Mobile & Cloud Software
DEVELOPMENT

CS496 FINAL REPORT

FALL 2015

Student:
Sam Quinn
Quinnsa@Oregonstate.edu

Professor:
Justin Wolford
WolfordJ@Oregonstate.edu

December 9, 2015

Contents

1 Portability

Android compiles binaries in the form of an APK. Inherently all the APK file extensions are compatible with all versions of Android but version checking is implemented within the APK itself. I compiled my final application with the SDK version 23 which is the latest version of the development kit. Due to the methods and Android API calls I have implemented I set the minimum SDK version to 22. I tested my application on the latest version of Android (6.0.0) but is backward compatible to Android (5.1.0). The versions of Android I have developed for are narrow, but since I did not use old calls it allowed me to ensure that my application will continue to work with newer versions without having to rebase the entire application.

With every version of Android's SDK there are additions and subtractions from the standard methods and Android API level calls. While developing my application I would find solutions to problems posted online not more than 2 years ago and the solution proposed had been created with deprecated calls. This gave me a first hand experience in how source code portability could be an issue when developing mobile applications.

I took advantage of as many built-in calls as I could and tried to not recreate Android functions, which is a fundamental development process in object oriented programming. By using a built-in function I not only know that it has been tested, they can also be updated and adapt to other versions in the future. Having the built-in functions changing has both pros and cons though. Adapting to newer versions of Android is a pro but if the method changes syntax it could cause the application to return undesired results or break completely.

As described in the lectures I tried to keep my source code as simple as possible. I avoided creating overly complicated class structures that would make finding a bug or a upgrading my application a pain. With the minimal locations that I would have to look when trying to implement a new feature or find an incompatibility I not only save myself the headache it also makes my code more portable.

The portable messaging syntax that I implemented universally was JSON. JSON is not only very popular, human readable, and easy to parse it was also a good choice for my backend API. Every interaction from the backend API is done with JSON. Storing a new public encryption key or person form the application I send a POST request with a standard dictionary look up structure. Upon a successful POST it will in turn return a JSON object with the new encryption key or person. GET requests return JSON as well and parsed within my application.

2 Usability

My application is very simple. I never intended the application to do anything more than store public encryption keys and people objects. This was mainly due to the fact that I almost did not have enough time to even finish those two requirements. Because I kept the application straightforward and simple it greatly increased the usability. I did not implement features that did not directly improve my initial goal of saving public encryption keys and people objects. Every thing that I have implemented has a function and a purpose that is required to meet my design goals, absolutely no bloat.

Aesthetics of my application are not the greatest. The android studio has great templates that implement standard Android material design, which make my application look fairly nice and modern. I definitely wanted to make sure was that the users of my application were not getting distracted or lost because of the appearance. I did not add any flashy pictures,animations, or create anything too ugly that would deter the user of the main goal.

There are several locations where the user needs to make external API calls to the backend database where processing time can vary. I did not implement any type of progress bar or working circle. This could potentially lead users to the notion of their action not fully being executed, but I did implement final toast notifications regarding the completion of tasks. For example when a encryption key has been successfully stored a toast message would appear on the screen over everything else. Android toast notifications are very simple to implement as well as very informative to the user because of the always on top nature.

With the use of a navigation fragment within the application the navigation is consistent across all activities. Since you can reuse a fragment multiple times every navigation side bar across the entire application is the same one with all the items in the same location. I have also taken advantage of the top navigation area where settings usually go to implement the login and logout activity. I decided on the login, logout feature to be nested within the top navigation bar form common application design standards.

The only aesthetic problem I had come across was the switching of orientations. For some reason when the phone would turn landscape things would get wonky. I had solved the problem of the application crashing but with the way Android switches orientation the whole layout is redrawn. Inside of the edit public encryption key activity I have the public encryption keys available to edit in a list that is set to invisible and the save public encryption key fragment is laid over the top. When the orientation is switched, the screen is redrawn and the previously invisible list becomes visible again which is posted up in the background. This is an undesired feature and would be addressed if I had more time.

3 Reliability

My application at its final commit has a fairly reliable functionality. During the development I had implemented checks on the data that users are able to enter. For example if a user decides to not enter a form field that is required it will prompt the user to fill in the missing field before they could continue. The backend API also enforces required data fields. The API will refuse to save any public encryption key that does not satisfy all the required data fields. This proactive approach to data checking will eliminate the problem of incomplete data entries within the database.

The Backend of the application is implemented with Google's App Engine. I will not go into too much detail about Google's App Engine other than uptime and data resilience. Google App Engine's uptime is roughly 99.95% according to the service level agreement

(SLA). Uptime is very important to my application since every public encryption key is read directly from the API. If the API server was down you would still be able to store a cached version of the public encryption key if you were saving, but would not be able to view any public encryption keys. Google App Engine's datastore is also highly replicated that offers great data retention across all servers. If one server crashes the data is stored in multiple locations. With the data stored in multiple location it can also help with load balancing. For the current state of my application I never think that the database would be stressed since the API calls are fairly simple and would not require much computing power. But is nice to know that my application backend could handle it.

4 Performance

My application is very lightweight and requires very little system resources to function. This is mainly due to the fact that I tried to keep it as simple as possible covering only the mandatory features. I have identified the bottleneck of my application and it is making the API calls on the network. While the network calls are very small (less than 10 kb) this had never actually been a problem during my testing.

I have implemented my API in such a way that the data stored and retrieved is very simply. I do not need use complicated queries with join, and, or or operators. While the NDB datastore is designed to optimise all applications and can work with much more data intensive applications I thought it was still important to optimize my application as much as possible, but I could have done a better job. The first performance hit is with the way I retrieve data from the API. While Google App Engine is designed to make quick database queries the way I retrieved data is poor. I return all public encryption keys at once and do the filtering at the application layer. This works, but I should have taken advantage of the performance and functionality of the Google App Engine datastore.

The second performance hit that I did not remedy within my application is the number of network requests. I do not cache anything within my application. It would be much better for the server costs if there were actually more users using my application to store data caches. If my application stored data caches the user's public encryption keys should not change from anyone other than themselves so it would be easy to determine if a sync to the API needs to happen or not. Right now every time my application switches activities, even though they are using the same list of encryption keys, each activity must make a request to the API and retrieve all public encryption keys. If my application had a large user base the way I have it implemented now would be extremely costly and would be less efficient.

5 Security

While the initial design of my application was supposed to share public encryption keys there was not much need for security. The existing public encryption key databases out there today often times do not even have an option to login. Contrary to correct secure application development I added my security features later in the development rather than thinking about security from day one. After I added the login feature to my application it opened the new possibility to store private encryption keys. Private encryption keys must be kept very secure and should never have the chance for anyone besides the owner to view them. If a private encryption key is compromised it would deconstruct the fundamentals of public/private encryption. With the current state of my security on my application I would not trust it with private encryption keys just yet. I implemented the login with Google's Play Services, which makes the mobile application fairly secure but the API backend never verifies the encryption key. When a user requests to see their own encryption keys it actually queries the API to return all encryption keys and filters out all encryption keys that do not match their Google user Id. It would be very easy to get anyones stored encryption key using the API rather than the application.

Google play services sign-in my application uses the same authentication method as Gmail, Youtube, Play store and many other professionally developed applications. Because I did not create my own account system I never had to deal with user misfortunes like forgetting a password. If a user happens to forget their password, all that is necessary is to follow the password recovery for their Google account. Implementing this sign-in service really reduced the overhead of creating a secure login system and takes care of many housekeeping features.

With the use of Google's play services as an authentication method my application uses smart reauthentication. When a user has not used the devices for a while or has been flagged for suspicious activity Google's sign-in will ask for the user to reauthenticate. Google's sign-in also allows for an encrypted cached copy of the account data so the user does not have to login every time they want to access the application.

6 Interoperability

My does not interact with the device in many scenarios. The only case that my application practices device level interoperability is when it uses the internal storage. When the application notices that it is offline, instead of just not saving the encryption key that had been entered it will store the encryption key locally. This adds fluidity and reliability to the application and is not possible without adding the permission to store data to the device's internal storage.

When the application sends data or requests to the API it uses typical POST and GET calls. These calls are interoperable with all systems, IOS, Android, and command line. While all of the application code is strictly written for Android the backend API can be implemented anywhere.

Every message that is sent back to the application from the backend API is done with JSON. JSON is also very interoperable amongst all the different platforms. Within the Android libraries there are JSON objects that allow the parsing of JSON very simple and easy.