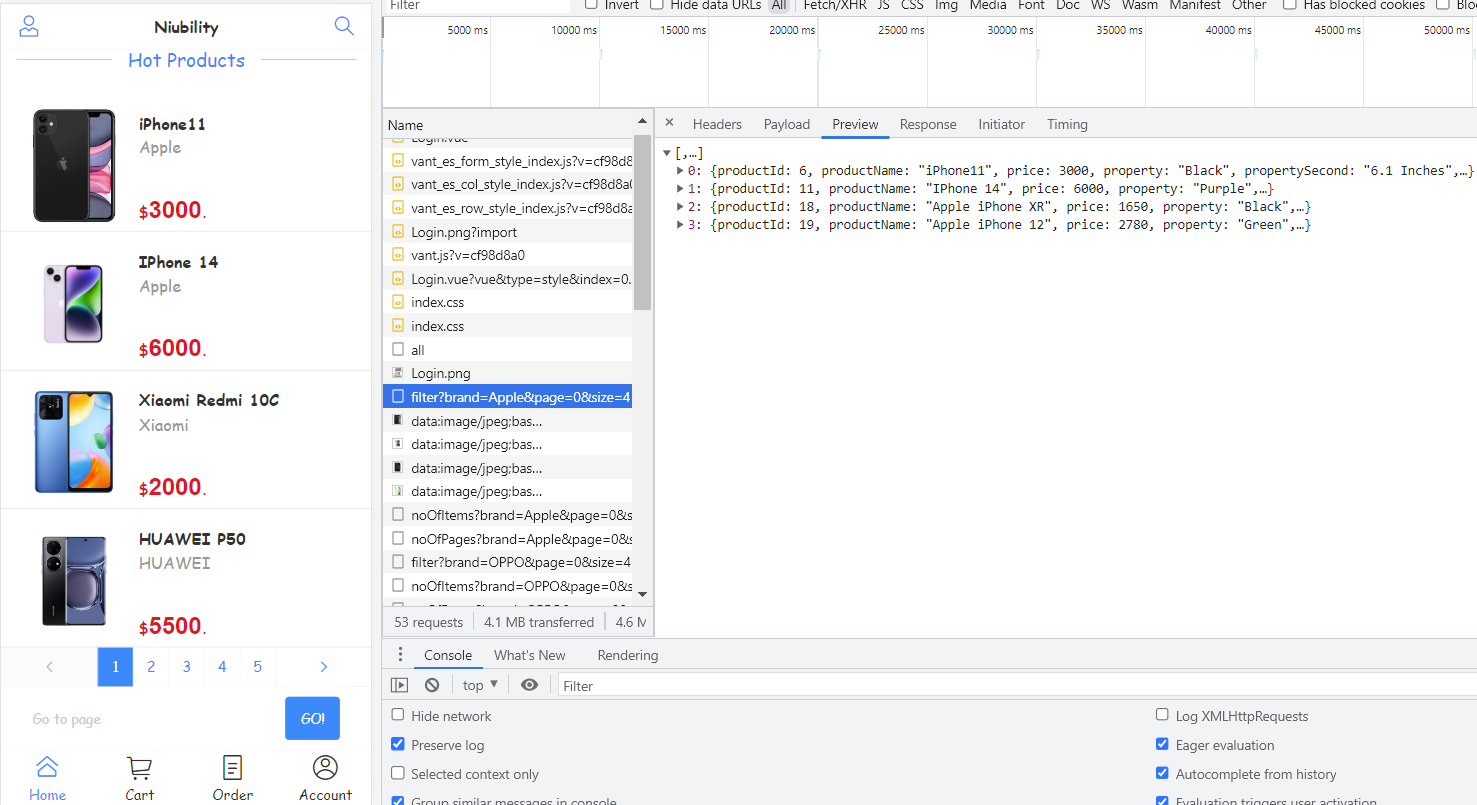
There are mainly two approaches of implementing pagination: client-side pagination and server-side pagination. They are simply slicing the whole product list either on the server or on the client. Our solution is the hybrid of both server-side and client-side pagination in the homepage and in the search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







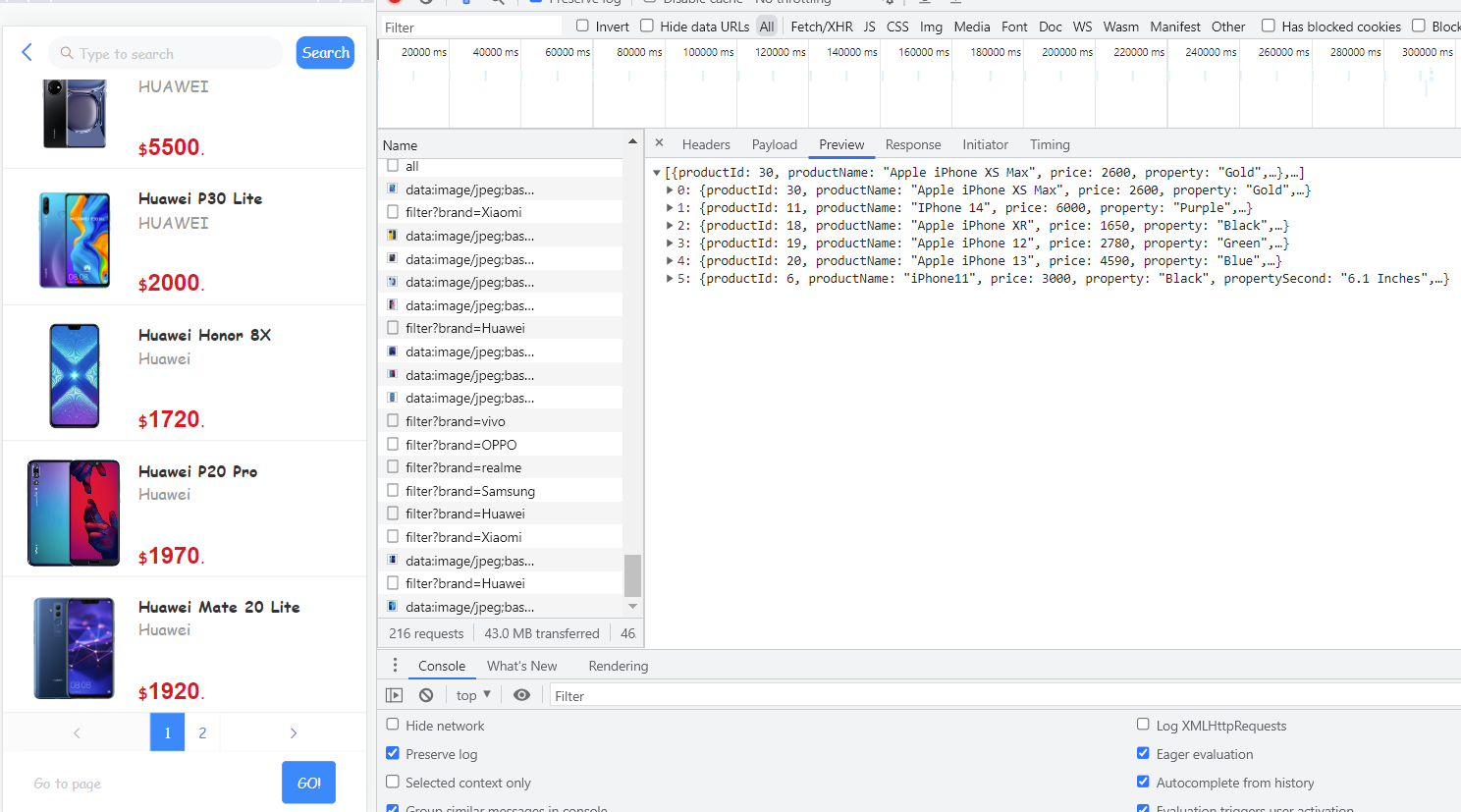


Page 1

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**

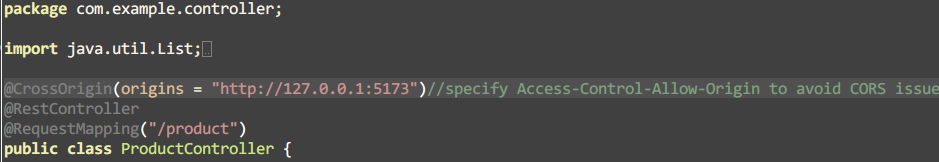
*Cross-origin resource sharing* (CORS) is a mechanism which restricts access of resources sent by a different domain. According to the *same-origin security* policy forbids cross-origin access, for instance, Ajax call, to another origin to resources.

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.

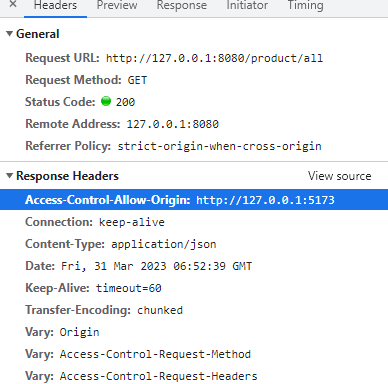


*Figure 1. without adding the control header*

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**

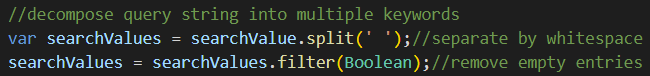
In consideration of user experience, the search function of the App should be efficient, accurate, and highly available.

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.

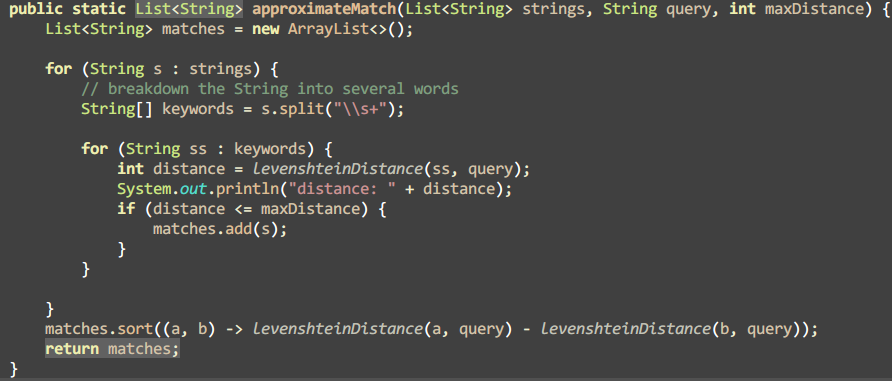
Keyword Processing

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

****

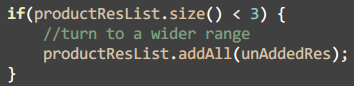
*Figure 1. Split the User Input Text*

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

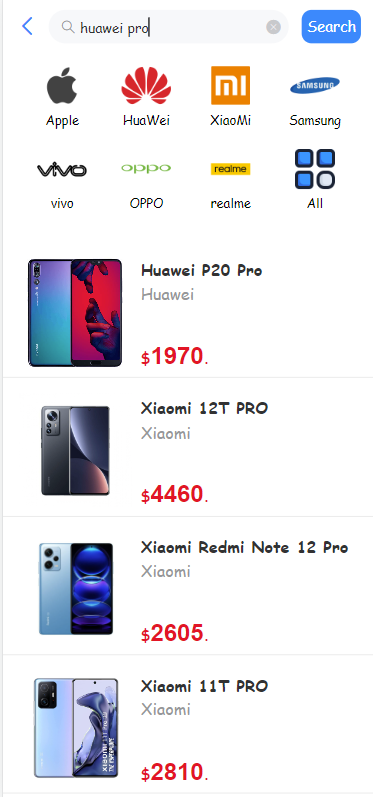


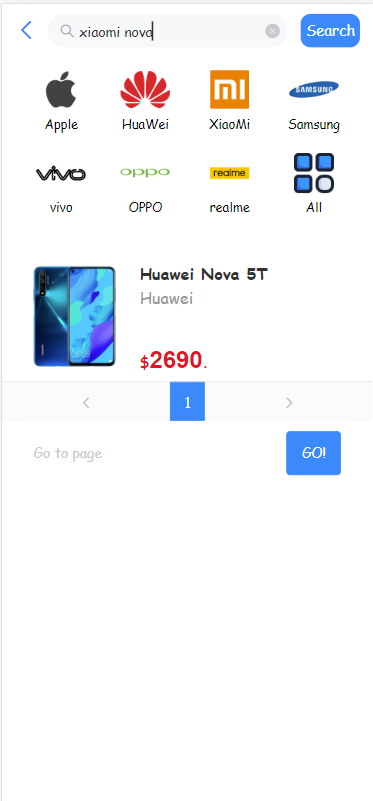
*Figure 2. Approximate Match Function*

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*



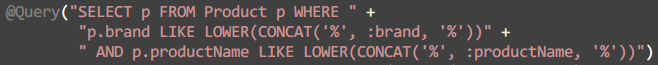


*Figure X. Complementary Results*

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”.

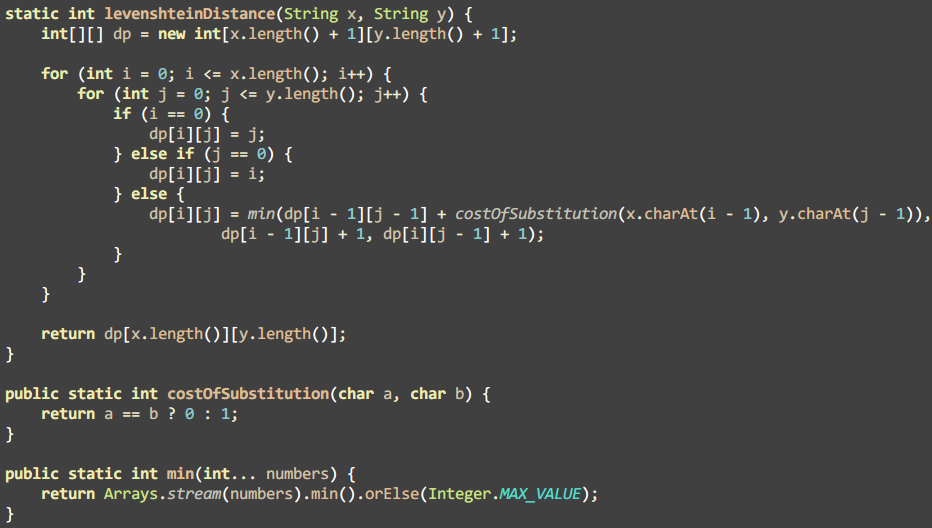
Fuzzy Match

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.



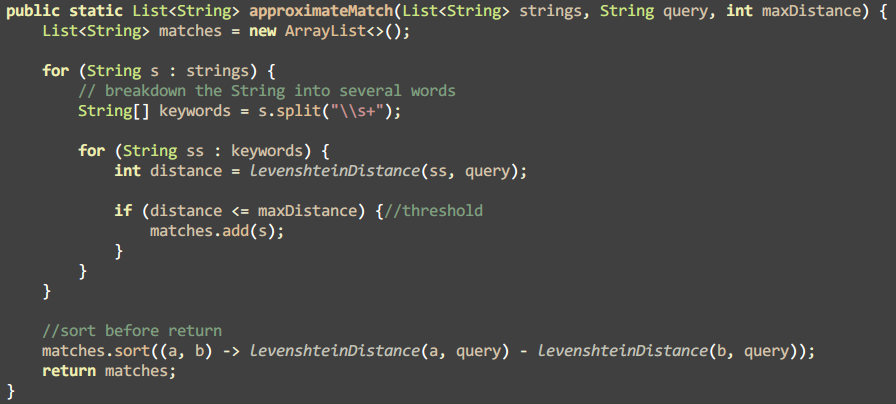
*Figure 1. @Query Annotation*

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.

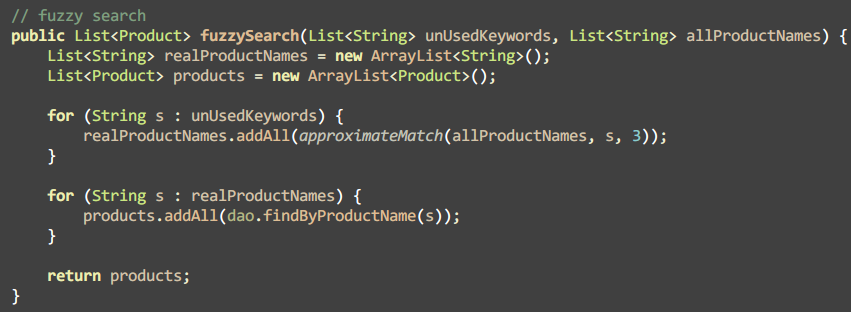


*Figure 2. Levenshtein Distance Algorithm*

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.

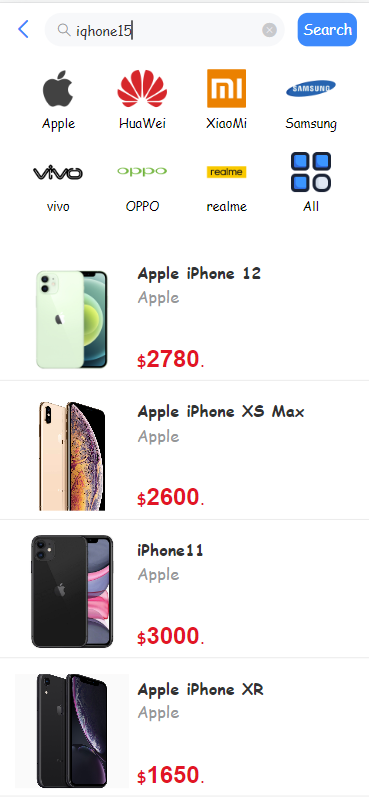


*Figure 2. Approximate Match Function*



*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.

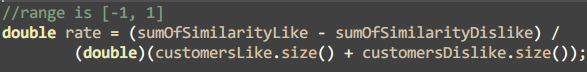
The algorithm is based on Jaccard index. Formula 1 calculates similarity between 2 users given the products they like (L1, L2) and dislike (D1, D2). Formula 2 derives the probability of a customer preferring a product from the similarity of the user and the users who have rated that product. For in-depth 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.

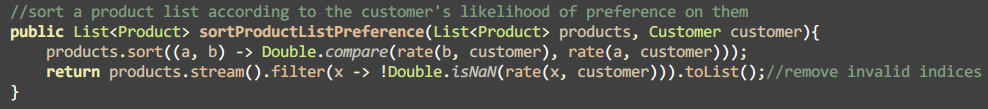
similarity index of two users

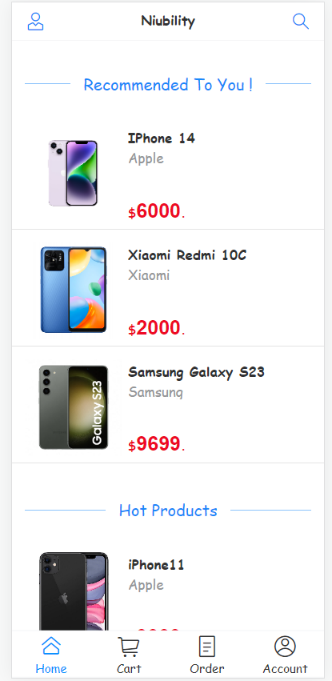
recommendation engine algorithm

*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].

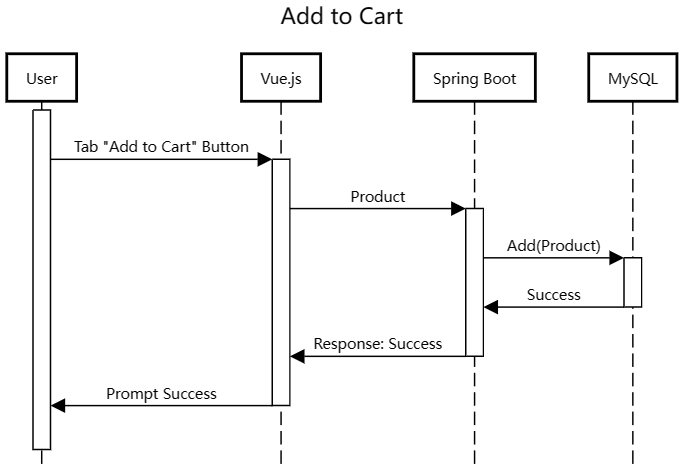
<https://developer.mozilla.org/en-US/docs/Glossary/CORS>

<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>.

****