안드로이드 정리 설명서

18년 1월 28일

내용

[앱 작동순서 3](#_Toc504926690)

[안드로이드 클래스들 6](#_Toc504926691)

[안드로이드 xml들 7](#_Toc504926692)

[주요 페이지 – 메인 페이지 8](#_Toc504926693)

[주요 페이지 – 맵 페이지 9](#_Toc504926694)

[주요 페이지 – 맵 페이지 11](#_Toc504926695)

[- LocationRequestor 11](#_Toc504926696)

[주요 페이지 – 맵 페이지 13](#_Toc504926697)

[- serverConnectorST 13](#_Toc504926698)

[헨들리 코드 16](#_Toc504926699)

[통신코드 18](#_Toc504926700)

# 앱 작동순서

메인 페이지 -> 로그인 페이지 -> 지도 페이지

-> 회원가입 페이지

메인페이지 :

메인페이지는 처음 로딩시 필요한 기능들이 들어갈 예정이다.

현제 들어가 있는 기능은, 안드로이드 기기 내에 보관되어있는 sid를 뽑아와,

세션 유무를 확인하는 기능이다.

세션 유무 확인에 실패하면(sid 비일치, sid없음), 로그인 페이지로 넘어간다.

세션 유무 확인에 성공하면, 곧바로 지도 페이지로 넘어간다.

로그인페이지 :

이 페이지는 로그인을 위해서, 로드 되면서 안드로이드 앱 내에 저장되어있는

userId, userPw를 불러온다, 값이 없는경우에는 그냥 빈 창이 뜬다.

Id pw 를 입력하고 로그인시, 로그인 요청을 보낸다,

로그인 성공시 맵으로 넘어가며, 새롭게 발급받은 sid를 안드로이드 기기 내에 저장한다.

실패시 실패 매세지를 띄운다.

회원가입을 누를 경우 회원가입 창으로 이동한다.

회원가입 페이지 : 해당페이지는 아이디 비번 이름 전화번호를 우선적으로요구한다,

아직 자세한 기능은 없으며, 필수 입력정보가 만족되면 서버로 회원가입을 요청하며,

성공시 userId,userPw를 안드로이드 기기에 저장하고, 로그인 창으로 넘어간다,

지도페이지 :

우선 다음 지도를 세팅한 이후,

FusedMapLocation을 통하여, 현제 위치를 요청한다.

mapLocation의 결과를 반환하면, 맵에 내 위치를 받아온 위치로 업데이트 한다.

이후, 내위치,내 아이디 를 서버로 넘겨 주변의 거래,내 활동중인거래 를 요청합니다.

돌려받은 결과는 주변의 거래별로 list에 담고, 지도에 출력한다.

다음, 내 활동중인거래를 리스트에 담으며 우측의 드라이버 RecyclerView업데이트를 한다.

다음, 내 활동중인 거래중 상태가 4(입금완료) 인 거래를 추려내여,

해당 거래들의 픽업 대기시간을 서버에 요청한다.

서버에서 돌려받은 픽업대기시간을 현 시간과 비교하여,

픽업 예정시간 보다 5분전 인 상황이면,

곧바로 빠른 위치제공을 위한 스레드를 작동시키며, 해당 거래들을 리스트에 담아,

확인한다. 아직 픽업 예정시간이 5분 이상인 거래건들은, 타이머를 맞추어 주어,

5분 이하가 되면, 상기 서술한 스레드를 작동시키게끔 한다.

빠른 위치제공 스레드는 해당하는 거래번호와, 내 위치를 담아 서버에 요청하게 된다.

돌려받은 결과값은 각 거래에 해당하는 드라이버의 위치정보를 돌려받게 된다.

돌려받은 드라이버의 위치정보와 거래상태번호 는,

우선 상기에 설명되었던 드라이버RecyclerView에 각 드라이버의 위치정보를 업데이트해 주며,

다음, 받은 거래상태번호가 4가 아닌지 검사를 한다,

만일 4가 아닌 상황(5, 상품 픽업 완료) 이 되면,

빠른 위치제공 스레드의 작동 필요가 없어지므로, 해당 리스트의 값을 제거해 준다.

이로써 다음 빠른 위치제공 스레드에서는 거래번호가 4가 아닌,

그 거래건의 요청은 사라지게 된다. 모든 빠른 위치제공 필요 거래가 사라지면,

스레드는 종료한다.

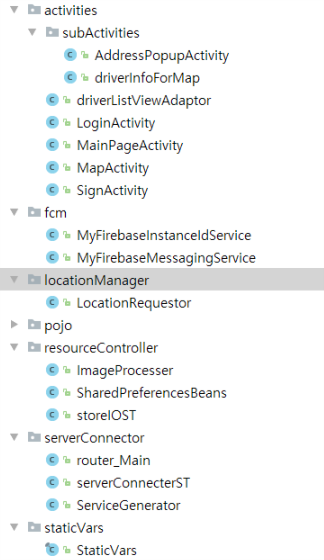
지도에서 드라이버RecyclerView를 클릭하게되면,

해당하는 드라이버의 위치로 맵poiitem을 생성하고, 해당 좌료포 지도를 이동시킨다,

이후 하단에서 해당 드라이버의 자세한 인적사항을 보는 창을 띄우며,

드라이버의 기본정보를 서버로부터 보내어, 자세한 인적사항을 받아 입력 시킨다.

# 안드로이드 클래스들



현제 정리된 class파일들.

Activity 하위 클래스는 말 그대로 activity를 관리하는 클래스들이 모여있다.

Fmc 폴더는 fmc message를 위해 필요한 서비스이며, 추후 추가 삭제가 일어날 수 있다

locationManager 폴더는 현제 기기의 위치를 알아내기 위한 기능들이 있는 파일이다.

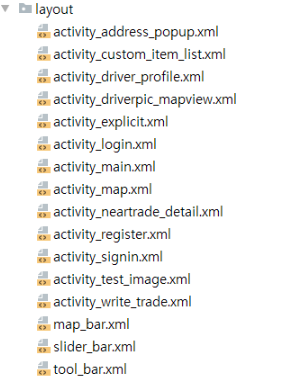
Pojo 폴더는 json관리룰 위한 파일들 뿐이다, 단일로써 보면 이해가 가지 않기 때문에, 통신이나 activity를 보며 같이 읽어주면 이해가 빠를 것.

resourceCOntroller는 안드로이드에 리소스를 관리하는 폴더이며, 현제는 이미지 저장, 이미지 제처리 기능만 존재한다

serverConnector의 하위 폴더들은 모두 통신을 위한 필수 기능들 이다.

staticVars 는 static값들을 보관한다.

# 안드로이드 xml들



이부분은 정리가 안되어 있기때문에 없거나 쓰지 않는 기능들이 다수 존재한다.

대부분은 적혀있는대로 주요 페이지의 엑티비티를 담당하며,

Custom\_item\_list는 주변 거래 객체를 클릭할시 나오는 페이지 인데, 아직까지 부실하기 짝이없다.

Driver\_profile는 내 활동 거래중인 드라이버를 클릭했을때, 나오는 간편 정보를 표시하는 xml

Driverpic\_mapview는 드라이버RecyclerView에 들어가는 드라이버 이미지이다.

나머지는 같이 만들었거나, 쓰지않거나 도입 대기중인 엑티비티들 이다.

# 주요 페이지 – 메인 페이지

Activity/MainPageActivity.class 에서 앱의 시작을 담당한다.

매우 간단하게 onCreate onStart checkSession 세 함수뿐이다.

SharedPreferencesBeans는 안드로이드에서 제공하는 기기 내의 자정소인데, 해당 변수들은 앱의 초기화, 삭제 후 제설치,강제 값 제거 전까지 남아있다. 그러므로 이곳에 인증 관련 정보를 남겨두면 손쉽게 관리가 된다.

checkSession함수는, 바로 sharedPreferencesBeans로부터 값을 가져와 서버에서 확인을 하는 기능이다.

스레드 부분에서 조금 햇깔릴건데, Thread객체를 생성할 때, runnable객체를 같이 생성한다, 하지만 기본 runnable를 추가하지 않고, 미리 선언되어있는 serverConnector/serverConnecterST 를 추가한다,

이후 run함수에 checkSessionAlive를 호출한다고 적어놓았는데, 해당 함수는 serverConnectorST안에 들어가 있다.

후, serverConnectorST에서 결과를 myHandler로 메시지 형식으로 보내게 된다.

이 헨들러는 메인 헨들러로써, mainPageActivity 에 implement 되어있다.

그냥 클래스 내부에 handler를 선언해도 되지만, 이전에 설명했던 이유로

(<https://juicybrainjello.blogspot.kr/2018/01/android-using-mainhandler-cause-memory.html>)

이너 클래스로 weakReference 타입으로 클래스를 만든다.

앞으로 나올 모든 Activity에서는 이런 방식으로 메인 헨들러를 implements한다.

보낸 message는 handleMessage 함수에서 처리한다.

받은 메시지는 항상 key:”result” value type int 로 결과를 반환한다.

헨들러 메시지 종류와 종류는 해당 페이지를 참조한다.

받은 메시지에 따라 성공/실패여부를 따지며, 그에 따른 결과를 세팅 해 준다.

# 주요 페이지 – 맵 페이지

위에 이미 한차례 가볍게 설명을 했지만, 핵심 기능은 다 들어가 있다.

제일 처음 사용하는 LocationRequestor 를 우선 알아볼 것이다.

이후, 다음에서 제공하는 MapView를 이용하여, 선언하고, LocationRequestor의 초기화 설정의 결과에 따라 내 위치를 업데이트 하게된다, 이로써 MapVIew 업데이트도 끝이나게 된다, 하단에서 LocationRequestor를 읽으면 나올, 511 코드 반환이 일어나, Map의 myHandler에 메시지가 전송된다면,

업데이트된 해당 좌표를 들고, 서버에 통신을 시작하게 된다.

mapActivity에 이너클래스인 myHandler의 handleMEssage, switch문의 511 case를 참조하면,

서버 커넥터에 통신을 하는 스레드를 볼수 있다. serverConnectorST에 선언되어있는 mapRefresh 함수에, 방금 얻어낸 따끈따끈한 좌표를 입력해 실행시키게 된다.

serverConnectorST역시 이제 아래에 설명이 될것이다.

서버로통신이 끝나서 정보를 받아오게되면 myHandler의 handleMessage switch문에 521 로 결과가 들어온다

그럼 받아온 pojo를 이용해

주변 거래들을 지도에 뿌려주는 setNearTrade 함수를 작동시키고,

내 활동 거래를 관리하는 setActiveDriverStatus 함수를 작동시킨다.

setNearTrade는 복잡해 보이나, nearTradePoiItem 리스트에 신규 거래를 담고, 해당 리스트의 모든 거래를 지도에 띄울 뿐이다.

이건 다음맵의 이해가 있으면 쉽다.

setActiveDriverStatus는 조금 복잡하다.

최대한 쉽게 하기 위해 주석을 달았다.

기본적인 작동구조는 위와 동일한데, 활동 드라이버의 이미지들이 우측에 뜨는 RecyclerView들도 관리하기 때문에 부득이하게 리스트가 여러 개 쓰이게 된다, 추후 최적화를 통해 한 개 내지 두개를 줄일 수 있으리라 판단하고 있다.

이곳에서 거래번호가 4번인 것들을 새롭게 updateRepeaterTradeList에 추가된다.

setActiveDriverStatus에서 일련의 작업들을 거쳐서 updateRepeaterTradeList를 추가 수정 삭제 비우기 함수들이 필요에 따라 작동되게 한다.

그리고 setActiveDriverStatus함수의 작업이 끝이나면서, requestTradeListScadule 함수를 실행한다.

이는 새롭게 상태번호가 4인 거래를 찾았을 때 해당 거래들을 모아 서버로 픽업 시간을 요청하는 함수이다.

위와 동일하게 serverConnectorST 를 통해 서버와 통신한다.

돌아오는 헨들러코드는 723이며, 돌아온 결과들은(역시 json으로 여러 개가 올 수 있다) addTradeScadule 함수를 부른다.

Pojo를 확인하면 알다싶이 각 거래의 상태번호, 시간이 나오는데, addTradeScadule에서는

unixtimestamp 타입으로 돌아온 시간을 현제시간과 비교하여, 5분 이하인지 이상인지 확인을 하는 함수이다. 상기 설명대로 시간이 남아있으면 Timer를 돌리고, 아닌경우 바로 스레드로 serverConnectorST 로 통신을 한다.

조금 다른점은 이번 serverConnectorST를 implements한 스레드는 updateRepeaterTradeList가 빌 때 까지 계속 반복한다.

while문으로 반복하는 것을 볼수 있다.

그럼 어떻게 이 리스트를 비우느냐면, 통신의 결과를 통해서 관리한다.

이 스레드의 통신 결과는 헨들로코드 731로 반환된다. updateDriverList 함수를 통해서 처리되는데,

이 함수는 드라이버 정보들이 담겨있는 myActiveTradeDriverList와 RecyclerView에 있는 드라이버 정보를 함께 수정하는 함수이다.

이후 각 거래의 상태번호가 4가 맞는지 확인을 하는데(checkThisTradeStatusisFour) 만일 4가 아니라면 updateReteaterTradeList에 해당 거래를 빼는 것 이다.

현제까지 핵심 기능은 이정도 설명으로 끝이 날 것 같다.

추가 작업에 의해 역시 핵심인 맵 페이지또한 많이 바뀔수 있다.

아래부터는 locationRequestor와 serverConnectorST 에 대한 설명이 진행된다.

## 주요 페이지 – 맵 페이지

## - LocationRequestor

해당 엑티비티는 현제 내 위치를 업데이트 하기 위한 기능을 담당하는데

각각 내부 정의 runnable, 내부 헨들러가 존재하며,

생성자에서 기본 세팅들을 모두 끝낸다.

솔직히 이부분은 완벽하게 파악된게 아니기 때문에 의문점이 조금 존재한다

우선 createLocationCallback으로 콜백을 선언하고 정의한다,(헌데 아직까지 쓰이진 않았음)

그뒤, createLocationRequest 함수에서 실제로 위치를 요청하는 기능을 세팅한다,

여기에서 핵심적인 변수인

Interval, FastestInterval, Priority 를 설정하게 되는데, 이는 각각

업데이트 주기, 빠른 업데이트 주기, 위치 요청 수준 우선도 를 설정하는데,

기본적으로 업데이트 주기별로위치 업데이트를 갖가된다, 하지만 클라이언트에서 고의적으로 빠른 업데이트 요청을 할 때, 빠른 업데이트 주기 시간에 해당한다면, 업데이트를 해준다. 이 의미는 기본적으로 10초에 한번씩 위치업데이트를 하지만, 필요에따라 초당 한번씩 요청을 할 때, 이전버전에서는 아무리 요청을 해도, 실제 업데이트 주기는 10초이기 때문에 0초부터 9초까지는 동일한 위치를 반환할 뿐이다, 하지만 빠른 주기 시간이 2초로 설정되어있다면 2 4 6 8 10초에는 업데이트가 된 위치가 반환된다는 말이다.

위의 두 시간을 알맞게 조율해서, 추후 최적화 된 시간을 설정하는 것이 성능면에서 유리할 것.

마지막으로 위치 요청 수준은 0 1 2 3 래벨이 있는데,

0 : no power, 그 어떠한 위치요청 기술을 이용하지 않음, 주로 3rd party앱을 통해 업데이트된 위치를 가져올 때 씀.

1 : low power 최소한의 기술을 이용하여 위치를 업데이트하므로, 배터리 성능을 최소한으로 하지만, 오차범위는 평균 2km까지도 가능하게 된다.

2 : balanced power 적절한 기술을 사용하여 위치를 업데이트하므로, 배터리도 눈에 띄이게 사용하는 반면, 오차 범위는 200m까지 가능하다

3 : max power, 모든 위치정보를 이용하여 위치를 뽑기 때문에, 평균적으로 20m까지 보장하나, 배터리 소모는 시간당 2%(단일 앱만 해당하는 계산)이라는 무시무시한 배터리 소모량을 자랑한다.

이기 때문에, 기본 상황에서는 2번, 빠른 위치업데이트를 요할땐 3번을 번갈아 쓰게 해야할것으로 판단 중 이다.

이제 buildLocationSettingRequest 를 통해 아까 입력된 세팅값들ㅇ르 적용시킨다.

그뒤 getLastLocationForSendHandleMsg 함수를 통해서, 실제로 위치를 요청을 시작한다.

결과를 옳게 받는다면, 헨들러코드 515를 맵 헨들러에 반환한다.

이렇게 처음 위치 세팅이 끝이났으며, 맵 엑티비티에서 주기적으로 계속 내 위치를 요청하게 되는데, 이 작업은 아까 만든 내부 스레드를 통해 작동시킨다. 이너 클래스인 LocationUpdater를 참조하여 보면,

내 위치를 뽑아내어, 맵 엑티비티의 헨들러로 511 코드와 함께 결과값을 반환하는 스데르가 선언되어있음을 볼 수 있다.

해당 스레드를 관리하는 것은, LocationRequestor클래스의 메인 헨들러이다, LocationHandler 이너 클래스를 보게되면, 단 하나의 handleMessage문이 있는데, 이는 방금 만들어진 스레드를 실행시키며, 멈추게 하기위한 flag를 받지 않을시, 정해진 시간뒤에, 다시 해당 헨들러를 작동시키게 만든다,

언제는 업데이트 주기, 업데이트 수준, 업데이트 중지 등을 위해 이렇게 복잡하게 설계되어있다.

## 주요 페이지 – 맵 페이지

## - serverConnectorST

실은 serverConnectorST는 범용적으로 모든 서버통신에 쓰이므로, 맵 페이지에만 쓰이는 것은 아니다.

하지만 이 문서에서 기본적인 작동구조를 설명할것이고, 다른 통신들도 모드 같은 방식이기 때문에, 두번 이상 설명은 하지 않을 것 이다.

상기에서 서술했다싶이 맵 업데이트를 위해 우선 이 Runnable을 불러오게 된다, 생성자를 보면

의외로 간단하게 해당 Runnable을 불러온 엑티비티의 Context와, mainHandler정보를 받아온다,

이는 통신 결과를 돌려줄 대상을 명확하게 하기 위해서 이다.

이로써 세팅은 끝이난다. 상기의 map page에서는 스레드를 실행하고 mapRefresh 함수를 실행한다고 한다,

우선 run 메소드를 확인한다, 디버그용 코드 한줄뿐이다, 그냥 스레드를 실행할 때 run함수에서만 실행되기 때문에, 필수로 만들어져 있을 뿐이다. 그럼 mapRefresh메소드로 가본다. 이제 좀 복잡하다.

여기서부터 후술되는 기술은 Retrofit 2.0 을 이용한 서버통신 부분이다. 더 자세한 혹은 응용은 Retrofit 공식문서를 참조하는 것이 빠르다. 물론 해당 문서를 읽어도 http통신에 대한 기초지식이 없으면 매우 힘들다.. 나도 읽을때마다 뭔 개소린가 싶다.

우선 router\_Main에 MapInterface 를 service 라는 변수로 지목한다,

이는 serverConnector/router\_Main 클래스를 참조한다. 한번 들어가 본다.

해당 클래스는 통신을 위한 기본 세팅들을 모아둔 프리셋 정도로 생각하면 된다, 올바른 용어가 맞는지는 아직도 의문이지만 나는 이 클래스를 라우터라 명하고 있다.

해당 클래스는 죄다 인터페이스 뿐이다. 우선 mapRefresh에 쓰일 mapInterface를 읽어본다.

두개의 Call 을 명시했으며, 한 개 내지 여러 개의 어노테이션들에 범벅이 되어있다.

통신을 보낼때는 여러가지가 있지만, 가장 크게 보자면 보내는 정보 자체를 크게

Header와 body 로 나눈다고 생각한다.

Header에는 보통 request를 하는 부분에서의 정보가 들어가 있다.

우리는 header에 아주 처음에 언급했던 sid값을 보내는 것 이외에는 여타 통신과 크게 다른점은 존재하지 않는다, sid는 추후 다시 언급될것이다.

header에 들어가 있는 정보중 하나는 response로 받아들일 body에 들어가 있는 정보가 무엇인지에 대한 설명이 있다,

해당 설명이 없다면, body에 담긴 데이터가 무엇인지 알수 없으며, 클라이언트는 정체모를 데이터를 받고 어떻게 처리해야 할지 모른체로 에러를 토해낼것 이다.

그렇기에 mapInterface의 어노테이션에 @Headrs 에 response로 body가 무슨 타입인지 명시를 한다.

우리가 지금 당장 알아보는 Call은 getMapInfo 라는 녀석인데, 이놈의 헤더 어노테이션은 json이라고 명시가 되어있다. 이말은 response로 json타입을 받는다는 것이다.

다름 어노테이션을 읽어보면, @FormUrlEncode 라고 하는데, 이는 request로 보내는 body가 무엇인지 명시해 주는 것 이다, 이또한 명시하지 않으면, 서버는 혼돈 파괴 망가를 일으키며 엉뚱한 값을 받고 터지거나, 미친 값을 response로 보내게 될것이다.

다음으로는 통신 메소드를 정의하고 있다. @POST(‘/url’) 은 POST 타입으로 /url 이라는 주소로 보내겠다는 의미이다.

이제 Call을 읽어본다.

Call의 타입을 꺽쇠에 지정되어있다 우리가 보고있는 것은 <MapRepo> 로 되어있는데, 이는 response로 받은 데이터를 어디에 담을것인가에 대한 정보이다, 일반적인 값들만 받는다면 <Void>여도 된다, 하지만 지금처럼 무언가 크거나 특별한 자료의 경우 pojo 데이터를 받아 추가적인 변환이 필요하다.

그래서 우리는 MapRepo라는 pojo에 받은 json값을 넣겠다고 명시하는 것 이다.

getMapInfo는 단순히 이 Call의 변수 이름이다. 그 뒤 괄호는 request로 보낼 값들을 명시하는 부분이다.

@Field로 body에 “locationLat’ 이라는 변수에 담을 String형 locationLat 이다.

처음 필드 어노테이션 옆의 “locationLat”은 실제 전송할 데이트에 붙는 key값 이름이며, 마지막에 붙는 변수명은 우리가 보낼때에 받을 변수 명을 명시해주는 것 이다.

어려워도 몇번 따라치면 돌아가는 꼴을 이해 할 수 있을 것 이다.

이제 기본적인 route를 다 보았다. 다시 serverConnetorST 에 mapRefresh 메소드로 돌아가자

이제 service 변수명이 무엇인지 알았다, 초기화를 보자, ServiceGenerator를 생성하여, createService 변수를 부른다..

serverConnector / ServiceGenerator 클래스로 다시 가보자.

끔찍한 변수들이 잔뜩있는데, 핵심은 아까 말한 Retrofit 2.0 그리고 추가적으로 Okhttp 3.1 의 선언을 하는 클래스들 일 뿐이다. 모든 서비스들의 선언을 스레드에 쓸 필요는 없기 때문에 분리시켜놓은것 뿐이다. 다시 돌아가자.

createService의 인자값으로 아까 우리가 쓰기로 한 MapInterface 를 사용한다고 명시되어있다. 이제 서비스는 선언과 초기화가 끝이났다. 통신을 시작한다.

다음줄은 Call<MapRepo> call; 인데 MapInterface와 동일하다는 것을 알수 있을 것 이다.

Interface일뿐이니 실제 구현은 아니기에, 여기서 실체화를 하기 때문에 또 써주는 것 이다.

이제 이 call의 초기화이다. 선언한 service에서 MapInterface의 getMapInfo 를 부르며, 인자값으로 우리가 보낼 좌표를 입력해 놓았다. 이제 Interface를 써서 call을 준비했다.

Call을 callback을 만들어 명령 큐에 집어넣는다. Call이 하나뿐이라면 그냥 작동 되겠지만, 나중에 call 요청이 많게 되면, 큐의 순서에 따라 call들이 순차적으로 작동되게 하기위해 retrofit에서 설계 해 놓았다.

Callback을 쓴다고 했으니, implement할 메소드들이 있다. 자연스럽게 코드를 치면

onResponse 와 onFailure 가 나오게 되는데, 서버에서 실제 통신이 성공되면(결과는 어찌됬든) onResponse가 작동한다, 반면 타임아웃시간(ServiceGenerator에서 설정해둔)까지 반응이 없으면, onFailure 기 직동된다.

돌아온 값은 response 에 들어가 있으며, 우리는 위에서 response로 json을 받으며, 받은값을 MapRepo에 넣는다고 선언했었다.

MapRepo를 생성하여, response.body를 넣어주면, 우리는 json을 MapRepo에 넣게 된 것이 된다.

이제 이 통신을 요청한 클래스로 값을 돌려보낼 것 이다.

serverConnectorST를 선언하면서 알고있는 mHandler 에게 Message 로 MapRepo를 보내준다.

그리고 내가 정해놓은 약속인, 통신 결과를 헨들러로 보낼떼, “result” 코드를 반드시 모든 통신마다 작성하여 보내게 했다.

헨들러 코드는 다른 페이지에서 설명한다.

여기까지 통신방법의 설명이 끝이난다. 매우 복잡해 보이지만, 나름대로 규칙과 편의를 위한 모듈화가 진행 되었기 때문에, 두어번 보고 따라치면 손쉽게 통신을 설계할 수 있다.

# 헨들리 코드 :

헨들러코드는 통신의 결과 등을 받아 다른 엑티비티에서 처리를 쉽게 하기 위해 직접 만든 상태 코드들 이다. 계속 수정중이며 언제든지 바뀔 수 있다는점을 명시해야 한다.

1. 100 : 회원정보 요청
   1. 110 : 1차 회원가입
      1. 111 : 회원가입 성공
      2. 112 : 회원가입 실패, 중복된 아이디
      3. 113 : 회원가입 실패 중복된 전화번호
2. 200 : 로그인 관련 요청
   1. 211: 로그인 성공
   2. 212 : 아이디, 혹은 비밀번호 오류
3. 3xx : 이미지 프로세싱 요청들
   1. 31x : 내 이미지 업데이트 요청
      1. 311 : 내 이미지 업데이트 성공
      2. 312 : 내 이미지 업데이트 실패
   2. 32x : 내 이미지 다운로드 요청
      1. 321 : 내 이미지 다운로드 성공
      2. 322 : 내 이미지 다운로드 실패
4. 4xx : 세센 관련 요청들
   1. 401 : 현제 계정의 세션이 살아있음
   2. 402 : 현제 계정의 세션이 만료됨
5. 5xx : 맵 관련 요청들
   1. 51x : 내 위치 요청
      1. 511 : 내 위치 업데이트 성공
      2. 512 : 내 위치 업데이트 실패
      3. 513 : 내 위치 업데이트 실패 requestLocationUpdate 캐치
      4. 514 : 내 위치 업데이트 실패 requestLocationSetting 오류
      5. 515 : 맵 초기화를 위한 센터값 업데이트 성공
      6. 516: 맵 초기화를 위한 센터값 업데이트 실패
   2. 52x : 메인 지도기능 요청
      1. 521 : 맵 요청 성공
      2. 522 : 맵 요청 실패 서버로부터 200코드를 받지 못함
6. 6xx : 드라이버 관련
   1. 61x : 드라이버 약식 정보 요청
      1. 611 : 드라이버 약식 정보 받아옴
      2. 612 : 드라이버 약식 정보 얻기 실패(없는 드라이버)
      3. 613 : 드라이버 약식 정보 얻기 실패(알수없는 이유)
7. 7xx : 거래 관련
   1. 71x :
   2. 72x : 거래건의 시간 요청
      1. 721 : 거래건의 상품 픽업 정보를 성공적으로 받아옴.
      2. 722 : 거래건의 상품 픽업 정보를 받아오는대에 실패함.
      3. 723 : 거래 상품 픽업 정보 리스트를 성공적으로 받아옴
      4. 724 : 거래 상품 픽업 정보 리스트를 받아오는대에 실패함.
   3. 73x : 특정한 거래의 드라이버 위치와 상태코드 요청
      1. 731 : 성공적으로 받아옴
      2. 732 : 받아오는 대에 실패함
8. 9xxx : 오류코드들
   1. 9110 : 1차 회원가엡이서 예기지 못한 에러코드 발생.
   2. 9210 : 로그인 시도 중 예기치 못한 에러코드 발생.
   3. 9310 : 내 이미지 전송 실패 : file read failed
   4. 9311: 내 이미지 전송 실패 : io exception
   5. 9400 : 세션확인 작업중 예기치 못한 에러코드 발생.
      1. 9401 : 세션 만료
   6. 9001: 정해져 있지 않는 비정상 상태코드
   7. 9999 : 서버와 통신자체가 실패함.

# 통신코드

이는 www에서 규약한 통신코드인데, serverConnectorST 에서 나타나는 것을 볼 수 있다.

이것떄문에 헨들러코드와 통신코드가 햇깔릴까 싶어 한번 더 명시하고, 자주 쓰이는 코드만 정리하여 설명한다. 다시한번 이 코드는 오직 serverConnectorST의 response.code에서만 등장한다.

혼동하지 말자.

200 ok

201 created

202 accepted

203 non authoritative account

204 non content

400 bad request

401 non authorized

403 forbidden

404 not found

408 timeout

502 server closed

504 gateway timeout