CS492 A조 기술문서

20160633 조재민
20170057 김 건
20170719 한승희

프론트 엔드

1. 어플리케이션 화면 구성

1) Login Activity

login activity는 사용자가 어플리케이션을 처음 실행했을 때 보게 되는 화면으로, 아래 사진과 같이 구성되었다. login activity는 크게 새로운 이름과 아이디, 비밀번호를 활용해 회원가입을 진행하는 기능과 이미 만들어 둔 계정 정보를 이용해 로그인을 진행하는 기능으로 구성되어 있다.

먼저 회원가입 기능의 경우, 새 계정 만들기 버튼을 누르면 회원가입을 진행할 수 있는 dialog가 나오게 된다. dialog는 이름, 아이디, 비밀번호, 확인용 비밀번호를 입력할 수 있는 4개의 EditText로 구성되어 있으며 해당 내용을 입력한 뒤 회원가입 버튼을 누르면 사용자가 입력한 정보를 서버로 보내게 된다. 서버는 이 내용을 바탕으로 계정을 생성하고 유저 정보를 저장한다.
로그인 기능의 경우 아이디와 비밀번호를 입력할 수 있는 2개의 EditText로 구성되어 있으며 해당 내용을 입력한 뒤 로그인 버튼을 누르게 되면 사용자가 입력한 정보를 서버로 보내게 된다. 이후 서버로부터 받은 응답 내용에 따라 사용자 계정이 확인될 경우 서버로부터 받은 유저 정보를 저장하고 main activity로 넘어가게 되며 그렇지 않은 경우 아이디 또는 비밀번호가 일치하지 않는다는 내용의 toast message를 띄우게 된다.

2) Main Activity

Main activity는 activity_main.xml과 연결되어 있으며 세부 fragment들을 전환할 수 있는 탭으로 구성되어 있다. View pager 기능을 통해 구현되어 있으며, ViewPagerAdapter를 통해 사용할 수 있다. 탭 버튼을 누르거나, 슬라이드를 하게 되면 선택된 fragment들이 호출되어 나온다.
탭은 아래에 있는 바를 통해 전환할 수 있는데, 기본적으로 text를 적어서 구분하고 있으며, 아이콘을 삽입하여도 크기가 매우 작다. 그래서 탭 레이아웃 위에 따로 이미지를 불러오고, 선택된 탭 위의 이미지는 검정색으로 채워진 이미지로 대체하는 방법으로 구현하였다.

그리고 내부 fragment들에서 선언할 수 없는 함수들인 showSelect(), showSelectProfile(), share()의 경우에 main activity에서 선언한 후, 내부 fragment에서 호출하여 사용하게 된다.

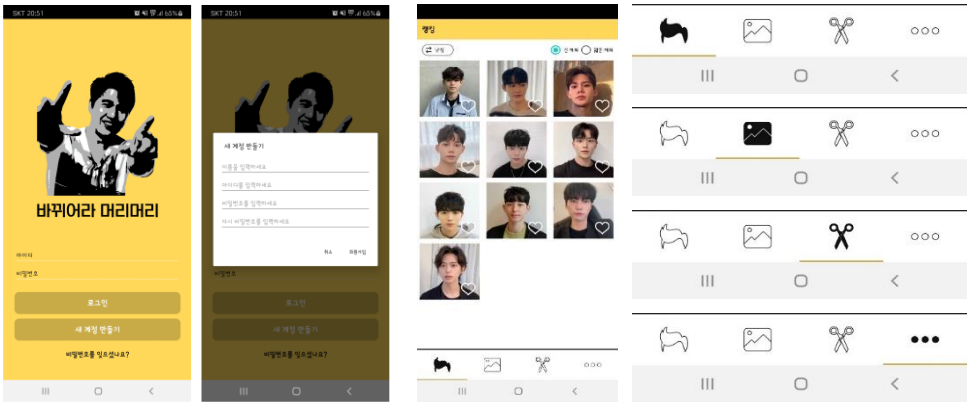

그림 1 로그인 화면 (좌), Main Activity (중), 탭 종류 (우)

3) Ranking Tab

Ranking Tab은 fragment_1, RecyclerViewAdpater_1, 그리고 fragemt_1.xml로 구성되어 있다. 사진의 개수와 상관없이 보여줄 수 있도록 recycler View를 활용하여 구성 되어있다. fragment_1에서 현재 선택된 성별과 머리 길이에 맞는 헤어 사진이 ClassPhoto라는 클래스로 선언되어 List<ClassPhoto>로 recycler view adapter에 들어간다. recycler view 한 칸마다 사진과 좋아요 수, 해시태그 등의 정보가 담겨있다.

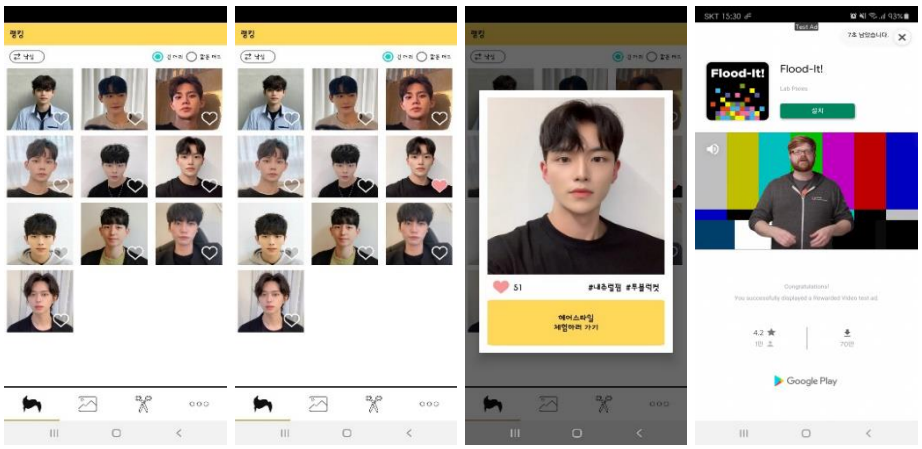

그림 2 Ranking Tab

Recycler view의 한 칸 한 칸 조작을 위해서 onBindBiewHolder()의 position이라는 변수를 사용하여 컨트롤 한다. 하트를 누르게 되면 좋아요 / 좋아요 취소가 되면서 해당 사진의 좋아요 수가 변하게 된다. 만약 사진을 누르게 되면 dialog가 나타나는데, dialog_photo.xml을 통해 디자인되어 있다. dialog엔 사진이 크게 나타나면서 좋아요 수와 해시태그 등의 정보가 나타난다. 밑에 있는 '헤어스타일 체험하러 가기'라는 버튼을 누르면 AdMob의 reward형 광고가 나온다.
광고를 끝까지 보면 look_ad라는 Boolean 변수가 true로 전환이 되며, look_ad가 true일 경우에 PictureActivity로 전환이 되면서 딥 페이크를 위해 셀카를 찍을 수 있다.

4) Gallery Tab

Gallery Tab는 fragment_2, RecyclerViewAdpater_2, 그리고 fragemt_2.xml로 구성되어 있다. 사진의 개수와 상관 없이 보여줄 수 있도록 recycler View를 활용하여 구성 되어있다. fragment_2에서 현재 유저의 ID로 요청되어 있는 딥 페이크 사진 목록과 진행된 단계를 서버에서 받아온다. ClassGallery라는 클래스로 선언되어 List<ClassGallery>로 recycler view adapter에 들어간다. recycler view 한 칸마다 딥 페이크에 사용된 헤어 스타일과 진행중인 상태가 표기가 된다. 그 중, 완료된 사진은 클릭하면 dialog가 나오면서 완료된 사진을 확인할 수 있는데, dialog_gallery.xml을 통해 디자인되어 있다. 서버에서 완료된 사진을 받아와서 보여준다. 우측 하단의 세 버튼을 통해 다운로드, 공유, 삭제 기능을 사용할 수 있다. 다운로드를 누르면, 현재 보이는 사진이 갤러리에 저장이 되며, 공유 버튼을 누르면 main activity에 선언했던 share() 함수가 실행이 되며 공유를 위한 activity로 전환이 된다. 삭제의 경우 recycler view adapter에 들어오는 리스트에서 지움과 동시에 서버에서도 이 사진에 대해 삭제를 요청한다.

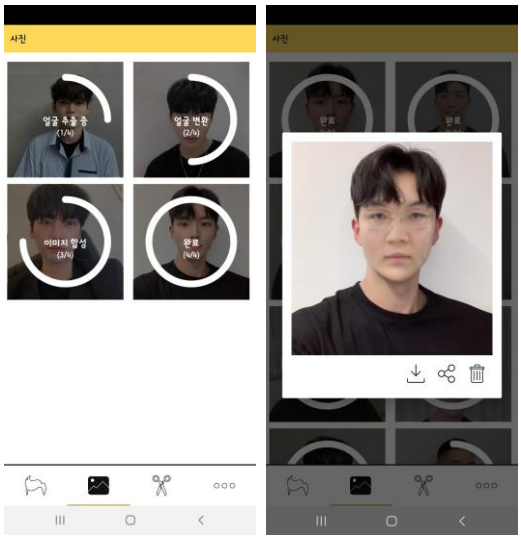

그림 3 Gallery Tab

5) Map Tab

Map Tab는 fragment_3, RecyclerViewAdapter_3, 그리고 fragment_3.xml로 구성되어 있다. Fragment는 크게 두 부분으로 나뉘어져 있는데, 위에는 Google Map API를 활용하여 지도를 나타내고, 아래에는 Recycler View를 만들어서 미용실들의 리스트를 보여주고 있다. 유저의 GPS 정보를 활용하여 유저의 위도와 경도를 알아내면 지도는 유저의 현재 위치를 중심으로 표시가 되고, 위치 정보가 서버로 전송되어 유저 근처에 있는 미용실들의 리스트를 받아올 수 있다. Store_data라는 클래스로 선언되어 List<Store_data>로 recycler view adapter에 들어간다. recycler view 한 칸마다 미용실의 이름과 사진, 별점, 커트 가격 등이 표기되어 있다. 그 중, 하나를 선택해서 누르게 되면 dialog가 나오면서 미용실에 대해 더 자세한 정보를 확인할 수 있는데, dialog_storecall.xml을

통해 디자인되어 있다. 여기에선 추가로 영업시간과 전화번호, 그리고 예약 가능 시간이 나온다. 예약 가능 시간 중 하나를 선택하게 되면 해당 미용실에 예약을 할 수 있는데, 본인 사진 추가 버튼을 누르게 되면 Main Activity에서 선언했던 showSelect() 함수가 실행되어 사진을 선택할 수 있도록 사진첩이 실행된다.

6) Mypage Tab

Mypage Tab 는 fragment_4, RecyclerViewAdapter_4, 그리고 fragment_4.xml 로 구성되어 있다. Fragment 는 크게 두 부분으로 나뉘어져 있는데, 위에는 유저의 프로필을, 아래에선 recycler view 를 만들어서 세세한 설정들의 리스트를 보여주고 있다.

프로필에선 로그인할 때 서버에서 받아온 유저의 정보들을 보여주고 있는데, 프로필 사진 부분을 클릭하게 되면, main activity 에서 선언했던 showSelectProfile() 함수가 실행되고, 여기서 선택한 사진으로 유저의 프로필이 변경된다.

Recycler view 로 구성되어 있는 설정들은 recycler view adapter 에서 대부분의 동작이 이루어지는데, 어떤 설정을 클릭하였는지는 position 이라는 변수를 통해 구별을 하게 된다. 그 중 멤버십 버튼을 클릭하게 되면, dialog 가 나타나는데, 이 dialog 에서는 프리미엄 회원과 일반 회원 사이의 전환이 가능하다. 그리고 로그아웃 버튼을 클릭하게 되면 현재 로그인 중인 계정에서 나가게 되면서 다시 로그인 화면이 나타난다.

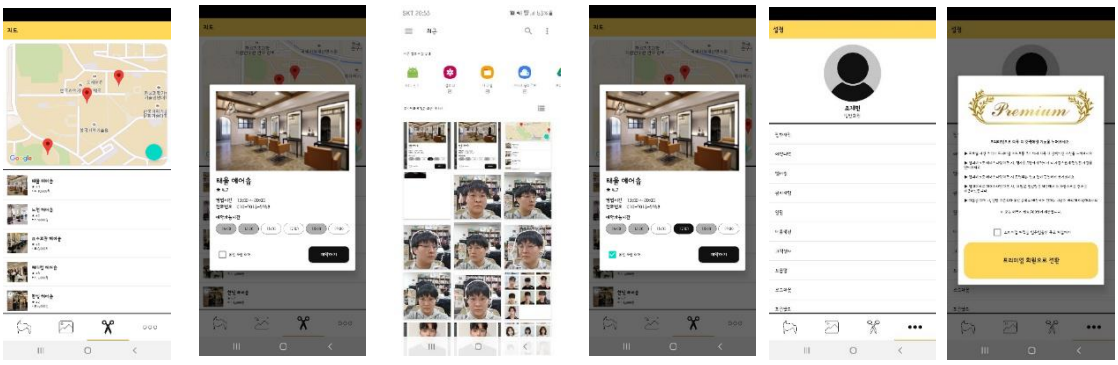

그림 4 Map Tab (좌), Mypage Tab (우)

7) Picture Activity

Picture Activity는 사용자가 본인의 사진을 찍을 때 활용된 activity로, 아래와 같이 구성되었다. 이 activity에서는 디바이스의 camera를 활용해 사진을 찍을 수 있도록 하고, 사진을 찍은 이후 서버로 해당 사진을 전송하는 역할을 한다.

Camera를 어플리케이션에서 실시간으로 볼 수 있도록 하기 위해서 CameraSurfaceView class를 활용하였다. CameraSurfaceView class는 카메라 raw 데이터를 가져와 우리가 만든 layout에서 볼 수 있게 해주며, layout에서 CameraSurfaceView 객체를 만들고 해당 객체에 camera를 연결해 줌으로써 활용할 수 있다.

사용자가 카메라 모양 버튼을 누르게 되면 사진이 찍히고 해당 사진을 위쪽의 빈칸에 표시하며, 필요한 만큼의 사진을 찍은 이후에는 OK 버튼을 활성화해 서버로 찍은 사진을 업로드 할 수 있도록 하였다. 이 때 유저 정보와 유저가 고른 헤어스타일 정보를 함께 보내 서버에서 딥 페이크 과정을 받은 사진과 헤어스타일을 통해 진행하고 유저 별 갤러리 DB를 관리할 수 있도록 하였다.

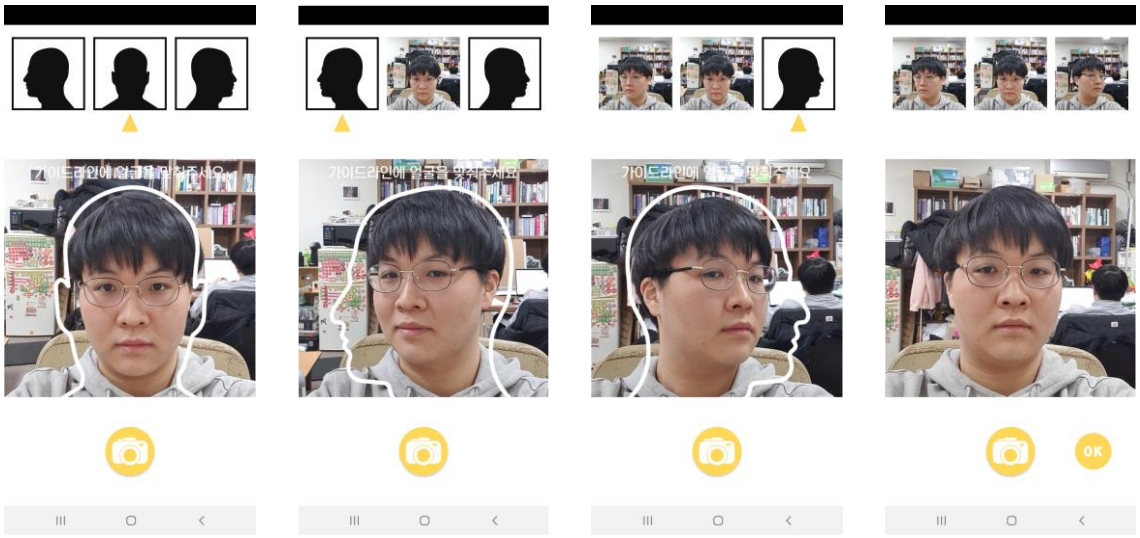

그림 5 Picture Activity

2. 라이브러리 활용

1) retrofit2

retrofit 라이브러리는 Square사에서 제공하는 오픈소스 라이브러리러 REST API를 안드로이드에서 쉽게 이용할 수 있게 해주는 도구이다. 이 프로젝트에서 우리는 retrofit2 라이브러리를 활용한 서버와의 http request, response통해 여러 가지 정보를 주고받았다. retrofit2 라이브러리를 활용해 서버와 통신하기 위해서는 여러가지 다음과 같은 과정이 필요하다.

먼저, RetrofitClient class에서 retrofit2의 통신을 위한 기본적인 구조체를 정의해야 한다. 여기에는 통신하고자 하는 서버의 baseURL이 포함되며, 이후 통신을 하는 과정에서 이 구조체를 활용하여 request, response를 진행하게 된다. 또, 이 구조체를 활용해 통신을 하기 위해서는 프론트 엔드

어플리케이션과 백 엔드 서버 간의 request, response와 관련된 규칙이 필요하다. 이들은 어떤 정보들을 request하며 보내고 어떤 정보들을 서버로의 response로부터 받아올지에 대한 내용이며 이 프로젝트에서는 이러한 정보들을 편리하게 handle하기 위해 request data class와 response data class를 정의하여 통신에 활용하였다. 정보 class를 활용하기 위해 JAVA 오브젝트들의 직렬화, 역직렬화를 해주는 구글의 gson library를 활용하여 구성하였으며, 이러한 정보 class 안에는 정보를 다른 곳에 활용하기 위한 다양한 convenience function들도 정의할 수 있다. 마지막으로 ServiceApi class에서는 위에서 언급한 규칙을 정의한다. 해당 class에 정의된 통신 macro를 통해 request URL과 request, response 정보를 정의할 수 있으며 이 프로젝트에서는 위에서 정의한 request data class의 constructor를 활용하여 request를 진행하였고 response data class를 활용하여 서버의 response를 진행하였다.

간단한 예시를 들기 위해 이 프로젝트에서 사용된 login 통신 과정을 예시로 들어보겠다. 먼저 앞서 언급한 RetrofitClient class는 아래 사진과 같이 구성되었다. Class 안에 정의된 Retrofit 구조체에서 사용하는 BASE_URL이 우리가 백 엔드 서버에서 external connection에 사용한 ngrok 사이트의 도메인을 가지는 주소로 설정되어 있는 것을 확인할 수 있다.

```
public class RetrofitClient {
    private static String BASE_URL = "https://9c7f9b5ac0e2.ngrok.io/";
    private static Retrofit retrofit = null;

    public static Retrofit getClient() {
        if (retrofit == null) {
            retrofit = new Retrofit.Builder()
                    .baseUrl(BASE_URL)
                    .addConverterFactory(GsonConverterFactory.create())
                    .build();
        }
        return retrofit;
    }
}
```

그림 6 RetrofitClient Class

또, 아래 사진의 LoginData class는 login 관련 request를 사용하기 위해 사용되는 정보 class이며 LoginResponse class는 login관련 서버로부터의 response를 handle하기 위해 사용되는 정보 class이다. 프론트 엔드 어플리케이션에서는 id와 password 정보를 http 프로토콜의 body에 담아 서버로 보내고, 서버로부터 user_id, user_name, user_type, status의 유저 관련 정보들을 받음을 확인할 수 있다.

```java
public class LoginData {
    @SerializedName("id")
    String id;

    @SerializedName("password")
    String password;

    public LoginData(String get_id, String get_password) {
        this.id = get_id;
        this.password = get_password;
    }
}
```

```java
public class LoginResponse {

    @SerializedName("user_id")
    private String user_id;

    @SerializedName("user_name")
    private String user_name;

    @SerializedName("user_type")
    private int user_type;

    @SerializedName("status")
    private String login_status;

    public String get_id() { return user_id; }
    public String get_name() { return user_name; }
    public int get_type() { return user_type; }
    public String get_status(){ return login_status; }
    public Boolean check_status() {
        String pass = "OK";
        if (login_status.equals(pass)){
            return true;
        } else{
            return false;
        }
    }
}
```

그림 7 LoginData Class (좌), LoginResponse Class (우)

마지막으로, 아래 사진의 ServiceApi class 안에 login 관련 규칙이 userLogin function으로 정의되어 있는 것을 볼 수 있다. BASE_URL/login/login의 주소로 통신을 진행하며 LoginData class의 정보와 POST method를 활용해 request를 진행하고 서버로부터 받은 response를 LoginResponse class를 통해 handle하도록 정의된 것을 알 수 있다.

```java
public interface ServiceApi {
    @POST("/login/login/")
    Call<LoginResponse> userLogin(@Body LoginData data);
```

그림 8 ServiceApi Interface

이렇게 정의된 다양한 class들은 아래 사진처럼 구조체를 만들고 해당 구조체를 통해 request를 진행한 뒤 response를 받아오는 형태로 활용된다. 아래의 onResponse 함수의 overriding을 통해 response를 handle한 뒤 check_status라는 LoginResponse class 내에 정의된 convience function을 활용하여 정보를 처리하고 있음을 볼 수 있다.

```java
service = RetrofitClient.getClient().create(ServiceApi.class);
```

그림 9 ServiceApi class 호출

```
private void startLogin(LoginData data) {
    System.out.println("start login");
    service.userLogin(data).enqueue(new Callback<LoginResponse>() {
        @Override
        public void onResponse(Call<LoginResponse> call, Response<LoginResponse> response) {
            LoginResponse result = response.body();
            if (result != null) {
                if (result.check_status()) {
                    System.out.println(result.get_status());
                    System.out.println(result.check_status());
                    String login_status = result.get_name() + "님 환영합니다!";
                    Toast.makeText( context: SplashActivity.this, login_status, Toast.LENGTH_SHORT).show();
                    Intent intent = new Intent( packageContext: SplashActivity.this, MainActivity.class);
                    startActivity(intent);
```

그림 10 프론트 엔드의 로그인 함수

다른 통신 방식과는 다르게 이미지를 서버로 보내고 받기 위해서는 예외적으로 multipart 방식을 사용하였다. Multipart란 데이터를 http 프로토콜의 body 부분에 여러 부분으로 나눠 담아 보내고 받는 방식으로, http 프로토콜을 활용한 파일 전송에 널리 사용되는 방식이다. 우리 프로젝트에서는 아래 사진과 같이 활용되었다.

```
@Multipart
@POST("/request/request/")
Call<PictureResponse> uploadPicture(
        //@Path("template")userId: Int,
        @Part("description") RequestBody description,
        @Part MultipartBody.Part file1,
        @Part MultipartBody.Part file2,
        @Part MultipartBody.Part file3,
        @Part("id") String user_id,
        @Part("hair") String user_hair
);
```

그림 11 retrofit2 Multipart Request 정의

2) google map

Google map 라이브러리는 google에서 제공하는 지도 관련 API이다. 이 기능을 활용하기 위해서는 먼저 Google 계정을 활용해 Maps SDK for android의 사용자 인증 정보와 API key를 발급받아야 한다. 또, 이 API key를 AndroidManifest 파일에 입력하여 어플리케이션이 사용할 수 있도록 해야 한다.

```
<meta-data
    android:name="com.google.android.gms.ads.APPLICATION_ID"
    android:value="API-KEY"/>
```

그림 12 AndroidManifest에 API key 입력

위 과정을 진행한 이후에는, onMapReady function의 overriding을 활용해 사용할 수 있다. layout에서 MapView 객체를 만들고 onMapReady function 내부에서 해당 MapView에 Googlemap 객체를 create하는 방식으로 사용된다. 아래 사진에서 onMapReady function의 overriding을 확인할 수 있다.

```
@Override
public void onMapReady(GoogleMap googleMap) {
    g_googleMap = googleMap;

    MyApplication myApp = (MyApplication) getActivity().getApplication();
    myApp.gset_map(g_googleMap);

    adapter = new RecyclerViewAdapter_3(getContext(),store_list, g_googleMap,getActivity());
    recyclerView.setAdapter(adapter);

    MapsInitializer.initialize(this.getActivity());
    goToLocationZoom(now_lat, now_lon,  zoom: 15);
    add_markers();
}
```

그림 13 onMapReady function

추가적으로, 현재 위치를 확인하기 위해서는 GpsTracker class를 활용하였다. 해당 class에는는 스마트폰의 GPS 데이터를 활용해 현재 장치의 latitude와 longitude를 반환하는 함수들이 있다. 이 함수들을 활용해 현재 위치로 google map이 보여주는 위치를 update할 수 있도록 하였으며 아래 사진과 같이 구성되었다.

```
private void updateLoc(){
    now_lat = gpsTracker.getLatitude();
    now_lon = gpsTracker.getLongitude();
    System.out.println(now_lat);
    System.out.println(now_lon);
    goToLocationZoom(now_lat,now_lon, zoom: 15);
}
```

그림 14 updateLoc function

3) Google AdMob

Google에서 제공하고 있는 AdMob은 개발자를 고려하여 설계된 광고 라이브러리로, 자동화 도구를 통해 앱 수익을 늘리고 사용자 경험을 개선하며 활용 가능한 유용한 정보를 얻을 수 있다. Native 광고, Reward 광고, Banner 광고 등, 다양한 형태의 광고를 제공하여 개발자들이 자신의 어플리케이션에 맞는 형태를 골라 사용할 수 있는데, 우리는 이 중에서 Reward 광고를 사용했다.
구글 AdMob을 사용하기 위해선 우선, 실명 인증된 구글 계정을 통해 로그인하고, AdMob에 개발하려는 어플리케이션을 등록해야 한다. 그 후, 앱에 넣고 싶은 광고의 형태를 '광고 단위'로 앱에 추가한다.

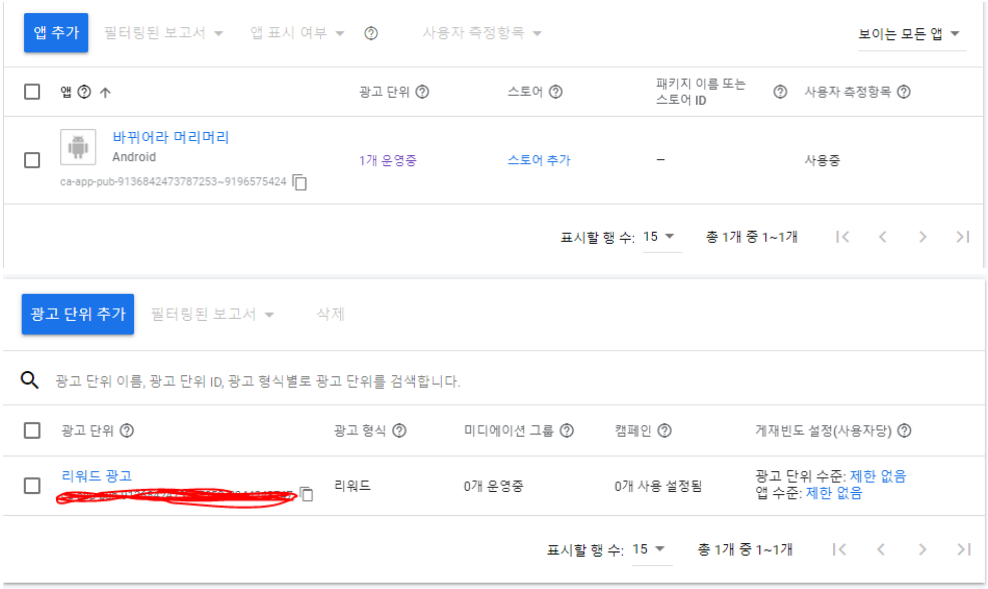

그림 15 Google AdMob Reward 광고 추가

그 후, https://developers.google.com/admob/android/quick-start?hl=ko에서 안내하는 대로, 앱에서 광고를 게재하고 싶은 부분에 광고를 넣으면, 광고를 사용할 수 있다. 우리는 Reward 광고를 사용했기 때문에 아래처럼 광고를 끝까지 본 경우, 광고를 닫았을 경우 등 각각의 경우에 따라 실행할 동작을 나누었고, RecyclerViewAdapter_1에서 일반 회원이 '헤어스타일 체험하러 가기' 버튼을 누르면 광고가 실행되도록 설정했다.

백 엔드

1. 백 엔드 서버 구축

본 프로젝트에서 백 엔드 서버를 구축하기 위해 사용한 4가지 핵심요소는 다음과 같다. 우선 웹 서버 프레임 워크로는 Django를 선정하였다. Django는 파이썬 기반 프레임워크로 쉽게 모델을 생성하고 변경이 가능하다는 장점이 있다. 또한 본 프로젝트에서 사용한 데이터베이스인 SQLite3와의 상성이 잘 맞아, SQLite3 내부의 데이터를 쉽게 접근할 수 있다. SQLite3는 오픈소스 데이터베이스로 DB 관리를 위한 별도의 프로세스를 상시로 실행하고 있지 않아도 되며, python의 sqlite3 라이브러리를 통해 쉽게 조작할 수 있다는 장점이 있다. 딥 페이크 모델로는 FaceSwap github 라이브러리를 사용하였다. FaceSwap은 소스 이미지와 타겟 이미지가 주어졌을 때 주어진 이미지를 바탕으로 새로운 모델을 학습시키는 방식을 채택하여 다른 딥 페이크 모델에 비해 더 자연스러운 이미지를 생성해준다. 마지막으로 구축한 백 엔드 서버를 외부의 IP에서 접속이 가능하도록 만들기 위해 Ngrok을 사용하였다. 본 프로젝트에서 사용한 KCloud 서버는 NAT에 연결이 되어 특정 포트를 열어 서버를 구동한다 하더라도 외부에서는 해당 서버에 접속이 가능하지 않다. Ngrok을 사용하면 KCloud 서버의 특정 포트로 연결이 되는 URL을 생성이 되어 해당 URL을 이용하는 방식으로 외부 IP에서 동작하는 안드로이드 어플과 Django서버와의 통신을 구축하였다.

2. 백 엔드 서버 개발 환경 구축

1) Anaconda

수업에서 제공받은 KCloud 서버에 백 엔드 서버 개발환경을 구축하기 위해 Anaconda를 활용하기로 결정하였다. 본 수업에서 Anaconda를 설치하는 과정은 다음과 같다.

```
wget https://repo.anaconda.com/archive/Anaconda3-2020.11-Linux-x86_64.sh #
download anaconda installer
bash ./Anaconda3-2020.11-Linux-x86_64.sh # install anaconda
```

그림 16 Anaconda 설치 프로그램 다운로드 및 설치

Anaconda를 설치한 이후, 'backend'라는 이름의 새로운 virtual environment를 생성한 이후, 해당 environment에 우리의 파이썬 개발환경을 구축하였다. 사용한 python 버전은 python 3.7.9이며 Environment를 생성하는 과정은 다음과 같다.

```
conda create -n backend python=3.7.9 # create new environemnt
conda env list # Show all env list
conda activate backend # activate backend environment
conda deactivate # deactivate backend environment
```

그림 17 Environment 생성 및 Environment 활성화

2) FaceSwap 설치

본 프로젝트에서는 딥 페이크 이미지를 생성하기 위해서 FaceSwap github 라이브러리를 활용하였다. FaceSwap은 입력으로 제공한 소스 이미지와 타겟 이미지에 최적화된 모델을 새로 학습하여 딥 페이크 이미지를 생성하여 다른 모델에 비해 자연스러운 이미지를 만들어 낼 수 있다는 장점이 있다. FaceSwap을 설치한 과정은 다음과 같다.

```
conda activate backend # activate backend environment
git clone https://github.com/deepfakes/faceswap.git # clone repository
cd faceswap/ # change to faceswap directory
python setup.py # install faceswp
python faceswap.py -h # Check installation is successfully done
```

그림 18 FaceSwap 라이브러리 설치

```
Enable AMD support? [y/N] n
Enable Docker? [y/N] n
Enable CUDA? [Y/n] y
Please ensure your System Dependency are met. Continue? [y/N] y
```

그림 19 설치 도중 옵션 선택

3) Ngrok 설치

외부 IP에 위치하고 있는 안드로이드 기기에서 백 엔드 서버로의 접근을 가능하게 만들기 위해 Ngrok을 설치하였다. Ngrok은 conda 명령어를 활용하여 서버에 설치 하였으며 과정은 다음과 같다.

```
conda activate backend # Activate backend environment
conda install -c conda-forge pyngrok # install ngrok
```

그림 20 Conda 명령어를 활용한 Ngrok 설치

4) Django 설치 및 기타 라이브러리 설치

a) Django

Django를 활용한 개발환경을 구축할 때에는 파이썬의 가상환경을 사용하는 것이 권장된다. 따라서 본 프로젝트에서는 anaconda로 생성한 environment 위에 Django 구동을 위한 파이썬 가상환경을 생성하여 개발을 진행하였다. 가상환경을 생성하고 Django를 설치하는 과정은 다음과 같다.

```
python3 -m venv djangoEnv # Create new virtual environment
source djangoEnv/bin/activate # activate environment
pip install Django
```

그림 21 파이썬 가상환경 생성 Django 설치

b) 기타 라이브러리

백 엔드 서버를 구동하기위해 다음과 같은 추가 라이브러리를 설치하였다. 우선 첫 번째로 간단한 이미지 프로세싱을 진행하기 위해 Pillow 라이브러리를 설치 하였으며 버전은 8.0.1 버전을 사용하였다. 그리고 Django 내에서 rest API를 활용하기 위해 Django rest framework 라이브러리를 설치하였고 버전은 3.12.2를 활용하였다. 그 외의 사용한 라이브러리의 정보는 Github Repository에 올린 backendrequirement.txt와 djangorequirement.txt를에 기록해 두었다. 이들은 각각 backend anaconda environment와 djangoEnv environment에서 pip freeze 명령어를 통해 추출한

파이썬 라이브러리의 정보가 담겨있다.

3. Screen을 통한 원격 프로세스 유지

본 프로젝트에서 백 엔드 서버를 개발하기 위해서는 ssh 접속을 통해 서버에 들어가 작업을 수행해야만 했다. 하지만 ssh 접속으로 실행한 프로세스는 원격 접속이 끊기는 순간 프로세스가 종료되기 때문에 해당 프로세스에 Django 서버를 구동 시킬 경우 서버를 원격접속이 끝난 경우 Django 서버 지속되지 않는 상황이 발생한다. 따라서 리눅스 소프트웨어인 screen을 이용하여 원격 접속이 끊더라도 Django 서버를 유지할 수 있게 하였다. 사용한 screen 명령어는 다음과 같다.

```
screen -S [process name] # 새로운 screen process를 생성한다.
screen -x [process name] # 기존의 screen process에 접속한다.
screen -list # 실행되어있는 screen process을 보여준다.
```

그림 22 사용된 Screen 명령어

4. 서버구동 코드

본 프로젝트의 백 엔드 서버를 완전히 구동하기 위해서는 총 3개의 프로세스를 실행시켜야 한다. 첫 번째 프로세스는 외부 IP에서 서버에 접근할 수 있도록 Ngrok을 실행되어 있다. 두번째 프로세스는 안드로이드 어플과 Rest API를 통해 통신을 진행할 Django 서버가 구동되어 있다. 마지막에는 DB에서 딥 페이크 이미지 생성 리퀘스트를 확인하고 딥 페이크 모델을 구동하는 프로세스가 실행되어 있다. 위의 3개의 프로세스를 실행하는 코드는 다음과 같다.

```
screen -S ngrok
~/ngrok http 20000
```

그림 23 Ngrok 실행

```
screen -S server
conda activate backend
source ~/djangoEnv/bin/activate
python ~/backend/repo/manage.py runserver 0.0.0.0:20000
```

그림 24 Django Server 실행

```
screen -S deepfake
conda activate backend
source ~/djangoEnv/bin/activate
cd /home/ubuntu/backend/repo/media
python start_deepfake.py
```

그림 25 FaceSwap 실행

5. 데이터베이스 구축

본 프로젝트에서는 백 엔드 서버에 어플리케이션 관련 정보를 관리하기 위해 SQLite3로 데이터베이스를 구축하였다. 서버에 사용하는 데이터베이스는 별도의 설치 없이 Django를 처음 설치 하였을 때 기본으로 같이 제공되는 SQLite3 DB를 그대로 활용하였다. 본 프로젝트의 DB는 총 3 종류의 테이블로 구성되어있다. 3종류의 테이블의 이름과 역할은 다음과 같으며 이들의 Relational Data Model Schema 는 그림 26에서 확인할 수 있다.

1) Login Info table - 로그인 관련 정보(아이디, 비밀번호)를 담고 있어 어플리케이션의 로그인 기능이나 회원가입 기능을 수행할 때 활용된다.
2) User Info table - 사용자 관련 정보를 담고 있으며 로그인을 수행할 때 프론트 엔드로 사용자 정보를 전송하기 위해 사용된다.
3) Request Info - 딥 페이크 이미지 생성 리퀘스트 관련 정보를 저장하고 있으며, 프론트 엔드로 각 리퀘스트의 진행상황 또는 완료된 이미지를 전송하기 위해 사용된다.

Login Info

UID	Password

User Info

UID	User_name	User_type	Profile_image

Reqeust Info

Index	UID	RID	Face_path	Hair_path	Converted_path	Progress

Hairshop Info

HID	Shop_name	Latitude	Longitude	Profile_image	Review	Phone	Business_hour

Hairshop_Time Info

Index	HID	Time_table

그림 26 각 테이블의 Relational Data Model Schema

6. 서버 API 구축

프론트 엔드와 백 엔드 간의 통신을 이루기 위해 REST API를 활용하여 총 10종류의 API를 만들었다. 10종류의 API는 로그인 기능, 딥 페이크 이미지 생성 기능, 그리고 유저 정보 수정 기능에 적용되었다.

1) 로그인 기능

로그인 기능에는 Login API 그리고 Register API가 활용되었으며 각각의 역할은 다음과 같다.

a) Login API

사용자의 어플리케이션 로그인 기능을 수행하기 위한 API로 프론트 엔드에서 POST login/login/의 주소로 아이디와 비밀번호 데이터를 백 엔드로 전송해오면 서버는 해당 아이디, 비밀번호 쌍이 DB의 Login Info table에 존재하는지 확인을 해준 이후, 매칭이 되는 데이터가 있다면 Status 값 'OK'와 함께 User Info의 해당 아이디를 사용하는 사용자 정보 값을 Response로 반환해준다. 만약 DB 상에 입력된 아이디를 찾지 못했을 경우 Status 값 'FAIL'을 반환해준다. Login API의 처리과정은 그림 27에서 확인할 수 있다.

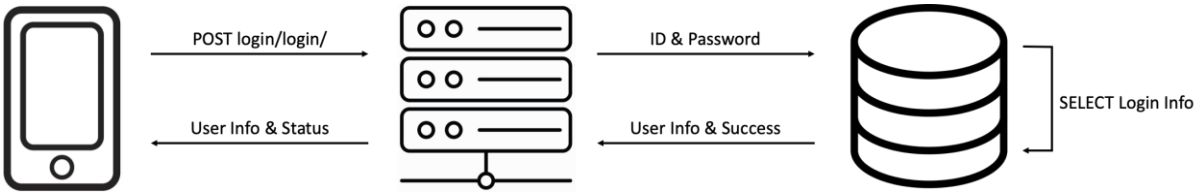

그림 27 Login API

b) Register API

어플리케이션의 회원가입 기능을 수행하기 위한 API로 프론트 엔드에서 POST login/register/의 주소로 아이디, 사용자 이름, 그리고 비밀번호의 정보를 백 엔드로 전송하면, 서버는 해당 아이디를 가지는 다른 계정이 있는지 확인을 한 후, 중복되지 않은 아이디라면 Login Info 테이블에 해당 로그인 정보를 추가하고 User Info에 해당 사용자의 정보를 추가한다. 이후 프론트 엔드로 Status 값 'OK'을 반환해준다. 만약 중복된 아이디가 있었다면 Status 값 'FAIL'을 반환해준다. Register API의 처리과정은 그림 28에서 확인할 수 있다.

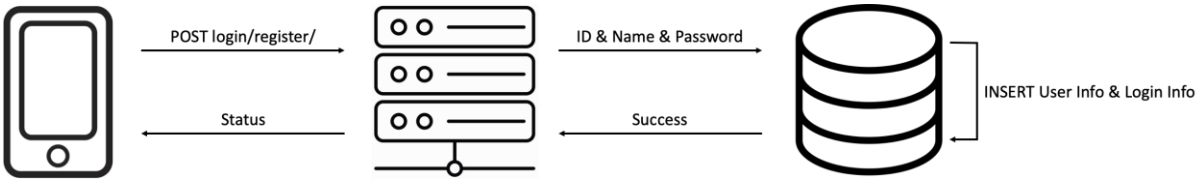

그림 28 Register API

2) 딥 페이크 이미지 생성 기능

딥 페이크 이미지 생성 기능에는 Request API, Update API, Download API, 그리고 Delete API가 활용되었으며 각각의 역할은 다음과 같다.

a) Request API

딥 페이크 이미지 생성 리퀘스트 생성 기능을 수행하기 위한 API로 프론트 엔드에서 POST request/request/의 주소로 아이디, 사용자 얼굴 이미지, 헤어스타일 인덱스를 백 엔드로 전송하면 서버는 Request Info 테이블에 리퀘스트 관련 정보를 추가하고 정상적으로 처리되었다면 프론트 엔드로 Status 값 'OK'와 함께 Request ID 값을 반환해준다. 이후, 딥 페이크 생성 프로세스가 새로운 리퀘스트를 발견한다면 딥 페이크 이미지 변환을 시작한다. Request API의 처리과정은 그림 29에서 확인할 수 있다.

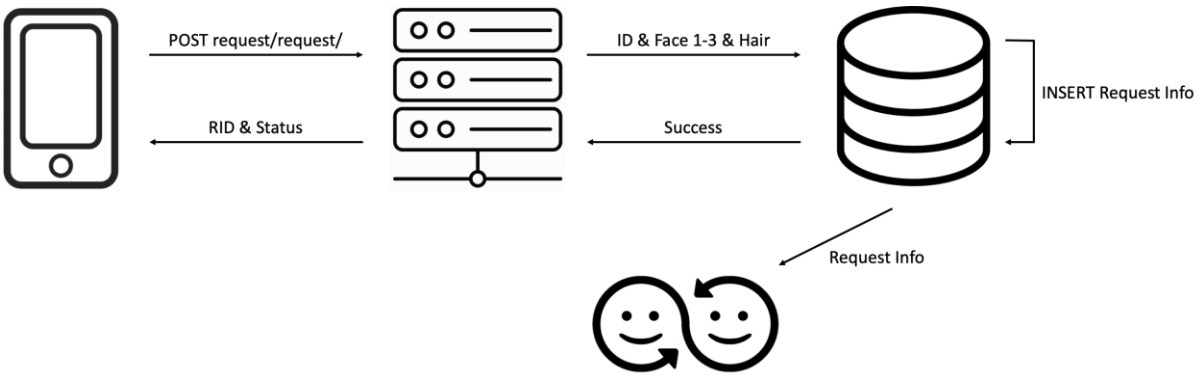

그림 29 Request API

b) Update API

프론트 엔드에서 요청해둔 리퀘스트의 정보를 업데이트를 하기위한 API로 어플리케이션에서 request/update/의 주소로 아이디 정보를 백 엔드로 전송하면, 서버는 Request Info 테이블에서 해당 아이디를 가진 유저가 요청한 리퀘스트 정보를 리스트로 변환하여 해당 정보 리스트를 반환한다. Update API의 처리과정은 그림 30에서 확인할 수 있다.

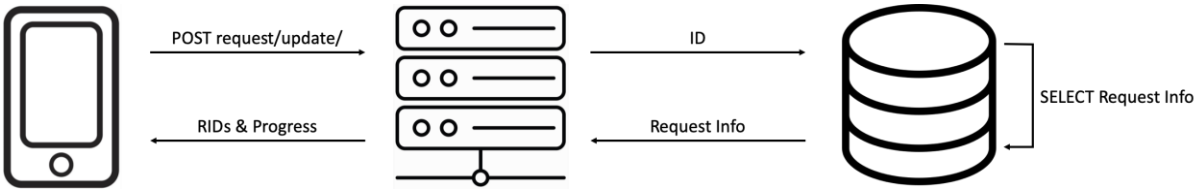

그림 30 Update API

c) Download API

완료된 리퀘스트 항목의 결과 이미지를 서버에서 받기 위한 API로 프론트 엔드에서 사용자의 아이디와 리퀘스트 아이디 값을 백 엔드로 전송하면, 서버는 완성된 딥 페이크 이미지를 비트맵으로 변환하여 Status 값과 함께 프론트 엔드로 반환한다. Download API의 처리과정은 그림 31에서 확인할 수 있다.

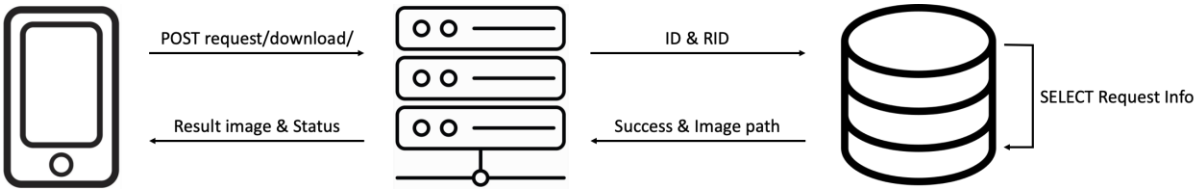

그림 31 Download API

d) Delete API

완료된 리퀘스트 항목을 DB에서 제거하기 위한 API로 프론트 엔드에서 사용자의 아이디와 리퀘스트 아이디 값을 백 엔드로 전송하면, 서버는 Request Info 테이블에서 해당 리퀘스트 항목을 제거한다. 해당 과정이 성공적으로 마쳤다면 Status 값 'OK'를 프론트 엔드로 반환한다. Delete API의 처리과정은 그림 32에서 확인할 수 있다.

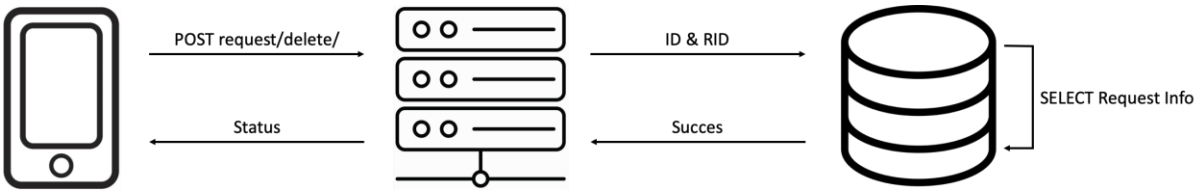

그림 32 Delete API

3) 미용실 상호작용 기능

미용실 상호작용 기능에는 UpdateShop API, 그리고 Reservation API가 활용되었으며 각각의 역할은 다음과 같다.

a) UpdateShop API

사용자 주변의 미용실 데이터를 얻기 위한 API로 프론트 엔드에서 사용자의 현재 위치 정보를 백 엔드로 전송하면, 서버는 Hairshop Info와 Hairshop Time Info 테이블에서 현재 위치 주변의 미용실들의 정보를 찾아와 Status값 'OK'와 함께 프론트 엔드로 반환한다. UpdateShop API의 처리과정은 그림 33에서 확인할 수 있다.

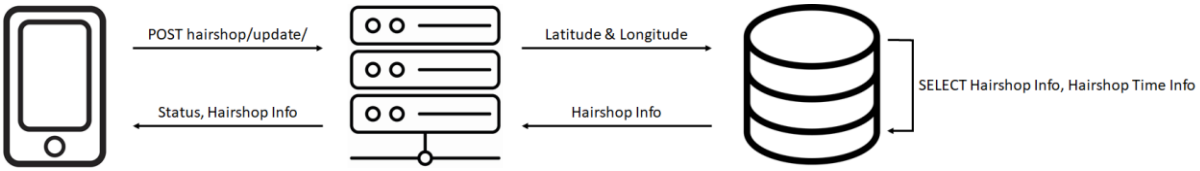

그림 33 UpdateShop API

b) Reservation API

미용실에 예약을 하기 위한 API로 프론트 엔드에서 사용자의 아이디, 미용실의 아이디 그리고 딥 페이크 이미지를 서버로 전송하면, 서버는 Hairshop Info와 User Info에서 사용자의 정보와 미용실의 정보를 불러와 해당 미용실에 사용자의 예약정보를 전송한다. 성공적으로 예약정보가 전송되었다면 Status 'OK'를 프론트 엔드로 반환한다. Reservation API의 처리과정은 그림 34에서 확인할 수 있다.

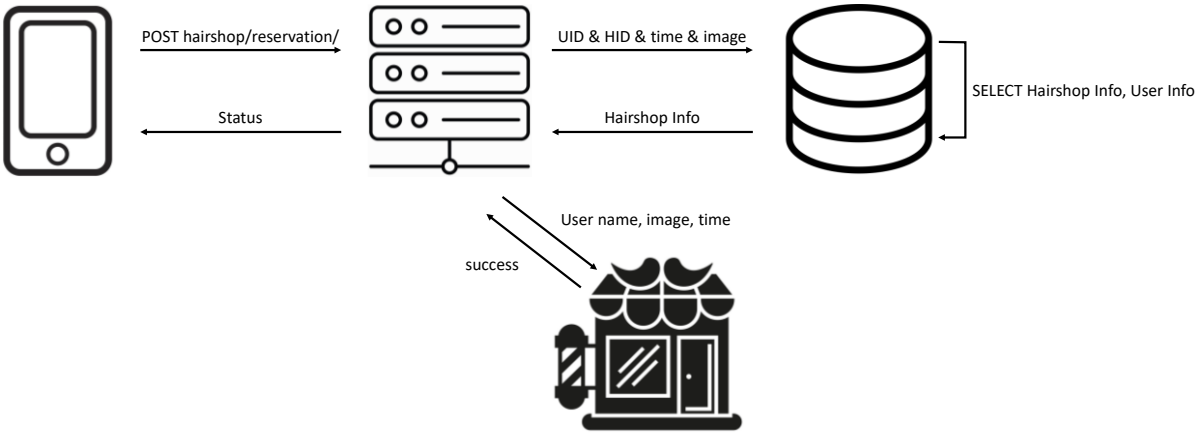

그림 34 Reservation API

3) 유저 정보 수정 기능

유저 정보 수정 기능에는 Upgrade API, Downgrade API, SetProfile API, 그리고 ResetProfile API가 활용되었으며 각각의 역할은 다음과 같다.

a) Upgrade API

사용자의 유저 타입을 Normal에서 Premium으로 전환하는 기능을 수행하기 위한 API로 프론트 엔드에서 POST user/upgrade/의 주소로 사용자의 아이디를 백 엔드로 전송하면, 서버는 User Info 테이블의 해당 사용자 User_type값을 Premium으로 수정한 이후 프론트 엔드로 Status 값 'OK'를 반환해준다. Upgrade API의 처리과정은 그림 32에서 확인할 수 있다.

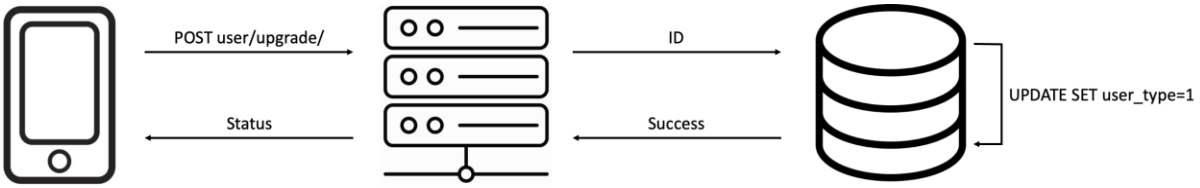

그림 35 Upgrade API

b) Downgrade API

사용자의 유저 타입을 Premium에서 Normal으로 전환하는 기능을 수행하기 위한 API로 프론트 엔드에서 POST user/downgrade/의 주소로 사용자의 아이디를 백 엔드로 전송하면, 서버는 User Info 테이블의 해당 사용자 User_type값을 Normal으로 수정한 이후 프론트 엔드로 Status 값 'OK'를 반환해준다. Downgrade API의 처리과정은 그림 33에서 확인할 수 있다.

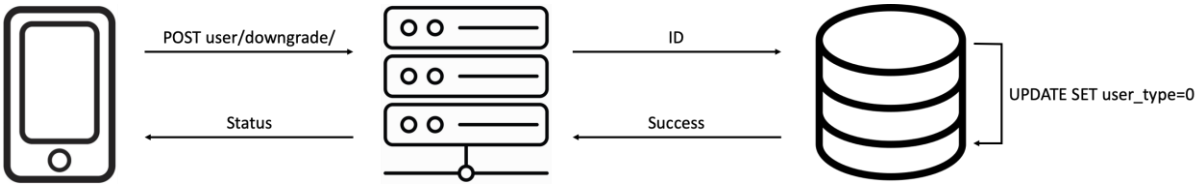

그림 36 Downgrade API

c) SetProfile API

사용자의 프로필 사진을 적용하는 기능을 수행하기 위한 API로 프론트 엔드에서 POST user/set/의 주소로 사용자의 아이디와 프로필 사진 데이터를 백 엔드로 전송하면, 서버는 프로필 사진 데이터를 서버에 저장한 이후 User Info 테이블의 Profile값에 저장한 사진의 경로를 수정한다. 해당 과정이 성공적으로 처리되었을 경우 Status 값 'OK"를 반환 해준다. SetProfile API의 처리과정은 그림 34에서 확인할 수 있다.

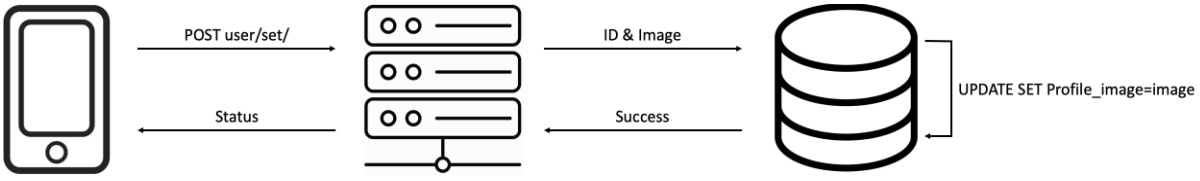

그림 37 SetProfile API

d) ResetProfile API

사용자의 프로필 사진을 제거하는 기능을 수행하기 위한 API로 프론트 엔드에서 POST user/reset/의 주소로 사용자의 아이디를 백 엔드로 전송하면, 서버는 저장된 프로필사진을 삭제한 이후 User Info 테이블의 Profile값을 NULL로 수정한다. 해당 과정이 성공적으로 처리되었을 경우 Status 값 'OK"를 반환 해준다. ResetProfile API의 처리과정은 그림 35에서 확인할 수 있다.

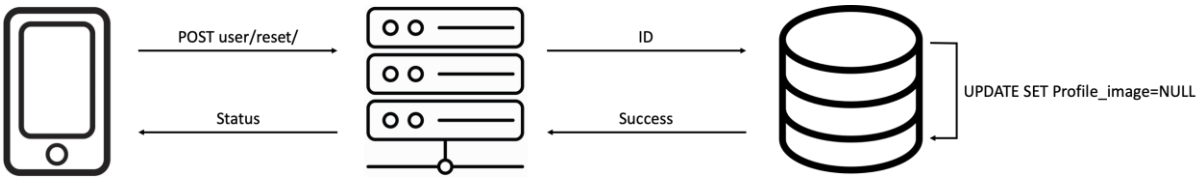

그림 38 ResetProfile API

소스코드

프론트 엔드 - https://github.com/jjm7307/HairChanger

백 엔드 - https://github.com/skygun88/HairChangerBackend