CRADLE VSA Application

Android Code Documentation

Sachin Raturi
301265401
Cmpt 415
Spring 2020

Abstract

The **Cradle VSA** (Vital Signs Alert) is a portable medical device that can read a patient's blood pressure and heart rate, mainly used to detect abnormalities during pregnancy. The Cradle VSA application can record the readings taken by the Cradle VSA device and upload it to the server. Our application is designed for remote areas with an unstable internet connection; thus, so we implemented an SMS feature.

This document takes a deep dive into the code, explaining various aspects of the applications. The document is created with the new developers in mind who wish to work on the project in the future. The code documentation shall provide background information on why the code is structured the way it is.

Models

The application uses various models such as Reading, Patient, Settings, etc. Although some models are relatively simple, others require some explanations.

Reading

The Reading model is one of the most critical and sophisticated classes. It stores the necessary information regarding the readings, the patient (**More in Database section**), urine test result, etc. It also contains other information such as *ZonedDateTime* of when the cradle reading was taken, uploaded to the server, device information, app version, etc. Some of the information, such as device information, was initially added to upload the debug data to the server. However, since moving towards Firebase, most of the debugging fields are now obsolete. Furthermore, the *Reading* class also contains a static method to convert the model into JSONObject. This method is used in the application to convert the java class into JSON format before sending it to the server.

Settings

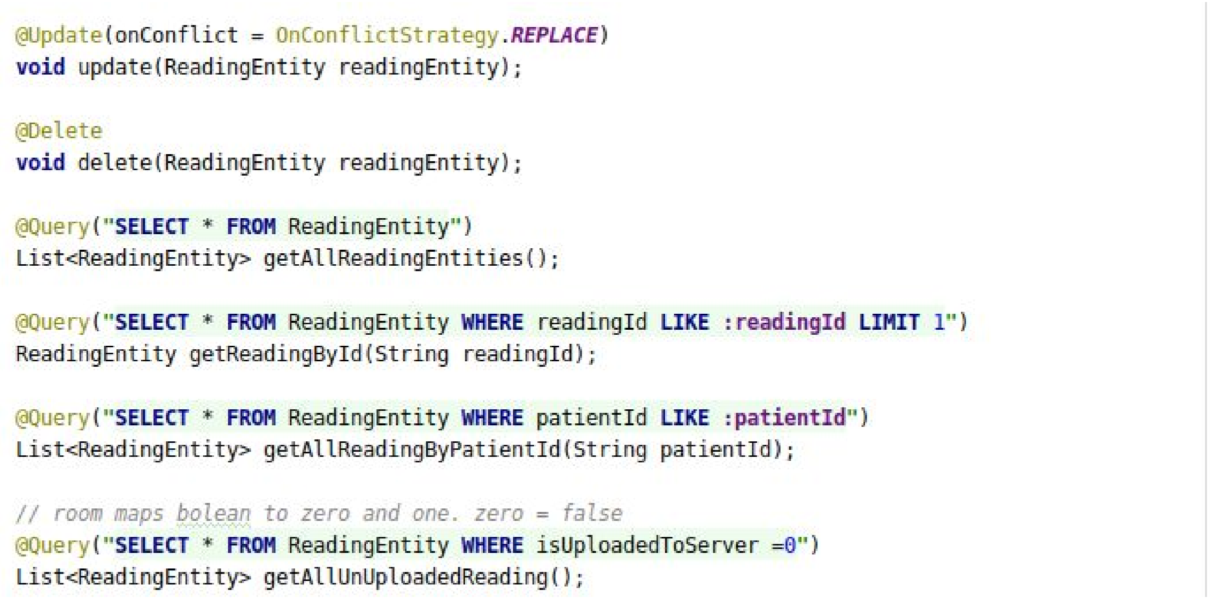
The *Settings* class contains all the URLs for all the server API calls. It also contains sharedPref data that can be updated on the settings page in the application. However, currently, most of the sharedPref settings are ignored by the app.

Database

The application uses the Android Room database, which is an abstraction over the sqlite database. The database has one table: Reading table. The reading table has three columns: reading_data, patient_id, and readingId. The readingId is a UUID string. The reading model has data such as *ZonedDateTime* that is inconvenient to save into its column, so we decided to keep it as a JSON string. This decision saved much overhead code from converting in between classes and SQLite data types.

Room saves us from boilerplate code to convert from java object to sqlite column and vice versa. Room also provides simple migrations and is designed to work with live data, which is useful in case of thousands of patients. Furthermore, Room by default does not allow queries on the main thread so that we do not block the UI with large queries.

The *ReadingEntity* class represents the reading table. The *DaoAccess* interface has all the queries for the database such as *getAllReading(), getReadingsByPatientId(), etc.*



```
@Update(onConflict = OnConflictStrategy.REPLACE)
void update(ReadingEntity readingEntity);

@Delete
void delete(ReadingEntity readingEntity);

@Query("SELECT * FROM ReadingEntity")
List<ReadingEntity> getAllReadingEntities();

@Query("SELECT * FROM ReadingEntity WHERE readingId LIKE :readingId LIMIT 1")
ReadingEntity getReadingById(String readingId);

@Query("SELECT * FROM ReadingEntity WHERE patientId LIKE :patientId")
List<ReadingEntity> getAllReadingByPatientId(String patientId);

// room maps bolean to zero and one. zero = false
@Query("SELECT * FROM ReadingEntity WHERE isUploadedToServer =0")
List<ReadingEntity> getAllUnUploadedReading();
```

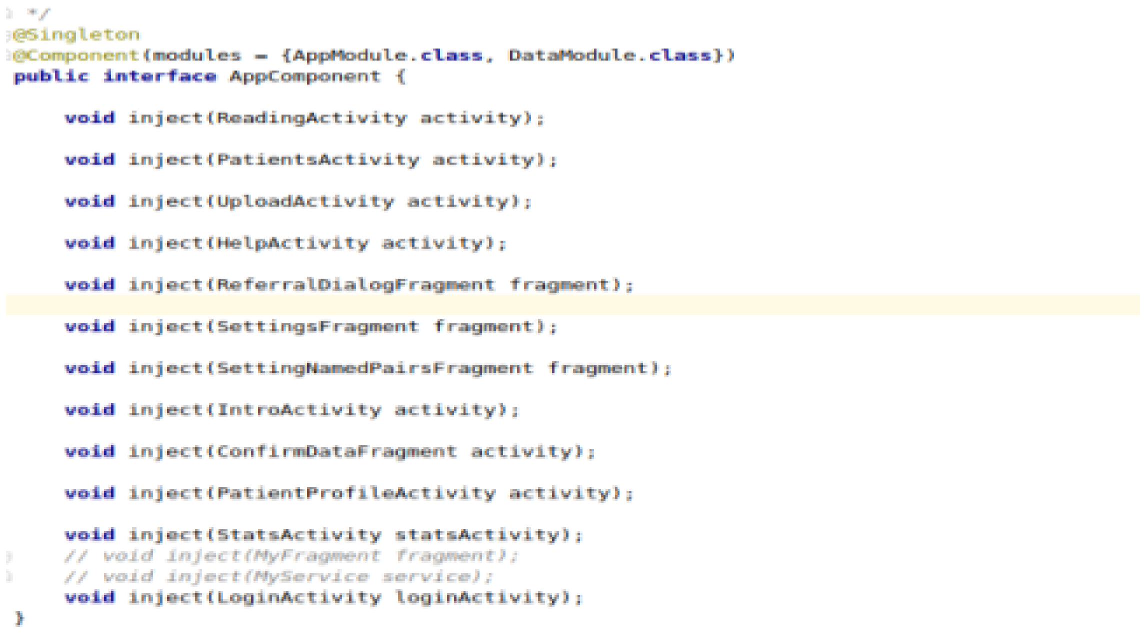
cradlemobile/app/src/main/java/com/cradle/neptune/database/DaoAccess.java

Initially, the Patient model did not exist, so all the patient information was part of the Reading model. However, the *Patient* class was later extracted out of the *Reading* class and was put inside *Reading* class as a variable. This design is quite troublesome as the server has readings inside the patient model.

Dagger

Dagger is a dependency injection framework for android. The idea is to decouple class implementation from their dependencies. Dependency injection has many benefits, such as easily sharing a class in the application, better testability, etc.

You can specify the objects to inject throughout the application inside the *DataModule* class in the dagger package. The *AppComponent* interface inside the dagger package lets you declare all different activities and classes to inject the objects.

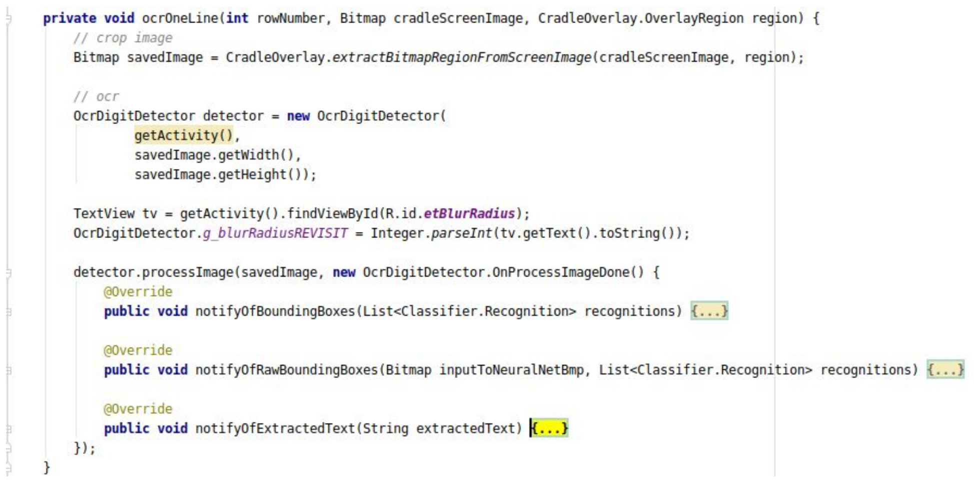


```
 */
@Singleton
@Component(modules = {AppModule.class, DataModule.class})
public interface AppComponent {

    void inject(ReadingActivity activity);

    void inject(PatientsActivity activity);

    void inject(UploadActivity activity);

    void inject(HelpActivity activity);

    void inject(ReferralDialogFragment fragment);

    void inject(SettingsFragment fragment);

    void inject(SettingNamedPairsFragment fragment);

    void inject(IntroActivity activity);

    void inject(ConfirmDataFragment activity);

    void inject(PatientProfileActivity activity);

    void inject(StatsActivity statsActivity);
    // void inject(MyFragment fragment);
    // void inject(MyService service);
    void inject(LoginActivity loginActivity);
}
```

cradlemobile/app/src/main/java/com/cradle/neptune/dagger/AppComponent.java

OCR

Optical Character Recognition recognizes text inside an image. In the Cradle VSA app, OCR is used to identify the numbers on the Cradle device when we take a picture of the reading. Tensorflow is used to train the text recognition model. Once the user has finished capturing the image, *CameraFragment* class creates multiple image files to extract the numbers from the images. In the *ConfirmFragment*, *the OcrDigitDetector class* processes the image and fills the extracted figures into the respective fields.



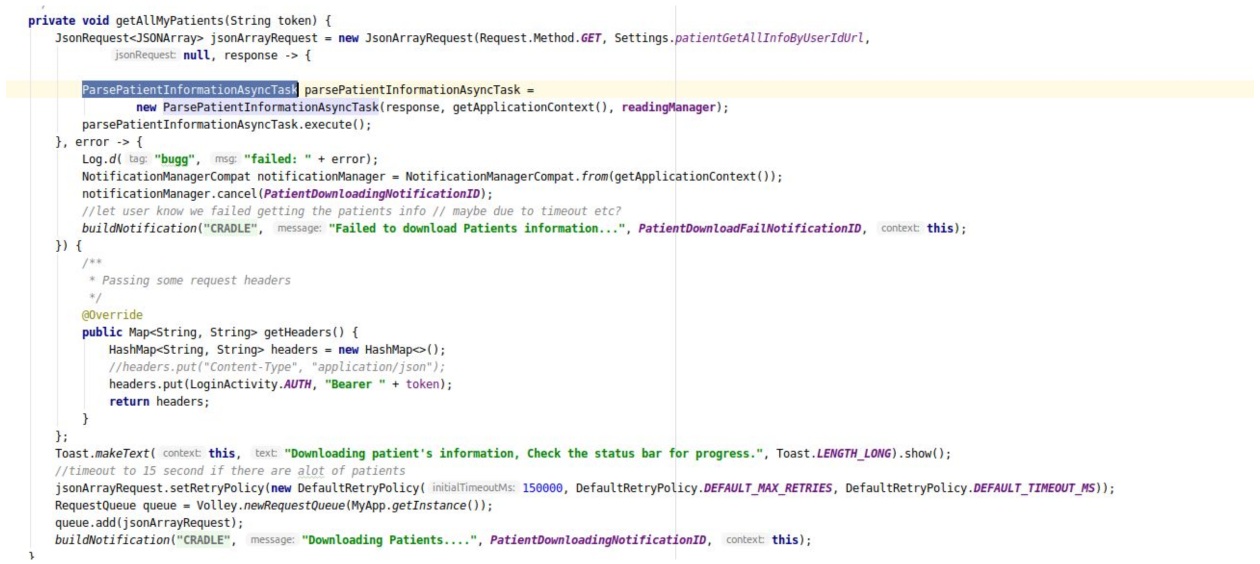
```
private void ocrOneLine(int rowNumber, Bitmap cradleScreenImage, CradleOverlay.OverlayRegion region) {
    // crop image
    Bitmap savedImage = CradleOverlay.extractBitmapRegionFromScreenImage(cradleScreenImage, region);

    // ocr
    OcrDigitDetector detector = new OcrDigitDetector(
            getActivity(),
            savedImage.getWidth(),
            savedImage.getHeight());

    TextView tv = getActivity().findViewById(R.id.etBlurRadius);
    OcrDigitDetector.g_blurRadiusREVISIT = Integer.parseInt(tv.getText().toString());

    detector.processImage(savedImage, new OcrDigitDetector.OnProcessImageDone() {
        @Override
        public void notifyOfBoundingBoxes(List<Classifier.Recognition> recognitions) {...}

        @Override
        public void notifyOfRawBoundingBoxes(Bitmap inputToNeuralNetBmp, List<Classifier.Recognition> recognitions) {...}

        @Override
        public void notifyOfExtractedText(String extractedText) {...}
    });
}
```

cradlemobile/app/src/main/java/com/cradle/neptune/view/ui/reading/ConfirmDataFragment.java

Activities (Screens)

Login Activity

LoginActivity is the first screen users see after downloading the application. The login screen uses volley to make a *JsonObjectRequest* to the server with the user credentials. When the activity receives a successful response, we use shared preferences to save the *userId* and *token* for future use. The activity then makes another API call to the server for all the patient's data who are related to the current user. One of the concerns was parsing a large number of patient entries. To tackle this, we used the *ParsePatientInformationAsyncTask* class to move all the work to a separate thread. Doing this frees our main UI thread, and therefore there is no lag. Since async runs on its thread, we are free to let it kill the current activity and move on to the next activity.



```
private void getAllMyPatients(String token) {
    JsonRequest<JSONArray> jsonArrayRequest = new JsonArrayRequest(Request.Method.GET, Settings.patientGetAllInfoByUserIdUrl,
            jsonRequest: null, response -> {

        ParsePatientInformationAsyncTask parsePatientInformationAsyncTask =
                new ParsePatientInformationAsyncTask(response, getApplicationContext(), readingManager);
        parsePatientInformationAsyncTask.execute();
    }, error -> {
        Log.d( tag: "bugg",  msg: "failed: " + error);
        NotificationManagerCompat notificationManager = NotificationManagerCompat.from(getApplicationContext());
        notificationManager.cancel(PatientDownloadingNotificationID);
        //let user know we failed getting the patients info // maybe due to timeout etc?
        buildNotification("CRADLE",  message: "Failed to download Patients information...", PatientDownloadFailNotificationID,  context: this);
    }) {
        /**
         * Passing some request headers
         */
        @Override
        public Map<String, String> getHeaders() {
            HashMap<String, String> headers = new HashMap<>();
            //headers.put("Content-Type", "application/json");
            headers.put(LoginActivity.AUTH, "Bearer " + token);
            return headers;
        }
    };
    Toast.makeText( context: this,  text: "Downloading patient's information, Check the status bar for progress.", Toast.LENGTH_LONG).show();
    //timeout to 15 second if there are alot of patients
    jsonArrayRequest.setRetryPolicy(new DefaultRetryPolicy( initialTimeoutMs: 150000, DefaultRetryPolicy.DEFAULT_MAX_RETRIES, DefaultRetryPolicy.DEFAULT_TIMEOUT_MS));
    RequestQueue queue = Volley.newRequestQueue(MyApp.getInstance());
    queue.add(jsonArrayRequest);
    buildNotification("CRADLE",  message: "Downloading Patients....", PatientDownloadingNotificationID,  context: this);
}
```

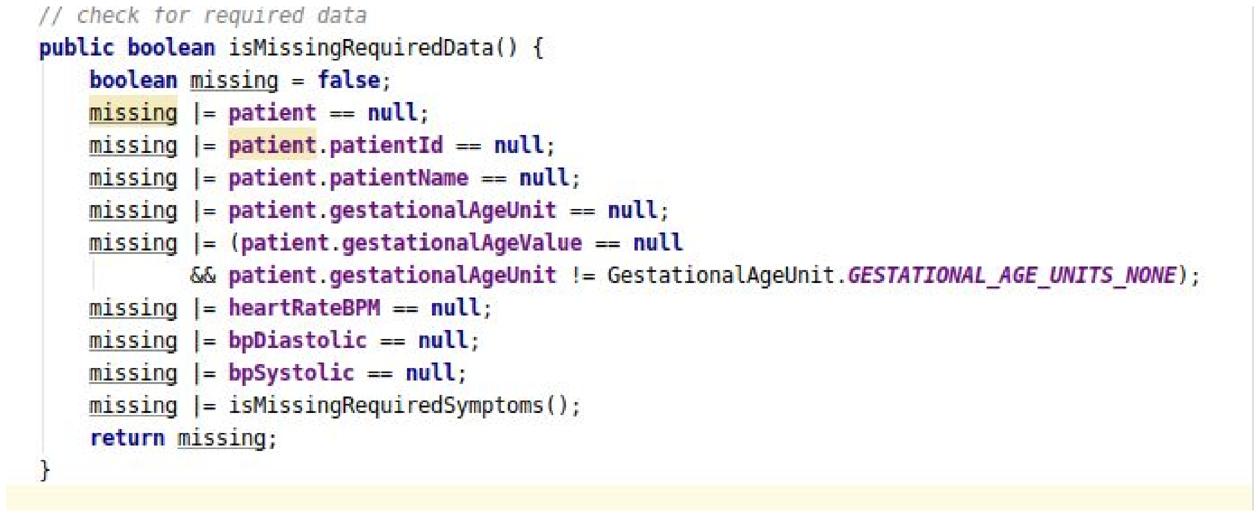
cradlemobile/app/src/main/java/com/cradle/neptune/view/LoginActivity.java

ReadingActivity

The reading activity uses tabs with fragments to collect patient's information, symptoms, Cradle VSA readings, etc. There are five fragments: *PatientInfoFragment ,SymptomsFragment, CameraFragment, ConfirmDataFragment, SummaryFragment.* All the fragments have overridden functions: *onBeingDisplayed()* and *onBeingHidden().* These functions save data to the *Reading* class as well as load data from the model class to display on the screen.

SummaryFragment

The SummaryFragment is the final fragment of the Reading activity. Here, we validate all the data, such as the patient's information, valid heartbeat number, etc. We cannot save the reading unless all the data is verified.

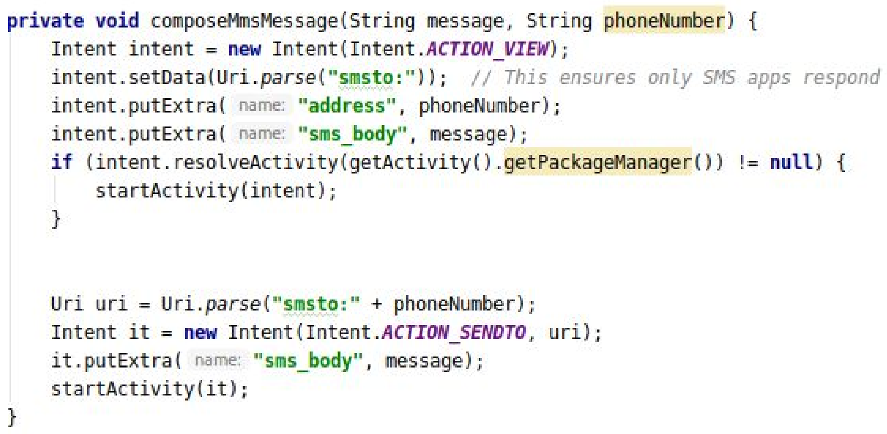


```
// check for required data
public boolean isMissingRequiredData() {
    boolean missing = false;
    missing |= patient == null;
    missing |= patient.patientId == null;
    missing |= patient.patientName == null;
    missing |= patient.gestationalAgeUnit == null;
    missing |= (patient.gestationalAgeValue == null
            && patient.gestationalAgeUnit != GestationalAgeUnit.GESTATIONAL_AGE_UNITS_NONE);
    missing |= heartRateBPM == null;
    missing |= bpDiastolic == null;
    missing |= bpSystolic == null;
    missing |= isMissingRequiredSymptoms();
    return missing;
}
```

cradlemobile/app/src/main/java/com/cradle/neptune/view/ui/reading/SummaryFragment.java

ReferralDialogFragment

From the SummaryFragment, ReferralDialogFragment launches to send a referral. We can send a referral either via a web API call or through SMS. Since google Google Play Store does not allow applications that automate sending SMS, we can only send an intent to launch the SMS application and fill in the data.

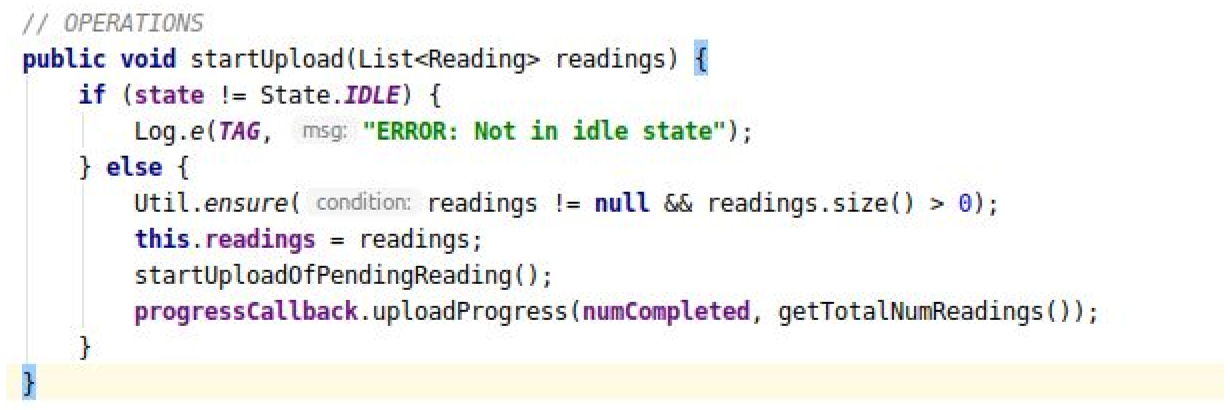


```
private void composeMmsMessage(String message, String phoneNumber) {
    Intent intent = new Intent(Intent.ACTION_VIEW);
    intent.setData(Uri.parse("smsto:"));  // This ensures only SMS apps respond
    intent.putExtra( name: "address", phoneNumber);
    intent.putExtra( name: "sms_body", message);
    if (intent.resolveActivity(getActivity().getPackageManager()) != null) {
        startActivity(intent);
    }


    Uri uri = Uri.parse("smsto:" + phoneNumber);
    Intent it = new Intent(Intent.ACTION_SENDTO, uri);
    it.putExtra( name: "sms_body", message);
    startActivity(it);
}
```

cradlemobile/app/src/main/java/com/cradle/neptune/view/ui/reading/ReferralDialogFragment.java

UploadActivity

The UploadActivity synchronizes data between the application and the server. The *MultiReadingUploader* class handles uploading multiple readings to the server. The *MultiReadingUploader* class also updates the client (UploadActivity) via callbacks.



```java
// OPERATIONS
public void startUpload(List<Reading> readings) {
    if (state != State.IDLE) {
        Log.e(TAG,  msg: "ERROR: Not in idle state");
    } else {
        Util.ensure( condition: readings != null && readings.size() > 0);
        this.readings = readings;
        startUploadOfPendingReading();
        progressCallback.uploadProgress(numCompleted, getTotalNumReadings());
    }
}
```

cradlemobile/app/src/main/java/com/cradle/neptune/view/ui/network_volley/MultiReadingUploader.java

Initially, A zip file that included reading (JSON data), OCR image, etc. was being encrypted and uploaded to the server. However, since a new server was created from scratch, the zip file was abandoned. We started sending the JSON data directly to the server using the *MultipartRequest* class. Furthermore, we upload the OCR images to the firebase storage.

Firebase

The application uses firebase crash analytics, performance monitor, test lab, and firebase storage. Crash analytics is instrumental in getting a notification about recent crashes and gives information such as log, type of phone, the operating system, etc. We use the firebase test lab to run a scripted or unscripted UI test with multiple phones. The firebase storage holds the OCR images used to train the TensorFlow model.

End note

Since we were initially given a default application. Most of the changes in the application structure were constrained by evaluating the ever changing customer's need and the time frame to work on the project. For example, Initially, we wanted to put the Reading class inside the Patients class. However due to the time constraint and the amount of refactoring needed, we decided to put *Patient* class inside the *Reading* class.