Getting Started with Maps SDK

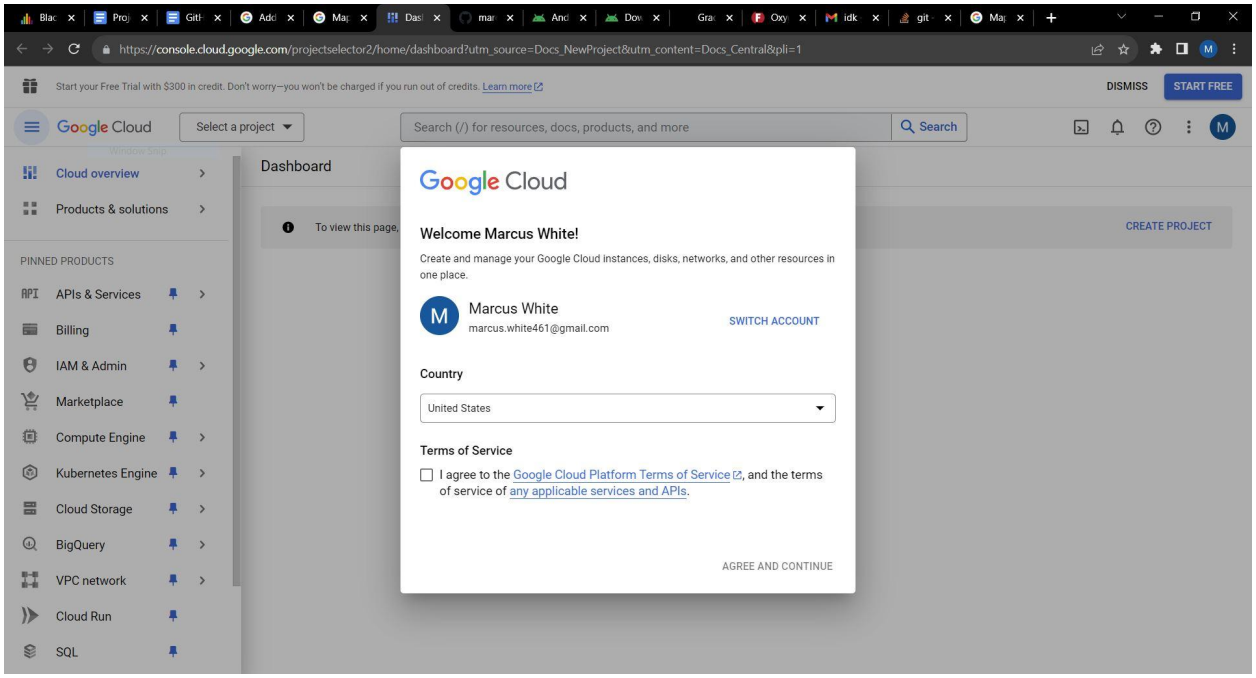
By Marcus White, Dan George, and Brady Triola

Overview

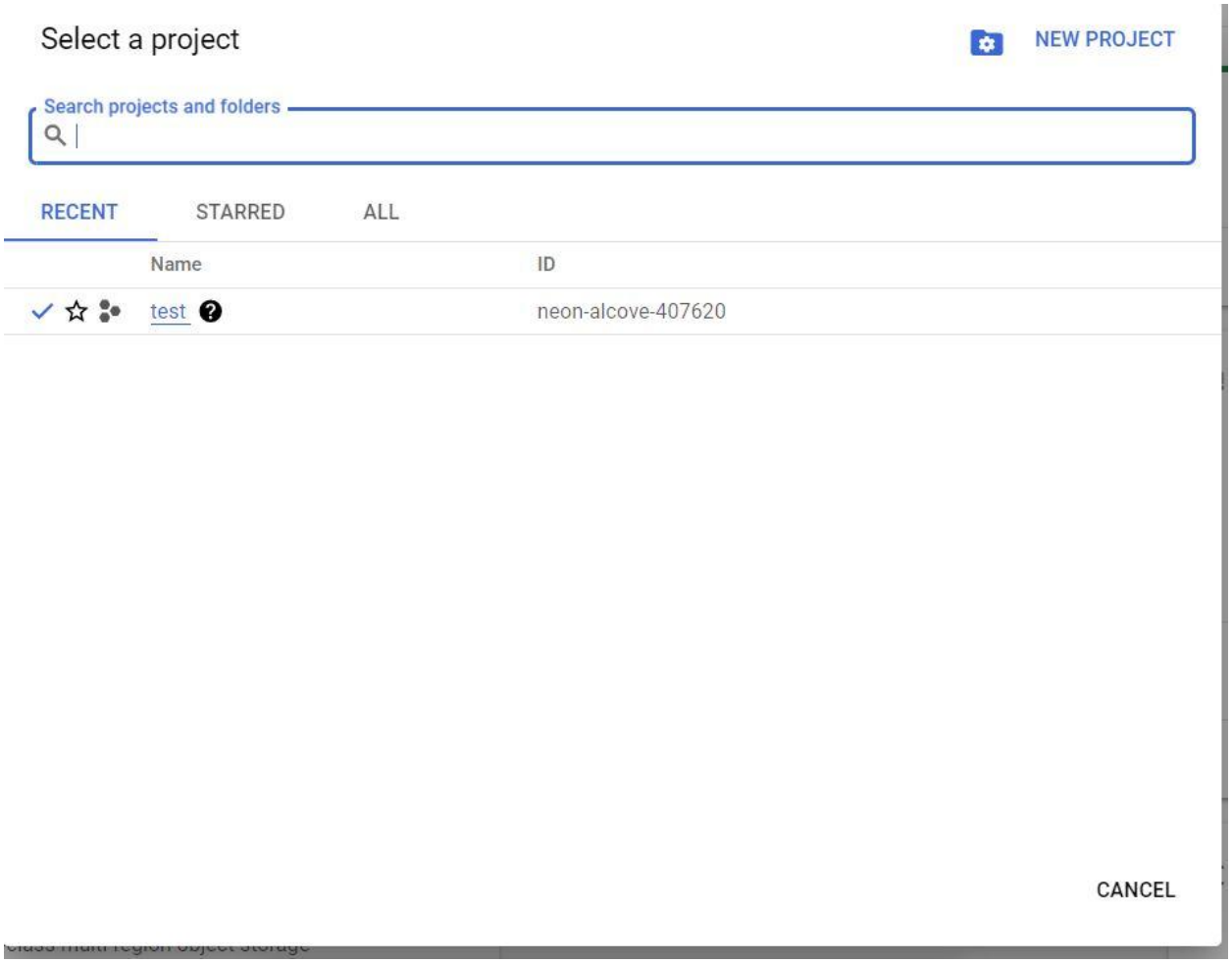
To preface everything done in this tutorial is done using Android Studio Giraffe/Hedgehog version and using the Kotlin language. Before getting started make sure you have a Google Cloud Project created, enable the SDK or API, and then get the keys for whichever you choose. To do this you must first create an account on Google Cloud.

Creating Google Cloud Account/Project

By clicking on this link you will be brought to this page:



From Here you will create an account and agree to their terms and conditions. Then in the top right you must create a new project.



Once you have created your project you must enable the Maps SDK for it which you can do here. After this generate your API/SDK key and youre ready to get started with your android studio project

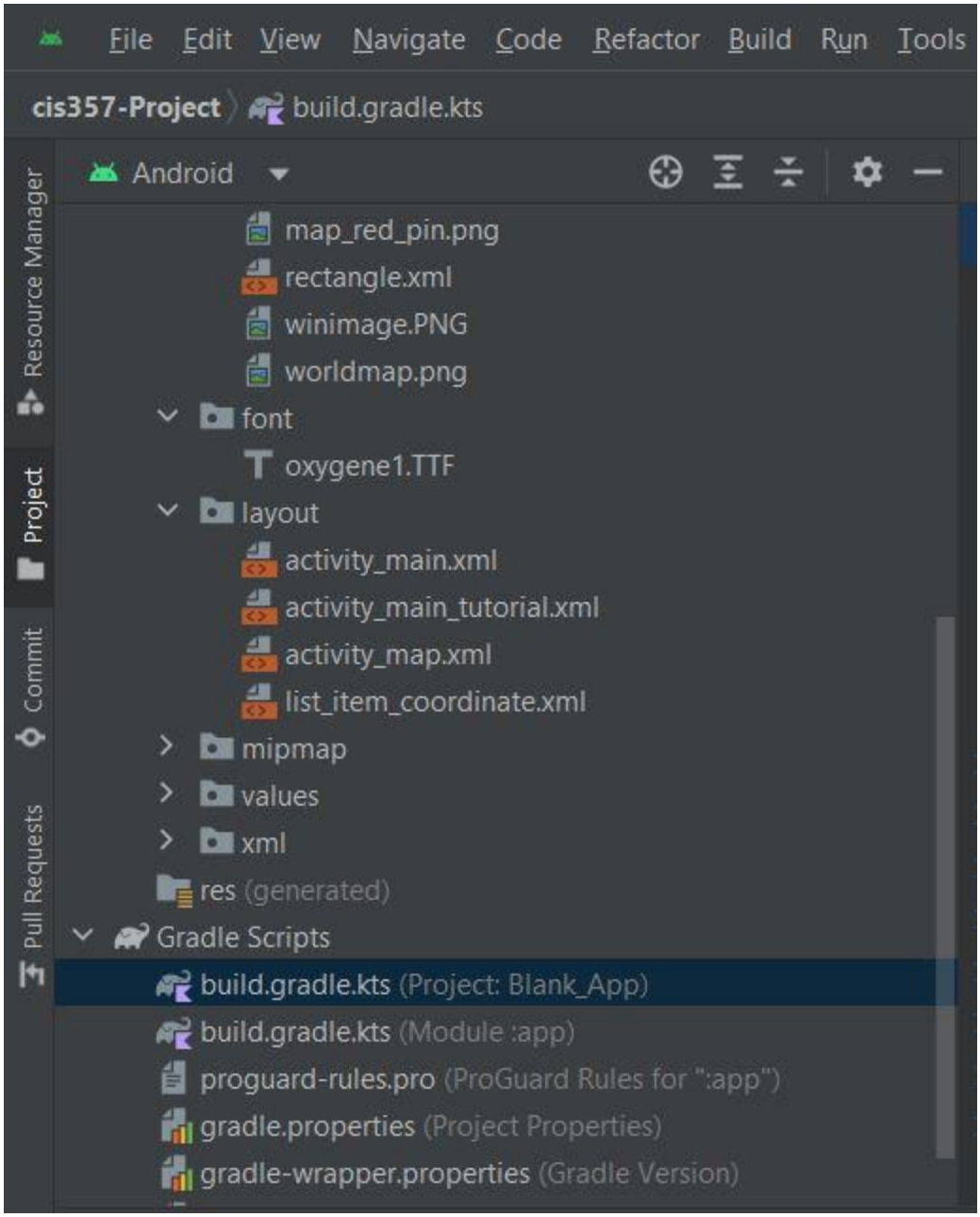
Android Studio

1. Open up your Android studio and click **Create New Project.** Once the New Project window pops up, under **Phone and tablet**, select a **Empty Activity**, and click **Next**.

2. Then create a Google Maps activity by clicking on **File**, **New**, **Google**, **Google Maps Activity,** from there select what language you want to use (this tutorial is using kotalin), and hit **Finish**.
3. Lastly is adding your API key to your app

Adding API to App

1. Open up your Project Manager on the left hand side and navigate to your build.gradle on the project level



2. Once you open up the file add the following code to the dependencies element under buildscript

```
//TODO:
//buildscript {
//    dependencies {
//        // ...
```

```
//        classpath
"com.google.android.libraries.mapsplatform.secrets-gradle-plugin:secrets-gradle
-plugin:1.3.0"
//    }
//}
```

3. Then open your module level build gradle and add the following code to the plugins element.

```
plugins {
   id("com.android.application")
   id("org.jetbrains.kotlin.android")
   id("com.google.android.libraries.mapsplatform.secrets-gradle-plugin")
   }
```

4. Save your files and sync the project with the gradle. This step will take around 10 minutes and you must let this sync or it'll cause problems in your project.
5. Once this is done you must locate a file named “local.properties” which will be at the bottom of your gradle scripts folder. Once you find it click on it and add the following code: “MAPS_API_KEY=YOUR_API_KEY”. Replace the “your api key” with the API key you generated you generated in the beginning.
6. Lastly find your AndroidManifest.xml file which is localled at the top of your project manager in the manifest folder. Then add the following code:

```
<meta-data
   android:name="com.google.android.geo.API_KEY"
   android:value="${GOOGLE_MAPS_API_KEY}" />
```

Implementing Maps

Clicking Pins

A crucial feature of the Map SDK for android is the ability to click, adjust, modify location pins, and grab the coordinates of the pins. You can adjust how a pin is place, what happens to the pin when its clicked, where to store the coordinates, and many more options you can explore in its documentation.

Starting off well talk you through step by step on how to set, adjust, and use pins.

To start you must find the Map Activity file which was created when you created the Google Maps view. This file will hold all the code pertaining to the google map and everything associated with it. It'll also hold our demos main chunk of code logic for the game.

```
class MapActivity : AppCompatActivity(), OnMapReadyCallback,
GoogleMap.OnMapLongClickListener,
  GoogleMap.OnMarkerClickListener {


  //Google map object: this is where we do all our runtime actions
  private var mGoogleMap:GoogleMap? = null
  private var markerCount = 0
  private var lifeCount = 3
  //list of markers
  private var selectedMarker: Marker? = null
  private val gvsu = LatLng(42.9636004, -85.8892062)
```

```
private val markerList: MutableList<Marker> = mutableListOf()
private val markerColors: MutableMap<Marker, Float> = mutableMapOf()
private var coordinateList: List<LatLng> = emptyList()
//recycler view
lateinit var recyclerView: RecyclerView
private lateinit var recyclerViewContainer: LinearLayout
private lateinit var customAdapter: CustomAdapter
```

Some of the major functions in this class included onMarkerClick and onMapLongClick. You will use onMarkerClick to register any clicks the users does on the markers on the map and the OnMapLongClick to place markers.

onMarkerClick:

To change a marker when it is selected you must first initialize it first. As below:

```
private var selectedMarker: Marker? = null
```

Then incorporate the logic of a marker not being selected and then being selected.

```
override fun onMarkerClick(marker: Marker): Boolean {
  // show markerList
  showRecyclerView()

  // set Marker
  if (selectedMarker == null) {
     // No marker is currently selected
     selectedMarker = marker
```

```
marker.setIcon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_ORANGE))
        customAdapter.setSelectedCoordinate(marker.position)
```

After checking to see if it something has been selected itll set selectedMarker to the marker that you clicked on, then that marker is set to any color that you choose, and the coordinate is sent to customAdapter as the setSelectedCoordinate. See code above for example.

```
} else {
    // A marker is already selected
    if (marker.position == selectedMarker?.position) {
        // The clicked marker has the same coordinates as the selected marker
        // Player does not lose a life, and the marker turns green

selectedMarker?.setIcon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_GREEN))
        markerColors[marker] = BitmapDescriptorFactory.HUE_GREEN
        checkWinCondition()
        selectedMarker = null
```

The next else segment deals with the case of having a marker already selected and choosing the correct coordinate, this means that one of the markers is orange! In this case, if the orange marker matches the coordinate selected, then the marker will turn green indicating that you got it right! Then the selectedMarker will be set to null so that you can make another selection!

```
} else {
        // The clicked marker has different coordinates from the selected marker
```

```
        // Player loses a life, and the marker turns blue

selectedMarker?.setIcon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_RED))
        Toast.makeText(this, "Wrong coordinate, Lost a life!", Toast.LENGTH_SHORT).show()
        markerColors[marker] = BitmapDescriptorFactory.HUE_RED
        decreaseLifeCount()
        selectedMarker = null
    }

    hideRecyclerView()
}
```

The last case deals with the current marker not matching the selected coordinate. This means that you got it wrong, the marker will then turn red and the player will lose a life! You only get 3 lives to start, so use them wisely! Lastly, the onMarkerClick method calls to hide the recycler view so that the next marker can be selected!

onMapLongClick:

```
override fun onMapLongClick(pointClicked: LatLng) {
    Log.d("DANS:", "lat is:$pointClicked")
    markerCount ++
    val newMarker = mGoogleMap!!.addMarker(MarkerOptions()
        .position(pointClicked)
        .icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_RED))
```

```
        .title("Marker $markerCount"))
    //add marker to list
    if (newMarker != null) {
        markerList.add(newMarker)
        markerColors[newMarker] = BitmapDescriptorFactory.HUE_RED
        updateCoordinateList()
    }

}
```

This function's purpose was to add a marker when the user long clicks on the map. In order to give each marker a unique ID we used markerCount ++ to increment each time a new marker was added and then used $markerCount to name the marker (Marker #) with the # being equal to markerCount. We also used the variable markerColors to determine what color each of the markers were. This was helpful in detecting a win, when all of the markerColors were green.

OnMapReady:

```
override fun onMapReady(googleMap: GoogleMap) {
    mGoogleMap = googleMap
    mGoogleMap!!.setOnMapLongClickListener(this)

    //default screen on GVSU
    mGoogleMap!!.addMarker(MarkerOptions().position(gvsu)
        .icon(BitmapDescriptorFactory.defaultMarker(BitmapDescriptorFactory.HUE_BLUE)))
    mGoogleMap!!.moveCamera(CameraUpdateFactory.zoomTo(12.0F))
    mGoogleMap!!.moveCamera(CameraUpdateFactory.newLatLng(gvsu))

    //marker/pin listener
    googleMap.setOnMarkerClickListener(this)

}
```

Another important function is OnMapReady(). This function was used to create the google map, start the click listener, and set the starting position to GVSU with a blue pin on GVSU.

Game Quality Functions:

```
private fun updateCoordinateList() {
    coordinateList = markerList.map { it.position }
    customAdapter.notifyDataSetChanged()
}
private fun showRecyclerView() {
    recyclerViewContainer?.visibility = View.VISIBLE
    recyclerView?.visibility = View.VISIBLE

}

private fun hideRecyclerView() {
    recyclerViewContainer?.visibility = View.GONE
    recyclerView?.visibility = View.GONE

}

private fun updateLifeCounter() {
    findViewById<TextView>(R.id.lifeCount).text = "LIVES: $lifeCount"
}

private fun decreaseLifeCount() {
    lifeCount--
    updateLifeCounter()
    if (lifeCount <= 0) {
        Toast.makeText(this, "You lost! Try again!", Toast.LENGTH_SHORT).show()
        gameOver()
    }
}
private fun checkWinCondition() {
    val allMarkersAreGreen = markerColors.values.all { it ==
BitmapDescriptorFactory.HUE_GREEN }

    if (allMarkersAreGreen) {
        // All markers are green, notify the player of the win
        Toast.makeText(this, "Congratulations! You won!", Toast.LENGTH_SHORT).show()
        gameOver()
    }
}

private fun gameOver() {
    finish()
```

```
}
```

These functions were put in place to make this map feel more like a game. These are to check win conditions, loss conditions, game over if the player loses, etc… Without these, the app would just be a map!

CustomAdapter:

```
package com.example.blank_app

import android.view.LayoutInflater
import android.view.View
import android.view.ViewGroup
import android.widget.TextView
import androidx.recyclerview.widget.RecyclerView
import com.google.android.gms.maps.model.LatLng


class CustomAdapter(
    private val coordinateList: List<LatLng>,
    private val onCoordinateClickListener: (LatLng) -> Unit
) : RecyclerView.Adapter<CustomAdapter.CoordinateViewHolder>() {

    private var selectedCoordinate: LatLng? = null

    inner class CoordinateViewHolder(itemView: View) : RecyclerView.ViewHolder(itemView) {
        val textViewCoordinate: TextView = itemView.findViewById(R.id.textViewCoordinate)

        init {
            itemView.setOnClickListener {
                val position = adapterPosition
                if (position != RecyclerView.NO_POSITION) {
                    onCoordinateClickListener(coordinateList[position])
                }
            }
        }
    }

    override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): CoordinateViewHolder {
        val view = LayoutInflater.from(parent.context)
            .inflate(R.layout.list_item_coordinate, parent, false)
```

```
        return CoordinateViewHolder(view)
    }

    override fun onBindViewHolder(holder: CoordinateViewHolder, position: Int) {
        val coordinate = coordinateList[position]
        val text = "Lat: ${coordinate.latitude}, Lng: ${coordinate.longitude}"
        holder.textViewCoordinate.text = text

        // Highlight the selected coordinate
        holder.itemView.isSelected = coordinate == selectedCoordinate
    }

    override fun getItemCount(): Int {
        return coordinateList.size
    }

    fun setSelectedCoordinate(selectedCoordinate: LatLng) {
        this.selectedCoordinate = selectedCoordinate
        notifyDataSetChanged()
    }
}
```

We also used a class named CustomAdapter to help run our maps project, the main use of this adapter was to handle inputs like our coordinateList and display it in the ways that we wanted. Between these two major classes there were many calls to update or notify data set changes and the google API was helpful in determining when we needed them.

Conclusion:

The Google Maps SDK for Android is a helpful set of tools for app creators. It lets them easily add location-based features to their apps. It can show maps, customize markers, and make interactive maps that look good and are easy for users to use. The SDK can do more than just maps—it also helps with finding locations, planning routes, and updating information in real-time. This makes it useful for a lot of different apps, like navigation or tracking services. The SDK also works well with other Google services, has lots of information available, and has a

supportive community, making it a great choice for developers who want to add strong mapping features to their apps and improve the user experience.