ChatGPT

Google Fit API Integration Guide for Android (Kotlin & Java)

Integrating Google Fit into your Android app enables you to read and use health and fitness data (steps, heart rate, body measurements, etc.) collected from the user’s device or other fitness apps. Google Fit is provided as part of Google Play services and uses OAuth 2.0 for user authorization. *(Note: Google is deprecating the Google Fit APIs in 2026, with new API sign-ups disabled as of May 2024. Existing integrations will continue to work for now, but new projects are encouraged to migrate to Health Connect in the future.)*

In this guide, we'll cover:

- **Setup & Integration:** Project setup, dependencies, and enabling Google Fit in your app (with **Kotlin** and **Java** examples).
- **Authentication & Permissions:** Handling user sign-in and requesting the appropriate Google Fit OAuth permissions.
- **Reading Fitness Data:** How to access **Activity** data (e.g. step count), **Heart Rate** data, and **Body** data (weight, height) from Google Fit.
- **Data Formats:** Understanding the structure of data returned by Google Fit (DataSets, DataPoints, fields).
- **Using the Data:** Ideas for visualizing and providing feedback with the retrieved data in your app.
- **Common Issues & Solutions:** Troubleshooting tips for common pitfalls (permission issues, empty data, etc).

Let's get started with the setup and integration steps.

Setup and Integration

1. Enable Google Fit API and Obtain Credentials

Before writing any code, perform these one-time setup tasks:

- **Google API Console:** Enable the **Google Fit API** for your project and create an **OAuth 2.0 Client ID** for Android. This typically involves specifying your app’s package name and SHA-1 signing certificate in Google Cloud Console. The resulting OAuth Client ID will be used by Google Play services to authenticate your app.

- **Google Account:** Make sure you have a Google account for testing Google Fit. You can use your personal account or create a test account. Google Fit data is tied to the user’s Google account.

- **Google Play Services on Device:** Ensure the test device (or emulator) has up-to-date Google Play services, since Google Fit functionality is delivered via Google Play services on Android.

2. Add Gradle Dependencies

In your app module’s `build.gradle` (or Gradle KTS) file, add the Google Play services Fitness and Auth libraries. These libraries provide the Google Fit APIs and authentication support:

```
dependencies {
    implementation 'com.google.android.gms:play-services-fitness:21.3.0'
    implementation 'com.google.android.gms:play-services-auth:21.4.0'
}
```

(If using Kotlin DSL for Gradle, the syntax is similar but in Kotlin style.) This ensures Gradle will download the Google Fit SDK when building your app.

3. Create a Google Fit API Client (GoogleSignIn Integration)

Google Fit on Android uses the user's Google account for authentication. Instead of managing raw OAuth tokens yourself, you’ll use the Google Sign-In API to request the specific Google Fit scopes your app needs. The general process is:

a. Build FitnessOptions: Specify which data types and access types (read and/or write) your app will use. This is done with a `FitnessOptions` object. For example, to read the user's step count, heart rate, weight, and height, you add those data types with read access:

Kotlin – Building a `FitnessOptions` with required data types:

```kotlin
val fitnessOptions = FitnessOptions.builder()
    .addDataType(DataType.TYPE_STEP_COUNT_DELTA, FitnessOptions.ACCESS_READ)
    .addDataType(DataType.TYPE_HEART_RATE_BPM, FitnessOptions.ACCESS_READ)
    .addDataType(DataType.TYPE_WEIGHT, FitnessOptions.ACCESS_READ)
    .addDataType(DataType.TYPE_HEIGHT, FitnessOptions.ACCESS_READ)
    .build()
```

Java – Equivalent `FitnessOptions` building:

```java
FitnessOptions fitnessOptions = FitnessOptions.builder()
    .addDataType(DataType.TYPE_STEP_COUNT_DELTA, FitnessOptions.ACCESS_READ)
    .addDataType(DataType.TYPE_HEART_RATE_BPM, FitnessOptions.ACCESS_READ)
    .addDataType(DataType.TYPE_WEIGHT, FitnessOptions.ACCESS_READ)
    .addDataType(DataType.TYPE_HEIGHT, FitnessOptions.ACCESS_READ)
    .build();
```

Each data type corresponds to an OAuth scope under the hood. For example, `TYPE_STEP_COUNT_DELTA` is under the "fitness.activity.read" scope, heart rate is under "fitness.heart_rate.read", and weight/height are under "fitness.body.read". All these are **restricted scopes**, meaning you will eventually need to get your app verified by Google if you publish it (more on this later). For now, just request what you need – **do not request more scopes than necessary**, as users are more likely to grant access to clearly justified, minimal scopes.

b. Get the GoogleSignIn Account: Use Google Sign-In to get a `GoogleSignInAccount` that has the FitnessOptions. You do not need to present an email/password login UI; GoogleSignIn will handle the account picker if the user is not already signed in on the device.

```
val account = GoogleSignIn.getAccountForExtension(this, fitnessOptions)
```

This retrieves the current signed-in Google account on the device (if any) and prepares it with the fitness options (scopes) we defined.

c. Request Permissions if Not Granted: Before accessing Google Fit data, you must ensure the user has granted the required fitness permissions. Use `GoogleSignIn.hasPermissions` to check if the `GoogleSignInAccount` already has the Fit scopes, and if not, call `GoogleSignIn.requestPermissions` to trigger the OAuth consent screen. For example:

```
if (!GoogleSignIn.hasPermissions(account, fitnessOptions)) {
    // Request the user to grant fitness permissions
    GoogleSignIn.requestPermissions(
        this,
        GOOGLE_FIT_PERMISSIONS_REQUEST_CODE,
        account,
        fitnessOptions
    )
} else {
    // Permissions already granted, proceed to access data
    accessGoogleFitData()
}
```

In Java, the logic is similar: use `GoogleSignIn.hasPermissions(account, fitnessOptions)` and call `GoogleSignIn.requestPermissions(activity, REQUEST_CODE, account, fitnessOptions)` if needed.

d. Handle the Authorization Result: When the user responds to the permission prompt, you'll get a callback to your Activity's `onActivityResult` (or in newer APIs, use the Activity Result API). If `resultCode == Activity.RESULT_OK` and the requestCode matches your `GOOGLE_FIT_PERMISSIONS_REQUEST_CODE`, then the user granted access and you can proceed to read data. If the result is not OK, the user denied access – you should handle this (perhaps by explaining why the permission is needed and prompting again or disabling related features).

```kotlin
override fun onActivityResult(requestCode: Int, resultCode: Int, data: Intent?)
{
    super.onActivityResult(requestCode, resultCode, data)
    if (requestCode == GOOGLE_FIT_PERMISSIONS_REQUEST_CODE) {
        if (resultCode == Activity.RESULT_OK) {
            accessGoogleFitData()  // user granted permission
        } else {
            // Permission was denied
            Log.w(TAG, "Google Fit permissions were NOT granted.")
        }
    }
}
```

e. Summary: At this point, if permissions are granted, you have a signed-in account with the appropriate scopes. You can now create Google Fit API clients (History, Sensors, Sessions, etc.) to access data. For most read operations, we will use the **History API** via the `HistoryClient` (for reading historical or aggregate data) and occasionally the **Sensors API** or **Recording API** for real-time data or background subscriptions.

4. Android Runtime Permissions

In addition to Google Fit's OAuth scopes, certain data types also require Android runtime permissions due to their sensitivity:

- **Activity Recognition Permission:** To record or read user activity/steps in the background, Android 10+ requires the `ACTIVITY_RECOGNITION` permission. Google Fit step count data (`TYPE_STEP_COUNT_DELTA`) is considered an *activity* data type requiring this permission for recording. If your app is actively recording steps (via the Recording API), request this permission in your manifest and at runtime.

- **Body Sensors Permission:** For real-time heart rate from on-device sensors (like a Wear OS watch or phone sensor), the `BODY_SENSORS` permission is required on Android 6.0+ when recording heart rate data. If you are only reading historical heart rate data that was already recorded by another app or device, you typically don't need to request BODY_SENSORS; however, if your app itself uses sensors to measure heart rate, you must have this permission in the manifest and request it at runtime.

Make sure to declare these permissions in the AndroidManifest.xml and use `ActivityCompat.requestPermissions` to request them from the user when needed (usually right before you start recording data). For example, before subscribing to step or heart data, check `ContextCompat.checkSelfPermission` for the required permission.

With setup and authentication in place, we can now access Google Fit data.

Accessing and Reading Google Fit Data

Google Fit organizes data into data types (steps, heart rate, weight, etc.), delivered as **data points** with timestamps and values. You will typically use the **History API** to read data. Below, we provide examples for reading **Activity data (steps)**, **Heart Rate data**, and **Body data (weight & height)**. Code examples are given in both Kotlin and Java.

1. Reading Step Count (Activity Data)

Use case: Retrieve the user's step count. This can be done as a daily total or a time-series of steps over a range. Google Fit offers an easy way to get the current day's step total.

Kotlin – Daily Step Total Example:

```
// Assume fitnessOptions and account are already obtained as above
Fitness.getHistoryClient(this, GoogleSignIn.getAccountForExtension(this,
fitnessOptions))
    .readDailyTotal(DataType.TYPE_STEP_COUNT_DELTA)
    .addOnSuccessListener { result ->
        val totalSteps = result.dataPoints.firstOrNull()
            ?.getValue(Field.FIELD_STEPS)?.asInt() ?: 0
        Log.i(TAG, "Total steps today: $totalSteps")
        // TODO: use totalSteps in UI (e.g., update a steps counter)
    }
    .addOnFailureListener { e ->
        Log.e(TAG, "There was a problem getting step count.", e)
    }
```

In this snippet, `readDailyTotal()` directly returns the total steps recorded for **today** (since midnight). We extract the first data point's `Field.FIELD_STEPS` value, which is the step count. If there are no data points (meaning no steps recorded yet today), we default to 0.

Java – Daily Step Total Example:

```
Fitness.getHistoryClient(this, GoogleSignIn.getAccountForExtension(this,
fitnessOptions))
    .readDailyTotal(DataType.TYPE_STEP_COUNT_DELTA)
    .addOnSuccessListener(result -> {
        int totalSteps = 0;
        if (!result.getDataPoints().isEmpty()) {
            totalSteps = result.getDataPoints().get(0)
                             .getValue(Field.FIELD_STEPS).asInt();
        }
        Log.i(TAG, "Total steps today: " + totalSteps);
        // TODO: use totalSteps (e.g., display in UI)
    })
    .addOnFailureListener(e -> {
        Log.e(TAG, "There was a problem getting step count.", e);
    });
```

This achieves the same result using Java: we check if the result has at least one DataPoint and then retrieve the steps value.

Custom Time Range for Steps: If you want more than today's total – for instance, a week's worth of daily step counts – you can use a `DataReadRequest`. For example, to get daily step totals for the past week, you can aggregate step count data by day:

```
// Define time range: start = 7 days ago, end = now
val endTime = LocalDateTime.now().atZone(ZoneId.systemDefault())
val startTime = endTime.minusWeeks(1)
val readRequest = DataReadRequest.Builder()
    .aggregate(DataType.AGGREGATE_STEP_COUNT_DELTA)  // aggregate daily steps
    .bucketByTime(1, TimeUnit.DAYS)                  // bucket into 1-day
intervals
    .setTimeRange(startTime.toEpochSecond(), endTime.toEpochSecond(),
TimeUnit.SECONDS)
    .build()

Fitness.getHistoryClient(this, GoogleSignIn.getAccountForExtension(this,
fitnessOptions))
    .readData(readRequest)
    .addOnSuccessListener { response ->
        // response contains buckets of one day each
        for (bucket in response.buckets) {
            for (dataSet in bucket.dataSets) {
                // Each dataSet in a bucket will have the aggregated step count
for that day
                val stepDp = dataSet.dataPoints.firstOrNull()
                val dayCount = stepDp?.getValue(Field.FIELD_STEPS)?.asInt() ?: 0
                Log.i(TAG, "Steps on ${stepDp?.getStartTimeString()}:
$dayCount")
            }
        }
    }
    .addOnFailureListener { e ->
        Log.w(TAG, "Error reading weekly steps.", e)
    }
```

The above uses `aggregate(DataType.AGGREGATE_STEP_COUNT_DELTA)` which tells Google Fit to return an aggregated daily total for each day in the range. Each bucket corresponds to one day, and contains a data set with a single aggregated DataPoint for steps. We used helper functions `getStartTimeString()` (as in the official docs) to format timestamp; you can also convert timestamps manually. This approach is efficient since the aggregation is done by Google Fit (less data transfer and no need to sum up individual readings).

Understanding Step Data: Google Fit step counts are typically recorded as delta values (steps taken over an interval). A DataPoint of `TYPE_STEP_COUNT_DELTA` has a start time, end time, and an integer field for "steps" (Field.FIELD_STEPS) representing steps during that interval. The daily total we fetched is essentially an aggregate of all those deltas for the day. If you prefer raw data (perhaps to see intraday step distribution), you could request `DataType.TYPE_STEP_COUNT_DELTA` with a time range (using `read()` instead of `aggregate()`) and you'll get many DataPoints each covering a smaller time interval (e.g., step count every few minutes or hours, depending on how the data was recorded).

2. Reading Heart Rate Data

Use case: Retrieve the user's heart rate measurements. Heart rate is part of Google Fit's **Health Data** category and represents instantaneous beats per minute (BPM) readings. Each heart rate DataPoint typically has a timestamp (when the measurement was taken) and one field for BPM value.

To read heart rate data, you will often specify a time range of interest (for example, the last hour, last day, or a specific workout session). Unlike steps, heart rate data isn't usually aggregated into a single daily value (it fluctuates throughout the day), so you will get a series of data points.

Example: Read all heart rate data from the last 24 hours.

```
val endTime = System.currentTimeMillis()
val startTime = endTime - TimeUnit.DAYS.toMillis(1)  // 24 hours ago

val readRequest = DataReadRequest.Builder()
    .read(DataType.TYPE_HEART_RATE_BPM)  // request raw heart rate data points
    .setTimeRange(startTime, endTime, TimeUnit.MILLISECONDS)
    .build()

Fitness.getHistoryClient(this, GoogleSignIn.getAccountForExtension(this,
fitnessOptions))
    .readData(readRequest)
    .addOnSuccessListener { response ->
        // The response may contain one or more DataSet(s) for heart rate
        for (dataSet in response.dataSets) {
            for (dp in dataSet.dataPoints) {
                val timestamp = dp.getTimestamp(TimeUnit.MILLISECONDS)
                val bpm = dp.getValue(Field.FIELD_BPM).asFloat()
                Log.i(TAG, "Heart rate at ${Date(timestamp)} = $bpm BPM")
                // TODO: use the heart rate value (e.g., plot it on a graph)
            }
        }
    }
    .addOnFailureListener { e ->
        Log.e(TAG, "Error reading heart rate data", e)
    }
```

In this snippet, we use `DataReadRequest.Builder().read()` instead of `aggregate`, because heart rate is an instantaneous measurement and we likely want the individual readings. We loop through the returned DataSets and DataPoints to extract each heart rate value. The field for heart rate DataPoints is `Field.FIELD_BPM` (a float). Each DataPoint's timestamp can be obtained via `dp.getTimestamp()` or `dp.getEndTime()` (since for instantaneous readings, the end time is used as the timestamp).

Important: Heart rate data in Google Fit might come from a wearable (Wear OS watch), a chest strap, or even manually entered. If no heart rate sensor is providing data, you might simply get an empty result. There is also an aggregate data type `AGGREGATE_HEART_RATE_SUMMARY` that can give average/max/min over a time bucket – for example, average heart rate per hour – but using raw readings as shown above is straightforward for most needs.

Java – Heart Rate (24h) Example:

```
long endTime = System.currentTimeMillis();
long startTime = endTime - TimeUnit.DAYS.toMillis(1);

DataReadRequest readRequest = new DataReadRequest.Builder()
        .read(DataType.TYPE_HEART_RATE_BPM)
        .setTimeRange(startTime, endTime, TimeUnit.MILLISECONDS)
        .build();

Fitness.getHistoryClient(this, GoogleSignIn.getAccountForExtension(this,
fitnessOptions))
    .readData(readRequest)
    .addOnSuccessListener(response -> {
        for (DataSet dataSet : response.getDataSets()) {
            for (DataPoint dp : dataSet.getDataPoints()) {
                long timestamp = dp.getTimestamp(TimeUnit.MILLISECONDS);
                float bpm = dp.getValue(Field.FIELD_BPM).asFloat();
                Log.i(TAG, "Heart rate at " + new Date(timestamp) + " = " +
bpm);
            }
        }
    })
    .addOnFailureListener(e -> Log.e(TAG, "Error reading heart rate data", e));
```

Note on Heart Rate Sensors: If you want continuous live heart rate monitoring within your app, you might use the **Sensors API** to listen for real-time heart rate updates from a wearable. That involves `SensorsClient.findDataSources` and `SensorsClient.add()` listeners, and requires the device to have a sensor or a connected wearable. In many cases, however, simply reading the history (possibly after the fact) is sufficient to get the user's heart rate during workouts or throughout the day.

Also note that if *your app* is recording heart rate from a sensor in real-time, you must request the `BODY_SENSORS` permission as mentioned earlier. For reading historical data collected by Google Fit or other apps, that may not be necessary.

3. Reading Body Data (Weight and Height)

Use case: Access the user's body measurements such as weight and height. These are part of Fit's *Body* data types and are considered sensitive (under the "fitness.body" scope). Typically, weight and height are not continuously changing data – they're recorded when the user updates them (e.g., weighing themselves or setting profile info).

Google Fit represents weight and height as instantaneous readings (a single timestamp with a value). The latest value is usually what you want (e.g., the most recent weight entry).

Reading Weight:

Weight is measured in **kilograms** in Google Fit. The data type is `DataType.TYPE_WEIGHT`, with a float field `Field.FIELD_WEIGHT` (in kg). To get the user's current weight, a common approach is to query for weight data and take the latest entry.

For example, to fetch the latest weight:

```
val endTime = System.currentTimeMillis()
val startTime = 0L  // from epoch start to now
val readRequest = DataReadRequest.Builder()
    .read(DataType.TYPE_WEIGHT)
    .setTimeRange(startTime, endTime, TimeUnit.MILLISECONDS)
    .setLimit(1)                        // get at most 1 data point (the most
recent)
    .build()

Fitness.getHistoryClient(this, GoogleSignIn.getAccountForExtension(this,
fitnessOptions))
    .readData(readRequest)
    .addOnSuccessListener { response ->
        var latestWeightKg: Float? = null
        // The response for a .read on a single data type will have at most one
DataSet
        response.dataSets.forEach { dataSet ->
            dataSet.dataPoints.firstOrNull()?.let { dp ->
```

```
                latestWeightKg = dp.getValue(Field.FIELD_WEIGHT).asFloat()
                val weightTime = dp.getTimestamp(TimeUnit.MILLISECONDS)
                Log.i(TAG, "Latest weight: $latestWeightKg kg (recorded at $
{Date(weightTime)})")
            }
        }
        if (latestWeightKg == null) {
            Log.i(TAG, "No weight data available.")
        }
    }
```

Here we set a very broad time range (from 0 to now) and `setLimit(1)`. By default, the History API returns the **most recent** data points first when a limit is applied, so this should give us the latest weight entry (if any) [1]. We then read the `Field.FIELD_WEIGHT` value. If the user never logged weight, we get none.

Reading Height:

Height is measured in **meters** in Google Fit's data (DataType.TYPE_HEIGHT, field `Field.FIELD_HEIGHT`). You can use a similar approach as weight. Height is often set once (user's profile) and doesn't change often. For example:

```
val readRequest = DataReadRequest.Builder()
    .read(DataType.TYPE_HEIGHT)
    .setTimeRange(0, System.currentTimeMillis(), TimeUnit.MILLISECONDS)
    .setLimit(1)
    .build()

Fitness.getHistoryClient(this, GoogleSignIn.getAccountForExtension(this,
fitnessOptions))
    .readData(readRequest)
    .addOnSuccessListener { response ->
        val heightMeters = response.dataSets.firstOrNull()
            ?.dataPoints?.firstOrNull()
            ?.getValue(Field.FIELD_HEIGHT)?.asFloat()
        if (heightMeters != null) {
            Log.i(TAG, "User height: $heightMeters m")
        } else {
            Log.i(TAG, "No height data available.")
        }
    }
```

This fetches the latest height. You might convert meters to centimeters or feet/inches for display as needed. For example, 1.75 m = 175 cm.

Java – Weight example (latest entry):

```
DataReadRequest readRequest = new DataReadRequest.Builder()
    .read(DataType.TYPE_WEIGHT)
    .setTimeRange(0, System.currentTimeMillis(), TimeUnit.MILLISECONDS)
    .setLimit(1)
    .build();

Fitness.getHistoryClient(this, GoogleSignIn.getAccountForExtension(this,
fitnessOptions))
    .readData(readRequest)
    .addOnSuccessListener(response -> {
        Float latestWeightKg = null;
        for (DataSet dataSet : response.getDataSets()) {
            for (DataPoint dp : dataSet.getDataPoints()) {
                latestWeightKg = dp.getValue(Field.FIELD_WEIGHT).asFloat();
                long time = dp.getTimestamp(TimeUnit.MILLISECONDS);
                Log.i(TAG, "Latest weight: " + latestWeightKg + " kg at " + new
Date(time));
            }
        }
        if (latestWeightKg == null) {
            Log.i(TAG, "No weight data available.");
        }
    });
```

This code checks each DataPoint (though with `setLimit(1)`, there will be at most one).

Note: Weight and height data are considered **sensitive**. If you plan to use them in a published app, Google will require your OAuth consent screen to be **verified** and comply with their data policies. During development/testing (with limited users), you can use them, but expect an "unverified app" warning if not verified.

4. Data Structure and Format in Google Fit

When you read data from Google Fit, it’s important to understand how the data is structured:

- **DataPoint:** The atomic unit of data. Each `DataPoint` represents a single measurement or metric. It contains: a data type (e.g. *step count delta*, *heart rate bpm*), a timestamp (or start and end time interval), and one or more fields with values. For example:
- A step count DataPoint of type `TYPE_STEP_COUNT_DELTA` might have: start time = 10:00, end time = 10:10, and field `steps = 100` (meaning 100 steps were taken in that 10-minute window).
- A heart rate DataPoint of type `TYPE_HEART_RATE_BPM` might have: timestamp = 10:05, field `bpm = 80` (meaning 80 beats per minute at that moment).

- A weight DataPoint of type `TYPE_WEIGHT` might have: timestamp = Jan 1, 2025, field `weight = 70.0` (kg).

- **DataSet:** A collection of DataPoints of the *same type*. For example, a DataSet could hold all step count DataPoints between two dates, or all heart rate readings from a session. When you call `HistoryClient.readData()` with a `DataReadRequest`, the response contains one or more DataSet objects (one for each data type requested). If you requested multiple data types in one go, you might get multiple DataSets in the result.

- **Bucket:** Used when you aggregate data by time or by activity. The `DataReadResponse` can organize DataSets into **buckets** (e.g., one bucket per day). Each bucket then contains DataSet(s). In our step aggregation example above, we got `response.buckets`, each with a DataSet of steps for that day.

Iterating through data: Typically, after `readData`, you handle the response by iterating through buckets (if used) and then through data sets and data points. For example (Kotlin pseudo-code):

```
for (bucket in response.buckets) {
    for (dataSet in bucket.dataSets) {
        Log.i(TAG, "DataSet for type: ${dataSet.dataType.name}")
        for (dp in dataSet.dataPoints) {
            Log.i(TAG, "Data point:")
            Log.i(TAG, "\tType: ${dp.dataType.name}")
            Log.i(TAG, "\tStart: ${dp.getStartTime(TimeUnit.SECONDS)}")
            Log.i(TAG, "\tEnd: ${dp.getEndTime(TimeUnit.SECONDS)}")
            // Log each field value for the data point
            for (field in dp.dataType.fields) {
                Log.i(TAG, "\tField: ${field.name} Value: $
{dp.getValue(field)}")
            }
        }
    }
}
```

This would output lines describing each data point (type, time, fields). In Java, the nested loops are analogous. Understanding this structure helps you extract the data you need. Usually, for a given query, you know what type you asked for and thus what fields to expect, so you can directly access those fields. For instance, if you query `TYPE_STEP_COUNT_DELTA`, you know each DataPoint has a `Field.FIELD_STEPS`. If you query `TYPE_WEIGHT`, each DataPoint has `Field.FIELD_WEIGHT`, etc. For aggregate types, fields could be `Field.FIELD_AVERAGE`, `FIELD_MAX`, `FIELD_MIN` (e.g., in heart rate summary).

Units: Always note the units of the fields (Google Fit uses metric units by default): - Step count: just an integer count (no unit). - Heart rate: float, in *beats per minute*. - Weight: float, in *kilograms*. - Height: float, in *meters*. - Calories: float, in *kilocalories*. - Distance: float, in *meters*. - etc.

You may convert these for display (e.g., kg to lbs, meters to feet) based on user preference.

Utilizing Google Fit Data in Your App

Raw data is only as useful as the insights or features you build on top of it. Here are some ideas and best practices for using the Google Fit data in a user-friendly way:

- **Visualize Progress with Charts:** Plot the user's step counts over time (daily, weekly, monthly) using a line chart or bar chart. For example, show a bar graph of the past 7 days of steps to highlight trends or consistency. Similarly, graph heart rate over a workout duration, or weight change over months. Visual cues help users see their progress or patterns at a glance.

- **Set Goals and Provide Feedback:** Leverage the Google Fit data to help users set goals (if they haven't already via Google Fit). For instance, if the user's daily step goal is 10,000 steps, show a progress ring or percentage of goal completed for the day. Encourage them when they hit milestones (Google Fit uses "Heart Points" and "Move Minutes" as well – you can retrieve those too if needed). If using heart rate, you could detect when the user reaches certain heart rate zones and provide feedback (common in fitness apps during exercise).

- **Insights and Health Tips:** Use the data to derive simple insights. For example: "You walked 2,000 steps more today than yesterday, great job!" or "Your resting heart rate this week is slightly above last week's – consider some relaxation exercises." Be careful with health interpretations (don't act as a medical app unless you are one), but basic comparative insights can increase user engagement.

- **Personalize the User Experience:** If you know the user's height/weight (and thus can calculate BMI), you might tailor some content or recommendations. For example, if an app provides workout or diet tips, knowing the user's body metrics can help personalize that content. **Privacy note:** Always handle such sensitive data carefully and with user consent, per Google's policies.

- **Combine Data Types:** More advanced uses could combine multiple data streams. For instance, during a run (if you have location/route and heart rate), you can show how heart rate changed along the route. Or correlate step counts with active minutes or calories expended (Google Fit can provide calories estimates if weight is known).

- **Sync with Google Fit App:** If the user also uses the official Google Fit app, try to keep your data in sync. Google Fit's cloud will typically sync data from multiple apps. For example, if your app records a workout session (perhaps via the Sessions API), it will show up in Google Fit app, and vice versa. This gives users a seamless experience – they can choose where to view their data. Google Fit's Sessions and Goals APIs could be explored if you want to show detailed workout info or the user's Fit goals.

- **User Control:** Provide users the option to disconnect Google Fit or choose which data to share. Not every user will be comfortable sharing all data. Google Fit scopes are granular (e.g., a user might grant step count but not heart rate). Your app should handle gracefully if certain data is not accessible (for example, if the user denied heart rate permission, maybe disable heart-related features or prompt again with justification).

Common Issues and Troubleshooting

Working with Google Fit can involve some trial and error. Here are common issues developers encounter and how to address them:

- **Permission Denied or Missing Scopes:** If your calls to Google Fit return an error or empty data, the first thing to check is that the OAuth permission was granted for the data type you want. For example, if `GoogleSignIn.hasPermissions` is false or you get a `SecurityException`, the user likely did not approve the required scope. **Solution:** Re-initiate the permission request, possibly with a clearer explanation to the user of why the data is needed. Only request the scope at the point it's needed (incremental authorization) to improve the chance the user understands the context. Also verify that you added the data type to FitnessOptions (it's easy to forget to add one of them).

- **Android Permission Not Granted:** Similar to above, if you attempt to record sensor data and nothing happens, ensure you have the runtime permission. For example, if step data isn't coming through on Android 10+, confirm `ACTIVITY_RECOGNITION` permission is granted. If you try to register a Sensors listener for heart rate and get no updates, check `BODY_SENSORS` permission. **Solution:** Request the permission via `ActivityCompat.requestPermissions` and handle the user's response. You can also check for these at app startup and explain why they are needed (e.g., "Allow access to physical activity to track your steps").

- **No Data / Empty DataSets:** It's possible to execute a read request correctly and still get an empty result. Common reasons:

- The user has no recorded data for that type or time range. For example, asking for heart rate data when the user doesn't wear a heart-rate sensor will return nothing.
- Data exists in Google Fit cloud but hasn't synced to the device yet (if the user recently used another device). The Fit SDK usually uses local cached data for performance. It syncs periodically (every few hours). If you suspect this, you can force server sync by using `enableServerQueries()` on the DataReadRequest [1], which tells Google Fit to fetch from server if not present locally.
- If you are recording data via the Recording API, remember that it only starts collecting from the moment you subscribe. You won't get historical data prior to subscription. For instance, if you subscribe to step count now, you won't get yesterday's steps (unless another app already recorded them).

Solutions: - Ensure data is being recorded: For steps and other ongoing data, you can use the **Recording API** to subscribe your app to those data types so that Google Fit will continuously record them in the background. For example, after obtaining permissions, call:

```
Fitness.getRecordingClient(this, account)
    .subscribe(DataType.TYPE_STEP_COUNT_DELTA)
    .addOnSuccessListener(aVoid -> Log.i(TAG, "Subscribed to step count."))
    .addOnFailureListener(e -> Log.w(TAG, "Failed to subscribe.", e));
```

This ensures step data is tracked (even if your app is not running). Similarly, you could subscribe to `TYPE_HEART_RATE_BPM` if the device can measure heart rate. Subscriptions persist, so do this once (e.g., on first app use after permissions). - If data should be there (e.g., the Google Fit app shows it) but your app's query returns empty, try adding `.enableServerQueries()` to your DataReadRequest to fetch latest from the cloud [1]. Also double-check your time range and units (if you use seconds vs milliseconds incorrectly, you might query an incorrect time window). - For daily totals, remember they reset at midnight local time. If you test around midnight, you might see today's count drop to zero.

- **Inconsistent Data or Duplicates:** Sometimes developers worry about duplicate data points (e.g., if multiple apps record the same steps). Google Fit should handle de-duplicating data from the same source. However, if you combine data from different sources (like phone and watch both contributing steps), the `com.google.step_count.delta` *derived* stream called **"estimated_steps"** merges those. Use the `DataSource` for `"estimated_steps"` (as shown in the Google Fit docs) to get the unified step count. If you simply call `readDailyTotal`, it already uses all sources by default. For custom DataReadRequests, you can specify particular DataSources if needed to avoid double counting. In general, use aggregated **derived data** provided by Google (like estimated_steps, which avoids counting duplicates from multiple sensors).

- **OAuth Consent Screen "Unverified App":** If you're testing and see a warning that the app is unverified, that's because you are using sensitive scopes (like fitness.body or fitness.heart_rate) without having gone through Google's verification process. During development with a few test users, you can bypass the warning (by clicking advanced -> proceed). **Solution:** Before releasing, submit your app for verification. Google will want to ensure your app's use of the data complies with their policies. They categorize the fitness scopes as **restricted** (as seen in the scopes table), meaning verification and perhaps a privacy policy are required. Plan ahead, as verification can take time. If you cannot get verified, consider limiting to less sensitive scopes if possible.

- **Debugging Tips:** Use logcat generously when reading data. Log the DataSets and DataPoints as shown above to see what's coming in. Sometimes the data might be there but you parsed it incorrectly. For example, forgetting to use the correct `Field` constant will result in no value. Always use the Field associated with the DataType (e.g., `Field.FIELD_WEIGHT` for weight, `Field.FIELD_STEPS` for step count, etc., as per Google Fit documentation). The Android Studio debugger can also inspect the `DataPoint` objects – they contain fields like `dataType`, `timestamp`, and `fields` which you can examine.

- **Using the Google Fit REST API vs Android SDK:** This guide focused on the Android SDK (Google Play services) approach. If you ever need to use the REST API (e.g., server-to-server or for an unsupported platform), be aware the authentication flow is different (OAuth web flow, obtaining tokens). The data formats are similar, but you'll manually handle HTTP requests. For most Android apps, the Play services SDK is easier and preferred.

- **Migrating to Health Connect:** As a forward-looking note, Google is converging health and fitness data access into **Health Connect** (on Android). Health Connect is a newer platform (on Android 13+ as of writing) that allows sharing fitness/health data between apps in a user-controlled way. It covers similar data (exercise, sleep, vitals, etc.). Since Google Fit's cloud API will eventually shut down (2026), you might consider using Health Connect for future development. The programming model is different (read/write data via Android permissions and a local database). If maintaining a Fit integration now, keep an eye on migration guides.

By following this guide, you should be able to integrate Google Fit into your Android app to read activity data like steps, heart rate measurements, and body metrics. Always test thoroughly with real data (you can use the Google Fit app or a Wear OS emulator to generate some dummy data if needed). With proper permissions handling and a good user experience around the data, Google Fit can greatly enhance your app's fitness features.

References:

- Google Fit Android API Documentation – **Android APIs Overview**
- Google Fit API Authorization and Data Types – **Data Types & Scopes**
- Google Fit Developer Guides – **History API (Read Data)**
- Google Fit Developer Guides – **Recording API (Subscribe)**
- Official Google Fit Samples and Codelabs (for practical code examples).

---

[1] How to integrate Google Fit with your Android app • intent
https://www.withintent.com/blog/integrate-google-fit-with-android-app/