Goetz GPS Android App

Team members: **Ajay Gupta,**
Beshr Ibrik,
Dinil Karunaratne,
Shazil Arif,
Youssef Rizkalla

ICS4U

Date Started: June 11th, 2018
Last Updated: June 18th, 2018
Date Delivered: June 18th, 2018

Networking and Location Tracking

While the pathfinding can be done manually by having the user input which room they are standing at, implementing a fully operational real-time location tracker is more efficient and optimal especially for new students who may be completely clueless about their location.

At first, the approach to the location tracking was to use the concept of triangulation, however we soon realized that this method may be too complex and there may be an easier approach. David Moniaga arranged a meeting for us with a network engineer from Aruba Networks (the schools network provider). It came to our knowledge that almost all rooms are equipped with an access point. We realized that if we could have a unique identifier for each access point we could technically have a user's location and since the MAC address of each access point in the school is unique we just needed to get our hands on the addresses of each access point. A list of the MAC addresses of all Access Points was provided to us by Aruba Networks.

After have all MAC addresses, the approach was to do the following:

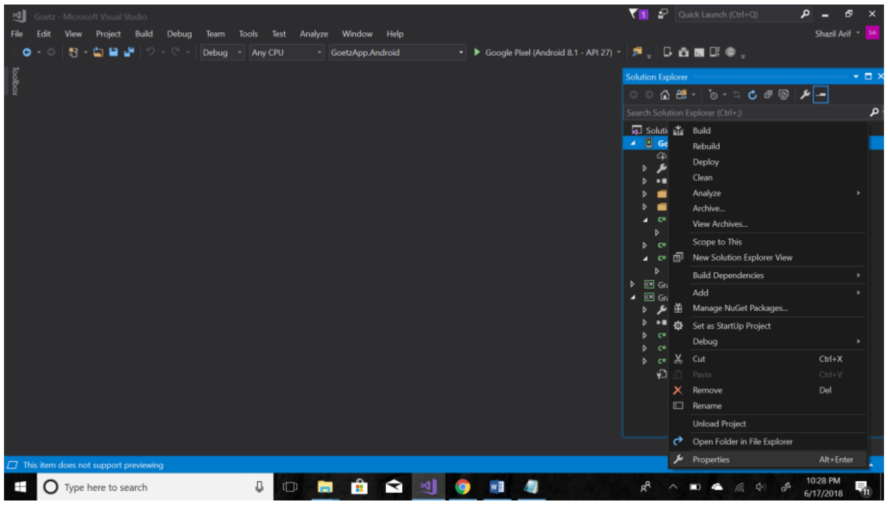
- Get required permission for using device's wireless features.
- Access all access points picked up user's device.
- Find the distance from the user's device to each access point.
- Find the Access Point with the minimum distance and retrieve its MAC address.
- Find the corresponding room to the MAC address stored in a text file-based database containing all the MAC addresses.

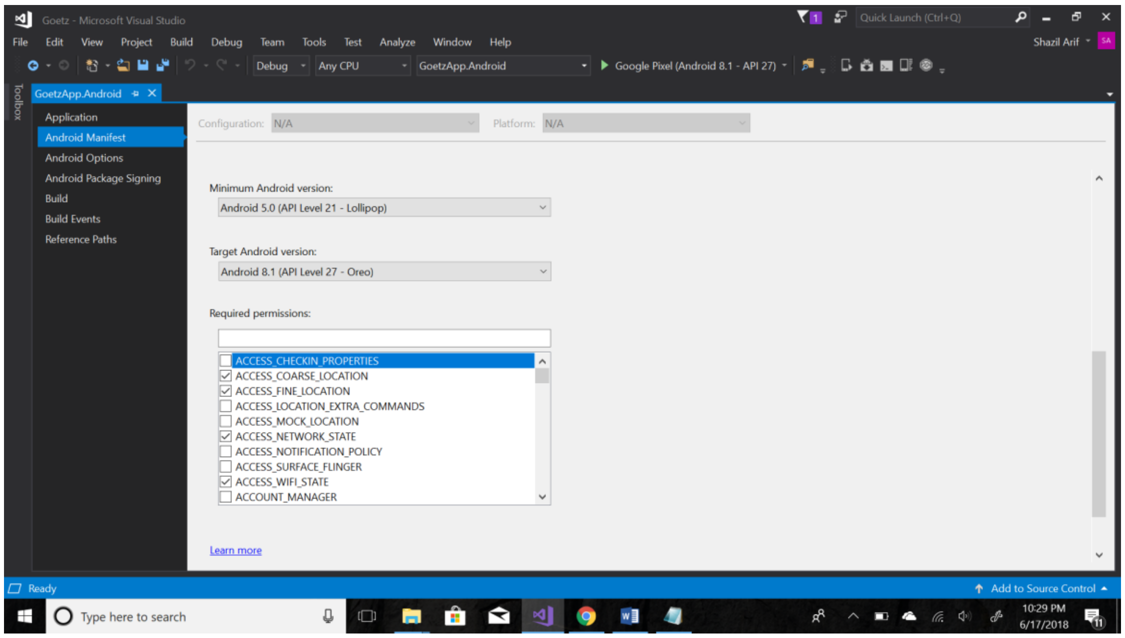
Getting Required Permissions

To access the Wi-Fi capabilities of a user's device we must first be able have permission to access these features. Permissions must be added manually in the android manifest file, the easiest way to do this is as follows:

On visual studio go to > View > Solution Explorer > Select the Android project > Right Click > Properties > Android Manifest > Check off required permissions.

A visual may help:





Configuration: N/A Platform: N/A

Minimum Android version: Android 5.0 (API Level 21 - Lollipop)

Target Android version: Android 8.1 (API Level 27 - Oreo)

Required permissions:

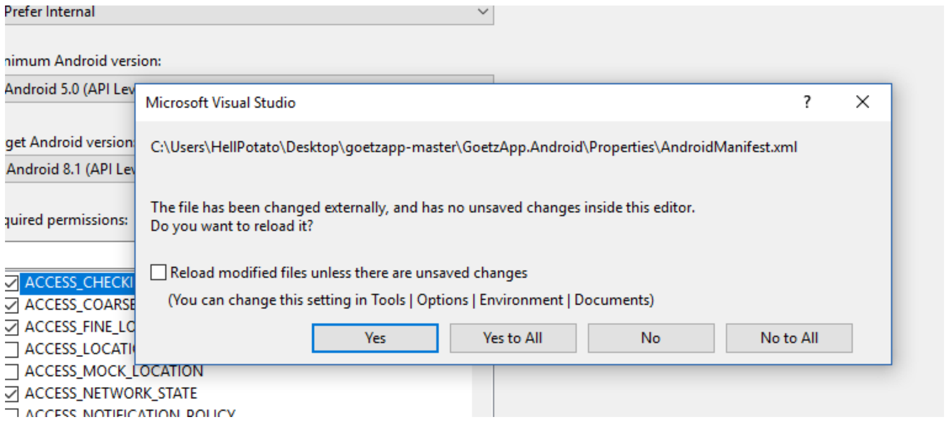
- [ ] ACCESS_CHECKIN_PROPERTIES
- [x] ACCESS_COARSE_LOCATION
- [x] ACCESS_FINE_LOCATION
- [ ] ACCESS_LOCATION_EXTRA_COMMANDS
- [ ] ACCESS_MOCK_LOCATION
- [x] ACCESS_NETWORK_STATE
- [ ] ACCESS_NOTIFICATION_POLICY
- [ ] ACCESS_SURFACE_FLINGER
- [x] ACCESS_WIFI_STATE
- [ ] ACCOUNT_MANAGER

Permissions can be now be added here manually.
Along with adding these permissions manually, certain permission must be confirmed by the user, such as the ACCESS_FINE_LOCATION, which is required for scanning Wi-Fi networks.
Note: **A wireless network scan cannot be performed without permission to the ACCESS_FINE_LOCATION permission granted manually by the user.**

The following permissions must all be checked off in the Android Manifest tab:

1. ACCESS_NETWORK_STATE
2. ACCESS_FINE_LOCATION
3. ACCESS_COARSE_LOCATION
4. ACCESS_WIFI_STATE
5. CHANGE_WIFI_STATE
6. CHANGE_WIFI_MULTICAST_STATE

Once a permission is checked off the following message will show up:



Microsoft Visual Studio

C:\Users\HellPotato\Desktop\goetzapp-master\GoetzApp.Android\Properties\AndroidManifest.xml

The file has been changed externally, and has no unsaved changes inside this editor.
Do you want to reload it?

Reload modified files unless there are unsaved changes
(You can change this setting in Tools | Options | Environment | Documents)

Select **Yes** to this message. This message shows up because the permissions are being written to the android manifest xml file.

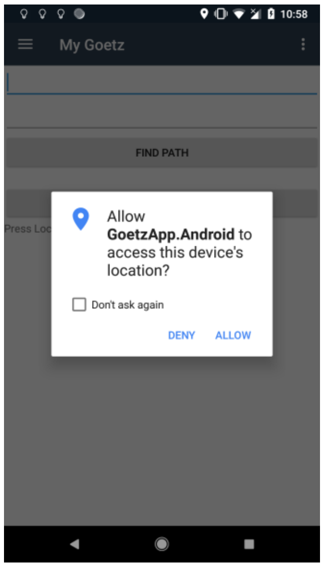
Now, before we start any wireless network scan, we must check if the ACCESS_FINE_LOCATION permission has been granted and if not, we must request the user for it.

The following are certain built in methods used for requesting permissions.

Activity.CheckSelfPermission(string Permission Name) checks whether a permission is granted or not (we check this because the user may have granted it previously and we do not want to prompt this each time).

Permission.Granted/Denied returns 0 if a permission is granted or -1 if a permission is denied.

ShouldShowRequestPermissionRationale(string Permission Name) returns true if a user previously denied access to a permission. It returns false if user denied access to a permission in the past and checked off the "Don't ask again". If the user checked off the latter, we will not further insist on gaining permission, the user may have accidentally checked this, but we cannot assume.
RequestPermissions (string array that contains the permissions, int Request Code) shows a pop-up message (note: the developer does not have control over the design/text of how this message looks and is standard) that the request the user for access to the desired permissions. The request code is a custom value assigned to identify the permission when there are multiple permissions being requested. The goal is to achieve a permission request that looks like:

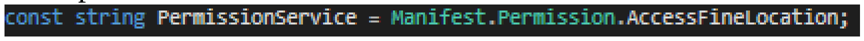


OnRequestPermissionsResult(int Request Code, string array containing permission, Permission[] AllowedPermission) is a callback method which means it is triggered after RequestPermissions() is called. This method returns the results of the requested permission. It returns a Permission[ ] array which includes the results of all permissions that were included in the string array passed into RequestPermission() method. The result is either 0 or -1. 0 indicating granted and -1 indicating denied.

Combining all these methods we can write pseudo code but before that, we must educate ourselves about two things:

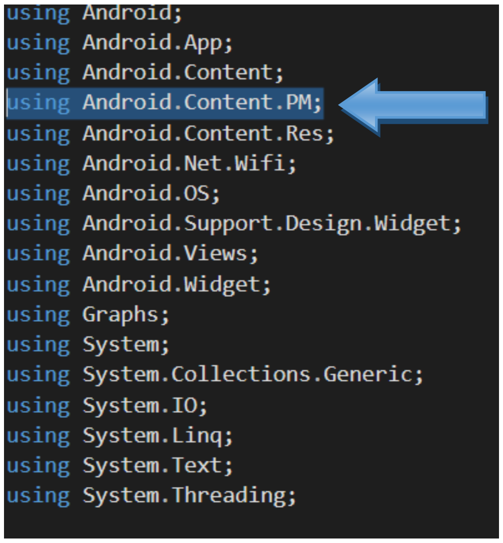
- The name space using Android.Content.PM is necessary to request permissions and use the methods described previously.
- Using Manifest.Permission.PermissionName we can assign a string a permission name.

Example:



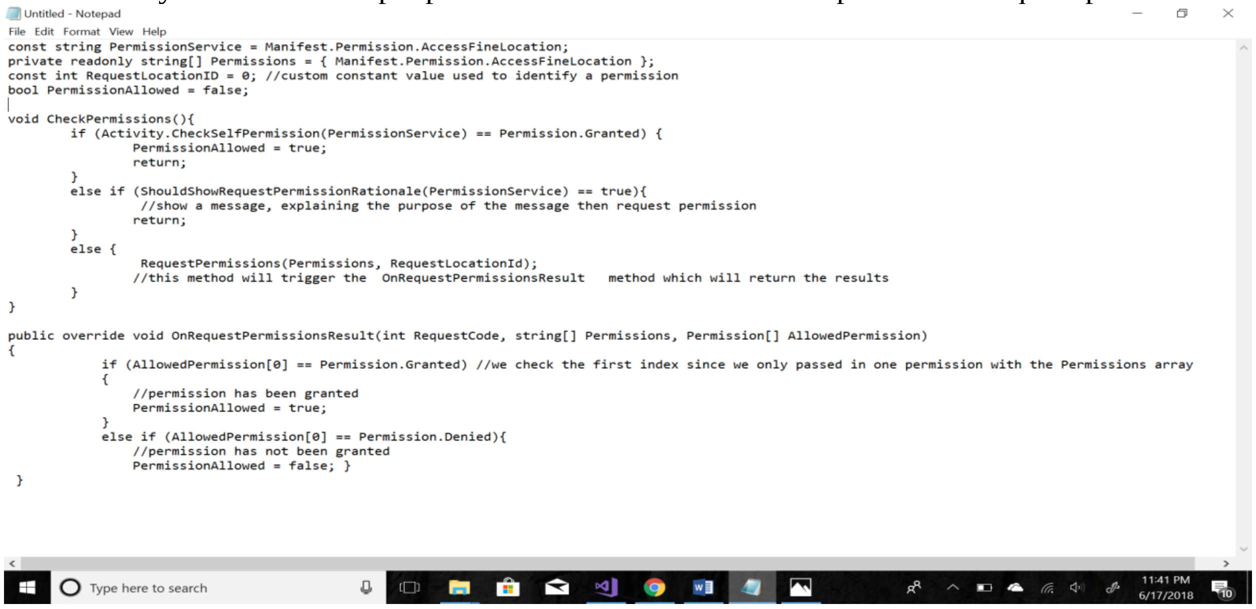
```
const string PermissionService = Manifest.Permission.AccessFineLocation;
```

Next, the required name space must be added as follows:



```
using Android;
using Android.App;
using Android.Content;
using Android.Content.PM;
using Android.Content.Res;
using Android.Net.Wifi;
using Android.OS;
using Android.Support.Design.Widget;
using Android.Views;
using Android.Widget;
using Graphs;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
```

Now we may write some simple pseudo code to demonstrate the permission request process.



```
const string PermissionService = Manifest.Permission.AccessFineLocation;
private readonly string[] Permissions = { Manifest.Permission.AccessFineLocation };
const int RequestLocationID = 0; //custom constant value used to identify a permission
bool PermissionAllowed = false;

void CheckPermissions(){
        if (Activity.CheckSelfPermission(PermissionService) == Permission.Granted) {
                PermissionAllowed = true;
                return;
        }
        else if (ShouldShowRequestPermissionRationale(PermissionService) == true){
                 //show a message, explaining the purpose of the message then request permission
                return;
        }
        else {
                 RequestPermissions(Permissions, RequestLocationId);
                //this method will trigger the  OnRequestPermissionsResult   method which will return the results
        }
}

public override void OnRequestPermissionsResult(int RequestCode, string[] Permissions, Permission[] AllowedPermission)
{
            if (AllowedPermission[0] == Permission.Granted) //we check the first index since we only passed in one permission with the Permissions array
            {
                //permission has been granted
                PermissionAllowed = true;
            }
            else if (AllowedPermission[0] == Permission.Denied){
                //permission has not been granted
                PermissionAllowed = false; }
 }
```

After we have permission to access the devices Wi-Fi capabilities we are ready to initiate a Wi-Fi scan.

The above code was an adaptation from a GitHub open source contributor.
Source:
https://github.com/egarim/XamarinAndroidSnippets/blob/master/XamarinAndroidRuntimePermissions

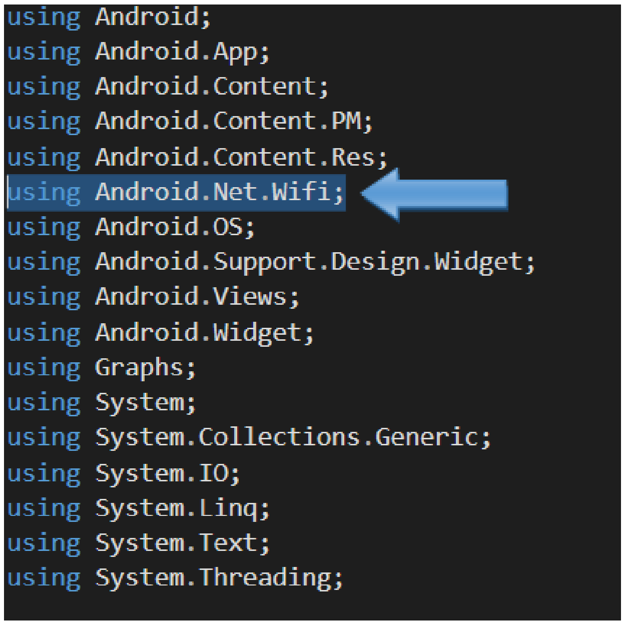
Scanning All Access Points

Before we explain the process of scanning different access points, it may be useful to familiarize ourselves with some terminology.

- **MAC address** – short for **media access control.** This is a unique identifier assigned to a network interface controller (NIC) for communications at the data link layer of a network segment. An example would be the NIC installed into a cellphone. This NIC is responsible for interacting with wireless networks and has an assigned constant value, known as its MAC address.
- **BSSID** – The MAC address of a wireless router.
- **Level/Signal** – the measured strength of a wireless network with the units "dbm", short for decibel-milliwatts.
- **Frequency** – A measure of how fast Wi-Fi waves are propagating. This is measured as the number of cycles the radio waves complete per second, commonly known as the unit Hertz.

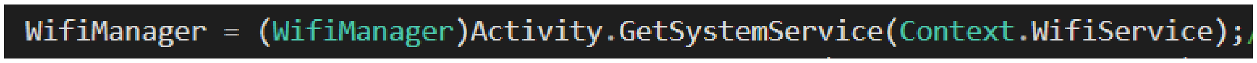
To access an android device's Wi-fi capabilities, retrieve wireless network information etc. we must use the WifiManager class.

We Start by declaring the "using Android.Net.Wifi" namespace:



```
using Android;
using Android.App;
using Android.Content;
using Android.Content.PM;
using Android.Content.Res;
using Android.Net.Wifi;
using Android.OS;
using Android.Support.Design.Widget;
using Android.Views;
using Android.Widget;
using Graphs;
using System;
using System.Collections.Generic;
using System.IO;
using System.Linq;
using System.Text;
using System.Threading;
```

To create an instance of the WifiManager class we use the following Syntax:



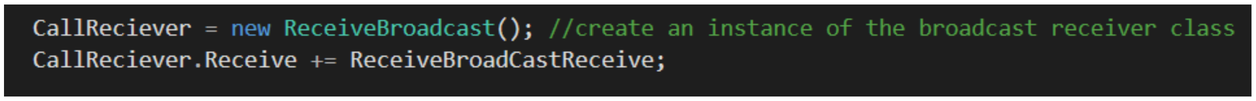
```
WifiManager = (WifiManager)Activity.GetSystemService(Context.WifiService);
```

We can then use the built-in function "WifiManager.StartScan()", which initiates a scan for wireless networks. We can then use "WifiManager.ScanResults" to retrieve a list of scanned wireless networks. Before we retrieve this list, it is important to note that the scanning process is asynchronous, and the scan results are **not** readily available. To receive this list of available networks we create a custom class that is a subclass from the BroadCastReceiver class. Broadcast receivers simply respond to broadcast messages from other applications or from the system itself. These messages are sometimes called events or intents. (source: https://www.tutorialspoint.com/android/android_broadcast_receivers.htm)

When we start the scan, the broadcast receiver is automatically triggered when it "listens" to any action that is registered within it. The "action" in our scenario is the availability of the scan results. The idea here is that the broadcast receiver "listens" for when scan results are available. The callback method "OnReceive()" in the class is then triggered which fires up our method in the original activity

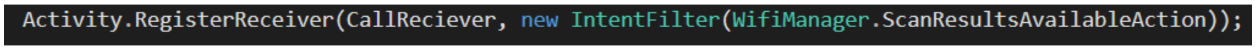
We must also "Register" and "Unregister" a broadcast receiver when we want to access it. It is necessary to unregister, so it does not keep running in the background and when the activity is started again, multiple broadcast receivers are not registered.

Here we create an instance of our custom broadcast receiver class and on the receive event we call the "ReceiveBroadCastReceive" method, where we can acquire our scan results.



```
CallReciever = new ReceiveBroadcast(); //create an instance of the broadcast receiver class
CallReciever.Receive += ReceiveBroadCastReceive;
```

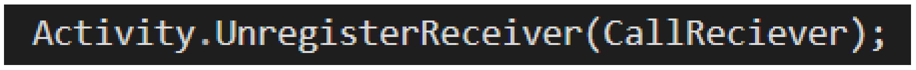
Register the broadcast receiver in our OnResume() method as follows:



```
Activity.RegisterReceiver(CallReciever, new IntentFilter(WifiManager.ScanResultsAvailableAction));
```

A OnResume() method is a callback method that is called any time the user resumes the current activity. Any process that must be initialized, resumed is placed in this method.

We must also unregister the broadcast receiver, so it does not "listen" for any events that it was registered for. Unregister as follows:



```
Activity.UnregisterReceiver(CallReciever);
```

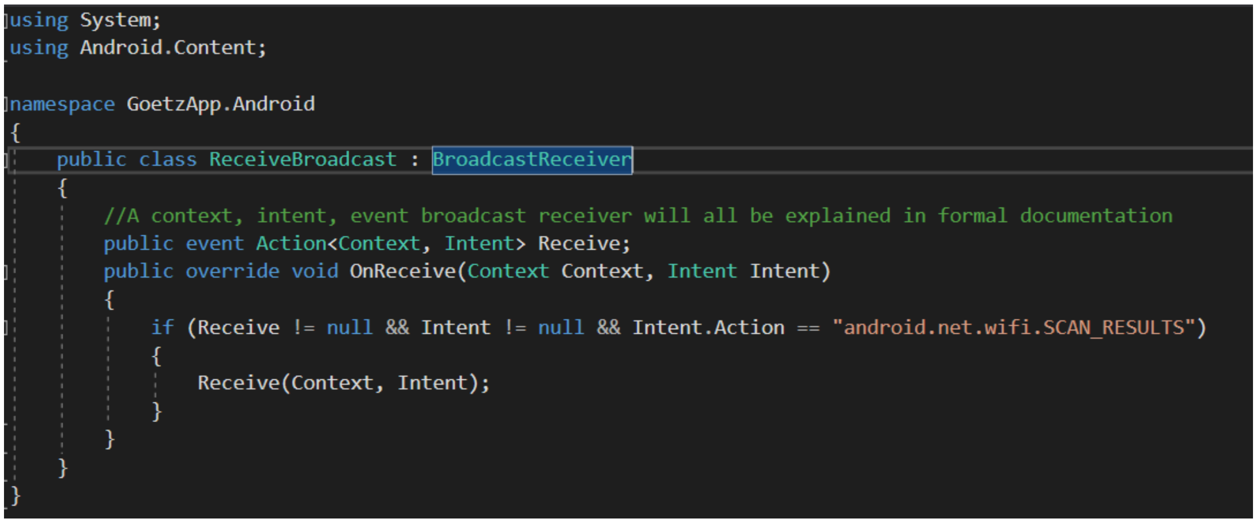
The above code is usually placed in the OnPause() method. An OnPause is a callback method that is executed when the user leaves a fragment/activity. We unregister the receiver here, so it is not running in the background.

For further information on OnPause() and OnReceive() callback methods see the following link : https://stuff.mit.edu/afs/sipb/project/android/docs/training/basics/activity-lifecycle/pausing.html

The broadcast receiver will listen for when scan results are available and call the Receive event which calls up the ReceiveBroadCastReceive() method. In our ReceiveBroadCastReceive() method we can retrieve the list of scanned networks as:

```
var Networks = WifiManager.ScanResults;
```

The following is an adaptation of a custom broadcast receiver class from an online source:
Source: https://www.youtube.com/watch?v=ktOQNBIfhQ4



```
using System;
using Android.Content;

namespace GoetzApp.Android
{
    public class ReceiveBroadcast : BroadcastReceiver
    {
        //A context, intent, event broadcast receiver will all be explained in formal documentation
        public event Action<Context, Intent> Receive;
        public override void OnReceive(Context Context, Intent Intent)
        {
            if (Receive != null && Intent != null && Intent.Action == "android.net.wifi.SCAN_RESULTS")
            {
                Receive(Context, Intent);
            }
        }
    }
}
```

We have now successfully retrieved a list of all Access Points detected by the user’s device, we are ready to use this data to locate our user.

Calculating the Distance from an Access Point

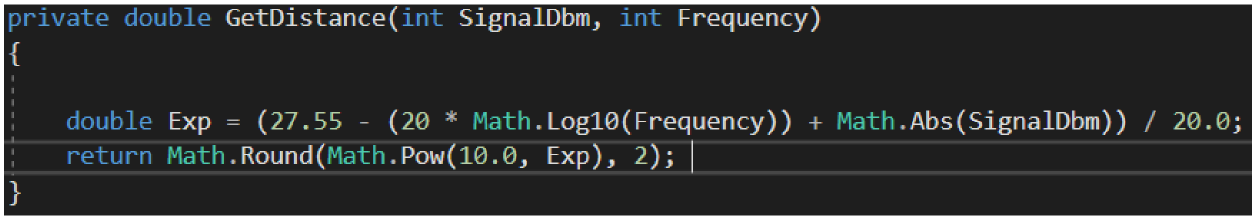
To calculate the distance from an access point we need two bits of information. The frequency of the network in megahertz and signal strength in decibel milliwatts. Luckily, the WifiManager class allows us to retrieve this information.

Once we have these two bits of information we use the following formula:

$$d = 10^{\frac{27.55-(20Log(F))+DBM)}{20}}$$

Where the value 27.55 is a fixed constant. "F" is the fequency measued in Megahertz and "DBM" is the signal strength in decibel milliwatts. It is important to note that in actual applications of the formula we use the **abolsute value** of the signal strength in dbm. The value of the constant 27.55 varies depending on the units one would like to retrive the distance in, as well the units of the frequency. For frequency in megahertz and distance calculated in metres we use the constant value of 27.55. For other constants see: https://en.wikipedia.org/wiki/Free-space_path_loss#Free-space_path_loss_in_decibels.

A programmatic implemenation of the formula is fairly straight forward. The formula was implemented as follows:



```
private double GetDistance(int SignalDbm, int Frequency)
{

    double Exp = (27.55 - (20 * Math.Log10(Frequency)) + Math.Abs(SignalDbm)) / 20.0;
    return Math.Round(Math.Pow(10.0, Exp), 2);
}
```

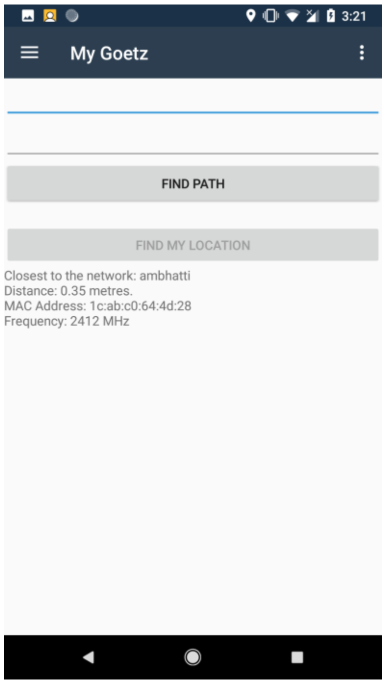
The formula was retrieved from: https://stackoverflow.com/questions/11217674/how-to-calculate-distance-from-wifi-router-using-signal-strength and was verified through several other sources. Multiple tests were also performed to ensure this formula gives an accurate distance.

Finding the Minimum Distance and Retrieving the Mac address

We can then find the access point with the minimum distance and retrieve its Mac address

Finally, we are done tracing our user's location. To update the user's location in real-time we use Thread.Sleep and set the timer to 1 millisecond and recursively call the method that initiates a Wi-fi scan.

Sample of retrieved Wi-Fi info:



Closest to the network: ambhatti
Distance: 0.35 metres.
MAC Address: 1c:ab:c0:64:4d:28
Frequency: 2412 MHz