ZzangBaguni

박준영[1], 신근호[1], 송은기[2], and 이상협[2]

[1]성균관대학교 수학과
[2]성균관대학교 시스템경영공학과

Sungkyunkwan University 캡스톤설계프로젝트 SWE3028

Abstract. 우리 생활에서 반드시 필요한 행위인 장보기는 외식을 하는 빈도수가 줄어들고 직접 장을 봐 집에서 차려 먹는 가구가 늘어나고 있는 추세인 지금 그 필요성이 더 높아졌다. 그러나 방문하는 마트마다 모두 가격이 달라 소비자는 어떤 마트를 선택해야 자신이 가장 저렴하게 장을 볼 수 있는지 비교하기 어렵다. 따라서 우리는 사용자가 선택한 위치 주변의 마트들에서 장을 봤을 때의 예상 가격을 표시하고, 사고자하는 식자재를 전부 한 곳에서 살 때 어디가 가장 싼지 알려주는 식자재 통합 가격 비교 서비스 짱바구니를 개발할 것이다. 짱바구니는 사용자들이 좀 더 저렴한 가격으로 식자재를 구매할 수 있게 도와줄 것이다.

Keywords: 생필품 · 가격비교 · 지도

1 Introduction

요리를 하기 위해 식자재를 구입하거나, 생활하는데 필요한 다양한 제품들을 구매하는 등 장보기는 우리 생활에서 반드시 필요한 행위이다. 물론 언제나 집에서 요리를 하기에는 번거롭기도 하고 더 맛있는 음식을 먹고 좋은 분위기의 식당에서 식사를 하기 위해 외식을 하기도 한다. 하지만, 코로나 19의 영향과 물가 상승의 영향으로 인해 외식을 하는 빈도 수가 줄어들고 직접 장을 봐 집에서 차려 먹는 가구가 늘어나고 있는 추세이다.

누구나 마트를 가보고 제품을 구입해본 경험은 있을 것이다. 소비자가 제품을 구입하기 위해 어느 마트를 선택할 것인지에 대해 영향을 주는 요인은 다양하겠지만, 이 프로젝트에서는 가격에 주목했다. 같은 프랜차이즈 마트가 아니라면 방문하는 마트마다 모두 가격이 달라 소비자는 어떤 마트를 선택해야 자신이 가장 저렴하게 장을 볼 수 있는지 알 수 없다.

물론 이런 불편함을 해결하기 위해 여러 서비스들이 존재했다. 그 중 하나인 참가격 서비스는 데이터가 정기적으로 업데이트 되고, 등록된 점포의 수도 많았지만, 마트를 조회하는 범위가 시 단위로 주어져 실제로 활용해 장을 보는데 사용하기에는 부적합했다. 다른 하나인 소비자 물가정보 서비스는 서울특별시와 경기도 내에서만 한정된 서비스를 제공하며, 장바구니의 전체 가격을 산정해주지만, 참가격과 비슷하게 구 단위로 결과를 보여줘 한 눈에 어느 마트가 가장 저렴한지 파악하기 어려웠다. 또한, 두 서비스 모두 장바구니에 한 가지 제품은 한 개씩만 담을 수 있어 실용성이 떨어지는 문제점을 공통적으로 가졌다.

짱바구니 프로젝트는 사용자의 위치를 기반으로 담은 장바구니 가격을 직관적으로 비교하여 장을 보는 것 경험을 개선하기 위해 시작하였다. 해당 목적에 맞게 짱바구니는 사용자 주변 매장을 지도를 통해 보여주고 장바구니에 담긴 물건들의 가격을 쉽게 비교할 수 있도록 정보를 제공한다. 또한 매장까지 실제 거리를 계산해 정보를 전달한다.

2 Design

이번 장에서는 짱바구니의 전반적인 디자인을 서술한다. 사용자가 직접 마주하는 View에서 시작해서 보이지 않는 서버와 데이터베이스 설계까지 설명할 것이다.

2.1 Overall Architecture

짱바구니 웹 서비스는 크게 세 가지 페이지로 나눌 수 있다. 각 페이지의 목적은 다음과 같다.

1. *HomeView*는 첫 페이지로 사용자의 위치 정보를 알아내는 것을 목적으로 한다.
2. *MainView*는 위치 정보를 전달 받아 사용자가 원하는 상품을 장바구니에 담는 것을 목적으로 한다.
3. *MapView*는 사용자가 장바구니에 담은 상품의 가격 정보를 기반으로 지도에 표현하는 것을 목적으로 한다.

다음은 짱바구니 서비스의 동작 흐름을 보여주는 화면이다. 해당 디자인은 Figma[1]를 사용해 제작하였다.

– *HomeView*: 사용자가 처음 마주하는 화면이다. 앞서 언급했던 사용자의 위치 정보를 알아내는 기능으로 주소 입력창과 현위치를 주소로 설정 버튼을 구현하고자 하였다.

Fig. 1: HomeView에서의 작동 순서

– *MainView*: 주소를 입력받거나 현위치를 파악하면 주변 매장의 상품들을 한 눈에 볼 수 있는 화면이다. 카테고리에 따라 상품을 조회할 수 있고, 장바구니에 담고 수량조절 및 삭제기능을 수행할 수 있다.

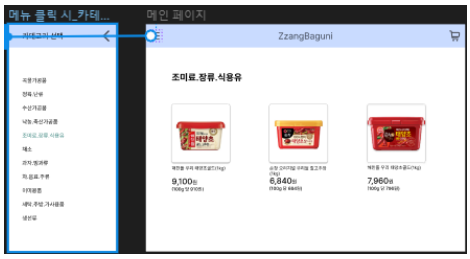

Fig. 2: MainView에서의 카테고리 조회 기능

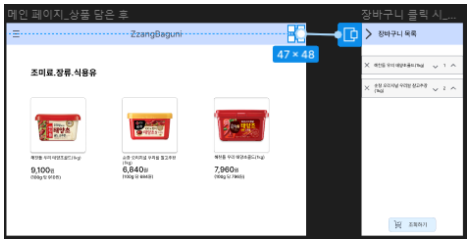

Fig. 3: MainView에서의 장바구니 기능

– *MapView*: 사용자가 장바구니에 상품을 모두 담은 후 '지도에서 보기'버튼을 누르면 이동하는 화면이다. 장바구니에 담은 상품들의 총 가격을 매장 별로 정렬하여 왼쪽에 리스트 형태로 보여주고, 최저가인 매장은 별도로 표시해주는 기능을 구현하고자 하였다.

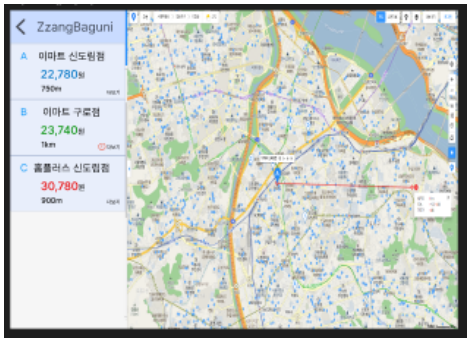

Fig. 4: MapView에서의 기능

2.2 HomeView

사용자가 서비스에 접속하면 처음으로 보게 되는 랜딩 페이지이다. 사용자 주변 매장의 물품을 불러오기 위해서는 사용자의 주소가 필요한데 HomeView에서는 두 가지 방법으로 주소를 설정할 수 있다.
첫번째로 다음 도로명주소 API 서비스[2]를 사용해 사용자가 설정하고 싶은 주소를 직접 검색하여 입력할 수 있다.
두번째로 브라우저에서 위치 정보 사용을 허용할 경우 현재 브라우저를 사용중인 위치를 사용자의 좌표로 사용하는 기능을 구현하였다.

2.3 MainView

HomeView에서 주소를 입력받은 후 이동하는 페이지이다. HomeView에서 도로명주소를 입력하였을 경우 서버에서 해당 주소를 좌표로 변환해주고 위치정보를 사용하였을 경우 좌표를 바로 MainView로 전달해준다. MainView에서는 해당 좌표를 기반으로 서버에서 인근 매장의 상품들을 받아와 그리드뷰 형태로 사용자에게 표시해준다.
사용자는 좌측의 카테고리 메뉴를 사용해 많은 상품들 중 원하는 카테고리의 상품에 편리하게 접근할 수 있고 가격 글씨 위에 마우스를 올려 인근 매장들에서 해당 상품의 가격 정보를 확인할 수도 있다.
원하는 상품을 찾았을 경우 담기 버튼을 사용해 장바구니에 상품을 담을 수 있고 우측 장바구니 메뉴를 클릭하면 담긴 상품을 확인할 수 있다. 장바구니 메뉴에서 상품을 개수를 추가하거나 삭제할 수 있고 장바구니 구성을 완료하였을 경우 지도에서 보기 버튼을 사용해 MapView로 넘어갈 수 있다.

2.4 MapView

다음으로 MapView는 크게 MainView에서 받아온 위치 정보를 토대로 근처 매장들을 우선순위에 따라 나열해서 리스트로 보여주는 사이드바 부분과 해당 매장들의 위치를 보여줄 지도부분으로 나뉜다.
사이드바에서는 MainView에서 넘겨받은 좌표, 장바구니 정보를 사용해 MapView 페이지가 마운트된 직후 백엔드 서버의 /prices API에 AXIOS 요청을 보내 매장 리스트를 렌더링한다.
매장들의 위치를 지도상에서 보여주는 지도 부분은 네이버 지도 API[3]를 사용하여 구현하였다.

2.5 API 서버

앞서 살펴본 여러 View에서 필요한 정보를 수집하고 이후 사용하기 용이하게 변형하여 제공하기 위해 서버가 반드시 필요하다. 짱바구니에는 여러 웹 백엔드 프레임워크 중 Flask[4]가 적합하다고 판단되어 Flask를 사용해 API 서버를 설계하였다. Flask는 Python 기반의 마이크로 웹 백엔드 프레임워크로 웹 백엔드 프레임워크 중 가장 가벼워 개발 속도가 빠르고 필요한 기능만 쉽게 확장할 수 있다.

MVC 패턴

소프트웨어 디자인 패턴 중 하나로 Model-View-Controller를 뜻한다. 핵심은 비즈니스 로직과 view를 분리하고 이를 controller가 관리 가능하다는 점이다. 예를 들어 사용자가 가격 정보(View)가 필요할 경우 Controller가 이를 Model을 통해 DB에서 데이터를 가져오거나 Service를 통해 데이터를 정제하여 사용자에게 응답한다. 다음은 Model, View, Controller에 대한 설명이다.

- *Model*: DB를 관리하는 부분과 비즈니스 로직(Service)로 구성되어 있다. 해당 프로젝트의 경우 DB에 CRUD하는 로직들로 구성되어 있고 Service에는 이미지 크롤링, 도로명 주소 변환 등 필요한 비즈니스 로직이 있다.

- *View*: Jinja Templete 엔진을 사용한 html 렌더링이 가능하나 해당 프로젝트에서는 Vue.js를 사용하는 관계로 서버에서는 별도로 표현하지 않는다. 대신 사용자가 필요한 정보를 일정 호출을 받으면 정해진 규격의 데이터를 전달하는 API를 나타낸다.

- *Controller*: 사용자가 URI와 필요한 parameter를 이용해 호출하면 해당하는 URI에 맞게 View를 응답하거나 Model을 업데이트 하는 등의 로직을 호출한다.

API 설계

다음은 전반적인 필요한 API 기능과 클라이언트와 서버의 통신 흐름이다.

- *HomeView*에서 사용자의 위치를 도로명 주소로 받아왔을 때 이를 좌표로 변환하여 응답한다. 이때 해당 좌표 정보는 현재 위치를 기반으로 필요로 하는 정보를 받아올 때 사용된다.

- *MainView*에서 필요한 API는 좌표 정보를 기반으로 반경 5km안의 매장에 있는 상품들을 불러오는 것이다. 또한 카테고리에 따라서 상품이 달라지기 때문에 그 카테고리에 맞는 상품을 불러오는 기능도 필요로 한다. 이렇게 불러온 상품 정보는 사용자가 장바구니에 담아 서버에 전달한다. 추가적으로 상품 별로 가격을 미리 확인할 수 있도록 주변 매장의 해당 상품 가격 정보를 전달한다.

- *MapView*에서는 사용자가 장바구니에 담은 상품을 기반으로 가격 정보를 불러와 각 매장 별로 장바구니 전체 가격 정보를 불러온다. 그 외에도 매장 위치 등의 정보를 함께 불러온다.

정리하면 전체적인 흐름은 다음과 같다.

1. 사용자의 위치를 도로명 주소 혹은 좌표를 이용해 파악한다. 만약 도로명 주소일 경우 API를 호출해 좌표로 변환한다.

2. 사용자의 좌표를 통해 반경 5km안의 매장의 상품을 파악한다. 카테고리 별로 분류해 상품을 전달한다.

3. 사용자가 장바구니에 상품을 담으면 주변 매장의 해당 상품 가격을 파악한다. 추가로 지도에 보여줄 수 있도록 매장 위치도 함께 전달한다.

2.6 Database

상품, 매장 정보 및 가격 정보를 저장하기 위해 데이터베이스가 필요하다. 해당 정보를 제공하는 곳마다 데이터 구조가 달라 하나의 스키마로 통일시키는데 별도의 처리가 필요하고 무엇보다도 사용자가 최적의 매장을 선택하는데 필요한 기준이 여러가지 있어 상품, 매장에 대한 최대한 다양한 정보를 담기 위해 고정된 스키마를 사용하지 않는 NoSQL 데이터베이스 중 MongoDB[5]를 사용하였다.

DB 설계

공공기관에서 수집한 데이터들을 사용자에게 자유롭게 제공하는 공공데이터포털에서 데이터를 수집하였으며 그 안의 "한국소비자원 생필품 가격 정보"[6]를 기반으로 데이터베이스를 구축하였다. 크게 세가지 Collection으로 구성되어 있다.

- *goodlist*: 모든 상품의 정보를 담고 있는 것으로 이름, 용량, 분류(다섯 자리의 카테고리) 등으로 구성되어 있다. 자세한 구성요소는 다음 Fig. 5에서 확인 가능하다.


```
goodId: 991
goodName: "농심 새우깡(90g)"
goodSmlclsCode: 30205004
goodTotalCnt: 90
goodTotalDivCode: "G"
goodUnitDivCode: "G"
```


Fig. 5: goodlist 예시

- *entplist*: 모든 매장 정보를 담고 있는 것으로 매장 명, 도로명 주소, 지번 주소 등으로 구성되어 있고 추가로 도로명 주소를 통해 변환한 위도, 경도 정보도 가지고 있다. 자세한 구성요소는 다음 Fig. 6에서 확인 가능하다.


```
entpId: 1561
entpName: "롯데슈퍼흡전점"
entpTelno: "null"
entpTypeCode: "SM"
plmkAddrBasic: "경기도 수원시 장안구 흡전동 89-5"
plmkAddrDetail: "null"
postNo: "16357"
roadAddrBasic: "경기도 수원시 장안구 서부로 2161"
roadAddrDetail: "(흡전동)"
latitude: 37.3012700271384
longitude: 126.972459841823
```


Fig. 6: entplist 예시

– *price*: 매장, 상품 별로 매칭된 가격 정보를 담고 있는 것으로 상품ID, 매장ID로 구분하여 가격을 확인할 수 있다. 그 밖에 조사일 정보가 있어 이를 통해 2주마다 갱신되는 API 정보를 이용해 DB도 업데이트 한다. 자세한 구성요소는 다음 Fig. 7에서 확인 가능하다.


```
goodInspectDay: "20221021"
entpId: 1561
goodId: 991
goodPrice: 1280
```


Fig. 7: price 예시

3 Implementation

3.1 Front-End

HomeView 구현

HomeView는 Navigation Bar, 비디오, Address.vue 컴포넌트로 구성되어 있다. Navigation Bar는 HomeView와 AboutView를 이동할 수 있는 간단한 메뉴이다.

비디오는 Adobe Stock의 무료 라이센스 비디오 중 Grocery Store를 키워드로 검색하였을 때 노출되는 영상을 활용하였다. HTML5 비디오 플레이어를 화면 중앙에 위치시켰고 비디오 플레이어 위에 불투명 레이어를 한 겹 추가하여 비디오 위에 위치한 Address.vue 컴포넌트의 글씨가 잘 보이도록 하였다.

Address.vue는 HomeView에서 핵심적인 역할을 하는 vue 컴포넌트로 다음 도로명주소 API와 Vue Geolocation 기능을 사용할 수 있는 vue 컴포넌트이다. 검색창과 검색 버튼에는 클릭 이벤트가 발생할 시 팝업으로 다음 도로명 주소 API를 띄우도록 해놓았다. 주소 검색창 하단의 초록색 “현재 위치를 주소지로 설정” 링크는 클릭 이벤트가 발생할 시 브라우저에서 위치 정보 사용을 허용할 것인지 물어보는 알림창이 발생하고 허용할 경우 위치 정보를 사용해 좌표를 받아온다.

Address.vue의 도로명주소 API 또는 Geolocation 둘 중 어떤 기능을 사용하더라도 주소 검색 또는 현재 위치를 받아온 후 MainView로 자동으로 넘어가도록 구현하였다.

MainView 구현

MainView에서는 Sidebar.vue, Card.vue, AddedProductList.vue 세 가지의 vue 컴포넌트를 활용하였다.

1. Sidebar.vue는 MainView에서 표시되는 상품들을 카테고리 별로 탐색할 수 있는 카테고리 메뉴를 보여주는 사이드바이다. 모바일 화면의 경우 사이드바가 항상 표시될 경우 화면을 차지해 상품들을 탐색하는데 방해가 될 수 있으므로 상단의 네비게이션 바의 버튼을 통해 토글 할 경우에만 좌측에서 나타나는 형태의 off canvas 형태의 사이드바로 구현하였다. 기본적으로 전체 카테고리가 선택되어 있고 각 카테고리를 선택함에 따라 메뉴에서 해당 카테고리가 하이라이트 처리 된다.

2. Card.vue는 MainView 중앙에서 표시되는 상품 카드 컴포넌트이다. 상품의 이미지, 카테고리, 담기 버튼과 가격 텍스트로 이루어져 있다. 담기 버튼을 클릭할 경우 장바구니에 상품을 담을 수 있으며 여러번 클릭할 경우 장바구니에 담긴 상품의 개수가 증가한다. 가격 텍스트에 마우스 오버 이벤트가 발생할 경우 사용자의 좌표 주변의 상점들에서 해당 상품의 가격 정보들을 최저가 순으로 표시하는 테이블이 상품 이미지 위로 노출되며 마우스 오버 이벤트가 종료될 경우 다시 상품 이미지가 노출된다.

3. AddedProductList.vue는 장바구니 기능을 구현한 vue 컴포넌트이다. Sidebar.vue와 동일한 off canvas 형태의 사이드바로 상단 네비게이션 바의 쇼핑카트 버튼을 통해 토글 할 경우 우측에서 나타나는 형태이다. 상품 카드의 담기 버튼을 통해 상품을 담았을 경우 Cart.vue에서 상품을 확인할 수 있고 상품의 개수를 추가하거나 줄일 수 있고 상품을 삭제할 수도 있다. 사용자가 원하는 장바구니 구성을 완료하였을 경우 MapView로 넘어가는 “지도에서 보기” 버튼을 장바구니 사이드바 하단에 배치하였다.

MapView 구현

MapView는 크게 리스트와 지도 두 부분으로 나누어져 있다. 리스트의 경우 서버에서 가격 정보 데이터를 받아와서 총 가격을 계산하고 상품의 유무를 확인 후 선정한 우선순위를 통해 정렬한 후 vue template에 리스트 렌더링하는 과정을 거쳤다. 이때 세 가지 기준의 우선순위는 다음과 같다.

1. 장바구니에 담긴 상품이 매장에 모두 존재할 때

2. 총 가격이 더 저렴할 때

3. 매장까지의 실제 거리가 더 가까울 때

추가로 직선 거리를 이용하는 것보다 실제거리를 사용하는 것이 사용자에게 직관적으로 느껴진다 판단했다. 이를 측정하는 데에는 Naver Map Driving API[7]를 사용하였고 두 좌표를 통해 가는 길, 거리를 확인할 수 있었다. 다음 Listing 1.1는 총 가격을 계산하는 예제 코드이다.

```
<script>
calculatePrice(){
    let productsInfo = this.$store.getters.getcartProducts;
```

```
    for(let idx in this.entp){
        let row = this.entp[idx];
        let entpId = row["entpId"];
        let total = 0; let cnt =0;
        for(let i=0;i<productsInfo.length;i++){
            let priceItem = this.price[entpId][productsInfo[i
    ]["goodName"]];
            let QuantityItem = productsInfo[i]["goodQuantity"
    ];
            if(!isNaN(priceItem*QuantityItem)) {
                total += priceItem * QuantityItem;
            }
            else{
                cnt++;
                this.price[entpId][productsInfo[i]["goodName"
    ]] = "-";
            }
        }
        row["total_price"] = total;
        row["no_product"] = cnt;
    }
    this.sort_entp();
}
</script>
```

Listing 1.1: 매장 리스트 구현

지도의 경우 서버에서 사용자가 지정한 위치를 받아와서 네이버 지도 API를 이용해 지도 객체를 vue template에 표시해주었다. 지도 구현에 대한 자세한 코드는 다음 Listing 1.2에서 확인할 수 있다.

```
<template>
 <main class="d-flex flex-nowrap">
    <!--리스트 관련 템플릿 생략-->
  <div class="col">
   <div id="map"></div>
  </div>
 </main>
</template>
<script>
let reqBody = {goods: this.$store.getters.getcartOnlyId,
    latitude: this.latitude, longitude: this.longitude};
this.axios.post("https://zzangbaguni.shop/prices",reqBody).
    then((res)=>{
 this.entp = res.data.entp;
 this.good = res.data.good;
 this.price = res.data.price;
 this.calculatePrice();
 var mapOptions = {
        //맵옵션 생략
 };
```

```
 var map = new window.naver.maps.Map("map",mapOptions);
 this.map = map;
 var infowindow = new window.naver.maps.InfoWindow();
 this.setmarker(this.entp,map,infowindow);
 this.displayPlaces(this.entp,map);
}).catch((err)=>{
 console.log(err);
});
</script>
```

Listing 1.2: 지도 객체 구현

3.2 Back-End

Overall Architecture

앞서 필요한 API를 설계하였다. 이번 장에서는 이 설계된 API를 구현하기 위해서 필요한 몇 가지 기능과 쿼리문을 최적화한 과정, 테스트 환경에 대해 살펴본다.

기능 구현

주소를 다루는 로직은 크게 두 가지가 있다.

- 카카오맵 API[8]를 이용한 도로명 주소 좌표로 변환
- Haversine library[9]를 이용한 반경 5km 내의 매장 검색

첫번째 로직은 카카오맵 주소 검색 API를 활용한 것으로 도로명 주소를 parameter로 위도와 경도 정보를 얻을 수 있다. 해당 정보를 entplist collection에 update 한다. 두번째 로직은 여러 정보를 불러올 때 사용자 주변 매장만 필요하다는 것을 위한 로직으로 반경 5km내의 매장ID 만을 list형태로 정리하여 출력한다. 이때 앞에서 구한 좌표를 통해 거리를 측정하는데 여기서 haversine library를 사용하여 거리를 측정한다. 이는 두 점의 위,경도 좌표를 통해 직선거리를 구하는 library 이다. 자세한 코드는 다음 Listing 1.3에서 확인할 수 있다.

```
def get_distance(a, b):
    return haversine(a, b, unit="km")

def find_near_entp(now_pos, rad=5):
    entp_list = find_all_entp()
    entpId_near = []
    for entp_row in entp_list:
        x, y = entp_row['latitude'],entp_row['longitude']
        dist = get_distance(now_pos, (x, y))
        if dist <= rad:
            entpId_near.append(entp_row['entpId'])
```

```
12      return entpId_near
```

Listing 1.3: 일정 반경 내의 매장 불러오기

다음은 사용자에게 보여줄 상품 이미지를 처리하는 과정이다. 웹 크롤링을 돕는 라이브러리 Selenium[10]을 사용하여 상품 이미지를 불러온다. 여러 이미지를 반복적으로 불러와야 하기 때문에 브라우저를 직접 동작시켜 시뮬레이션을 통해 컨텐츠를 가져오는 Selenium을 택하였다. 이때 Chrome Webdriver를 사용하여 구글 이미지 검색을 통해 수집한다. 자세한 코드는 다음 Listing 1.4에서 확인할 수 있다.

```
def search_keyword(driver, keyword, goodId):
    search = driver.find_element(By.XPATH, '/html/body/div
    [1]/div[3]/form/div[1]/div[1]/div[1]/div/div[2]/input')
    search.send_keys(keyword)
    search.send_keys(Keys.RETURN)
    image = driver.find_elements(By.CSS_SELECTOR, '#islrg >
    div.islrc > div:nth-child(2) > a.wXeWr.islib.nfEiy > div.
    bRMDJf.islir > img')
    a = image[0].get_attribute('src')
    urllib.request.urlretrieve(a, 'images/'+str(goodId)+ '.
    jpg')
```

Listing 1.4: selenium을 이용한 이미지 크롤링

보다 원활한 속도를 위해 병렬처리를 통하여 진행하였으며 필요한 상품 list를 쪼개어 4개의 멀티프로세스를 이용하여 처리하였다.

DB 구현

다음은 각종 로직을 구현하는데 필요한 Data를 불러오는 Model에 대한 구현 방법이다. 크게 두 가지로 구분할 수 있다.

- 2주마다 갱신되는 Data를 insert/update 하는 것
- 필요한 조건에 맞는 Data를 read(find)하는 것

첫번째의 경우 price collection의 새로운 조사일의 정보를 update한다. 다음은 Open API를 통해 해당 조사일의 data를 불러와 replace하는 과정이다. 이때 upsert Option은 중복되는 값이 있다면 update, 없다면 insert를 하는 쿼리이다. 자세한 코드는 다음 Listing 1.5에서 확인할 수 있다.

```
def update_price(date):
    data = load_price(date)
    collect_price = db.price
    for row in data:
        collect_price.replace_one({"goodId": row['goodId'], "
    entpId": row['entpId']}, row, upsert=True)
```

Listing 1.5: 가격 정보를 갱신하는 과정

두번째의 경우 필요한 정보를 find하는 과정으로 가장 많이 사용되는 경우이다. /prices API에서 사용되는 로직을 예시로 들어보자.

필요한 상황은 상품 ID List, 주변 매장 ID List를 통해서 각각의 상품 가격을 출력하는 것이다. 그리고 테스트 환경은 160만개의 price data에서 8328개의 가격 data를 찾아야 하는 상황이다. 이는 사용자가 모든 상품을 장바구니에 담았을 최악의 상황을 가정했다. 다음은 여러 케이스 별 challenge 상황이다. 자세한 코드는 Listing 1.6에서 확인할 수 있다.

```
def find_price(goodId, entp_id_list):
    collect_price = db.price
    entp = new_find_entp(entp_id_list)
    price = list(collect_price.find({"entpId": {"$in":
    entp_id_list}, "goodId": goodId}, {"entpId": 1, "
    goodPrice": 1, "_id": 0}).sort("goodPrice"))
```

Listing 1.6: DB에서 가격정보를 불러오는 상황

1. 모든 상품, 매장 경우의 수를 돌려가며 가장 간단한 방법을 택하였을 때
   모든 경우의 수를 반복적으로 find()하기 때문에 983.53초라는 실제로 사용할 수 없는 시간이 걸렸다.

2. $in 연산자를 통해 find() 함수를 수정하였을 때
   다음 코드처럼 찾는 ID를 하나씩 매칭한 것이 아니라 한번의 쿼리로 모든 list 안의 요소를 찾도록 하는 연산자를 이용하여 개선한 결과 4.5초가량 걸렸다.

3. 탐색 기준이 되는 상품ID와 매장ID에 Index를 부여하였을 때
   탐색 속도를 향상시키기 위해 두 ID에 composite Index를 부여하였다. 그 결과 0.6초가량 소요되어 충분히 사용할 수 있을 만큼 작동하였다. 대신 탐색을 제외한 추가, 삭제 등에서 소요시간이 더 걸리지만 이는 탐색에 비해 자주 작동하는 쿼리가 아니고 소요시간이 길어도 서버 안에서 발생하는 쿼리이므로 사용자의 경험에는 영향을 미치지 않는다.

Case	**1**	**2**	**3**
time(sec)	983.5	4.5	0.6

Table 1: Case 별 Query 소요 시간

3.3 AWS, CI/CD

AWS

짱바구니 웹 어플리케이션은 AI를 사용하지 않고 실제 사용자가 많은 상태가 아니므로 구현에 많은 연산이나 트래픽을 감당할 수 있는 서버가 필요하지 않다고 판단하였고 12개월 간 무료로 사용할 수 있는 Free-Tier 요금제를 제공하는 Amazon Web Service의 (EC2)Elastic Compute Cloud를 사용하였다. 다음 과정을 통해 AWS EC2에 짱바구니 웹 어플리케이션을 배포할 준비를 하였다.

1. EC2 프리티어 인스턴스를 생성한다. 프리티어 인스턴스의 기본 볼륨은 8GB이나 젠킨스에서 빌드 리소스가 차지하는 용량이 많아 프리티어 최대 볼륨인 30GB로 설정하였다.

2. 프리티어 인스턴스의 램 용량은 1GB이므로 원활한 서비스 및 배포를 위해 SWAP 공간을 2GB 할당해준다. [11]

3. 설치 및 관리의 용이성을 위해 프론트엔드의 Vue와 Nginx 웹서버, 백엔드의 Flask, MongoDB 모두 도커 컨테이너로 사용하였다. 도커 컨테이너를 사용하기 위해 EC2에 도커를 설치해준다.

4. 웹 어플리케이션 서비스를 위해 필요한 80, 443 포트와 개발 단계에서 필요한 기타 포트를 방화벽 인바운드 규칙에서 개방해준다.

CI/CD

CI/CD는 Continuous Integration, Continuous Distribution의 약자로 빌드/배포/테스트 등의 자동화를 의미한다. 도커를 사용하였기 때문에 OS, package 등의 호환성 문제 없이 nginx, flask, mongodb 컨테이너를 설치할 수 있지만 프론트엔드인 vue, 백엔드인 flask에 수정사항이 생겼을 경우 배포 전 도커 이미지로 빌드하는 과정을 한번 더 거쳐야 하기 때문에 도커를 사용하지 않을 때에 비해 배포 과정이 다소 복잡하고 길어질 수 있다. 도커를 사용하지 않더라도 개발 단계에서 빌드 배포 과정은 자주 발생하기 때문에 여기서 발생하는 시간을 단축시키기 위해 개발 초기에 CI/CD툴인 Jenkins를 사용하여 빌드/배포 과정을 자동화하였다. 짱바구니의 Jenkins 설정 과정은 다음과 같다.

1. EC2 인스턴스에 도커를 사용해 Jenkins를 설치한다.

```
    $ docker run -d --name jenkins -p 8080:8080
    -v /jenkins:/var/jenkins_home
    -v /usr/bin/docker:/usr/bin/docker
    -v /var/run/docker.sock:/var/run/docker.sock
    -u root jenkins/jenkins:lts

```

Listing 1.7: 젠킨스 컨테이너 실행

   이 때 호스트의 도커를 Docker-Out-Of-Docker 모드로 사용하기 위해 -V(Volume) 옵션을 통해 Host의 Docker 경로를 Mount 해준다.

2. 깃허브에서 소스코드를 가져와 빌드에 사용하기 위해 팀 레포지토리에 접근할 수 있는 깃허브 Credential을 등록해준다.

3. 빌드 성공, 실패 알림을 Slack을 통해 받아보려면 Slack Credential도 등록하고 Slack에 Jenkins 확장프로그램을 설치한다.

4. 프론트엔드, 백엔드 빌드 과정을 담은 .jenkinsfile 문서를 작성하여 깃허브 레포지토리에 push 하고 빌드 시 해당 jenkinsfile을 사용하도록 설정한다. jenkinsfile

은 pipeline syntax를 따르고 environment, stages, post 등의 단계를 통해 SCM Pull, Build, Deploy 등의 단계를 정의하여 파이프라인처럼 실행할 수 있다.

4 Limitations and Discussions

4.1 데이터 부족

이번 프로젝트는 공공데이터 포탈 API에서 받아온 데이터만을 사용하여 구현했다. 따라서 데이터 수가 부족했고 API에서 제공하는 정보만을 받을 수 있었기에 그 외의 정보들을 알 수 없었다. 또한 해당 API는 갱신주기가 2주이기 때문에 짧게 유지되는 물품 할인을 제대로 반영할 수 없었다. 이 문제는 향후 공공기관 또는 민간기업의 유료 데이터 등을 제공받아서 해결할 수 있을 것으로 보인다.

4.2 매장 추천 알고리즘

10주라는 한정적인 시간을 두고 구현을 완료해야했던 프로젝트의 시간적 한계로 인해 매장 추천 알고리즘이 프로젝트 초기단계에서 구상했던 것에 비해 구현 되지 못한 부분이 있었다. 예를 들어 초기 구상단계에서는 상품들의 총 무게, 거리별로 유류비 혹은 교통비를 계산해서 상품들의 총 가격과 더불어 우선순위의 기준으로 설정하려 했으나 현재는 총 가격과 단순한 이동거리로 우선순위를 설정하고 있다. 이는 향후 이러한 미구현점들을 고려하려 프로젝트를 추가로 진행한다면 해결할 수 있을 것으로 보인다.

5 Related Work

1. 참가격
   참가격[12]은 한국소비자원에서 운영하는 가격정보 종합 포털사이트이다. 참가격은 전국 단위 유통업체(대형마트, 기업형 슈퍼마켓, 백화점, 편의점)에서 판매하는 가공식품, 생활용품, 신선식품 등 생필품 128개 품목(337개 상품)의 판매가격을 격주 조사하여 제공한다. 이 정보를 바탕으로 품목 별 지역 최저가 매장, 전국 평균가 등의 정보를 확인할 수 있다.
   해당 사이트는 공공데이터포털 “한국소비자원 생필품 가격 정보”[6]에서 데이터셋을 사용한다.

2. 소비자 물가정보 서비스
   소비자 물가정보 서비스[13]는 한국소비자단체협의회에서 운영하는 가격정보 종합포털사이트로 서울, 경기 지역에서의 대형마트 및 슈퍼마켓의 생필품 품목별 가격정보를 제공하고 있다. 매월 말마다 데이터를 제공한다.

3. 토마토:우리동네 장보기
   해당 앱은 주변 마트 찾기, 세일 정보, 배달 등의 기능을 한다. 해당 데이터셋은 토마토[14]라는 마트 POS기 시스템을 제공하는 회사에서 직접 제공한 것이다.

4. 편의점 서비스
   각 편의점에서는 크게 두 가지 서비스를 제공하고 있다. 첫째, 해당 편의점의 판매 품목 가격 정보 및 할인 상품 정보 제공한다. 둘째, 편의점 APP을 통해서 각 업체마다의 재고 정보를 제공한다. 편의점 별로 자체적으로 정보를 제공하고 있고 APP에서만 재고를 확인할 수 있는 점 편의점 끼리 비교가 불가능하다는 점을 볼 수 있다.

6 Conclusion

지금까지 장바구니에 담긴 물건의 가격을 직관적으로 비교할 수 있는 웹 서비스 짱바구니에 대해 살펴보았다. 짱바구니의 주요 기능은 사용자의 위치를 기반으로 반경 5km내의 매장에 있는 상품을 보여주고 해당 상품을 장바구니에 담아 전달하면 근처 매장에서의 가격 정보를 알 수 있는 것이다. 이때 짱바구니의 다른 서비스와의 차별점은 무엇보다도 직관적으로 지도를 통해 가격을 살펴보고 매장이 얼마만큼 거리가 떨어져 있는지, 또 물품의 유무까지 고려하여 최적의 매장을 시각적으로 확인할 수 있다는 점이다. 매장과의 거리도 단순히 직선거리가 아닌 실제 이동거리인 만큼 이후 주유비 등을 고려하여 최적의 매장 추천의 확장성도 열려있는 부분이다. 현재로선 공공 데이터에 의존하는 경향이 있지만 이후 다양한 매장 정보 업체와 컨택을 통해 충분히 발전 가능성이 있다고 생각한다. 짱바구니를 통한 직관적이고 현명한 소비로 여러분들의 장을 보는 경험이 개선되기를 바란다.

References

1. Figma, https://www.figma.com
2. Daum Postcode Service User Guide, https://postcode.map.daum.net/guide
3. Naver Map API Guide, https://navermaps.github.io/maps.js.ncp/docs/index.html
4. Flask, https://flask.palletsprojects.com/en/2.2.x
5. MongoDB, https://www.mongodb.com/home
6. 공공데이터포털 한국소비자원 생필품 가격 정보, https://www.data.go.kr/tcs/dss/selectApiDataDetailView.do?publicDataPk=3043385.
7. Naver Map driving API, https://api.ncloud-docs.com/docs/ai-naver-mapsdirections-driving
8. Kakao Map API Guide, https://developers.kakao.com/docs/latest/ko/local/dev-guide
9. Haversine, https://pypi.org/project/haversine
10. Selenium, https://www.selenium.dev/documentation
11. EC2 memory swap, https://aws.amazon.com/ko/premiumsupport/knowledge-center/ec2-memory-swap-file
12. 참가격, https://www.price.go.kr/tprice/portal/main/main.do.
13. 소비자물가정보서비스, http://price.consumer.or.kr.
14. 토마토, https://www.tomato-market.co.kr/index.html.