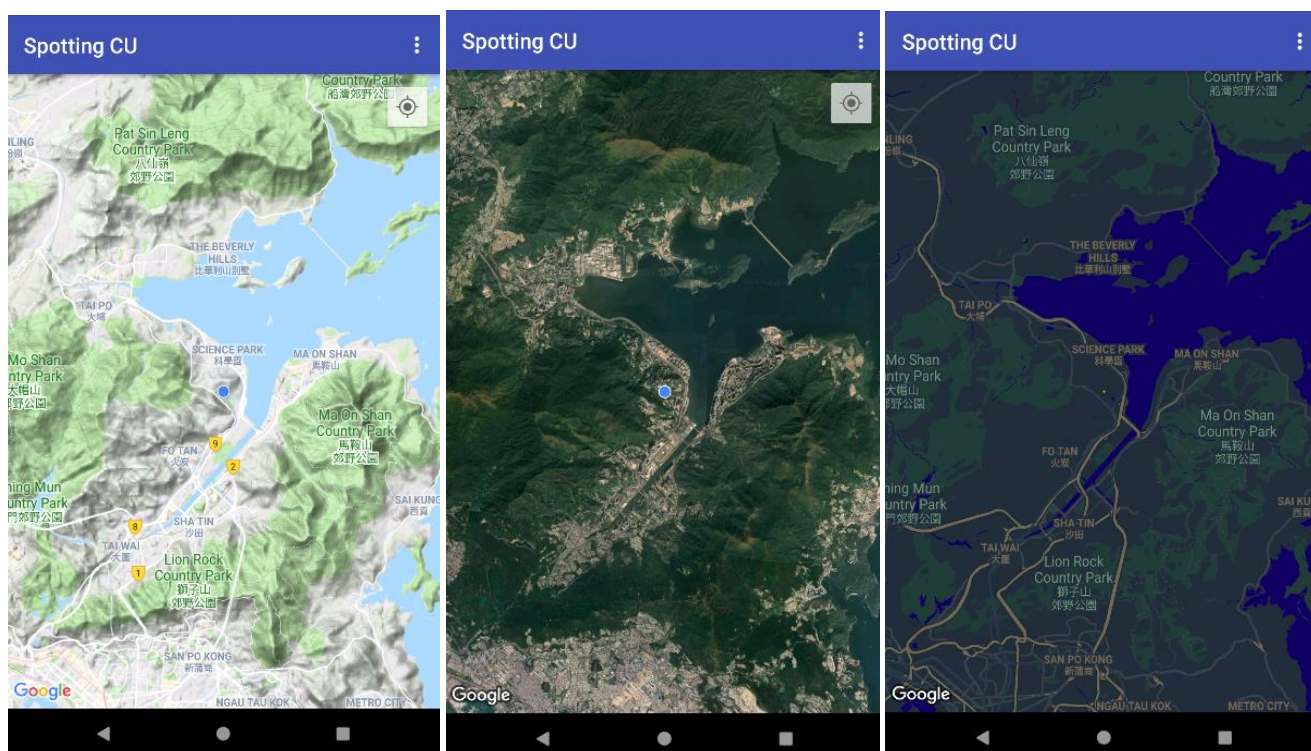


CSCI3310 Mobile Computing & Application Development

Lab 7 – Google Map: Spotting CU

Introduction

In this lab you create the Spotting CU app, which is a styled Google Map. The Spotting CU app allows you to drop markers onto locations, see your location in real time.



Todo

- Get an API key from the Google API Console and register the key to your app.
- Set Runtime permissions
- Include external libraries in your build.gradle file.
- Create the Spotting CU app, which has an embedded Google Map.
- Add custom features to your app such as markers, styling, and location tracking.
- Enable location tracking and Street View in your app.

1. Set up the project and get an API Key

The Google Maps API, like the Places API, requires an API key. To obtain the API key, you register your project in the Google API Console. The API key is tied to a digital certificate that links the app to its author. For more about using digital certificates and signing your app, see [Sign Your App](#).

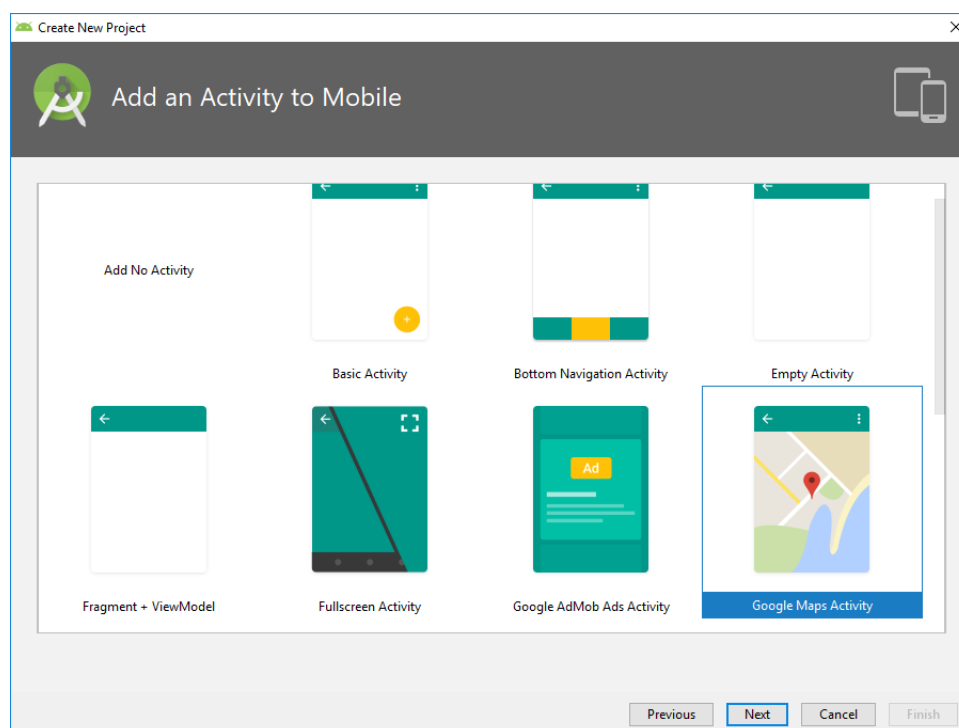
In this lab, you use the API key for the debug certificate which is insecure by design for release. Published Android apps that use the Google Maps API require a second API key for the release certificate. For more, see [Get API Key](#).

Android Studio includes a Google Maps Activity template, which includes a `google_maps_api.xml` file containing a link that simplifies obtaining an API key.

Note: If you want to build the Activity without using the template, follow the steps in the [API key guide](#) to obtain the API key without using the link in the template.

1.1 Create the Spotting CU project with the Maps template

1. Create a new Android Studio project.
2. Name the new app "Spotting CU". Accept the defaults until you get to the **Add an Activity** page.
3. Select the **Google Maps Activity** template.
4. Leave the default **Activity Name** and **Layout Name**.
5. Change the **Title** to "Spotting CU" and click **Finish**.



Android Studio creates several maps-related additional files:

google_maps_api.xml

- You use this configuration file to hold your API key.
The template generates two google_maps_api.xml files: one for debug and one for release. The file for the API key for the debug certificate is located in `src/debug/res/values`. The file for the API key for the release certificate is located in `src/release/res/values`. In this lab we only use the debug certificate.

activity_maps.xml

- This layout file contains a single fragment that fills the entire screen. The [SupportMapFragment](#) class is a subclass of the Fragment class. You can include

SupportMapFragment in a layout file using a `<fragment>` tag in any ViewGroup, with an additional attribute:

```
android:name="com.google.android.gms.maps.SupportMapFragment"
```

MapsActivity.java

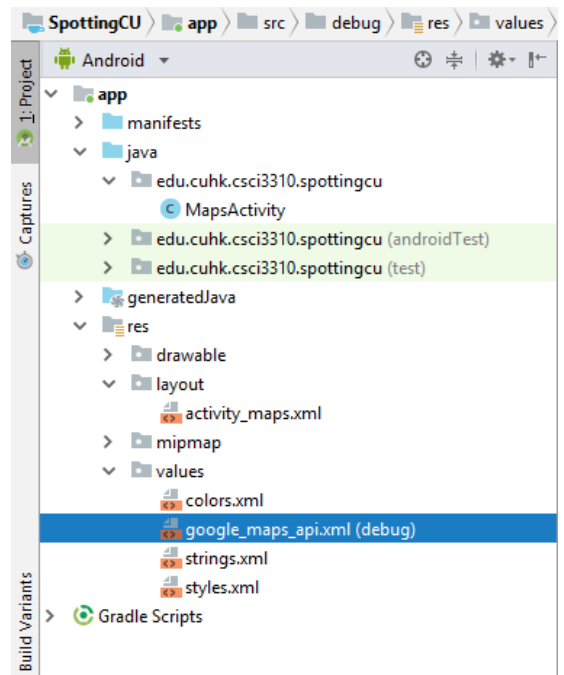
- The MapsActivity.java file instantiates the `SupportMapFragment` class and uses the class's `getMapAsync()` method to prepare the Google Map. The activity that contains the `SupportMapFragment` must implement the [OnMapReadyCallback](#) interface and that interface's `onMapReady()` method. The `getMapAsync()` method returns a `GoogleMap` object, signifying that the map is loaded.

If you run the app now, the map fails to load. If you look in the logs, you see a message saying that your API key is not properly set up. In the next step, you obtain the API key to make the app display the map.

Note: If you test the Spotting CU app on an emulator, use a system image that includes Google APIs and Google Play. Select an image that shows **Google Play** in the **Target** column of the virtual-devices list.

1.2 Obtain the API key

1. Open the debug version of the `google_maps_api.xml` file. The file includes a comment with a long URL. The URL's parameters include specific information about your app.
2. Copy and paste the URL into a browser.
3. Follow the prompts to create a project in the Google API Console. Because of the parameters in the provided URL, the API Console knows to automatically enable the Google Maps Android API



4. Create an API key and click **Restrict Key** to restrict the key's use to Android apps. The generated API key should start with Alza.
5. In the `google_maps_api.xml` file, paste the key into the `google_maps_key` string where it says **YOUR_KEY_HERE**.
6. Run your app. You have an embedded map in your activity, with a marker set in Sydney, Australia. (The Sydney marker is part of the template, and you change it later.)

Note: The API key may take up to 5 minutes to take effect.

2. Add map types and markers

Google Maps include several map types: normal, hybrid, satellite, terrain, and "none." In this task, you:

1. add an app bar with an options menu that allows the user to change the map type.
2. move the map's starting location to your own location.
3. add support for markers, which indicate single locations on a map and can include a label.

2.1 Add map types

In this step, you add an app bar with an options menu that allows the user to change the map type.

1. To create a new menu XML file, right-click on your `res` directory and select **New > Android Resource File**.
2. In the dialog, name the file `map_options`. Choose **Menu** for the resource type. Click **OK**.
3. Replace the code in the new file with the following code to create the map options. The "none" map type is omitted, because "none" results in the lack of any map at all.

```
<?xml version="1.0" encoding="utf-8"?>
<menu xmlns:android="http://schemas.android.com/apk/res/android"
      xmlns:app="http://schemas.android.com/apk/res-auto">
  <item android:id="@+id/normal_map"
        android:title="@string/normal_map"
        app:showAsAction="never"/>
  <item android:id="@+id/hybrid_map"
        android:title="@string/hybrid_map"
        app:showAsAction="never"/>
  <item android:id="@+id/satellite_map"
        android:title="@string/satellite_map"
        app:showAsAction="never"/>
  <item android:id="@+id/terrain_map"
        android:title="@string/terrain_map"
        app:showAsAction="never"/>
</menu>
```

4. Create string resources for the `title` attributes.

5. In the `MapsActivity` file, change the class to extend the `AppCompatActivity` class instead of extending the `FragmentActivity` class. Using `AppCompatActivity` will show the app bar, and therefore it will show the menu.
6. In `MapsActivity`, override the `onCreateOptionsMenu()` method and inflate the `map_options` file:

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.map_options, menu);
    return true;
}
```

7. To change the map type, use the `setMapType()` method on the `GoogleMap` object, passing in one of the map-type constants. Override the `onOptionsItemSelected()` method. Add the following code for giving different menu options:

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    // Change the map type based on the user's selection.
    switch (item.getItemId()) {
        case R.id.normal_map:
            mMap.setMapType(GoogleMap.MAP_TYPE_NORMAL);
            return true;
        case R.id.hybrid_map:
            mMap.setMapType(GoogleMap.MAP_TYPE_HYBRID);
            return true;
        case R.id.satellite_map:
            mMap.setMapType(GoogleMap.MAP_TYPE_SATELLITE);
            return true;
        case R.id.terrain_map:
            mMap.setMapType(GoogleMap.MAP_TYPE_TERRAIN);
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

8. Run the app. See how the map's appearance changes under different menu options.

2.2 Move the default map location

By default, the `onMapReady()` callback includes code that places a marker in Sydney, Australia, where Google Maps was created. The default callback also animates the map to pan to Sydney. In this step, you make the map pan to your home location without placing a marker, then zoom to a level you specify.

1. In the `onMapReady()` method, remove the code that places the marker in Sydney and moves the camera.
2. Go to www.google.com/maps in your browser and find your building.

3. Right-click on the location and select **What's here?**

Near the bottom of the screen, a small window pops up with location information, including latitude and longitude.

4. Create a new `LatLng` object called `home`. In the `LatLng` object, use the coordinates you found from Google Maps in the browser.
5. Create a `float` variable called `zoom` and set the variable to your desired initial zoom level. The following list gives you an idea of what level of detail each level of zoom shows:


- 1: World
- 5: Landmass/continent
- 10: City
- 15: Streets
- 20: Buildings

6. Create a `CameraUpdate` object using `CameraUpdateFactory.newLatLngZoom()`, passing in your `LatLng` object and zoom variable. Pan and zoom the camera by calling `moveCamera()` on the `GoogleMap` object, passing in the new `CameraUpdate` object:

```
mMap.moveCamera(CameraUpdateFactory.newLatLngZoom(home, zoom));
```

7. Run the app. The map should pan to your home and zoom into the desired level.

2.3 Add map markers

Google Maps can single out a location using a marker, which you create using the `Marker` class. The default marker uses the standard Google Maps icon: 

You can extend markers to show contextual information in [info windows](#).

In this step, you add a marker when the user touches and holds a location on the map. You then add an `InfoWindow` that displays the coordinates of the marker when the marker is tapped.

1. Create a method stub in `MapsActivity` called `setMapLongClick()` that takes a final `GoogleMap` as an argument and returns `void`:

```
private void setMapLongClick(final GoogleMap map) {}
```

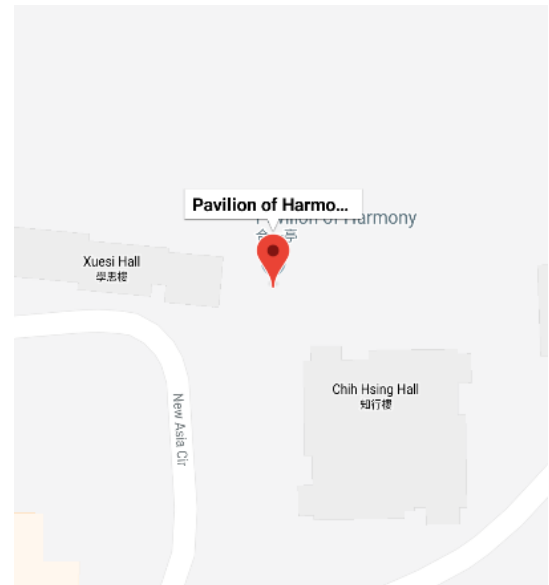
2. Use the `GoogleMap` object's `setOnMapLongClickListener()` method to place a marker where the user touches and holds. Pass in a new instance of `OnMapLongClickListener` that overrides the `onMapLongClick()` method. The incoming argument is a `LatLng` object that contains the coordinates of the location the user pressed:

```
private void setMapLongClick(final GoogleMap map) {  
    map.setOnMapLongClickListener(new GoogleMap.OnMapLongClickListener() {  
        @Override  
        public void onMapLongClick(LatLng latLng) {  
        }  
    });  
}
```

3. Inside `onMapLongClick()`, call the `addMarker()` method. Pass in a new `MarkerOptions` object with the position set to the passed-in `LatLng`:

```
map.addMarker(new MarkerOptions().position(latLng));
```

4. Call `setMapLongClick()` at the end of the `onMapReady()` method. Pass in `mMap`.
5. Run the app. Touch and hold on the map to place a marker at a location.
6. Tap the marker, which centers it on the screen.



Navigation buttons appear at the bottom-left side of the screen, allowing the user to use the Google Maps app to navigate to the marked position.



To add an info window for the marker:

1. In the `MarkerOptions` object, set the `title` field and the `snippet` field.
2. In `onMapLongClick()`, set the `title` field to "Dropped Pin" (by creating new string resource with such a resource value). Set the `snippet` field to the location coordinates inside the `addMarker()` method.

```
map.setOnMapLongClickListener(new GoogleMap.OnMapLongClickListener() {
    @Override
    public void onMapLongClick(LatLng latLng) {
        String snippet = String.format(Locale.getDefault(),
            "Lat: %1$.5f, Long: %2$.5f",
            latLng.latitude,
            latLng.longitude);

        map.addMarker(new MarkerOptions()
            .position(latLng)
            .title(getString(R.string.dropped_pin))
            .snippet(snippet));
    }
});
```

3. Run the app. Touch and hold on the map to drop a location marker. Tap the marker to show the info window.

2.4 Add POI listener (Optional)

By default, points of interest (POIs) appear on the map along with their corresponding icons. In this step, you add a `GoogleMap.OnPoiClickListener` to the map. This click-listener places a marker on the map immediately, instead of waiting for a touch & hold. The click-listener also displays the info window that contains the POI name.

1. Create a method stub in `MapsActivity` called `setPoiClick()` as follows:

```
private void setPoiClick(final GoogleMap map) {}
```

2. In the `setPoiClick()` method, set an `OnPoiClickListener` on the passed-in `GoogleMap`:

```
map.setOnPoiClickListener(new GoogleMap.OnPoiClickListener() {
    @Override
    public void onPoiClick(PointOfInterest poi) {
    }
});
```

3. In the `onPoiClick()` method, place a marker at the POI location. Set the title to the name of the POI. Save the result to a variable called `poiMarker`.

```
public void onPoiClick(PointOfInterest poi) {
    Marker poiMarker = mMap.addMarker(new MarkerOptions()
        .position(poi.latLng)
        .title(poi.name));
}
```


Call `showInfoWindow()` on `poiMarker` to immediately show the info window.

```
poiMarker.showInfoWindow();
```

4. Call `setPoiClick()` at the end of `onMapReady()`. Pass in `mMap`.
5. Run your app and find a POI such as a college canteen. Tap on the POI to place a marker on it and display the POI's name in an info window.

3. Style your map

You can customize Google Maps in many ways, giving your map a unique look and feel. In this step you customize the look and feel of the *content* of the `MapFragment`, using methods on the `GoogleMap` object. You use the online [Styling Wizard](#) to add a style to your map and customize your markers.

3.1 Style your marker

You can personalize your map further by styling the map markers. In this step, you change the default red markers to match the night mode color scheme.

1. In the `onMapLongClick()` method, add the following line of code to the `MarkerOptions()` constructor to use the default marker but change the color to blue:

```
.icon(BitmapDescriptorFactory.defaultMarker  
      (BitmapDescriptorFactory.HUE_BLUE))
```

2. Run the app. The markers you place are now shaded blue, which is more consistent with the night-mode theme of the app.

Note that POI markers are still red, because you didn't add styling to the `onPoiClick()` method.

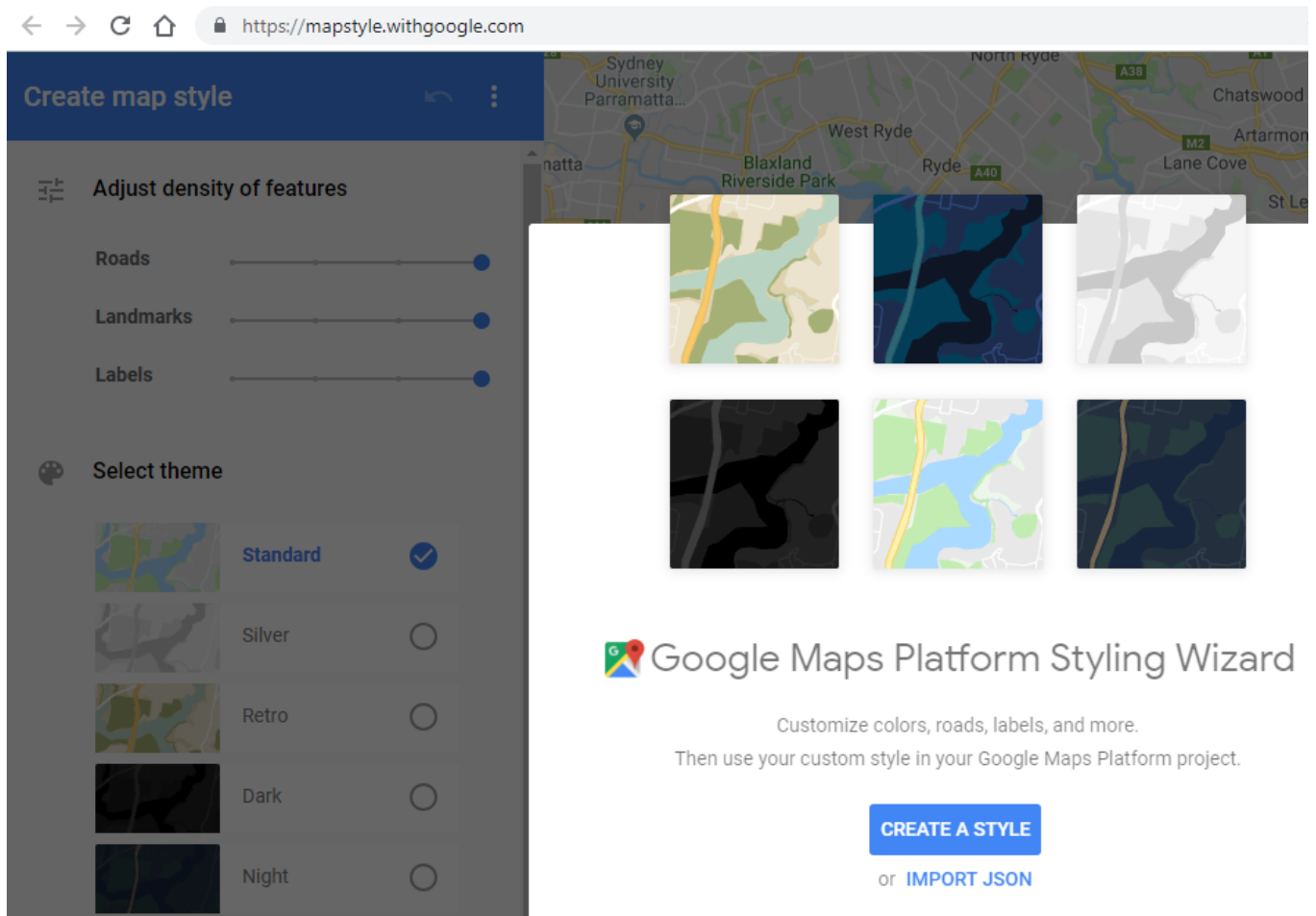
3.2 Add a style to your map (Optional)

To create a customized style for your map, you generate a JSON (JavaScript Object Notation) file that specifies how features in the map are displayed. Here is an example snippet of such a style JSON file:

```
[  
  {  
    "elementType": "geometry",  
    "stylers": [  
      {  
        "color": "#242f3e"  
      }  
    ]  
  },  
  {  
    "elementType": "labels.text.fill",  
    "stylers": [  
      {  
        "color": "#746855"  
      }  
    ]  
  },  
]
```

```
{
  "elementType": "labels.text.stroke",
  "stylers": [
    {
      "color": "#242f3e"
    }
  ]
},
```

You don't have to create this JSON file manually: Google provides the [Styling Wizard](https://mapstyle.withgoogle.com/), which generates the JSON for you after you visually style your map.



In this task, you style the map for "night mode," meaning that the map uses dim colors and low contrast for use at night.

Note: Styling only applies to maps that use the normal map type.

1. Navigate to <https://mapstyle.withgoogle.com/> in your browser.
2. Select **Create a Style**.
3. Select the **Night** theme.
4. Click **More Options** at the bottom of the menu.
5. At the bottom of the **Feature type** list, select **Water > Fill**. Change the color of the water to a dark blue (for example, #160064).
6. Click **Finish**. Copy the JSON code from the resulting pop-up window.

7. In Android Studio, create a resource directory called **raw** in the **res** directory. Create a file in **res/raw** called **map_style.json**.

8. Paste the JSON code into the new resource file.

9. To set the JSON style to the map, call **setMapStyle()** on the **GoogleMap** object. Pass in a **MapStyleOptions** object, which loads the JSON file.

The **setMapStyle()** method returns a boolean indicating the success of the styling. If the file can't be loaded, the method throws a [Resources.NotFoundException](#).

Copy the following code into the **onMapReady()** method to style the map. You may need to create a TAG string for your log statements:



```
try {
    // Customize the styling of the base map using a JSON object defined
    // in a raw resource file.
    boolean success = googleMap.setMapStyle(
        MapStyleOptions.loadRawResourceStyle(
            this, R.raw.map_style));
    if (!success) {
        Log.e(TAG, "Style parsing failed.");
    }
} catch (Resources.NotFoundException e) {
    Log.e(TAG, "Can't find style. Error: ", e);
}
```

10. Create the String constant, e.g. "SpotCU", for TAG as needed for being shown in the logcat display.

11. Run your app. The new styling should be visible when the map is in normal mode.



3.3 Enable location tracking (Optional)

To display the device location on your map without further use of Location data, you can use the [location-data layer](#). Location-data layer adds a **My Location** button  to the top-right side of the map. When the user taps the button, the map centers on the device's location. The location is shown as a blue dot  if the device is stationary, and as a blue chevron if the device is moving.

Enabling location tracking in Google Maps requires a single line of code. However, you must make sure that the user has granted location permissions (using the runtime-permission model); to request location permissions and enable the location tracking:

1. In the `AndroidManifest.xml` file, verify that the `FINE_LOCATION` permission is already present. Android Studio inserted this permission when you selected the Google Maps template.
2. To enable location tracking in your app, create a method in the `MapsActivity` called `enableMyLocation()`.
3. In the `enableMyLocation()` method, you check for the `ACCESS_FINE_LOCATION` permission. If the permission is granted, enable the location layer. Otherwise, request the permission:

```
private void enableMyLocation() {  
    if (ContextCompat.checkSelfPermission(this,  
        Manifest.permission.ACCESS_FINE_LOCATION)  
        == PackageManager.PERMISSION_GRANTED) {  
        mMap.setMyLocationEnabled(true);  
    } else {  
        ActivityCompat.requestPermissions(this, new String[]  
            {Manifest.permission.ACCESS_FINE_LOCATION},  
            REQUEST_LOCATION_PERMISSION);  
    }  
}
```

4. Call `enableMyLocation()` from the `onMapReady()` callback to enable the location layer.
5. Override the `onRequestPermissionsResult()` method. If the permission is granted, call `enableMyLocation()`:

```
@Override  
public void onRequestPermissionsResult(int requestCode,  
    @NonNull String[] permissions,  
    @NonNull int[] grantResults) {  
    // Check if location permissions are granted and if so enable the  
    // location data layer.  
    switch (requestCode) {  
        case REQUEST_LOCATION_PERMISSION:  
            if (grantResults.length > 0  
                && grantResults[0]  
                == PackageManager.PERMISSION_GRANTED) {  
                enableMyLocation();  
                break;  
            }  
    }  
}
```

```
}
```

6. Create a constant

```
private static final int REQUEST_LOCATION_PERMISSION = 1;
```

7. Run the app. The top-right corner now contains the **My Location** button, which displays the device's current location.

Note: When you run the app on an emulator, the location may **not** be available. If you haven't used the emulator settings to set a location, the location button will be unavailable.

References

- [Getting Started with the Google Maps Android API](#)
- [Adding a Map with a Marker](#)
- [Map Objects](#)
- [Adding a Styled Map](#)
- [GoogleMap](#)
- [SupportMapFragment](#)