# Maps in Android

Displaying spatial data with the Google Maps SDK

# Google Maps SDK — What You Get

One dependency, one API key, and you get an interactive map with built-in gestures and rendering.

**Map Tiles**
Satellite, terrain, and road maps with pinch-to-zoom

**Markers**
Colored pins with titles, snippets, and info windows

**Camera**
Set position, zoom, and tilt in code

**My Location**
Blue dot showing the user's GPS position

The SDK handles rendering and interaction. You provide the data.

# Setup — API Key & Manifest

Two things to configure: one Gradle dependency and one API key in the manifest.

**AndroidManifest.xml**

```
<application ...>
    <meta-data
        android:name=
          "com.google.android.geo.API_KEY"
        android:value=
          "@string/google_maps_key" />
    ...
</application>
```

**build.gradle (app)**

```
dependencies {
    // Google Maps
    implementation
      'com.google.android.gms:'
      + 'play-services-maps:20.0.0'
}
```

Cloud Console UI and key-creation steps change frequently.
Follow Google's current guide: developers.google.com/maps/documentation/android-sdk/start

# Debug vs. Release Signing & SHA-1

Google ties your API key to your app's signing certificate.
Debug and release use different certificates.

**Debug Keystore**
Auto-generated by
Android Studio at
~/.android/debug.keystore

Same key on one machine,
different across machines

**Release Keystore**
You create with keytool
and guard it carefully

Used for Play Store
and production API keys

SHA-1 fingerprint = hash of the signing certificate (not your code).
Register it in the Cloud Console so only YOUR app can use YOUR key.

```
./gradlew signingReport
```

Debug and release have different SHA-1 fingerprints — register both.

# SupportMapFragment

The map lives in a Fragment declared in XML.
Your Activity implements OnMapReadyCallback.

**content_main.xml**

```
<fragment
  android:name="com.google.android
    .gms.maps.SupportMapFragment"
  android:id="@+id/mapFrag"
  android:layout_width="match_parent"
  android:layout_height="match_parent"/>
```

**MainActivity.kt**

```
class MainActivity : AppCompatActivity(),
    OnMapReadyCallback {
  private lateinit var map: GoogleMap

  override fun onCreate(...) {
    val mapFrag = supportFragmentManager
      .findFragmentById(R.id.mapFrag)
        as SupportMapFragment
    mapFrag.getMapAsync(this)
  }
  override fun onMapReady(gMap: GoogleMap) {
    map = gMap
  }
}
```

getMapAsync is the handshake — the map initializes asynchronously, then calls onMapReady.

# OnMapReady & Camera

Once the map is ready, set the initial camera position.
LatLng + zoom level define what the user sees.

**FloodWatch — MainActivity.kt**

```
override fun onMapReady(googleMap: GoogleMap) {
    map = googleMap

    // Center on Austin at a zoom level showing the metro area
    val austin = LatLng(30.2672, -97.7431)
    map.moveCamera(CameraUpdateFactory.newLatLngZoom(austin, 11.0f))

    // Set up listeners, observe data...
}
```

Zoom levels
1 = world
5 = continent
10 = city
15 = street
20 = building

moveCamera jumps instantly; animateCamera gives smooth transitions.

# Markers

A marker is a pin on the map. Set its position, title, snippet, and color.

**FloodWatch — adding a colored marker**

```
val color = when (station.level) {
    FlowLevel.NORMAL   -> BitmapDescriptorFactory.HUE_GREEN
    FlowLevel.ELEVATED -> BitmapDescriptorFactory.HUE_YELLOW
    FlowLevel.HIGH     -> BitmapDescriptorFactory.HUE_RED
}
map.addMarker(
    MarkerOptions()
        .position(LatLng(station.latitude, station.longitude))
        .title(station.siteName)
        .snippet("Flow: 142.0 ft³/s  Gage: 5.20 ft")
        .icon(BitmapDescriptorFactory.defaultMarker(color))
)
```

.title() and .snippet() appear in the default info window automatically

BitmapDescriptorFactory provides HUE_GREEN, HUE_YELLOW, HUE_RED, etc.

# Click Listeners & Street View

Respond to taps on markers or the map itself.
The Street View Static API returns a photo for any lat/lng.

## Marker & map click listeners

```
map.setOnMarkerClickListener { marker ->
    showStreetView(marker.position,
        marker.title ?: "")
    false  // don't suppress info window
}

map.setOnMapClickListener {
    binding.streetViewCard
        .visibility = View.GONE
}
```

## Street View Static API URL

```
val url = "https://maps.googleapis.com"
    + "/maps/api/streetview"
    + "?size=600x300"
    + "&location=$lat,$lng"
    + "&key=$apiKey"

// Same API key, just enable
// "Street View Static API"
// in the Cloud Console
Glide.with(this).load(url)
    .centerCrop()
    .into(binding.streetViewIV)
```

Info window = the popup bubble
showing .title() and .snippet()
that appears when you tap a marker

Return false → your code runs
AND the info window still shows
Return true → info window suppressed

# Location Permissions

The my-location layer requires runtime permission.
Request first, then enable on the map.

**Request permission (onCreate)**

```
val permReq = registerForActivityResult(
  ActivityResultContracts
    .RequestMultiplePermissions()
) { perms ->
  if (perms.getOrDefault(
      ACCESS_FINE_LOCATION, false))
    locationPermissionGranted = true
}
permReq.launch(
  arrayOf(ACCESS_FINE_LOCATION))
```

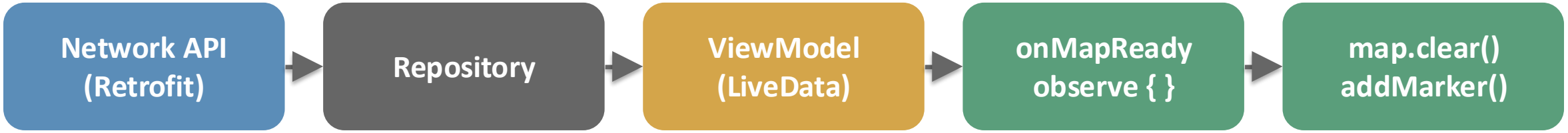
**Enable on the map (onMapReady)**

```
if (locationPermissionGranted
    && checkSelfPermission(
        ACCESS_FINE_LOCATION)
      == PERMISSION_GRANTED) {
  map.isMyLocationEnabled = true
  map.uiSettings
    .isMyLocationButtonEnabled = true
}
```

Location is opt-in: request at runtime, enable on the map, guard with a permission check.

# Architecture — ViewModel + Map

The map is just another observer of your ViewModel's LiveData.
Same architecture as RecyclerView-based screens.




**RecyclerView screen**

```
viewModel.items.observe(this) {
    adapter.submitList(it)
}
```

**Map screen**

```
viewModel.stations.observe(this) {
    map.clear()
    for (s in it) map.addMarker(...)
}
```

The map is a view, not a data source. ViewModel owns the data, map just renders it.