Android Wear Docs

Release 1.1

Michael Hahn

April 30, 2015

Contents

By Michael Hahn, May 2015

What is Android Wear?

Android Wear is the Google API for smart watches, Google Glass, and other wearable devices. These wearables work in conjunction with an Android smartphone to put useful information at your fingertips. There is now a wide selection of Android watches to choose from, including those that provide GPS and offline music. This means you can now use the watch for common activities, like exercise or navigation, even when it is not paired with your phone.

Now that Apple has introduced their smart watch, wearables have crossed the tipping point of just being a consumer plaything to an integral part of work and play. Remember that It was just a few years ago when the smart phone was introduced, and now it is part of everyday life.

If you are new to Android Wear you first need to learn how to present timely and concise information on the small screen of a wearable, and then become familiar with the software and tools that make it all possible. This site helps developers quickly work through the basic concepts and develop their first wearable app. If you are interested in developing apps for the Apple Watch, see http://www.applewatchdocs.com.

Contents:

How Does Android Wear Work?

By Michael Hahn, December 2014

The easiest way to learn how Android Wear works is to install the Android Wear companion app on your handheld device, pair with an Android watch or emulator and try out a few examples. This site focuses on wearable app development, so it starts by explaining how to set up a development environment that you can use to build and run sample code provided by Android.

1.1 Set Up the Development Environment

To try out Android Wear in a development environment, perform the following tasks:

1. Install Android Studio.

2. Install the Android Wear system image and support packages using the Android SDK Manager.

 • Android 5.0 (API 21)

 • Android SDK Tools 24 or later

 • Android Support Library 21 or later

 • Android Support Repository 11 or later

 • Android Wear ARM EABI v7a System Image 21 or later (for wearable emulator)

1.2 Set Up Your Handheld Device

1. Install the Android Wear companion app on your handheld device.

2. Start the Android Wear companion app.

 The first time you launch the app, an onscreen message reminds you that Android Wear is not a notification listener. Follow the onscreen instructions.

3. Enable USB debugging on your handheld device.

 Your handheld device disables USB debugging by default, and the option to enable it can be hidden as well. For Samsung Galaxy, you must open Options, select About Phone, and then click Build Number seven times. This adds Developer Options to the Options menu, where you can enable USB debugging.

4. Connect your handheld device to the computer with a USB cable.
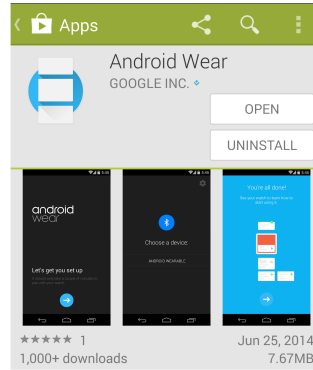
Accept any warning or security messages displayed on either the handheld device or computer.

5. Verify that the handheld device successfully connected to the computer using the following command:

```
adb devices
List of devices attached
a1b2c3d5 device
```

If a device is not displayed or it displays with an error such as unauthorized, you must resolve that problem before proceeding.

Note: The adb executable is located in the Platform Tools directory of your Android SDK. Add it to your path if necessary.

1.3 Set Up Your Wearable

You can try the sample apps using either an Android device or emulator. A wearable initially displays the default watchface, which varies by device. An emulator generally defaults to a digital watch face on a sky background. Notifications are displayed as they arrive at the bottom of the display. The following example shows that it is early in the morning and you have not exercised yet.
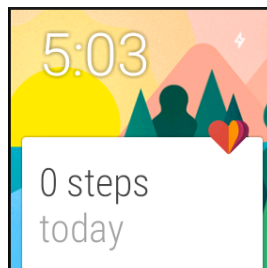
You can swipe vertically to scroll through other notifications, swipe to the right to delete the current notification, and swipe to the left to view any associated actions. The notifications displayed on the wearable are the same as those listed in the handheld, in the action bar pull down.

1.3.1 Android Emulator Setup

1. Start the Android AVD Manager.

```
android avd
```

Note: The adb executable is located in the Tools directory of your Android SDK. Add it to your path if necessary.

2. Click **Create** to define a new Android Wear emulator.

Figure 1.1: Make sure to check the Keyboard option. You might need it later.

3. Click **OK** to save your changes.

4. Click **Start** to launch the the emulator.

   Initially the emulator screen shows the time and two icons on a cloud background.
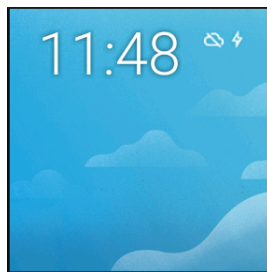
Start an Emulator Debug Session

1. Enter the following command in a command window.

   ```
   adb -d forward tcp:5601 tcp:5601
   ```

2. In the Android Wear companion app, choose **Pair with emulator**.

When the Android Wear companion app successfully pairs with the emulator, the action bar displays Emulator Connected.

On the emulator, the cloud icon disappears, and notifications are displayed as they are received.

1.3.2 Wearable Device Setup

1. Pair a wearable device with your handheld using the Android Wear companion app.

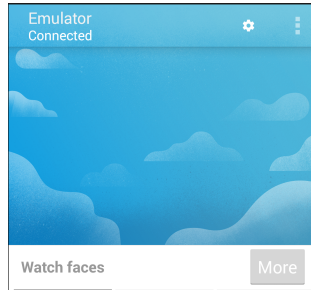   When you first pear with your wearable, the Android Wear companion app provides a short tutorial that introduces the Wearable UI and basic functionality. From the main UI where you can then change the watchface, enable voice actions, and browse suggested apps.

2. Enable bluetooth debugging on the wearable.

   Your wearable device disables USB debugging by default, and the option to enable it can be hidden as well. You must open Settings, select About, and then click Build Number seven times. The Settings menu then includes Developer options, where you can enable debugging over bluetooth.

Start a Device Debug Session

1. On the handheld, open the Android Wear companion app.

2. Tap the Settings icon. The Settings dialog is displayed.

3. Enable Debugging over Bluetooth. The following is displayed initially:

   ```
   Host: disconnected
   Target: connected
   ```

4. Enter the following command on your computer.

   ```
   adb forward tcp:4444 localabstract:/adb-hub
   adb connect localhost:4444
   ```

5. The companion app changes to the following:

   ```
   Host: connected
   Target: connected
   ```

1.4 Next Steps

You are now up and running with Android Wear, and ready to move on to your first wearable app. Initially, you write an app that can display notifications and receive user inputs from a wearable device, but runs code on the handheld device only. Later you can develop more powerful software that runs Android code on the wearable device as well.

What About the Sample Apps?

By Michael Hahn, December 2014

Google provides a variety of sample applications for wearables that demonstrate the basic capabilities of Android Wear. All are Android Studio projects that you can download, compile, and run on a wearable. This section provides an introduction to the following examples.

- Watchface

- ElizaChat

- RecipeAssistant

- Notifications

2.1 Open a Sample Project in Android Studio

1. If you have not already done so, *Set Up the Development Environment*.

2. Start Android Studio. The Welcome screen is displayed.

3. Select **Import an Android code sample** from the Quick Start panel. The Browse Samples page is displayed.

4. Select the desired sample from the list of samples.

5. Click **Next**. The Sample Setup page is displayed.

6. Accept the defaults and click **Finish**.

7. Verify that the project opens without errors. Sometimes you need to install or update sdk packages.

8. Start the Android Wear companion app on your Android handheld, if necessary.

2.2 Launch the Sample App or Service

2.2.1 Handheld Code

1. Select **Application** in the toolbar.

2. Select **Run**. If the sample has a default Activity, it is started.

3. If the sample does not have a default activity, the Run/Debug Configuration dialog is displayed. Select **Do Not Launch Activity**, then click **OK**.

4. When prompted to choose a device, select your handheld device and click **OK**.

5. Wait for the Application to fully compile and start.

2.2.2 Wearable Code

If the sample has Wearable code perform these steps.

1. Select **Wearable** in the toolbar.

2. Select **Run**. If the wearable has a default Activity, it is started.

3. If the Wearable does not have a default activity, the Run/Debug Configuration dialog is displayed. Select **Do Not Launch Activity**, then click **OK**.

4. When prompted to choose a device, select your wearable device and click **OK**.

2.3 Try the Watchface Sample

This sample installs six watch face samples. These include a variety of analog and digital time displays, including full-screen displays of the current time and split-screen displays of time and timely information. These samples are a starting point for developers interested in creating innovative new watch faces.

The watch face sample consists of services only; it is not an application that you can launch from an icon. On the handheld, you view the sample watch faces in the opening page of the Android Wear companion app.
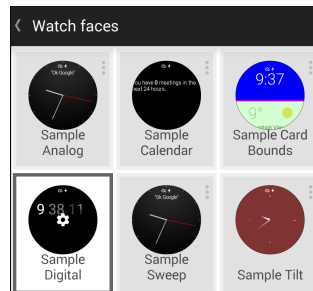
On the wearable, you view the sample watch faces in Settings, under the Change Watch Face. In the small screen, you must scroll through them one at at time.
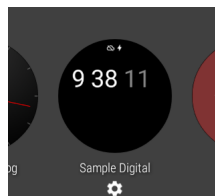
On both the handheld and wearable, watch faces display a gear icon if they have customizable settings. In the Android Wear companion app, selecting the sample digital watch face displays the following choices:

On the wearable, you can only choose a background color when you select the icon:

WatchFace

Digital watch face configuration
(com.example.android.wearable.watchface.DigitalW
atchFaceService)

Background	Black
Hours	White
Minutes	White
Seconds	Gray

Background color

⚫ Black

🔵 Blue
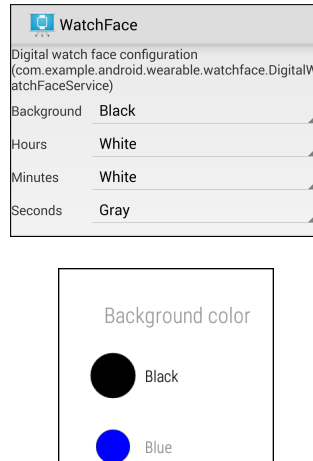
2.4 Try Eliza Chat

The Eliza Chat sample app shows how you might implement a Personal Digital Assistant on a wearable device. Eliza is the assistant in this example. Eliza posts responses on the wearable emulator and you enter questions by tapping a reply icon. Normally you would provide voice inputs, but that is not implemented in this sample. For now, you simply type what you have to say.

When you first launch the app, Eliza asks how she can help you.
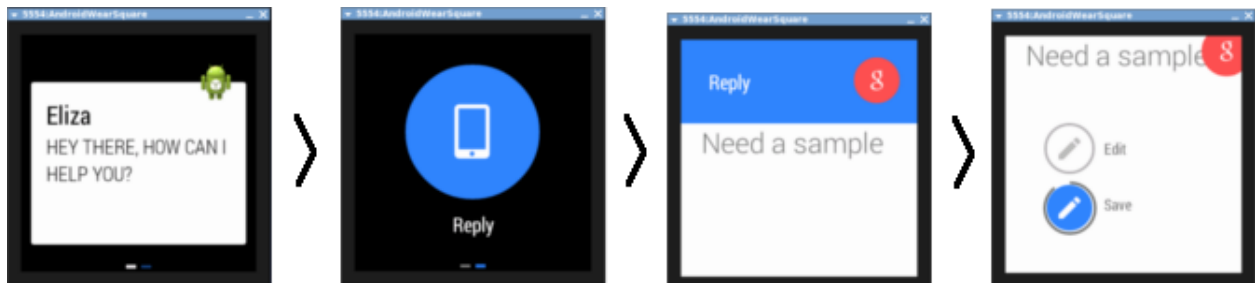
To reply, swipe the screen to the left, select the reply icon, and then type your demand in the Reply form. Before the Eliza app accepts your demand, you choose from two options, Edit or Save. This sequence demonstrates a typical UI pattern, which consists of a notification, a reply, and a fixed choice.

Eliza then responds to your question and you can continue with the dialog. The entire session is recorded on the handheld device. The following screen shows the transcript for several exchanges with Eliza.

2.5 Try Recipe Assistant

The Recipe Assistant app is an example of a fully-functional Android application that extends it capabilities to a wearable device. On the handheld device, you can scroll through the full recipe and steps. On the wearable device you view the recipe steps one at a time on separate pages.

The app starts on the mobile device with three recipes from which to choose. Select the first one and you get the guacamole recipe.

Click Start in the upper right corner, and the recipe is displayed on the wearable emulator.
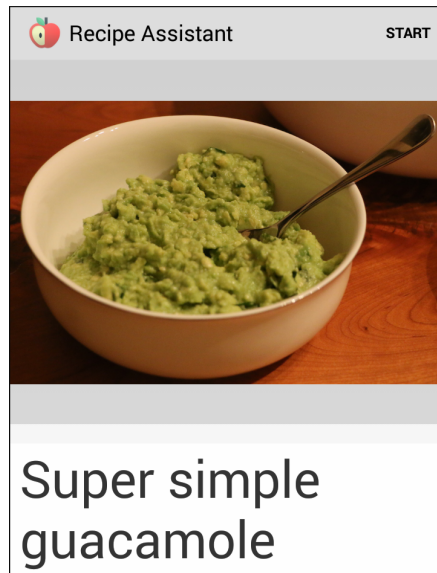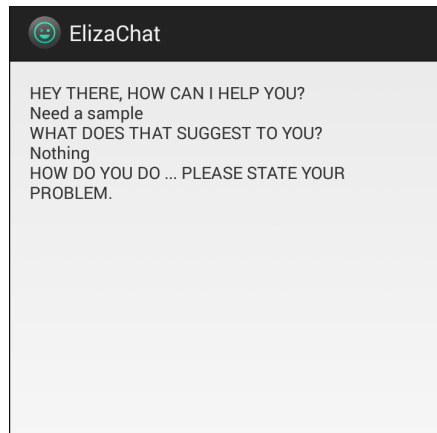
Swipe left to move through the steps. For each step you can tap to view more and swipe up or down to move through the content.

2.6 Try Wearable Notifications

The Wearable Notifications sample app provides a simple way to try out different combinations of wearable UI notifications and patterns. The onscreen choices closely match the capabilities of the `WearableNotifications` class, so this is a good opportunity to learn about available displays. Later we will experiment with the underlying code.
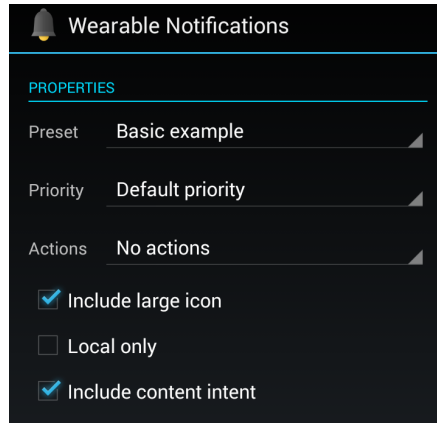
The UI is basically a smorgasbord of wearable notification possibilities. The best way to use this app is simply to try out all the choices. You can select a basic notification type (preset), set its priority, and attach action icons. You can also include a background image that provides context for the text notification. The Android Wear app is another way to view different notification examples. The examples in the app are based on use cases, like traffic, weather, and sports, rather than underlying API options like Big Screen and Big Text.

Android Wear Suggest

By Michael Hahn, January 2015

The Suggest context stream is one of the core functions for Android Wear. It consists of a sequence of notifications about timely information, such as incoming messages or upcoming appointments. It can also display useful information about a task at hand, such as preparing a recipe or communicating with a digital assistant.
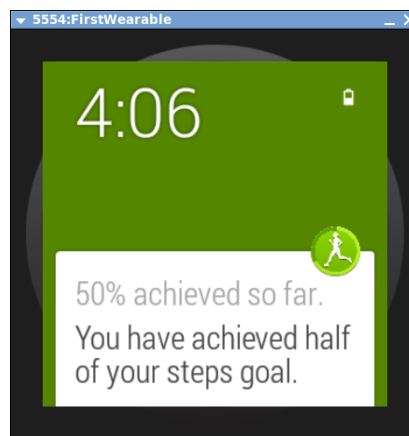
This section explains how to display your own custom notifications on a wearable device. The easiest way is to create a normal Notification, initialize it with your custom message, and send it using the NotificationManager. These notifications are displayed on both the handheld device and wearable emulator with a similar level of detail.

Normal text notifications are only the beginning however. Android 4.1 introduced three additional styles: Big Picture, Big Text, and Inbox. The big picture example demonstrates one way to add a contextual image to the notification.
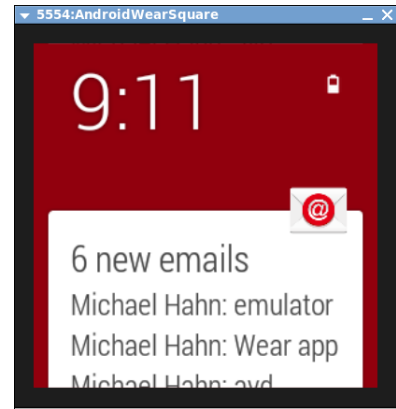
Android Wear adds even more styles that improve the user experience on the small screen of a wearable device. These styles make it possible to group or add pages to notifications. The email example shows how messages are grouped to reduce the number of notification delivered to a wearable.

You do not have to rely any of these stock UI styles. You can create your own full-screen layout that best suits your custom wearable application. Just keep it simple and be consistent in presentation and usage with other wearable displays. For example, do not try to replicate the grid layout of the handheld device - the wearable is just too small for this approach. Users just glance at their watch, speak simple commands, or tap and swipe the screen.

3.1 First Android Wear Suggest

This section explains how to create your first Android Wear notification and add it to the Suggest context stream on an Android wearable, or emulator. The new project wizard in Android Studio creates a project with two main activities, one for the handheld device and another for the wearable. To create your first suggest notification, add code in the handheld activity only, located in the "mobile" branch of the project hierarchy. The preinstalled software on a wearable device or emulator handles the task of receiving and displaying notifications from the handheld.

3.1.1 Create a Project

This section explains how to create a new project using Android Studio Version 1. If you have an earlier version, update it before starting this procedure.

1. Launch Android Studio.

2. Select **Start a New Android Studio Project** in the Welcome screen.

3. In the new project dialog, enter your Application Name, Domain, and a project directory. Click **Next**.

4. In the form factors dialog, select **Phone and Tablet** and **Wear**. Keep the default SDKs, and then click **Next**.

5. In the mobile activity dialog, select **Blank Activity**. Click **Next**.

6. In the mobile option dialog enter an Activity Name for the handheld. Accept the defaults or the other fields and click **Next**. The name for an activity that generates a wearable "suggestion" might be SuggestActivity.

7. In the wear activity dialog, select **Blank Wear Activity**. Then click **Next**.

8. In the wear options dialog enter an Activity Name for the wearable. Accept the defaults for the other fields and click **Finish**. You can use the same name for the wearable activity that you used for the handheld activity.

3.1.2 Modify the Handheld Activity

1. Import the the packages that support wearable features into the handheld Activity. As a minimum you need the following notification packages:

```
import android.support.v4.app.NotificationManagerCompat;
import android.support.v4.app.NotificationCompat;
import android.support.v4.app.RemoteInput;
import android.app.Notification;
```

2. Add Android Wearable features to a Wearable extender object, for example ShowBackgroundOnly, to the on-Create method.

```
NotificationCompat.WearableExtender wearableExtender =
    new NotificationCompat.WearableExtender()
        .setHintShowBackgroundOnly(true);
```

3. Create a normal Android notification using the NotificationCompat.Builder and set desired properties, including those defined in the WearableExtender.

```
Notification notification =
    new NotificationCompat.Builder(this)
        .setSmallIcon(R.drawable.ic_launcher)
        .setContentTitle("Hello Android Wear")
        .setContentText("First Wearable notification.")
        .extend(wearableExtender)
        .build();
```

3. Optionally, apply a release 4.1 style to the normal notification, such as the one used in the Big Picture example (NotificationCompat.BigPictureStyle).

4. Get an instance of the Notification Manager service.

```
NotificationManagerCompat notificationManager =
    NotificationManagerCompat.from(this);
```

5. Dispatch the notification.

```
int notificationId = 1;
notificationManager.notify(notificationId, notification);
```
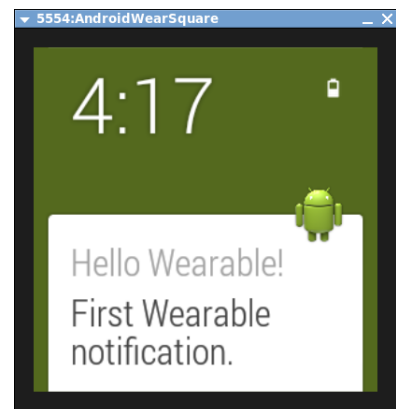

The result of this example is a notification with a title and body. If necessary, scroll through other notifications to view this one.

3.2 Example

The full Android Studio project for this example is posted at https://github.com/LarkspurCA/WearableSuggest.

Android Wear Demand

By Michael Hahn, January 2015

The Demand context is one of the core functions for Android Wear. A demand is displayed as a large icon, typically when you swipe a displayed suggestion (notification). You tap on an icon to perform the desired demand.
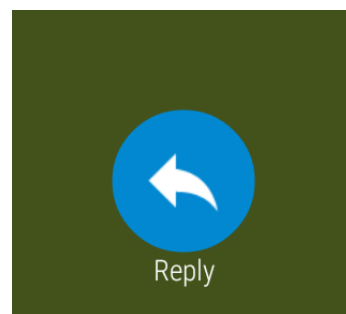

When new email arrives for example, you swipe to the left to scroll through the demand icons, such as open, reply, and archive. You can create custom icons for your app as well. This section explains how to code your own wearable demand that handles a voice reply.

4.1 First Android Wear Demand

If you have not already done so, *Create a Project*. The new project wizard in Android Studio creates a project with two main activities, one for the handheld device and another for the wearable. To create your first demand, add code in the handheld activity only, located in the "mobile" branch of the project hierarchy. The software preinstalled on a wearable device or emulator handles the task of displaying notifications and handling demands on the wearable side.

4.1.1 Add Build Dependencies

Add build dependencies for the wearable support package to the build.gradle (Module:mobile) file in Gradle Scripts folder, if necessary.

```
dependencies {
  compile fileTree(dir: 'libs', include: ['*.jar'])
  wearApp project(':wear')
  compile 'com.android.support:appcompat-v7:21.+'
  compile 'com.google.android.gms:play-services:6.5.+'
}
```

4.1.2 Modify the Handheld Activity

Your handheld activity initiates the process by creating a notification that includes an action (demand). When a user views the notification and invokes the demand, the wearable broadcasts the demand to the handheld for processing. You build the notification with a hierarchy of objects, intent -> pending intent -> notification action -> wearable extender, and finally the notification itself.

1. Create an Intent that defines the action that the handheld device should take in response to a wearable demand.

   The Intent has the following parameters:

   > The context of the activity, service, or broadcast receiver on the handheld that handles wearable demands.

   > The name of the class on the handheld that receives wearable demands.

   > A constant that specifies a demand action.

   > An extra with details about the requested action.

   The following example shows how to create an intent for a reply demand. The DemandIntentReceiver is defined later in *Create a Demand Receiver*.

   ```
   public static final String ACTION_DEMAND = "com.androidweardocs.ACTION_DEMAND";
   public static final String EXTRA_MESSAGE = "com.androidweardocs.EXTRA_MESSAGE";
       public static final String EXTRA_VOICE_REPLY = "com.androidweardocs.EXTRA_VOICE_REPLY";

   Intent demandIntent = new Intent(this, DemandIntentReceiver.class)
     .putExtra(EXTRA_MESSAGE, "Reply selected.")
     .setAction(ACTION_DEMAND);
   ```

2. Create a PendingIntent to include in the notification.

   A PendingIntent wraps the intent to grant the privileges it needs for to execute in your application. It contains the context of the activity, service, or broadcast receiver that will receive the demand, and the Intent object itself. This example creates a PendingIntent using the context of a broadcast receiver. Use getActivity instead of getBroadcast if your activity receives demands.

   ```
   PendingIntent demandPendingIntent =
       PendingIntent.getBroadcast(this, 0, demandIntent, 0);
   ```

3. Create a RemoteInput object to hold a voice reply from the wearable device. A voice request or response is a common action for a wearable device because of the small size of the UI.

   ```
   String replyLabel = getResources().getString(R.string.app_name);
   RemoteInput remoteInput = new RemoteInput.Builder(EXTRA_VOICE_REPLY)
     .setLabel(replyLabel)
     .build();
   ```

4. Create a wearable action.

   The following example creates an wearable action that uses a standard reply icon and label, adds the pending intent, and the RemoteInput object for voice.

   ```
   NotificationCompat.Action replyAction =
     new NotificationCompat.Action.Builder(R.drawable.ic_reply_icon,
       getString(R.string.reply_label), demandPendingIntent)
       .addRemoteInput(remoteInput)
       .build();
   ```

5. Create a WearableExtender for the a notification and add the wearable action.

```
NotificationCompat.WearableExtender wearableExtender =
   new NotificationCompat.WearableExtender()
        .addAction(replyAction);
```

6. Create a notification and extended it with the wearable extender just created. The following example creates a notification that includes a reply action (demand).

```
Notification notification =
   new NotificationCompat.Builder(this)
      .setContentTitle("Hello Wearable!")
      .setContentText("First Wearable demand.")
      .setSmallIcon(R.drawable.ic_launcher)
      .extend(wearableExtender)
      .build();
```

7. Get an instance of the Notification Manager service.

```
NotificationManagerCompat notificationManager =
   NotificationManagerCompat.from(this);
```

8. Dispatch the extended notification.

```
int notificationId, notification;
notificationManager.notify(notificationId, notification);
```

4.1.3 Create a Demand Receiver

When a user makes a demand, the wearable broadcasts an intent that contains the details. The handheld receives the broadcast and takes an appropriate action. The following example defines a BroadcastReceiver for a voice demand that simply logs the results.

```
public class DemandIntentReceiver extends BroadcastReceiver{

   @Override
   public void onReceive(Context context, Intent intent) {

      if (intent.getAction().equals(Handheld.ACTION_DEMAND)) {String message =
               intent.getStringExtra(Handheld.EXTRA_MESSAGE);
        Log.v("MyTag","Extra message from intent = " + message);
        Bundle remoteInput = RemoteInput.getResultsFromIntent(intent);
        CharSequence reply = remoteInput.getCharSequence(Handheld.EXTRA_VOICE_REPLY);
        Log.v("MyTag", "User reply from wearable: " + reply);
      }
   }
}
```
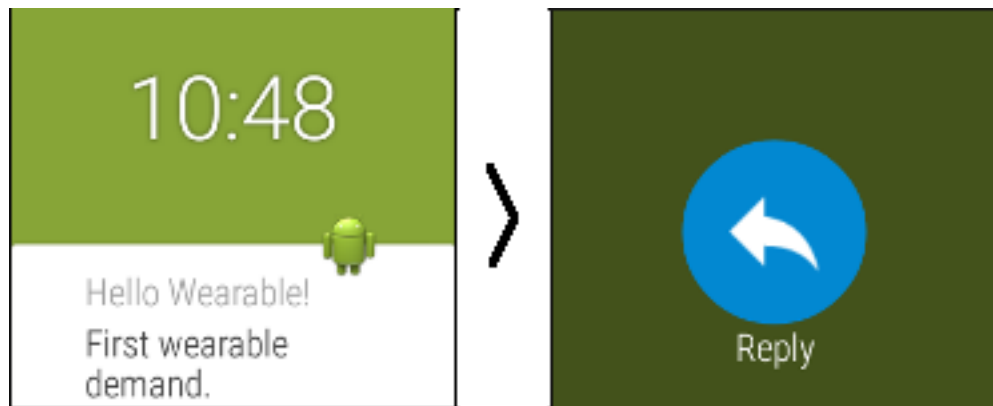
You also need to modify the manifest file to accept the broadcast. Add a receiver section within the application section.

```
<receiver android:name=".DemandIntentReceiver" android:exported="false">
  <intent-filter>
    <action android:name="com.androidweardocs.first_wearable.ACTION_DEMAND"/>
  </intent-filter>
    </receiver>
```

The sample code first displays the notification text, which is simply a title and body. Swiping to the left displays the reply icon with the reply label.

Selecting the reply icon displays the voice prompt, and then the confirmation after you speak (or type on the emulator).

The example DemandIntentReceiver receives the broadcasted intent and extracts the voice input. Log messages show the value of the extra text within the intent and the voice string from the remote input.

```
MyTag: Extra message from intent = Reply icon selected.
MyTag: User reply from wearable: hello handheld
```

4.2 Example

The full Android Studio project for demands is posted at https://github.com/LarkspurCA/WearableDemand.

CHAPTER 5

Wearable Application Launch

By Michael Hahn, April 2015

Android wearable devices have an Android operating system, so you can develop applications for wearables much as you do for handheld devices. Simple handheld apps can potentially run in a standalone mode on the wearable, without being paired with a handheld. More commonly however, wearable apps work in conjunction with a handheld for voice recognition, Internet access, and other services.

Android Studio creates a very simple "Hello World" wearable app when you use the new project wizard. If you select Wearable as an option, the wizard creates both a handheld and a wearable app. Typically you install and launch this app using Android SDK tools.

5.1 Voice Activation

An easy way to launch a wearable app is with a voice command. Either say "OK Google" or touch the screen. At the "Speak Now" prompt tell Google to start your app, for example "start first wearable". The wearable then confirms your voice command and launches the the app.
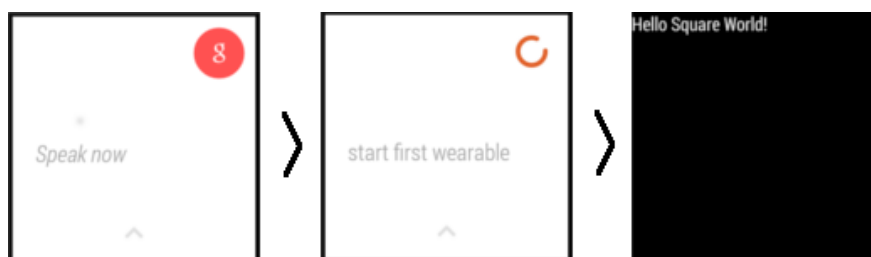
5.2 Menu Activation

To manually launch the app, touch the watch face and scroll to the last action, which is "Start...". Then select your app from the list of installed apps, in this case First_Wearable.

So far, not a single line of new code was necessary; the Android Studio new project wizard created it all. The upcoming sections show how to add more interesting content and actions.
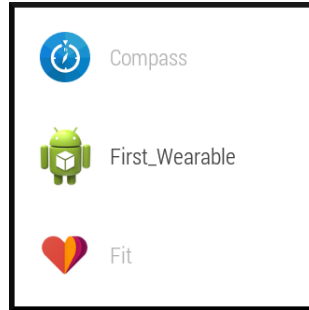
5.3 Handheld Activation

A great way to start your wearable app is from a notification on the wearable. This is useful when you already have a handheld app and want to extend features to a wearable. The handheld app starts the launch process by sending a data object to the wearable that contains the information the wearable needs to post a notification locally. Users swipe this notification and tap the Open icon to launch the wearable portion of your app. They return to the notification stream when the wearable app closes.

Often times you want to launch the wearable app with extra data that tells it what feature to open or what data to display. For example, a general handheld golf app might open the wearable for a specific golf course. This section includes details about adding such data to the launch process. Without this, the wearable app simply launches in its default mode.

5.3.1 Prerequisite

This procedure relies on communication through the data layer. Communication through the data layer requires setup of Google Play Services for wearables in both the handheld and wearable devices (see *Data Layer DataMap Objects*).

5.3.2 Post a Notification

1. First create a DataMap object that includes the title and body for the notification. Optionally, include any extra data that you want to pass to your wearable app when it starts. Then send this DataMap object to the wearable data layer, along with a path constant that identifies the purpose of the data. The SendToDataLayerThread class in this example is defined in *Data Layer DataMap Objects*.

```
String WEARABLE_START_PATH = "/wearable_start";

// Create a DataMap
DataMap notifyWearable = new DataMap();
notifyWearable.putString("title", "Notification Title");
notifyWearable.putString("body", "Start now?");
// Optional extra data to use when starting wearable
notifyWearable.putString("extra", "extra data")
// Send to data layer
new SendToDataLayerThread(WEARABLE_START_PATH, notifyWearable).start();
```

2. In the wearable, receive the DataMap in a WearableListenerService class. The following excerpt shows a sample override for the onDataChanged method of the service.

```
@Override
public void onDataChanged(DataEventBuffer dataEvents) {
  DataMap dataMap;
```

```java
    for (DataEvent event : dataEvents) {
        // Check the event type
        if (event.getType() == DataEvent.TYPE_CHANGED) {
            // Check the data path
            if (path.equals(WEARABLE_START_PATH)) {
                // Create a local notification
                dataMap = DataMapItem.fromDataItem(event.getDataItem()).getDataMap();
                sendLocalNotification(dataMap);
            }
        }
    }
}
```

3. In the wearable, implement the procedure that constructs and posts a demand (notification) that can launch your app. Optionally, the Pending Intent in this notification can include extra data for the wearable app.

```java
private void sendLocalNotification(DataMap dataMap) {
  int notificationId = 001;

  // Create a pending intent that starts this wearable app
  Intent startIntent = new Intent(this, HoleActivity.class).setAction(Intent.ACTION_MAIN);
  // Add extra data for app startup or initialization, if available
  startIntent.putExtra("extra", dataMap.getString("extra"));
  PendingIntent startPendingIntent =
    PendingIntent.getActivity(this, 0, startIntent, PendingIntent.FLAG_CANCEL_CURRENT);

  Notification notify = new NotificationCompat.Builder(this)
    .setContentTitle(dataMap.getString("title"))
    .setContentText(dataMap.getString("body"))
    .setSmallIcon(R.drawable.ic_launcher)
    .setAutoCancel(true)
    .setContentIntent(startPendingIntent)
    .build();

  NotificationManagerCompat notificationManager = NotificationManagerCompat.from(this);
  notificationManager.notify(notificationId, notify);
}
```

The wearable notification stack now includes a notification inviting the user to launch your wearable app. A swipe to the left displays the launcher icon, which the user clicks to launch the app.

4. In the wearable app, receive and process any extra information. Normally, you implement this within the onCreate override of your app.
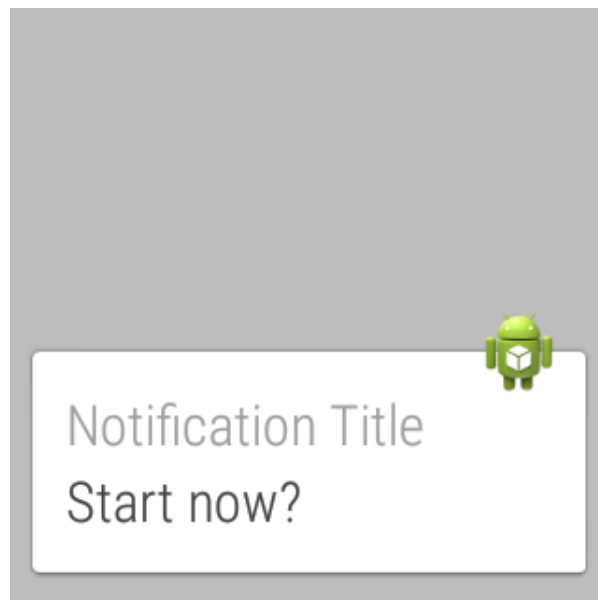
```java
// Check for extra data in the intent
// If present, extract and use

Bundle extras = getIntent().getExtras();
if (extras != null) {

    // Get the extra data
    String extraData = extras.getString("extra");
    ...
    // Act on the extra data
    ...
}
```
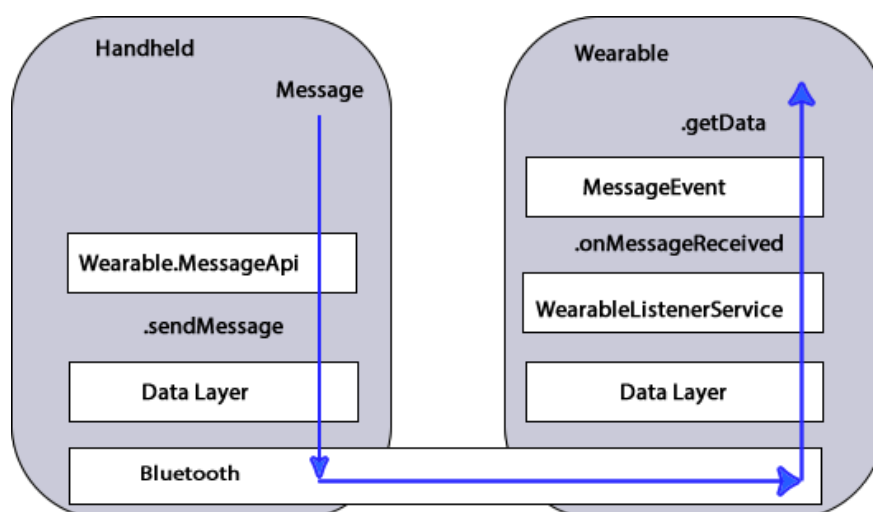
Data Layer Messages

By Michael Hahn, January 2015

An application that runs on a wearable device usually utilizes some of the capabilities of a paired handheld device. This means you need two separate Android apps, one that runs on the wearable and another that runs on the handheld. These two apps communicate with one another over the bluetooth link that connects the two devices.

Google Play services Version 5 and later include a Wearable Message API that provides access to the data layer of a data communications link between the two devices. Messages or data move down the protocol stack on the sending side, across the bluetooth link, then up the stack on the receive side. The following diagram shows how a simple message flows through the wearable communications link.

In this example, a handheld sends a message to a wearable using the sendMessage method of the Wearable.MessageApi. On the receiving side, a WearableListenerService monitors the data layer and invokes the on-MessageReceived callback when a message arrives. The listener service then performs some application-specific task based on the received message. The WearableListenerService is not the only way to receive data, but it is easy to implement because Android Wear manages its life-cycle.

6.1 First Wearable Message

If you have not already done so, *Create a Project*. The new project wizard in Android Studio creates a project with two main activities, one for the handheld device and another for the wearable. These two activities use the same package

name, which is essential for the wearable data layer to work.

Data layer messages can originate from either a handheld or wearable. For bidirectional messaging both the handheld and wearable should implement a message sender and listener.

6.1.1 Implement a Message Sender

This section describes how to send messages to the data layer. The example code shown sends messages from the handheld to the data layer.

Add Build Dependencies

Add the following build dependencies to the build.gradle file (Module:mobile) in the Gradle Scripts folder, if necessary.

```
dependencies {
  compile fileTree(dir: 'libs', include: ['*.jar'])
  compile 'com.android.support:appcompat-v7:21.0.+'
  compile 'com.google.android.gms:play-services:6.5.+'
}
```

Add Metadata for Google Play Services

Add Google Play services metadata statement to the manifest of the sending device:

```
<application>
...
  <meta-data android:name="com.google.android.gms.version"
   android:value="@integer/google_play_services_version" />
</application>
```

Create the Message Sender

To send a message , update the code in the main Activity of the sending device. For the handheld, modify the code in the **mobile** branch of the Android Studio project.

1. Build a Google Play Services client for the Wearable API.

```
public class MessageActivity extends Activity implements
  GoogleApiClient.ConnectionCallbacks,
  GoogleApiClient.OnConnectionFailedListener {

  GoogleApiClient googleClient;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_handheld);

    // Build a new GoogleApiClient for the Wearable API
    googleClient = new GoogleApiClient.Builder(this)
      .addApi(Wearable.API)
      .addConnectionCallbacks(this)
      .addOnConnectionFailedListener(this)
      .build();
```

```
    }
        // Data layer and lifecycle implementation (Step 2)
        ...
}
```

2. Add callback methods for the data layer and lifecycle events. For simplicity, send a message in the onConnected callback method.

```java
// Connect to the data layer when the Activity starts
@Override
protected void onStart() {
  super.onStart();
  googleClient.connect();
}

// Send a message when the data layer connection is successful.
@Override
public void onConnected(Bundle connectionHint) {
  String message = "Hello wearable\n Via the data layer";
  //Requires a new thread to avoid blocking the UI
  new SendToDataLayerThread("/message_path", message).start();
 }

// Disconnect from the data layer when the Activity stops
@Override
protected void onStop() {
  if (null != googleClient && googleClient.isConnected()) {
    googleClient.disconnect();
    }
    super.onStop();
}

// Placeholders for required connection callbacks
@Override
public void onConnectionSuspended(int cause) { }

@Override
public void onConnectionFailed(ConnectionResult connectionResult) { }
```

3. Define a class that extends the Thread class and implements a method that sends your message to all nodes currently connected to the data layer. This task can block the main UI thread, so it must run in a new thread.

```java
class SendToDataLayerThread extends Thread {
  String path;
  String message;

  // Constructor to send a message to the data layer
  SendToDataLayerThread(String p, String msg) {
    path = p;
    message = msg;
  }

  public void run() {
    NodeApi.GetConnectedNodesResult nodes = Wearable.NodeApi.getConnectedNodes(googleClient).awa
    for (Node node : nodes.getNodes()) {
      SendMessageResult result = Wearable.MessageApi.sendMessage(googleClient, node.getId(), pat
      if (result.getStatus().isSuccess()) {
        Log.v("myTag", "Message: {" + message + "} sent to: " + node.getDisplayName());
      }
```

```
      else {
        // Log an error
        Log.v("myTag", "ERROR: failed to send Message");
      }
    }
  }
}
```

6.1.2 Implement a Message Listener

You can monitor the data layer for new messages using either a listener service or listener activity. This section explains how to implement a listener service for messages. For the wearable, modify the code in the **wear** branch of the Android Studio project.

Add Build Dependencies

Add wearable SDK dependencies to the build.gradle file (Module:wear) in the Gradle Scripts folder, as necessary.

```
dependencies {
  compile fileTree(dir: 'libs', include: ['*.jar'])
  compile 'com.google.android.support:wearable:1.1.+'
  compile 'com.google.android.gms:play-services-wearable:6.5.+'
}
```

Add Listener Service to Manifest

Enable the data layer listener in the manifest file.

```
<uses-feature android:name="android.hardware.type.watch" />

<application
  ...
  <service android:name=".ListenerService">
    <intent-filter>
      <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
    </intent-filter>
  </service>
</application>
```

Create a Listener Service Class

Create a listener in the wear application that extends the WearableListenerService. This example logs any received message to the debug output.

```
public class ListenerService extends WearableListenerService {

  @Override
  public void onMessageReceived(MessageEvent messageEvent) {

    if (messageEvent.getPath().equals("/message_path")) {
      final String message = new String(messageEvent.getData());
      Log.v("myTag", "Message path received on watch is: " + messageEvent.getPath());
      Log.v("myTag", "Message received on watch is: " + message);
```

```
    }
    else {
      super.onMessageReceived(messageEvent);
    }
  }
}
```

6.1.3 Display Received Messages

The wearable listener service cannot directly update the wearable UI because it runs on a different thread. The following example shows how to forward received messages to the main Activity using the LocalBroadcastManager.

1. In the ListenerService class, broadcast the received data layer messages locally.

```java
@Override
public void onMessageReceived(MessageEvent messageEvent) {

    if (messageEvent.getPath().equals("/message_path")) {
        final String message = new String(messageEvent.getData());

        // Broadcast message to wearable activity for display
        Intent messageIntent = new Intent();
        messageIntent.setAction(Intent.ACTION_SEND);
        messageIntent.putExtra("message", message);
        LocalBroadcastManager.getInstance(this).sendBroadcast(messageIntent);
    }
    else {
        super.onMessageReceived(messageEvent);
    }
```

2. In the wearable Activity, register to receive broadcasts from the ListenerService.

```java
@Override
protected void onCreate(Bundle savedInstanceState) {
    // Basic UI code, generated by New Project wizard.
            ...

    // Register the local broadcast receiver, defined in step 3.
    IntentFilter messageFilter = new IntentFilter(Intent.ACTION_SEND);
    MessageReceiver messageReceiver = new MessageReceiver();
    LocalBroadcastManager.getInstance(this).registerReceiver(messageReceiver, messageFilter);
}
```

3. In the wearable Activity, define a nested class that extends BroadcastReceiver, implements the onReceive method, and extracts the message. This example receives and displays the message the wearable UI.

```java
public class MessageReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String message = intent.getStringExtra("message");
        // Display message in UI
        mTextView.setText(message);
    }
}
```

Keep in mind that this example is not a full implementation. You must unregister your application from the local broadcast manager when the application stops. Otherwise you can duplicate the registration of the same application,

---

which results in duplicate local broadcasts.

6.1.4 Try the First Data Layer App

Make sure the handheld and wearable are successfully paired. If not, see *Set Up the Development Environment*.

Start the "wear" app in Android Studio. It displays the default Hello message generated by the Android Studio new project wizard:

Hello Square World!
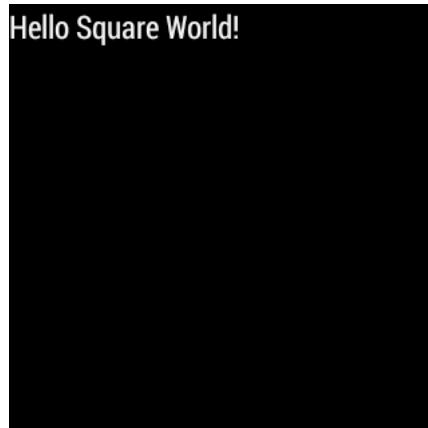
Then launch the handheld app. The wearable display changes to the message sent from the handheld device through the wearable data layer.

Hello wearable
Via the data layer
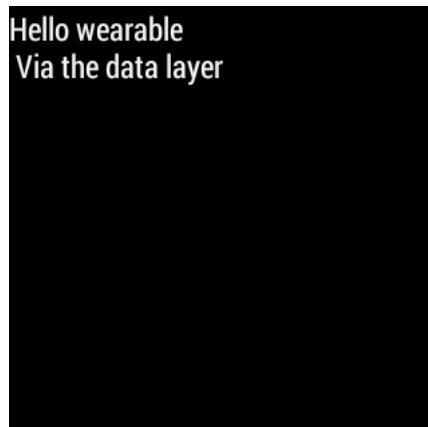
6.2 Example

The full Android Studio project for data layer messages is posted at https://github.com/LarkspurCA/WearableMessage.git.

Data Layer DataMap Objects

By Michael Hahn, January 2015

The wearable data layer can sync either messages or data. A message contains a single text string, but data is typically wrapped in a DataMap object. A DataMap is similar to a Bundle in that it contains a collection of one or more of data types, stored as key/value pairs. Android uses a Bundle to encapsulate data exchanged between activities. Similarly, wearable apps can use a DataMap to encapsulate the data exchanged over the wearable data layer.

Google Play services Version 5 includes a Wearable Data API that provides access to the data layer of a data communications link between the two devices. App data moves down the protocol stack on the sending side, across the bluetooth link, then up the stack on the receiving side. The following diagram shows how a handheld sends a data to a wearable using a DataMap and the Wearable.DataApi.
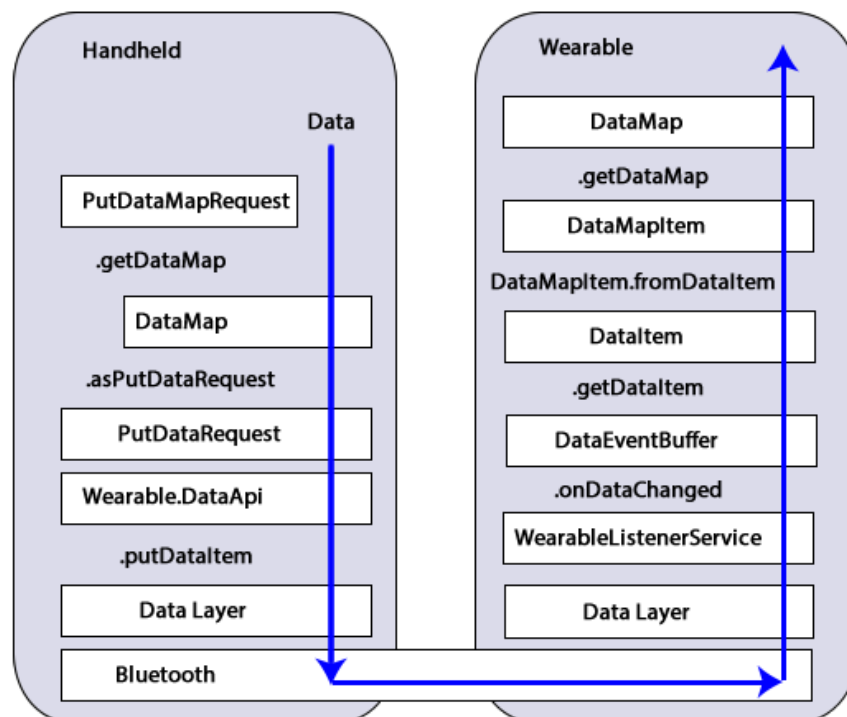
On the handheld side, you start with a PutDataMapRequest. This is a helper class that simplifies the process of creating the DataMap, DataItem, and PutDataRequest objects that are needed for sending data. On the receiving side a WearableListener Service receives changed data and returns a buffer of DataEvents. You get a DataItem from

the buffer, convert it to a DataMapItem, convert that to a DataMap object, and then get the original handheld data. A WearableListenerService is not the only way to receive data, but it is easy to implement because Android Wear manages its life-cycle.

7.1 First Wearable Data

If you have not already done so, *Create a Project*. The new project wizard in Android Studio beta creates a project with two main activities, one for the handheld device and another for the wearable. These two activities use the same package name, which is essential for the wearable data layer to work.

Data layer transfers can originate from either a handheld or wearable. For bidirectional data both the handheld and wearable should implement the code in this section.

7.1.1 Add Metadata for Google Play Services

Add Google Play services metadata statement to the manifest of the sending device:

```
<application>
...
  <meta-data android:name="com.google.android.gms.version"
   android:value="@integer/google_play_services_version" />
</application>
```

7.1.2 Add a Data Sender

To send a data object , update the code in the main Activity of the sending device.

1. Build a Google Play Services client that includes the Wearable API.

```java
public class Handheld extends Activity implements
  GoogleApiClient.ConnectionCallbacks,
  GoogleApiClient.OnConnectionFailedListener {

  GoogleApiClient googleClient;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_handheld);

    // Build a new GoogleApiClient
    googleClient = new GoogleApiClient.Builder(this)
      .addApi(Wearable.API)
      .addConnectionCallbacks(this)
      .addOnConnectionFailedListener(this)
      .build();
  }
      // Data layer and lifecycle implementation (Step 2)
      ...
}
```

2. Add callback methods for the data layer and lifecycle events. For simplicity, send a data object in the onConnected callback method. In this example the path of the data object is "wearable_data" and the data is a DataMap

object that contains a golf course hole number and the distances to the front, middle, and back pin locations. The receiving side can use the path to identify the origin of the data.

```java
// Connect to the data layer when the Activity starts
@Override
protected void onStart() {
    super.onStart();
  googleClient.connect();
}

// Send a data object when the data layer connection is successful.

@Override
public void onConnected(Bundle connectionHint) {

  String WEARABLE_DATA_PATH = "/wearable_data";

  // Create a DataMap object and send it to the data layer
  DataMap dataMap = new DataMap();
  dataMap.putLong("time", new Date().getTime());
  dataMap.putString("hole", "1");
  dataMap.putString("front", "250");
  dataMap.putString("middle", "260");
  dataMap.putString("back", "270");
  //Requires a new thread to avoid blocking the UI
  new SendToDataLayerThread(WEARABLE_DATA_PATH, dataMap).start();
 }

// Disconnect from the data layer when the Activity stops
@Override
protected void onStop() {
  if (null != googleClient && googleClient.isConnected()) {
    googleClient.disconnect();
    }
    super.onStop();
}

// Placeholders for required connection callbacks
@Override
public void onConnectionSuspended(int cause) { }

@Override
public void onConnectionFailed(ConnectionResult connectionResult) { }
```

3. Define a class that extends the Thread class and implements a method that sends your data object to all nodes currently connected to the data layer. This task can block the main UI thread, so it must run in a new thread. This can be an inner class.

```java
class SendToDataLayerThread extends Thread {
  String path;
  DataMap dataMap;

  // Constructor for sending data objects to the data layer
  SendToDataLayerThread(String p, DataMap data) {
    path = p;
    dataMap = data;
  }

  public void run() {
```

```
NodeApi.GetConnectedNodesResult nodes = Wearable.NodeApi.getConnectedNodes(googleClient).awa
for (Node node : nodes.getNodes()) {

    // Construct a DataRequest and send over the data layer
    PutDataMapRequest putDMR = PutDataMapRequest.create(path);
    putDMR.getDataMap().putAll(dataMap);
    PutDataRequest request = putDMR.asPutDataRequest();
    DataApi.DataItemResult result = Wearable.DataApi.putDataItem(googleClient,request).await()
    if (result.getStatus().isSuccess()) {
      Log.v("myTag", "DataMap: " + dataMap + " sent to: " + node.getDisplayName());
    } else {
      // Log an error
      Log.v("myTag", "ERROR: failed to send DataMap");
    }
  }
 }
}
```

7.1.3 Add a Data Receiver

You can monitor the data layer for new data objects using either a listener service or listener activity. This section explains how to implement a listener service for data objects.

1. Enable the listener service in the manifest file for the wear application.

```
<uses-feature android:name="android.hardware.type.watch" />

<application
  ...
  <service android:name=".ListenerService">
    <intent-filter>
      <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
    </intent-filter>
  </service>
</application>
```

2. Create a listener in the wear application that extends the WearableListenerService and implements onData-aChanged. This example filters incoming data events for those of TYPE_CHANGED, checks for a data path of "/wearable_data", then logs the data item to the debug output.

```
public class ListenerService extends WearableListenerService {

  private static final String WEARABLE_DATA_PATH = "/wearable_data";

  @Override
  public void onDataChanged(DataEventBuffer dataEvents) {

  DataMap dataMap;
  for (DataEvent event : dataEvents) {

    // Check the data type
    if (event.getType() == DataEvent.TYPE_CHANGED) {
      // Check the data path
      String path = event.getDataItem().getUri().getPath();
      if (path.equals(WEARABLE_DATA_PATH)) {}
        dataMap = DataMapItem.fromDataItem(event.getDataItem()).getDataMap();
        Log.v("myTag", "DataMap received on watch: " + dataMap);
```

```
                }
            }
        }
    }
```

7.1.4 Using Received Data

In this example, a background service receives the data. If you need this data in the UI or elsewhere, you can broadcast the results locally, as described in *Display Received Messages*. Just add a Bundle (DataMap.toBundle) as the intent extra, instead of a simple message string.

7.2 Example

The full Android Studio project for data layer DataMap objects is posted at https://github.com/LarkspurCA/WearableDataMap.git.

Wearable GPS

By Michael Hahn, March 2015

Wearables are great when you are on the go, especially when you are out for a run or looking for destination. It would be great to just glance at your watch for your current location, rather than pulling out your handheld. Now that some wearables have a built-in GPS sensor, you can continue to use location-based apps even when the wearable is not paired with a handheld.

You implement location services in a wearable using the Google Play Fused Location Provider, just as you would in a handheld. The Android operating system takes care of choosing the GPS sensor (wearable or handheld) and implements any necessary communications between devices. The handheld GPS is preferred when both devices have a GPS sensor, and the switchover between sensors is automatic when pairing status changes.

You can create your first location-aware wearable app without writing a single line of handheld code. You don't even need to implement a handheld Activity. The section shows how.

8.1 First Wearable GPS

If you have not already done so, *Create a Project*. The new project wizard in Android Studio beta creates a project with two main activities, one for the handheld device and another for the wearable. These two activities use the same package name, which is essential for the wearable GPS to work.

8.1.1 Add Permissions and Metadata for Google Play Services

Modify the wearable manifest file to permit location service access and define Google Play Services metadata. This example gives access permission for course location only.

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION" />

 <application>
...
    <meta-data android:name="com.google.android.gms.version" android:value="@integer/google_play_
</application>
```

Add Build Dependencies

Add the following build dependencies to the wearable build.gradle file (Module:wear) in the Gradle Scripts folder, if necessary.

```
dependencies {
  compile 'com.google.android.support:wearable:1.1.0'
  compile 'com.google.android.gms:play-services-wearable:6.5.+'
  compile 'com.google.android.gms:play-services-location:6.5.+'
}
```

8.1.2 Add a Location Listener

1. Build a Google Play Services client that includes the LocationServices API.

```java
public class WearableActivity extends Activity implements
  GoogleApiClient.ConnectionCallbacks,
  GoogleApiClient.OnConnectionFailedListener
  LocationListener {

  GoogleApiClient googleClient;

  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_handheld);

  // Build a new GoogleApiClient
  googleClient = new GoogleApiClient.Builder(this)
    .addApi(LocationServices.API)
    .addConnectionCallbacks(this)
    .addOnConnectionFailedListener(this)
    .build();
  }
      // Location listener implementation (Step 2)
      ...
}
```

2. Register a location listener for the Fused Location API.

```java
// Connect to Google Play Services when the Activity starts
@Override
protected void onStart() {
      super.onStart();
  googleClient.connect();
}

    // Register as a listener when connected
@Override
public void onConnected(Bundle connectionHint) {

  // Create the LocationRequest object
  LocationRequest locationRequest = LocationRequest.create();
  // Use high accuracy
  locationRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
  // Set the update interval to 2 seconds
  locationRequest.setInterval(TimeUnit.SECONDS.toMillis(2));
  // Set the fastest update interval to 2 seconds
  locationRequest.setFastestInterval(TimeUnit.SECONDS.toMillis(2));
  // Set the minimum displacement
  locationRequest.setSmallestDisplacement(2);
```

```
    // Register listener using the LocationRequest object
    LocationServices.FusedLocationApi.requestLocationUpdates(googleClient, locationRequest, this)
  }

  // Disconnect from Google Play Services when the Activity stops
  @Override
  protected void onStop() {

    if (mGoogleApiClient.isConnected()) {
        LocationServices.FusedLocationApi.removeLocationUpdates(mGoogleApiClient, this);
        googleClient.disconnect();
    }
    super.onStop();
  }

  // Placeholders for required connection callbacks
  @Override
  public void onConnectionSuspended(int cause) { }

  @Override
  public void onConnectionFailed(ConnectionResult connectionResult) { }
```

3. Implement the LocationListener callback for location updates.

```
  @Override
  public void onLocationChanged(Location location){

      // Display the latitude and longitude in the UI
      mTextView.setText("Latitude:  " + String.valueOf( location.getLatitude()) +
                        "\nLongitude:  " + String.valueOf( location.getLongitude()));
  }
```
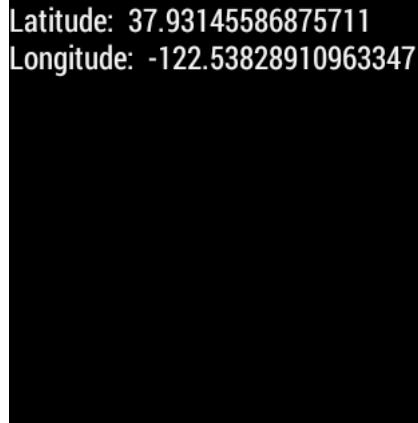
This example displays the current latitude and longitude in the wearable UI. The location is from the handheld GPS.

When the wearable is not paired with the handheld, for example when you leave the wearable behind to exercise, the GPS automatically switches to the onboard GPS.

This screen capture is from the Sony SmartWatch 3. The number of digits displayed is greater now, providing a visual clue that the wearable GPS is active. When you return and re-pair with the handheld, the GPS automatically switches back to the handheld and the display returns to its original format.

8.2 Verify GPS Sensor

This simple example works for all wearables, with or without a GPS sensor. Those without GPS must pair with a handheld to get location updates. A more complete implementation verifies the presence of a GPS sensor and pairing status before using location services, and warns users or reduces functionality when necessary. You can verify the presence of a GPS sensor as follows:

```
getPackageManager().hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS
```

8.3 Example

The working example for this section is at https://github.com/LarkspurCA/WearableGPS.

8.4 Golf Rangefinder Example

Golf is an activity where you often want to know the distance to the next hole so you can choose the perfect club for the shot. There are plenty of rangefinders on the market today, but few are as small and convenient as a smart watch. The golf rangefinder example (Clipon Caddie) is a sample application that utilizes the GPS concepts in this section to perform a useful task for golfers, to determine the current location and display the number of yards to the current hole. The full source code code is available at https://github.com/GolfMarin/CliponCaddie and the installable app is at http://cliponcaddie.com.

Contact Us

Android Wear Docs

415 924-7733

mike@androidweardocs.com

@DroidWearDocs

Indices and tables

- *genindex*
- *search*