2017

Design and Implementation of an Improved Android Application for Bard Shuttle Services

Chance O'Neihl Wren
Bard College

Design and Implementation of an Improved Android Application for Bard Shuttle Services

A Senior Project submitted to
The Division of Science, Mathematics, and Computing
of
Bard College

by
Chance Wren

Annandale-on-Hudson, New York
May, 2017

Abstract

With the growing population of Bard College, the need for the college’s shuttle system continues to grow. As a result, enabling the Bard community to quickly and easily access the shuttle schedules and times, has also become more important in the daily of life of Bard College's inhabitants. Although Bard College has a mobile application for Android and iPhone mobile devices alike, there was a growing demand for a new improved shuttle application for Android mobile devices. This project seeks to improve the functionality, user friendliness, and availability of shuttle schedules to the Bard Community, in the form of a new mobile application. This project comprises of three parts: application design, implementation, and beta-testing. The application design consists of the design and structure of the user interface and backend database. The implementation consists of what tools were crucial to the development of the application, such as the type of database used to store shuttle information, the programming language used, and the development environment, just to name a few. Lastly, beta-testing consisted of a small group of Bard students that volunteered to use an early version of the application for a specified amount of time to provide feedback for possible changes and improvements.

Table of Contents

Abstract **1**

Acknowledgements **3**

Chapter 1: Introduction **4**

1.1 Problem Description 4

1.2 Project Goals 5

1.3 Contributions 6

Chapter 2: Background **7**

2.1 Previous Bard Shuttle Application 7

2.2 Resources and Tools 8

2.3 SQL/SQLite and Online Database Connection 10

2.4 Android Studio Integrate Development Environment (IDE) 11

2.5 Github 12

Chapter 3: Mobile Application Design **14**

3.1 Use Case Diagrams 14

3.2 SQLite Database Setup and Structure 17

3.3 User Interface (UI) Design 20

3.4 User Interface Technical Implementation 22

3.5 SQLite Database Implementation 34

3.6 Alarm Services Implementation 46

Chapter 4: Conclusion **53**

4.1 Beta Tester Feedback and Limitations 53

4.2 Future Work 56

Appendices 58

A Functions for AlertDialogs for CampusShuttle class **58**

B Android Studio Interface Images **59**

C Android Studio Emulator Images **60**

D Bard College Shuttle Github Repository **62**

Bibliography **63**

Acknowledgements

I would first like to thank my friends and family whose love and support for me thus far has enabled me to pursue my passions throughout my life and especially my college career. I would also like to thank Sven Anderson for teaching an amazing course on Android Application Development, without which my project wouldn’t have been possible. Lastly, I also want to extend my heartfelt appreciation and gratitude for my supervisor Khondaker Salehin who guidance, expertise, and encouragement were vital to the completion of this project.

1

Introduction

1.1 Problem Description

Since enrolling at Bard College, the shuttle services have played a pivotal role in the quality of campus life for not only students like myself, but teachers and other Bard College staff alike. Bard College currently offers two mobile applications for easy access to its shuttle schedule. One is present on the iPhone and the other on Android devices, with each helping to provide students and staff with a more user-friendly way of viewing the campus shuttle schedule. Both have room for various changes and improvements, but since my expertise is limited to that of the Android device platform, it will be the sole focus of my project. The current Bard College shuttle application available for Android devices is simple, user-friendly and provides up-to-date shuttle arrival and departure times based on the user’s selected start and destination point. Unfortunately, although simple and easy to use, the current shuttle app user interface needs to be updated to better fit the resolution of newer devices and android operating systems. Secondly, since the shuttle app is maintained by an individual that no longer attends Bard College in some instances it has taken weeks, sometimes months for the application to be updated to the new school year’s shuttle schedule and times, and as of 2017 the current shuttle application no longer provides any shuttle times. Thirdly, the current shuttle application only provides shuttle times for the local Campus Shuttle whose route extends from the town of Tivoli to the Hannaford’s supermarket. Although Bard College offers shuttle services to local train stations, Rhinebeck, Woodbury Commons Mall, Kingston Mall, and special shuttle schedules for the L&T program, summer and winter break, neither current mobile application give users the option of viewing their shuttle times. Lastly, Although Bard College maintains the available list of shuttle times on their transportation website, the college does not natively maintain the shuttle application for android devices themselves.

1.2 Project Goals

The goals of this project are to create a new and improved version of the Bard College Shuttle mobile application for android devices. This new mobile application seeks to address and fix several if not all of the issues with the original application, while at the same time providing a few much-needed enhancements. The new shuttle application will have an updated user interface and icon, one whose colors and orientation represent Bard College to the best of its fashion. It will also include the local Campus Shuttle schedule and times, in addition to at least but not limited to the Weekend Train Shuttle Schedule, Area Shuttle Schedule and the Dutchess County Loop Bus Schedule that travels to and from Bard College. Several of these schedule additions are contingent on whether or not I will be able to find and/or construct a method by which the Bard College transportation department can update the schedule for the mobile application themselves. In the event that I cannot, the Campus Shuttle schedule, and Loop Bus schedule will be provided only. In addition, users will be able to set an alarm to remind themselves of their selected shuttle arrival or departure time for any and all shuttle schedules that are included in the application. The code for the application, which was programmed in Java will also be available for Bard College Affiliates via Github, an online open-source repository, should the need for any changes or improvements arise after my graduation. Lastly, I will utilize the Google Play Store, the official app store for Android smartphones and tablets to host the final version of my application for download.

1.3 Contributions

Though the idea for the project was conceived by myself, it would not have been possible without first taking a course on Android Mobile Application Development taught by Professor Sven Anderson. In addition, various open source tools such as Github, an online open-source repository that allowed me to store and share my code as needed, Android Studio, the official integrated development environment (IDE) for the Android platform by Google were all essential resources that without which, the project would not have been possible.

2

Background

2.1 Previous Bard Shuttle Application

Prior to beginning my project and initial design of the new shuttle app, I felt that it was crucial that I first examine the previous application in terms of its current design and how it operated, such as the methods used to store the shuttle data and the way in which it was coded. The previous Bard Shuttle app is presented below in Figures 2.0.1 and 2.0.2:

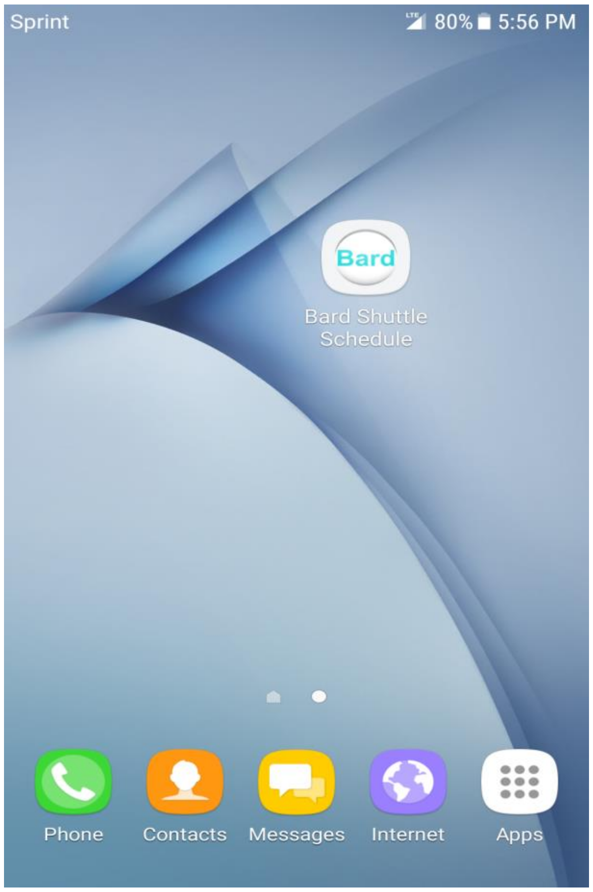
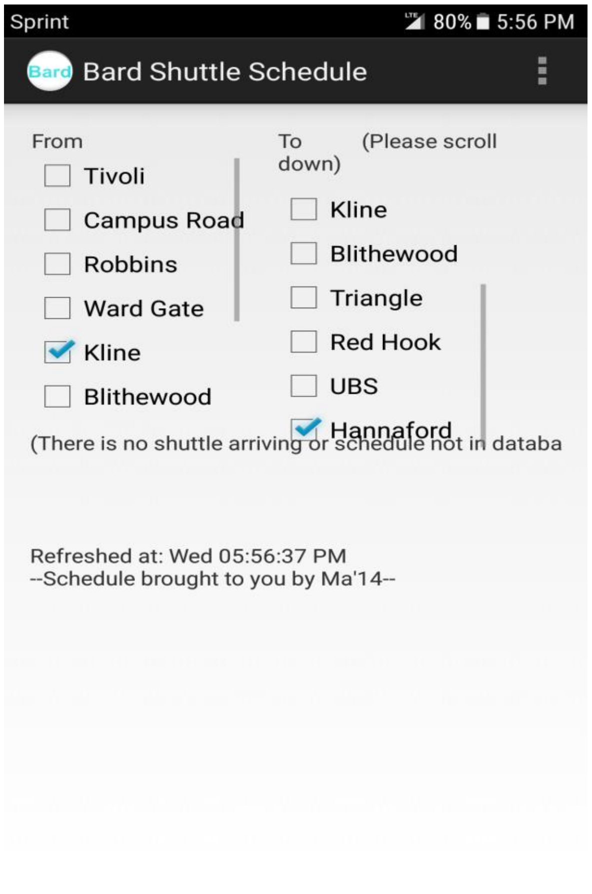


Figure 2.0.1. Bard Shuttle Schedule Icon



From
- Tivoli
- Campus Road
- Robbins
- Ward Gate
- [✓ Kline]
- Blithewood

To (Please scroll down)
- Kline
- Blithewood
- Triangle
- Red Hook
- UBS
- [✓ Hannaford]

(There is no shuttle arriving or schedule not in databa

Refreshed at: Wed 05:56:37 PM
--Schedule brought to you by Ma'14--

Figure 2.0.2. Bard Shuttle Schedule UI

The look, feel and goal of the previous application was quite simple. The user simply opened the application, selects their starting point and destination and the nearest arrival time for the shuttle would be displayed along with a list of all future arrival times. To my surprise, I later discovered that the earlier shuttle application had been created by a previous Computer Science major like myself for their senior project, who graduated years prior. This would make the application that they designed several years old. I wasn't unable to get into contact with the student that designed it, so as a result, I could not examine their code and programming of the previous application in depth. Fortunately, I noticed that the shuttle application was not connected to an online database and I surmised that the previous shuttle application was either using an SQLite database to store shuttle times or storing shuttle times within the application itself by some other means.

2.2 Resources and Tools

One of the reasons I assumed that the previous Bard Shuttle Application utilized an SQLite database was because the application did not require an internet connection to access or update the shuttle schedule. In addition, any changes that may have been made to the Campus Shuttle schedule over the past 4 years required users to download an application update from the Google Play Store. This means that the shuttle schedule was likely directly stored within the application itself, and as a result did not require a connection to the internet when users used the application.

SQL (pronounced "ess-que-el") stands for Structured Query Language and is used to communicate with a database. According to ANSI (American National Standards Institute), it is the standard language for relational database management systems. [2] SQLite, on the other hand, is a much more compact library. With all features enabled, the library size can be less than 500KB (Kilobytes), depending on the target platform, which is extremely small for a database. The most important advantage of SQLite databases is that they can be stored and read directly from the devices that may be stored on without the need of an online server, which in this case would the mobile app. [1]

In order for the project to be at all possible, it required a powerful yet intuitive enough development environment to code the application. The development environment of choice was Android Studio, the official integrated development environment for Android devices. Although coding the app single-handedly, the popularity of Android Studio made it easy to learn new coding concepts, how to use the environment, as well as its powerful design tools for the user interface. There were also an innumerable amount of texts, forums, and online tutorials available to help me cut my learning curve, but it was still very much a challenge. Lastly, I used a web-based version control repository known as Github to incrementally save any changes or advancement I made to the application during its creation. Since I wanted the opportunity for other students of Computer Science to be able to make changes or improvements to the application after my graduation, the use of Github was crucial. To clarify, other programmers won't be able to make changes to what I have already coded within the application, but they will be able to make copies of the code for their own use. This way the integrity of my own project remains very much intact. Changes that I make to the application's code on Github will not affect the final version of the new shuttle app that will be available on the Google Play Store where users will be able to download my application as well as millions of others.

2.3 SQL/SQLite and Online Database Connection

The ideal configuration for the new Bard Shuttle Application would be to connect the application to an online SQL database to access and display the shuttle times and schedules, while also allowing a small version of the database to be downloaded onto the device in the event that the device cannot connect to the internet to access the database. An SQL database is relational database meaning that it is structured to recognize relations among stored items of information. The relations that are shared among information within the database is entirely up to the user that designs it. For instance, imagine a supermarket had various customers and every customer was known by his or her customer ID and naturally every customer would have had a history of purchases that they have completed over a given course of time.

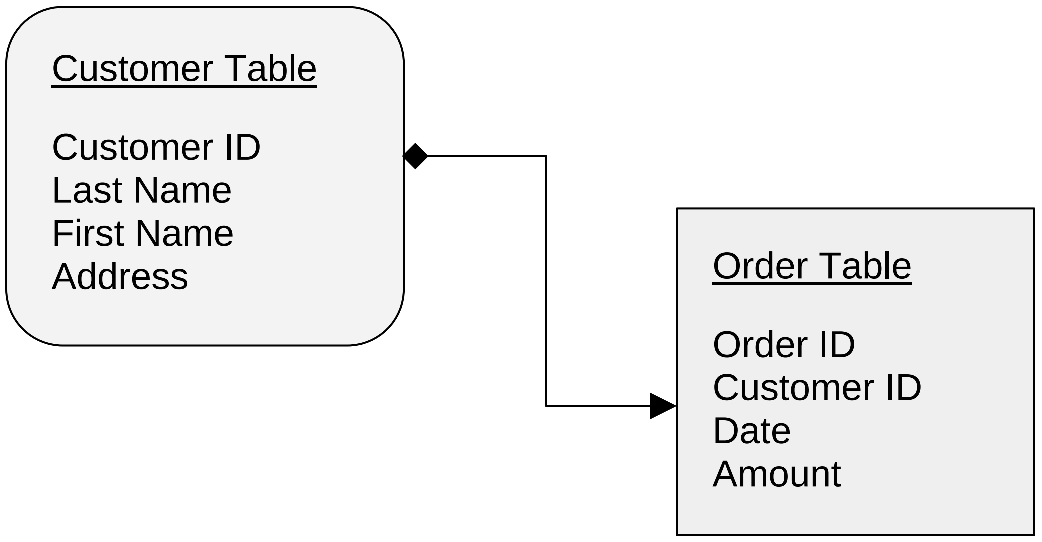


Figure 2.0.3 SQL Database Relationship Example

Take figure 2.0.3 above as a small example of our database that will consist of customers and orders. If a customer wanted to return an item to our store we would first need to access our database to verify their purchase. This would involve looking for their order via their customer ID and then simply finding and displaying every order associated with that customer's ID.

Finding information within a relational database like SQL involves having the user to Query the database. A query is an inquiry into the database using a SELECT statement. A query is used to extract data from the database in a readable format according to the user's request. [3] The SELECT statement for our example SQL database in Figure 2.0.3 would the following:

SELECT * FROM Order Table, Order ID, Date, Amount, WHERE Customer ID (ID used for search) LIKE (Matches) 'Customer ID'

Within this query, the asterisk simply selects all columns within the given table, so instead of the asterisk, one could simply have used Customer ID to specify that our search would be limited to that single column. Directly after the term "FROM" we specify which table we are going to pull the information we need from. In this case, we would need to find and read the "Order ID", "Date", and "Amount", FROM the Order Table as long the Customer ID within Order Table matches the one from our customer. As straightforward as accessing an SQL database may seem, looking up information in a much bigger database can become very complicated, very quickly. In the example used with simply a single customer and order, we only used two tables of information to encompass the database. If we were using a SQL database for a thousand or millions of customers the number of tables would dramatically increase.

2.4 Android Studio Integrated Development Environment (IDE)

Although there were other development environments to develop the application such as Eclipse, which has been and in some places still is a widely-used Android development environment, learning Android Studio offered a unique balance between simplicity and power. Android Studio offered great options to plan and customize the user interface (UI) of the new shuttle application, excellent organization of Java classes, consisting of thousands of lines of code, and also enabled me to connect directly to my personal repository on Github in order to save my daily or weekly changes and improvements to the application. This was extremely useful when my personal computer wasn't available. With Android Studio, I was able to securely download my project at its current state and continue to work on it on any Mac or PC that had Android Studio installed. (Appendix B) Using Android Studio's Android Emulator, I was able to test and run my application on a virtual android device directly on my desktop, eliminating the need for a physical android device during various stages of my project. (Appendix C) What was most useful about this tool was that since I would not know the current version of the Android operating system that my user's android device would have I believed it was important that my application was able to run on android devices and operating systems that may be several years old. Android Studio allowed me to emulate dozens of different android devices of varying capabilities in order to ensure backward and future compatibility of the new Bard shuttle application with newer Android devices.

2.5 Github

As the world's leading software development platform, with a community of over 21 million people and over 56 million projects being hosted, GitHub was the online repository of choice for my particular project. During the initial planning for the application, GitHub became a great source to find, test and look at other open-source shuttle mobile applications projects that had been done by other students and small organizations. One of the key features of GitHub that was crucial to building the shuttle application was Version Control. Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later. [4] This unique feature also lets you and others work together on projects from anywhere. These features also include the use of "Branches" that allow you or collaborators of the project to easily work on and save different versions of the same project or code in different "Branches". Since I was the only developer for my project I only required one main branch to maintain. Other branches would have been useful, but normally if I did not like a change that was made to the code I could easily reverse those changes using the Version Control system that was able to connect to the Android Studio IDE, via the internet. One would assume that it would useful to create at least two branches of my project one consisting of the version of the application geared toward connecting to an online server and another not. Instead, in order to make it a little simpler, I disabled certain functions and wrote the code in such a way that when an online connection to an SQL server is required modification to the current code will be easier. Most importantly, all code for the project is well commented and explained so functions written within the code will be easy to understand and modify as needed. A screenshot of this project's Github repository can be seen in Appendix D.

3
Mobile Application Design

Chapter 3 will provide initial design steps and implementation of the shuttle application. With thousands of lines of code spread across numerous files and Java classes, only samples of code used to implement the most important designs and functionalities will be discussed in detail. By the conclusion of this chapter, even those that are not avid programmers or familiar with databases will have a firm grasp of how the application operates systematically. More detailed views of the code will be provided in the Appendix.

3.1 Use Case Diagrams

In order to clearly illustrate the mobile application model, Use Case Diagrams are used. Use Case Diagrams are used simply depict the actions or sequence of actions that Users, the individual using the application, will take to use the mobile application. One Use Case would involve the android device of the user simply retrieving shuttle information directly from an SQL Database via an internet server as presented in Figure 3.0.1.

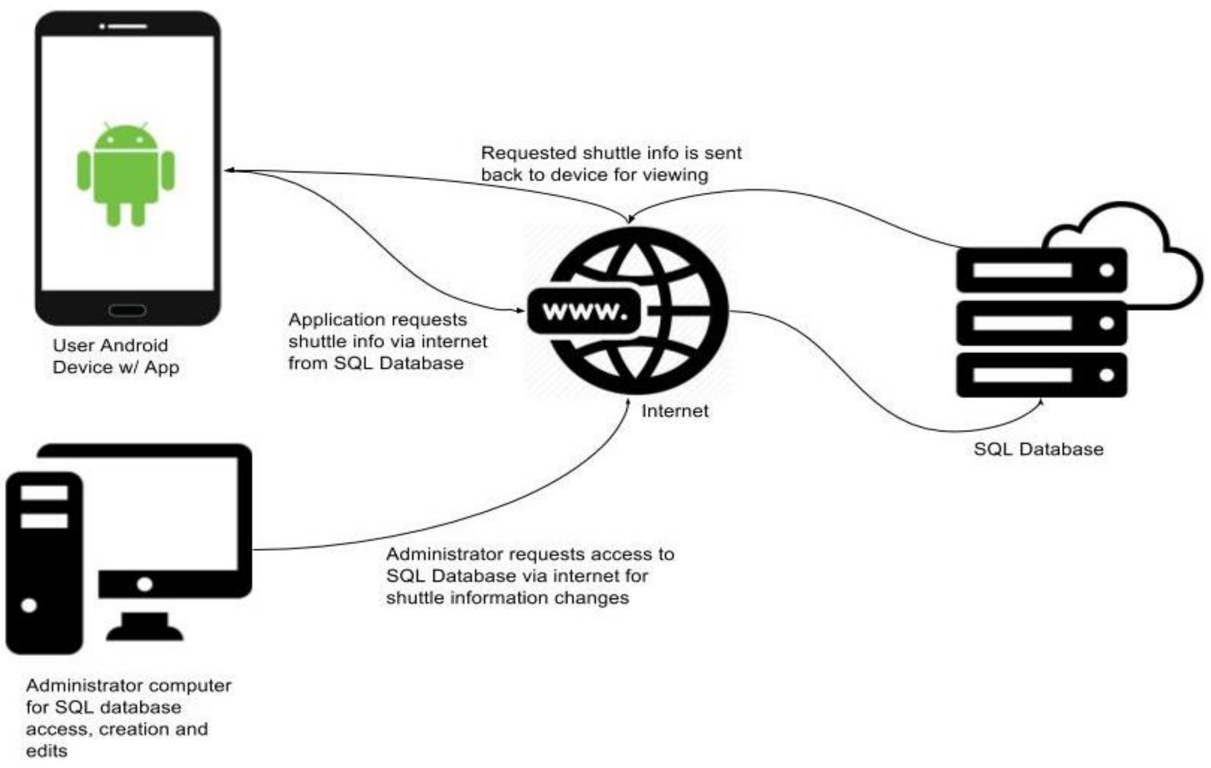

Requested shuttle info is sent back to device for viewing
User Android Device w/ App
Application requests shuttle info via internet from SQL Database
www.
Internet
SQL Database
Administrator requests access to SQL Database via internet for shuttle information changes
Administrator computer for SQL database access, creation and edits

Figure 3.0.1. Use Case Diagram for SQL Database Connection

Assuming that the SQL database has been created by the administrator via a laptop or desktop computer, they can view or edit the database when the need arises. With the shuttle application installed on their android mobile devices, users are able to request shuttle arrival times and other needed information from the SQL database via an internet connection. In addition, the user's shuttle application will download a copy of the shuttle database directly onto the device as well in the form of an SQLite database. This way users will have constant access to the database of shuttle schedules as a contingency if internet access is somehow not available.

Another Use Case involves a simpler setup, with the shuttle application only having access to the shuttle schedules via an SQLite database, a database directly stored within the shuttle application itself. Figure 3.0.2 below illustrates this Use Case well.

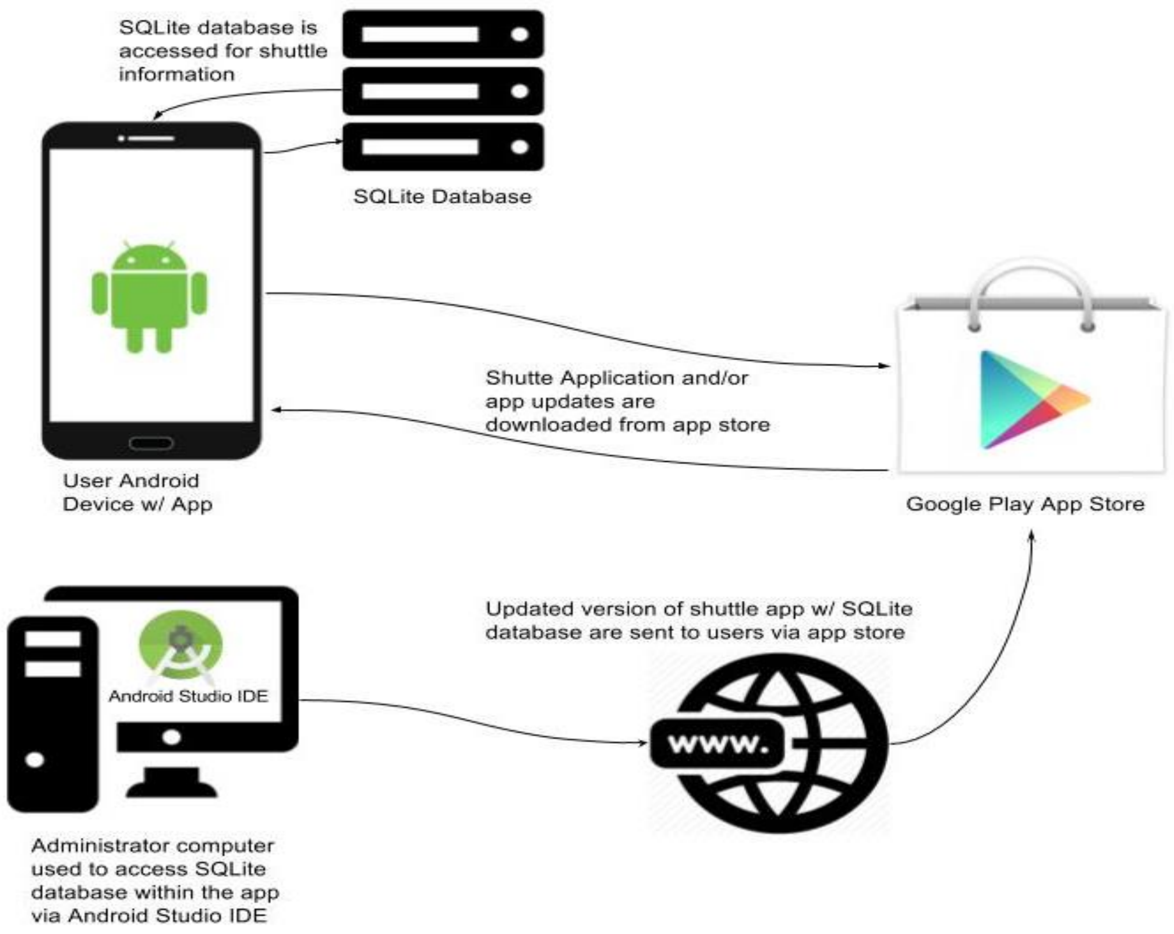
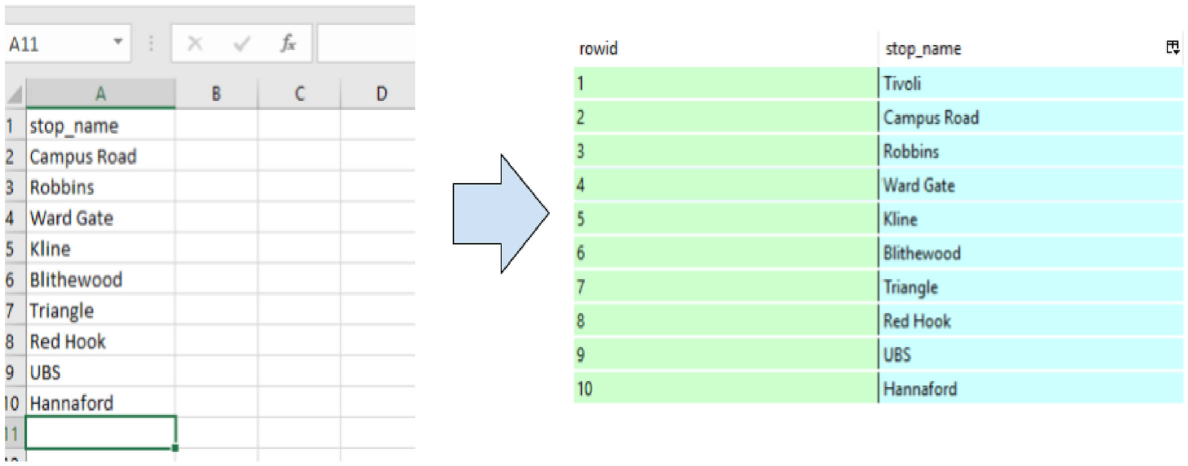

SQLite database is accessed for shuttle information
SQLite Database
User Android Device w/ App
Shutte Application and/or app updates are downloaded from app store
Google Play App Store
Android Studio IDE
Updated version of shuttle app w/ SQLite database are sent to users via app store
www.
Administrator computer used to access SQLite database within the app via Android Studio IDE

Figure 3.0.2. User Case Diagram for SQLite Database Connection

Unlike the previous Use Case, in order for the shuttle schedule to be updated, it will require the application's code to be accessed via the Android Studio IDE. This way the SQLite database that was created and edited on the administrator desktop can be stored within the application itself. Once changes to the shuttle schedule or another feature of the application are made the updated version of the application must be uploaded to the Google Play Store via an internet connection. In order for the user to have the updated version of the application or shuttle schedule, they must download the updated version of the application from the app store. This process may be slightly less efficient since one must re-download the entire shuttle application for a simple schedule change, but it is much easier to maintain without the need for an online server.

3.2 SQLite Database Setup & Structure

Typically, the creation of an SQLite database involves the use of command line prompts in order to create and add data to a database. This process can be extremely time-consuming especially if the database in question is quite large. In order to combat this, there are numerous free and paid tools available on the internet that can allow one to create and maintain an SQL or SQLite database easily without lengthy code writing. My tool of choice for the creation of the database for this project was SQLite Manager, an add-on program that is used in conjunction with Mozilla Firefox. As the name suggests SQLite Manager allows a user to manage any SQLite database on their respective computer. This tool allowed me to cut the creation time of the shuttle schedule database in half by allowing me to input shuttle times into a Microsoft Excel Comma Separated Value file or CSV for short. Then SQLite Manager allows me to then convert the CSV file into an SQLite database with a few simple clicks. Once the file is converted and open within SQLite Manager the database can be edited easily using its simple interface. (Figure 3.0.26)



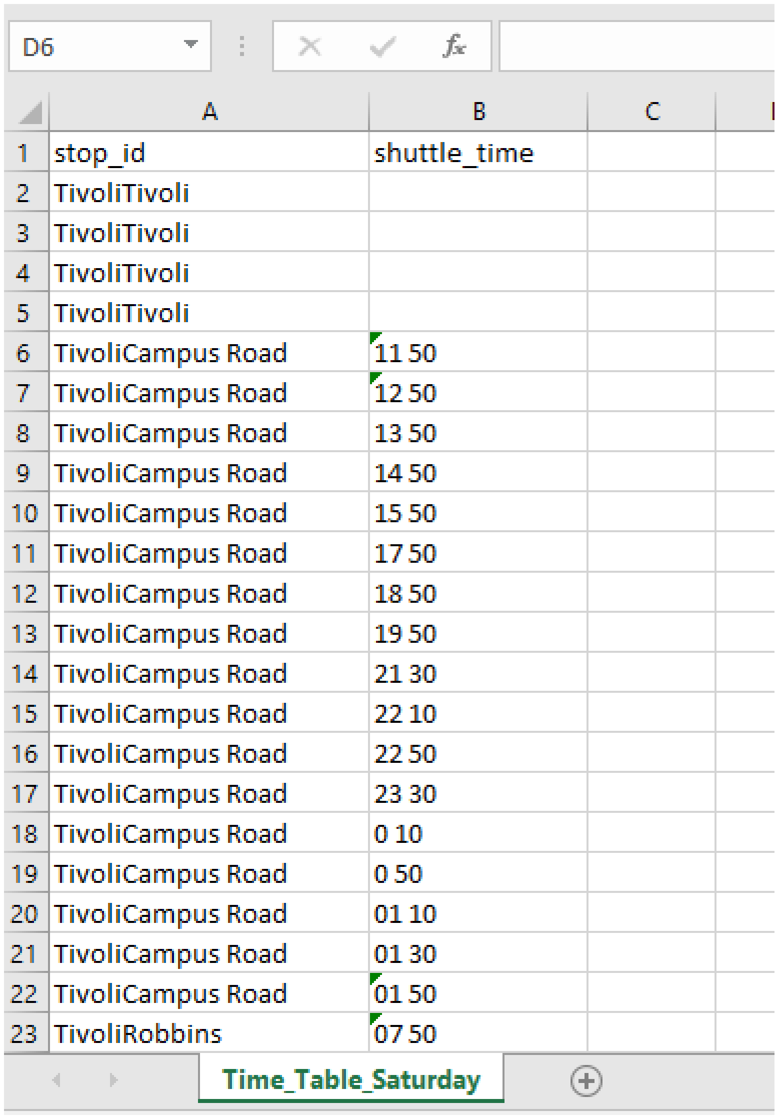
	A	B	C	D
1	stop_name			
2	Campus Road			
3	Robbins			
4	Ward Gate			
5	Kline			
6	Blithewood			
7	Triangle			
8	Red Hook			
9	UBS			
10	Hannaford			
11				

rowid	stop_name
1	Tivoli
2	Campus Road
3	Robbins
4	Ward Gate
5	Kline
6	Blithewood
7	Triangle
8	Red Hook
9	UBS
10	Hannaford

Figure 3.0.26 Stops_Table for shuttle stop names in excel (Left), Converted table in SQLite Manager (Right)

Once the file was converted I could still easily reorganize the data as I saw fit and add stop names as needed. A function was written within the code that accesses the database for the names of the campus shuttle stops to be used as selection options within the application.

Using a copy of the Bard College shuttle schedule available on the Bard Transportation website I simply typed in the shuttle times into Microsoft Excel in the format shown in Figure 3.0.27.



	A	B	C
1	stop_id	shuttle_time	
2	TivoliTivoli		
3	TivoliTivoli		
4	TivoliTivoli		
5	TivoliTivoli		
6	TivoliCampus Road	11 50	
7	TivoliCampus Road	12 50	
8	TivoliCampus Road	13 50	
9	TivoliCampus Road	14 50	
10	TivoliCampus Road	15 50	
11	TivoliCampus Road	17 50	
12	TivoliCampus Road	18 50	
13	TivoliCampus Road	19 50	
14	TivoliCampus Road	21 30	
15	TivoliCampus Road	22 10	
16	TivoliCampus Road	22 50	
17	TivoliCampus Road	23 30	
18	TivoliCampus Road	0 10	
19	TivoliCampus Road	0 50	
20	TivoliCampus Road	01 10	
21	TivoliCampus Road	01 30	
22	TivoliCampus Road	01 50	
23	TivoliRobbins	07 50	

Figure 3.0.27 Campus Shuttle times for Saturday via Microsoft Excel

The structure that was chosen for the database was largely contingent on what would work most seamlessly with Java and what would require less code overall. As seen in Figure 3.0.27 the shuttle times were saved in a 24hr format so that I would be able to easily determine for users which shuttle times were in the Morning and Afternoon. This option allowed coding to be much simpler. Had I chosen to store the shuttle times in a 12hr format or added an ‘AM’ or ‘PM’ next to the shuttle time, I wouldn’t have been able to perform simple computation needed for the shuttle alarm functionality because the shuttle times would be read as Strings and not Integers by Java. Although I could have written more code to simply locate the number within each cell and convert them into integers, it would have required more complicated and lengthy code to be written.

Because SQL is a relational database each set of shuttle times needed to correspond with a particular start and destination point. That is why in Figure 3.0.27 each cell within the *stop_id* column contains the name of a potential start and destination point that may be chosen by the user. Put simply, the list of the shuttle times shown to the user is dependent upon which start and destination point they choose. So, if I users starting point was Tivoli and their destination point was Campus Road, every shuttle arrival time pertaining to that combination of shuttle stop names would be presented to the user. The only drawback was the amount of time it took to type the shuttle information within each cell since there are ten shuttle stops and the arrival times can range from none to 20, just over one thousand rows of data were needed for the Campus Shuttle database. The initial creation of the database was tedious, but adding or deleting new stops and shuttle times won’t nearly take as much time.

Lastly, in another effort to simplify the Java code for accessing the database every shuttle schedule was separated into separate tables based on the day of the week and whether or not the current semester was Fall or Spring. An example of this can be seen in Figure 3.0.28.

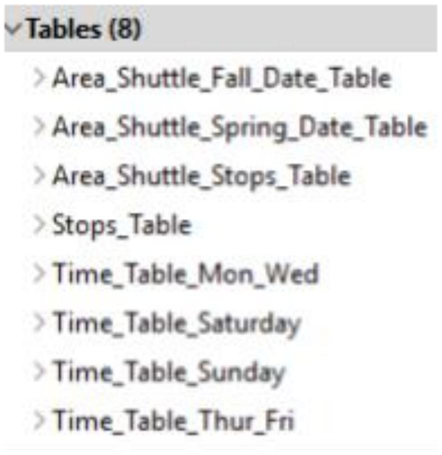

Tables (8)
- Area_Shuttle_Fall_Date_Table
- Area_Shuttle_Spring_Date_Table
- Area_Shuttle_Stops_Table
- Stops_Table
- Time_Table_Mon_Wed
- Time_Table_Saturday
- Time_Table_Sunday
- Time_Table_Thur_Fri

Figure 3.0.28 List of tables in test database via SQLite Manager UI

Taking this approach allowed me to steer clear of using extremely long and complicated SQL query SELECT statements to sort through the data had I included all the shuttle times in one giant table.

3.3 User Interface (UI) Design

The user interface of the previous Bard Shuttle Application was great in terms of its simplicity but was lacking largely in terms of the number of the shuttle schedules and functionality offered to users. If the new shuttle application was going to include more features it needed a slightly more robust user interface. I first began my design with the initial shuttle selection screen that would be presented to the users upon opening the application as presented in Figure 3.0.3.

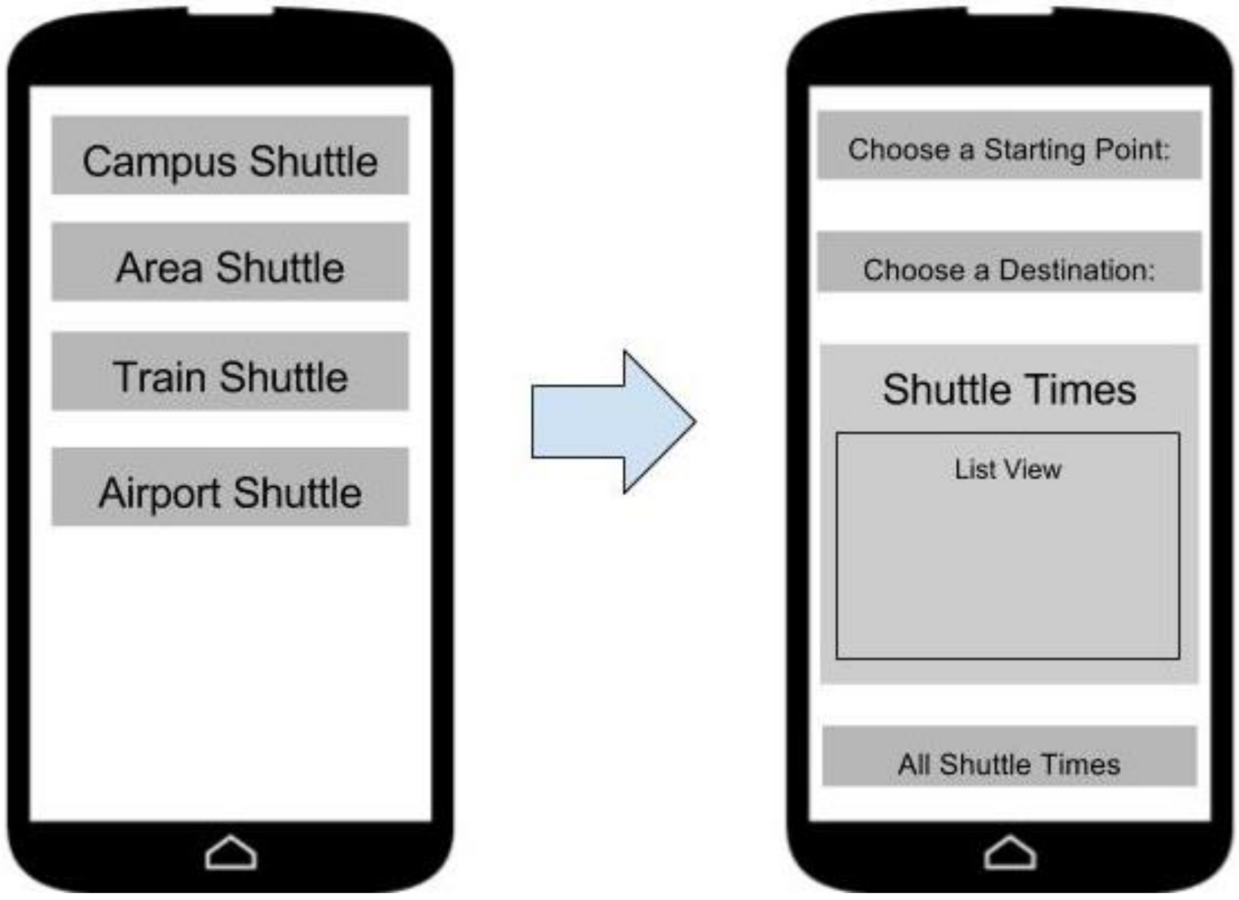


Figure 3.0.3 Shuttle Selection Menus

Although the final User Interface turned out slightly different, many aspects remained the same. A user would simply choose which shuttle service they needed and would then be presented with an option to choose a starting point and destination. Once their start and destination point is chosen users will then be presented with a Listview that displays a particular shuttle's arrival times for the day. A Listview is simply a view that shows items in a vertically scrolling list. If a User wishes to view the full list of shuttle arrival times that may have already past then they will have the option of viewing that list via the "All Shuttle Times" button.

Furthermore, in the event that a user wanted more shuttle information that the application did not directly provide I wanted users to be able to easily visit the Bard College Transportation website. The website would include temporary changes to the Shuttle Schedules and/or delays due to inclement weather that the shuttle application will not have. An easy way to implement this would be simply providing another button that opens a web browser already installed on the user's device that takes them to the transportation website. Already, the primary selection menu is cluttered with arguably an unappealing display of simple buttons but my final implementations on section 3.4 address those concerns.

3.4 User Interface Technical Implementation

Ultimately, I decided that the best option for the initial shuttle selection menu was a Navigation Drawer and Alert Dialogs for starting and destination point selection as demonstrated in Figures 3.0.4 and 3.0.5.

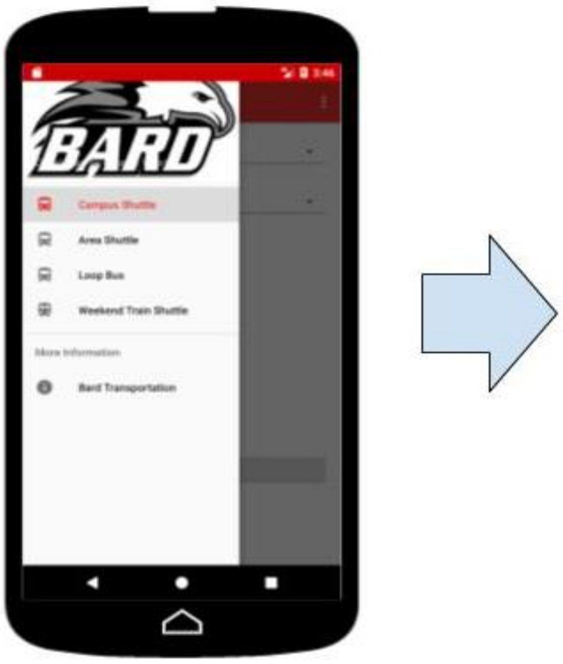


Figure 3.0.4 Navigation Drawer (Left) & Starting/Destination Point Selection (Right)

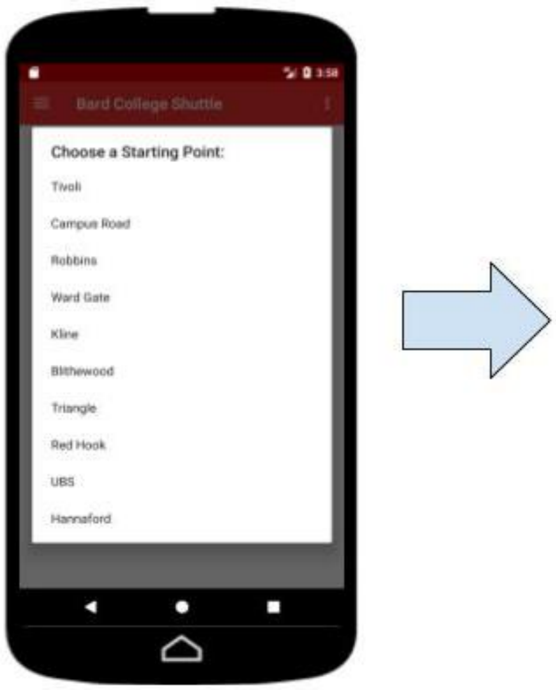


Figure 3.0.5 Start and Destination AlertDialog (Left) and Shuttle Times Listview (Right)

In order to implement the Navigation Drawer as presented in Figure 3.0.4, construction of its own independent Java Class named *ShuttleSelectionDrawer* was necessary. This Java Class provides any Java code that deals with user interactions and functionality of the Navigation Drawer. The code presented in Figure 3.0.6 are all components, if you will, of the function

```
public class ShuttleSelectionDrawer extends AppCompatActivity
        implements NavigationView.OnNavigationItemSelectedListener {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_shuttle_selection_drawer); //
        Toolbar toolbar = (Toolbar) findViewById(R.id.toolbar);
        setSupportActionBar(toolbar);

        // Default built-in Navigation Drawer layout
        DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
        ActionBarDrawerToggle toggle = new ActionBarDrawerToggle(this, drawer,toolbar,
                R.string.navigation_drawer_open, R.string.navigation_drawer_close);
        drawer.setDrawerListener(toggle);
        toggle.syncState();
        NavigationView navigationView = (NavigationView)findViewById(R.id.nav_view);
        navigationView.setNavigationItemSelectedListener(this);
        navigationView.getMenu().getItem(0).setChecked(true);

        //Opens Navigation Drawer as soon as App starts by selecting the Campus
Shuttle option first
        FragmentManager fragmentManager = getFragmentManager();
        fragmentManager.beginTransaction().replace(R.id.content_frame, new
CampusShuttle()).commit();
        drawer.openDrawer(Gravity.LEFT); //Navigation Drawer open from left side of
screen.
    }
```

Figure 3.0.6 onCreate() function of ShuttleSelectionDrawer Java Class onCreate()

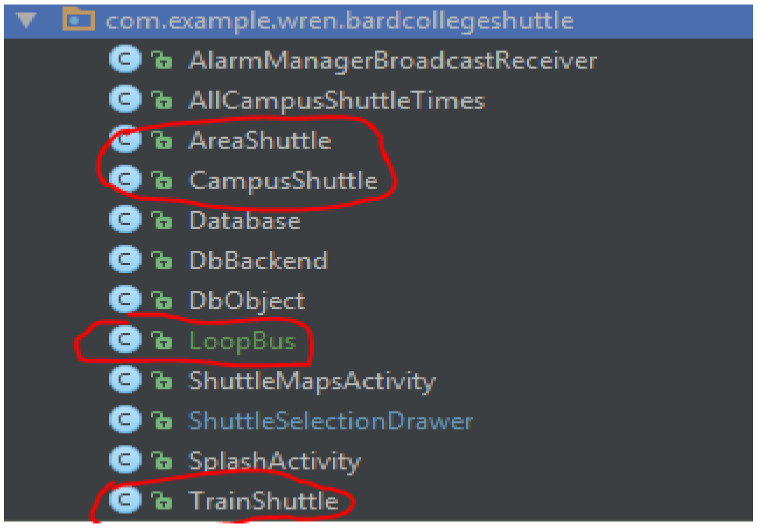
In congruence with several other Java classes, when the application is started all of the Layouts (visual structure for a user interface presented in an XML file) are invoked and the Navigation Drawer consisting over our custom shuttle options is presented. This single function also allows the developer to change whether the Navigation Drawer appears on the left side of the device screen as seen in Figure 3.0.4 or on the right side. Even more important, the placement of buttons, a background picture, and general layout aspects are defined in separate XML (Layout) files. The options to choose a particular shuttle are defined within the *activity_shuttle_selection_drawer_drawer.xml* file presented below.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android">

    <group android:checkableBehavior="single">
        <item
            android:id="@+id/activity_campus_shuttle"
            android:icon="@drawable/ic_directions_bus_black_24dp"
            android:title="Campus Shuttle"/>
        <item
            android:id="@+id/activity_area_shuttle"
            android:icon="@drawable/ic_directions_bus_black_24dp"
            android:title="Area Shuttle"/>
        <item
            android:id="@+id/activity_loop_bus"
            android:icon="@drawable/ic_directions_bus_black_24dp"
            android:title="Loop Bus"/>
        <item
            android:id="@+id/activity_train_shuttle"
            android:icon="@drawable/ic_train_black_24dp"
            android:title="Weekend Train Shuttle"/>
    </group>

    <item android:title="More Information">
        <menu>
            <item
                android:id="@+id/transportation_website"
                android:icon="@drawable/ic_info_black_24dp"
                android:title="Bard Transportation"/>
        </menu>
    </item>

</menu>
```

Figure 3.0.7 activity_shuttle_selection_drawer_drawer.xml

One can observe in the XML code, that every shuttle option within the Navigation Drawer in Figure 3.0.4 directly corresponds with the items defined with the *activity_shuttle_selection_drawer_drawer.XML* file. Other things such as icons, text size and color, and selection order can all be modified within this file and others like it.

Upon selection of a shuttle service, users will then need to be redirected to the Activity that allows them to view shuttle times for a particular shuttle service. More importantly, every shuttle option presented in Figure 3.0.4 and the *activity_shuttle_selection_drawer_drawer.XML* file corresponds with a separate Java Class as seen in Figure 3.0.8.



```
com.example.wren.bardcollegeshuttle
    AlarmManagerBroadcastReceiver
    AllCampusShuttleTimes
    AreaShuttle
    CampusShuttle
    Database
    DbBackend
    DbObject
    LoopBus
    ShuttleMapsActivity
    ShuttleSelectionDrawer
    SplashActivity
    TrainShuttle
```

Figure 3.0.8 Application Classes via Android Studio IDE

A function located within our *ShuttleSelectionDrawer* Java Class named *onNavigationItemSelected*, respectively, that is primarily comprised of conditional if statements are programmed to open a particular shuttle class upon being pressed, such as the *CampusShuttle*, *LoopBus*, *AreaShuttle* and *TrainShuttle*. In order to provide the best

```
@SuppressWarnings("StatementWithEmptyBody")
@Override
public boolean onNavigationItemSelected(MenuItem item) {
    // Handle navigation view item clicks here.
    int id = item.getItemId();
    FragmentManager fragmentManager = getFragmentManager();

    if (id == R.id.activity_campus_shuttle) {
        fragmentManager.beginTransaction().replace(R.id.content_frame, new
CampusShuttle()).commit();
    } else if (id == R.id.activity_area_shuttle) {
        fragmentManager.beginTransaction().replace(R.id.content_frame, new
AreaShuttle()).commit();
    } else if (id == R.id.activity_train_shuttle) {
        fragmentManager.beginTransaction().replace(R.id.content_frame, new
TrainShuttle()).commit();
    }  else if (id == R.id.transportation_website) {
        Intent intent = new Intent (Intent.ACTION_VIEW,
Uri.parse("http://blogs.bard.edu/transportation/"));
        startActivity(intent);
    }

    DrawerLayout drawer = (DrawerLayout) findViewById(R.id.drawer_layout);
    drawer.closeDrawer(GravityCompat.START);
    return true;
}
```

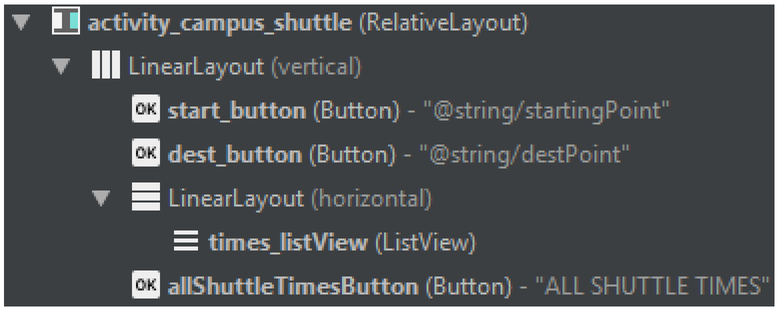
Figure 3.0.9 onNavigationItemSelected() function in ShuttleSelectionDrawer Java Class

user experience and further future-proof the application for easy and possibly more robust changes in the future I chose to have each shuttle service be a separate Fragment. A Fragment is usually used as part of an activity's user interface and contributes its own layout to the activity. To provide a layout for a fragment, you must implement the *onCreateView()* callback method, which the Android system calls when it's time for the fragment to draw its layout.[5] Although there is no rule that you should use only use fragments as opposed to Activities (a single screen with a user interface just like window or frame), Google states that it is much better to use fragments wherever it is possible. This is primarily for the sake of simplicity. An Activity is a single screen with a user interface, meaning that you can only display one single activity on the screen at a time. In order to switch to another activity, they must kill their current activity and start another.

Rather than start a brand-new activity every time, every shuttle service fragment is in a sense swapped out with another when its corresponding button is pressed. Imagine having a wall in a home that only has enough space for one picture frame, but you only have room for one picture and every time you wanted to see a new picture on the wall you would have to swap out the entire picture frame that contains a new photo. Rather than buying a new frame for every picture, Fragments allow you to reuse a single frame but simply swapping the picture when you want to view another. Hence, Fragments act in the same way as Activities on a surface level but operate differently on a systematic level.

Furthermore, Fragments also allow multiple activities to run simultaneously on a single screen, depending on a developer's need. In the code shown in Figure 3.0.9, each Java Class created for a shuttle service is its own Fragment and each fragment is called upon or opened when its corresponding button is clicked. There aren't any clear gains in performance, but given a few modifications to the code utilizing fragments will allow easier User Interface compatibility with Android Tablets, which have much bigger screens and are more capable of handling more on-screen activities given the bigger screen real estate.

In order to select a starting and destination point. I decided to use an AlertDialog. An AlertDialog box is a special dialog box that is displayed in a graphical user interface typically when something unexpected has occurred that requires immediate user action. [6] In our case, the display of the AlertDialog box will be completely expected and won't necessarily be used in the case of an error. Instead, I have used the AlertDialog box as a means for users to choose their starting point and destination. Within the *activity_campus_shuttle.XML* file in Figure 3.0.10 the final layout for the application is presented and completely follows my initial plans for the layout presented earlier in section 3.3.



```
▼ activity_campus_shuttle (RelativeLayout)
    ▼ LinearLayout (vertical)
        start_button (Button) - "@string/startingPoint"
        dest_button (Button) - "@string/destPoint"
        ▼ LinearLayout (horizontal)
            times_listView (ListView)
        allShuttleTimesButton (Button) - "ALL SHUTTLE TIMES"
```

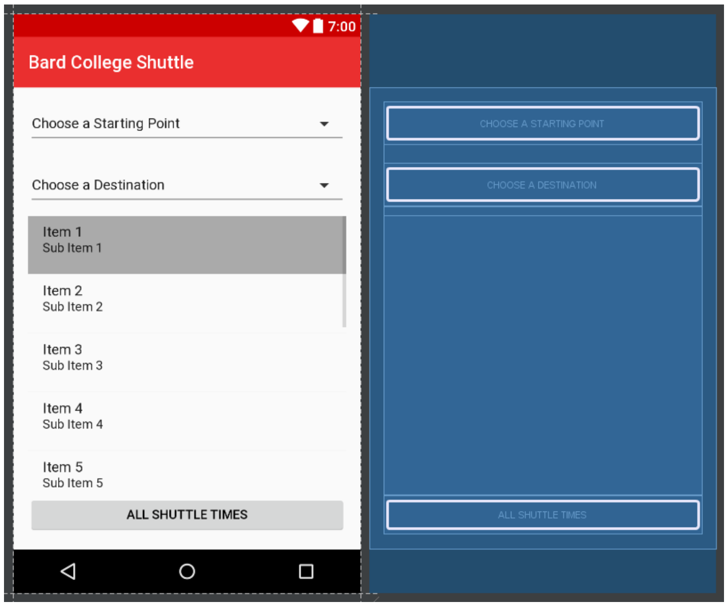


Figure 3.0.10 Campus Shuttle Layout via activity_campus_shuttle.xml

It is important to note that I did, in fact, use Buttons for users to press when choosing their start and destination point. The reason I am bringing this to your attention is because to more experienced android application developers it would appear that I am utilizing what are called Spinners as my means of start and destination selection as indicated by the small downwards pointing arrows. (Figure 3.0.10) In short, Spinners are simply drop down menus in which a user may simply view or select from a list of given options. The only drawback to using Spinners is the difficulty of using multiple spinners simultaneously in my implementation. The first problem with using two Spinners was that upon choosing a shuttle service the application would automatically select the first option from the list of choices a user had to pick from. Secondly, in order to fix the first issue and also have each Spinner prompt the user to choose an option, it would involve heavily modifying the built-in code for the Spinners. Thirdly, since I wanted to pull shuttle times from the SQLite database based on two options, the start and destination point. Using Spinners would only allow the user to request shuttle times from the database once and wouldn't allow the user to request other shuttle times until they restarted the application. To fix this issue would involve writing more code and modifying more built-in functions.

In order to avoid such problems, I decided to use two Buttons and upon pressing either Button an AlertDialog is presented. This AlertDialog box then prompts the user to choose a starting point and destination from the list provided as presented in Figure 3.0.11.

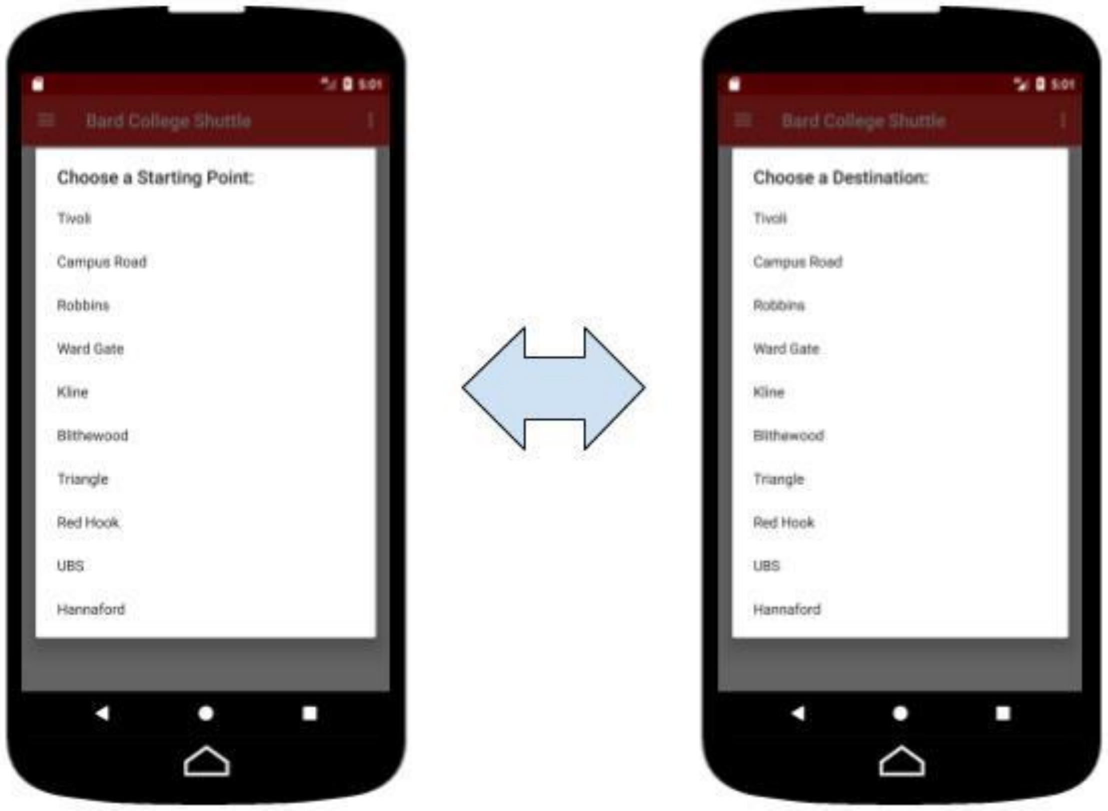


Figure 3.0.11 Alertdialog for Starting and Destination Selection

Once the user chooses their start and destination, the SQLite database is accessed and the user is presented with a list of future shuttle arrival times beginning with the nearest time first.

This is demonstrated in Figure 3.0.12.

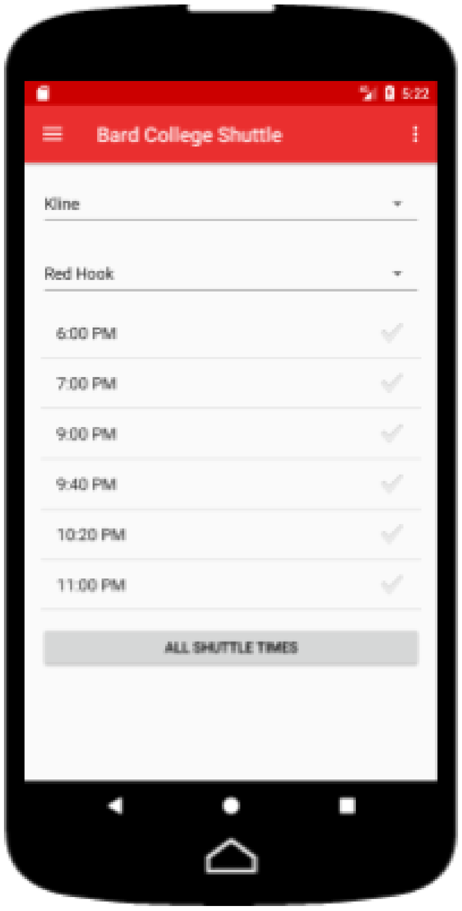


Figure 3.0.12 available shuttle arrival based on time and start/destination points

Upon choosing their start and destination, the shuttle times presented to the user will differ depending on the current time of day and the particular day of the week. This ensures that useless shuttle times (i.e. shuttle times that have passed) are not displayed. If there are more available shuttle times than what the user's screen can accommodate then users will be able to scroll through the list of times up or down with simple finger gestures. The XML code for the Listview for the full layout will be shown in Figure 3.0.13. Since the XML code and layouts of all shuttle services are very similar, the XML code for the Campus Shuttle Service will only be presented here. An individual will only need a firm grasp of a single layout in order to master them all, intuitively.

```xml
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/activity_campus_shuttle"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context="com.example.wren.bardcollegeshuttle.CampusShuttle">
    <LinearLayout
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:orientation="vertical"
        android:weightSum="1"
        tools:ignore="UselessParent">
        <Button
            android:id="@+id/start_button"
            style="@style/Widget.AppCompat.Spinner.Underlined"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:text="@string/startingPoint"
            android:textSize="16sp"
            />
        <Button
            android:id="@+id/dest_button"
            style="@style/Widget.AppCompat.Spinner.Underlined"
            android:layout_width="match_parent"
            android:layout_height="wrap_content"
            android:layout_marginTop="20dp"
            android:text="@string/destPoint"
            android:textSize="16sp"/>
        <LinearLayout
            android:layout_width="match_parent"
            android:layout_height="319dp"
            android:orientation="horizontal">
            <ListView
                android:id="@+id/times_listView"
                android:layout_width="match_parent"
                android:layout_height="match_parent"
                android:layout_marginTop="10dp"
                android:choiceMode="singleChoice"
                android:listSelector="@android:color/darker_gray"/>
        </LinearLayout>
        <Button
            android:id="@+id/allShuttleTimesButton"
            android:layout_width="match_parent"
            android:layout_height="43dp"
            android:layout_weight="0.82"
            android:text="ALL SHUTTLE TIMES"
            android:textColor="#000000"/>
        </LinearLayout>
</RelativeLayout>
```

Figure 3.0.12 Full Campus Shuttle Service XML Layout via activity_campus_shuttle.xml

3.5 SQLite Database Implementation

In the final implementation of the shuttle application, an SQLite database was used. Implementation of an SQLite database involved the creation and use of three particular classes: *Database*, *DbBackend*, and *DbObject*, all of which are interconnected and rely on an exchange

```
public class Database extends SQLiteAssetHelper {

    private static Database mInstance = null;

    private static final String DATABASE_NAMES = "TestDB5.sqlite";

    private static final int DATABASE_VERSION = 1;

    public static Database getInstance(Context ctx) {

        // Use the application context, which will ensure that you
        // don't accidentally leak an Activity's context.
        // See this article for more information: http://bit.ly/6LRzfx
        if (mInstance == null) {
            mInstance = new Database(ctx.getApplicationContext());
        }
        return mInstance;
    }

    public Database(Context context) {
        super(context, DATABASE_NAMES, null, DATABASE_VERSION);
    }
}
```

Figure 3.0.13 Java code for Database class

of necessary information concerning the database in order to access it. The *Database* class presented in Figure 3.0.13.

The *Database* class although small is often required in order to utilize an SQL or SQLite database. It only consists of a single function that is designed to gather the necessary information of a database before attempting to establish a connection. The function requires a database name which is ‘TestDB5.SQLite”, a version number which is ‘1’, and the context, which is the current state of the database. When attempting to use an SQLite database one only needs to have a copy of this class and provide the name of the database in which they are using. The SQLite file must be stored in the “Assets” folder within Android Studio after opening the project. This is also shown in Figure 3.0.14. The folder paths within Figure 3.0.14 can also be manually created by the user if they are not already present.

The code within the *DbObject* class in Figure 3.0.15 simply establishes a connection to the SQLite database. The code operates in three simple steps. First, it gathers the required

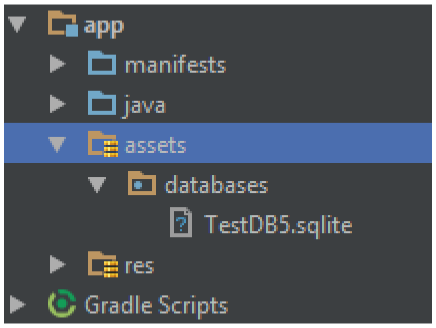


Figure 3.0.14 Location of SQLite Database file via Android Studio Project

database information we collected in the previous *Database* class within the *DbObject* function via *dbhelper = new Database (context)*, which references our previous *Database* class. Secondly, the following line of code, *this.db = dbHelper.getReadableDatabase();*, pulls the “readable” database found in the *Database* class and stores it into the variable *this.db*. Lastly, the connection to the SQLite database “TestDB5.sqlite” is established via the *getDbConnection*() function, in which *this.db*, the variable set to our “readable” database, is returned and the data is ready to be utilized.

```
public class DbObject {
    public static Database dbHelper;
    private SQLiteDatabase db;

    public DbObject(Context context) {
        dbHelper = new Database(context);
        this.db = dbHelper.getReadableDatabase();
    }

    public SQLiteDatabase getDbConnection(){
        return this.db;
    }

    public void closeDbConnection(){
        if(this.db != null){
            this.db.close();
        }
    }
}
```

Figure 3.0.14 Java code for DbObject Class

Once the connection to our database has been established we can then begin viewing and even in some cases changing or manipulating data. For the purposes of this mobile application no code was written that helps add new data to our database. The code has only been written simply to access the database, read and sort the data, use it for needed calculations, filter out useless data and then allow the user to view it. The data within the SQLite database used for this mobile application are simply names of shuttle stops, dates and times. Utilization of the data read from the database will be made possible from functions created in *DbBackend* class shown in Figure 3.0.15.

Similar to that of the XML layouts of the shuttle service, each shuttle service will follow a similar coding format as well. In order to fully complete each shuttle service, the first step was to write functions to populate the shuttle stops by accessing the database. In Figure 3.0.16 there are two functions that populate the shuttle stops, one for the starting points and one for the destination points.

```
public String populateStartStops(){
    final DbBackend dbBackend = new DbBackend(getActivity());
    final String[] stopLists = dbBackend.getShuttleStops(databaseTableName);
//populates list of stops
    Button startButton = (Button) getActivity().findViewById(R.id.start_button);
    startButton.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
            AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
            builder.setTitle("Choose a Starting Point:");
            builder.setItems(stopLists, new DialogInterface.OnClickListener() {
                        @Override
                        public void onClick(DialogInterface dialog, int which) {
                            // the user clicked on stopLists[which]
                            startStop = stopLists[which].toString();
                            TextView startButtonTextView = (TextView)
getActivity().findViewById(R.id.start_button);
                            startButtonTextView.setText(startStop);
                            startButtonClicked = true;
                            twoButtonClicks();
                        }
                    }
            );
            builder.show();
        }
    });
    return startStop;
}

public String populateDestStops(){
    final DbBackend dbBackend = new DbBackend(getActivity());
    final String[] stopLists = dbBackend.getShuttleStops(databaseTableName);
//populates list of stops
    Button destButton = (Button) getActivity().findViewById(R.id.dest_button);
    destButton.setOnClickListener(new View.OnClickListener() {
        public void onClick(View view) {
            AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
            builder.setTitle("Choose a Destination:");
            builder.setItems(stopLists, new DialogInterface.OnClickListener() {
                        @Override
                        public void onClick(DialogInterface dialog, int which) {
                            // the user clicked on stopLists[which]
                            destStop = stopLists[which].toString();
                            TextView destButtonTextView = (TextView)
getActivity().findViewById(R.id.dest_button);
                            destButtonTextView.setText(destStop);
                            destButtonClicked = true;
                            twoButtonClicks();
                        }
                    }
            );
            builder.show();
        }
    });
    return destStop;
}
```

Figure 3.0.16 Functions that populate AlertDialogs for Start and Destination stops via CampusShuttle class

Since the *DbBackend* class is the class needed to access and use the data gathered from the SQLite database, a method to access functions between separate Java classes is needed. Within the Campus Shuttle class the following lines of code:

```
final DbBackend dbBackend = new DbBackend(getActivity());
final String[] stopLists = dbBackend.getShuttleStops(databaseTableName); //populates
list of stops
```

allows one to access the *getShuttleStops()* function located within the *DbBackend* class (Figure 3.0.17) that populates the AlertDialogs in the CampusShuttle class with the names of the shuttle stops.

```
public class DbBackend extends DbObject {
    public DbBackend(Context context) {super(context);}

    // Function to populate dialog window with Bard College Shuttle Stops for selected
database
    public String[] getShuttleStops(String databaseTableName){
        String query = "Select * from '"+databaseTableName+"'";
        Cursor cursor = this.getDbConnection().rawQuery(query, null);
        ArrayList<String> spinnerContent = new ArrayList<String>();
        if(cursor.moveToFirst()){
            do{
                String word =
cursor.getString(cursor.getColumnIndexOrThrow("stop_name"));
                spinnerContent.add(word);
            }
            while(cursor.moveToNext());
        }
        cursor.close();
        String[] allStops = new String[spinnerContent.size()];
        allStops = spinnerContent.toArray(allStops);
        return allStops;
    }
```

Figure 3.0.17 getShuttleStops() function via DbBackend class

As mentioned in previous sections, shuttle information of the different shuttle services have been stored within the same database, but simply within different tables. This way the data is simple and well organized. In order for the *CampusShuttle* class to call the *getShuttleStops()* function and populate the necessary AlertDialogs, a specific table name is needed. For Campus Shuttle stops the table "Stops_Table" is the table that needs to be accessed within the database. Once the name of the table that needs to be accessed is provided then the function *getShuttleStops()* is called and the list of shuttle stop names is stored within a list. Next, that list is returned to the *CampusShuttle* class where the function was called from and the code shown in Figure 3.0.16 simply acts to display that list in the AlertDialog. The AlertDialog is also programmed to respond to selections made by the user that allows them to choose their desired starting point and destination. Lastly, the list of next shuttle arrival times is only populated on screen if both a starting point and destination are selected.

Furthermore, the SQLite database must once again be accessed to present shuttle arrival times to the user. This is done by the *populateFutureTimes()* function shown in Figure 3.0.18.

```
public void populateFutureTimes(){
    ListView popFutureTimesListView = (ListView)
getActivity().findViewById(R.id.times_listView);
    final DbBackend dbBackend = new DbBackend(getActivity());
    final String [] newTimes = dbBackend.getFutureTimesForStartAndDest(startdestStop);
//populates ListView
    final ArrayAdapter<String> timeAdapter = new ArrayAdapter<String>(getActivity(),
android.R.layout.simple_list_item_checked, newTimes);
    popFutureTimesListView.setAdapter(timeAdapter);
    popFutureTimesListView.setOnItemClickListener(new
AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position, long
id) {
            Toast.makeText(getActivity(), newTimes[position] + " Shuttle Selected",
Toast.LENGTH_SHORT).show();
            setAlarmDialogBox(newTimes[position]);
        }
    });
}
```

Figure 3.0.18 populateFutureTimes() function via CampusShuttle class

This function acts extremely similar to get *populateShuttleStops()* function, only this time a list of future shuttle arrival times is presented to the user within a Listview. Retrieving future shuttle arrival times is done by the *getFutureTimesForStartAndDest()* function located in the

DbBackend class shown in Figure 3.0.19.

```
//Function to List Future Times for Start and Destination
public String [] getFutureTimesForStartAndDest(String startdestStop){
```

Figure 3.0.19 getFutureTimesForStartAndDest() function via DbBackend class

The *getFutureTimesForStartAndDest()* function is quite lengthy, but its task is rather important. The *getFutureTimesForStartAndDest()* function takes care of the following tasks:

1. Retrieving the current local time
2. Accessing the shuttle times that correspond with the particular day of the week (i.e Mon.-Wed, Thur.-Fri. Sat., and Sunday)
3. Filtering through the list of shuttle times by only adding shuttle arrival times that have not passed yet
4. Converting list of shuttle times from 24hr format to 12hr format and returning the final list of shuttle arrival times to the *CampusShuttle* class to be displayed

1. Retrieving the current local time (Figure 3.0.19-1)

```
//Function to List Future Times for Start and Destination
public String [] getFutureTimesForStartAndDest(String startdestStop){
    ArrayList<String> timeArray = new ArrayList<String>();

    //Get current hour and min
    Integer currentHour = 0;
    Integer currentMinute = 0;
    String currentHr = "";
    String currentMin= "";
    String currentTime= "";
    int cHour = 0;
    int cMinute = 0;
    String selectCurrentTimeQ = "SELECT STRFTIME('%H','NOW','LOCALTIME') AS HOUR,
STRFTIME('%M','NOW','LOCALTIME') AS MINUTE ";
    Cursor cur          = this.getDbConnection().rawQuery(selectCurrentTimeQ,null);
    String[] currentTimeSplit = new String[2];

    if(cur.moveToFirst()) {

        currentHour = Integer.parseInt(cur.getString(0));
        currentMinute = Integer.parseInt(cur.getString(1));
        currentTime = currentHour + ":" + currentMinute;
        currentTimeSplit = currentTime.split(":");
        currentHr = (currentTimeSplit[0]);
        currentMin = (currentTimeSplit[1]);
        cHour = Integer.parseInt(currentHr);
        cMinute = Integer.parseInt(currentMin);

    }
```

Figure 3.0.19-1 Code of getFutureTimesForStartAndDest() function for retrieving current local time

In order to properly return the filtered list of shuttle arrival times to the user, we must use the following standard SQL query to retrieve the local current time from the database using the following line of code:

String selectCurrentTimeQ = "SELECT STRFTIME('%H','NOW','LOCALTIME') AS HOUR, STRFTIME('%M','NOW','LOCALTIME') AS MINUTE ";

Storing our SELECT statement in a string saves us the trouble of typing such a long query again. Next, we then establish a connection to the database to begin reading data, while simultaneously creating an object known as a Cursor to reverse the database on a row by row basis. This is shown with following line of code:

Cursor cur = this.getDbConnection().rawQuery(selectCurrentTimeQ,null);

In this scenario, a cursor enables the rows in a result set (a set of data rows) to be processed sequentially. In SQL procedures, a cursor makes it possible to define a result set and perform complex logic on a row by row basis. [7] In other words, since the only piece of data we want at the moment is the local time this is will be read as a single row of data from the SQLite database. Lastly, in Figure 3.0.19-1 we check to see if there is a first row of data present, which will be the local time, and if so we simply parse the current local hour and minute into integers to be used later on in the function.

```
if(cur.moveToFirst()) {

    currentHour = Integer.parseInt(cur.getString(0));
    currentMinute = Integer.parseInt(cur.getString(1));
    currentTime = currentHour + ":" + currentMinute;
    currentTimeSplit = currentTime.split(":");
    currentHr = (currentTimeSplit[0]);
    currentMin = (currentTimeSplit[1]);
    cHour = Integer.parseInt(currentHr);
    cMinute = Integer.parseInt(currentMin);

}
```

2. Accessing the shuttle times that correspond with the particular day of the week (Figure 3.0.19-2)

Once we have retrieved the current local time, the next step is to access the proper table of shuttle arrival times within the SQLite database based on the day of the week. To check the particular of the week I used the following function:

```
//Function to check current day of week. (i.e Monday, Tuesday,
// etc and returns a corresponding number)
public Integer dayOfWeek(){
    Calendar calendar = Calendar.getInstance();
    int weekDay = calendar.get(Calendar.DAY_OF_WEEK);
    return weekDay;
}
```

The *dayOfWeek()* function will return a number that corresponds with a particular day of the week.(i.e. 5 = Thursday, 6 = Friday, and etc.) As seen in Figure 3.0.19-2, this function is used in conjunction with several if statements that use a particular SELECT statement based on the day of the week to choose which table of shuttle arrival times to access within the SQLite database.

```
String selectDatabaseTimeQ = "";

if (dayOfWeek() == 5 || dayOfWeek() == 6){ //Thursday or Friday
    //if dayofweek is thursday or friday query thursday and friday night time schedule
    selectDatabaseTimeQ = "SELECT * FROM Time_Table_Thur_Fri as TT WHERE TT.stop_id
LIKE '"+startdestStop+"' ";
}else if (dayOfWeek() == 7){ //Saturday
    //if day of week is saturday query saturday time schedule
    selectDatabaseTimeQ = "SELECT * FROM Time_Table_Saturday as TT WHERE TT.stop_id
LIKE '"+startdestStop+"' ";
}else if(dayOfWeek() == 1){ //Sunday
    //if day of week is sunday query sunday time schedule
    selectDatabaseTimeQ = "SELECT * FROM Time_Table_Sunday as TT WHERE TT.stop_id LIKE
'"+startdestStop+"' ";
}else{
    //else query regular monday through Wed. schedule
    selectDatabaseTimeQ = "SELECT * FROM Time_Table_Mon_Wed as TT WHERE TT.stop_id
LIKE '"+startdestStop+"' ";
}
```

Figure 3.0.19-2 Conditional Statements used to access the correct table in the database via DbBackend class

3. Filtering through the list of shuttle times by only adding shuttle arrival times that have not passed yet (Figure 3.0.19-3)

The next step is to only return shuttle arrival times that have not passed yet to the user. This involves a filtering process shown in Figure 3.0.19-3. Similar to the previous step once the

```
if (cursor1.moveToFirst()){

    String[] databaseTimeSplit = new String[2];
    if(cursor1.moveToFirst()) {
        do {
            String time =
cursor1.getString(cursor1.getColumnIndexOrThrow("shuttle_time"));
                //add AM to morning time && add PM to afternoon time
                databaseTimeSplit = time.split(" ");
                databaseHr = (databaseTimeSplit[0]);
                databaseMin = (databaseTimeSplit[1]);
                databaseHour = Integer.parseInt(databaseHr);
                databaseMinute = Integer.parseInt(databaseMin);

                if (databaseHour >= 12) {
                    databaseTime = databaseHour + ":" + databaseMinute + " PM";
                } else{
                    databaseTime = databaseHour + ":" + databaseMinute + " AM";
                }

                //only add times that haven't passed yet
                if (databaseHour >= cHour)
                {
                    if(databaseHour > cHour) {
                        time = databaseTime;
                        databaseTimes.add(time);
                    }
                    if(databaseHour == cHour && databaseMinute > cMinute) {
                        time = databaseTime;
                        databaseTimes.add(time);
                    }

                }
                if(databaseHour == 0 || databaseHour == 1 || databaseHour == 2 ){
                  time = databaseTime;
                    databaseTimes.add(time);
                }


            }
            while (cursor1.moveToNext()) ;
        }
```

Figure 3.0.19-3 Code that only adds shuttle arrival times that have not passed yet to a particular list.

proper query SELECT statement for a particular day is chosen every shuttle arrival time associated with the starting and destination point chosen by the user is read as long as the Cursor has a row of data left to traverse. As the Cursor traverses through each shuttle arrival time, the hour and min data within the database is converted to an integer and then compared to the current hour and current min. If the shuttle arrival time within the database has not already passed then it is added to a list that will be later presented back to the user on their screen.

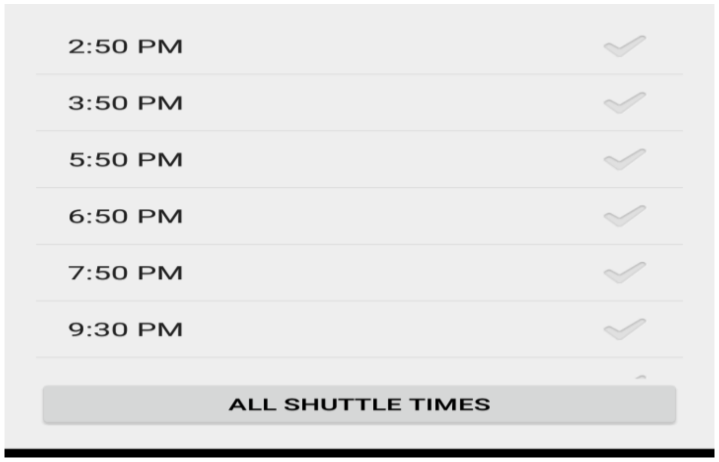
4. Converting list of shuttle times from 24hr format to 12hr format (3.0.19-4)

Just before the filtered list of shuttle arrival times is given back to the user for viewing every time is first converted into a 12hr format. The primary reason I decided to store shuttle times for the database in a 24hr format is because had they been saved in the database in a 12hr format there wouldn’t be any way for the device nor SQLite database to determine which time was in the morning or afternoon. (i.e. AM or PM) By using a 24hr time format within the database I am easily able to determine whether or not a time is for the morning or afternoon since hours from 0 to 11 are AM(Morning) times and hours from 12 to 23 are PM(Afternoon).

```
for (int i = 0; i < databaseTimes.size(); i++) {
        try {
            final SimpleDateFormat sdf = new SimpleDateFormat("HH:mm ");
            final Date dateObj = sdf.parse(databaseTimes.get(i));
            if (!timeArray.contains(new SimpleDateFormat("h:mm a").format(dateObj))){
                timeArray.add(new SimpleDateFormat("h:mm a").format(dateObj));
            }else{
                continue;
            }
        } catch (final ParseException e) {
            e.printStackTrace();
        }
    }
}

cursor1.close();
String[] allListView = new String[timeArray.size()];
allListView = timeArray.toArray(allListView);
return allListView;
```

Figure 3.0.19-4 Code used to convert shuttle arrival times from 24hr format to 12hr format

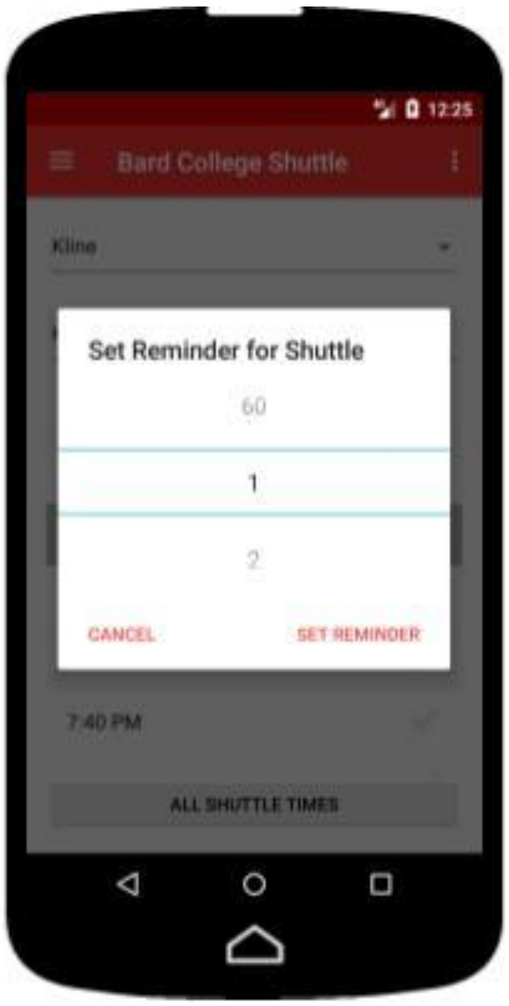
Once the time has been converted into its proper 12hr format for the user it is then stored into a list, which is named *allListView* in Figure 3.0.19-4. This final list of shuttle times is then passed back to the *CampusShuttle* class and then displayed on the screen, which gives users the following result.



2:50 PM ✓
3:50 PM ✓
5:50 PM ✓
6:50 PM ✓
7:50 PM ✓
9:30 PM ✓

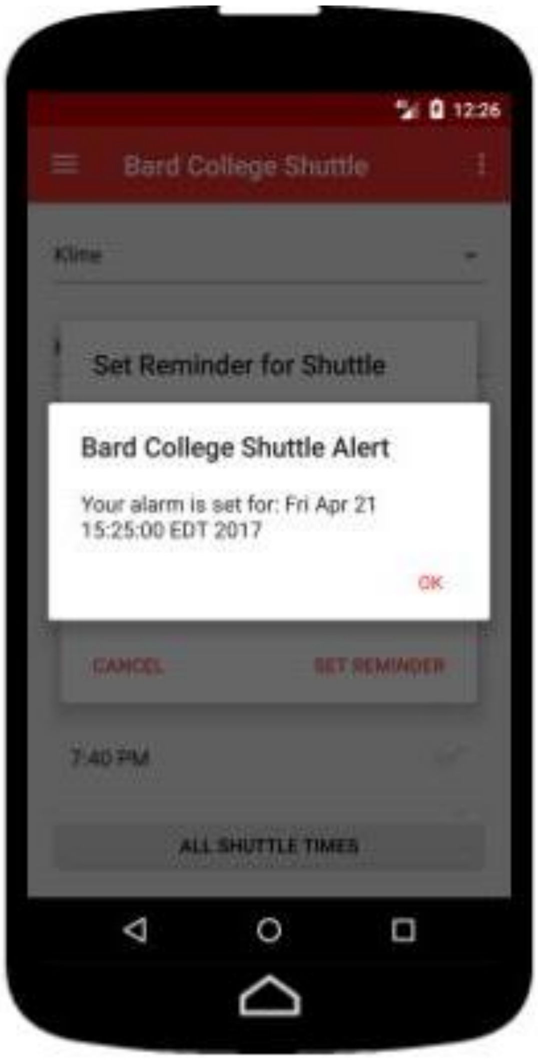
3.6 Alarm Services Implementation

One of the key functionalities that I wanted to offer for the new shuttle application was the ability to set alarms to remind users of a shuttle's arrival. For the Campus Shuttle service, users are given the option to set an alarm between 1 and 60 minutes before a shuttle's arrival. Users are presented with the option to set an alarm by simply selecting a shuttle arrival time from the list provided as demonstrated in Figure 3.0.20.



Figurer 3.0.20 Shuttle Reminder AlertDialog 1

The user would simply choose their desired reminder time, press ‘Set Reminder’, which presents an AlertDialog reminding users of the time they have chosen for their alarm to sound. Depending on the user’s device settings their devices will either vibrate or play their default notification sound, while also providing them with a notification in the notification bar as shown in Figure 3.0.21.



Your alarm is set for: Fri Apr 21 15:25:00 EDT 2017

Figure 3.0.21 Shuttle Reminder AlertDialog 2

Although there are two functions within the *CampusShuttle* class that are responsible for AlertDialogs pertaining to the alarms, those will be available in the Appendix A. The focus of this section will be the following functions:

1. oneTimeAlarm()
2. setOneTimeAlarm()
3. notificationAlert()

These are the paramount functions that are most important when attempting to understand at least on a surface level how the application works.

1. oneTimeAlarm ()

The *oneTimeAlarm()* function located in the *CampusShuttle* class is quite simple. (Figure 3.0.22) Once a user has chosen their shuttle reminder time, the *oneTimeAlarm()* function retrieves the current date from the SQLite database via a function within the *DbBackend* class. Next, the current time of the day, minutes chosen by the user and the current date are used as parameters for the setOneTimeAlarm() function located within the *AlarmManagerBroadcastReceiver* class to set the alarm.

```
/**
 * This function is used to set one time Alarm based on the time and date
 * selected by the user
 */
public void onetimeTimer() {
    final DbBackend dbBackend = new DbBackend(getActivity());
    selectedDate = dbBackend.getSQLDate();

    alarm = new AlarmManagerBroadcastReceiver();

    Context context = getActivity().getApplicationContext();
    if (!alarm.equals(null)) {

        busAlarmTime = alarm.setOneTimeAlarm(context, time, setMinuteForAlarm,
selectedDate);
        alarmSetAlertDialogBox(busAlarmTime);

    } else {
        Toast.makeText(context, "Alarm needs parameter", Toast.LENGTH_LONG).show();
    }
}
```

Figure 3.0.22 oneTimeAlarm() function via Campus Shuttle class

2. setOneTimeAlarm ()

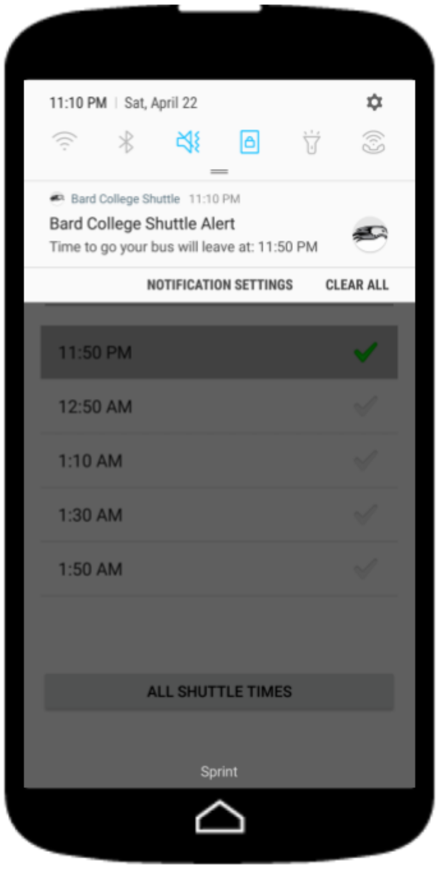
The code shown in Figure 3.0.23 looks as simple just as much as looks daunting. Using the current time, date and number of minutes chosen by the user, the *setOneTimeAlarm()* function is called when the 'Set Reminder' Button is pressed. It sets the Alarm for a specific time which is equal to Bus time minus the minutes chosen by the user before which Alarm should ring or vibrate the device. The function is also responsible for creating the needed PendingIntent for the alarm notification.

```
/**
 * This is the function which is called on click of 'Set Alarm' Button. It
 * sets the Alarm for time which is equal to Bus time minus the minutes
 * before which Alarm should ring.
 *
 * @param context
 *            Application Context
 * @param time
 *            Bus Time selected by the user
 * @param beforeMinutes
 *            Minutes before the Bus timing when the Alarm should start
 *            Ringing
 * @param selectedDate
 *            Date selected for setting Alarm
 */
public String setOneTimeAlarm(Context context, String time, int beforeMinutes, String
selectedDate) {

    Log.i("AlarmManagerBroadcastReceiver.setOneTimeAlarm()::", "Setting the Alarm");

    SimpleDateFormat simpleDateFormat = new SimpleDateFormat("hh:mm a");
    Date date = null;

    try {
        date = simpleDateFormat.parse(time);
    } catch (ParseException e) {
        Log.e("AlarmManagerBroadcastReceiver.setOneTimeAlarm()::",
                "Error occurred while Parsing the Bus time, " + e.getMessage());
    }

    alarmManager = (AlarmManager) context.getSystemService(Context.ALARM_SERVICE);
    Intent intent = new Intent(context, AlarmManagerBroadcastReceiver.class);
    intent.putExtra(ONE_TIME, Boolean.TRUE);
    intent.putExtra(BUS_TIME, time);

    Calendar calender = Calendar.getInstance();
    calender.setTime(new Date(selectedDate));
    PendingIntent pendingIntent = PendingIntent.getBroadcast(context, 0, intent, 0);

    calender.set(Calendar.HOUR, date.getHours());
    calender.set(Calendar.MINUTE, date.getMinutes() - beforeMinutes);

    Log.i("AlarmManagersetOneTimeAlarm()::", "Alarm set for : " + calender.getTime());

    alarmManager.set(AlarmManager.RTC_WAKEUP, (calender.getTimeInMillis()),
pendingIntent);

    return calender.getTime().toString();
```

Figure 3.0.23 setOneTimeAlarm() Function via AlarmManagerBroadcastReceiver class

3. notificationAlert ()

In order for an alarm to be set what is called PendingIntent must be created and passed to the *AlarmManagerBroadcastReceiver* class. A PendingIntent which is given to another application grants a particular application "permission" to perform the operation you have specified [8]. The operation we have specified for the shuttle application to perform is to sound an alarm at a specified time, turn on a LED light notification (if the device has the capability) and present the user with a notification within their devices taskbar. (Figure 3.0.24) The code is presented in Figure 3.0.25 to ensure a surface level understanding at least for non-programmers.



Bard College Shuttle Alert
Time to go your bus will leave at: 11:50 PM

Figure 3.0.24 Shuttle Arrival Notification via Android Device Notification Drawer

```
/**
 * This function displays a notification in the task bar and also plays an alert
 * sound and vibration for alarm.
 *
 * @param context
 *             Context of the application
 * @param intent
 *             Intent of the application
 */
public void notificationAlert(Context context, Intent intent) {

    Bundle extras = intent.getExtras();
    if(intent != null)
    {
        NotificationManager mNotifyMgr =
                (NotificationManager)
context.getSystemService(Context.NOTIFICATION_SERVICE);

        NotificationCompat.Builder mBuilder = (NotificationCompat.Builder) new
NotificationCompat.Builder(context)
                .setSmallIcon(R.drawable.ic_launcher)
                //example for large icon
                .setLargeIcon(BitmapFactory.decodeResource(context.getResources(),
R.drawable.ic_launcher))
                .setContentTitle("Bard College Shuttle Alert")
                .setContentText("Time to go your bus will leave at: " +
extras.getString(BUS_TIME))
                .setOngoing(false)
                .setPriority(NotificationCompat.PRIORITY_DEFAULT)
                .setAutoCancel(true);
        Intent i = new Intent(context, AlarmManagerBroadcastReceiver.class);
        pendingIntent = PendingIntent.getActivity(context, 0, i,
PendingIntent.FLAG_ONE_SHOT);

        // example for blinking LED
        mBuilder.setLights(Color.RED, 200, 500);
        mBuilder.setSound(Settings.System.DEFAULT_NOTIFICATION_URI); //default
notification sound
        mBuilder.setContentIntent(pendingIntent);
        mNotifyMgr.notify(12345, mBuilder.build());
        Vibrator vibrator = (Vibrator)
context.getSystemService(Context.VIBRATOR_SERVICE);
        // Start without a delay
        // Vibrate for 200 milliseconds
        // Sleep for 200 milliseconds
        //repeat 3 times
        long[] pattern = {0, 200, 200, 200, 200, 200, 200};
        // The '0' here means to repeat indefinitely
        // '0' is actually the index at which the pattern keeps repeating from (the
start)
        // To repeat the pattern from any other point, you could increase the index,
e.g. '1'
        vibrator.vibrate(pattern,-1);

    }
}
```

Figure 3.0.25 notificationAlert() function via AlarmManagerBroadcastReceiver class

4
Conclusion

4.1 Beta Tester Feedback & Limitations

Although the Bard Shuttle Application may have come a long way in terms of improvements I felt that it was important to ask for feedback from at least a small number of students. This way I could get an idea of what aspects of the application were limited and what aspects could undergo further changes and improvements. So, I asked my beta-testers to complete a brief survey of their experience using the application. The results were the following:

Figure 4.1: Question 1, How satisfied were you with using the New Bard Shuttle App? ***(7 Responses)***

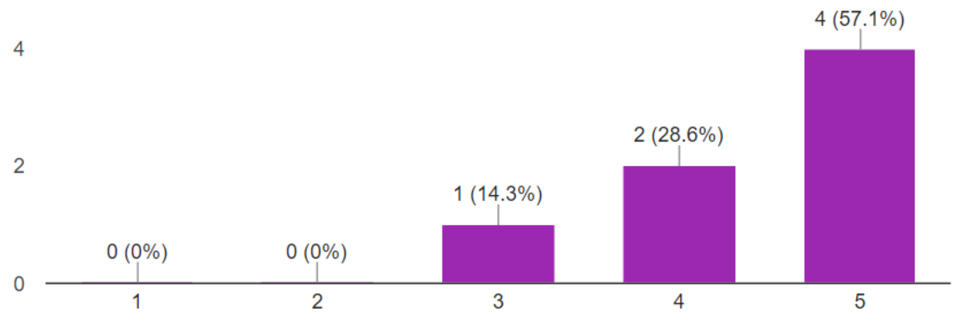


Figure 4.2: Question 2, How satisfied were you with using the New Bard Shuttle App? ***(7 Responses)***

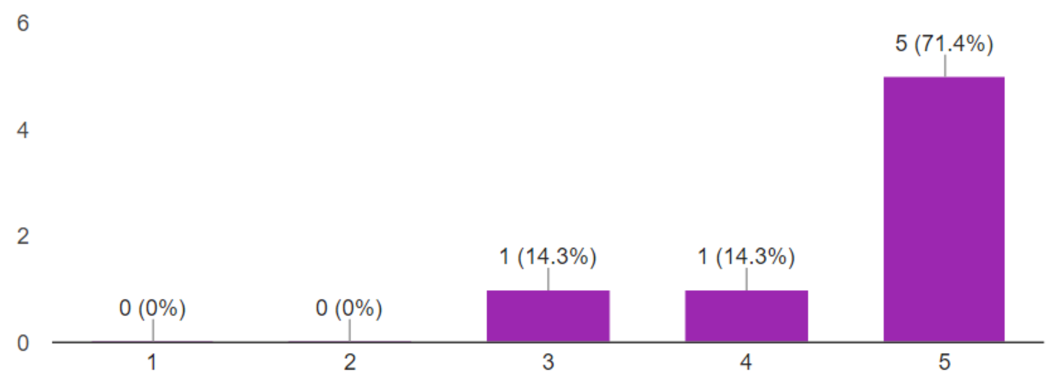


Figure 4.3: Question 3, How would you rate the usefulness of the shuttle reminder functionality? ***(7 Responses)***

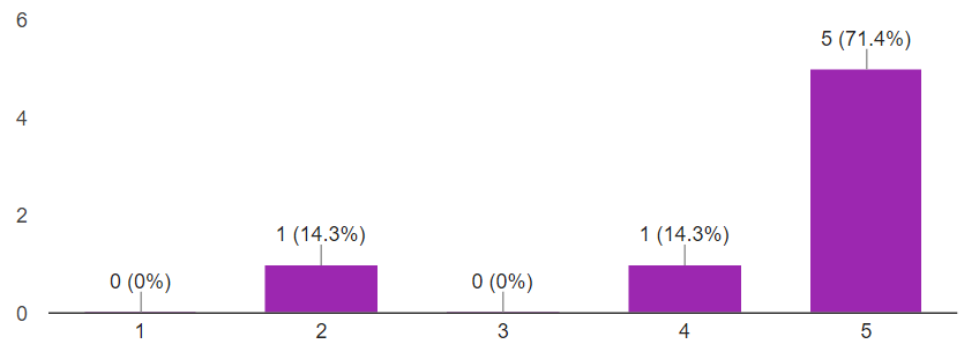


Figure 4.4 Question 4, How satisfied were you with the overall experience using the application? ***(7 Responses)***

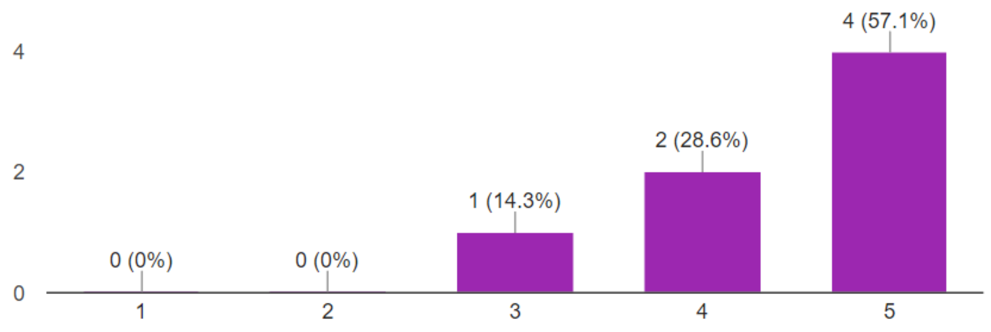
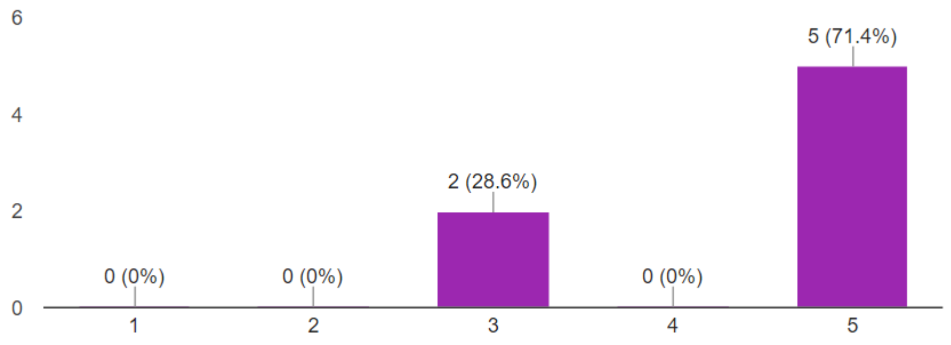


Figure 4.5: Question 5, If you have used the previous Bard Shuttle Application offered on Android Devices how would you rate the improvements the new application offers compared to the previous one? ***(7 Responses)***



Overall the feedback on the application was positive and in term of comments and possible improvements the following were noted:

1. **Addition of Loop Bus Schedule**
2. **Rotating the device after selecting a start and destination point restarts the application**
3. **Once an Alarm has been set, there is no way to unset the alarm**

4. **Only one alarm can be set at a time and you can't view the alarm that you set for later**
5. **Show estimated arrival time for destination**

Although there were only seven beta-testers, I received invaluable feedback. All were satisfied with the UI and out of the five concerns noted by the beta-testers, some have the potential to be fixed rather quickly depending on the implementation. During the remainder of my enrollment at Bard College, I plan to address as many of these problems as possible before my graduation. If not, there will be room for other programs to make additions to the application upon contacting me.

4.2 Future Work

My ultimate goal was to have an operational user case from section 3.1 Figure 3.0.1 that depicted the Bard Shuttle Application accessing an online SQL database in order to receive necessary shuttle times and information. Unfortunately, this would have required either an individual staff member or Bard College department such as the I.T or Transportation department to update the database every year. This final implementation would have been ideal since those responsible for updating the database would never need to access the shuttle application's programming code. The only reason the code would need to be accessed was in the case of a bug or if the addition or change of a feature was necessary. Until a connection to an online SQL database is possible an SQLite database will be used, but this will in turn only allow users to have access to the Campus Shuttle and Loop Bus services since their schedules are highly unlikely to be changed. In the event that the shuttle schedules are changed postgraduation, I will only need to update the database from my personal computer and push an application update to the Google Play Store for users.

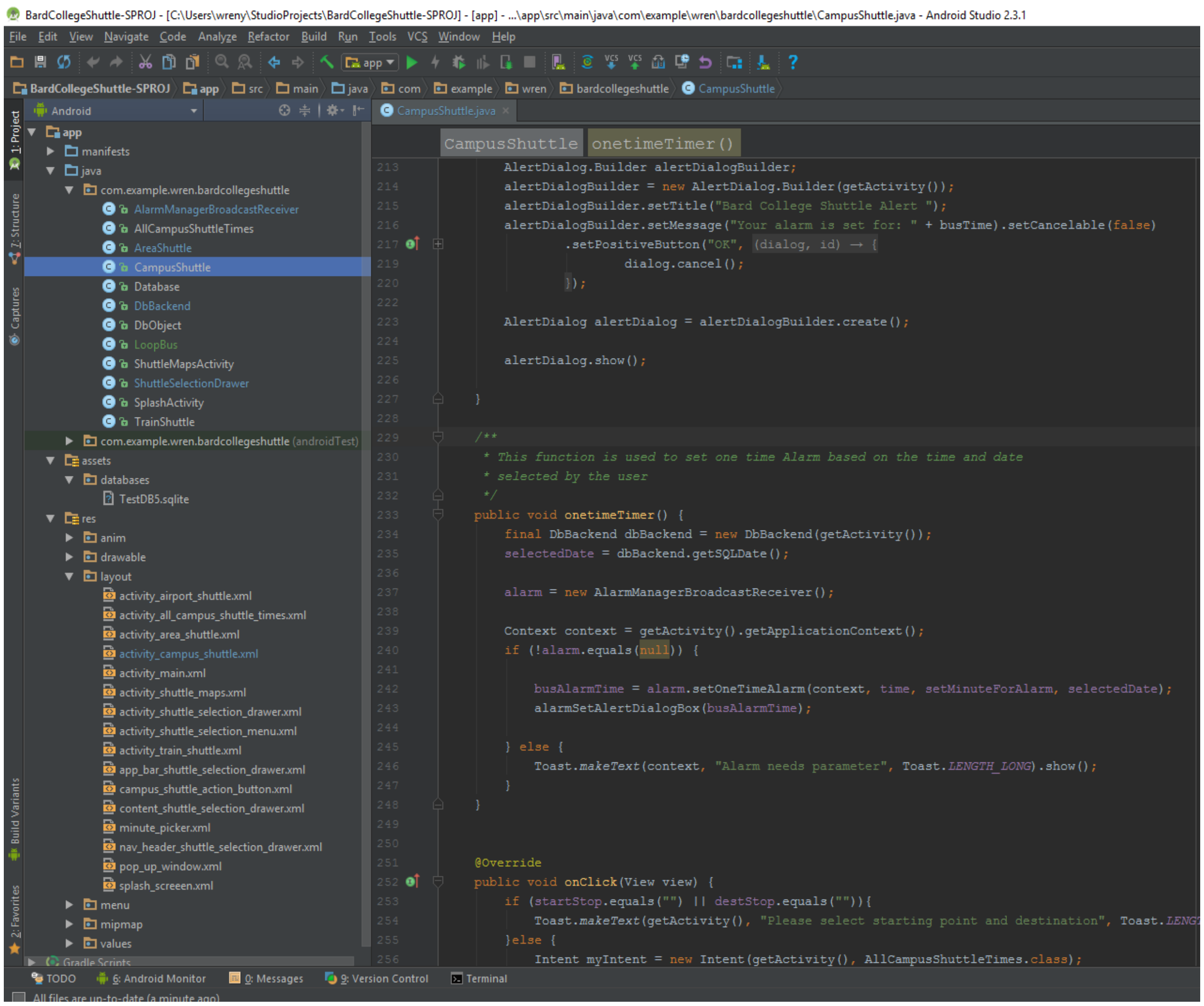
Working on this project has been not only been my most exciting projects in my college career, but it has also been a learning experience of a lifetime. Completion of this project has sparked my interest in Mobile Application Development and now I plan on pursuing a career in the field as well as in Technology Consulting, which deals heavily with technology implementation similar to my senior project, but on a much larger scale.

Appendix

Appendix A: Functions for AlarmDialogs for CampusShuttle class

```
/**
 * This function displays an Alert Dialog Box with Number Picker, So
 * users can choose the number of minutes for which they want to set their alarm
 * The minutes users can choose from are from 1-60
 * @param busTime
 *            This parameter holds the value of time for which alarm is set
 */
public void setAlarmDialogBox(final String busTime){
    final NumberPicker numberPicker = new NumberPicker(getActivity());
    numberPicker.setMaxValue(60);
    numberPicker.setMinValue(1);
    AlertDialog.Builder builder = new AlertDialog.Builder(getActivity());
    builder.setView(numberPicker);
    builder.setTitle("Set Reminder for Shuttle");
    builder.setPositiveButton("SET REMINDER", new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {
            setMinuteForAlarm = numberPicker.getValue();
            time = busTime;
            Log.i("setAlarmDialogBox::", "Time:" + time);
            Log.i("setAlarmDialogBox::", "BusTime:" + busTime);
            onetimeTimer();
        }
    });
    builder.setNeutralButton("CANCEL", new DialogInterface.OnClickListener(){
        @Override
        public void onClick(DialogInterface dialog, int which) {
            dialog.cancel();
        }
    });
    builder.create();
    builder.show();
}
/**
 * This function displays an Alert Dialog Box with 'Ok' button, informing
 * user that alarm has been set for the selected date and time
 * @param busTime
 *            This parameter holds the value of time for which alarm is set
 */
private void alarmSetAlertDialogBox(String busTime) {
    AlertDialog.Builder alertDialogBuilder;
    alertDialogBuilder = new AlertDialog.Builder(getActivity());
    alertDialogBuilder.setTitle("Bard College Shuttle Alert ");
    alertDialogBuilder.setMessage("Your alarm is set for: " +
busTime).setCancelable(false)
            .setPositiveButton("OK", new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int id) {
                    dialog.cancel();
                }
            });

    AlertDialog alertDialog = alertDialogBuilder.create();
    alertDialog.show();
```

Appendix B: Android Studio Interface Image

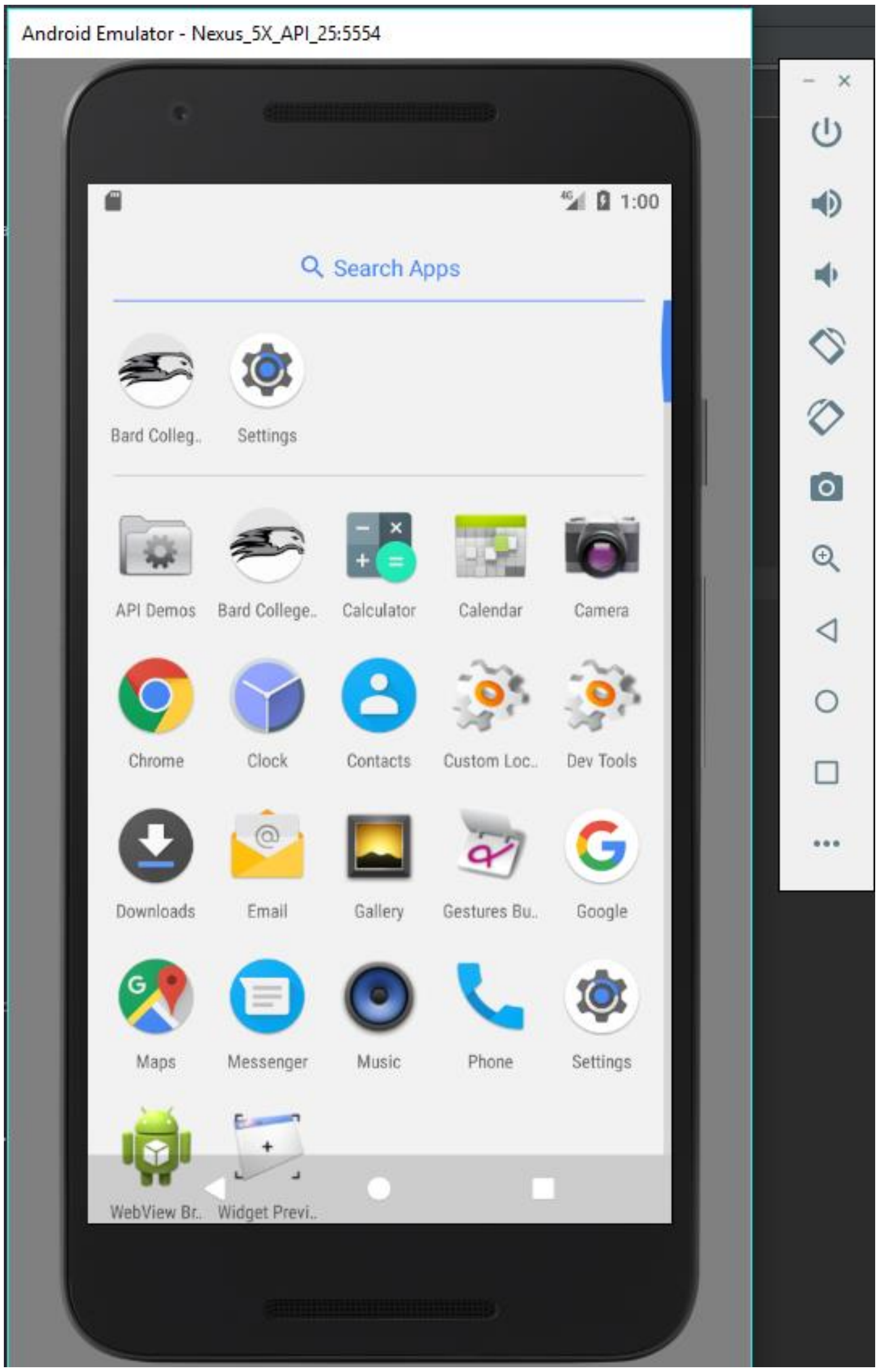

File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
BardCollegeShuttle-SPROJ app src main java com example wren bardcollegeshuttle CampusShuttle
Android
CampusShuttle.java
app
manifests
java
com.example.wren.bardcollegeshuttle
AlarmManagerBroadcastReceiver
AllCampusShuttleTimes
AreaShuttle
CampusShuttle
Database
DbBackend
DbObject
LoopBus
ShuttleMapsActivity
ShuttleSelectionDrawer
SplashActivity
TrainShuttle
com.example.wren.bardcollegeshuttle (androidTest)
assets
databases
TestDB5.sqlite
res
anim
drawable
layout
activity_airport_shuttle.xml
activity_all_campus_shuttle_times.xml
activity_area_shuttle.xml
activity_campus_shuttle.xml
activity_main.xml
activity_shuttle_maps.xml
activity_shuttle_selection_drawer.xml
activity_shuttle_selection_menu.xml
activity_train_shuttle.xml
app_bar_shuttle_selection_drawer.xml
campus_shuttle_action_button.xml
content_shuttle_selection_drawer.xml
minute_picker.xml
nav_header_shuttle_selection_drawer.xml
pop_up_window.xml
splash_screeen.xml
menu
mipmap
values
Gradle Scripts
1: Project
7: Structure
Captures
Build Variants
2: Favorites
TODO
6: Android Monitor
0: Messages
9: Version Control
Terminal
CampusShuttle
onetimeTimer()

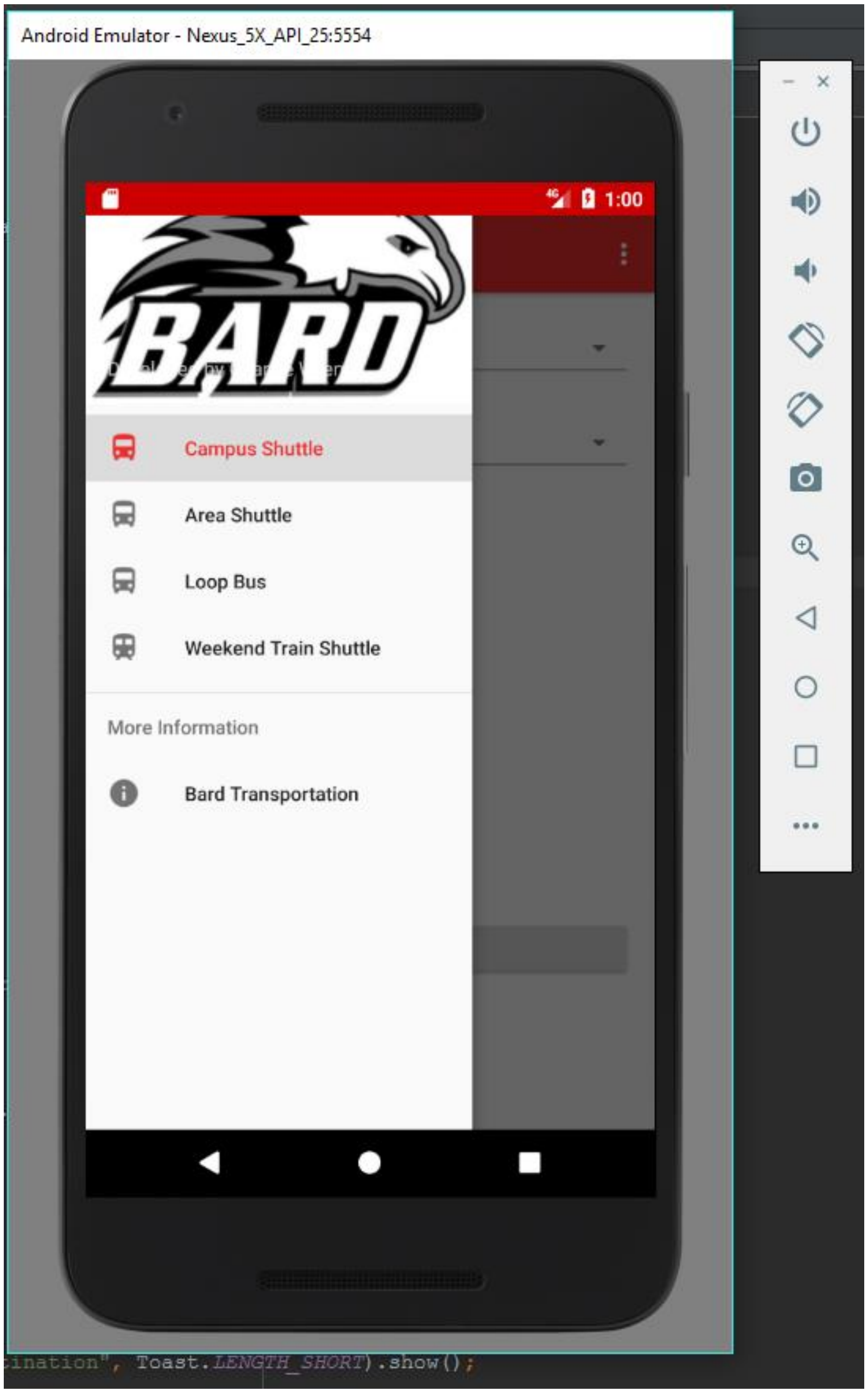
BardCollegeShuttle-SPROJ - [C:\Users\wreny\StudioProjects\BardCollegeShuttle-SPROJ] - [app] - ...\app\src\main\java\com\example\wren\bardcollegeshuttle\CampusShuttle.java - Android Studio 2.3.1

```java
        AlertDialog.Builder alertDialogBuilder;
        alertDialogBuilder = new AlertDialog.Builder(getActivity());
        alertDialogBuilder.setTitle("Bard College Shuttle Alert ");
        alertDialogBuilder.setMessage("Your alarm is set for: " + busTime).setCancelable(false)
                .setPositiveButton("OK", (dialog, id) → {
                        dialog.cancel();
                });

        AlertDialog alertDialog = alertDialogBuilder.create();

        alertDialog.show();

    }

    /**
     * This function is used to set one time Alarm based on the time and date
     * selected by the user
     */
    public void onetimeTimer() {
        final DbBackend dbBackend = new DbBackend(getActivity());
        selectedDate = dbBackend.getSQLDate();

        alarm = new AlarmManagerBroadcastReceiver();

        Context context = getActivity().getApplicationContext();
        if (!alarm.equals(null)) {

            busAlarmTime = alarm.setOneTimeAlarm(context, time, setMinuteForAlarm, selectedDate);
            alarmSetAlertDialogBox(busAlarmTime);

        } else {
            Toast.makeText(context, "Alarm needs parameter", Toast.LENGTH_LONG).show();
        }
    }


    @Override
    public void onClick(View view) {
        if (startStop.equals("") || destStop.equals("")){
            Toast.makeText(getActivity(), "Please select starting point and destination", Toast.LENG
        }else {
            Intent myIntent = new Intent(getActivity(), AllCampusShuttleTimes.class);
```

All files are up-to-date (a minute ago)

Appendix C: Android Studio Emulator Images

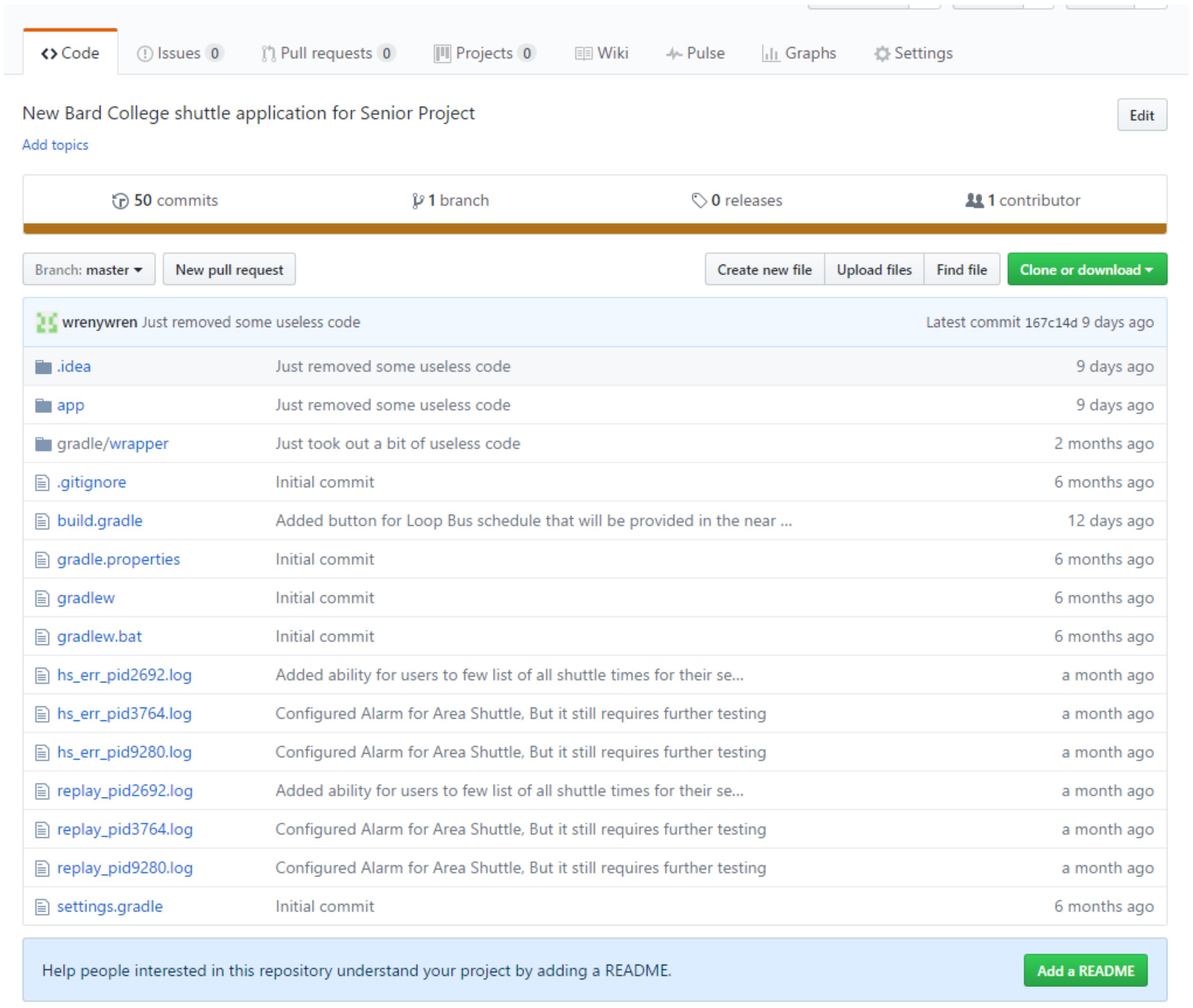




Android Emulator - Nexus_5X_API_25:5554

```
ination", Toast.LENGTH_SHORT).show();
```

Appendix D: Github Repository Image



New Bard College shuttle application for Senior Project

50 commits | 1 branch | 0 releases | 1 contributor

wrenywren Just removed some useless code — Latest commit 167c14d 9 days ago

.idea	Just removed some useless code	9 days ago
app	Just removed some useless code	9 days ago
gradle/wrapper	Just took out a bit of useless code	2 months ago
.gitignore	Initial commit	6 months ago
build.gradle	Added button for Loop Bus schedule that will be provided in the near ...	12 days ago
gradle.properties	Initial commit	6 months ago
gradlew	Initial commit	6 months ago
gradlew.bat	Initial commit	6 months ago
hs_err_pid2692.log	Added ability for users to few list of all shuttle times for their se...	a month ago
hs_err_pid3764.log	Configured Alarm for Area Shuttle, But it still requires further testing	a month ago
hs_err_pid9280.log	Configured Alarm for Area Shuttle, But it still requires further testing	a month ago
replay_pid2692.log	Added ability for users to few list of all shuttle times for their se...	a month ago
replay_pid3764.log	Configured Alarm for Area Shuttle, But it still requires further testing	a month ago
replay_pid9280.log	Configured Alarm for Area Shuttle, But it still requires further testing	a month ago
settings.gradle	Initial commit	6 months ago

Help people interested in this repository understand your project by adding a README.

Bibliography

[1] "About SQLite". Sqlite.org. N.p., 2017. Web. 6 Apr. 2017.

[2] "What Is SQL". *SQL Course*. Web. 6 Apr. 2017.

[3] Stephens, Ryan K, Arie Jones, and Ronald R Plew. Sams Teach Yourself SQL In 24 Hours. 3rd ed. Print.

[4] “1.1 Getting Started - About Version Control." Git. GITHUB, n.d. Web. 23 Apr. 2017.

[5] “Fragments." Android Developers. N.p., n.d. Web. 23 Apr. 2017.

[6] "AlertDialog." Android Developers. N.p., n.d. Web. 23 Apr. 2017.

[7] "Part 1: How cursors work." SearchSQLServer. N.p., n.d. Web. 23 Apr. 2017.

[8] "PendingIntent." Android Developers. N.p., n.d. Web. 23 Apr. 2017.