**UB Collecting**

**Developer’s Guide**

Prepared by Blake Cooper ([blakecoo@buffalo.edu](mailto:blakecoo@buffalo.edu))

Updated 12/15/2021

**Contents:**

1. [About the project](#AboutProject)
2. [Quick-start guide for developers](#QuickstartGuide)
3. [Understanding the app (overview)](#UnderstandingApp)
   1. [Basic operation](#Understanding_Basic)
   2. [Database/Data Types](#Understanding_Data)
4. [GitHub repository](#Repo)
   1. [Branches](#Repo_Branches)
   2. [Files in *master* branch](#Repo_Master)
5. [Codebase](#Codebase)
   1. [Location](#Codebase_Location)
   2. [Structure](#Codebase_Structure)
      1. [Application](#Codebase_Structure_App)
      2. [UI](#Codebase_Structure_UI)
      3. [Data](#Codebase_Structure_Data)
      4. [Utilities](#Codebase_Structure_Utils)
      5. [Activities](#Codebase_Structure_Activites)
      6. [Miscellaneous](#Codebase_Structure_Misc)
6. [Data synchronization](#Synchronization)
   1. [Requirements](#Synchronization_Reqs)
   2. [Firebase as synchronization tool](#Synchronization_Firebase)
   3. [Implementation overview](#Synchronization_Implementation)
   4. [Firebase credentials](#Synchronization_FirebaseCreds)
   5. [Further reading on Firebase](#Synchronization_FirebaseLinks)
   6. [Notes on abstraction and other synchronization options](#Synchronization_Abstraction)
7. [Suggestions for future work](#Suggestions)
   1. [Front-end/UI](#Suggestions_UI)
   2. [Back-end](#Suggestions_Backend)
   3. [Database](#Suggestions_Database)
   4. [Synchronization](#Suggestions_Synchronization)
8. [Glossary of common terms](#Glossary)

**1. About the project**

The [**KPAAM-CAM research project**](https://ubwp.buffalo.edu/kpaamcam/) began in 2014 with the long-term goal of exploring language change in a linguistically diverse region of Cameroon. The project seeks to answer high-level questions like:

* How much overlap is there in the linguistic repertoire of various villages in a region?
* What languages are likely to be used in a given village (or among a given people) based on their other characteristics?
* When a translation from a given language cannot be directly produced (due to lack of corpus data, for example), what strategies exist for producing a translation via an intermediary language?

The **UB Collecting Android app** is being developed to support KPAAM-CAM and similar research projects. The app provides a tool for researchers to elicit and collect diverse data from the field, by creating and using research instruments (questionnaires) from within the app.

This guide was written to orient new developers to the UB Collecting app (and the KPAAM-CAM research project it supports), by providing a reference for the existing codebase and functionality, as well as suggestions for future development.

The current version of the app has been almost exclusively developed by undergraduate research assistants, each of whom has worked for around six months to a year on the app. The main goal has been to reimplement a pilot version of the app successfully deployed using an earlier version of the FAIMS system. Because of this history, different parts of the app are in different stages of implementation, often reflecting different research priorities of different developers.

To do (mostly for Jeff), list all developers on app.

Blake Cooper

Anthony Feliciano

...

[[back](#_top)]

**2. Quick-start guide for developers**

New to UB Collecting? The best way to get started is to clone the GitHub repo, then explore open issues in Jira.

1. Clone the GitHub repo

This project’s GitHub repository is located at [github.com/jcgood/KPAAM-CAM](https://github.com/jcgood/kpaamcam).

The “KPAAM-CAM GitHub Repository” section of this document includes more information about what is included in the repository, as well as suggestions for organizing branches and committing your work.

1. Review open tasks in Jira

Project work is documented and tracked on the project’s Jira board:   
[KPAAM-CAM.atlassian.net/jira/software/projects/UBCOLL/boards/1](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1)

The “Suggestions for Future Work” section of this document organizes the open issues in Jira by type of work (front-end/UI, back-end, database and synchronization). You will need to register an account and have it added to this project to see the board.

1. Set up your IDE

Android Studio is the most commonly used IDE for Android development. For this project, you’ll be required to create one or more Android device emulators using Android Studio. More work should be done to document which device emulators, and which versions of Android, do and don’t work with the app in its current state. A “recent” device (as of 2021) with the latest version of Android should work “out of the box,” although this cannot be guaranteed.

[[back](#_top)]

**3. Understanding the app (overview)**

*Basic operation*

The UB Collecting app assists researchers by allowing them to create research questionnaires and use them to conduct research in the field.

When opening the app for the first time, a researcher’s workflow might look like the following:

1. Open the app and create an account with the Admin role
2. Log in using that account
3. Create three new Questions for a research questionnaire
4. Create the new research Questionnaire, by adding the three questions created in the previous step
5. Define a Field Trip object to organize all responses to the Questionnaire together by virtue of being conducted in the same trip

Note the use of the **Admin** role in step 1. Admins can create Questions and Questionnaires for use by Interviewers. Users with the Interview role, by contrast, are able to define Field Trips, during which they collect data, administer Questionnaires (created by an Admin), and add research subjects as interviewees.

At present, Admins and Interviewers always have separate accounts even if the same person can take on both roles in the project.

While the Interviewer is conducting research, saving Answers elicited from the Questionnaire designed by the Admin (likely a supervisor or mentor), the app is storing all the information generated thus far—from the researcher’s identity and that of the subjects, to the Questionnaire, Questions and Field Trip information—both in a local, SQLite database on the client device, and, assuming there is an internet connection available, to a cloud database designed to distribute data collected by one researcher to everyone using the app.

Diagram

Description automatically generated

*Fig. 1 UB Collecting app workflow*

*Database/data types*

To provide maximum customization and flexibility to users both creating research instruments and collecting data in the field, the app defines 21 different data types. Each of these data types is defined by a corresponding file in the *ubcollecting/data/models* folder, and its corresponding table in the client database is created using a file in the *ubcollecting/data/tables* folder.

Together, these files define and implement the schema used for the underlying SQLite database on the user’s device (see below).

A screenshot of a computer

Description automatically generated with low confidence

*Fig. 2 Database Schema*

[[back](#_top)]

**4. KPAAM-CAM GitHub Repository**

*Branches*

The KPAAM-CAM GitHub repository has many branches, most of which are stale and can safely be ignored (but may be useful to keep as a reference to past work).

Good practice suggests creating a new branch for each new feature (i.e., each branch should correspond to a numbered issue on the project’s [Jira board](http://kpaam-cam.atlassian.net/jira/software/projects/UBCOLL/boards/1)). Jeff Good, the owner of the repository, may have to give you permission to create branches first.

Tested and completed work on a new feature can be merged into the *development* branch, where new features can be tested together before inclusion with the main version of the app.

The *master* branch contains the latest version of the app. Conventionally, *master* gets its updates from the *development* branch exclusively. It may be convenient to “schedule” updates to *master* at the end of each semester or year, or as new features are completed.

*Files in* master *branch*

The following is a list and brief description of files and directories in the *master* branch.

|  |  |
| --- | --- |
| File/Folder | Description |
| AndroidApp | Contains the codebase for the UB Collecting Android app. *UBCollecting* is the root directory for the codebase (see the following section). |
| AndroidDataCollectionScript | A series of bash scripts and xml files intended for use with the FAIMS field data collection platform. FAIMS was used in a pilot phase of the development of the app and is not part of the UB Collecting app at this time, and these scripts may be ignored for those making non-FAIMS related updates to the app. |
| UBIRTransformer | A Java program for exporting data from the app’s database to the [UB Institutional Repository](https://ubir.buffalo.edu/xmlui/). |
| folderExtractor | This is Java program to extract data from the app’s database to a folder. At the time of writing, it has not been updated in 4 years, and can be safely ignored for most work on the app. It may not work with the latest version, and it was developed so that Interviewers in the field could easily explore the data had collected even if it had not yet been uploaded to a servery. |
| userDocumentation | Contains the “UB Collecting User Guide,” a thorough description of the app’s functionality. While written for users, this guide is recommended for developers as it explains the various interactions between different data types (Roles, Persons, Questionnaires etc.). |
| .gitignore | Instructions to ignore desktop-specific settings files created by the Android Studio IDE (or other JetBrains IDEs, like IntelliJ etc.) and others when committing to the repo. |
| LICENSE | Standard license file. The UB Collecting app is licensed under a GNU Public License v3.0 |
| README | The readme file that appears below the list of files on GitHub. |

[[back](#_top)]

**5. Codebase**

*Location*

The relevant files for developers can be found in the *UBCollecting* directory under *app/src* (per Android conventions). The main folder contains three items:

* the *java* folder, containing the app’s java code (i.e., most of the app’s codebase);
* the *res* folder, containing additional resources (mostly in XML format) on which the app’s code relies; and
* the **AndroidManifest.xml** file, which, per Android conventions, determines the permissions for the app when running in the Android OS.

The *java* folder is where developers will spend most of their time. What follows is an explanation of the files in this directory (inside *java/edu/buffalo/cse/ubcollecting*).

*Structure*

Application

This directory contains one file: **App.java**, an Activity governing global app-level configurations. App.java handles the following:

* Ensuring that the required Application.onCreate() method is called, which establishes an Android application’s required startup processes;
* Initializing an instance of the app’s context to share with non-Activity objects (see note below);
* Initializing helper objects for handling data: **DatabaseHelper**, **DatabaseManger**, **FirebaseApp** and a HashMap of **FireBaseSynch** objects for the app’s various data types;
* Checks the app’s preferences file to see if this is the first time running the app; if so, the app establishes a few pieces of data (e.g., the Admin and Interviewer roles) necessary to run the app.

**Note about context sharing:** The way the UBCollecting app handles context sharing is imperfect; because of [how contexts work in Android](https://developer.android.com/reference/android/content/Context) (specifically that they are created and destroyed as needed by the OS), it isn’t a best practice to store a context as a variable and refer to it on an ongoing basis, as this app does. This is a compromise meant to allow the various non-Activity objects in the app, specifically those related to data management, to access the app’s context when required. While this implementation works well for these purposes, it leads to “context leak” (memory leaks related to obsolete contexts being saved beyond their designed lifecycle), and can theoretically create errors when the context being referenced is obsolete (although this has not been observed in the app’s behavior so far).

Data

For each of the app’s 21 data types (see section 4, “Database and Data Types,” for more information), there are two files in the *data* directory: a **table** and a **model** (found in folders within the data directory by the same name).

* **Table** files define the columns for the SQLite database table corresponding with that data type, as well as operations related to that table, such as comparing data entries;
* **Model** files define the data type itself, as well as the setters and getters used to interact with data of that type.

Critically, the app uses these Table and Model files to generalize data operations within the app. For example, the main insert() method used by the SQLite database (found in the **Table.java** file within the *data/*t*ables* directory) works by collecting the table columns and getters from the Table and Model files relevant to the data type to define the fields to be included:

|  |
| --- |
|  |
| try { |
|  | String key = tableColumns.get(i); |
|  | Object value = getters.get(i).invoke(model, null); |
|  | insertContent(values, key, value); |
|  | } catch (IllegalAccessException e) { |
|  | e.printStackTrace(); |
|  | } catch (IllegalArgumentException e) { |
|  | e.printStackTrace(); |
|  | } catch (InvocationTargetException e) { |
|  | e.printStackTrace(); |
|  | } |
|  |  |

*Fig. 3 Code from generalized insert() method in Table.java*

The data folder also houses four utilities designed to interface between the app and its data:

* **DatabaseHelper.java** defines and initializes SQLite tables for the client-side database, and populates the initial data required by the app to function;
* **DatabaseManager.java** manages interactions with the client-side SQLite database: opening, closing etc.;
* **FireBaseCloudHelper.java** creates the necessary connections to the app’s server-side Firebase instance, and handles standard CRUD operations on the database in the cloud (see section 7, “Synchronization,” for more). This object synchronizes changes between the client SQLite database and the cloud, when necessary. This object also contains a schema validator utility designed to confirm that the structure and content of the server database matches those of the client, however this utility is resource-intensive, and should only be called for testing purposes (the production app should never call it); and
* **FireBaseSynch.java** is used to reference data in the server-side Firebase database. The app uses it to create references to specific data types via a HashMap of FireBaseSynch objects indexed to the name of the data type, in the App.Java file.

UI

This folder contains files supporting the business logic underlying the app’s GUI. This includes [Activities, which Android defines as a “single, focused thing a user can do”](https://developer.android.com/reference/android/app/Activity); [Fragments, which represent modular and reusable UI elements](https://developer.android.com/guide/fragments); as well as several utilities, managers, listeners and callbacks to handle various app operations.

It’s unclear for what reason (if any) the Activity files in this folder are separate from the Activities in the parent folder. A refactoring opportunity may exist here, in combining the Activity/Fragment files with those in the parent directory and moving other utility files to the Utilities folder in the parent directory.

Utilities

This folder contains two files: the first, **Constants.java,** defines constants that define features of the app’s unique data types; the second, **ErrorMessages.java,** contains no data and does not appear to be referenced elsewhere, and for this reason can probably be safely ignored.

Activities

In the parent folder (*java/edu/buffalo/cse/ubcollecting*) there are several Activity files, most of which provide GUI logic for each of the screens corresponding to the app’s unique data types.

For the files following the naming format **[unique data type]Activity.java** (for example, **AnswerActivity.java, FieldTripActivity.java** and so on), each one provides the user with the GUI necessary to input and (in some cases) validate that data type.

In some cases, these Activity files govern sub-processes *supporting* the data type, if not the data type directly; such is the case with **LoopActivity.java.**

Typical for Android apps, each Activity identifies a corresponding layout, an XML file found in the res/ folder (with the exception of EntryActivity.java, and abstract Activity used to define data operations extensible to the other Activities). These XML files are named similarly (but not *exactly*) like their corresponding Activity; below is a list of corresponding layout files for each Activity in the parent folder, along with a brief description of each Activity.

|  |  |
| --- | --- |
| Activity Name | Description |
| AnswerActivity.java | *Corresponding XML layout file:* **activity\_answer.xml**  This Activity allows the user to input a new Answer data type. A comment in the code claims this Activity is unused, although that should be confirmed by any new developers. |
| EntryActivity.java | *Corresponding XML layout file:* N/A  This Abstract Activity enforces the requirements for every other Activity that deals with data, declaring the methods necessary to view, insert and update Entries (*Entry* is the term the app uses when abstracting the Model object for an unknown data type). |
| FieldTripActivity.java | *Corresponding XML layout file:* **activity\_fieldtrip.xml**  Form that allows the user to input a new Field Trip data type into the app’s database. |
| FileActivity.java | *Corresponding XML layout file:* **activity\_file.xml**  Allows the user to input a new File data type into the app’s database. |
| LanguageActivity.java | *Corresponding XML layout file:* **activity\_language.xml**  Allows entry of new Language data type. |
| LanguageTypeActivity.java | *Corresponding XML layout file:* **activity\_language\_type.xml**  Allows entry of new Language Type data type. |
| LoopActivity.java | *Corresponding XML layout file:* **activity\_loop.xml**  This Activity supports the “loop” Question type, which allows a Question to “loop” any number of times based on previous input (for example, if a subject answered the question “How many siblings do you have?” with the answer “3,” a question like “What is your sibling’s name?” could loop three times to cover each identified sibling). |
| PersonActivity.java | *Corresponding XML layout file:* **activity\_person.xml**  Allows entry of new Person data type. |
| QuestionLangVersionActivity.java | *Corresponding XML layout file:* **activity\_question\_lang\_version.xml**  Allows entry of new Question Language Version data type. A comment in this code claims this Activity is unused so far, although this should be verified. |
| QuestionOptionActivity.java | *Corresponding XML layout file:* **activity\_question\_option.xml**  This Activity appears to be intended to allow the entry of a new Question Option data type. However, it is unfinished (with several //TODO comments) and marked in comments as being unused thus far. |
| QuestionPropertyActivity.java | *Corresponding XML layout file:* **activity\_question\_property.xml**  Allows entry of new Question Property data type. A comment in this code claims this Activity is unused so far, although this should be verified. |
| QuestionPropertyDefActivity.java | *Corresponding XML layout file:* **activity\_question\_property\_def.xml**  Allows entry of new Question Property Def data type. |
| QuestionnaireActivity.java | *Corresponding XML layout file:* **activity\_questionnaire.xml**  Allows entry of new Questionnaire data type. |
| QuestionnaireContentActivity.java | *Corresponding XML layout file:* **activity\_questionnaire\_content.xml**  Allows entry of new Questionnaire Content data type. A comment in this code claims this Activity is unused so far, although this should be verified. |
| QuesetionnaireTypeActivity.java | *Corresponding XML layout file:* **activity\_questionnaire\_type.xml**  Allows entry of new Questionnaire Type data type. |
| RoleActivity.java | *Corresponding XML layout file:* **activity\_Role.xml**  Allows entry of new Role data type. |
| SessionActivity.java | *Corresponding XML layout file:* **activity\_session.xml**  Allows entry of new Session data type. |
| SessionAnswerActivity.java | *Corresponding XML layout file:* **activity\_session\_answer.xml**  Allows entry of new Session Answer data type. |
| SessionPersonActivity.java | *Corresponding XML layout file:* **activity\_session\_person.xml**  Allows entry of new Session Person data type. |
| TableListActivity.java | *Corresponding XML layout file:* **activity\_table\_list.xml**  Includes the tools to generate lists of data in the UI based on the contents of a table in the database. Methods included bind a Table to the list, set the number of items in the list based on the underlying Table and render the UI accordingly. |
| TableSelectActivity.java | *Corresponding XML layout file:* **activity\_add\_questions.xml**  This Activity appears to be an abstract object defining how Tables are interacted with using the UI; however, it is not referenced elsewhere in the code (aside from AndroidManifest.xml) and therefore does not appear to affect the operations of the app. |
| TableViewActivity.java | *Corresponding XML layout file:* **activity\_table\_view.xml**  Includes the tools to list the data types on the main screen, with tools to add or update data in that data type’s corresponding Table. |

Miscellaneous

* **Gatekeepers.java -** This object defines Booleans to govern whether certain Question Properties (audio, video, photo, text, list) should be used in the app. Currently, only SHOULD\_USE\_TEXT\_QUESTION\_PROP is set to true, and all others are set to false, however neither this object nor its variables appears to be referenced anywhere else in the code.
* **SyncAdapter.java** - An extension of [AbstractThreadedSyncAdapter](https://developer.android.com/reference/android/content/AbstractThreadedSyncAdapter) used in the SyncService object (see below).
* **SyncService.java** - An extension of Android’s [Service](https://developer.android.com/guide/components/services) component. This file, and SyncAdapter.java, appear to be intended to introduce synchronous operations in the background of the app. However, they are not referenced anywhere else in the codebase, and so have likely never been implemented.

[[back](#_top)]

**6. Data Synchronization**

In 2021, work began to implement synchronization of data across multiple instances of the app, so that field researchers and admins could coordinate work and share data. Prior to this, each instance of the app kept its own onboard database, and was only shared outside the app via scripts to export the data and host it elsewhere.

*Requirements for synchronization*

* Data generated by *researchers* (questions, questionnaires and so on) need to be uploaded to a central location and then distributed to other instances of the device, so researchers can share research instruments and other data. This is the high-level definition of *synchronization* for the purposes of this project.
* Data generated by *research subjects* (answers to questions) need to be synchronized so researchers in another location can reference data being generated in the field.
* People identified in the app as research subjects should be “identifiable” to other researchers working with the same individual. That is, when a researcher enters a subject’s information, the app should confirm whether that person has already been added and, if so, connect any new data to existing data about that person.
* Data synchronization must account for environments with poor connectivity, preferably caching data to be synchronized when connectivity is limited or unavailable and synchronized automatically once the connection has improved.
* Data synchronization must account for data versioning. For example, if a field researcher changes a questionnaire for use in the field, the new version should be made available to other researchers, but not necessarily *overwrite* the previous version of the questionnaire in case other researchers wish to use the previous version.

This last point requires further consideration. What are the data versioning needs for this app? Should every version of every piece of data be archived? Can (or should) users be allowed to make decisions about when and how data updates on their devices? How does support for multiple versions of data across multiple devices disrupt the synchronization of data centrally? These questions should be answered before handing this aspect of implementation.

In 2021, Dr. Oliver Kennedy (UB professor whose research focus is distributed networks) provided the project team with a list of questions meant to provoke thought about the best way to implement a data synchronization service:

1. What kind of guarantees do you want to enforce, and where?
2. How does data change (e.g., are updates always inserts)?
3. Can simultaneous changes on two devices violate the guarantees?
4. If so, how do violations of those guarantees get resolved?
5. Are the client databases kept always up-to-date with the central server, or are they expected to be used in disconnected mode?

*Firebase as synchronization tool*

[Firebase](http://firebase.google.com/) was chosen because of its extensive documentation and reliability, the ability for devs to use a “self-service” model (i.e. doesn’t depend on outside teams), and its ability to meet the requirements defined for the app (above).

Initially, work had been done to use [FAIMS](http://www.fedarch.org/) for synchronization, due to its offline availability of data and focus on research use cases. But, because of the dynamic state of the FAIMS project (the development team was planning a complete redesign of FAIMS as of early 2021), it was decided to implement this functionality with Firebase instead.

*Implementation overview*

There are two files in the UB Collecting codebase relevant to the Firebase implementation: **FireBaseCloudHelper.java** and **FireBaseSynch.java**.

* **FireBaseCloudHelper** establishes a listener for the cloud connection—the current, limited implementation of the app’s requirement that network signal presence and strength can be checked for—and allows for insertion, updating and deleting of data from the cloud database.

Currently, where implemented, this object’s insert(), update() and delete() methods are called whenever corresponding inserts, updates and deletions are made in the client-side, SQLite database (typically in Activities related to generating or collecting data).

FireBaseCloudHelper also includes an object used for cloud database schema validation. It compares the database schema on the client device with the synchronized database in the cloud by converting the client SQLite database to a JSON object, then iterating over that object to check that the same structure and data are represented in Firebase’s cloud database. This is for debugging purposes; it is not recommended to run this validator during normal app operation (for example, the validator shouldn’t be run every time a write is made to the database, or even every time the app is started up), as the JSON iteration algorithm is not efficient and affects performance.

* **FireBaseSynch** keeps the individual instances of the app synchronized with changes happening elsewhere, via a listener referencing the Firebase cloud database in the getData() method. It also includes methods that update the UI, notifying the user when there are new changes to synchronize to the client’s device.

More work is needed to fully implement synchronization services; the FireBaseCloudHelper.update() method is not included in the codebase currently, and should be tested further.

Additionally, it is recommended that FireBaseCloudHelper be refactored so that it becomes a single, static object capable of serving network status and updates on request, rather than creating a new object whenever this service is needed (see section 8, “Suggestions for Further Work,” for more information).

*Firebase credentials*

**URL:** firebase.google.com

**User:** KPAAM-CAM123@gmail.com

**Password:** KPAAM-CAM2021

*Further reading on Firebase*

Google has extensive documentation relating to Firebase online at [firebase.google.com/docs](https://firebase.google.com/docs).

This project uses Firebase’s Realtime Database, which is documented here: <https://firebase.google.com/docs/database>. Please note that Cloud Firestore, another database system used with Firebase, is *not* used in this project.

*Notes on abstraction and other synchronization options*

Firebase is not intended to be the singular solution to synchronization in the UB Collecting app. Rather, it provides one solution out of potentially many for synchronization. The implementation of Firebase was intended to be “modular,” meaning that another data synchronization service could be configured and “plugged in” as required.

Future developers should consider implementing abstract equivalents of the FireBaseSynch and FireBaseCloudHelper objects, that set the requirements for functionality (presumably with methods for CRUD operations, like insert() and update()), but don’t define them. Then, additional synchronization services and tools could be integrated by extending these abstract methods and defining the necessary operations for synchronization.

[[back](#_top)]

**7. Suggestions for future work**

New developers should refer to the [UB Collecting app’s Jira page](https://kpaamcam.atlassian.net/browse/UBCOLL) for a list of issues to be completed. Work on this app generally falls into the following categories, each one with potential work to be done:

*Front-end/UI*

1. Add a ‘back’ button [[UBCOLL-2](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-2)]
2. Replace any hardcoded strings in UI with string variables in the res folder [[UBCOLL-51](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-51)]

*Back-end*

1. Remove Toast notifications and Log outputs used in testing [[UBCOLL-68](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-68)]
2. Fix an error on admin account creation that causes login processes to fail and the app to crash [[UBCOLL-71](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-71)]
3. Add ‘forgot password?’ functionality [[UBCOLL-4](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-2)]
4. Fix bug: Field Trip start dates can be later than end dates [[UBCOLL-27](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-27)]
5. Prevent users from creating different accounts with the same email and password [[UBCOLL-28](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-28)]
6. Add recursive dynamic questions [[UBCOLL-6](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-6)]
7. Fix bug: app crashes on loading camera activity for first time [[UBCOLL-49](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-49)]
8. Redesign app architecture to conform to latest Android standards [[UBCOLL-72](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-72)]

*Database*

1. Review database schema for normalization [[UBCOLL-60](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-60)]
2. Fix bug: role is ‘null’ after creating new account [[UBCOLL-77](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-77)]

*Synchronization*

1. Rewrite DatabaseHelper.java to populate initial data from synchronization server, if available [[UBCOLL-45](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-45), [UBCOLL-76](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-76)]
2. Implement ability to periodically check for presence and strength of network signal [[UBCOLL-14](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-14)]
3. Enforce authorization for cloud database interaction [[UBCOLL-50](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-50)]
4. Implement the ability for versioning of data in the server database [[UBCOLL-59](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-59)]
5. Make large media uploads conditional on adequate network signal [[UBCOLL-64](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-64)]
6. Optimize synchronization to avoid performance issues [[UBCOLL-66](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-66)]
7. Update the ‘version’ value in the cloud data when changes are made on the client [[UBCOLL-73](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-73)]
8. Re-implement FireBaseCloudHelper as a static object to improve performance and stability [[UBCOLL-78](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-78)]
9. Implement full update synchronization between client and server [[UBCOLL-81](https://kpaamcam.atlassian.net/jira/software/projects/UBCOLL/boards/1?selectedIssue=UBCOLL-81)]

[[back](#_top)]

**8. Glossary of terms**

**Admin –** Users with the Admin role enabled can create Questionnaires to be used as research instruments within the app. Compare this to the Interviewer role; Interviewers can only elicit responses using existing Questionnaires.

**Entry –** theabstract term for an item of data. When users create or collect data, it is often handled as an instance of an Entry, rather than its specific data type, so data operations can be generalized.

**FAIMS –** a project to “build tools for digital data collection in the field,” the FAIMS project (short for Field Acquired Information Management Systems) has built tools to address some of the use cases of the UB Collecting app, especially the creation of research instruments and the synchronizing of data across devices. As of 2021, FAIMS is being redesigned; for the time being, it is not being used in this app, but may be a viable option for some of its functionality in the future.

**Firebase –** A cloud data management and synchronization service from Google. As of 2021, Firebase is used as the synchronization element in the UB Collecting app. For more information, see the Synchronization section.

**Jira –** A website used to organization projects, typically software development projects managed using the Agile methodology. The UB Collecting app uses Jira to track outstanding issues and tasks.

**KPAAM-CAM –** KPAAM-CAM(Key Pluridisciplinary Advances on African Multilingualism – Cameroon)Top of Form

Bottom of Form

is the research project of which the UB Collecting app is part. **Top of Form**

**Bottom of Form**

**Model –** the abstract definition of the app’s unique data types. Each of these data types extends the Model class and defines how the data is created, updated and deleted.

**Realtime Database** – the name of the cloud database used for this project, available from the Firebase dashboard.

**UB Collecting –** name of the Android app developed as part of the KPAAM-CAM project.

**UB Institutional Repository –** a UB-internal repository designed to collect, preserve, and distribute UB's research and scholarship. While some work has been done to incorporate the UB Collecting app’s functionality with the UBIR, most developers working on this app will not need to interact with the UBIR.

[[back](#_top)]