

---

# **Android Wear Docs**

***Release 1.2***

**Michael Hahn**

**Aug 25, 2018**



---

## Contents

---

<b>1</b>	<b>How Does Android Wear Work?</b>	<b>3</b>
1.1	Set Up the Development Environment . . . . .	3
1.2	Set Up Your Handheld Device . . . . .	3
1.3	Set Up Your Wearable . . . . .	4
1.4	Next Steps . . . . .	7
<b>2</b>	<b>What About the Sample Apps?</b>	<b>9</b>
2.1	Open a Sample Project in Android Studio . . . . .	9
2.2	Launch the Sample App or Service . . . . .	10
2.3	Try the Watchface Sample . . . . .	10
2.4	Try Eliza Chat . . . . .	10
2.5	Try Recipe Assistant . . . . .	12
2.6	Try Wearable Notifications . . . . .	14
<b>3</b>	<b>Android Wear Suggest</b>	<b>15</b>
3.1	First Android Wear Suggest . . . . .	16
3.2	Example . . . . .	17
<b>4</b>	<b>Android Wear Demand</b>	<b>19</b>
4.1	First Android Wear Demand . . . . .	19
4.2	Process User Demands . . . . .	22
4.3	Try the First Demand . . . . .	23
4.4	Example . . . . .	25
<b>5</b>	<b>Wearable Application Launch</b>	<b>27</b>
5.1	Voice Activation . . . . .	27
5.2	Menu Activation . . . . .	28
5.3	Handheld Activation . . . . .	28
<b>6</b>	<b>Data Layer Messages</b>	<b>31</b>
6.1	First Wearable Message . . . . .	32
6.2	Example . . . . .	37
<b>7</b>	<b>Data Layer DataMap Objects</b>	<b>39</b>
7.1	First Wearable Data . . . . .	39
7.2	Example . . . . .	43

<b>8</b>	<b>Wearable GPS</b>	<b>45</b>
8.1	First Wearable GPS . . . . .	45
8.2	Verify GPS Sensor . . . . .	47
8.3	Example . . . . .	48
8.4	Golf Rangefinder Example . . . . .	48
<b>9</b>	<b>Always-On Wearable</b>	<b>49</b>
9.1	Enable the Always On Feature . . . . .	49
9.2	Customize the Ambient Display . . . . .	50
9.3	Try the App . . . . .	52
9.4	Example . . . . .	53
<b>10</b>	<b>Contact Us</b>	<b>55</b>
<b>11</b>	<b>Indices and tables</b>	<b>57</b>

## **What is Android Wear?**

By Michael Hahn, February 2017

Android Wear 2.0 is the latest Google software for smart watches and other wearable devices. It includes a complete redesign of the user interface and significantly expands the capabilities when the watch is not paired with the phone, even normal cellphone calls. Along with the Apple smart watch, wearables have crossed the tipping point of just being a consumer plaything to an integral part of work and play.

If you are new to Android Wear software development, you need to learn how to present timely and concise information on the small screen of a wearable, and also become familiar with the software and tools needed for application development. This site helps new developers quickly 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, August 2018

The easiest way to learn how to develop Android Wear applications is to install the Wear OS by Google Smartwatch on your handheld device, pair with an Android watch or emulator, and try out the [Android sample projects](#).

### 1.1 Set Up the Development Environment

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

1. Verify that you have a current Java JDK installed by running the following commands from a console:

```
java -version
java version "1.8.0_101"
Java(TM) SE Runtime Environment (build 1.8.0_101-b13)
Java HotSpot(TM) 64-Bit Server VM (build 25.101-b13, mixed mode)

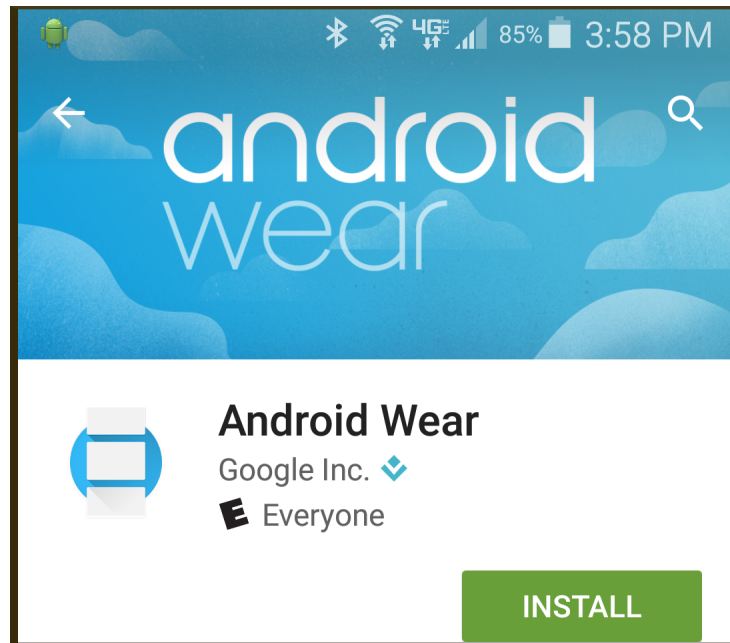
javac -version
javac 1.8.0_10
```

Both commands should display version 1.8. If not, install the Oracle Java SE Development Kit.

2. Download the [Android Studio Package](#) for your host and follow the installation instructions.
3. Start Android Studio (*android-studio/bin/studio.sh*). If you have a previous installation, you can import settings from there. Otherwise, a wizard walks you through the new installation setup procedure.
4. For convenience, add a desktop shortcut or menu item to launch Android Studio.

### 1.2 Set Up Your Handheld Device

1. Launch Google Play and Install the ‘Wear OS by Google Smartwatch’ app on your handheld device.



2. Start the Wear OS 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. Open Settings, Developer Options, USB Debugging, enable.

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,, Software Information and then click Build Number seven times. This adds Developer Options to the Options menu, so 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
```

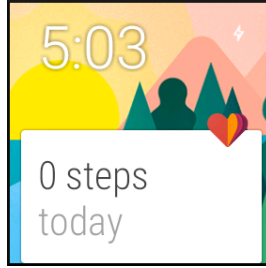
If a device is not displayed or it displays with an error such as unauthorized, you must resolve that problem before proceeding.

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.



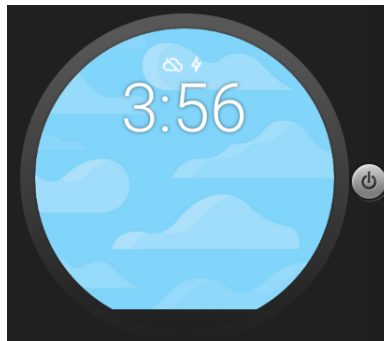


You can swipe vertically to scroll through other notifications, swipe to the right to delete the current notification, and swipe to the left to view any associated actions. The notifications displayed on the wearable are the same as those listed in the handheld, in the action bar pull down.

### 1.3.1 Android Emulator Setup

1. Start the Android AVD Manager. In Android Studio, select **Tools > Android > AVD manager**.
2. Click **Create Virtual Device** to define a new Android Wear emulator.
3. Select **Wear** in the **Category** pane and then choose a hardware profile. Click **Next**.
4. Select a system image. For example choose Nougat API Level 25 for an Android Wear 2 wearable. If necessary, click **Download** and wait for the download to complete.
5. Click **Next**. The Verify Configuration dialog is displayed.
6. Scroll down and click **Show Advanced Settings** and select **\*\*Enable Keyboard Input**.
7. Verify the configuration for the new Wearable emulator and click **Finish**.
8. Click **OK** to save your changes.
9. Click the start icon of your new emulator to launch it.

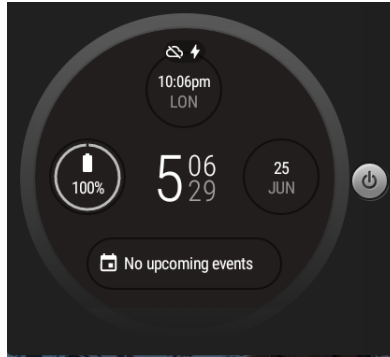
An Android Wear Version 1 emulator initially displays the time with two icons on a cloud background.



A Version 2 emulator shows the a new watch face with the time surrounded by the charge level, date, upcoming events, or other Tourbillon.

### Start an Emulator Debug Session

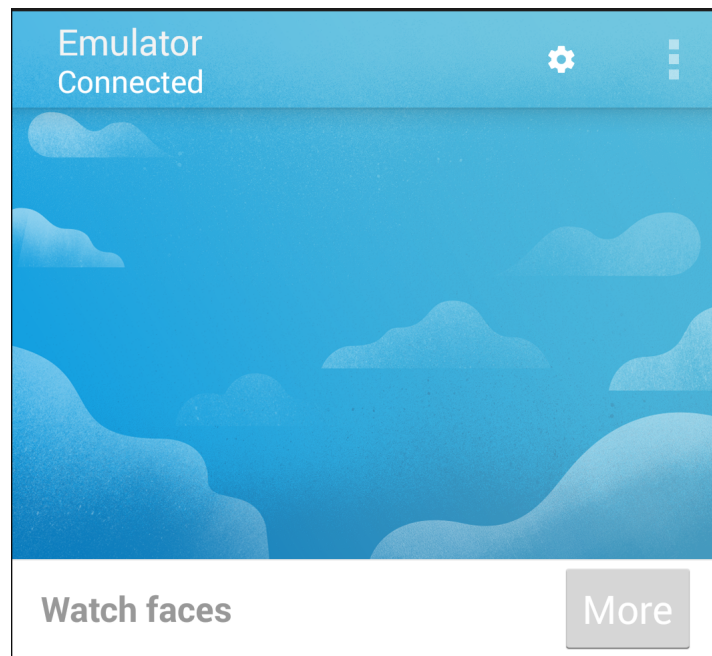
1. Enter the following command in a command window.



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

2. In the Wear OS on the handheld, select **Add a new watch** from the dropdown menu on the left.
3. In the window that opens, choose **Pair with emulator** from the dropdown menu.

When the Wear OS 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 Wear OS.

When you first pair with your wearable, the Wear OS 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. Select Settings, Developer, ADB Debugging and Debug over bluetooth.

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 Wearable Device Debug Session

1. On the handheld, open the Wear OS.
2. Scroll down to Settings and select Advanced Settings. The Settings dialog is displayed. Enable Debugging Over Bluetooth.
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
```

Note: For IPv4 hosts you can substitute `127.0.0.1` for `localhost`.

5. The Debugging Over Bluetooth setting 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 a time.

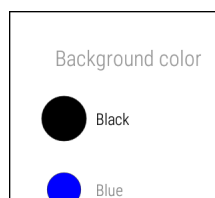
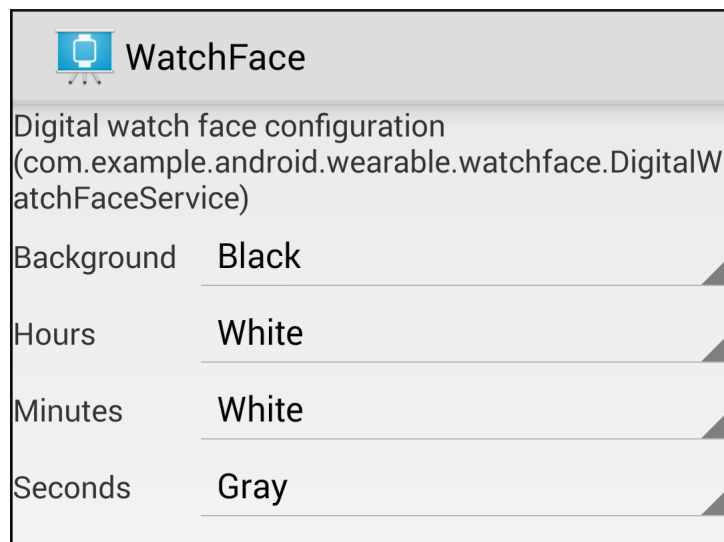
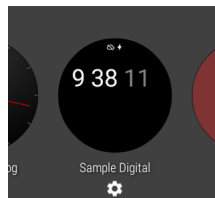
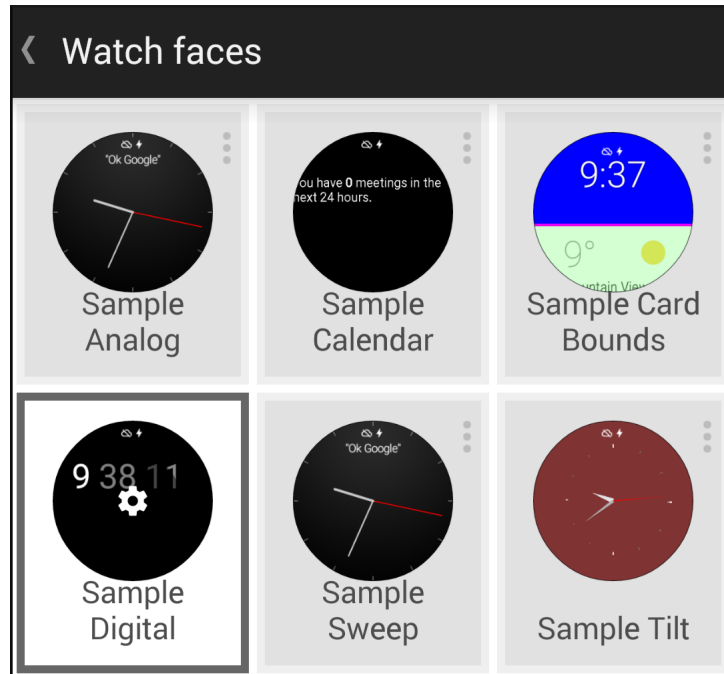
On both the handheld and wearable, watch faces display a gear icon if they have customizable settings. In the Android Wear companion app, selecting the sample digital watch face displays the following choices:

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

## 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.

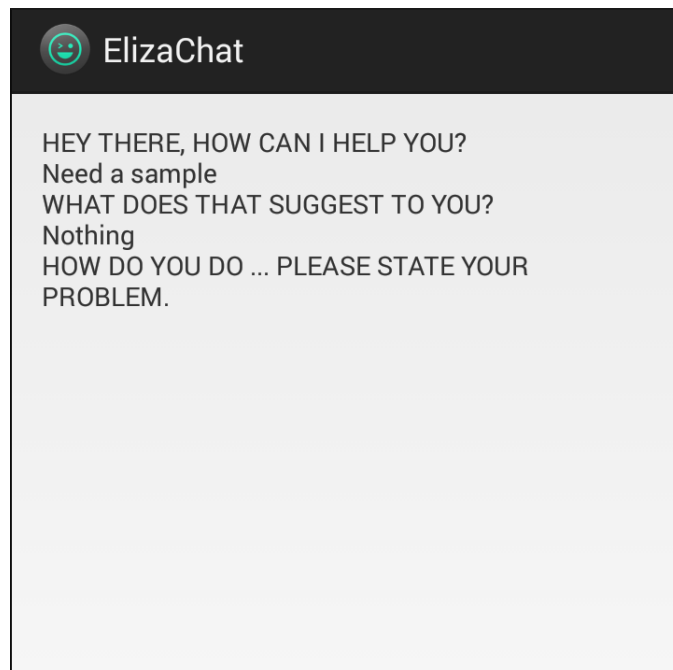
When you first launch the app, Eliza asks how she can help you.





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

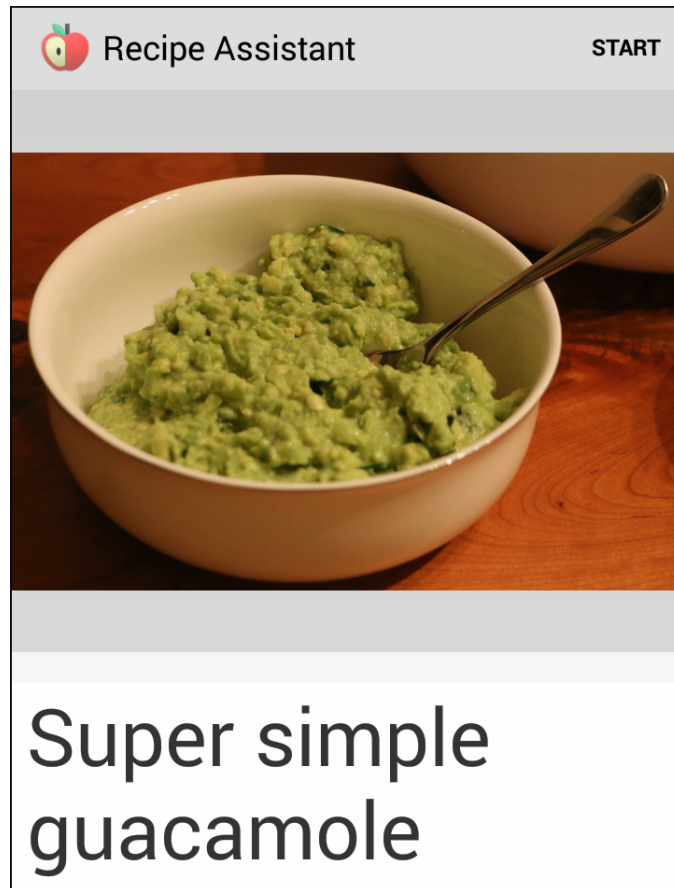
The Recipe Assistant app is an example of a fully-functional Android application that extends its capabilities to a wearable device. On the handheld device, you can scroll through the full recipe and steps. On the wearable device you view the recipe steps one at a time on separate pages.

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.

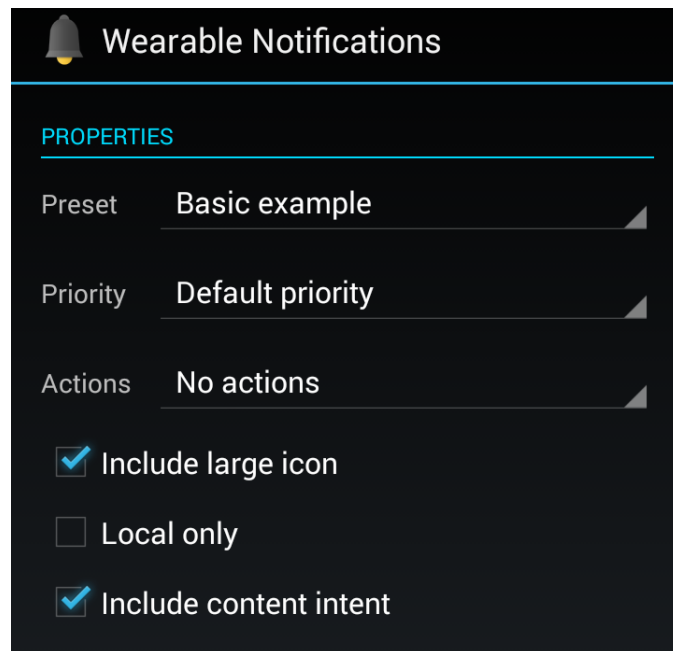
Swipe left to move through the steps. For each step you can tap to view more and swipe up or down to move through the content.





## 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.

## CHAPTER 3

---

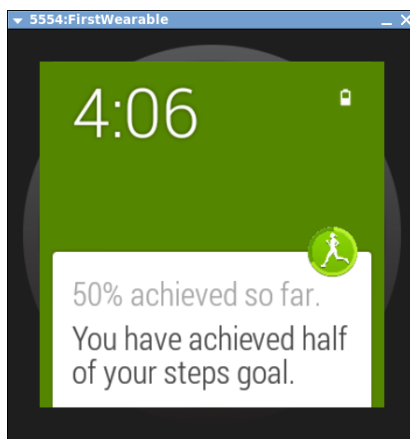
### Android Wear Suggest

---

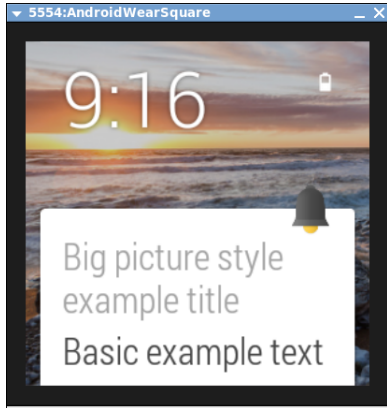
By Michael Hahn, February 2017

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

Using Android Studio, new\_wear\_app. The project name used in the example code is WearableSuggest.

### 3.1.2 Modify the Handheld Activity

Modify the onCreate method of the handheld activity as follows.

1. Optionally, add Android Wearable features to a Wearable extender object. This example adds the `HintShowBackgroundOnly` option.

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

2. 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. Create a graphic for the notification by copying the ic\_launcher.png (hdpi) icon from the mipmap folder to the drawable folder.
4. Optionally, apply a release 4.1 style to the normal notification, such as the one used in the Big Picture example (NotificationCompat.BigPictureStyle).
5. Get an instance of the Notification Manager service.

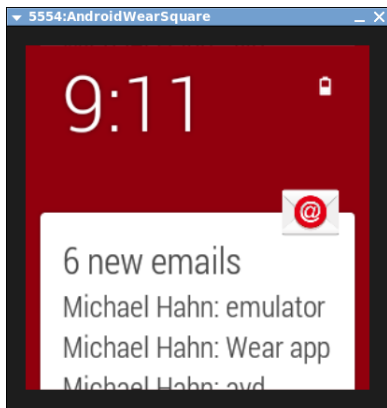
```
NotificationManagerCompat notificationManager =
    NotificationManagerCompat.from(this);
```

6. Dispatch the notification.

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

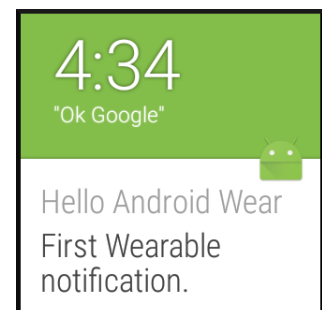
This app creates a notification that is sent to the handheld and forwarded to the wearable. If you do not see any notifications on the wearable, enable wearable notifications on the handheld. If necessary on the wearable, scroll through other notifications to view this one. The Hello World displayed on the handheld screen is part of the default layout created by the new project wizard, not the notification.

The basic Suggest functionality on a wearable is handled entirely by its default system software; no custom wearable app is required.



## 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, April 2017

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, `new_wear_app`. 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, which is located in the “mobile” branch of the project hierarchy. The wearable operating system handles the task of presenting a notification to the user and returning any demand to the handheld.

The demand process on a wearable begins when the handheld activity creates a notification that includes an action (demand). When a user views the notification on the handheld and invokes the demand, the wearable broadcasts the users demand to the handheld for processing.

The handheld code you write consists of two parts, one that sends a notification to the wearable and another that receives the received demand.

### 4.1.1 Create a Broadcast Receiver to Receive Demands from the User

When the wearable user makes a demand, the wearable broadcasts it back to the handheld. The following example shows how to receive the demand in the handheld.

1. Add a `BroadcastReceiver` class to the project. In Android Studio, select Add from the File menu and choose Java Class. Name the class `DemandIntentReceiver`, set the Superclass to `BroadcastReceiver`. paste in the following code.

```

public class DemandIntentReceiver extends BroadcastReceiver{

    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";

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

        if (intent.getAction().equals(this.ACTION_DEMAND)) {
            String message = intent.getStringExtra(this.EXTRA_MESSAGE);
            Log.v("MyTag", "Extra message from intent = " + message);
            Bundle remoteInput = RemoteInput.getResultsFromIntent(intent);
            CharSequence reply = remoteInput.getCharSequence(this.EXTRA_VOICE_REPLY);
            Log.v("MyTag", "User reply from wearable: " + reply);

            // Optional, broadcast demand string to handheld activity for display_
            onscreen.
            //
            String replyString = reply.toString();
            Intent messageIntent = new Intent();
            messageIntent.setAction(Intent.ACTION_SEND);
            messageIntent.putExtra("reply", replyString);
            LocalBroadcastManager.getInstance(context).sendBroadcast(messageIntent);
        }
    }
}

```

2. Modify the manifest (AndroidManifest.xml) to accept a broadcast from the wearable. Add the following 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>

```

### 4.1.2 Create a Notification that Includes a Demand Prompt

You build the notification with a hierarchy of objects, intent -> pending intent -> notification action -> wearable extender, and finally the notification itself. Add all the code in this section to the onCreate method of the handheld activity.

1. Create an intent.

```

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

```

2. Create a PendingIntent to include in the notification.

A PendingIntent wraps the intent to grant the privileges it needs 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. You can 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 a wearable action, adds the pending intent, and the `RemoteInput` object for voice. The `ic_reply_icon` is a graphic copied from the SDK to the handheld drawable folder.

```
NotificationCompat.Action replyAction =
    new NotificationCompat.Action.Builder(R.drawable.ic_reply_icon,
        "Reply", 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);
```

### 4.1.3 Dispatch the Notification

1. Create a notification and extended it with the wearable extender just created. The following example creates a notification that includes a reply action (demand). The `ic_launcher` is located in the `mipmap` folder; copy it to the drawable folder.

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

2. Get an instance of the Notification Manager service.

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

3. Dispatch the extended notification.

This notification is displayed in the wearable demand notifications. When the user selects this notification, scrolls to the right, selects the demand icon, and speaks the demand, the demand text is broadcast back to the handheld.

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

## 4.2 Process User Demands

User demands are received in the `onReceive` method of the broadcast receiver previously defined. They can be parsed and processed according to the application needs.

For this example the user demands are simply displayed on the handheld screen. The class that receives the demand cannot directly display the demand text onscreen because the handheld operates in a different context. A simple way to handle this is to broadcast a local message to the handheld app that contains the demand text.

### 4.2.1 Add an ID to the Handheld TextView

Open the layout file for the handheld display (`activity_handheld.xml`) and set the id of the text view to `demand_text`. At the same time you can change the default text to a user prompt, as follows.

```
<TextView
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:id="@+id/demand_text"
    android:text="Check your wearable for the demand prompt"
... />
```

### 4.2.2 Define a Local Broadcast Receiver

Define a nested `BroadcastReceiver` class within the `HandheldReceiver` class that can receive the wearable demand forwarded from the `DemandIntentReceiver`. This class extracts the demand contained in the message and displays it on the handheld display.

```
public class MessageReceiver extends BroadcastReceiver {

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

        // Display the received demand
        TextView demandView = (TextView) findViewById(R.id.demand_text);
        String demand = demandView.getText() + intent.getStringExtra("reply");
        demandView.setText(demand);
    }
}
```

### 4.2.3 Forward Received Demands to a Local Broadcast

The example `BroadcastReceiver` class already has the necessary code to forward received demands. See [Create a Broadcast Receiver to Receive Demands from the User](#).

### 4.2.4 Receive and Display the Wearable Demand

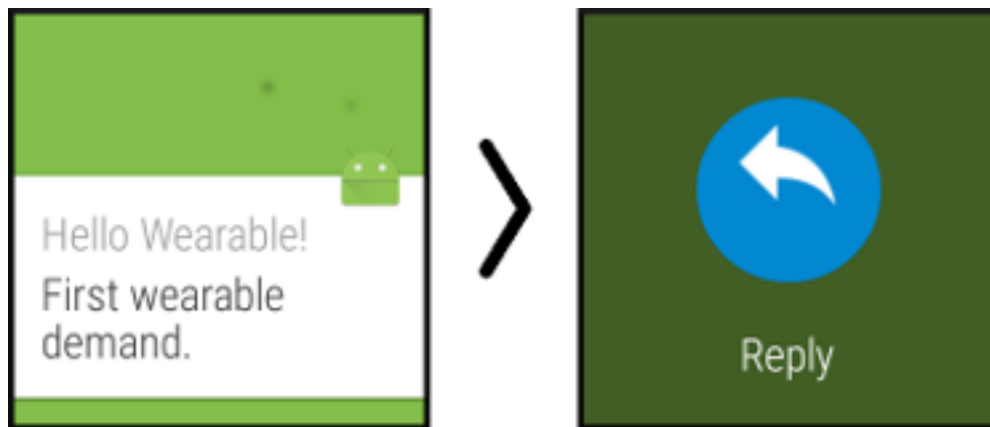
Create an instance of the local message receiver in the handheld code and register it to receive messages.

```
IntentFilter messageFilter = new IntentFilter(Intent.ACTION_SEND);
MessageReceiver messageReceiver = new MessageReceiver();
LocalBroadcastManager.getInstance(this).
    registerReceiver(messageReceiver, messageFilter);
```

## 4.3 Try the First Demand

Your new app creates and dispatches a notification that includes a demand when it first opens. Both the handheld and wearable receive and post this notification. The wearable adds it to the notification stream, so scroll through the notifications until you see the demand.

Swipe up on the demand notification to place it in full view, then swipe to the left. The Reply icon is displayed.



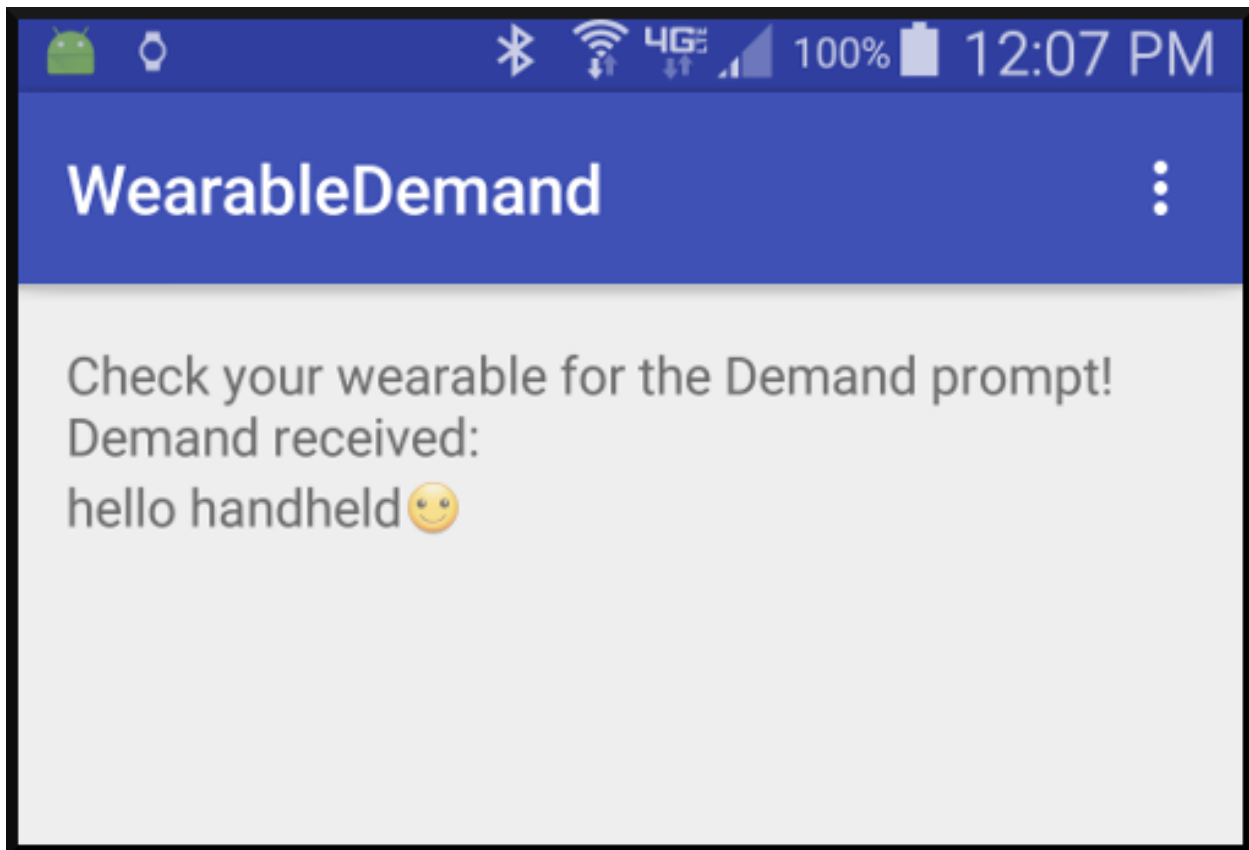
Tap the reply icon to display the voice demand or emoticon. Voice is the default, so just start speaking to enter a voice demand.



In this example the voice demand is *hello handheld*. Android converts your voice input to text and begins sending it to the handheld. You can abort this by tapping the blue Sending icon. When Android finishes sending the demand to the handheld, it returns to the Reply prompt. You could then, for example, tap Reply and then Draw Emoji.



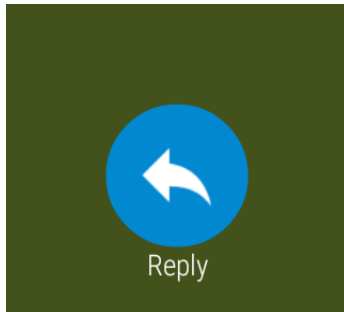
In the first prompt you can draw the emoji you want or just click the emoji icon. Either way you get a grid of emoji icons to choose from. Tap the desired emoji and Android sends it to the handheld.



Note: The emoji functionality is all implemented by the Android system software.

## 4.4 Example

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





## Wearable Application Launch

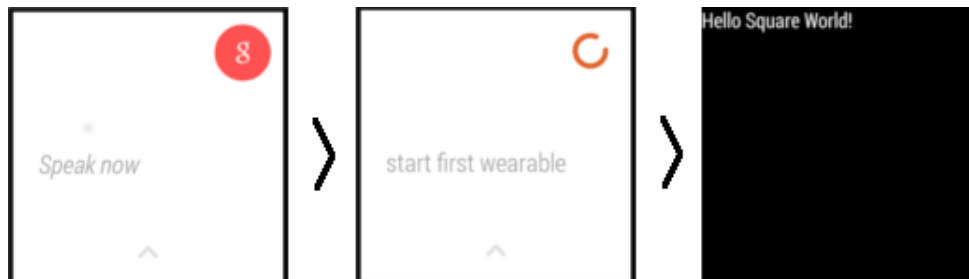
By Michael Hahn, April 2015

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.

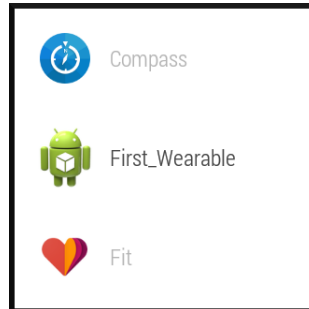
### 5.1 Voice Activation

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");
```

(continues on next page)



(continued from previous page)

```

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;
    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.

```

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

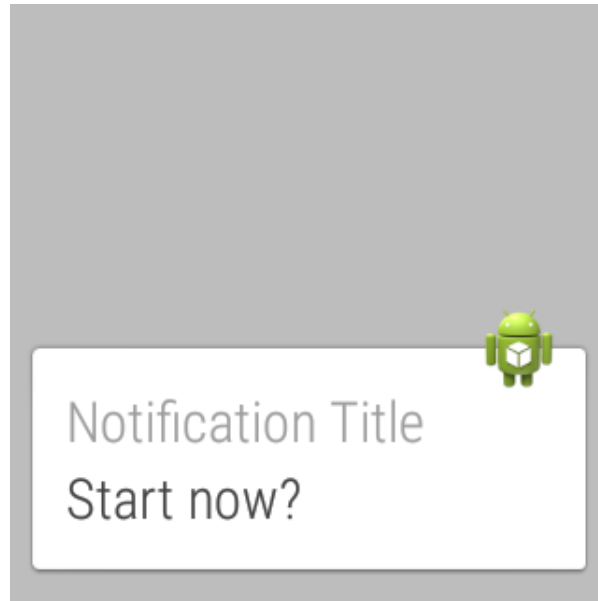
    // Create a pending intent that starts this wearable app
    Intent startIntent = new Intent(this, HoleActivity.class).setAction(Intent.
    ↪ACTION_MAIN);
    // Add extra data for app startup or initialization, if available
    startIntent.putExtra("extra", dataMap.getString("extra"));
    PendingIntent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

```
// 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
    ...
}
```

---

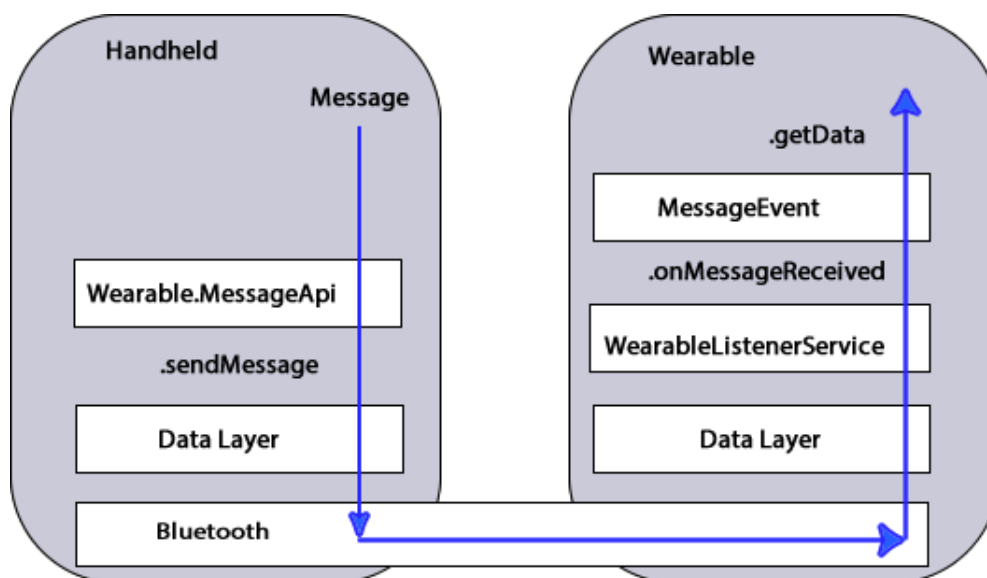
Data Layer Messages

---

By Michael Hahn, November 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

First new\_wear\_app. The new project has two applications, one for the handheld device and another for the wearable. These two applications 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;
```

(continues on next page)

(continued from previous page)

```

@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.

```

// 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.

```

class SendToDataLayerThread extends Thread {
    String path;
    String message;
}

```

(continues on next page)

(continued from previous page)

```

// 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).await();
    for (Node node : nodes.getNodes()) {
        SendMessageResult result = Wearable.MessageApi.
    ↪sendMessage(googleClient, node.getId(), path, message.getBytes()).await();
        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" />

```

(continues on next page)

(continued from previous page)

```

    </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.

```

@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`.

```

@Override
protected void onCreate(Bundle savedInstanceState) {

```

(continues on next page)

(continued from previous page)

```
// 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.

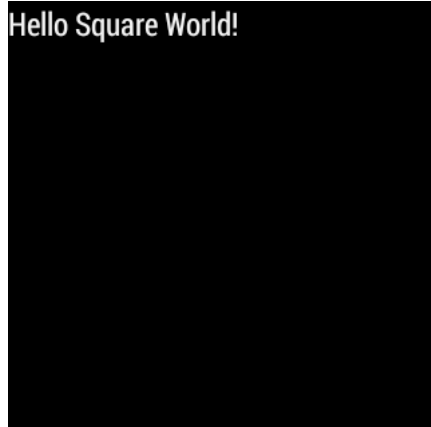
```
public class MessageReceiver extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        String message = intent.getStringExtra("message");
        // Display message in UI
        mTextView.setText(message);
    }
}
```

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, November 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 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.

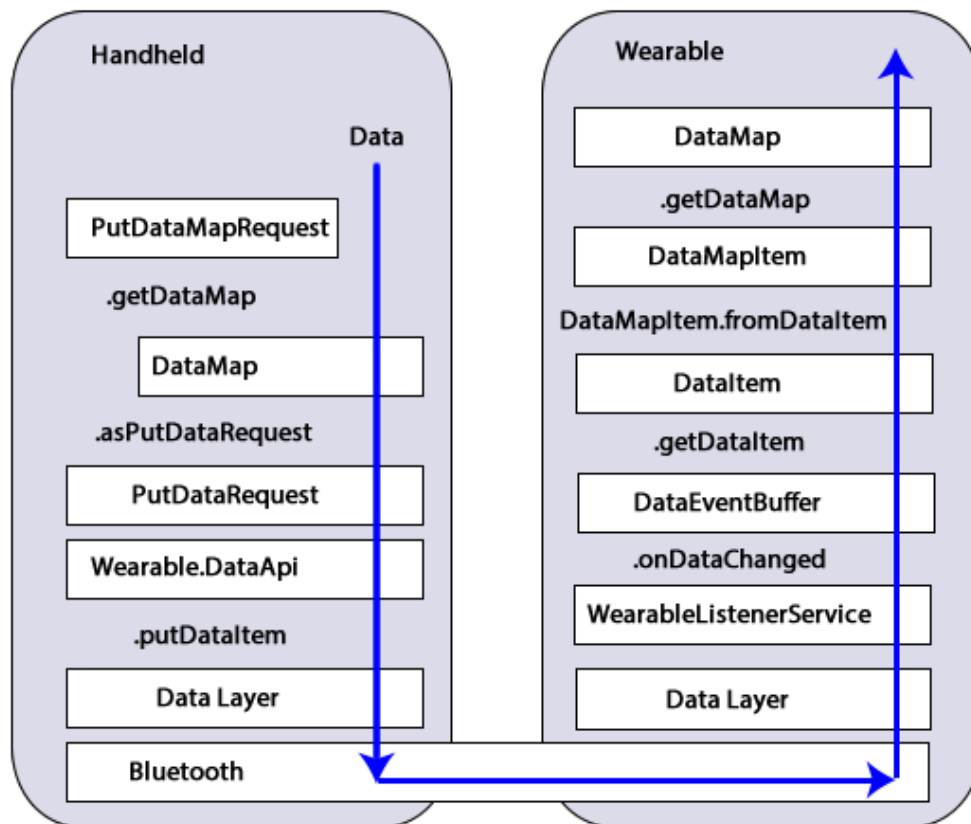
## 7.1 First Wearable Data

First new\_wear\_app. The new project has two applications, one for the handheld device and another for the wearable. These two applications 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 data sender and listener.

### 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.

```
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)
```

(continues on next page)

(continued from previous page)

```

        .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.

```

// 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.

```
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() {
        // 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 successfully to data_
        ↪layer ");
        }
        else {
            // Log an error
            Log.v("myTag", "ERROR: failed to send DataMap to data layer");
        }
    }
}
```

### 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 `onDataChanged`. 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";
```

(continues on next page)

(continued from previous page)

```
@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>.





By Michael Hahn, December 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

First new\_wear\_app. The new project has two applications, 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 access to the fine location, and define Google Play Services metadata. Update the manifest for both mobile and wear applications. This is necessary when creating an APK later.



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

```

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

    </application>

</manifest>

```

## 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:1.3.0'
    compile 'com.google.android.gms:play-services-wearable:68.3.0'
    compile 'com.google.android.gms:play-services-location:8.3.0'
}

```

### 8.1.2 Add a Location Listener

1. Extend the wearable activity for the Google Play Services location API.

```

public class MainWearActivity extends Activity implements
    GoogleApiClient.ConnectionCallbacks,
    GoogleApiClient.OnConnectionFailedListener,
    LocationListener {
    ...
}

```

2. Create a Google API Client and add the Fused Location API service.

```

GoogleApiClient googleApiClient;

@Override
protected void onStart() {
    super.onStart();
    if (googleApiClient == null) {
        googleApiClient = new GoogleApiClient.Builder(this)
            .addApi(LocationServices.API)
            .addConnectionCallbacks(this)
            .addOnConnectionFailedListener(this)
            .build();
    }
    googleApiClient.connect();
}

// Create a Location Request and 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));
}

```

(continues on next page)

(continued from previous page)

```

// 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 (googleApiClient.isConnected()) {
        LocationServices.FusedLocationApi.removeLocationUpdates(googleApiClient,
↳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.


## 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 before using location services, and warns users or reduces functionality when necessary. You can verify the presence of GPS hardware on the wearable using the following code:

```

getPackageManager().hasSystemFeature(PackageManager.FEATURE_LOCATION_GPS

```



```
Latitude: 37.93145586875711  
Longitude: -122.53828910963347
```

Even if this returns false however, the wearable can still get GPS services from the handheld as long as it is paired and the handheld GPS option is enabled.

## 8.3 Example

The working example for a Wearable GPS is at <https://github.com/LarkspurCA/WearableGPS>. It also enables the always-on feature to keep the GPS app from timing out to the watchface after a few seconds of inactivity.

## 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, displaying the number of yards from the current location to the upcoming hole. The full source code is available at <https://github.com/GolfMarin/CliponCaddie> and the installable app is at <http://cliponcaddie.com>.

---

## Always-On Wearable

---

By Michael Hahn, November 2015

Some Wearable apps are most useful when they remain in the foreground for an extended period of time. For example, a golf app needs to be available for an entire round of golf and a shopping app needs to be visible for all purchases. At the same time, these apps need to conserve power by changing to the low-power ambient mode during periods of user inactivity. The always-on feature adds this capability to Android wearable apps.

An always-on app never times out to the watchface. While a user views or interacts with the app it operates in an interactive mode that displays content in a bright full-color screen. After a few seconds on inactivity the app switches to the ambient mode, where content is simple and dimmed. A twist of the wrist or tap on the screen restores the normal interactive mode. When finished, a swipe to the right closes the app and returns to the watchface.

This section describes how to create your first always-on application.

### 9.1 Enable the Always On Feature

To create your first Always On app, `new_wear_app`. The new project has two apps, one for the handheld device and another for the wearable. The handheld app is needed only because wearable apps are always packaged within a handheld app.

#### 9.1.1 Verify the SDK Versions

Verify the SDK and build tools versions specified in the `build.gradle` (Module:mobile) and `build.gradle` (Module:wear) files.

```
android {  
    compileSdkVersion 23  
    buildToolsVersion "23.0.2"  
  
    defaultConfig {  
        applicationId "com.androidweardocs.alwayson"    }  
}
```

(continues on next page)

(continued from previous page)

```
minSdkVersion 21
targetSdkVersion 23
...
}
```

### 9.1.2 Verify Build Dependencies

Verify the dependencies specified in the build.gradle (Module:wear) file.

```
dependencies {
    compile 'com.google.android.support:wearable:1.3.0'
    provided 'com.google.android.wearable:wearable:1.0.0'

    compile 'com.google.android.gms:play-services-wearable:8.3.0'
}
```

### 9.1.3 Add WAKE\_LOCK Permission and Uses-Library to the Manifest

Add the WAKE\_LOCK permission at the top level of the manifest. To run this app on a pre-22 APK, you set the wearable library requirement to false in the application section of the manifest.

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.androidweardocs.alwayson" >
    <uses-permission android:name="android.permission.WAKE_LOCK" />
    <application
        <uses-library android:name="com.google.android.wearable"
        ↪android:required="false" />
        '''
    </application>
</manifest>
```

### 9.1.4 Create a WearableActivity Class

To enable the Always On feature, extend your main activity from WearableActivity, and invoke setAmbientEnabled in the onCreate method.

```
public class AlwaysOn extends WearableActivity {
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_always_on);
        setAmbientEnabled();
        ...
    }
}
```

## 9.2 Customize the Ambient Display

When the wearable enters ambient mode you change the display to a minimum power theme. Informational content should be changed to white or grey content on a black background. Controls are hidden. On return to the interactive

mode, you reverse the changes to display everything with full colors and brightness.

### 9.2.1 Handle Ambient Mode Transitions

Handle entry and exit from the ambient mode by implementing `onEnterAmbient` and `onExitAmbient` respectively. The following entry example changes the background to black and the text to white with antialias disabled.

```
@Override
public void onEnterAmbient(Bundle ambientDetails) {
    super.onEnterAmbient(ambientDetails);
    mTextView.setBackgroundColor(Color.BLACK);
    mTextView.getPaint().setAntiAlias(false);
    mTextView.setTextColor(Color.WHITE);
}
```

On return to interactive mode reverse all your ambient mode changes.

```
@Override
public void onExitAmbient() {
    mTextView.setBackgroundColor(Color.CYAN);
    mTextView.getPaint().setAntiAlias(true);
    mTextView.setTextColor(Color.BLACK);
    super.onExitAmbient();
}
```

### 9.2.2 Update Content in Ambient Mode

In the ambient mode you refresh the display with current content by implementing `onUpdateAmbient`. This method is called every 60 seconds. For more frequent updates you can respond to incoming data, use the alarm manager, or implement a `Handler`. To benefit from the power-saving capabilities of the ambient mode, keep the update interval greater than 10 sec.

This simple example appends a number to the hello message, which it increments every 60 seconds.

```
@Override
public void onUpdateAmbient() {
    super.onUpdateAmbient();
    mTextView.setText("Hello Square World! " + i.toString());
    i++;
}
```

### 9.2.3 First App Display Setup

The default theme for the wearable displays white text on a black background, which is a typical ambient mode display. To try out the always-on display features, modify the wear layout to specify black text on a cyan background, and set the width and height of the text view to fill the parent:

```
<TextView
    android:id="@+id/text"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="@string/hello_square"
    android:textColor="@color/activeText"
```

(continues on next page)

(continued from previous page)

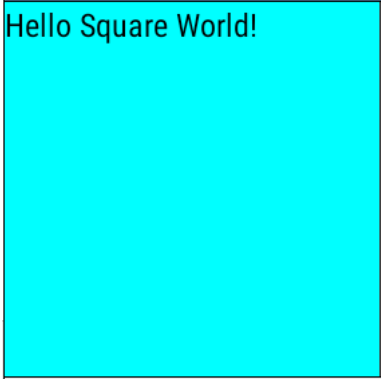
```
android:background="@color/activeBackground"  
/>
```

Define the new colors in a resource file, colors.xml:

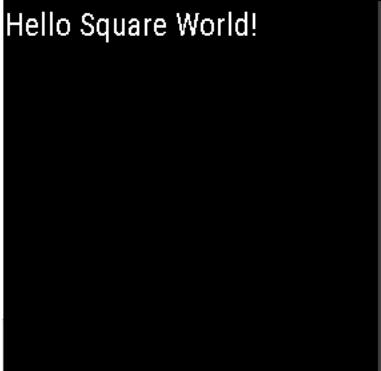
```
<resources>  
  <item name="activeBackground" type="color">#FF00FFFF</item>  
  <item name="activeText" type="color">#FF000000</item>  
</resources>
```

## 9.3 Try the App

Verify the app on an emulator or device (see *Set Up Your Wearable*). You need Lollipop 5.1 (API 21) as a minimum to use Always On. To start the app, when select Run -> Wear from the Run menu on Android Studio. The interactive screen displays the Hello message in black text on a cyan background.

A square screenshot of a watch face. The background is a solid cyan color. In the top-left corner, the text "Hello Square World!" is displayed in a black, sans-serif font.

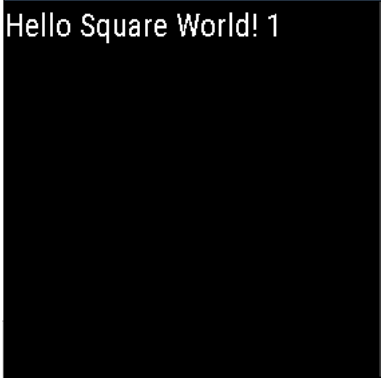
When the watch enters the ambient mode, the text color changes to white and the background to black. Had there been any buttons or controls, they would have been hidden.

A square screenshot of a watch face. The background is a solid black color. In the top-left corner, the text "Hello Square World!" is displayed in a white, sans-serif font.

After one minute the ambient display is modified, in this example to include a number.

To stop this app and return to the watchface, tap screen to enter the interactive mode and swipe to the right.





Hello Square World! 1

## 9.4 Example

The working example for this section is at <https://github.com/LarkspurCA/WearableAlwaysOn>.



## CHAPTER 10

---

### Contact Us

---

Android Wear Docs

415 924-7733

[mike@androidweardocs.com](mailto:mike@androidweardocs.com)

@DroidWearDocs



# CHAPTER 11

---

## Indices and tables

---

- `genindex`
- `search`