Pynny

UALR CPSC 4367: Mobile Applications

Final Report

Zachary King | zcking@ualr.edu

Fall 2017

Project Statement

Nowadays we use our smartphones to assist with nearly everything. The mobile application I am proposing to develop is called *Pynny*, and it is the next enhancement of mobile assistance in the field of finance. I have designed and implemented a mobile application for the Android platform, that allows users to easily track their finances, understand their data, and export useful and meaningful reports from their mobile device.

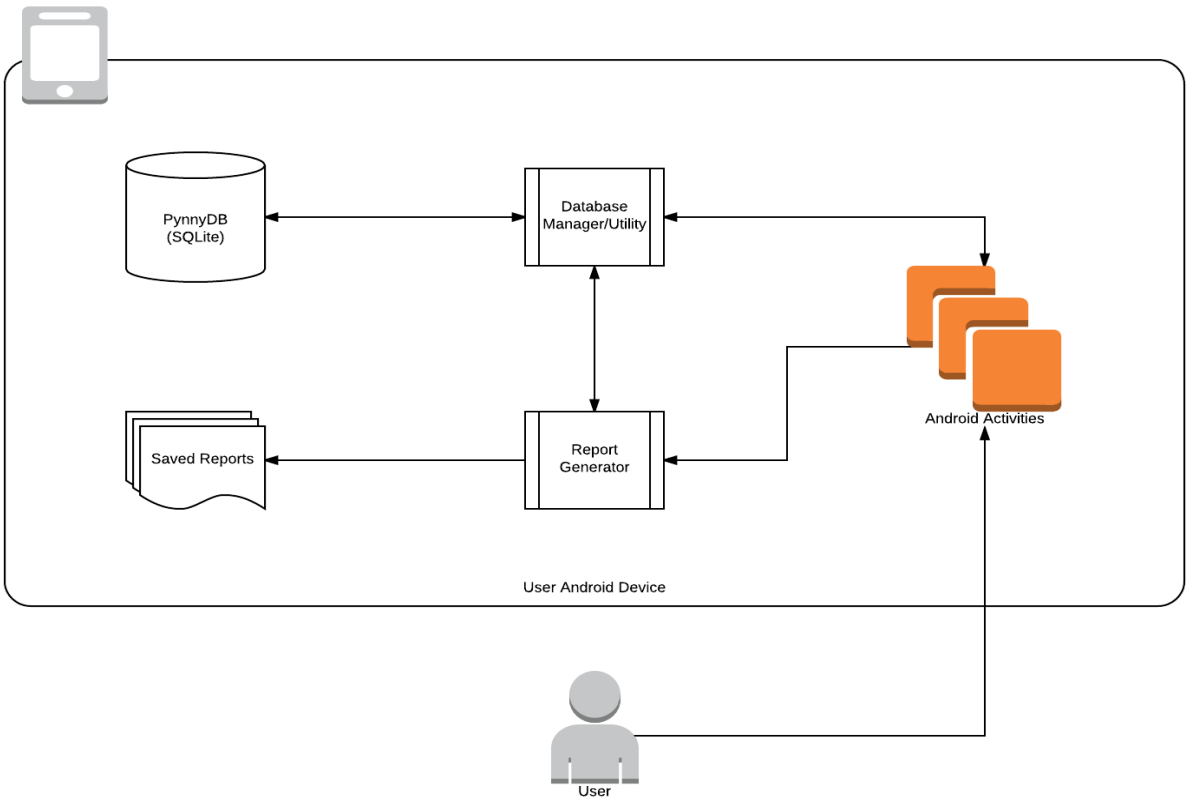
I was highly motivated to build this budgeting application for several reasons. The existing applications for budgeting, financial management, and tracking are all crowded with unnecessary features, difficult to use, or lack the reporting functionality I believe is vital for data insight. Previously, I created a similar, but more robust web application using Python and the Django framework for these same reasons, and am also building the Pynny mobile app as a mobile-friendly and offline version of the web app. The primary unique characteristic for this app that sets it apart from others is the usage simplicity and rich reporting functionality. I believe this app has the potential for a significant impact on the way average people such as myself control their spending; if you can *see* how you spend your money, you can better manage it.

Application Design

This application is composed of the following components, or modules, which interact with one another to compose a modular, high-performant, data-driven user experience:

- Data Models – for mapping data from storage to the application layer
- Database Manager – utility for accessing and operating on data in SQLite
- Activities – user interaction/interface for individual models and collections
- Reporting Manager – utility for generating reports and saving to device

The following diagram depicts the high-level architecture of the application; note that the application is purposefully designed for offline smartphone use. Note that I am beyond the design phase and much of the design is transparent throughout the application implementation and evaluation.



Application Implementation

Data Models

The core component of the application is the data, since the application is completely data-drive. This module encompasses the application layer level wrapper around this data, and defines what it is mapped to from the database storage.

The data models are Java beans, or serializable POJOs (Plain Old Java Objects) [0], defined as a class of attributes and the corresponding getters and setters; this is quite conventional and makes for an intuitive mapping from the database to the application layer. The data models are structured as detailed in the following tables:

CATEGORY
ID: long
Name: String
IsIncome: boolean

WALLET
ID: long
Name: String
Balance: double
CreatedAt: String

TRANSACTION
ID: long
Amount: double
Category: *Category*
Description: String
CreatedAt: String
Wallet: *Wallet*

BUDGET
ID: long
Goal: double
Balance: double
Month: String
Category: *Category*
Wallet: *Wallet*

Database Manager

The database manager is a utility class providing an API to access and perform operations on the data stored within the SQLite database. This class uses the singleton design pattern to enforce strict usage and to reduce memory and increase performance.

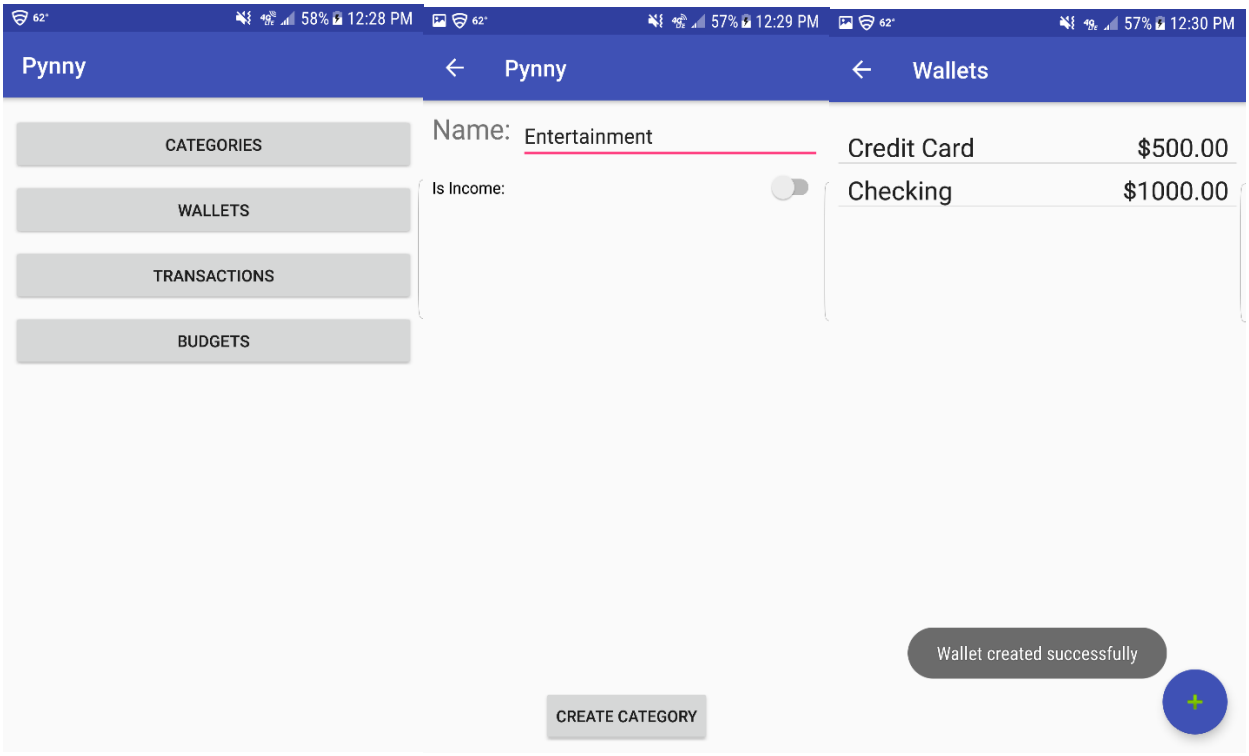
The database manager implements methods for getting individual data elements, getting entire collections of data elements, adding data elements, deleting data elements, and updating data elements. Thus, this singleton utility class provides a convenient CRUD (Create, Read, Update, Delete) [1] API into the database layer. Since I'm using SQLite as the database backend, the manager class can easily extend the *android.database.sqlite.SQLiteOpenHelper* [2] to accomplish this.

Android Activities

The core connectivity between the components and user is the Android activities. Activities are used to implement the individual and collection interactions between the user and their data, as well as using the report generation functionality.

Due to the scope of this project, the interfaces were kept minimum, and target smartphones only; the main focus of this application is on the data as this is a data-driven application. ListViews are used to display collections of data elements, which are mapped to the interface with corresponding adapters. I chose to use ListView because it enables the implementation of the adapters to be extensions of the existing Android *CursorAdapter* [3], built specifically for mapping data from a relational database to the user interface; thus, I believe this is best for performance and contextual purposes. For example, I have implemented a *CategoryCursorAdapter* for mapping the Category data to a XML layout, and likewise for the other models.

Below are figures depicting just a few of the layouts for some of the activities, showing views such as a “create new Category” activity, and a “list Wallets” activity.



Report Manager

The manager class for generating reports is another singleton design pattern, which acts as a utility class for abstracting the creation, serialization, and deserialization of reports. However, it's implementation leverages the existing database manager's API and exports the fetched data to a serialized format on the device's disk.

The export feature for reports is very straightforward; the Android file system can be interacted with through *android.os.Environment* [4], and the file management and writing is the same as any other Java file-operating process (using *java.io.File* primarily).

Initially, I hoped to also have the bandwidth to integrate cloud storage as optional destination(s) for generated reports (i.e. Google Drive, Dropbox, etc.). However, due to the constraints of my timeline, these feature plans are being reserved for a later release of the application.

Testing and Evaluation

Throughout my design and implementation phases I have kept performance and practicality in mind as much as possible. I have performed several manual tests to verify the functionality of the CRUD (Create, Read, Update, and Delete) operations, using logs, the Android Monitor, and by connecting directly to the SQLite database on the device. Although time did not permit it for the initial release of this application, I plan to implement a foundation of unit and integration tests using JUnit [5][6] to rely on as the development of the application continues beyond this course.

Source Code and Presentation

I approached development on this project very seriously and used Git to manage version control throughout its lifecycle. The source code for this application can be found at https://github.com/zcking/PynnyAndroid. The **master** branch is considered the current "release" of the app; the **develop** branch is where future development will propagate to, before ultimately being merged into **master** as new releases. This branching strategy is well-illustrated here: http://nvie.com/img/git-model@2x.png. The presentation slides can also be found in this repository's root directory for future reference.

References

[0] http://www.geeksforgeeks.org/pojo-vs-java-beans/

[1] https://en.wikipedia.org/wiki/Create,_read,_update_and_delete

[2] https://developer.android.com/reference/android/database/sqlite/SQLiteOpenHelper.html

[3] https://developer.android.com/reference/android/widget/CursorAdapter.html

[4] https://developer.android.com/reference/android/os/Environment.html

[5] http://junit.org/junit4/

[6] https://developer.android.com/studio/test/index.html#test_types_and_location

Experiences and Thoughts

As the development of this application began its course, I found there were myriad instances of general repetition in the designs. Each data element had to be modeled, have database API operations written for it, a cursor adapter written for it, and the relative activities matching the CRUD operations. I found myself writing very similar implementations for these steps across the four data models, and I believe there is some opportunity for refactoring because of this.

Additionally, I would like to comment that although database storage was among the last topics covered in class, I began implementing it on my own early as one of the initial steps in project; I found this extremely helpful, especially for a data-driven app, and would've liked to have seen the topic be covered in class sooner.

Future features I would like to implement include:

- Cloud storage destinations for reports
- Charts for data visualization (i.e. trends, spending, etc.)
- Fragments for tablet support as well