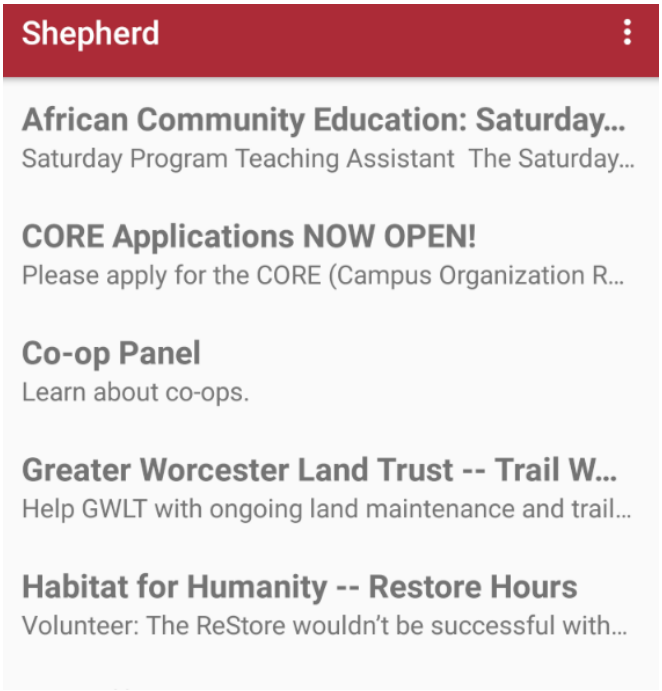
Easy Check-in App

a mobile app using location-awareness

(CS4518 Final Project Report)



Team members:

Steven Huynh
Yifei Jin
Zixin Luo
Matthew Beader

Date:
Dec 14th, 2017

1.0 Abstract

The Easy Check-in app is focused on finding a better solution for student organizations of Worcester Polytechnic Institute (WPI) to keep track of attendance for their events. While the current methods of tracking attendance are not accurate and very time consuming, event attendance is a significant statistic for the event organizers for planning future events and receiving an annual budget from the Student Government Association (SGA). In order to improve current methods for attendance tracking, we decided to simplify the attendance check-in process and apply it to a WPI student's mobile phone. Following this approach, we built a mobile app that allows students to check in to an event on their phone, taking advantage of mobile devices' capacity for location-awareness. With this app, we believe attendance taking will be less intrusive and more students will be accounted for, so the accuracy of event attendance will increase.

2.0 Introduction

One of the tasks student organization leaders of Worcester Polytechnic Institute must perform is attendance tracking for their events. Having statistics on event attendance helps these leaders learn what events and activities are popular among students and partially determines the organization's budget for next year by the SGA.

From observing how student organizations keep track of events, we have concluded that their current methods, such as sign-in sheets and card swiping, are intrusive and inaccurate. Having attendees sign their names on paper requires every attendee to perform a relatively time-consuming interaction. Having attendees swipe identification cards on card readers is prone to inaccuracy if card reader operators are not properly trained or if attendees do not bring their ID cards.

To provide a better solution for tracking event attendance, we propose an Android mobile application to be used by event attendees that will allow WPI student organizations to perform attendance tracking with accuracy and minimal intrusion for the attendees. The mobile app will provide students with information about on-campus events, navigation to a certain event, and an automatic check-in notification when the student arrives at the event location. We believe that data accuracy and minimal user intrusion for attendance tracking can be realized by taking advantage of modern mobile devices' capacity for location-context awareness. Since students today carry their mobile phones almost everywhere, our check-in app becomes a better identity than students' ID cards. Moreover, the process for check-in is also simplified: only one click on their mobile devices will finish the check-in. In addition, the app will also be a valuable implementation of mobile computing: it provides navigation service for individual user according to user's preference on events he/she wants to go; it keeps tracking user's location and interacts with the user when at specific event location. With the above features, we think it encourages more students to check-in and saves organizers' human resources for attendance tracking: students will be notified when they should check-in and the process is much simpler than before.

3.0 Design

Features

To achieve our goals for the app, we chose to have the following features: a login page as the launch portion of the app since we want the app to be a sufficient identity for the student (a replacement of student's ID card); after login, the student will be able to see list of events that are happening on the WPI campus; a student can record their interest in an event; the app provides access to navigation to the event location; eventually, when the student arrives at the event location, the app will automatically send a notification to ask the student to check-in; if the student select "yes", the attendance will be stored and sent to the organizer at the end of the event. For more details on each feature, we will discuss them in the following sections.

3.1 UI

Our app contains five distinct UI screens and, as it is intended for WPI use, features a color scheme of red, gray, and white. The first screen a user sees is a simple splash screen containing the name of the application on a plain background. Next, a login screen is shown, containing fields for a user's email address, password and two buttons, one to login and one to switch to a registration screen. Rather than displaying the buttons in exactly the same way, the switch screen button is made to look more like the link commonly shown below login forms on websites such as GitHub. The registration screen contains all the same elements as the login screen, so to distinguish it from the login screen, as different background color is used. If the user was previously logged in, the user does not need to log in through the login screen again. Invalid actions such as logging in with an email for which an account does not exist, logging in with an invalid password, and registering with an email that already exists are not carried out and instead indicated to the user.

After a user is logged in, the user has access to the full application, which is centered around a screen with a list of events the user can scroll though. This screen also features an options menu with an option to sign out. Each element of the event list is clickable, opening an event details screen when clicked. The event details screen displays a checkbox allowing the user to express their interest in an event as well as the following information about an event: name, time, location, and description. It also can launch Google Maps turn by turn directions to the event's location. When the user arrives

at an event's location, a notification is displayed. Clicking this notification opens the app to the event details screen. This screen then shows a menu allowing the user to check in to the event. Interactions between the different main UI elements is shown in Figure 1. See section 5.3 for UI layouts.

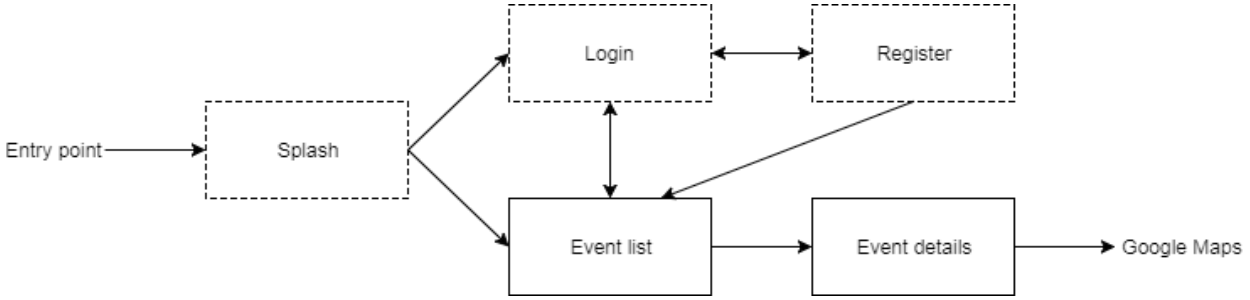


Figure 1. Arrows represent the flow between the different UI screens. Dashed rectangles indicate a screen would not be kept on the Activity stack.

3.2 Database

For the data storage of the app, we chose to use ElephantSQL- an open-source database hosted in the cloud which has PostgreSQL as a service. We chose this database service for the following reasons. First, saving data in a SQL database is the best way to explain the relationships among our data, other databases, like Firebase (which uses JSON to store data), cannot illustrate the relationships we want to show among the three different tables like SQL does. Second, hosting the database in the cloud allows multiple users to be interacting with our database at the same time. In this case, although using Google Cloud Platform and SQLite together will also give us a SQL database hosted in the cloud, but the cost for using cloud data and the complexity of setting up the connection server. This leads to the third reason why we prefer to use PostgreSQL provided by ElephantSQL. ElephantSQL provides a free 1000-row SQL database that the team can use for this project.

The structure of the three database tables we are using is as the following:

- The users' accounts information. This is used at the sign in page for comparing email/password pairs and storing new accounts at registration. The accounts table has two fields, "userid" which is "text" and "password" which is also saved as "text". The "userid" is the primary key so that one WPI email address can only create one account.
- The information about the campus events. This data is available at CampusLabs platform. In 2017, WPI adopted CampusLabs as their

community management system, which provides them with analytics on students (e.g. club involvement), as well as the ability to promote upcoming events. However, specific approval is needed to be able to use the CampusLabs API to get the resources about campus events, therefore, we built a table that contains the similar features as the CampusLabs API for our application. This way, once the SGA like the idea of this app, they can simply switch to the CampusLabs database. This table has the following features for each event: the date, name, location, room, time, and organizer. All the data is saved as text so it is easy to process with the same helper functions. The event's name is the primary key, assumng that there are no two events have the exact same name. The location name is used for providing information for the navigation function we have in the app. It is a plain text name instead of latitude and longitude because our app is expected to be used for events happening on WPI campus. We tested that within Worcester, "Rubin Campus Center" will provide users with the correct navigation in Google Maps, therefore location names can be used both for display and for navigation. The event's organizer was saved as the organizer's email address to make it easier for the app to send organizers attendance data after the event.

- The most important data we need to save is the attendance of the events. The attendance table has two fields, the attendee user id and the event name. The attendance table represents a multiple-to-multiple relationship between the events table and the accounts table. Once a user chooses to check in at an event's location, the user and the event name he/she chose to attend will be saved in the attendance table. The data in the attendance table will be sent to the organizer at the end of the event.

In a nutshell, the core data we need to save in order to accomplish the features we mentioned at the beginning of section 3.0 are user account information, event information and the information about attendance. To make sure the relationship among data can be shown clearly and the data can be interacted by multiple users, we use a PostgreSQL database hosted in the cloud and free for a limited size.

3.3 Google Maps and GPS

One key feature of our app is to show the location of the event and when the user clicks the address, the use is able to navigate to said location. Since

we do not have a large amount of time for developing the application, we are using an intent to open Google Maps to navigate. Google Maps come preinstalled on most Android phones. It also has a complete and easy-to-use API for developers. From the API site, you can find everything you need for using Google Maps. Even though most phones would have it, of course, we still need to check whether it is installed before we actually use it. That is why we finally chose Google Maps for navigation.

Another important part of our app is allowing users to check in if and only if the user is in a certain range of a point where the event is happening. The app should be able to use the real-time location of the device and when the device approaches the location, sending a notification to inform the user to check in. The event locations are all stored in the database mentioned above. Thus, the app will read the data from the database, show the data in UI, and finally send the user a notification when the user is close enough to a point.

According to the Google developer page, the Google Play services location APIs are preferred over the Android framework location APIs (android.location) as a way of adding location awareness to an app. The location APIs available in Google Play services facilitate adding location awareness to an app with automated location tracking, geofencing, and activity recognition.

Eventually, we chose to use Google Play services location APIs, more specifically, geofencing as the key function to make the app location-aware. Based on the description, geofencing combines awareness of the user's current location with awareness of the user's proximity to locations that may be of interest. To mark a location of interest, you specify its latitude and longitude. In order to do that we used another open library, Geocoder. To adjust the proximity for the location, you add a radius. The latitude, longitude, and radius define a geofence, creating a circular area, or fence, around the location of interest. That is exactly what we need for the app. What is more important is that there are tons of great tutorials on how to use geofences. Consequently, geofence is used in our app.

4.0 Implementation

4.1 UI

Each of the five main UI screens is implemented as their own activity. SplashActivity is designated as the launcher activity, which quickly displays a drawable of the app name on a red background while loading the rest of the

app. To determine which activity to open next, SharedPreferences is checked for the key "email." If it is found, the user is already logged in and EventListActivity is opened, otherwise LoginActivity is opened. After firing the intent, SplashActivity finishes.

LoginActivity's layout is implemented as two EditTexts and two Buttons on a red background. Upon clicking the login button, the database is queried for the user input email and password. If this is successful, the email is placed into SharedPreferences and EventListActivity is opened. RegisterActivity is nearly identical to LoginActivity, however it inserts the given email and password into the database and the background of the layout is gray. Attempts at invalid actions such as logging in with an invalid password or an account that does not exist do not execute and displays a toast indicating which error occurred. Both activities finish after opening the event list.

EventListActivity features a RecyclerView where the Adapter queries the database for events and the ViewHolder for each event contains its name and description. When an element of the RecyclerView is clicked, an EventDetailActvity containing the information for the clicked event is opened. EventListActivity also has an options menu containing a single item. This item allows a user to log out by removing the "email" key from SharedPreferences, starting a LoginActivity and finishing the EventListActivity.

The layout for EventDetailActivity contains TextViews for the event name, time, location and description, as well as a CheckBox for whether the user is interested in the event. The TextView for the location contains an onclick listener to fire an intent containing a "google.navigation" URI. This launches Google Maps with directions to the event's location.

Upon selecting the CheckBox labeled "Interested?", the action toggles the creation of a Geofence as a pending Intent which fires a notification when a GEOFENCE_TRANSITION_ENTER event occurs (see Section 4.3 for more information about Geofence creation). This notification stores an ID for an Event, which is used to create an EventDetailActivity with information about the event the user is presently attending. Creating this activity by clicking on the notification also causes a Dialog wrapped in a Fragment, CheckInDialogFragment, to display; this is possible by sending an Intent extra which is used by EventDetailActivity.onCreate() to determine whether to display that fragment. The dialog's "positive" button, labeled "Check in", creates and executes an addAttendance object, through EventDetailActivity.onDialogPositiveClick().

4.2 Database

In order to implement the database we designed, we created three tables. The first table, “accounts”, stores the “userid” and “password” (a sample “accounts” database is shown in Table 1). In the table, the type for both fields are “text” and the userid is the primary key.

Table 1. The accounts table.

dates	name	location	room	time	organizer
2017-12-9	The 15th Annual SHOW IN 24 HOURS	Sanford Riley Hall	Little Theatre	7,10	AYO@wpi.edu
2017-12-12	Great Problems Seminar	Rubin Campus Center	Odeum	1,3	ralamarre@wpi.edu
2017-12-12	IMGD Invited Talk	Fuller Laboratories	320	1,2	imgd@wpi.edu

The second table is called “events”. It stores information about events that are happening on the WPI campus. The “events” table has six fields: date, name, location, room, time, and organizer (a sample “events” database is shown in Table 2). In the table, the type for all six fields are “text” and the event’s name is the primary key. There are all in “text” also because we are displaying the event information to the users and “text” is simpler to display.

Table 2. The events table.

dates	name	location	room	time	organizer
2017-12-9	The 15th Annual SHOW IN 24 HOURS	Sanford Riley Hall	Little Theatre	7,10	AYO@wpi.edu
2017-12-12	Great Problems Seminar	Rubin Campus Center	Odeum	1,3	ralamarre@wpi.edu
2017-12-12	IMGD Invited Talk	Fuller Laboratories	320	1,2	imgd@wpi.edu

The third table is called “attendance”. It stores information about a event’s attendance. This table defines the relationship between the “accounts” table and the “events” table - multiple-to-multiple. The “attendance” table has two fields: user id and event name (a sample “attendance” database is shown in Table: “attendance”). In the table, the type of both fields is “text” and there is no primary key.

Table 3. The attendance table.

attendance	event
zluo2@wpi.edu	event1

The relationship between the "accounts" table and the "events" table is multiple-to-multiple and are connected by the "attendance" table. The detailed relationship can be seen from the following database schema in Figure 2.

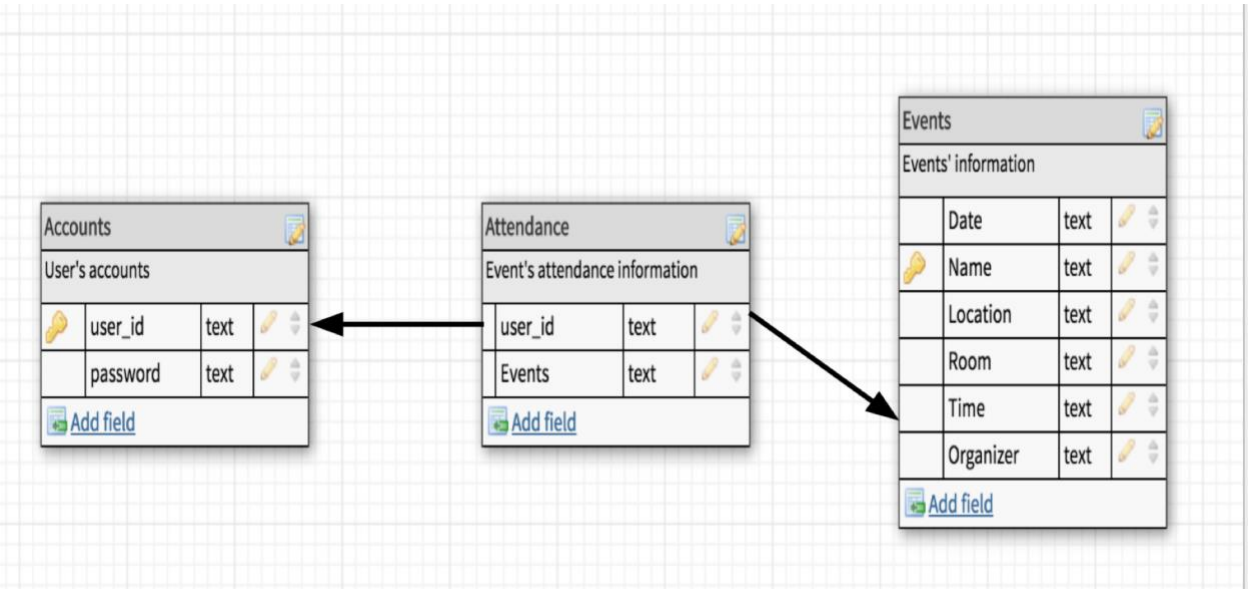


Figure 2. The schema of the database.

The most important part for the database management of this app is how to connect to a PostgreSQL database hosted in the cloud. To do this one must first download the right version of the Java Database Connectivity (JDBC) API. The JDBC is "the industry standard for database-independent connectivity between the Java programming language and a wide range of databases including SQL databases." We are using Java 8 and a PostgreSQL database, so we go to the PostgreSQL website and download PostgreSQL JDBC 4.2 Driver, 42.1.4, a binary jar file, from https://jdbc.postgresql.org/download.html.

After downloading the jar file, you will need to add it to the Android project. To accomplish this, we did the following. First, right click the project app folder, select "New" -> "Module," next select "import jar/ARR package," then import select the jar file to import, finally click "Ok" and

“Done”. Then the folder with the jar file should be visible on the right column of the project, as shown in Figure 3.

Figure 3. PostgreSQL folder inside the Android Studio Project.

After including the jar file required for JDBC, the following is the code needed for connecting to a PostgreSQL database. First of all, a new class that extends AsyncTask is created. One important thing to remember is that network connections cannot be done in the main thread. In order to connect to the cloud database in this case, all the connections should happen in AsyncTask’s doInBackground() method. In addition, all the connection calls need to be wrapped within try-catch statements in order to catch the SQLException. An AsyncTask subclass declaration is shown in Figure 4.

```
public class getEventList extends AsyncTask<Void, Void, List<String>> {
```

Figure 4. Declaration of a class extending AsyncTask.

In this sample class, attention to the three generics of AsyncTask is needed. The first is the type of input, the second is the type of progress units published during the background computation and the last is the type of the output to get from the connection. After understanding the class, Figure 5 is an example of the helper class getEventList, which loads the event list from the database for display to the user.

```java
@Override
public List<String> doInBackground(Void... arg0) {
    System.out.println("-------- PostgreSQL "
            + "JDBC Connection Testing ------------");

    try {

        Class.forName("org.postgresql.Driver");

    } catch (ClassNotFoundException e) {

        System.out.println("Where is your PostgreSQL JDBC Driver? "
                + "Include in your library path!");
        e.printStackTrace();
        return null;

    }

    System.out.println("PostgreSQL JDBC Driver Registered!");

    Connection connection = null;

    try {

        connection = DriverManager.getConnection(
                url: "jdbc:postgresql://baasu.db.elephantsql.com:5432/grqcridh", user: "grqcridh",
                password: "rBnvjIGDSdpTyhRPh7CO_xD8rGG9olya");

    } catch (SQLException e) {

        System.out.println("Connection Failed! Check output console");
        e.printStackTrace();
        return null;

    }
```

Figure 5. Example for connecting to the ElephantSQL PostgreSQL database.

From Figure 5 one can see that, after check for the "org.PostgreSQL.Driver", some information about PostgreSQL is needed in order to set up the connection. This information includes "jdbc:postgresql://server: port/datebase_name", "username", and "password", all of which can be found from the ElephantSQL web browser when the database is created as shown in Figure 6.

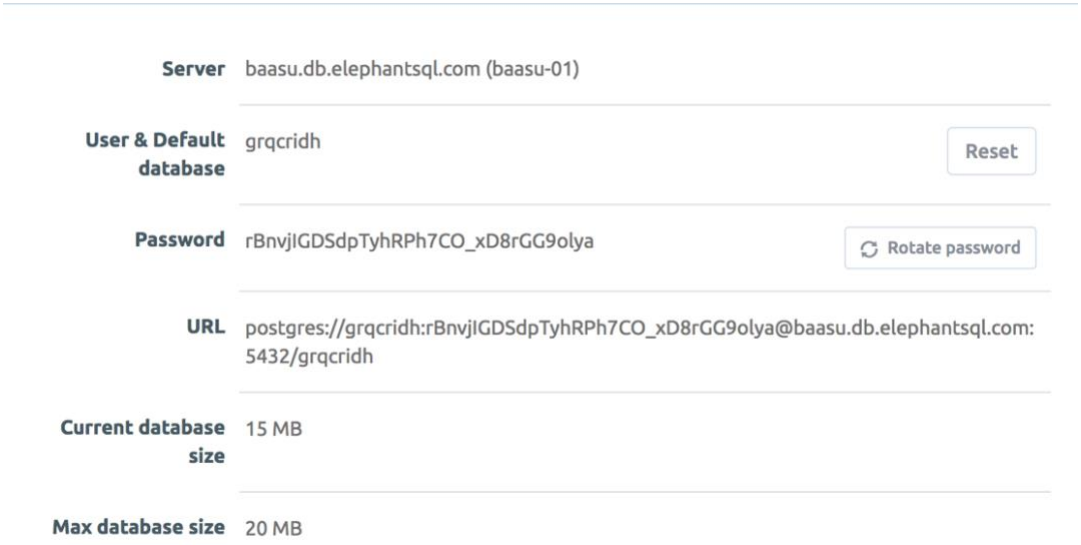


Figure 6. ElephantSQL database information.

After the connection, one can save to and load from the database.

```
if (connection != null) {
    System.out.println("Connection successful!");
    try {
        Statement st = connection.createStatement();
        ResultSet event = st.executeQuery( sql: "SELECT * FROM events");
        event.next();
        do{
            events.add(event.getString( columnLabel: "dates") + "," +
                    event.getString( columnLabel: "name") + "," +
                    event.getString( columnLabel: "location") + "," +
                    event.getString( columnLabel: "room") + "," +
                    event.getString( columnLabel: "time") + "," +
                    event.getString( columnLabel: "organizer"));
        }while(event.next());
        st.close();
        return events;
    } catch (SQLException e) {
        e.printStackTrace();
    }
} else {
    System.out.println("Failed to make connection!");
}
```

Figure 7. Database query.

In Figure 7, all elements are selected from the events table. To do so, first create a statement from the connection. After that create a ResultSet to store the result from executing the SQL query. One thing that needs to be remembered is that the result always starts with the row of the column names. To read the actual data, next() must be used to get to the next row. Moreover, getString() will give a string result for a particular column in the table. After all the action is done, the connection must be closed. For this application, other functions such as addAttendance, addAccount, and getPassword were also created. They all follow the same connection rules and if data does not need to be read from the database, a ResultSet is not necessary. All that is needed is executeUpdate() on the statement as shown in Figure 8.

```
try {
    Statement st = connection.createStatement();
    st.executeUpdate( sql: "INSERT INTO attendance " + "VALUES ('"+userid+"','"+event+"')");
    st.close();
} catch (SQLException e) {
    e.printStackTrace();
}
```

Figure 8. Example of executeUpdate().

After all the helper functions were set up, the data must make its way to the MainActivity class. The easiest way to do this is to create an object of the getEventList class in the main activity, then call execute().get() on said object to get the returned data and update the existing database, as shown in Figure 9.

```
try {
    //Get all events as a string list, value in each column is separated by  ","
    List<String> events = eventList.execute().get();
    //Add userid and event name to attendance table
    att.execute("zluo2@wpi.edu","event1");
    //Get a password by userid
    String password = pass.execute("zluo2@wpi.edu").get();
```

Figure 9. Getting data from the database with .execute().get().

Last but not least, if the connection still fails, the permission, "<uses-permission android:name="android.permission.INTERNET" />," in the Android manifest may not be set. The most important part for this application's database management is to connect to the SQL database hosted in the cloud. To do that, we installed JDBC file and used JDBC for the connection. In addition, helper functions for different SQL queries were built for future use in the application.

4.3 Google Map and GPS

Since there is not a large amount of time for the application, we use Google Maps to navigate to an event's location. Google Maps is easy to use and implement based on the API. Simply create an intent with android.content.Intent.ACTION_VIEW attribute and a URI to open Google Maps and show the given location.

```
Intent intent = new Intent(android.content.Intent.ACTION_VIEW,
    Uri.parse("google.navigation:q=an+address+city"));
```

Figure 10. Intent to launch Google Maps navigation.

For example, the code above will open Google Maps and look for “an address city”.

For the GPS part, as our app requires checking if the device is within some range of a point, Geofence is used. The first step in requesting geofence monitoring is to request the necessary permission. To use geofencing, the permission ACCESS_FINE_LOCATION must be requested. To request this permission, add the following element as a child element of the <manifest> element in the Android manifest:
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>.
To use an IntentService to listen for geofence transitions, add an element specifying the service name. This element must be a child of the <application> element as shown in Figure 11.

```
<application
   android:allowBackup="true">
   ...
   <service android:name=".GeofenceTransitionsIntentService"/>
<application/>
```

Figure 11. Requesting the Geofence transition service.

To access the location APIs, create an instance of the Geofencing client as shown in Figure 12.

```
private GeofencingClient mGeofencingClient;

// ...

mGeofencingClient = LocationServices.getGeofencingClient(this);
```

Figure 12. Accessing the location API of Geofence.

Now to create and add Geofences. First, use Geofence.Builder to create a geofence, setting the desired radius, duration, and transition types for the geofence. For example, to populate a list object named mGeofenceList as shown in Figure 13.

```
mGeofenceList.add(new Geofence.Builder()
    // Set the request ID of the geofence. This is a string to identify this
    // geofence.
    .setRequestId(entry.getKey())

    .setCircularRegion(
            entry.getValue().latitude,
            entry.getValue().longitude,
            Constants.GEOFENCE_RADIUS_IN_METERS
    )
    .setExpirationDuration(Constants.GEOFENCE_EXPIRATION_IN_MILLISECONDS)
    .setTransitionTypes(Geofence.GEOFENCE_TRANSITION_ENTER |
            Geofence.GEOFENCE_TRANSITION_EXIT)
    .build());
```

Figure 13. Building a Geofence.

After that specify geofences and initial triggers. Figure 14 uses the GeofencingRequest class and its nested GeofencingRequest.Builder class to specify the geofences to monitor and to set how related geofence events are triggered.

```
private GeofencingRequest getGeofencingRequest() {
    GeofencingRequest.Builder builder = new GeofencingRequest.Builder();
    builder.setInitialTrigger(GeofencingRequest.INITIAL_TRIGGER_ENTER);
    builder.addGeofences(mGeofenceList);
    return builder.build();
}
```

Figure 14. Building the Geofence request.

Next, define an intent for geofence transitions. The Intent sent from Location Services can trigger various actions in the app, but should not start an activity or fragment, because components should only become visible in response to a user action. In many cases, an IntentService is a good way to handle the intent. An IntentService can post a notification, do long-running background work, send intents to other services, or send a broadcast intent. Figure 15. shows how to define a PendingIntent that starts an IntentService.

```
public class MainActivity extends AppCompatActivity {

    // ...

    private PendingIntent getGeofencePendingIntent() {
        // Reuse the PendingIntent if we already have it.
        if (mGeofencePendingIntent != null) {
            return mGeofencePendingIntent;
        }
        Intent intent = new Intent(this, GeofenceTransitionsIntentService.class);
        // We use FLAG_UPDATE_CURRENT so that we get the same pending intent back when
        // calling addGeofences() and removeGeofences().
        mGeofencePendingIntent = PendingIntent.getService(this, 0, intent, PendingIntent.
                FLAG_UPDATE_CURRENT);
        return mGeofencePendingIntent;
    }
```

Figure 15. Using PendingIntent.

It is time to add geofences. To add geofences, use the GeofencingClient.addGeofences() method. Provide the GeofencingRequest object, and the PendingIntent.

We need to handle Geofence Transitions. When Location Services detects that the user has entered or exited a geofence, it sends out the Intent contained in the PendingIntent you included in the request to add geofences. This Intent is received by a service like GeofenceTransitionsIntentService, which obtains the geofencing event from the intent, determines the type of Geofence transition(s), and determines which of the defined geofences was triggered. It then sends a notification as the output. Figure 16 shows how when the user clicks the notification, the app's main activity appears.

```
public class GeofenceTransitionsIntentService extends IntentService {
    // ...
    protected void onHandleIntent(Intent intent) {
        GeofencingEvent geofencingEvent = GeofencingEvent.fromIntent(intent);
        if (geofencingEvent.hasError()) {
            String errorMessage = GeofenceErrorMessages.getErrorString(this,
                    geofencingEvent.getErrorCode());
            Log.e(TAG, errorMessage);
            return;
        }

        // Get the transition type.
        int geofenceTransition = geofencingEvent.getGeofenceTransition();

        // Test that the reported transition was of interest.
        if (geofenceTransition == Geofence.GEOFENCE_TRANSITION_ENTER ||
                geofenceTransition == Geofence.GEOFENCE_TRANSITION_EXIT) {

            // Get the geofences that were triggered. A single event can trigger
            // multiple geofences.
            List<Geofence> triggeringGeofences = geofencingEvent.getTriggeringGeofences();

            // Get the transition details as a String.
            String geofenceTransitionDetails = getGeofenceTransitionDetails(
                    this,
                    geofenceTransition,
                    triggeringGeofences
            );

            // Send notification and log the transition details.
            sendNotification(geofenceTransitionDetails);
            Log.i(TAG, geofenceTransitionDetails);
        } else {
            // Log the error.
            Log.e(TAG, getString(R.string.geofence_transition_invalid_type,
                    geofenceTransition));
        }
```

Figure 16. Geofence transitions.

Finally, the app needs to stop geofence monitoring when it is no longer needed or desired. We use PendingIntent to remove geofences.

```
mGeofencingClient.removeGeofences(getGeofencePendingIntent())
        .addOnSuccessListener(this, new OnSuccessListener<Void>() {
            @Override
            public void onSuccess(Void aVoid) {
                // Geofences removed
                // ...
            }
        })
        .addOnFailureListener(this, new OnFailureListener() {
            @Override
            public void onFailure(@NonNull Exception e) {
                // Failed to remove geofences
                // ...
            }
        });
```

Figure 17. Removing Geofences.

Lastly, after Geofence is set up, there is still one problem. The latitude and longitude for a location is not saved in the database, the only record is the name of the location. Therefore, in order to get the latitude and longitude with a location name, Geocoder is used. Geocoder is required to run on a thread and it is able to provide latitude and longitude with a given address.

```
public class GeocodingLocation {

    private static final String TAG = "GeocodingLocation";
    String value;


    public void getAddress(final String locationAddress, final Context context,final Handler handler) {

        new Thread(new Runnable() {
            @Override
            public void run() {
                Geocoder geocoder = new Geocoder(context, Locale.getDefault());
                String result = null;
                try {
                    List<Address> addressList = geocoder.getFromLocationName(locationAddress,  maxResults: 1);
                    if (addressList != null && addressList.size() > 0) {
                        Address address = addressList.get(0);
                        StringBuilder sb = new StringBuilder();
                        value = address.getLatitude() + "," + address.getLongitude();
                    }
                } catch (IOException e) {
                    Log.e(TAG,  msg: "Unable to connect to Geocoder", e);
```

Figure 18. Using Geocoding.

After the latitude and longitude are given, we need to return the output. In this case we use a handler as shown in Figure 19.

```
    } finally {
        if(value != null) {
            Message message = Message.obtain();
            message.setTarget(handler);
            message.what = 1;
            Bundle bundle = new Bundle();
            bundle.putString("address",value);
            message.setData(bundle);
            message.sendToTarget();
        } else
        {
            Message message = Message.obtain();
            message.setTarget(handler);
            message.what = 1;
            Bundle bundle = new Bundle();
            bundle.putString("address", "Did not find the location");
            message.setData(bundle);
            message.sendToTarget();
        }
```

Figure 19. Creating a handler.

After the latitude and longitude are saved, the handler is used to retrieve the data and set up geofence as shown in Figure 20.

```
//Give lat and lon and set geofences
private class GeocoderHandler extends Handler {
    @Override
    public void handleMessage(Message message) {
        String addr;
        switch (message.what) {
            case 1:
                Bundle bundle = message.getData();
                addr = bundle.getString( key: "address");
                break;
            default:
                addr = null;
        }
        String[] latlon = addr.split( regex: ",");
        lat = Double.parseDouble(latlon[0]);
        lon = Double.parseDouble(latlon[1]);
        System.out.println("lat: " + lat+ " lon: "+lon);
        populateGeofenceList(event_name, lat, lon);
    }
}
```

Figure 20. Creating Geofence with results of Geocoder.

Overall, to enable the navigation and location-context awareness features in our app, using the existing open libraries geofences and geocoder provides a great advantage and works great in our case.

5.0 Evaluation

5.1 Setup

For this project, the team is using a Nexus 5X running on API level 23, while building the application with minimum API level 23 and compiling to API level 26. For the database, we are using ElephantSQL- an open PostgreSQL database that hosted in the cloud. For the navigation function in the app, we used an intent to open the existing Google Maps app. For the GPS and location-context awareness in the app, we imported the Geocoder and Geofences libraries.

5.2 Methodology

In order to test the correctness of the project, we are first testing the app with all the features we listed in the design section, and then testing with some boundary situations. A standard use case of the app proceeds as follows:

- Launch the app
- Login page of the app, click to move to the registration screen (for a first time user)
- Registration page and register
- A user's dashboard which displays all the events on campus
- Logging out of the app
- User must log back in
- Detailed information about an event
- Click on the location of the event, will open a navigation
- Click on "interested?" will set a geofences for the in the app
- When the student arrives at the event location, the app will automatically send a notification
- Click on the notification brings the user back to the app

Edge cases for our application include the following:

- Attempt to login with an account that does not exist
- Attempt to login with an incorrect password
- Attempt to register with an email that already exists

5.3 Results

Standard use case

Running on a Nexus 5X running Android 6.0 and connected to WPI campus Wifi inside the library, the app is launched. As expected the splash screen appears. Since the user was not previously logged in, the login screen is displayed next.

Figure 21. The splash screen.

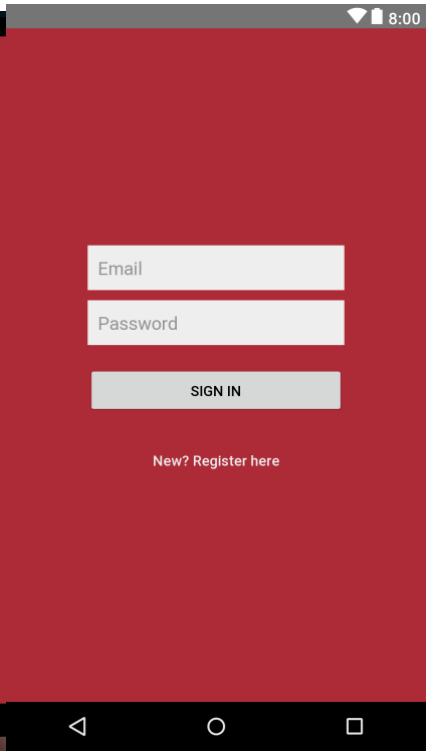
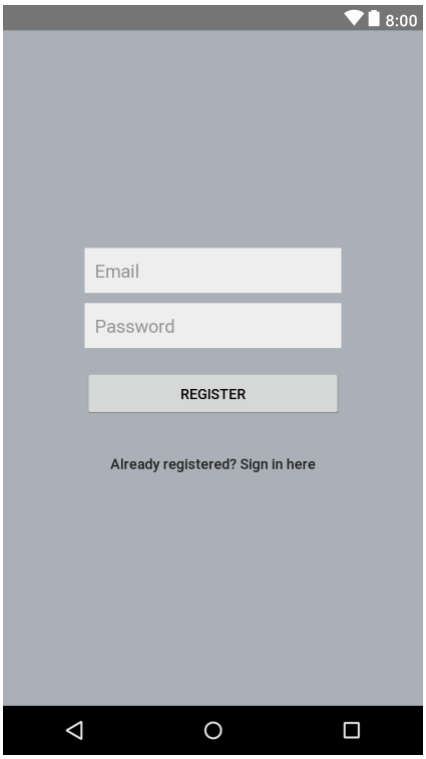


Figure 22. The login screen.

The "New? Register here" button is clicked, opening the registration screen. An email and password are put into their respective text fields and the register button clicked, both registering the user to the database and signing them in locally. This then opens the list of campus events.



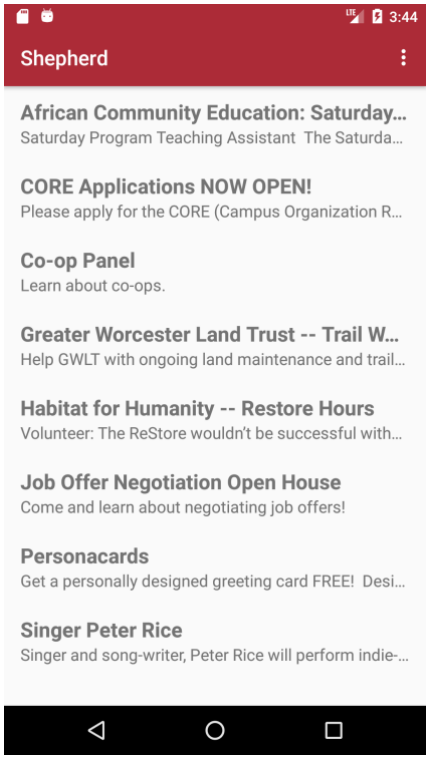


Figure 23. The registration screen. Figure 24. The event list.

To log out, the options menu in the top right corner is opened. This menu contains one item, "Log out," which is clicked, logging out the user and displaying the login screen. To continue using the app, the user must login again.

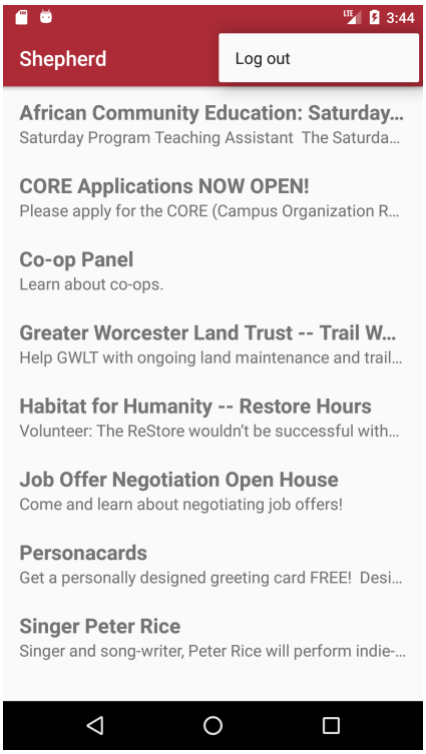


Figure 25. The options menu for the event list.

After logging back in with the same account, the event list is again shown. An event list element is clicked, opening the event details screen containing information about the selected event. Clicking the location of the event opens Google Maps with directions to the event's location.

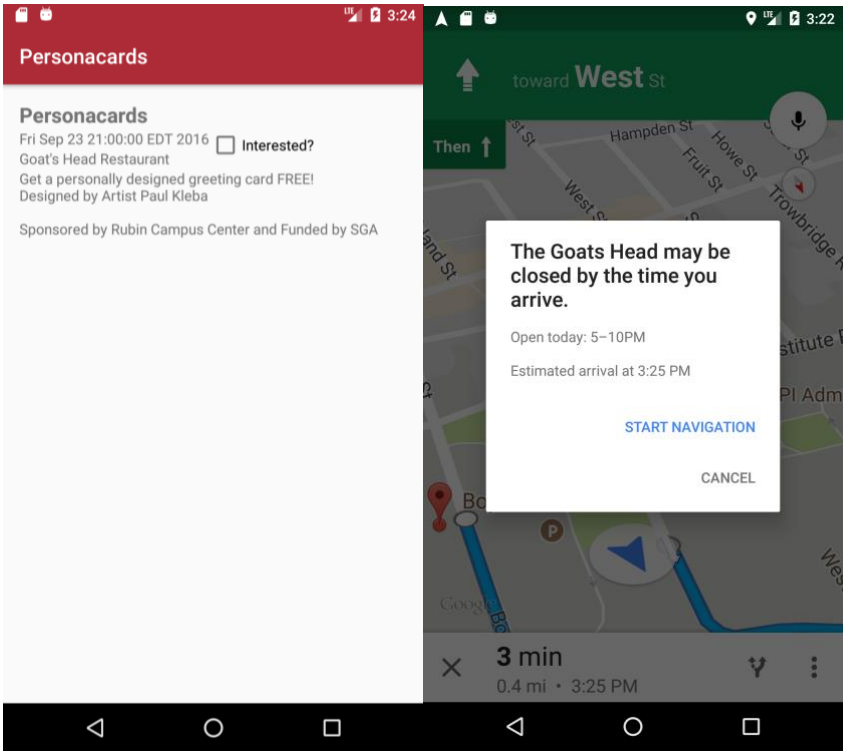


Figure 26. The event details.

Figure 27. Google Maps with directions.

A user's interest in the event can be expressed by selecting the "Interested?" checkbox. A notification is sent if the user is at the event's location after the during the event.

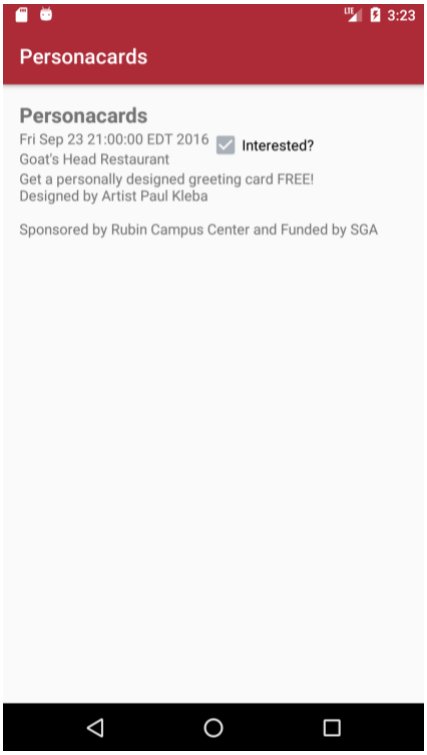


Figure 28. Selected "Interested?" checkbox.

After entering the location of an event during its listed time, a notification appears. Upon being clicked, it opens the app, displaying the check in menu. Clicking check in adds the user and the event to the attendance database.

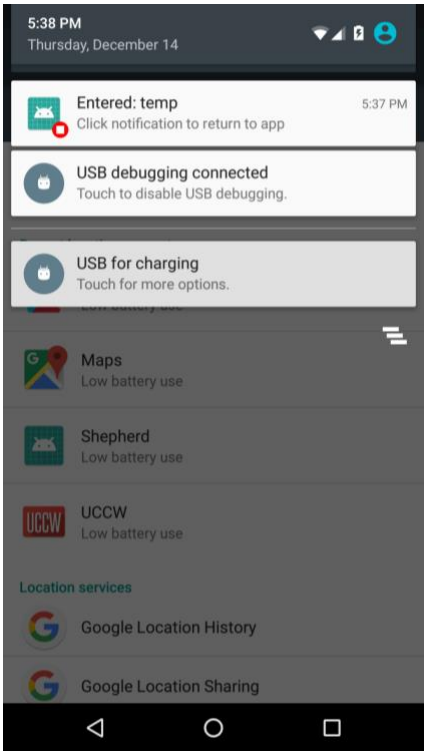


Figure 29. The notification appears.

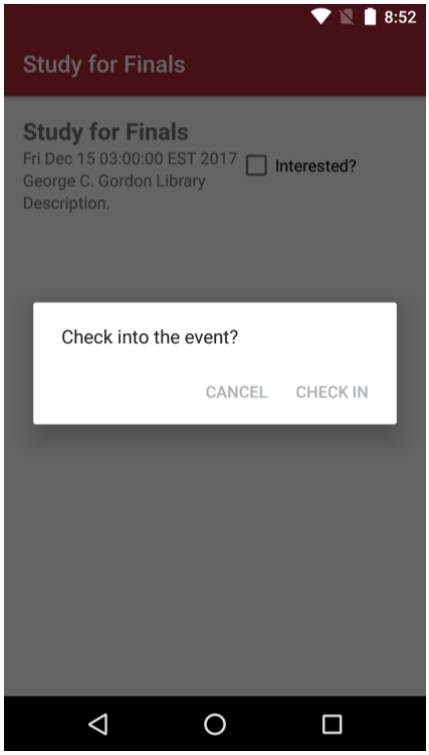


Figure 30. Check in menu.

This use case covers all database accesses as well as all transitions between activities in ideal conditions. All aspects of normal operation are functional.

Edge cases

To test login edge cases, the app is launched to the login screen. An email for which no account exists is entered. Instead of logging in the user, a toast indicating that no account exists is shown and the app remains at the login screen. An invalid password for an existing account is entered, also displaying a toast.

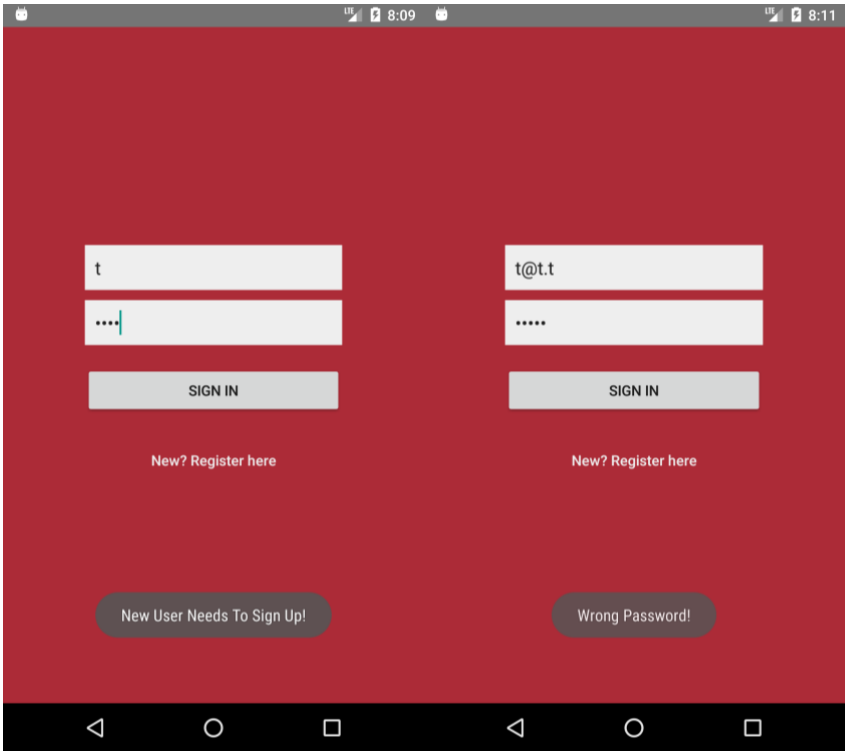


Figure 31. The user does not exist. Figure 32. The given password is invalid.

To test the registration edge case, the app is launched and the "New? Register here" button selected from the login screen. Now at the registration screen, an email is entered for which an account already exists. A toast is displayed to inform the user and no entry is added to the database.

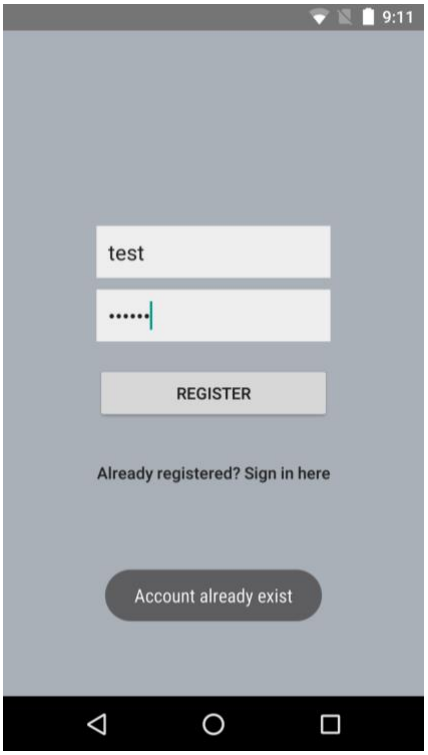


Figure 33. The account already exists.

6.0 Conclusion

In a nutshell, our team created a mobile application that takes advantage of location-context awareness to provide a better solution for campus event organizers to keep track of event attendance. The main features of the app are: requiring login with wpi username, displaying on-campus events to the users, allowing users to indicate their interest in attending an event and providing navigation to the location, displaying a notification for check-in when a user arrives at the event location, and storing the attendance into a database for the organizers. Two things we would suggest for future works related to this application are to try to acquire access to the CampusLabs API, which is the actual events database WPI uses, and add a guest sign in for visitors from outside WPI who want to join WPI events. During the implementation process, we made several changes to improve the application. One important change was switching from a local SQLite database to a PostgreSQL cloud database. We used a local SQLite database to start, as it is the simplest and easiest SQL database to implement in Android Studio. If it stayed that way, every user would be using a different database and cannot receive updates for events. Therefore, once we changed the database to cloud,

users could interact with outside world. Another change was changing the in-app map system to an intent that opened Google Maps. This change was done to take advantage of existing resources on the phone and save more time for implementation. Overall, by replacing sign-in sheet and students' ID cards with our mobile application, we believe it simplifies the check-in process and increases the accuracy of event attendance.