Extracted from:

Programming Google Glass, Second Edition

Build Great Glassware Apps with the Mirror API and GDK

This PDF file contains pages extracted from *Programming Google Glass, Second Edition*, published by the Pragmatic Bookshelf. For more information or to purchase a paperback or PDF copy, please visit http://www.pragprog.com.

Note: This extract contains some colored text (particularly in code listing). This is available only in online versions of the books. The printed versions are black and white. Pagination might vary between the online and printed versions; the content is otherwise identical.

Copyright © 2014 The Pragmatic Programmers, LLC.

All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Programming Google Glass

Second Edition

Build Great Glassware Apps with the Mirror API and GDK

Eric Redmond

Edited by Jacquelyn Carter

Programming Google Glass, Second Edition

Build Great Glassware Apps with the Mirror API and GDK

Eric Redmond

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in this book, and The Pragmatic Programmers, LLC was aware of a trademark claim, the designations have been printed in initial capital letters or in all capitals. The Pragmatic Starter Kit, The Pragmatic Programmer, Pragmatic Programming, Pragmatic Bookshelf, PragProg and the linking *g* device are trademarks of The Pragmatic Programmers, LLC.

Every precaution was taken in the preparation of this book. However, the publisher assumes no responsibility for errors or omissions, or for damages that may result from the use of information (including program listings) contained herein.

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Copyright © 2014 The Pragmatic Programmers, LLC.
All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form, or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior consent of the publisher.

Printed in the United States of America.
ISBN-13: 978-1-941222-18-8
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—October 22, 2014

Most of the previous chapter applied to Android applications in general, but now we're going to cover a distinctly Glass feature. If you describe Google Glass to most people, images of an endless timeline of static cards don't generally enter their minds. Instead, folks imagine something more interactive—applications that respond in real time. The Mirror API we covered in Part One was entirely about generating static cards in the timeline, but if we desire a timeline card to be more interactive, then Live Cards are the answer.

In this chapter we'll create an app that displays technical information about your Glass device, such as wifi strength, locale information, and battery temperature, all updated live in the timeline. This Live Card will change in real time when any values change. When making this app, we'll design a complex card layout using an XML configuration, and update the values it displays using both a Service and a Broadcast Receiver. Furthermore, we'll include a menu overlay, rendered by an Activity.

Planning a Live Card Project

In the last chapter we made an app that displayed a message to a user whenever our Glass device was plugged or unplugged. Although it was a decent example of implementing a Broadcast Receiver, it was hardly a good Glassware design. The message is short, exists outside of any normal Glass card, and it's sort of awkward to require that a user keep Glass on their face as they plug and unplug the USB.

A better design would be a single, easy to access card that changes state based on whether Glass is plugged in or not. This is where a Live Card comes in.

A Live Card application has a different lifecycle than an Activity. When we launch an Activity, it takes over the screen, and when we close the app it ceases to function. But as long as an Activity runs we could update its View on the screen indefinitely.

On the other hand, static cards created via the Mirror API are part of the timeline, but they rarely change. At least, they aren't real time.

But what if we want the best of both worlds—an application that runs as a card in the timeline, but with a view that we can update live whenever it suits us? Such a card would be most useful near the home card, where it's easy to access, rather than buried somewhere in the timeline's past.

• Click HERE to purchase this book now. discuss

Live Card Lifecycle

When a Live Card application is launched, rather than living in the timeline to the right of the home card, it lives immediately to the left, as you can see in the following figure. This makes it easy to launch, look at, and return to again whenever you want to see how the card has been updated.

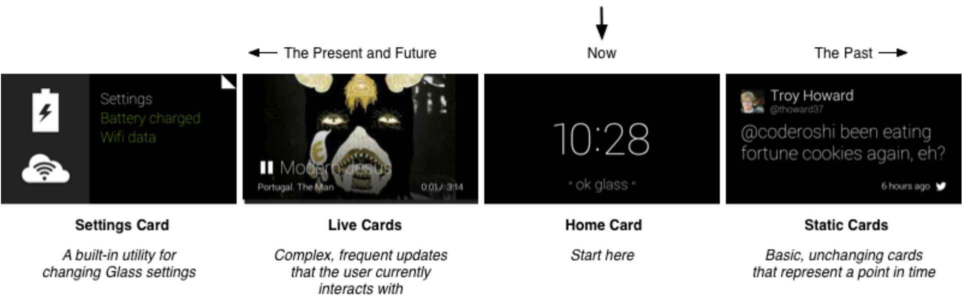


Figure 39—The Live Card is Left of the Home Card

This is a useful attribute for a range of applications, and is the most common type of Glassware you're likely to create. You get the power and benefits of an Activity, with the convenience of existing in the Glass timeline. Unlike static cards that always represent a point in history, a Live Card always represents now. Hence, its exalted place as the left hand to the almighty home card.

A LiveCard is not a View in itself. The LiveCard class is a bridge between a regular Android View and Glass's management of that View. This is important to keep in mind as we explore our first project.

Since a LiveCard can remain active even if a user switches to a different timeline card, it needs to be kept alive by a Service. This Service is responsible for creating the LiveCard and acting as its Context.

An important concept to understand about LiveCards is that they are mere containers. A LiveCard needs a View to show, just as does an Android Activity. There is an easy way to render a display on a LiveCard, as well as two hard ways. We'll start with the easiest, which is RemoteViews.[1] We'll cover a harder method in Chapter 12, *Advanced Rendering and Navigation*, on page ? using a screen callback object.

1. http://developer.android.com/reference/android/widget/RemoteViews.html

RemoteViews is a pretty easy concept, but is unlike the Views we created before. It's an object containing instructions for building a View. You generally define this RemoteViews object in one process, but it is executed remotely on another process.

This is important because the Glass timeline is its own Android process, distinct from our application. So we create a RemoteViews object in our Service, then populate it with an XML-defined layout and whatever view values we'll want to see. These could be text view values, or colors, or images, or a dozen other views. When the LiveCard is ready to be published, the remote layout information is rendered as an active view hierarchy on the Glass timeline process.

With this basic overview of LiveCards and RemoteViews, we're ready to go through the *Stats* project.

Collecting Android Stats

The project we're going to investigate in this chapter creates a LiveCard that displays Glass device statistics, named *Stats*. The information we've chosen to display are: battery power level, battery voltage, battery temperature, the Glass device's locale language and country, current time, whether the USB cable is plugged in, and an icon that represents wifi strength. When we're done, your card looks something like this.

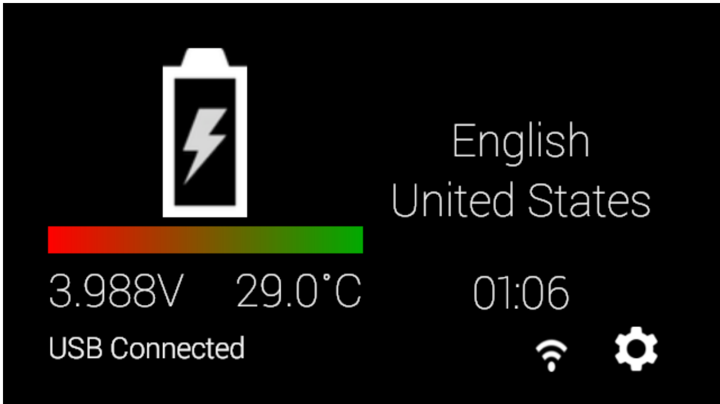


Figure 40—Our Live Card Stats Screen

Note that these values are retrieved from various sources, either from extracting language and time information from the device Locale object…

```
// Get the device's locale
```

```
Configuration config = getResources().getConfiguration();
String language = config.locale.getDisplayLanguage();
```

…or getting values from Global settings…

```
// Check if wifi is turned on
ContentResolver cr = context.getContentResolver();
int wifiOn = Settings.Global.getInt(cr, Settings.Global.WIFI_ON);
```

…or requesting information from a System Service…

```
// Get wifi signal strength
WifiManager wifiManager =
  (WifiManager)context.getSystemService(Context.WIFI_SERVICE);
int rssi = wifiManager.getConnectionInfo().getRssi();
int strength = WifiManager.calculateSignalLevel(rssi, 4) + 1;
```

…to the most common method, which we'll employ. We register a Broadcast Receiver to listen for relevant Intents, like the change in a battery's state, and extract extra data from the Intent object.

```
// EXTRA_TEMPERATURE is the battery temperature in tenths of a degree C
int temp = intent.getIntExtra(BatteryManager.EXTRA_TEMPERATURE, 0) / 10;
```

We place the code to gather this system's information into a helper class named StatsUtil. As an example, the getBatteryLevel method works like this.

chapter-11/Stats/src/glass/stats/StatsUtil.java

```
public static int getBatteryLevel( Intent intent ) {
  // EXTRA_SCALE gives EXTRA_LEVEL a maximum battery level
  int scale = intent.getIntExtra(BatteryManager.EXTRA_SCALE, 100);
  float level = (float)intent.getIntExtra(BatteryManager.EXTRA_LEVEL, 0) / scale;
  Log.d("StatsUtil", "power: " + (int)level * 100);
  return (int)level * 100;
}
```

We won't spend any more time on the details of these methods. Feel free to read the code if you're interested—there are many types of information stored in an Android system, and they're well documented on the Android developer website.[2] What's of interest for our Glassware development is how you decide when to change the information (BroadcastReceiver) and how to display that information (RemoteViews plus view layouts). Those are points we'll dig deeper into for our *Stats* project.

2. http://developer.android.com/guide/

Implementing a Live Card Glassware

With the basic ideas and tools out of the way, it's time to take a look at our *Stats* project. Since we know we want to create a Live Card, we need a Service to launch and manage the LiveCard. Our project's Service is called the StatsService component.

Similar to what we saw with Activities in Chapter 10, *An Android Introduction on Glass*, on page ?, the Service object also follows a lifecycle. Ours will begin with onStartCommand(). Services end with onDestroy(), which we'll also implement. Technically, we also will need to implement onBind() because it's required by the abstract parent class. We'll use this method in Chapter 12, *Advanced Rendering and Navigation*, on page ?, but let's pretend like it doesn't exist, for now.

We only want one LiveCard for this Service. It's generally considered bad form to create more than one LiveCard per Service, so if no liveCard exists, we make a new one with a unique tag name. Rather than thinking too much about a good unique name, it's just as easy to name it after our service class name.

chapter-11/Stats/src/glass/stats/StatsService.java

```
public final static String TAG = StatsService.class.getName();

public int onStartCommand(Intent intent, int flags, int startId) {
  if( liveCard == null ) {
    liveCard = new LiveCard( this, TAG );
    liveCard.setViews( remoteViews() );
    liveCard.setAction( buildAction() );
    liveCard.publish( PublishMode.REVEAL );
  }
  return START_STICKY;
}
```

With our LiveCard object in hand, we set some values (we'll cover that in a moment) and publish the Card. Setting the PublishMode.REVEAL means that once this card's views are loaded, show the user this card. The other option is PublishMode.SILENT, which will load the LiveCard in the background without interrupting the user. We return START_STICKY, which means that this Service should continue running in the background until we explicitly stop it.

Destroying a LiveCard is simple—just be sure to unpublish the object if it's still published, and remove the Service field reference.

chapter-11/Stats/src/glass/stats/StatsService.java

```
public void onDestroy() {
  if( liveCard != null && liveCard.isPublished() ) {
    liveCard.unpublish();
```

```
    liveCard = null;
  }
  super.onDestroy();
}
```

Let's go back and look at the most interesting LiveCard method called here, setViews(), and the RemoteViews object it receives.

RemoteViews and View Layouts

In the section Live Card Lifecycle on page 4 we discussed the general structure of how StatsService manages the LiveCard, and how the LiveCard object accepts RemoteViews. But we glossed over how this object somehow builds a real View.

Creating a RemoteViews object requires some information about the view it will eventually build. In this chapter, we're describing how to construct a view defining an XML Layout.

You can create a RemoteViews in a few ways, but the simplest is to pass in the package name of the Android project—the package field in the AndroidManifest.xml—and a layout id generated in the R class. The R.layout.stats id represents an XML-defined layout file that resides in res/layout/stats.xml.

chapter-11/Stats/src/glass/stats/StatsService.java

```
private RemoteViews remoteViews() {
  rv = new RemoteViews(getPackageName(), R.layout.stats);
  rv.setTextViewText(R.id.time, StatsUtil.getCurrentTime(this));
  rv.setTextViewText(R.id.connected, StatsUtil.getConnectedString(this));
  Configuration config = getResources().getConfiguration();
  rv.setTextViewText(R.id.language, config.locale.getDisplayLanguage());
  rv.setTextViewText(R.id.country, config.locale.getDisplayCountry());
  return rv;
}
```

This is a View that starts with a RelativeLayout that fills the page. Within that parent view are views: two relative layouts acting as left and right columns, and a linear layout acting as the Card's footer bar. Within this layout container view hierarchy are the elements that render as we saw in Figure 40, *Our Live Card Stats Screen*, on page 5.

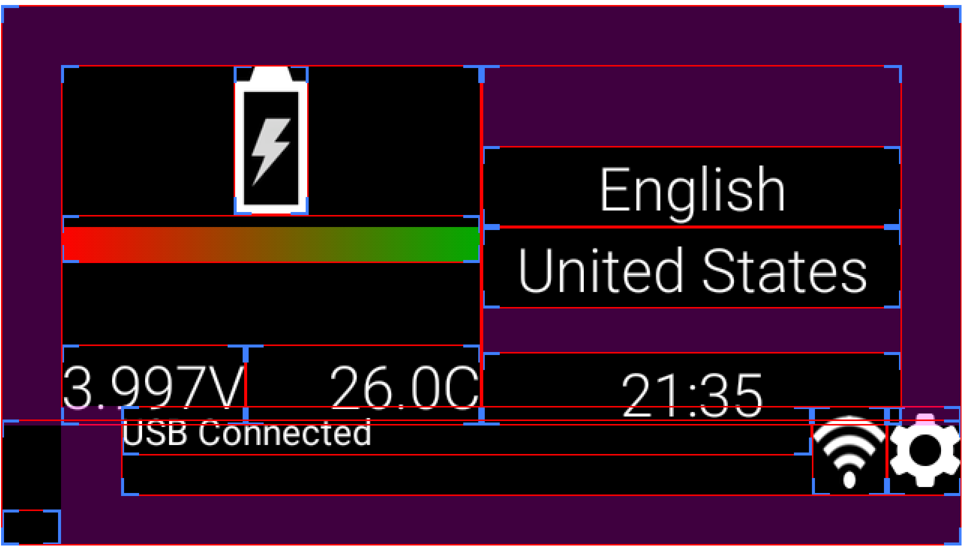
In the left column is a progress bar containing the current battery's charge level, text views for battery voltage and degrees, and a big battery image. The right column contains the device language, county, and current time, all as text views. Finally, the footer contains a message about whether the USB cable is connected, an icon for wifi strength, and a little icon on the left. That final icon is a helpful Glass UI standard, so that a user can immediately see what application they're running, but it's not required.

The following XML is a truncated version of the real stats layout XML, which can easily be pages long when you start including fields like android:layout_marginLeft.

interactive/stats.xml

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android">
  <RelativeLayout android:id="@+id/left_column">
    <ProgressBar android:id="@+id/battery_level"
        android:max="100"
        android:progress="50"
        android:progressDrawable="@drawable/battery_level" />
    <TextView android:id="@+id/battery_voltage"/>
    <TextView android:id="@+id/battery_degrees"/>
    <ImageView android:id="@+id/imageView1"
        android:src="@drawable/img_battery" />
  </RelativeLayout>
  <RelativeLayout android:id="@+id/right_column">
    <TextView android:id="@+id/language"
        android:textAppearance="?android:attr/textAppearanceMedium" />
    <TextView android:id="@+id/country"
        android:textAppearance="?android:attr/textAppearanceMedium" />
    <TextView
        android:id="@+id/time" />
  </RelativeLayout>
  <LinearLayout android:id="@+id/footer_container"
       android:orientation="horizontal">
    <TextView android:id="@+id/connected"
        android:textAppearance="?android:attr/textAppearanceSmall" />
    <ImageView android:id="@+id/wifi_strength"
        android:src="@drawable/ic_wifi_1" />
    <ImageView android:id="@+id/stats_icon_view"
        android:src="@drawable/ic_gear_50" />
  </LinearLayout>
</RelativeLayout>
```

In your Glass settings, you can go to *Developer settings* and set *Show layout bounds and margins* to *ON*. When our layout is ready to be rendered, it will display with all of the margins outlined, like the following.



Something to note about about our XML layout is that each element has an android:id field. The +@id/ prefix informs the Android development tools that it wants to generate this ID in th R class. This is convenient, so we can easily reference this layout in our code.

If you refer back to the remoteViews() method at the beginning of this section, you can see that we repeatedly call a method on the RemoteViews object, namely setTextViewText, containing a view id like R.id.time, and a value like StatsUtil.getCurrentTime(this). This is how we populate our LiveCard view with values.

Just like our generated R class in the last chapter, ADT will automatically generate R.layout.stats to reflect our new layout. Creating our RemoteViews is a simple matter of pointing it at our app's package and stats value.

You've done your job building the RemoteViews and associated layout data. Now it's up to Android and the Glass timeline to put these all together into a real rendered View when the time is right. We haven't yet finished populating our Glassware, but let's take a break from the RemoteViews for now. There's an important action we've overlooked.