The Pragmatic Programmers

Programming Google Glass

Second Edition

Build Great Glassware Apps with the Mirror API and GDK

Eric Redmond

Edited by Jacquelyn Carter

Under Construction: The book you're reading is still under development. As part of our Beta book program, we're releasing this copy well before a normal book would be released. That way you're able to get this content a couple of months before it's available in finished form, and we'll get feedback to make the book even better. The idea is that everyone wins!

Be warned: The book has not had a full technical edit, so it will contain errors. It has not been copyedited, so it will be full of typos, spelling mistakes, and the occasional creative piece of grammar. And there's been no effort spent doing layout, so you'll find bad page breaks, over-long code lines, incorrect hyphenation, and all the other ugly things that you wouldn't expect to see in a finished book. It also doesn't have an index. We can't be held liable if you use this book to try to create a spiffy application and you somehow end up with a strangely shaped farm implement instead. Despite all this, we think you'll enjoy it!

Download Updates: Throughout this process you'll be able to get updated ebooks from your account at *pragprog.com/my_account*. When the book is complete, you'll get the final version (and subsequent updates) from the same address.

Send us your feedback: In the meantime, we'd appreciate you sending us your feedback on this book at *pragprog.com/titles/erpgg2/errata*, or by using the links at the bottom of each page.

Thank you for being part of the Pragmatic community!

Andy & Dave

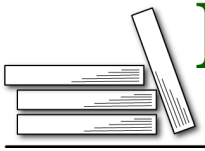
Programming Google Glass, Second Edition

Build Great Glassware Apps with the Mirror API and GDK

Eric Redmond

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



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 *https://pragprog.com*.

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

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: P1.0—February 13, 2015

Contents

Changes in the Beta Releases

P1.0, Feb 13, 2015

- Replaced all ADT examples in Part Two with Android Studio.
- Due to Google's recent announcement that the Glass Explorer program has ended, thus suspending any further purchase of Google Glass, we are following suit and suspending production of our book, Programming Google Glass. Thanks to everyone who took the leap of faith with us and reviewed or purchased the beta.

Beta 3, Dec 15

- Added the final chapter, Chapter 16, *Turning an Android App to Glassware*, on page 265.

Beta 2, November 13

- Added Chapter 15, *Preparing For the Real World*, on page 245.
- Updated with XE changes and added a few minor clarifications

Beta 1, October 21

Preface

Google Glass is the new wearable computer that everyone is talking about. Some people love it, some hate it, but folks can't seem to help but talk about it. In short, Glass is a head-mounted computer with an optical display, a camera, several sensors, and a voice or touch interface. You can walk around, hands-free, check your email, watch or take videos, play games, and most anything else you can do with a smartphone. Equally important, it's gaining traction for enterprise purposes, from fast-food restaurant employee training to use in surgical settings.

You can program Glassware (Glass applications) in two ways: by using the HTTP-based Mirror API web-service, or by creating native applications using the Glass Development Kit (GDK). This book covers how to program both.

You'll get a glimpse of what Glass is and what it is not, and how users can interface with Glass. In Part One of this book you'll learn how to develop a Glass application fast, by using the Mirror API to manipulate timeline cards and menus, track a Glass's geolocation, create rich interactions by responding to user inputs, and capture or serve user images and videos. In Part Two you'll learn how to shape user experience with the GDK by interacting with Glass hardware, from voice-to-text inputs, to QR code reading with the live camera, to building your own video game with fine-grained sensor inputs. You'll see how to properly design new Glassware or update existing applications to become Glassware. This is *the* book to read if you want a shortcut to this brave new world.

What's the Big Deal with Glass?

Imagine a world where computers did not exist, but the abilities that computers provided did. Rather than pulling a phone out of your pocket to talk with someone, you'd simply speak their name and you'd be connected. Or instead of taking out a laptop to play a video game, you would merely ask to play a game and it would appear. Or rather than sitting in front of a television screen

to watch your favorite movies, a panel would hover conveniently in the air visible only to you, or sharable with friends who wish to watch as well. The idea of Google Glass is not to add more technology to your daily life, but rather sit idly in the background, available when you need it. It's about getting technology out of your way, while still providing its benefits.

The first future-facing movie that I can recall containing consumer HUD (heads-up display) goggles was *Back to the Future 2*. This HUD was worn in the *future* year 2015 (I know, right?), not by a military commander or an airship pilot, but by young Marty McFly, Jr., as he sat with his family around the kitchen table. This was a consumer device capable of, at least, displaying text and accepting phone calls.

Although Glass is sometimes considered to be an augmented-reality device, it's better thought of as an ever-present optical interface for a mobile device. It's a self-contained computer, yes, but it is also most useful when paired with an external paired smartphone for some actions, such as geolocation or continuous Internet access. Glass is sometimes refered to as having the power of a smartphone available, without the inconvenience of digging in your pocket.

Is This Book for You?

This book is designed to help experienced developers quickly start writing Google Glass applications with the Mirror API. Although this book covers using the interface with Google's Java Mirror Client code, the Mirror API itself is an HTTP interface with support for many languages. This means that many of the lessons about the Mirror API itself can apply to languages beyond the simple Java client.

The pertinent code is covered in the book, and the rest can be downloaded along with the book (or from GitHub.[1])

You needn't be a Java expert to follow Part One of this book, but it can help to know your way around the syntax and Eclipse editor. You may also get more out of Part One if you're familiar with Google App Engine, although you can use any Platform as a Service (PaaS) or host your own Glassware applications. Part Two requires a much more in depth knowledge of Java, and there we'll be using The Intellij-based Android Studio.

1. https://github.com/coderoshi/glass

What's in This Book?

This book is intended to be read sequentially, from the first to last chapter. It covers most of the Mirror API and the GDK, but it's not designed to be a reference book. There are plenty of online documents[2] for more detailed reference material.

Part One

After an introduction to Glass and two styles of programming Glassware in Chapter 1, *Wrapping Your Head Around Glass*, on page 1, we will dive into the Mirror API web service. In Part One we work on slowly building up a complete Glassware, along the way using most of the components of the Mirror API. Although I'd recommend you read both the Mirror API and GDK parts to get a full appreciation of your Glassware options, you are free to skip straight to Part Two if you have no interest in the Mirror API service.

Chapter 2, *The Google App Engine PaaS*, on page 13

Here we set up our Mirror API development and deployment environments. Since the Mirror API must run as a web service, we need a web provider to host the application. We chose Google App Engine, just to keep it in the Google family. The Glassware we're creating is called Lunch Roulette, which randomly chooses a lunch option for you.

Chapter 3, *Authorizing Your Glassware*, on page 27

Using the Mirror API also requires that end users of your Glassware authorize your application to interoperate with their Glass. Here we follow the OAuth 2.0 steps necessary to authorize our application. OAuth can be complex, so we cover the minimum amount necessary to keep you moving.

Chapter 4, *Building the Timeline*, on page 41

Finally the wait is over—we dig into the actual Mirror API, starting with the Glass Timeline. We add and remove timeline items, and add custom menu options to Lunch Roulette.

Chapter 5, *Tracking Movement and User Responses*, on page 59

Learn how to subscribe to notifications of Timeline changes, such as when a user deletes a card, or chooses a custom menu option. We'll also track user movement by Geolocation. We'll leverage notifications to expand Lunch Roulette to choose a restaraunt currently close to the Glass user.

2. https://developers.google.com/glass/develop/

Chapter 6, *Making Glass Social*, on page 75

Here we finish up Lunch Roulette by adding some social aspects, such as calling the chosen restaraunt to make reservations, or sharing images with your Contacts, and the Lunch Roulette application. We also look into attaching assets, such as an image, to a user's timeline.

Chapter 7, *Designing for Glass*, on page 87

One of the most important aspects of learning Glass is how it changes the world of application design. With its smaller screen, lack of a touchscreen or keyboard, and alternative input options, you have to approach UX design differently.

Chapter 8, *Turning a Web App to Glassware*, on page 99

If you have an existing web application, this chapter is a useful exercise. We take an existing blog application, and build Glassware from it. This way we leverage existing code, but also allow users the option of interfacing with our blog using a web browser or Glass.

Part Two

The chapters in Part Two are designed to build up from simple to more complex examples. Unlike Part One, however, the examples in each chapter are self contained. You should follow these chapters in order, rather than jump around, since previous sections are sometimes refered to later.

Chapter 9, *Introducing the GDK*, on page 113

This chapter kicks off Part Two by outlining what the GDK is, and how it differs from the Mirror API. It also goes through the steps of setting up a GDK development environment, and how to sideload your own apps onto Glass.

Chapter 10, *An Android Introduction on Glass*, on page 125

You have to crawl before you can walk, so we'll start by crawling through the basics of Android development. Details about Android can, and does, fill several books on its own. So instead of a full treatment, we'll do a flyover of the parts you'll need to jump into GDK programming. We'll cover Android development basics, project layouts, and Android programming model objects like Intents, Views and Components.

Chapter 11, *Live Cards*, on page 147

The first GDK UI elements we'll introduce are Live cards. Live cards are interactive cards that are rendered in realtime, as opposed to the static variety

rendered by the Mirror API. We'll create and launch a live card application to gradually display statistics about the inner workings of Glass, including a simple menu to close it.

Chapter 12, *Advanced Rendering and Navigation*, on page 167

There are many ways to render live cards, but for applications that need video quality we must interact directly with the Glass rendering surface. We'll take advantage of this more complex method of rendering high frequency live cards to write a party application that animates balloons over an image. We'll create a more complex interface with multiple scrollable cards, and take photos with the Glass camera.

Chapter 13, *Voice and Video Immersions*, on page 189

Next we cover the other GDK UI element option called immersions. Immersions provide the full power of an Android application by running outside the constraints of the Timeline experience. We'll learn about immersions by creating a text-to-speech application with geolocation capabilities. We'll also create a QR code reading application that captures and renders video realtime from the Glass camera.

Chapter 14, *Creating an Immersive Game*, on page 219

We'll finish up our GDK tour by writing a side-scrolling video game for Glass. The game will use the built-in gravity sensor and gesture bar to control a player, while playing background music and sound effects.

Chapter 15, *Preparing For the Real World*, on page 245

Once we have the GDK under our belts, we'll look into the hardware of Google Glass. Then we'll cover some details about the Android operating system, and how to hack it, debug and optimize your applications. We'll optimize one of our previous examples to make it run nice and smoothly.

Chapter 16, *Turning an Android App to Glassware*, on page 265

Finally, we'll take an existing Android application and convert it into a Google Glass app. This follows a similar pattern to Chapter 8, *Turning a Web App to Glassware*, on page 99, where an existing application was introduced, the necessary user stories were extracted, and we redesigned the app to work for Glass. And like the Mirror API example, much of the app code will be reusable, allowing us to leverage existing investments.

Online Resources

You can download the code and other resources used in this book from the Pragmatic Bookshelf website or my GitHub repository.[3] [4] You are free to use this source code for anything you wish.

You'll also find the community forum and the errata-submission form on the Pragmatic site, where you can report problems with the text or make suggestions for future versions.

The official Google Mirror API is also an excellent resource for the most up-to-date changes in the API, as well as other viewpoints on creating Glassware.[5]

Getting Going

Wearable computers, like Google Glass, are a growing topic, getting larger by the day. We could have easily created a book twice this length on Glass concepts, the Mirror API, good design, musings on the future, and so on. It was a conscious decision to keep this book slim so you can get a quick head start on this future.

We're beginning an exciting new journey in software development and design. Let's take our first steps into it.

Eric Redmond
Feb 2015

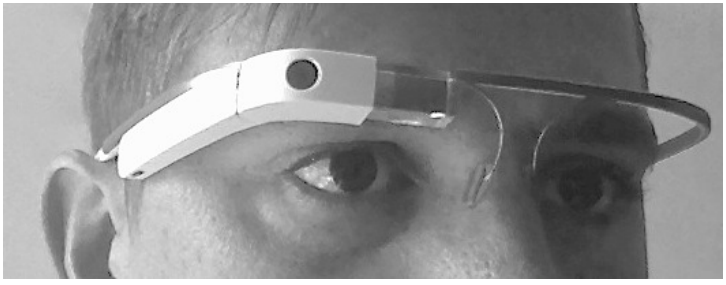
3. pragprog.com/book/erpgg/programming-google-glass
4. https://github.com/coderoshi/glassmirror
5. http://developers.google.com/glass/

CHAPTER 1

Wrapping Your Head Around Glass

The importance of Glass cannot be overstated. It is a huge leap forward in human-machine interface, and one step closer to a ubiquitous access of the world's information. Google cofounder Sergey Brin stated that the vision behind Glass was to improve information access from its current form: staring down at a mobile phone all day.

Glass is a head-mounted form factor for consuming or sharing information.[1] Glass isn't the first or, frankly, even the best wearable computer with an optical display. However, it does mark the first time a company the size and stature of Google has thrown its formidable resources at crafting a consumer-oriented wearable computer. This project is currently the best chance for wide adoption of head-mounted computers by regular people.



And as you can see in the figure, wearing Glass looks cool.

Getting to Know Glass

Glass's defining characteristic is its optical display that hovers above your right eye. As I've worn Glass quite a bit in my day-to-day life, I've come to

1. http://en.wikipedia.org/wiki/Computer_form_factor

learn that in its current form, It is wonderful for effortlessly acquiring and sending out information.

You'll know within a second or two if you want to open an email. You can send a photo or video without digging for your phone. You'll be quicker to respond to text messages since they pop up in your field of vision and can accept verbal replies. You can ask Google a question and get a reasonable answer. I recently asked Glass, "What's the capital of Belize?" while standing at a Belize-themed food cart (it's Belmopan).

On the other hand, many people believe Glass is some sort of virtual-reality device. In fact, most of the time the display is turned off. You'll get an audible notification of when you receive new items, but you won't walk around with the screen on, because it's both a distraction and a battery drain. Although a future generation of Glass will undoubtedly trend toward reality-augmenting capabilities, it's not very good for that sort of thing right now.

Learning to Navigate

All actions in Glass are done by voice command, head motion, or touch pad. When you start up Glass, you'll see a screen with the time. A handful of Google engineers call this the *clock screen*, but I've come to refer to it as the *home card* (who can say if it will always contain a clock?), and will for the rest of this book. A *card* is a single screen visible on the display, as the following figure shows.

The big plastic piece of Glass is where all of the electronics are housed, as you can see in Figure 1,*An overhead view of Glass*, on page 3. The back portion is the battery pack and the middle part in front of your ear is where the computer resides. The outermost casing is a touch interface called the *swipe bar*.

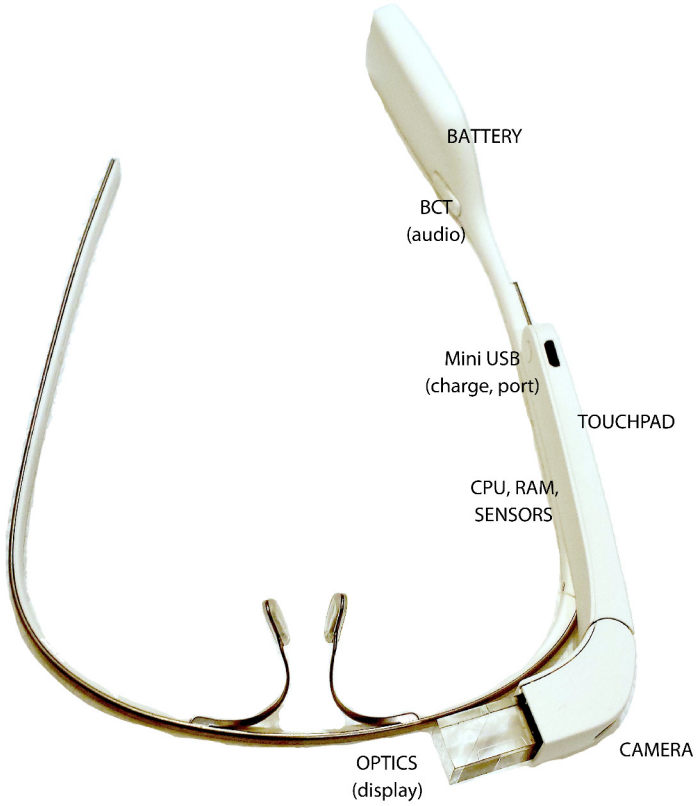
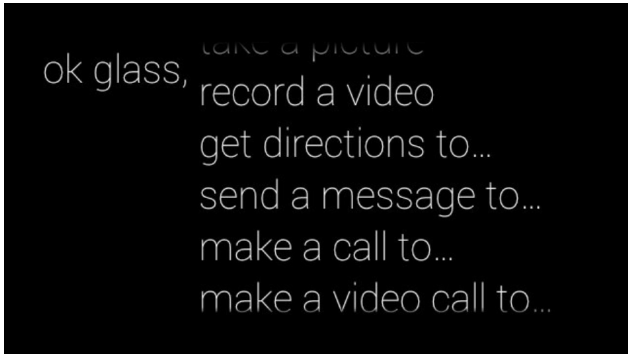


Figure 1—An overhead view of Glass

If you tap the swipe bar on the side of your Glass while it's in power-save mode (the screen is off), you'll start at the home card. From here, saying the voice trigger "OK, Glass" out loud will open up a menu of commands, as shown in the following figure.



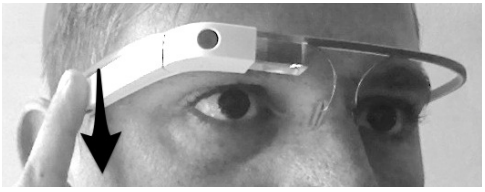
The list scrolls off the page, which leads us to the *head gesture*. Vertical scrolling in Glass is almost always performed by tilting your head up and down. As you look down, the list will scroll up as though you were holding up a dinner menu and perusing down it.

Saying any of the commands out loud will begin that action. The ellipses (...) at the end of the command represent the remainder of the request. Valid input is fairly intuitive.

If you need directions to Las Vegas, say "OK, Glass, get directions to Las Vegas." In that case, "OK, Glass" is the trigger, "get directions to" is the command, and "Las Vegas" is the input.

If you have a contact named "Mom" in your list, you can be a good kid and, assuming you're paired to your mobile phone, start a phone call with "OK, Glass, make a call to Mom."

Voice commands and head-bobbling are only part of using Glass. The other component is swipe-bar gestures. When you're ready to close an active card, you hold a single finger to the side and swipe down, as you see here.

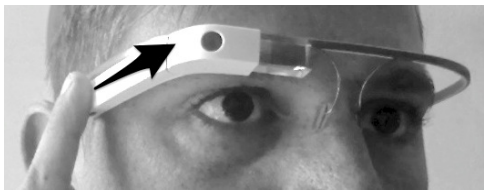


To exit a process, such as ending a phone call, you can touch two fingers to the swipe bar and swipe down. This is the *close* gesture, which goes straight back to the home card and power-save mode. Most times you'll use the single swipe action, but when in doubt, try two fingers.

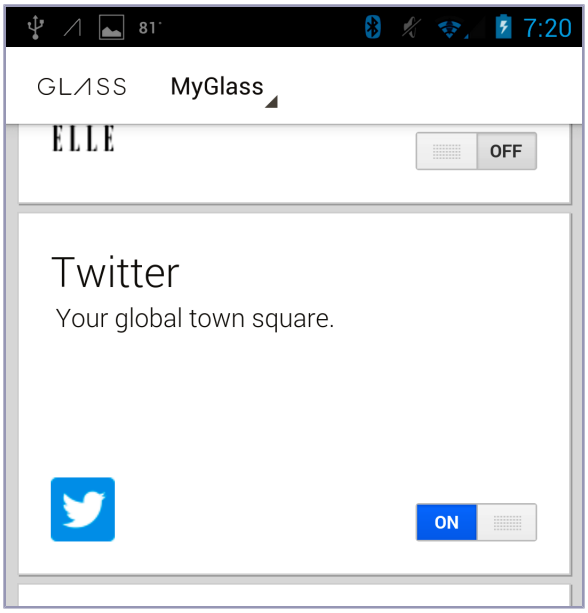
If you tap the side of your Glass again from the home card, you'll see a menu of options, not entirely unlike the voice-command menu. The difference here is that you can bring up your menu choices without speaking, which is useful in company meetings and movie theaters.

Timeline

From the home card, you can swipe forward, which will move your display through your timeline, as you can see in the following image. The timeline is a chronological listing of cards, each displaying a single cell of information. That information is usually simple, be it an email, a past phone call or SMS conversation, a Google search, or some information populated by a Glass application, called *Glassware*.

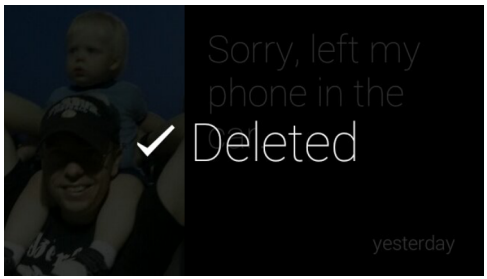


For example, you can sign up for the Twitter Glassware that keeps you up-to-date on new tweets. You can use the Android Glass app, which you see here, but you can also visit http://google.com/myglass for the same options.



Now anytime someone sends you a tweet, it creates a new card in your timeline, stamped with the time of creation.

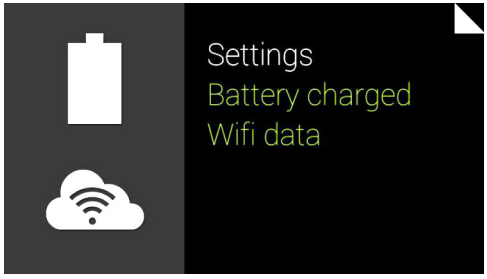
If you tap the swipe bar while viewing the card, aka “tap the card,” you’ll see a *menu* of options. These can be defined or customized with Glassware. If you want to delete a tweet from the timeline, you swipe through to the end of the menu list to the Delete action, and tap to choose it.



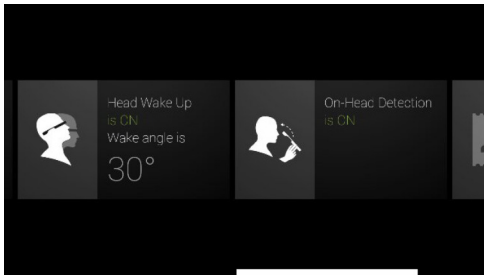
The action will pause for two seconds to give you a chance to abort. If you're really intent on letting it delete, it will signal success and be removed from the timeline. On the other hand, if you wish to abort or leave the menu entirely, swiping down vertically with one finger is the *cancel* or *back* command. Single-finger swipes along the swipe bar or down are the most common gestures you'll make.

If you start at the home card again and swipe the other direction, you'll see a set of non-timeline cards. There are relatively few cards on this side of the home card, since they're meant to be shortcuts. These are known as *pinned cards*, often populated by the Google Now application and containing information like the local weather or airline flight information. We'll cover pinning cards in Chapter 4, *Building the Timeline*, on page 41.

At the end of all pinned cards, you can see the settings card.



This is the center of all Glass-based configurations. You can scroll through settings options like those in the following image.



Tapping on this allows you to join a Wi-Fi network and specify on-head detection, debug mode, and other system settings.

There is plenty more to know about using Glass. If you need help, search the wealth of information online or contact the Glass support team.

Glass Hardware

These hardware specs are good to be familiar with, especially if you had any currently unattainable dreams of rendering highly detailed scenes on Glass in real time or running 24 hours uninterrupted. The hardware just isn't powerful enough for that yet. The design is rather basic, as you saw in Figure 1,*An overhead view of Glass*, on page 3.

Glass currently does not use cutting-edge technologies, but rather combines standard technologies in a cutting-edge manner. The following hardware details were gathered from a Sparkfun-sponsored teardown,[2] the blog post "Sensors on Google Glass,"[3] the GDK documentation,[4] and Glass tech specs.[5] Obviously any of these specs can and likely will change with later versions, but I've listed them here to give you an idea of what is packed into Glass's small form factor.

- Touchpad: Synaptics T1320A
- Processor: Texas Instruments OMAP4430 1.2Ghz dual-core ARMv7
- RAM: Elpida mobile DRAM 1GB
- Storage: 16 GB of SanDisk flash (12 GB usable)
- Battery: Lithium polymer 570 mAh
- Wi-Fi: 802.11b/g
- Bluetooth 4.0
- GPS: SiRFstarIV GSD4e
- Gyroscope, accelerometer, compass (each three-axis): MPU-9150 nine-axis MEMS MotionTracking
- Proximity sensor: LiteON LTR-506ALS
- Microphone: Wolfson WM7231 MEMS
- Audio: Bone-conduction transducer; Micro USB earphone attachment
- Camera: 5 megapixel photos, 720p videos
- Display: 640×360 LCOS array projected on a prism

One of the most interesting pieces on the list is the display. It requires very high-definition LCOS (liquid crystal on silicon) projecting an image onto a

2. http://www.catwig.com/google-glass-teardown/
3. http://thecodeartist.blogspot.com/2013/05/sensors-on-google-glass.html
4. https://developers.google.com/glass/develop/gdk/
5. https://support.google.com/glass/answer/3064128?hl=en&ref_topic=3063354

prism, which reflects an image onto the retina. Despite the prism's sitting about an inch from your eye, you see the equivalent of a 25-inch high-definition screen quite clearly from eight feet away.

The bone-conduction transducer (BCT) is also a nice touch, since it relieves the necessity for an earpiece for simple audio. Since the BCT conducts sounds into the ear by vibrating bones in the skull, it also frees up the ear to still hear ambient sounds, following one of Glass's design goals: to be nonintrusive. The downside of the BCT is that it can be hard to hear in noisy environments. For listening to audio long-term, a mono earbud USB attachment can be purchased.

Glass runs a custom version of Android. As of press time, that's Android version 4.4.2 (API level 19), which is relevant for writing Glassware.

You would be well served to understand Glass's hardware limitations. But this book is about what you *can* do with Glass, so let's move on to the much more interesting software side.

Glass Software

As with the shift from desktop to mobile-phone development, Glass presents new challenges for programmers. Its limited touch interface means that you have to consider other inputs, such as voice and head movements, to interact with Glassware. Its small 640×360 display presents viewport problems even greater than the average smartphone's. In this book you'll learn not only the technical details of writing Glassware with the Mirror API, but also the practical tradeoffs you'll have to consider.

For example, the built-in web browser effectively projects a portion of a website in front of you, and you move your head to pan around the page. It's as though you're looking at a large poster fixed in space, but you can view only a piece at a time. This is one way the Glass engineers used the head-bobble interface to solve the problem of a small display.

As mentioned previously, there are currently two ways to write Glassware: a Google-managed web service called the Mirror API, and a user-installed Android app option called the Glass Development Kit (GDK). This book covers the Mirror API.

Mirror API

The Mirror API is focused primarily on manipulating and reacting to changes to the Glass timeline. Your software communicates to the Mirror API, which Google in turn uses to populate your users' timelines in some way. All

Glassware users have authorized your Glassware to manipulate their timelines by approving the software and access scope using Google's OAuth 2.0 flow. Any discussion of the Mirror API must begin with an overview of writing software (see Chapter 2, *The Google App Engine PaaS*, on page 13) and authorizing a user (covered in Chapter 3, *Authorizing Your Glassware*, on page 27).

Glass Development Kit

The GDK is focused on expanding Glass's functionality beyond the constraints of the Mirror API and the preinstalled interfaces. We can share pictures using the Mirror API, but manipulating those photos on the Glass device requires the GDK. Nearly any application we could write for a very small mobile phone in Android we can write on Glass with the GDK.

The GDK's purpose is threefold: First, to interface with the user or with devices not provided by Glass Android or the Mirror API (for example, a notepad app that pairs with a Bluetooth keyboard). Second, to allow natively installed software that can run absent of an Internet connection (Mirror API timeline manipulations require a Wi-Fi connection). Third, to create applications outside the confines of the timeline (for example, a service that takes a picture when the user blinks).

The reasons someone chooses the Mirror API over a GDK app are similar to why a mobile-phone developer would choose a mobile-optimized web app over a native app.

Wrap-Up

Navigating Glass is easy, but it does present new challenges for developers. Its new form factor reduces the amount of information you can display on a screen, and limits interaction to certain voice commands and touch gestures (swipe forward, backward, up, or down). Bobble-head gestures present a new dimension for interactivity, but will also require great care for programmers and designers to use correctly—as with any new style of interface.

Our next couple of chapters are focused on setting up an environment where we can launch our Glassware and authorize users to use a Mirror API–based application. But more important than the technical details at this point is getting excited to embark on a new age in the computer-human interface!

Part I

The Mirror API

CHAPTER 2

The Google App Engine PaaS

The modern world runs on the Web. Banks, shopping, social networks, and more all run as web applications. These apps generally have two audiences: end users (customers) and other applications (third-party apps). Web apps that interact with users are generally called websites, whereas those that interact with other applications are called *web services*. The Mirror API is one of the latter, and we'll write an application to interact with it.

Since Glassware applications using the Mirror API require a web application to generate content, we have to begin with the steps necessary to create a web application. Unlike applications that are installed natively—the kind you might download on iOS or Android—Mirror API apps are installed and operate exclusively on a remote server.

But we get to take a shortcut. Rather than installing and configuring your own server, we'll take advantage of Google's Platform as a Service (PaaS) called *Google App Engine* (GAE). Using GAE, we can host our own Mirror API–based *Glassware* for free, up to five million page views a month. Look at Figure 2,*The actors in creating Glassware using the Mirror API*, on page 14. At the bottom is your application, which GAE will host.

You can interact with any approved user's Glass device through JavaScript Object Notation-encoded HTTP requests to the Mirror API. Google's servers handle the details of communicating directly with your user's Glass data on your behalf. But before you can start firing off requests that make your Glassware tick, you'll need two things: someplace to run your Glassware code—we'll be publishing code to Google App Engine—and authorization for your application to read from or write to a user's Glass data.

In this chapter we'll tackle the first of those items by setting up and publishing a GAE-hosted Java web application that we'll use throughout the book. This

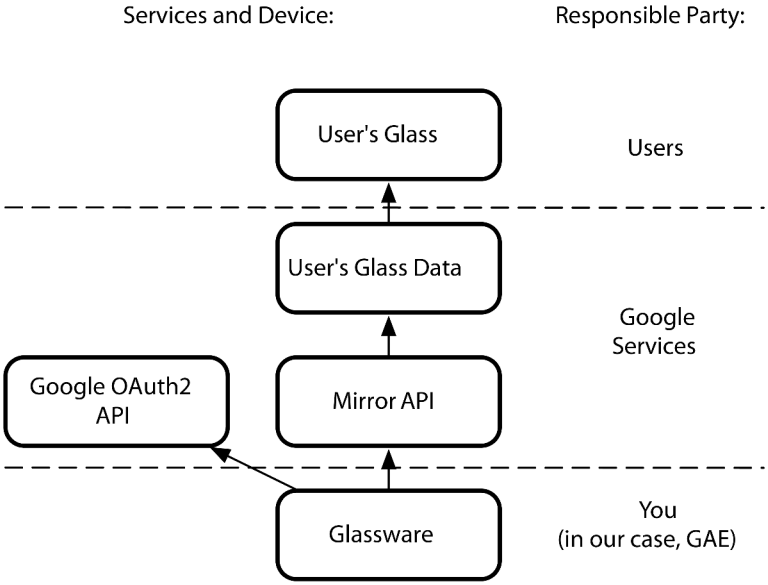


Figure 2—The actors in creating Glassware using the Mirror API

is the skeleton of our Glassware. Our project is called *Lunch Roulette*, which will suggest a random lunch idea to any of our Glass users. It'll start rather basic, but as we move on it will become increasingly sophisticated, complete with geolocation, restaurant suggestions, and fancy designs and images, and will even allow users to call the restaurant (perhaps to make a reservation).

But first we should set up a development environment.

Platform as a Service

Google App Engine is Google's PaaS. The idea behind all PaaS providers is to allow developers to deploy web-app code to the provider's infrastructure. PaaS handles many of the operation details, like installation, deployment, and adding servers, so you can focus on writing code. Although you could use other PaaS providers, such as Heroku, we'll stick with a full Google stack in this book.

Setting Up GAE

Before we dive into our Glassware, we'll need to set up a Google App Engine application. Happily, any Google account, such as Gmail, can be used to create one.

1. Visit https://appengine.google.com and click Create Application. If this is your first application, you'll have to verify your account.
2. Enter a unique Application Identifier (app ID), that will be your subdomain at GAE's 'appspot.com' domain—for example, 'glassbooktest'.
3. We'll be building an application called Lunch Roulette, so enter that in the Application Title field, as you see in the following figure.
4. Select Open to All Google Accounts Users under Authentication Options. This will allow any Glass user to try out your application.
5. Select High Replication under Storage Options. This is the default and future-proof storage engine, since Master/Slave is being deprecated in GAE.
6. Click Create Application.

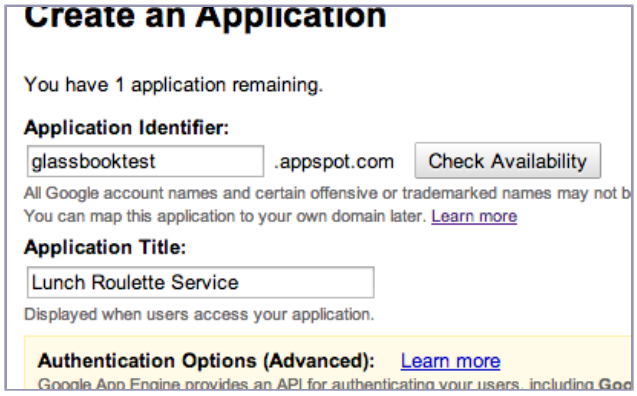


Figure 3—Creating an application

With everything set up, you can visit your web app. Its URL is your app ID prepended as a subdomain of appspot.com. Since the example ID in this book will be glassbooktest, you would visit https://glassbooktest.appspot.com.

The page will output *Error: Server Error*. This is expected since the app location exists but nothing has been deployed.

Android Developer Tools

Throughout Part One of this book we'll be using the Eclipse integrated development environment (IDE). Eclipse is very stable and popular, and Google provides several plug-ins that simplify GAE development and deployment. This code can weigh in at some heavy megabytes (around 200 MB), plus there

are some plug-ins. It can also be a bit of a memory hog, so don't go installing this on your Raspberry Pi. No matter the power of your system, these steps can take a while to download and install.

First, ensure you have the Java 7.0 Java Development Kit (JDK) installed on your machine. Any JDK type (OpenJDK, Oracle, and so on) should work, though I tend to stick with the Oracle release. Next, install Eclipse for your operating system.[1] All examples in Part One are based on Eclipse version 4.2 (Juno).

The first time you launch Eclipse it will have to create a workspace. This is just a directory that will house all projects created from the IDE, and some other working files.

Once Eclipse is finished launching, we need to install the GAE plug-ins. Select Help in the top menu bar, then Install New Software. A window will pop up to add available software; it should look something like Figure 4,*Adding the GAE plug-ins to Eclipse*, on page 17.

We need to enter a Work With URL that tells the IDE where to find the plug-ins: http://dl.google.com/eclipse/plugin/4.2. If that does not work, visit the Google Plugin for Eclipse (https://developers.google.com/eclipse/) website to find installation instructions, which will provide a correct URL.

Several plug-in options will populate. You'll need only the following:

- Google App Engine Tools
- Google Plugin for Eclipse
- SDKs -> Google App Engine Java SDK

Click Next, then wait a bit while your tools download and install. You may have to agree to some licenses and restart Eclipse after install.

With your tools installation complete, you should find a little Google logo in the lower corner of your IDE's workspace. Click it. A window will pop open requiring you to log in using the same Google account you used to create your GAE application. If you have multiple Google accounts, ensure you use the correct one.

Making a Web App

Now let's have some fun by making a simple GAE web application, building on the skeleton of our previously mentioned project, Lunch Roulette. Lunch Roulette will eventually become a web service that populates a user's Glass

1. http://developer.android.com/sdk/index.html

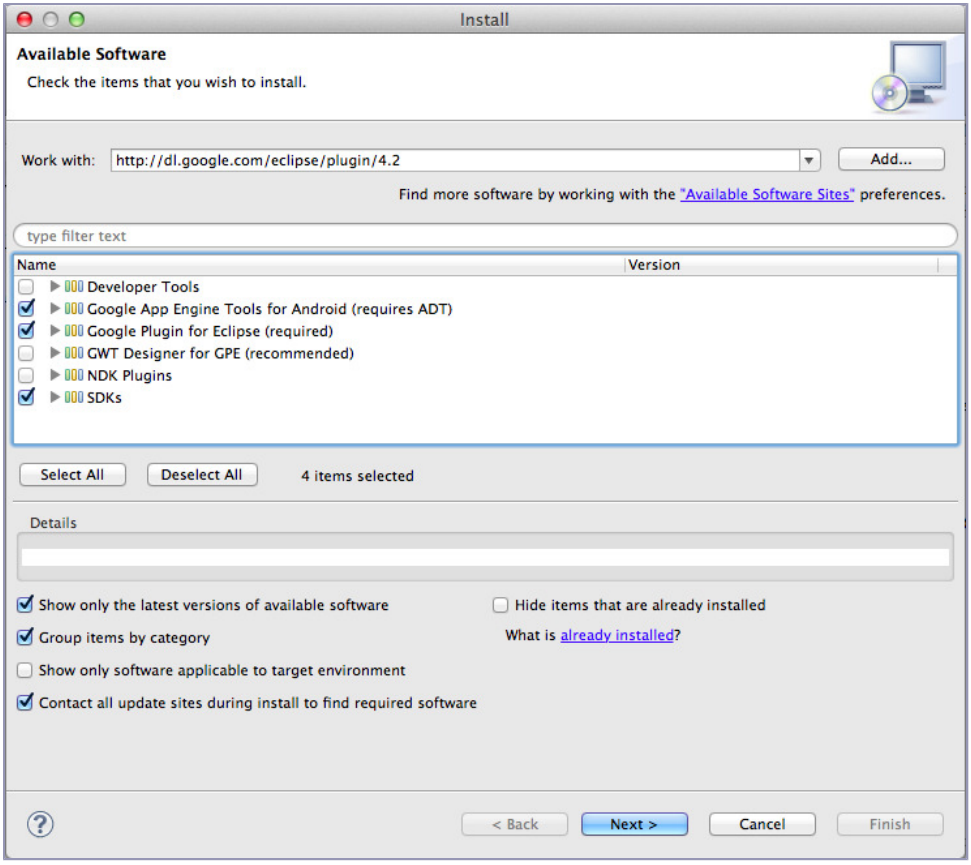


Figure 4—Adding the GAE plug-ins to Eclipse

timeline with lunch suggestions. But in this first pass we'll create a simple web app that outputs a random lunch suggestion every time we refresh the web page.

Creating a Java Web App

To create the Java web app that is the skeleton of Lunch Roulette, click on the Google icon in the top bar, and select New Web Application Project, like you see in Figure 5,*New Web Application drop-down option*, on page 18.

It will launch a web-app project wizard, where you can enter a project name and package. Enter *LunchRoulette* and *test.book.glass*, respectively. Uncheck Use Google Web Toolkit (we won't use it anywhere in this book), and ensure Use Google App Engine is checked. Also check Generate Project Sample Code, since this will be the basis of our Lunch Roulette project. Click Finish after

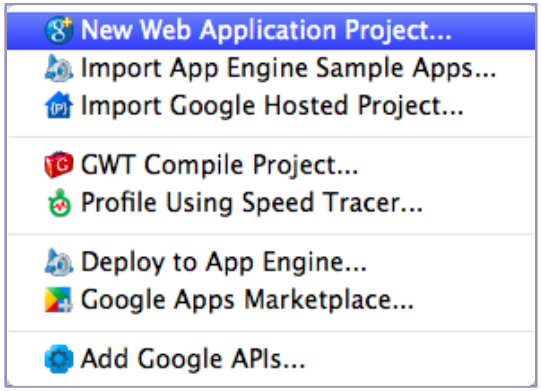


Figure 5—New Web Application drop-down option

your setup looks like the one in Figure 6, *Your setup should look like this*, on page 19.

You should now have a project in your Eclipse sidebar called LunchRoulette that contains some generated Java web-app files. The files worth looking at are LunchRouletteServlet.java and war/WEB-INF/web.xml. The LunchRouletteServlet.java file will contain the code we're going to write and the web.xml file contains the metadata for the web app—specifically, it defines which URL gets routed to which servlet.

We'll modify the generated Java servlet to accept HTTP GET requests at /lunchroulette and reply with a random lunch suggestion, such as Italian, Chinese, American, or whatever you add to the list. To output a random suggestion, replace the generated LunchRouletteServlet class with the following code.

```
public class LunchRouletteServlet extends HttpServlet
{
  /** Accept an HTTP GET request, and write a random lunch type. */
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws IOException, ServletException
  {
    resp.setContentType("text/plain");

    String lunch = LunchRoulette.getRandomCuisine();
    resp.getWriter().println(lunch);
  }
}
```

Finally, create a new class called LunchRoulette and define a new getRandomCuisine() method.

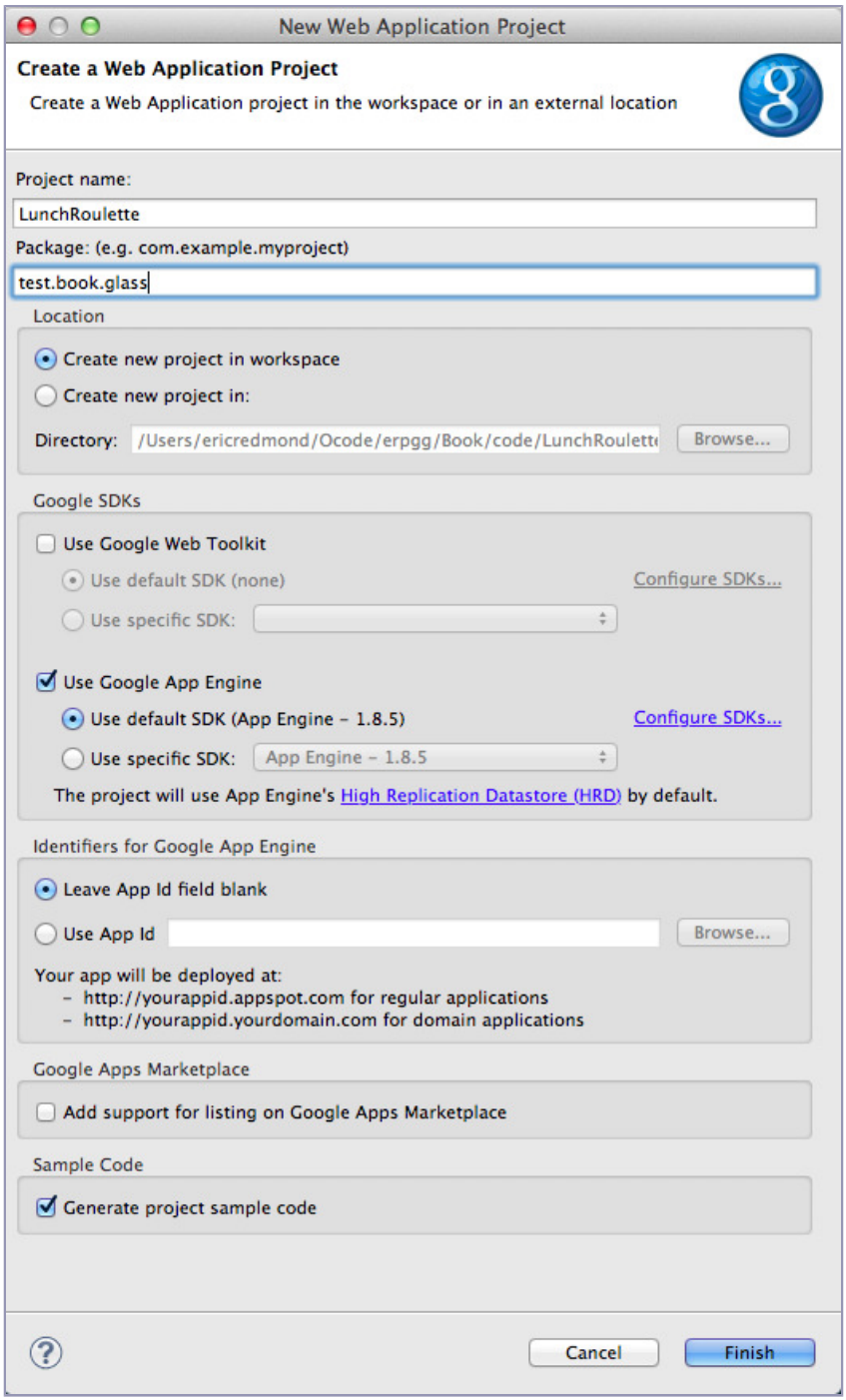


Figure 6—Your setup should look like this.

chapter-02/LunchRoulette/src/test/book/glass/LunchRoulette.java

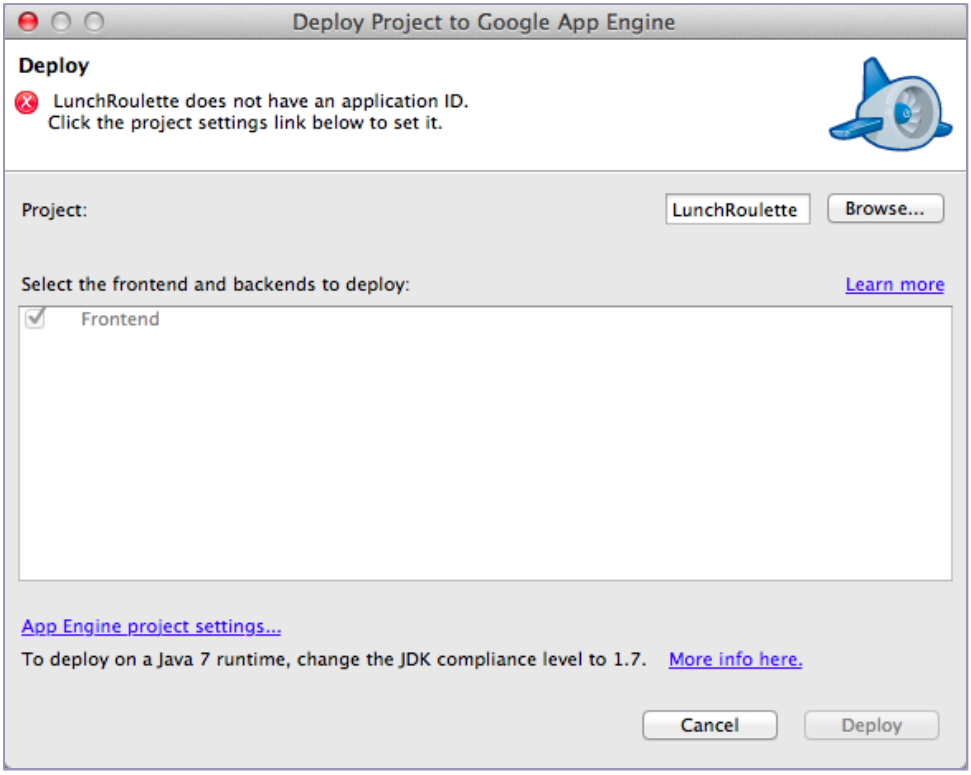
```
public static String getRandomCuisine()
{
  String[] lunchOptions = {
      "American", "Chinese", "French", "Italian", "Japenese", "Thai"
  };
  int choice = new Random().nextInt(lunchOptions.length);
  return lunchOptions[choice];
}
```

This file will contain most of the custom code we'll create as we build our Lunch Roulette app throughout the book.

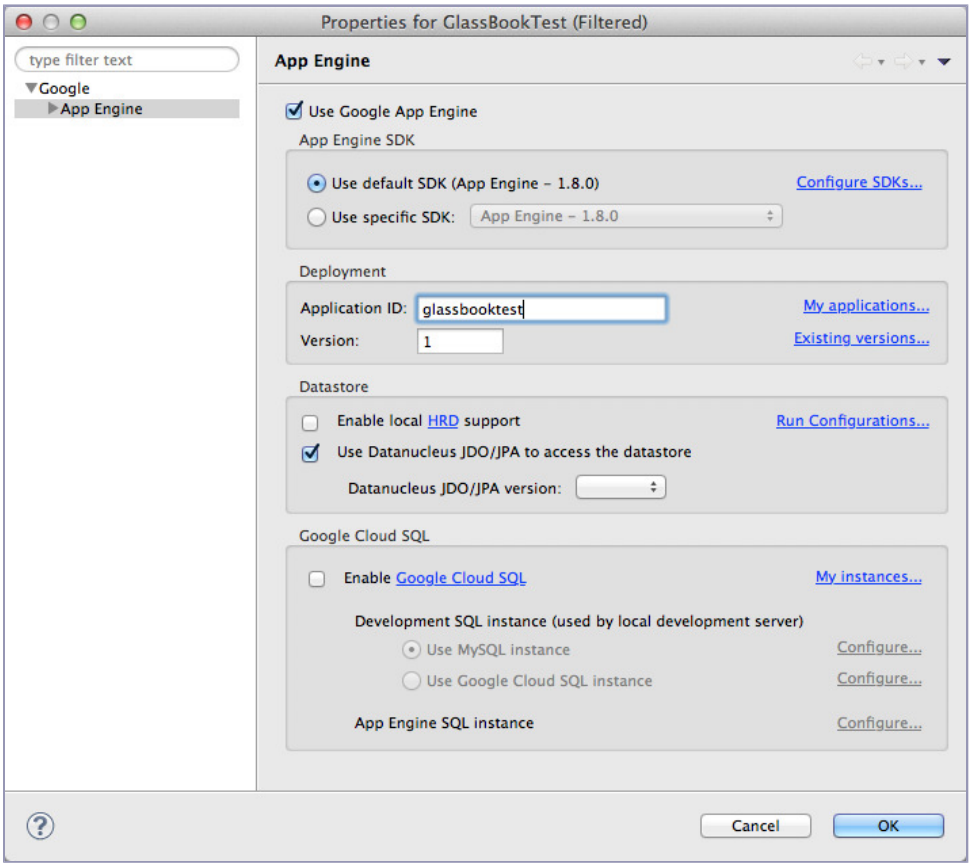
Our final step is to deploy the app to the GAE servers.

Deploying to the Web

You can either right-click the project and find Google in the menu, or select the project and click the Google icon in the top bar. Either way, choose Deploy to App Engine. You'll get a wizard that complains, "LunchRoulette does not have an application ID." Click the App Engine Project Settings link at the bottom to correct this.



In this pop-up (so many pop-ups!) enter your GAE application ID and click OK.



You should be back at the deployment wizard, so now you can click Deploy. You'll see a lot of text running through your console as it tries to deploy. If all goes according to plan, you'll see the log message "Deployment completed successfully."

```
------------ Deploying frontend ------------

Preparing to deploy:
  Created staging directory at: '/var/folders/lz/s8dy1m9x0r9g2z7zt4ktpsgr0000gn
    /T/appcfg6810984979425725784.tmp'
  Scanning for jsp files.
  Scanning files on local disk.
  Initiating update.
  Cloning 2 static files.
  Cloning 22 application files.
```

```
Deploying:
  Uploading 5 files.
  ... snip ...
  Deploying new version.
  Closing update: new version is ready to start serving.
  Uploading index definitions.

Deployment completed successfully
```

If anything goes wrong, the log will give a hint as to the problem. If your deployment fails, ensure you're logged in and that you've entered the correct application ID. Many other problems could arise here. The Glass Developers community and Stack Overflow are excellent places to find help.[2,3]

To verify everything is deployed, visit your GAE root URL (mine was 'https://glassbooktest.appspot.com/'). You'll be greeted by an index page (generated for you by the GAE plug-in) with a link to your 'LunchRoulette' servlet, like in the following figure.

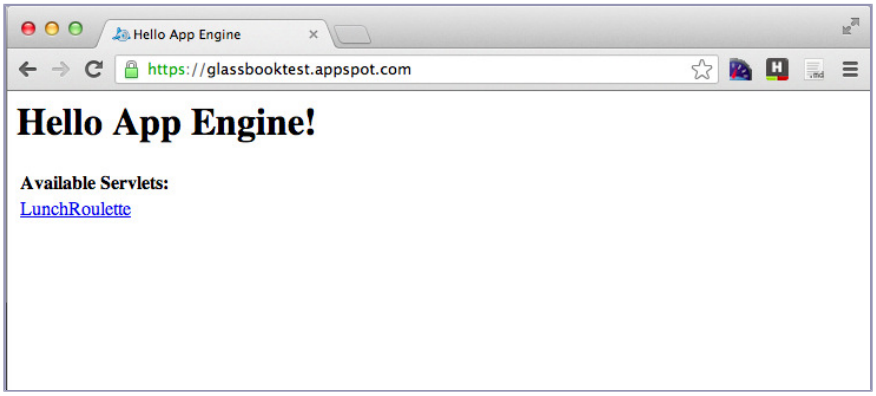


Figure 7—The index page with a link to your servlet

Click on the link to harvest the fruits of all of your hard work: a random lunch type—you should feel proud! Refresh to get a new lunch option.

Fancy Templates

So far our servlet has only sent plain text to the browser. Rather than writing HTML inline or using Java Server Pages—the common alternative to servlets in Java—we'll introduce a simple templating library called FreeMarker.[4]

2. https://developers.google.com/glass/community
3. http://stackoverflow.com/questions/tagged/google-mirror-api)
4. http://freemarker.sourceforge.net/

Download the GAE-compatible binary FreeMarker .jar, and drag it into your project's war/WEB-INF/lib directory. If the JAR doesn't show up as one of the Referenced Libraries, click the project's properties, select Java Build Path, and under Libraries choose Add JARs... then drill down to war/WEB-INF/lib, choosing the JAR.

We need to add a method to the LunchRouletteServlet that renders a template file with some given data, and write the result to the response stream. This function assumes the FreeMarker template generator can find template files under WEB-INF/views.

chapter-02/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
public static String render(ServletContext ctx, String template,
                            Map<String, Object> data)
    throws IOException, ServletException
{
  Configuration config = new Configuration();
  config.setServletContextForTemplateLoading(ctx, "WEB-INF/views");
  config.setDefaultEncoding("UTF-8");
  Template ftl = config.getTemplate(template);
  try {
    // use the data to render the template to the servlet output
    StringWriter writer = new StringWriter();
    ftl.process(data, writer);
    return writer.toString();
  }
  catch (TemplateException e) {
    throw new ServletException("Problem while processing template", e);
  }
}
```

Then change the servlet's doGet method to populate a random food and render the template, rather than simply printing out the string. We also need to set the content type as HTML.

chapter-02/LunchRoulette/src/test/book/glass/LunchRouletteServlet.java

```
public class LunchRouletteServlet extends HttpServlet
{
  /** Accepts an HTTP GET request, and writes a random lunch type. */
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws IOException, ServletException
  {
    resp.setContentType("text/html; charset=utf-8");

    Map<String, Object> data = new HashMap<String, Object>();
    data.put("food", LunchRoulette.getRandomCuisine());

    String html = LunchRoulette.render(
        getServletContext(), "web/cuisine.ftl", data);
```

```
    resp.getWriter().append(html);
  }
}
```

Finally, we need to create the template file under war/WEB-INF/views/web/cuisine.ftl. This is just regular HTML with an interpolated variable food inside of the ${ ... } directive.

chapter-02/LunchRoulette/war/WEB-INF/views/web/cuisine.ftl

```
<!doctype html>
<html>
  <head>
    <title>Lunch Roulette</title>
    <style>
    h2{ color:#db1; }
    body{ background-color:black; color:white; }
    </style>
  </head>
  <body>
    <article>
      <h2>Your Lunch</h2>
      <strong>${ food }</strong>
    </article>
  </body>
</html>
```

With our code in place, now is a good time to test it. You don't have to deploy the application to test these changes. Click the run icon in Eclipse (it looks like a green triangular play icon), or choose Run As -> Web Application. This runs a small GAE environment locally—basically, a Java web-app server. Then visit http://localhost:8888/. This is useful in later chapters, since you can test your Glassware without deploying your application to GAE.

If all works according to plan, you should be redirected to the Google authorization page, requesting access to view account information, your Glass timeline, and your Glass location.

This may also be a good time to choose Deploy to App Engine.

Wrap-Up

I promise we're making great strides toward our first Glassware application. We activated our Google App Engine service, set up the development environment, generated a web application, and deployed it to GAE. We also made a small step toward eventually customizing our Glass content with dynamically rendered HTML.

Soon we'll populate Glass timelines the world over with delicious lunch suggestions in Chapter 4, *Building the Timeline*, on page 41. But first, we must expand our existing web-server code to allow a user to log into the system. This is the crux of the next chapter.

CHAPTER 3

Authorizing Your Glassware

In Chapter 2, *The Google App Engine PaaS*, on page 13, we set up a simple Google App Engine (GAE)–hosted Glassware shell. But it wasn't Glassware because it could never run on Glass. To do so requires authorization. Authorization is required to both flag an application as real Glassware and interact with a user on a personal level, such as by inserting cards directly into a user's timeline.

Whereas authentication is about verifying who a user is, authorization is fundamentally about user experience. Formal definitions cite security and access control, but our purpose for authorization is to uniquely identify the correct user to customize his experience in some way by accessing and sometimes managing his data.

In this chapter we'll leverage our GAE application to hook into the Glassware application pool, allowing users to authorize our application. This is the last step necessary to create real Glassware.

Additionally, you'll learn how to personalize Glassware for users by accessing their protected data, with the users' permission, of course. We'll achieve this by authorizing a user, then using credentials to make requests.

In any web-based application (which all Mirror API applications are) the first step of a custom user experience is knowing who the user is. This need not involve personal information like a name or email address; it need only consist of some token that a user can share with the web application to identify herself.

Security, such as a password or OAuth 2.0, is to ensure that only that user has access to her own personal information, be it credit-card details or a collection of family photographs. We'll use Google's OAuth 2.0 interface for the rest of this book. OAuth can be quite complex, and sticking with one

implementation for one purpose will let us charge through this necessary but sometimes mind-melting topic.

Power and Responsibility

With great power comes great responsibility. Users give you access to their data (like Facebook pictures) or permissions to perform actions (such as posting tweets) so they can conduct their own pursuits. Users are not interested in helping your company mine data or more easily sell them products.

Google has policies to help make this agreement more explicit. You can and should read the details here: https://developers.google.com/glass/policies. In sum, allow users to manage their own data, take proper security measures, and don't ever share users' data.

Activating Your Mirror API

Before your app can begin authenticating users, you must activate its Mirror API and secure both a client ID and a client secret key. Here's the rub: until Glass is available to the general public, you must be a Glass Explorer to activate this API.

Like all API activations, you access this through Google's APIs Console. If you notice a message at the top of the screen that says something like "Try the New Cloud Console", click that. These steps won't work for the old API console.

Also please note that Google is still in the midst of creating the new Developer Console, and some of these steps are liable to change after publication.

Steps for activating are as follows:

1. Visit Google's new Developer Console.[1]
2. Select APIs & Auth, then toggle the Google Mirror API to ON.

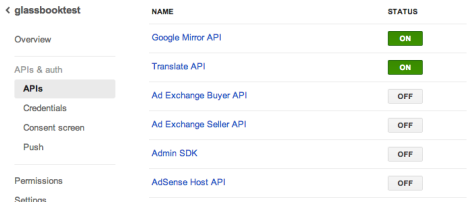


Figure 8—APIs and auth screen

1. https://console.developers.google.com/

3. Under the APIs & Auth tab, choose Credentials.

4. Click the Create New Client ID button. Keep the Web application type selected, and fill out the Authorized JavaScript origins to be the same HTTPS hostname as your appspot Glassware from Chapter 2, *The Google App Engine PaaS*, on page 13. My test domain was https://glassbook-test.appspot.com. Also add https://mirror-api-playground.appspot.com for future uses.

5. The Authorized redirect URI will automatically be populated with the paths oauth2callback, such as https://glassbooktest.appspot.com/oauth2callback. Replace the mirror-api-playground URI with http://localhost:8888/oauth2callback.

6. Click Create Client ID.

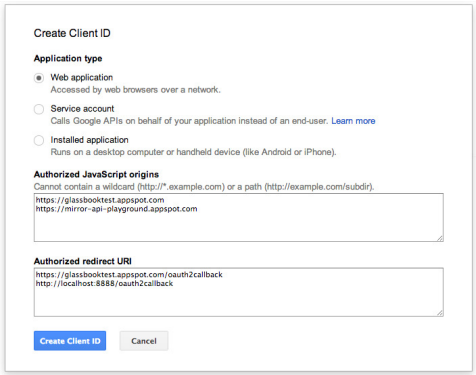


Figure 9—OAuth 2.0 Client ID content screen

7. Still under the OAuth section, take note of the Client ID and Client Secret values. The code throughout the book will refer to a client ID as [MY_NUMBER].apps.googleusercontent.com and a client secret as [MY_SECRET_KEY], but you should always replace those placeholders with your own values.

8. Finally click on the Create New Key button under Public API access, then choose Server key. There's no need to fill out any IP addresses, so just click Create. Take note of the API key, we'll use this at a later point.

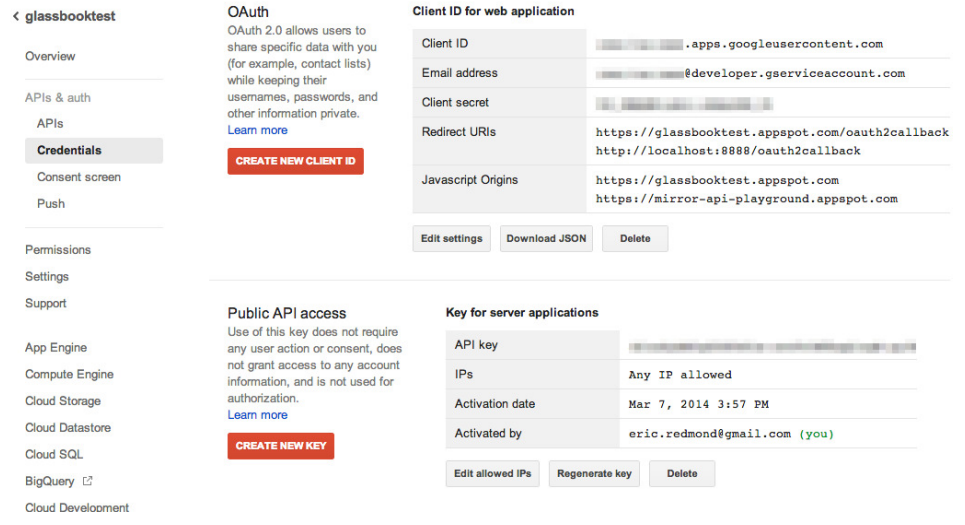


Figure 10—Completed Auth Credentials screen

You've now registered your application with Google. You'll use the Client ID and Client Secret values as part of the authorization process. But before we write any code, let's take a moment to cover what exactly OAuth 2.0 is.

A Short Primer on OAuth 2.0

OAuth 2.0 is the second version of an open API access delegation standard created by Twitter in early 2007. The most common use case of OAuth is to allow an end user to provide access to one API from another service. It's somewhat like owning a car (the API), but lending valet keys to someone you trust so that person can drive it for you. The valet key may start the ignition, but it cannot open the trunk. This can come in handy if you're sick and want someone to drive to buy you soup. In the virtual world you would, for example, give travel application TripIt access to your Facebook wall so it can post flights on your behalf.

OAuth terms can take a bit of getting used to. We won't survey all of the intricate details, such as multiple grant types or n-leggedness, but rather we'll stick with the bare minimum necessary to authenticate a Glassware user.

The authentication process requires at least four agents, or *roles*, working together.

Roles

The first role is the *resource owner*, which generally is the end user (except when it's not, such as in the case of machine-to-machine communication).

The *resource server* is the API with access to the user's information, such as your Twitter feed. The *authorization server* works with the resource server and manages *authorization tokens* after obtaining authorization. Generally from a user's point of view the resource server and the authorization server are part of the same service (for instance, Yahoo!).

The last role is the *client*, which is the third-party application attempting to access resource-server data. In our case, our Mirror application is the client, asking the resource owner to allow us access to the resource server.

Another example may help. When you authorize Middle (a blog server) to log in via your Twitter account, you are the *resource owner*, Twitter is the *resource server*, and Middle is the *client*.

The OAuth Dance

Recall that we signed up for Client ID and Client Secret codes through the Google APIs Console. Those act as our client credentials to Google's authorization server (https://accounts.google.com/o/oauth2/auth). With roles and credentials in hand, we can see how they all work together in relation to our Glassware running on GAE. In OAuth-speak, this is known as the *protocol flow*. You can see the following steps graphically in the next figure.

1. Auth is kicked off when an end user (*resource owner*) requests that our Glassware have access to his resources. This isn't actually part of the protocol flow, but is an important step.

2. OAuth starts when our Glassware server (the *client*) requests that the user (*resource owner*) authorize our Glassware to access resources. This is done in the application approval screen. The user is presented with a dialog requesting that he give your client permission to access some set of resources (called *scopes*). This dialog is generated by the very same authorization server we'll use in step 3, and is basically a web page that the user can log in through. Our client uses its 'client_id' and 'client_secret' so the authorization server knows which client is making the consent request. Since it's owned by Google and we want to access Google resources, we and the user both expect that source to be trusted. If the user denies permission, our adventure ends here. If the user consents, we continue to the next step.

3. Our client receives an *authorization code*, which is a short-lived, one-time code.
4. Our client sends that code to Google's authorization server.
5. Assuming the auth code is valid, the authorization server issues a reusable access token and a refresh token. An access token has a time-to-live, which helps ensure security. If the token ever fell into the wrong hands, an attacker would be limited in how much time he had to access a user's resources.

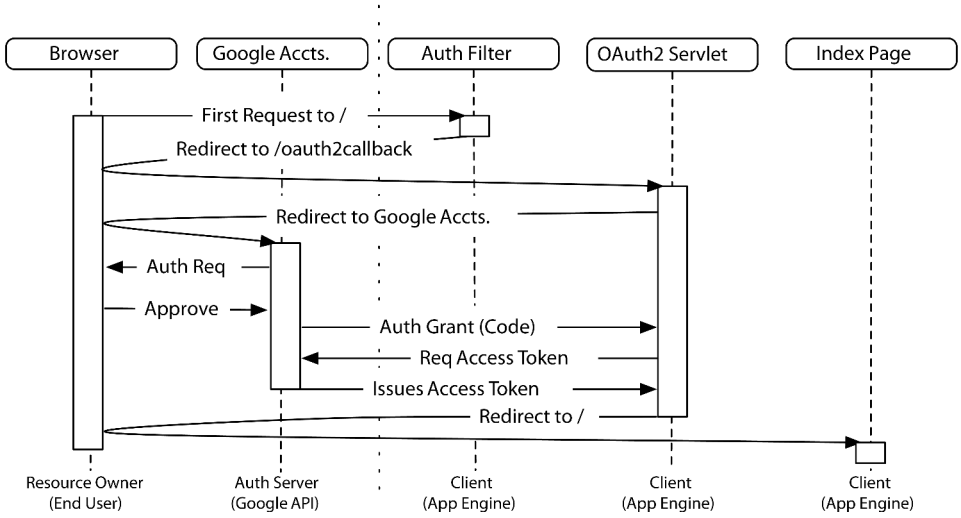


Figure 11—OAuth 2.0 protocol flow

The Glassware endpoints are URLs defined in your GAE application, which we'll create shortly.

At this point we have an access token that we use to issue all Mirror API requests. In fact, we use this token to issue any authorized Google API request. Using the token is the remainder of the flow. You can see the next set of steps in the following figure.

1. A request to the Google Mirror API requires that our Glassware sends the access token.
2. All responses from the Mirror API will be protected resources, assuming the access token is valid. It's up to Google's own system to ensure that the access token is authentic and has access to the chosen API resources.

3. If the access token is expired, we request a new access token from the auth server by using the refresh token.

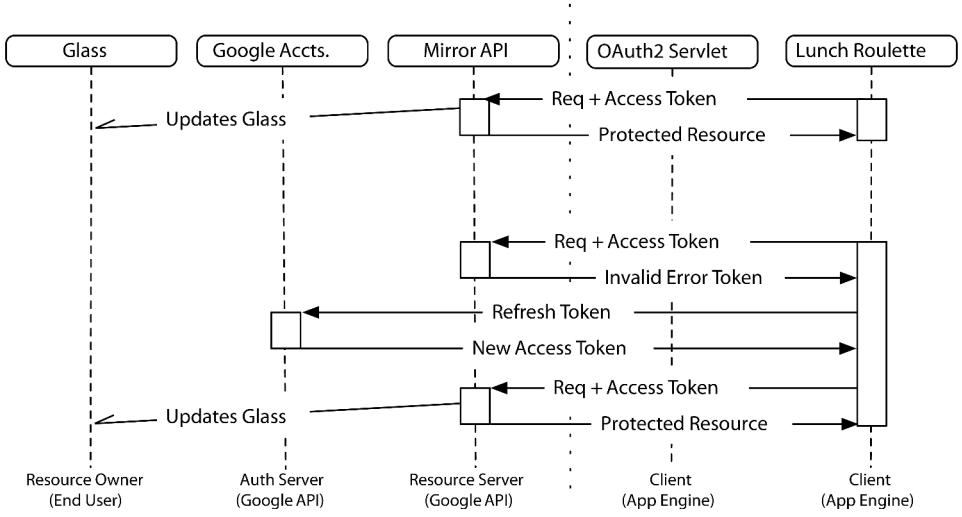


Figure 12—Using an access token to issue Mirror API requests

Using the access token is your normal workflow. However, occasionally that access token will expire, and your app will have to use a refresh token to get a fresh access token. Once you have the new token, your Glassware will continue making requests are usual. Happily for us, Google's API client library will handle all of this refresh business!

Scopes

So far we've assumed that once a client is authorized to access a resource server, we can access all information. But we skipped over one crucial detail: *scopes*.

When a user is presented with an authorization dialog, that dialog will contain a list of scopes that explain to the user which resources the Glassware wants to access. This allows the end user to make an informed decision. Sadly, most users never read this, so it's up to ethical programmers like us to ensure we never access more than we need to perform the job. If you're creating a Glassware that displays a daily stock-market ticker in a user's timeline, you don't need to access the user's email.

Since Google's OAuth 2.0 is built to be a general-purpose authorization service, OAuth scopes are much wider than simple Mirror API resources like user timelines.

These are some of the most interesting scopes:

- https://www.googleapis.com/auth/glass.timeline
- https://www.googleapis.com/auth/glass.location
- https://www.googleapis.com/auth/userinfo.profile
- https://www.googleapis.com/auth/userinfo.email

There is no universal list of scopes for all OAuth servers, though most allow access to basic user information like email or username. Google's is no exception. You can see a complete list of Google's OAuth scopes at the OAuth 2.0 Playground.[2]

A Quick Note on Scopes and Glassware Registration

Google pulled double duty with its scopes in the case of Glassware. When a user approves a glass.* scope (glass.timeline or glass.location), the client that is approved is added to a user's Approved Glass Applications list, found on http://google.com/myglass (this URL only works if you own Glass).

With scopes and the rest of OAuth under our belts, let's put the theory into practice with an implementation of the process we've been talking about. Don't worry; it's simpler than it may appear.

Applying OAuth to Create Glassware

We'll expand our Lunch Roulette application from the previous chapter to include authentication. This is a required step for activating a Glass application.

The crux of our authorization process is threefold:

1. Filter all incoming requests and redirect a request to be authorized if no user is in session or no credentials are stored.
2. Provide an OAuth 2.0 endpoint servlet. This does the real heavy lifting. It can act as either the redirect to the Google authorization server or the callback accepting and storing Google Accounts' responding auth code. It is also responsible for requesting the initial access token with the given authorization grant (a 'code' parameter).

2. https://developers.google.com/oauthplayground/

3. Provide a logout servlet to remove stored credentials and the user ID from session.

Setup

To simplify our OAuth client implementation, we'll use the google-api-client project auth code, which is a dependency of the google-api-services-mirror project.

Through the integrated development environment (IDE) you can get all of the necessary Java Archive (JAR) files. Right-click on the project, and click Google -> Add Google APIs…. In the pop-up window search for *mirror*, then select Google Mirror API and click the Finish button.

Some of the gritty implementation details of the next few sections we'll leave until the end, encapsulated within the class AuthUtils. All of the new Java classes we'll introduce here will be put under the test.book.glass.auth package.

Auth Filter

The first item on our agenda is to create an authorization filter. Luckily, Java web apps, including those managed by GAE, have a built-in *filters* concept. Basically, this means all incoming web requests will first be checked against the Filter implementation code, a class I've named AuthFilter. The filter can choose to make any changes before servlets have a chance to execute, or halt continued execution altogether.

In our case, we'll continue execution if a user ID in the current session has a corresponding access-token credential stored. If not, we'll redirect any request back into the OAuth process. Clearly, we can also only filter requests to paths other than /oauth2callback, or an infinite redirect loop will occur (the first callback request will have no token, thus redirect to the callback, and so on).

chapter-03/LunchRoulette/src/test/book/glass/auth/AuthFilter.java

```
public class AuthFilter implements Filter
{
  public void doFilter(ServletRequest req, ServletResponse res, FilterChain fc)
    throws IOException, ServletException
  {
    HttpServletRequest request = (HttpServletRequest)req;
    HttpServletResponse response = (HttpServletResponse)res;

    // If this path is not destined for a redirectable URI
    // and has no access token, redirect to the oauth2 path
    if (isRedirectable(request)
        && !AuthUtils.hasAccessToken(SessionUtils.getUserId(request)))
    {
```

```
      response.sendRedirect( AuthUtils.OAUTH2_PATH );
      return;
    }
    // execute any remaining filters
    fc.doFilter( request, response );
  }
  private boolean isRedirectable(HttpServletRequest request) {
    return !request.getRequestURI().equals( AuthUtils.OAUTH2_PATH );
  }

  public void init(FilterConfig fc) throws ServletException {}
  public void destroy() {}
}
```

The filter must also be registered with the web application, which we'll do by adding to the GAE Glassware's web.xml file.

chapter-03/LunchRoulette/war/WEB-INF/web.xml

```
<filter>
        <filter-name>authFilter</filter-name>
        <filter-class>test.book.glass.auth.AuthFilter</filter-class>
</filter>
<filter-mapping>
        <filter-name>authFilter</filter-name>
        <url-pattern>*</url-pattern>
</filter-mapping>
```

OAuth2Servlet

The workhorse of this chapter is all less than 100 lines of Java code, named OAuth2Servlet. It's just a regular HttpServlet, like the LunchRouletteServlet Glassware endpoint from Chapter 2, *The Google App Engine PaaS*, on page 13.

For any HTTP GET request to /oauth2callback, we want to check if any errors have been passed in, such as /oauth2callback?error=access_denied. An error simply outputs an error message to the user.

Otherwise, the callback will redirect to one of two URLs. If we have no authorization code, we'll forward to Google's authorization server. That server is responsible for asking the user's permission to grant your Glassware access to her resources. If denied, this same servlet is called again with the error=access_denied parameter set.

chapter-03/LunchRoulette/src/test/book/glass/auth/OAuth2Servlet.java

```
public class OAuth2Servlet extends HttpServlet
{
  protected void doGet( HttpServletRequest req, HttpServletResponse res )
      throws IOException
  {
    if( !hasError(req, res) ) {
```

```
      res.sendRedirect( doAuth(req) );
    }
  }
```

However, if access is granted, an authorization `code` parameter is sent back to this callback servlet. We use this code to build a `GoogleTokenResponse` object. From that object we can extract the logged-in user's ID and a credentials object. We store both of these in session and a GAE credential store, respectively.

chapter-03/LunchRoulette/src/test/book/glass/auth/OAuth2Servlet.java

```
private String doAuth(HttpServletRequest req)
    throws IOException
{
  String authCode = req.getParameter( "code" );
  String callbackUri = AuthUtils.fullUrl( req, AuthUtils.OAUTH2_PATH );

  // We need a flow no matter what to either redirect or extract information
  AuthorizationCodeFlow flow = AuthUtils.buildCodeFlow();

  // Without a response code, redirect to Google's authorization URI
  if( authCode == null ) {
    return flow.newAuthorizationUrl().setRedirectUri( callbackUri ).build();
  }

  // With a response code, store the user's credential, and
  // set the user's ID into the session
  GoogleTokenResponse tokenRes = getTokenRes( flow, authCode, callbackUri );

  // Extract the Google user ID from the ID token in the auth response
  String userId = getUserId( tokenRes );

  // Store the user if for the session
  SessionUtils.setUserId( req, userId );

  // Store the credential with the user
  flow.createAndStoreCredential( tokenRes, userId );

  return "/";
}
```

Like with any new servlet, we map its URL path by adding its information to `web.xml`.

chapter-03/LunchRoulette/war/WEB-INF/web.xml

```
<servlet>
        <servlet-name>oauth2callback</servlet-name>
        <servlet-class>test.book.glass.auth.OAuth2Servlet</servlet-class>
</servlet>
<servlet-mapping>
```

```
        <servlet-name>oauth2callback</servlet-name>
        <url-pattern>/oauth2callback</url-pattern>
</servlet-mapping>
```

The final piece of our authorization design is to let users log out.

LogoutServlet

Now we'll create a method for users to invalidate their own credentials. It's simply a servlet we're mapping to /logout, and any HTTP POST action removes the user's stored credentials, removes the user ID from session, and outputs a friendly response.

chapter-03/LunchRoulette/src/test/book/glass/auth/LogoutServlet.java

```
public class LogoutServlet extends HttpServlet
{
  protected void doGet( HttpServletRequest req, HttpServletResponse res )
      throws ServletException, IOException
  {
    AuthUtils.deleteCredential( SessionUtils.getUserId(req) );
    SessionUtils.removeUserId( req );
    res.getWriter().write( "Goodbye!" );
  }
}
```

Like with the other URLs, we need to inform web.xml:

chapter-03/LunchRoulette/war/WEB-INF/web.xml

```
<servlet>
        <servlet-name>logout</servlet-name>
        <servlet-class>test.book.glass.auth.LogoutServlet</servlet-class>
</servlet>
<servlet-mapping>
        <servlet-name>logout</servlet-name>
        <url-pattern>/logout</url-pattern>
</servlet-mapping>
```

Let's conclude these steps by investigating the authorization work that is wrapped up in a utility class.

Final Details

The meat of this chapter is the AuthUtils class, which wraps up code provided by the Google API client project .jar included by the Android Developer Tools IDE. This utility gets, deletes, checks, and stores credentials. It mainly builds AuthorizationCodeFlow objects (OAuth flow) which are responsible for helping with the authorization-server endpoint redirects as well as persisting user credentials.

chapter-03/LunchRoulette/src/test/book/glass/auth/AuthUtils.java

```
public static AuthorizationCodeFlow buildCodeFlow()
    throws IOException
{
  return new GoogleAuthorizationCodeFlow.Builder(
      new UrlFetchTransport(),
      new JacksonFactory(),
      WEB_CLIENT_ID,
      WEB_CLIENT_SECRET,
      SCOPES)
  .setApprovalPrompt( "force" )
  .setAccessType("offline")
  .setCredentialDataStore( getDataStore() )
  .build();
}
```

The scopes we wish to access from our users are their user-profile information, their Glass timeline, and their Glass location. This will allow us to access their user ID, post items (such as lunch suggestions) into their timeline, and access their location (to find nearby places). We'll use the glass.location scope in a later chapter.

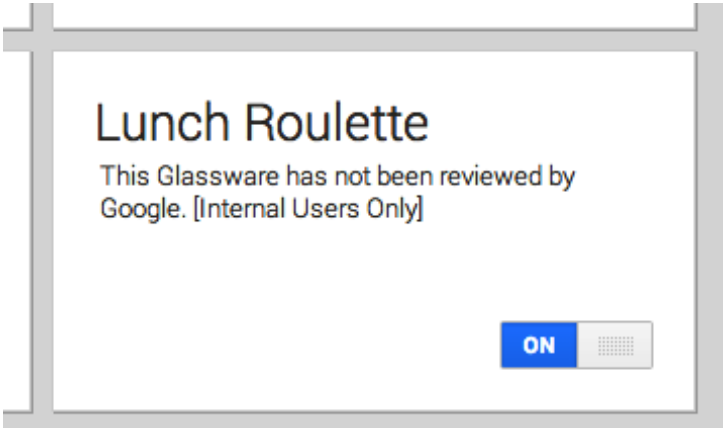
chapter-03/LunchRoulette/src/test/book/glass/auth/AuthUtils.java

```
public static final List<String> SCOPES = Arrays.asList(
  "https://www.googleapis.com/auth/userinfo.profile",
  "https://www.googleapis.com/auth/glass.timeline"
);
public static final String WEB_CLIENT_ID =
    "[MY_NUMBER].apps.googleusercontent.com";
public static final String WEB_CLIENT_SECRET = "[MY_SECRET_KEY]";
public static final String OAUTH2_PATH = "/oauth2callback";
```

The WEB_CLIENT_ID and WEB_CLIENT_SECRET values are populated from the Google Cloud Console's API Access -> Client ID we made earlier in this chapter.

Generally, we won't want to hard-code these values like we're doing here; rather, we want to populate them using environment variables within the appengine-web.xml file. For the time being, though, this will suffice.

If you visit your GAE URL after deploying your app, you should be redirected to an authorization screen before visiting the page. It will list your scopes, explaining to any potential user what your app can authorize. Assuming you approve, you'll be redirected back to your web app. And just to prove to yourself that this is real Glassware, visit http://google.com/myglass or the Android Glass app, and at the bottom of the list of applications you'll see Lunch Roulette.



Congratulations on creating your first Glassware!

Wrap-Up

Although OAuth 2.0 can be a complex protocol, Google has provided many tools to simplify its operation. Better still, it uses those tools (such as the `Credential` class) throughout the Mirror API client, making actions such as inserting a card into a user's timeline as simple as utilizing the user's credential object.

Now that we've covered authorizing an application, we can dig into the details of Glassware design, starting with creating and managing timeline elements.

CHAPTER 4

Building the Timeline

Remember history classes from school? At some point you likely encountered a graphical chronology, populated by points in time. For example, one point may read *June 18, 1815: Napoleon loses at Waterloo*, followed by a point that reads *June 22, 1815: Napoleon abdicates his throne*, and so on. This is a classic timeline, a chronological sequence of events.

In Glass, a timeline is the core organization unit through which Glassware operates. Rather than a desktop filled with icons or a web page filled with links, Glass stamps every card with a point in time, and places them in order. Ordering starts from the home card and stretches to the right indefinitely. As you scroll further right, the cards grow progressively older. Since new cards are the closest to the home card, they are the most accessible. The newest emails, Tweets, images, text messages, or whatever are the first reached. Each chronological item is called a *timeline item*, and it's what this chapter is about.

So far we only have a skeleton for Lunch Roulette. It consists of a Google App Engine (GAE)-hosted web app that lets users authorize our application, but it doesn't do much else. We want to populate a user's Glass with a random cuisine. Then the user should be able to take some action on that suggestion, such as pin the card for later or hear a cuisine read out loud. We'll finish up the chapter by taking advantage of Google App Engine's cron jobs.

Mirror HTTP Requests

The Mirror API is Google's front-end web service for populating a user's timeline, as well as allowing a user to interact with your web application with custom menu items or change notifications, like geolocation. For example, if your program issues an HTTP POST to the Mirror API's `/timeline`, you'll create a new item on a user's timeline. The Mirror API informs the user's Glass device

of the update whenever the Glass device is connected to Wi-Fi. Although the Mirror API acts as a middleman, for the sake of simplicity you can think of calls to the Mirror API as interacting directly with a user's Glass device. For all of resources we cover (not just the timeline) you will interact with them by making HTTP requests.

Glassware communicates to the Mirror API's URL endpoints, just like any other web service, such as Google maps or the Twitter API. Your app calls those URLs as an HTTP 1.1 protocol request, using the HTTP verbs POST, GET, PUT/PATCH, and DELETE. Those verbs roughly correspond to CRUD (create, read, update, delete) actions.

Glass is populated by the Google Mirror service, but Glass will not interact directly with our GAE application. Instead, Glass will connect with Google, and Google will act as a middle man, issuing requests to our GAE application, which sends or receives JavaScript Object Notation (JSON), as you can see in the following figure.

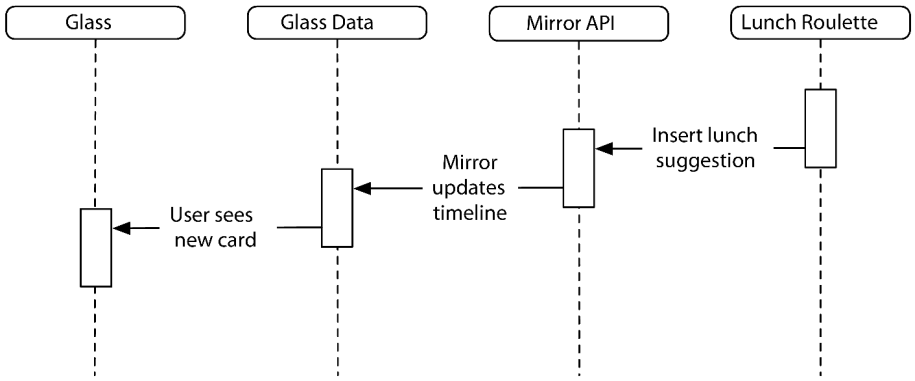


Figure 13—Flow of Glassware to Glass

If our GAE application needs to create some text on a user's timeline, we issue a POST to the Mirror API. The header and body of an HTTP request might look like this:

```
POST /mirror/v1/timeline HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer e057d4ea363fbab414a874371da253dba3d713bc
Content-Type: application/json
Content-Length: 25

{"text": "Italian"}
```

Of course, we don't really want to handle the complexity of building raw HTTP requests. So instead we write code that generates and delivers proper messages, including the Authorization Bearer token. The code in this book is Java, but you can interact with the Mirror API using any code that generates and consumes well-formed HTTP messages.

```
String userId = SessionUtils.getUserId( req );
Credential credential = AuthUtils.getCredential( userId );

Mirror mirror = new Mirror.Builder(
        new UrlFetchTransport(),
        new JacksonFactory(),
        credential)
    .setApplicationName("Lunch Roulette")
    .build();

Timeline timeline = mirror.timeline();
```

Starting at the top, we get the userId and credentials that were stored as part of the authorization steps in Chapter 3, *Authorizing Your Glassware*, on page 27 (note that req is just the current HttpServletRequest).

We use those credentials to build a Mirror object. This object handles all of the communication with Google's Mirror API, especially building the Authorization HTTP header field. This is how Google knows that your app is allowed to use the Mirror API, which user's Glass data you want to manipulate, and that the user has allowed you to manipulate that data.

Every step we've taken so far will be executed every time we need to communicate with Mirror, so let's wrap that up into a helper class called MirrorUtils.

chapter-04/LunchRoulette/src/test/book/glass/MirrorUtils.java

```
public static Mirror getMirror( HttpServletRequest req )
    throws IOException
{
  String userId = SessionUtils.getUserId( req );
  Credential credential = AuthUtils.getCredential(userId);
  return getMirror(credential);
}

public static Mirror getMirror( String userId )
    throws IOException
{
  Credential credential = AuthUtils.getCredential(userId);
  return getMirror(credential);
}

public static Mirror getMirror( Credential credential )
    throws IOException
```

```
{
  return new Mirror.Builder(
      new UrlFetchTransport(),
      new JacksonFactory(),
      credential)
  .setApplicationName("Lunch Roulette")
  .build();
}
```

Then use getMirror() in LunchRoulette to insert a new timeline item.

chapter-04/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
public static void insertSimpleTextTimelineItem( HttpServletRequest req )
    throws IOException
{
  Mirror mirror = MirrorUtils.getMirror( req );
  Timeline timeline = mirror.timeline();

  TimelineItem timelineItem = new TimelineItem()
    .setText( getRandomCuisine() );

  timeline.insert( timelineItem ).executeAndDownloadTo( System.out );
}
```

From the Mirror object we get a timeline object via the timeline() method, which we'll use to manipulate a user's timeline. Every Mirror API resource is accessed in Java from the Mirror object. In later chapters we'll use the location, subscription, and contacts resources by calling locations(), subscriptions(), and contact(), respectively.

With our timeline object in hand, we decorate a simple TimelineItem object and insert it into the timeline. Calling executeAndDownloadTo() and passing in System.out will stream the Mirror's raw JSON response to your Android Developer Tools console, giving us something like the following. It's a useful method for debugging purposes, but not much else, so we'll generally call the shorter execute() method.

```
{
  "kind": "mirror#timelineItem",
  "id": "1234567890",
  "selfLink": "https://www.googleapis.com/mirror/v1/timeline/1234567890",
  "created": "2013-09-05T17:50:18.738Z",
  "updated": "2013-09-05T17:50:18.738Z",
  "etag": "\"hzfI85yu0lKQdtWV4PO1jAbQxWw/Ur8Sr0qylBQOrj5CxBM9xX7-qog\"",
  "text": "Italian"
}
```

We'll do all of the work within the LunchRoulette class. To test the preceding code, replace your existing LunchRouletteServlet's doGet() method with the following

code. Rather than generating an HTML page in the browser with a random cuisine, it populates your Glass timeline with a food suggestion.

```
  public void doGet(HttpServletRequest req, HttpServletResponse resp)
      throws IOException, ServletException
  {
    LunchRoulette.insertSimpleTextTimelineItem( req );

    resp.setContentType("text/plain");
    resp.getWriter().append( "Inserted Timeline Item" );
}
```

You can now run the project and visit the http://localhost:8888/lunchroulette URL (assuming you're running the code locally on the default port 8888) on a web browser. It will trigger the creation of this card in your Glass, which looks like the following figure.

Figure 14—Lunch Roulette card

We've made a card! Visiting the /lunchroulette path in a browser is an easy way to test the creation of timeline items. It's useful for testing, but not quite so much in a production sense. Near the end of this chapter, we'll discuss how to automate card creation in a more meaningful way.

We're off to a good start, but let's see what more we can do to with timeline items.

Timeline Items

If a timeline is a sequence of chronological events, a timeline item is an individual event. Every timeline item has a timestamp and some sort of payload, be it text or HTML, an attachment like an image or video, or a more abstract asset like a contact with a phone number to call.

You can create, read, update, and delete timeline items. We'll dig a little deeper into creating some richer Lunch Roulette items.

Create a Timeline Item

The first action any Glassware app is likely to take is to create a new timeline item.

The Java code for creating a new card in the timeline is the timeline object's insert() method. You have to call the execute() methods to send the new timeline change to the Mirror API; otherwise, you've just populated the object locally.

chapter-04/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
TimelineItem timelineItem = new TimelineItem()
  .setTitle( "Lunch Roulette" )
  .setText( getRandomCuisine() );

TimelineItem tiResp = timeline.insert( timelineItem ).execute();

setLunchRouletteId( userId, tiResp.getId() );
```

This code generates a POST action to the Mirror API with a JSON body.

```
POST /mirror/v1/timeline HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer e057d4ea363fbab414a874371da253dba3d713bc
Content-Type: application/json
Content-Length: 25

{"text": "Italian", "title": "Lunch Roulette"}
```

The response message includes the newly generated timeline item ID, so we can save and access this timeline item later. We use the tiResp response to extract and save its ID in the GAE datastore.

chapter-04/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
DatastoreService store = DatastoreServiceFactory.getDatastoreService();
Key key = KeyFactory.createKey( LunchRoulette.class.getSimpleName(), userId );
Entity entity = new Entity( key );
entity.setProperty( "lastId", lunchRouletteId );
store.put( entity );
```

Since GAE provides for saving the ID in the datastore, and doing so doesn't have much to do with the Mirror API, we'll forgo a lot of detail. Suffice it to say it has many steps: create the DatastoreService object, create a unique Key for this user, and store an Entity that's merely a property map of key-values. If you launched your Glassware in another Platform as a Service, such as Heroku, you would store this value (and the credentials) in another database.

If you run what you have so far, every time you visit the /lunchroulette path you'll create a new card and save its ID.

More Timeline Fields

We've created some cards, but skipped a few dozen fields you can set on a timeline item. You can find a listing of these fields in Appendix 1, *HTTP and HTML Resources*, on page 283. In Java code, all field values can be set using a Java setter method of the same name. We've done a little of this already with setText(). To set a title value call the TimelineItem's setTitle() method, call setHtml() for html, and so on.

Get a Timeline Item

After creating a timeline item, you can use the timeline ID string you stored to pull the object down from Mirror. This is useful for a couple of reasons. First, it saves you from having to store any more timeline-item data than the ID. Second, a timeline item can change state over time, whether its HTML changed to show something new or there were other changes in metadata—like isDeleted having been changed to true, meaning the timeline item was deleted.

Later on we'll go over how we can be notified of any changes to timeline items. When we're notified of a change, we'll again use the timeline ID to get details of what changes occurred.

In Java, we call timeline.get(id) once we have the timeline ID we want. Let's build a simple helper method called getLastSavedTimelineItem() to get the last item stored for a user.

chapter-04/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
String id = getLunchRouletteId( userId );

TimelineItem timelineItem = timeline.get( id ).execute();
```

The timeline.get() will generate an HTTP GET using the ID /mirror/v1/timeline/{id}.

```
GET /mirror/v1/timeline/1234567890 HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer e057d4ea363fbab414a874371da253dba3d713bc
```

GET will return a JSON object similar to the POST action generated, with a few more fields populated (see Appendix 1, *HTTP and HTML Resources*, on page 283).

Update a Timeline Item

Rather than creating a new timeline item with every refresh of /lunchroulette, how about we just update an existing item with new information? After all, how many lunches will someone eat in a day?

In Java you set any values you want in the timeline item, and it will update that particular timeline item and show the user any visible changes, such as text. Since we made a change, we also want to move the item to the most recent location in the timeline by updating its display time to right now.

```
TimelineItem timelineItem = getLastSavedTimelineItem();

timelineItem.setText( getRandomCuisine() );
timelineItem.setDisplayTime( new DateTime(new Date()) );

timeline.update( timelineItem.getId(), timelineItem ).execute();
```

Making an update will automatically bump the item's updated field to the current timestamp. This is true of an update to any Mirror API resource type.

Using the update() method is a fine choice if you have the full TimelineItem object in hand, such as from a get() call. However, in case you want to update only specific fields, Mirror provides a patch() update. PATCH refers to an HTTP 1.1 action that is similar to a PUT, but rather than generating the full object you want to update, you instead provide only the specific fields you want to update. Other than that, you call it in the same way.

```
String lastId = getLunchRouletteId( userId );
TimelineItem timelineItem = new TimelineItem();

timelineItem.setText( getRandomCuisine() );
timelineItem.setDisplayTime( new DateTime(new Date()) );
timeline.patch( lastId, timelineItem ).execute();
```

Patching is generally more efficient than putting. Note that in the Java code, for update() we first had to get() the full timeline-item object before updating, but for patch() we only had to create a new object and set the fields we want to change. This saved us a Mirror call, and as our timeline-item objects get larger, it saves us from sending a bunch of superfluous fields.

The update() method builds a PUT request with a similar kind of JSON payload as the create() method.

```
PUT /mirror/v1/timeline/1234567890 HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer e057d4ea363fbab414a874371da253dba3d713bc
Content-Type: application/json
Content-Length: 21

{
  "kind": "mirror#timelineItem",
  "id": "1234567890",
  "title": "Lunch Roulette",
  "text": "French",
```

```
  "displayTime": "2013-09-17T15:38:55-07:00"
  "selfLink": "https://www.googleapis.com/mirror/v1/timeline/1234567890",
  "created": "2013-09-05T17:50:18.738Z",
  "updated": "2013-09-05T17:50:18.738Z",
  "etag": "\"hzfI85yu0lKQdtWV4PO1jAbQxWw/Ur8Sr0qylBQOrj5CxBM9xX7-qog\"",
}
```

As you can see here, patch() generates a PATCH request with a noticeably smaller payload.

```
PATCH /mirror/v1/timeline/1234567890 HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer e057d4ea363fbab414a874371da253dba3d713bc
Content-Type: application/json
Content-Length: 21

{
  "text": "French",
  "displayTime": "2013-09-17T15:38:55-07:00"
}
```

So which should you use, update() or patch()?

Since PUT requires so much more information, is often paired with a GET request, and removes fields that you don't set, most of the time you'll want to call patch(). There are cases, for example, when you want to remove a list of elements, and you'll have to call update() to set the list as empty since patch() will simply assume you want the list left as is.

Delete Timeline Item

Nothing good can last forever. Eventually you may want to delete an unneeded card. As long as you have the timeline item's ID string, you can tell Mirror to delete it for you. Calling this action will remove the item from the user's displayed timeline.

```
timeline.delete( itemId ).execute();
```

The HTTP that delete() generates is just as straightforward; we call a DELETE action on the /mirror/v1/timeline/{id} URL.

```
DELETE /mirror/v1/timeline/1234567890 HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer e057d4ea363fbab414a874371da253dba3d713bc
```

DELETE will wipe out most of the data from the card, but it won't immediately delete the item from Google's back end. This means you can still GET a deleted item. The only data that it will return, however, is a field named isDeleted set to true and a few read-only fields such as etag and selfLink.

Fancy Timeline Items

In Chapter 2, *The Google App Engine PaaS*, on page 13 we used a template library called FreeMarker to render a fancy HTML output. We'll leverage that library again, but this time instead of a web page, let's render the output to a timeline item using setHtml().

chapter-04/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
// get a cuisine, populate an object, and render the template
String cuisine = getRandomCuisine();
Map<String, String> data = Collections.singletonMap( "food", cuisine );
String html = render( ctx, "glass/cuisine.ftl", data );

TimelineItem timelineItem = new TimelineItem()
  .setTitle( "Lunch Roulette" )
  .setHtml( html )
  .setSpeakableText( "You should eat "+cuisine+" for lunch" );

TimelineItem tiResp = timeline.insert( timelineItem ).execute();
```

We also need to make a slightly different template file: war/WEB-INF/views/glass/cuisine.ftl. This is similar to the HTML template we made in Chapter 2, *The Google App Engine PaaS*, on page 13, but without the html, head, and body tags that Mirror would strip out.

chapter-04/LunchRoulette/war/WEB-INF/views/glass/cuisine.ftl

```
<article>
  <section>
    <h2 class="yellow">Your Lunch</h2>
    <strong>${ food }</strong>
  </section>
</article>
```

Whenever you set HTML, you should also setSpeakableText(). This is similar to text, but rather than being printed on a card to read, this text is read aloud to a user when a Read Aloud menu item is selected (more on menu items soon).

If you run this code, the view on your Glass should now be fancied up a bit from the plain text version, with a yellow Your Lunch title and, beneath it, a bold cuisine, as you see in Figure 15,*Lunch Roulette HTML card*, on page 51.

If you were to set HTML and text, only the HTML would render on the card. Setting plain text on a timeline item is simple, but limited. Although we'll use text again in other examples, there's rarely a good reason to avoid HTML. And since you'll be writing HTML, it's a good idea to pick an HTML templating library that you like.

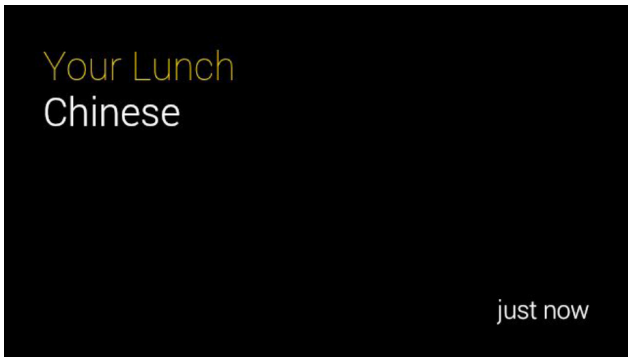


Figure 15—Lunch Roulette HTML card

Multicards: Bundles and Paginating

Sometimes you have more data that you can fit onto a single card, like a collection of photos. Sometimes you have many distinct cards that you want to group together, like commonly tagged items on a blogroll.

A *bundle* is a collection of cards. It visually distinguishes itself from single cards with a small page curl (a small inverted white triangle) in the upper-right corner of the first card, called the *cover card*—see the following figure.

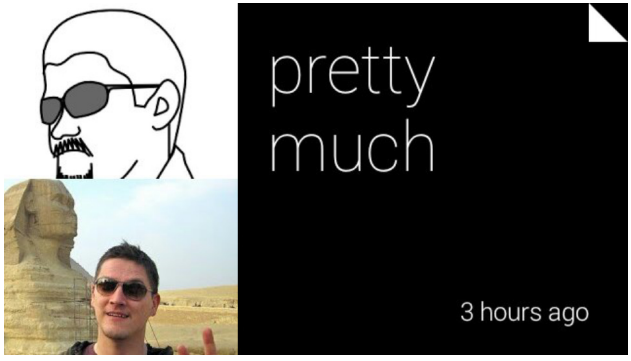


Figure 16—Cover-card chat bundle

You can paginate or flip through a bundle by tapping on the cover card to expand it, then swiping through each card. The Mirror API provides two ways to support multicards (cards with multiple screens): You can either thread multiple timeline items or paginate one timeline item.

Threading Bundles

The heavyweight method of bundling is to thread cards. In this case, you create separate cards, all of which share a bundleId. This is a good choice if you can't create all off the cards at once, if you want to make every card its own item, if you have an unbounded number of cards, or if you want to set text rather than HTML. Threading is so named because it's similar to an email thread, as opposed to paging, with is more akin to a single item with many pages.

In a threaded group of items, one can have isBundleCover set to true. If no cover item is set with isBundleCover, the most recently added timeline item will serve as a de facto cover.

Let's say you want Lunch Roulette to present a user with a main cuisine choice, but if the user doesn't like it, you want an alternate ready.

chapter-04/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
// Generate a unique Lunch Roulette bundle id
String bundleId = "lunchRoulette" + UUID.randomUUID();

// First Cuisine Option
TimelineItem timelineItem1 = new TimelineItem()
  .setHtml( renderRandomCuisine( ctx ) )
  .setBundleId( bundleId )
  .setIsBundleCover( true );
timeline.insert( timelineItem1 ).execute();

// Alternate Cuisine Option
TimelineItem timelineItem2 = new TimelineItem()
  .setHtml( renderRandomCuisine( ctx ) )
  .setBundleId( bundleId );
timeline.insert( timelineItem2 ).execute();
```

This will create two timeline items. The first choice is in front. Every successive alternate choice with the same bundleId will be folded under the cover card. You can see them side-by-side in the following figure.

Figure 17—The cover card and an alternate choice

Card versus Timeline Item, Redux

If you're still not clear on the difference between a timeline item and a card, you're not alone. The difference is subtle.

A *card* represents what a user sees on a single screen rendered as HTML or text, or with assets like a video or image. That's all. In fact, we *could* call it a screen if it weren't for those pesky menus, which don't count as cards at all but are still displayed on a screen. You can always swipe between cards.

A timeline item is more concrete. It's an object at a point in time on the timeline (unless the item is pinned). It may or may not contain multiple cards. When you thread a timeline item with several HTML pages, each page is displayed as a single card in a set of cards. But the overall object is still only one timeline item.

The confusion stems from the fact that usually a timeline item renders a single card. You should burn this nomenclature into your mind since it'll be used precisely throughout this book, although in some cases the terms really are interchangeable.

Paginating

There's another case where multiple cards can appear for a single timeline item. If you set `text` that spans more than one page, Glass will split the item into multiple pages, and the text break will end with an ellipsis. It isn't a bundle—it doesn't display a page curl in the corner. You read all of the text by tapping the card, which will open a menu with a Read More option. The option looks like Figure 18,*Paginating with Read More*, on page 53.

Figure 18—Paginating with Read More

Then, you'll be able to scroll from card to card just like in bundles. We'll revisit Read More and how to make a more nuanced use of this action via HTML in Chapter 7, *Designing for Glass*, on page 87.

You might ask what benefit a threaded bundle has over paginating. A general guideline is that if the cards are immediately known, part of a complete thought, or expected to be fixed, then use paginating. If the cards will grow in an unbounded way or are otherwise unique items, thread them.

We're about to discuss how to add menus to timeline items. Another benefit of bundling is that you can include a set of menus for each card, rather than the one menu that paginating allows.

Menus

Picture yourself going to a nice, table-service restaurant. The first thing you probably do is open a menu and make a selection from a listing of food options. While pizza may be an acceptable choice at an Italian *ristorante*, it would be a bit out of place in a French bistro.

Similarly, timeline items can provide a list of options pertaining to actions you can take on an item, such as reading aloud the text on the screen or sharing an item with a friend. Just like in a restaurant, some items make sense in context and some don't. A card that displays a video might be a strange place to offer Read Aloud.

A timeline item can have many menu items. Each menu item is attached to the timeline item itself as an array under `menuItems`. As usual, you can find the full list of menu-item options in Appendix 1, *HTTP and HTML Resources*, on page 283. The core option you'll need is `action`.

Actions

The first decision you'll make when adding a menu item is to decide what action the item will perform. Often this is the only decision you'll need to make. The `menuItems action` field can accept eight predefined actions, or is labeled as custom (covered in the next chapter).

We can use menu items to send voice messages and share a card, read a card aloud, place calls, display a navigation map from a card's `location` (covered in the next chapter), pin a card to a timeline, or delete it.

For example, let's add the ability to read aloud or delete a Lunch Roulette card.

chapter-04/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
public static void setSimpleMenuItems( TimelineItem ti, boolean hasRestaurant ) {
  // Add blank menu list
  ti.setMenuItems( new LinkedList<MenuItem>() );

  ti.getMenuItems().add( new MenuItem().setAction( "READ_ALOUD" ) );
  ti.getMenuItems().add( new MenuItem().setAction( "DELETE" ) );
}
```

The built-in menu actions do all of the heavy lifting. For example, if you add a DELETE action, when a user selects Delete on her Glass, Google will delete the timeline item for you. It does this by requesting a `DELETE` action of '/mirror/v1/timeline/{itemId}', not of your servlet. The only menu action that works a bit differently is Custom, which requires that you write your own code to respond to an event.

Please Let Us Delete

By default, cards you create have no menu. As you go out into the world and make your own Glassware, please add a Delete option to every timeline item you create. Without it, your Glassware users have no recourse to delete your items from their timelines. This causes a problem for those of us who obsessively purge our timelines of unused items. Your users will thank you.

Adding menu items can make demands on your code beyond simply adding the action. If your users can delete their timeline items, then your code may have to deal with the possibility that a timeline item is gone.

Pinning a Card

Whereas deleting a card changes whether it exists, pinning a card changes its state and where it lives on your Glass timeline. Pinning an item places it to the left of the home card, outside of the timeline. If you create an application with a single timeline item that's updated often, such as a stock ticker, pinning the item makes perfect sense.

We'll change up Lunch Roulette to operate differently in two cases. If we have a timeline item that a user has pinned, then we'll update that item inline. In any other case, we'll create a new one in the timeline.

chapter-04/LunchRoulette/src/test/book/glass/LunchRouletteServlet.java

```
public void doGet(HttpServletRequest req, HttpServletResponse resp)
    throws IOException, ServletException
{
  ServletContext ctx = getServletContext();
```

```
  String userId = SessionUtils.getUserId( req );
  TimelineItem timelineItem = LunchRoulette.getLastSavedTimelineItem(userId);

  // If it exists, isn't deleted, and is pinned, then update
  if (timelineItem != null
      && !(timelineItem.getIsDeleted() != null && timelineItem.getIsDeleted())
      && (timelineItem.getIsPinned() != null && timelineItem.getIsPinned()))
  {
    String html = LunchRoulette.renderRandomCuisine( ctx );
    timelineItem.setHtml( html );

    // update the old timeline item
    Timeline timeline = MirrorUtils.getMirror( userId ).timeline();
    timeline.patch( timelineItem.getId(), timelineItem ).execute();
  }
  // Otherwise, create a new one
  else {
    LunchRoulette.insertAndSaveSimpleHtmlTimelineItem( ctx, userId );
  }

  resp.setContentType("text/plain");
  resp.getWriter().append( "Inserted Timeline Item" );
}
```

We're calling some functions we created earlier. getLastSavedTimelineItem() retrieves the last timeline item ID saved for a user. If timelineItem exists, isn't deleted, and is pinned, then we update the HTML. Otherwise, we make a new item and save its id.

To test it, you need to create a Lunch Roulette timeline item by visiting /lunchroulette. Then on your Glass, go to the item, tap on the menu, then choose Pin. Glass will remove that item from your timeline, and place it to the left of your home card. Revisiting /lunchroulette will then update that card.

Besides being convenient for your users, pinning cards in this way and making them updatable can be a handy way to test during development. It saves you from having to chase down older cards in your timeline.

Cron Jobs

Up to this point, we've been manually visiting our /lunchroulette URL every time we want to add a new item to our timeline. This is hardly a production-worthy requirement. Instead, we want a new lunch suggestion automatically populated every day for each user of our Glassware.

So let's create a cron job that runs once a day and updates everyone's Glass with a suggestion for that day's lunch.

To do this in GAE, you'll need to create a cron.xml file under the WEB-INF directory. All this file needs to contain is an XML cron entry that has a URL to issue a GET request against and a schedule of when to run (we'll run every day at midnight GMT).

chapter-04/LunchRoulette/war/WEB-INF/cron.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<cronentries>
  <cron>
    <url>/lunchcron</url>
    <description>Load lunch suggestions for all users at midnight</description>
    <schedule>every day 00:00</schedule>
  </cron>
</cronentries>
```

You can find more information about the details of GAE's cron tasks on the developer website.[1]

Now we'll create a servlet that serves the /lunchcron endpoint. This is rather simple, since we've already written most of the code for our LunchRouletteServlet. The work in this servlet will be the same, but rather than run for one user, we iterate through all stored credentials and add a new timeline item for each user.

chapter-04/LunchRoulette/src/test/book/glass/LunchCronServlet.java

```
List<Key> removeKeys = new LinkedList<Key>();

DatastoreService datastore = DatastoreServiceFactory.getDatastoreService();
Query q = new Query("User");
PreparedQuery pq = datastore.prepare(q);
for( Entity result : pq.asIterable() )
{
  String userId = (String)result.getProperty("id");
  try {
    LunchRoulette.insertAndSaveSimpleHtmlTimelineItem(ctx, userId);
  } catch (GoogleJsonResponseException e) {
```

Before we upload this servlet, we must add an exception for /lunchcron to our AuthFilter code; otherwise, when cron attempts to execute the servlet it'll be redirected to the OAuth path.

With all that out of the way, after you upload the new servlet your Glass will autopopulate with a new lunch suggestion once a day.

To test your cron jobs, it's sometimes helpful to set the cron.xml schedule value to a higher frequency, like every 2 minutes. If anything goes wrong, visit

1. https://developers.google.com/appengine/docs/java/config/cron

https://appengine.google.com and choose your application. Both cron jobs and logs are in the menu and should be more than enough to troubleshoot any issues.

Wrap-Up

We're on a roll now. In this chapter we covered most of what an end user will ever see. We can create, update, read, and delete timeline items. We can stack multiple items by bundling, or let a user interact with a timeline item via a menu. We made the roulette card a bit more convenient by pinning it to the left of the timeline, and updating the same item on each visit.

Then we expanded beyond the regular reach of the Mirror API by using the GAE cron service to change the cuisine suggestion every day. In the next chapter we'll take this a bit further by replacing random cuisine names with real restaurants using Google Places.

Although the user sees and interacts with the timeline, there's a lot more going on in the back end. In the next chapter we'll take advantage of some Mirror services, such as action notifications and geolocation.

CHAPTER 5

Tracking Movement and User Responses

Everything we've done so far has been about presenting options to users. They authorize the app, and we show them cards and present menu items to them. This chapter is about what users don't see but they interact with for a deeper experience. These are back-end services that are affected by changes in a user's Glass state.

Back-end services in response to state changes are a common fixture in software applications. Imagine a case where you might consistently delete emails from the same address. With the subscription service, your Glassware can register to receive a notification of any timeline-item deletions. If a user deletes three emails from a given sender, that address is added to a spam list, which suppresses further timeline items added for that email address. Or, if you don't want your application to be so subtle, you can create a custom Spam menu item and subscribe to that, allowing users to be a bit more proactive. We'll take advantage of this behavior to create our own custom Lunch Roulette menu item.

With the location service you can get the current and past position of a Glass device. There are untold uses for geolocation. In our case, we'll use the latest location to find restaurants in the local area, and display a real nearby restaurant on the Lunch Roulette card with the help of Google Places. Just to show off maps a bit, we'll also display the location of the restaurant on a card.

Geolocation

No one likes to be lost. This helps explain why location services are estimated to be a $10 billion industry in 2015.[1] Geolocation in the Mirror API is the act

1. http://www.pyramidresearch.com/store/Report-Location-Based-Services.htm

of learning the location of a Glass device at some latitude/longitude point on planet Earth.

You application can use that information for all sorts of purposes, from getting directions to the cheapest fueling station to having the history of a nearby building projected onto your screen.

Your Glassware receives this information through the Location resource. But before your code can access a device's location, it must ask the user for permission.

OAuth Location Scope

In the previous couple of chapters we glossed over choosing the correct scope to access resources. Since all of our actions thus far have been on the timeline, the glass.timeline scope had been sufficient. But to access location resources only, without using the timeline, you should request the following scope:

https://www.googleapis.com/auth/glass.location

If you looked at the scopes in our AuthUtils code, we can use both scopes.

chapter-05/LunchRoulette/src/test/book/glass/auth/AuthUtils.java

```
public static final List<String> SCOPES = Arrays.asList(
  "https://www.googleapis.com/auth/userinfo.profile",
  "https://www.googleapis.com/auth/glass.timeline",
  "https://www.googleapis.com/auth/glass.location"
);
```

Practically speaking, the glass.timeline scope lets your app access a user's location. Using both Glass scopes doesn't hurt if you plan to reference location as well as the timeline. In fact, you should, since during authorization the user can see exactly what you'll be accessing.

One Location

When your application asks Google to track the location of a Glass device, Mirror provides the positions of the device over time, and assigns an ID to each location. The interesting bits of a Glass location object are the Glass device's latitude, longitude, and accuracy in meters. Like with all Mirror messages, id and kind will also be populated.

Location is a small but potent object, filled with plenty of information; you can find a complete listing in Appendix 1, *HTTP and HTML Resources*, on page 283.

Let's add a bit of geolocation to Lunch Roulette. To get the device's most recent location, pass in a special ID latest. If you want and have to get the ID of a certain location object, use that ID string instead.

chapter-05/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
Location location = mirror.locations().get("latest").execute();

double latitude = location.getLatitude();
double longitude = location.getLongitude();
```

The Java code will generate an HTTP GET request using the ID, and return a populated JavaScript Object Notation (JSON) body. Here's an example of what the latest location may generate.

```
GET /mirror/v1/locations/latest HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer e057d4ea363fbab414a874371da253dba3d713bc

{
 "kind": "mirror#location",
 "id": "latest",
 "timestamp": "2013-09-09T22:12:09.745Z",
 "latitude": 45.5142245,
 "longitude": -122.6807479,
 "accuracy": 49.0
}
```

You can't create or update a location yourself (the /locations resource accepts only GET actions, not POST, PUT, or DELETE). However, you don't need to, since Google populates locations for you.

Many Locations

If you want the history of a Glass device's movement, you can access a list of previous locations. We won't access a full list in Lunch Roulette (suggesting a restaurant near yesterday's position seems somewhat uncongenial), but it's useful to know how.

In Java, executing location.list() returns a LocationsListResponse object. You can get items via the getItems() method, which returns an iterable List.

```
locations = service.locations();
LocationsListResponse locList = locations.list().execute();
for ( Location loc : locations.getItems()) {
  System.out.println(loc);
}
```

You'll receive a JSON response with a list of locations (for some reason called items). Each location will contain an id, which your app could store to later retrieve a specific location.

```
GET /mirror/v1/locations HTTP/1.1
Host: www.googleapis.com
Authorization: Bearer e057d4ea363fbab414a874371da253dba3d713bc

{
 "kind": "mirror#locationsList",
 "items": [
  {
   "kind": "mirror#location",
   "id": "latest",
   "timestamp": "2013-09-09T22:22:08.640Z",
   "latitude": 45.5142245,
   "longitude": -122.6807479,
   "accuracy": 49.0
  }
 ]
}
```

Glass sends a location update every 10 minutes, so you may notice that location timestamps are about that length apart. The 10-minute window is a value built into the Glass device's software. Since there's no guarantee that a user hasn't modified her Glass in some way, or that she even has consistent Internet access, you can't ever count on an even 10-minute spread. Google prunes this list every so often, so you shouldn't have to worry about receiving too many items.

Using Location

With a location in hand, we can optionally display the location in a map by using a glass://map path, or provide a menu option to get directions from where a user currently is to a destination. The next two sections are examples of what you can do with a Glass's location. There are innumerably more.

Navigating to a Location

Now that you have the location of your Glass user, why not expand Lunch Roulette a bit? Rather than presenting the user with a random cuisine, you can use her location along with a map service like Yelp or Google Places to present the local restaurants.

The code necessary to search Google Places for a nearby cuisine is wrapped up into a PlaceUtils class as part of the book's downloadable code. The getRandom()

method chooses a random restaurant by searching for a nearby cuisine at a given latitude and longitude location.

You'll need to activate the Places API on your Google console.[2] This is in the same API console where you activated the Mirror API in Chapter 2, *The Google App Engine PaaS*, on page 13.[3] Then click on API Access and find the API key under Simple API Access. Set the AuthUtils.API_KEY constant in your Java code to that key.

chapter-05/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
// get a nearby restaurant from Google Places
Place restaurant = getRandomRestaurant(latitude, longitude);
// create a timeline item with restaurant information
TimelineItem timelineItem = new TimelineItem()
  .setHtml( render( ctx, "glass/restaurant.ftl", restaurant ) )
  .setTitle( "Lunch Roulette" )
  .setMenuItems( new LinkedList<MenuItem>() )
  .setLocation(
    new Location()
      .setLatitude( restaurant.getLatitude() )
      .setLongitude( restaurant.getLongitude() )
      .setAddress( restaurant.getAddress() )
      .setDisplayName( restaurant.getName() )
      .setKind( restaurant.getKind() ) );
// Add the NAVIGATE menu item
timelineItem.getMenuItems().add(
  new MenuItem().setAction( "NAVIGATE" )
);
// get a nearby restaurant from Google Places
Place restaurant = getRandomRestaurant(latitude, longitude);
// create a timeline item with restaurant information
TimelineItem timelineItem = new TimelineItem()
  .setHtml( render( ctx, "glass/restaurant.ftl", restaurant ) )
  .setTitle( "Lunch Roulette" )
  .setMenuItems( new LinkedList<MenuItem>() )
  .setLocation(
    new Location()
      .setLatitude( restaurant.getLatitude() )
      .setLongitude( restaurant.getLongitude() )
      .setAddress( restaurant.getAddress() )
      .setDisplayName( restaurant.getName() )
      .setKind( restaurant.getKind() ) );
// Add the NAVIGATE menu item
timelineItem.getMenuItems().add(
  new MenuItem().setAction( "NAVIGATE" )
);
```

2. https://developers.google.com/places/documentation/
3. https://code.google.com/apis/console/

Using the restaurant object, you can populate HTML with a real restaurant name and address. Better yet, you can let users add a NAVIGATE menu-item action, which renders a Get Directions menu item like in the following figure.

Figure 19—Getting directions

Tapping on the new Get Directions menu item will bring up a map to the restaurant, as you can see in the figure here..

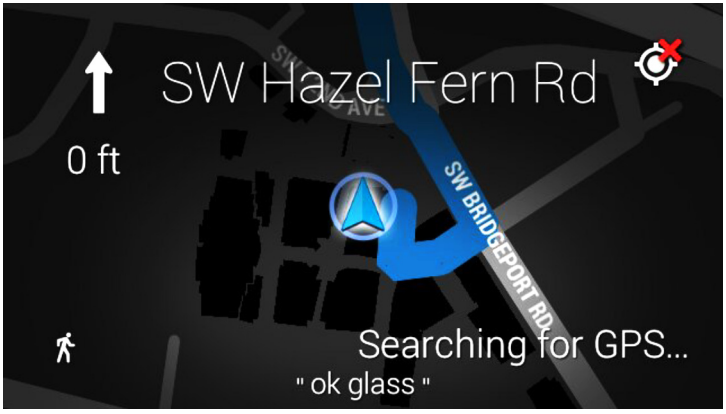


Figure 20—Navigation shows a map.

But the map displayed is that of a default map application that takes over your card. If you want to show a more customized map as part of a card, you can embed a map resource.

Showing Your Own Map

Since Glass is primarily a visual device, displaying information is always better than describing it. Let's enhance the Lunch Roulette timeline item to flag the chosen eatery on a map, presenting it to the user on half of the card, with the restaurant name on the other half.

The map parameters are a subset of Google's Static Maps API.[4] Width and height (w and h) are required params, and at least one of the following is required: center and zoom, marker, and polyline. You can read more details about these parameters in Appendix 1, *HTTP and HTML Resources*, on page 283.

But rather than calling the static map image–generating HTTP URL (for instance, http://maps.googleapis.com/maps/api/staticmap), we call the map action of the Glass protocol, which starts with glass://map. This allows Glass to natively generate the map rather than relying on the Google Maps API to transport a map image.

The cool part is that we can add a map simply by adding it to our template file. No code change is necessary, since we designed the Place class (restaurant object) to have a getter and a setter for latitude and longitude.

chapter-05/LunchRoulette/war/WEB-INF/views/glass/restaurant-map.ftl

```
<article>
<figure>
  <img src="glass://map?w=240&h=360&marker=0;${latitude},${longitude}&marker=1"
    width="240"
    height="360"/>
</figure>
<section>
  <h2 class="yellow">${ name }</h2>
  <strong>${ address }</strong>
</section>
</article>
```

When you rerun the project, you should notice a dramatic change on your Glassware card (see Figure 21,*Displaying a map alongside a restaurant*, on page 66). No longer are we stuck with plain text, or slightly less-plain HTML.

Now we have can display our own maps! This, along with the NAVIGATE menu item and Location resource, lets us visualize or navigate to and from any location, the two the most common uses of a map.

4. https://developers.google.com/maps/documentation/staticmaps/

Figure 21—Displaying a map alongside a restaurant

Subscriptions

The point of a subscription is to respond in some way to some event. If you have ever subscribed to a newspaper, you're familiar with the concept. Every time a new paper is available, you are notified by getting a copy with that day's articles. Rather than notifying you on a fixed basis (such as daily) Glass gives you a more real-time notification.

In Mirror's terms, whenever a state changes, you are notified. But to be notified you must subscribe, and for that, you use the Subscription resource. Your application can subscribe to several combinations of Glass events and actions, all of which can be found in Appendix 1, *HTTP and HTML Resources*, on page 283. You can create a new subscription for either a `timeline` or `location` event, called a `collection`, but you require the `glass.timeline` scope in your authorization.

Creating

In Lunch Roulette, we want to know if our timeline items are being pinned. We can't easily know if someone loves our app, but we can assume if the user has pinned a timeline item, he doesn't hate it. Let's subscribe to timeline events when a timeline item operation is UPDATE, since pinning a timeline item will update the `isPinned` field.

Unlike with a `timeline` or `location` request, you'll subscribe a user account to a notification only once. A great time to run this kind of code is during the authentication steps, so we'll add this code to the `OAuth2Servlet` callback.

Our first step is to populate the callbackUrl for the subscription. Although technically only a single URL is required, we need one for production and one for testing our app locally.

The production callback choice is the easy URL to build—just set a PROD_CALLBACK constant to something like https://glassbooktest.appspot.com/timelineupdatecallback, replacing glassbooktest with your own Google App Engine (GAE) subdomain. If you've been testing your Glassware so far by deploying it to GAE, you'll set your TEST_CALLBACK to the same value.

If you want to test against your local machine, note that callbackUrl must start with https://, which can be a problem if your test server doesn't accept secure requests. Google provides a development redirect resource at https://mirrornotifications.appspot.com/forward, which takes a url parameter to proxy.

Since the 'mirrornotifications' app doesn't exactly know about your laptop, you can't use localhost. Instead use a publicly reachable IP address and port or a dynamic domain name system (DNS).

Routing Requests to Your Test Machine

Since your test machine will have to be reachable from the Internet at large, testing a callback locally can be tricky. If you have a public IP address open to the Internet, set your URL to *http://YOUR_PUBLIC_IP/timelineupdatecallback*. More than likely you'll have a firewall, and if you have the ability to set up a DMZ and don't mind the risk involved, you can set that on your router.

Otherwise, you would be best served setting up a DNS redirect service, like noip.com. Many are free, and some allow port 80 redirects to your development port, like 8888.

All of these dynamic DNS services basically work the same, where you install some software and run a server that routes requests to some domain name like *glassbooktest.no-ip.biz* to your laptop over port 80.

If the service you choose does not offer port forwarding, you'll have to either set up another local server, like nginx, to forward port 80 requests to your development port or run your development server on port 80, which usually requires you to run your test Java web server as a privileged user like root. As I said, tricky.

With your callback URLs built, choose whether this code running isDevelopment() requires checking if the SystemProperty.environment.value() equals SystemProperty.Environment.Value.Development.

Now that you have the variables set up, creating a subscription is straightforward. You set the callback, timeline collection, and any operations.

chapter-05/LunchRoulette/src/test/book/glass/LunchRoulette.java
```
final String callbackUrl = isDevelopment() ? TEST_CALLBACK : PROD_CALLBACK;

Subscription tliSubscription = new Subscription()
  .setCallbackUrl( callbackUrl )
  .setVerifyToken( "a_secret_to_everybody" )
  .setUserToken( userId )
  .setCollection( "timeline" )
  .setOperation( Collections.singletonList( "UPDATE" ) );

mirror.subscriptions().insert( tliSubscription ).execute();
```

If you subscribe to the same collection and operations list, Mirror will update the other values, like the callbackUrl, rather than create a new subscription. You can verify this by listing all subscriptions after resubscribing multiple times.

```
SubscriptionsListResponse subscriptions = mirror.subscriptions().list().execute();
for (Subscription subscription : subscriptions.getItems()) {
  System.out.println( subscription );
}
```

This is a useful feature to halt accidental double, triple, or thousands of resubscriptions triggering a single callback more than once.

Getting, Updating, or Deleting

Getting an existing subscription is as simple as calling the get() method with the object's ID. It's a string, so you can store it easily enough. With an existing subscription object, you can make any changes you want, and update it using the update() method along with the ID.

Finally, you can call delete() any time you no longer need to subscribe. If a user ever de-authorizes your application, any subscriptions associated with that user will be removed from your subscriptions resource.

Accepting Notifications

For a newspaper subscription, your house is like the callback URL. When the paper arrives, you can take any action you wish in response, from reading the paper to lining a birdcage with it. The callback URL you provide will receive a POST notification JSON object, allowing your service to take some action when the subscribed event has happened.

Google expects your service to respond with a 200 HTTP code within 10 seconds. This is a pretty standard requirement for web-service callbacks. If you

need to execute a long-running job, you can put it in a task queue to be executed asynchronously.

Let's dig into the steps required to accept notifications; we'll craft a callback servlet that accepts notifications for the PIN menu action, as well as look at how to track a Glass device's location on the move.

Making a Callback Servlet

We need to create a new servlet to capture the /timelineupdatecallback POSTs that the Mirror API will request based on the subscription we made in the preceding section.

Unlike other actions we've taken so far, this servlet is called *from* the Mirror API with JSON bodies, rather than making requests *to* the Mirror API. So we have to extract data from the body of the request. Happily, the Mirror Java client comes equipped with a Notification class that can be correctly populated with JSON data using the JsonFactory object.

chapter-05/LunchRoulette/src/test/book/glass/notifications/TimelineUpdateServlet.java

```
public class TimelineUpdateServlet extends HttpServlet
{
  protected void doPost(HttpServletRequest req, HttpServletResponse res)
      throws IOException, ServletException
  {
    // Generate Notification from request body
    JsonFactory jsonFactory = new JacksonFactory();
    Notification notification =
      jsonFactory.fromInputStream( req.getInputStream(), Notification.class );

    // Get this user action's type
    String userActionType = null;
    if( !notification.getUserActions().isEmpty() )
      userActionType = notification.getUserActions().get(0).getType();

    // If this is a pinned timeline item, log who and which timeline item
    if( "timeline".equals( notification.getCollection() )
        && "UPDATE".equals( notification.getOperation() )
        && "PIN".equals( userActionType ) )
    {
      String userId = notification.getUserToken();
      String itemId = notification.getItemId();

      System.out.format( "User %s pinned %s", userId, itemId );
    }
```

JSON populates the notification object with many of the same fields as the /subscriptions resource body, like collection, operation, verifyToken, and userToken. This

is no accident, since when you subscribe to a timeline collection UPDATE operation for a given userToken, your callbackUrl will receive those populated values.

You can read about all valid Notification fields in Appendix 1, *HTTP and HTML Resources*, on page 283.

Just to be extra safe, we'll check that this notification is a timeline update caused by a user's PIN action. Each userAction is associated with some timeline-item operation, most of the time updating an existing item. Our code tested for the PIN userAction, but notifications provide several more options, which you can find in Appendix 1, *HTTP and HTML Resources*, on page 283.

UserAction contains only a type and a payload, but normally we only care about the type. The payload is useful only in one circumstance: a CUSTOM action type. We'll cover custom menu items in the next section.

Although we're merely logging the user ID and item ID, we could have done anything with this data—from storing a pin notification in our database to going a little overboard and sending the user an "Achievement: Congratulations on Pinning a Card" email.

Also note that we didn't actually need to check our notification collection or operation, since this callback should only ever be called for timeline updates. However, checking the user action is necessary, since you cannot subscribe to a particular action. If a PIN user action type occurs, you do one thing; if a CUSTOM action occurs, you do something else.

Location Subscriptions

Subscribing to location events is nearly identical to subscribing to timeline events. The benefit of subscribing to location events rather than calling the location's resource directly is that Mirror will trigger your code when your user's Glass device is in a new location.

This affords you some interesting behaviors, like the ability to label nearby points of interest in something resembling real time (within a 10-minute window, that is).

One of the first Glassware apps to take advantage of this behavior is Field Trip,[5] which creates a timeline item with details about a particular place of interest whenever you venture nearby. For example, I learned that a building I walk by on a daily basis housed the oldest bar in Portland.

5. http://www.fieldtripper.com/

The difference from the timeline is that the itemId field is populated with the latest location ID, which is, of course, latest.

```
{
  "collection": "locations",
  "itemId": "latest",
  "operation": "UPDATE",
  "userToken": "1234567890",
}
```

To get the location data, you'd call mirror.locations().get("latest").execute(); just like we did at the beginning of this chapter.

Custom Menu Items

Let's circle back around to menu items, since we're now experts at writing callbacks. All of the menu items we've covered so far have been predefined. You can define your own custom menu items by setting the menu item action to CUSTOM.

The notifications are the same as subscriptions, with one minor twist. A menu callback sends a userAction[].payload value, which contains the id of the custom menu item that was selected. This lets your application know which item was chosen. This may seem like overkill since you could always give each menu item its own distinct callbackUrl. But this gives you the option of creating a single menu callback to deal with multiple custom choices.

So far, Lunch Roulette has randomly chosen a cuisine once a day by using our cron job or visiting /lunchroulette to test. But what if the user doesn't like a chosen pizza place, or would really rather not have Greek for lunch? Let him choose an alternative by way of a CUSTOM menu item.

In the code that sets the Lunch Roulette menu item, add the following new menu item. What makes this different from every other menu we've created so far is a list of values.

chapter-05/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
// Set up two menu values for DEFAULT and PENDING
List<MenuValue> menuValues = new ArrayList<MenuValue>(2);
menuValues.add( new MenuValue()
  .setState( "DEFAULT" )
  .setDisplayName( "Alternative" ) );
menuValues.add( new MenuValue()
  .setState( "PENDING" )
  .setDisplayName( "Generating Alternative" ) );
// Add new CUSTOM menu item
timelineItem.getMenuItems().add( new MenuItem()
  .setAction( "CUSTOM" )
  .setValues( menuValues )
);
```

We're setting a DEFAULT state that displays the text *Alternative*. When a user taps on the custom menu item, the state changes to PENDING, whereby the menu will display *Generating Alternative*. Once the menu action is completed, if you set a CONFIRMED value, the name can change again. Since we didn't set CONFIRMED, the DEFAULT name will display instead.

Next, we need to subscribe to the custom menu item being selected. Happily we can piggy-back on the timeline collection UPDATE operation subscription we inserted as part of user auth in the OAuthServlet. This saves us a step, since we can use the existing LunchRouletteCallback as the callback URL. We'll just need to add some branching logic to run only if a user triggers a CUSTOM CUISINE action on a timeline.

chapter-05/LunchRoulette/src/test/book/glass/notifications/TimelineUpdateServlet.java

```
else if( "timeline".equals( notification.getCollection() )
    && "UPDATE".equals( notification.getOperation() )
    && "CUSTOM".equals( userActionType ) )
{
```

When our custom menu item is selected, we want to do two things: First, we get the bundle ID from the timeline item that called the menu item. If it doesn't have a bundle ID, generate one and set it. Second, we create another timeline item with a new restaurant and the same bundle ID, which threads the two items.

chapter-05/LunchRoulette/src/test/book/glass/notifications/TimelineUpdateServlet.java

```
UserAction userAction = notification.getUserActions().get(0);
if( "ALT".equals( userAction.getPayload() ) )
{
  // Add a new timeline item, and bundle it to the previous one
  String userId = notification.getUserToken();
  String itemId = notification.getItemId();
  Mirror mirror = MirrorUtils.getMirror( userId );
  Timeline timeline = mirror.timeline();
  // Get the timeline item that owns the tapped menu
  TimelineItem current = timeline.get( itemId ).execute();
  String bundleId = current.getBundleId();
  // If not a bundle, update this item as a bundle
  if( bundleId == null ) {
    bundleId = "lunchRoulette" + UUID.randomUUID();
    current.setBundleId( bundleId );
    timeline.update( itemId, current).execute();
  }
  // Create a new random restaurant suggestion
  TimelineItem newTi =
    LunchRoulette.buildRandomRestaurantTimelineItem( getServletContext(), userId );
  newTi.setBundleId( bundleId );
  timeline.insert( newTi ).execute();
}
```

After launching this code and generating a new Lunch Roulette item, in the list you should see a new menu item called Alternative. If you click on that menu item, a different restaurant item will be added to your timeline, bundled to the current card.

Wrap-Up

Location and subscription resources are passive activities that allow your software to act as a background service. Your apps can take actions based on what users do, without necessarily requiring that the user take an active role. The sole outliers here are custom menu items, which the user will actively choose, and your subscription will allow your app to produce a response when notified by Mirror.

We've now covered most of the Mirror API; external assets, such as contacts or timeline-item attachments, are up next.

CHAPTER 6

Making Glass Social

Needless to say, computers have evolved tremendously in the past few decades. Twenty years ago, processors were slower and ran hotter, memory was expensive, and displays were heavy and lower quality. But for many consumers, the biggest changes have been around networking. The advent of social networking raised an entire generation into savvy computer users and gave us the ability to express ourselves to friends and the world by sharing photos and videos. This ability is no longer an optional attribute of a mobile computing device.

This chapter covers the remainder of the Mirror API, which allows us to add depth to our application beyond a simple list of cards in a timeline. Contacts and attachments are enhancements to a Glassware. Unlike timeline items or menus, which are the basic necessary building blocks of Glassware, these final puzzle pieces will allow our application to store, retrieve, and share assets with others entirely though Glass. These are the tools we can use to add a social-network dimension to our Glassware.

Creating Contacts

A necessary component of any social activity is other contacts to share with. A contact, in the most general sense, is an individual or a group of things (be they humans or maybe even cats). It's a conceptual construct on who or what you can share with, and not necessarily other Glass users. There are two sides to contacts in the Mirror API: the contacts resource where users can manage their personal *contacts*, and timeline-item integration, where a contact is the *creator* or *recipient* of an item.

A contact is data about a person or group, like `displayName`, `phoneNumber`, or `imageUrls`. It represents someone you can share timeline items with, or someone you wish to more easily message or call on a regular basis.

Since Lunch Roulette is about connecting a potential patron (Glass user) with a restaurant, it stands to reason a user may want to add a restaurant as a contact, especially if the user plans to eat there at a later date. Let's add another custom menu option to a timeline item to allow a user to add the current restaurant timeline item to his contact list. It's just like our last custom menu item, but this time we'll give it the id of ADD_CONTACT to differentiate it from our Alternative menu item from the last chapter.

chapter-06/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
timelineItem.getMenuItems().add( new MenuItem()
    .setAction( "CUSTOM" )
    .setId( "ADD_CONTACT" )
    .setRemoveWhenSelected( true )
    .setValues( Collections.singletonList( new MenuValue()
      .setState( "DEFAULT" )
      .setDisplayName( "Add As Contact" ) )
    )
);
```

This adds a menu item to tap (and then removes it once it's chosen), but we'll need to know some information about the restaurant in order to create a contact. We have a couple of options for storing that information. Either we can use sourceItemId to set a unique ID on the timeline item to represent a particular restaurant, or we can set the contact to the timeline item.

You may notice that TimelineItem does *not* have a setContact method, but it does allow us to setCreator(). This is a bit of a double punch, since not only does this store our contact, allowing us to retrieve it later, but it also lets us leverage another menu-item action, called VOICE_CALL. This will make a call directly from the timeline item, since it *call*s the creator's phone_number.

chapter-06/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
timelineItem.getMenuItems().add(
    new MenuItem().setAction( "VOICE_CALL" ) );
```

Next let's set the restaurant contact on the timeline item. Since we have a restaurant's name and phone number, we can add those to the contact. The contact type can be either an INDIVIDUAL (the default) or a GROUP. We'll keep it as INDIVIDUAL, so it will show up in your Call menu-item contact list.

chapter-06/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
TimelineItem timelineItem = new TimelineItem()
  .setCreator(
    new Contact()
      .setDisplayName( restaurant.getName() )
      .setPhoneNumber( restaurant.getPhone() )
      .setType( "INDIVIDUAL" ) )
```

Mirror will call the TimelineUpdateServlet when the menu item is tapped, due to the subscription we created at the end of Chapter 5, *Tracking Movement and User Responses*, on page 59. Now we need to add another if statement to specify what to do with an ADD_CONTACT payload.

With the menu items and contact set, creation of this timeline item will populate a hidden display name and phone number. You can reference this with a regular timeline.get(), then pass that same contact object to the contact resource's insert() once, adding a unique ID. Unlike for other objects we've created (TimelineItem, Subscription), you are responsible for setting this contact's unique ID.

chapter-06/LunchRoulette/src/test/book/glass/notifications/TimelineUpdateServlet.java

```
else if( "ADD_CONTACT".equals( userAction.getPayload() ))
{
  Mirror mirror = MirrorUtils.getMirror( userId );
  Timeline timeline = mirror.timeline();
  Contacts contacts = mirror.contacts();

  TimelineItem timelineItem = timeline.get( itemId ).execute();

  Contact contact = timelineItem.getCreator();
  contact.setId( UUID.randomUUID().toString() );

  contacts.insert( contact ).execute();
}
```

Like all other resources with an insert() method, this generates an HTTP PUT with a contact's populated JavaScript Object Notation (JSON) object. You can see all contact fields in Appendix 1, *HTTP and HTML Resources*, on page 283.

After you've deployed and tested your new menu action, you should see a new contact in your list. You can view your list in code via contacts.list().execute(), which returns a ContactListResponse object (similar to LocationsListResponse or 'TimelineListResponse').

```
ContactListResponse list = contacts.list().execute();
for( Contact item : list.getItems() ) {
  System.out.println( item );
}
```

As with other resources, you can individually get, update, or delete each contact.

Sharing Assets with Glassware

Contacts are meant to be more than simply people or businesses. Your Glassware itself can actually be a contact. This means your users can share

items with a Glassware application. In my opinion, this was an odd design choice on Google's part, overloading Contacts to represent living individuals or groups *and* a software application. However, sharing a timeline item, such as images or video, requires a contact to share it with. So let's make do.

If you create a contact, coupled with a subscription to a SHARE event, then your Glassware can respond to notifications from timeline items that your application didn't create, such as photographs.

We'll expand Lunch Roulette again, this time by letting users share photos they've taken of their food. Once they take a photo, they can choose to share it with Lunch Roulette (the Twitter and Facebook apps function similarly). After a shared timeline item is created, our callback URL will be notified. For now we'll just log this notification to expand upon later in this chapter.

Let's begin by creating a new contact that represents Lunch Roulette. Just like we added a subscription after a successful user login, we can add this app as a contact.

chapter-06/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
public static void addAppAsContact( HttpServletRequest req, String userId )
    throws IOException
{
  Mirror mirror = MirrorUtils.getMirror(req);
  Contacts contacts = mirror.contacts();

  Contact contact = new Contact()
    .setId( "lunch-roulette" )
    .setDisplayName( "Lunch Roulette" )
    .setImageUrls( Collections.singletonList(
        PROD_BASE_URL + "/static/images/roulette.png" ) )
    .setPriority( 0L )
    .setAcceptTypes( Collections.singletonList( "image/*" ) );

  contacts.insert( contact ).executeAndDownloadTo( System.out );
}
```

This code is called in the OAuth2Servlet's doAuth method after a user has successfully logged in.

The crux of our contact object is the acceptTypes field. This means this contact will be displayed whenever an image MIME type has a menu-item Share action. To accept video, you could add the video/* wildcard type.

This contact's displayName will print out, but we'd like to make our contact a bit more visually appealing by giving this contact a picture. The setImageUrls() method takes a list of image URLs. We can host this image as a static file in

Google App Engine (GAE) by adding these lines to the GAE configuration file (appengine-web.xml). The code bundled with the book places them all under /static/images.

chapter-06/LunchRoulette/war/WEB-INF/web.xml

```
<static-files>
        <include path="/**.jpg" />
        <include path="/**.png" />
</static-files>
```

Next you need to be notified of any SHARE actions. When your user taps Share in a timeline-item menu, the Mirror service will create a copy of the timeline item being shared. It makes a copy because you can potentially make changes to that copy, such as by speaking, "OK, Glass, add a caption…" then whatever you say after that will display superimposed on that image, without changing the original. That means that to be notified of SHARE actions, you must subscribe to timeline INSERT operations.

Let's create a new callback servlet for timeline INSERT operations, with the URL path of /timelineinsertcallback.

chapter-06/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
final String tiCallbackUrl = isDevelopment() ?
    TEST_TI_CALLBACK : PROD_TI_CALLBACK;

Subscription tiSubscription = new Subscription()
  .setCallbackUrl( tiCallbackUrl )
  .setVerifyToken( "a_secret_to_everybody" )
  .setUserToken( userId )
  .setCollection( "timeline" )
  .setOperation( Collections.singletonList( "INSERT" ) );

mirror.subscriptions().insert( tiSubscription ).execute();
```

Finally, we need to create the new TimelineInsertServlet callback. The setup code in the doPost method is almost exactly the same as TimelineUpdateServlet. We extract the notification object from the request body, and grab the user and item IDs. Then we ensure this is a SHARE action, which we only log for now.

```
if( "SHARE".equals( userActionType ) )
{
  System.out.format( "User %s shared %s", userId, itemId );
}
```

When you launch the app and log in again, it can take a few minutes before the Lunch Roulette contact shows up on your contact list. Since we're setting the contact's ID, successive logins won't create the contact more than once. When it's all set up, take a photograph. If you tap on Share in the image's

menu, you'll be greeted by a new contact called Lunch Roulette, which should look like the following figure.

Figure 22—Lunch Roulette contact card

Selecting that contact will trigger a notification, and if you look in your GAE logs you should see a request to /lunchcallback with a nice message that your user shared a timeline item. You can get to these logs by visiting your GAE apps page,[1] selecting your app, then choosing Logs, as the following figure shows.

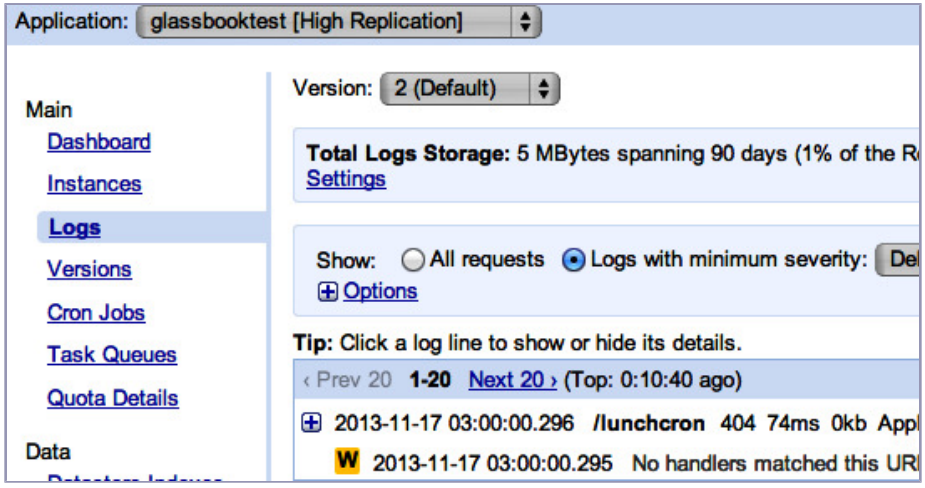


Figure 23—Google App Engine logs

It may not seem like we've done very much in this section, but we've come a long way since starting on the Mirror API. We stored a contact in our timeline item, created a custom menu, subscribed to custom menu taps, and inserted

1. https://appengine.google.com

a new restaurant contact in the list. We also integrated our app into the user experience a bit more by making it a target that a user can share images with—all in a few lines of code. There's more to do with contacts, such as updating, getting, and deleting them. But you've seen that done before in the timeline and subscription resources, and here it's exactly the same. So let's leave those details as your own exercise, and move on to multimedia attachments.

Getting and Setting Attachments

At its core, Glass is a visual device. Without its heads-up optical display, people probably wouldn't have much interest in Glass. You could tape a smartphone to the side of your face for a similar effect. Since the eyepiece makes all the difference, your applications should make every effort to present rich visual experiences. Attachments are how Glass sends and receives assets like pictures and videos.

Hopefully by now you're getting the hang of the timeline. We've seen nearly every aspect of timeline items, from displaying rich cards and bundles to creating menus and even attaching contacts. Attachments are the last remaining piece, and are our first foray into dealing with binary (non-JSON) data.

Getting an Attachment

Let's continue where we left off, after a user taps Share on an image menu. When the TimelineInsertServlet is notified about the SHARE action, we can extract that image attachment and store it. What should we do with that image? How about storing it along with its location, and when another Lunch Roulette user walks near that location, the image of food will pop up as a new timeline item? It's sort of like an ever-present public billboard of restaurant photos that stretches to anywhere on the globe!

First we get the timeline item that the SHARE action created. The TimelineItem's getAttachments() method returns a list of attachment ID strings. A timeline item can hold any number of attachments, but in our case we know the SHARE action is triggered only for the Lunch Roulette contact on an image, which will only ever return one.

Just like retrieving any other resource object, getting an Attachment from a TimelineItem is a matter of a simple get() method from a Timeline's Attachments. You'll use both the timeline-item ID and the attachment ID to download the

image data. You'll also need the image's content type. It could be a JPEG, PNG, GIF…who knows?

chapter-06/LunchRoulette/src/test/book/glass/notifications/TimelineInsertServlet.java

```
System.out.format( "User %s shared %s", userId, itemId );

Mirror mirror = MirrorUtils.getMirror( userId );
Timeline timeline = mirror.timeline();
Attachments attachments = timeline.attachments();

// get the newly created timeline item that triggered this notification
TimelineItem timelineItem = timeline.get( itemId ).execute();

// get the first attachment's id and content type
Attachment attachment = timelineItem.getAttachments().get(0);
String contentType = attachment.getContentType();
String attachmentId = attachment.getId();

// download the attachment image data
ByteArrayOutputStream data = new ByteArrayOutputStream();
attachments.get(itemId, attachmentId).executeMediaAndDownloadTo( data );
```

Note that we used a new execute method, executeMediaAndDownloadTo(), which pulls data into a ByteArrayOutputStream. Another common way to download attachment data is to call executeAsInputStream(), which returns an InputStream. But we chose the ByteArrayOutputStream so that we can easily get the data as a byte array and store it into our GAE DataStore as a Blob.

To store this image, we'll give it a key based on a 110-meter geographic boundary near the user's current location. We can achieve that with a neat trick of rounding the user's latitude and longitude values to the nearest three decimal places.[2]

chapter-06/LunchRoulette/src/test/book/glass/notifications/TimelineInsertServlet.java

```
byte[] image = data.toByteArray();
Blob blob = new Blob( image );
// Save the image in the datastore
DatastoreService store = DatastoreServiceFactory.getDatastoreService();

// Get the current location of Glass
Location location = mirror.locations().get("latest").execute();
String keyValue = String.format("%.3f,%.3f",
    location.getLatitude(), location.getLongitude());
// Store the image, type, and hash
Key key = KeyFactory.createKey("image", keyValue);
Entity entity = new Entity( key );
entity.setProperty("image", blob);
```

2. http://en.wikipedia.org/wiki/Decimal_degrees#Accuracy

```
entity.setProperty("type", contentType);
entity.setProperty("hash", LunchRoulette.toSHA1( image ));
store.put(entity);
```

This allows us to subscribe to a Lunch Roulette user's location. When the callback is notified of a change, the user's current location can be rounded to a nearby location. (We'll call this user *Alice*.) If any matching value exists in the datastore, that means another user (*Bob*) saved a photo at a nearby location, and then we can add that image to Alice's timeline. This is a very basic boundary query (rounded to the nearest 110 meters) turned into a simple key lookup.

The next step is to add a location collection subscription for UPDATE operations to the LunchRoulette.subscribe(), which you may recall is called by OAuth2Servlet's post-login procedure.

chapter-06/LunchRoulette/src/test/book/glass/LunchRoulette.java

```
final String locCallbackUrl = isDevelopment() ?
    TEST_LU_CALLBACK : PROD_LU_CALLBACK;

Subscription locSubscription = new Subscription()
  .setCallbackUrl( locCallbackUrl )
  .setUserToken( userId )
  .setCollection( "locations" )
  .setOperation( Collections.singletonList( "UPDATE" ) );

mirror.subscriptions().insert( locSubscription ).execute();
```

After a user has signed up for Lunch Roulette, every 10 minutes or so her latest location will trigger a notification (provided it has changed) to LocationsUpdateServlet. These notifications present themselves as UPDATE operations on the locations collection. From there, we get the given location and generate the same key that potentially contains an image.

Getting the entity from the datastore will throw an exception if no image exists. However, if an entity does exist, we merely need to unpack the data in the reverse of how we packed it, by getting the image data blob and content type. You'll then use the image data bytes and content type to build an InputStreamContent object, and insert it and a new TimelineItem into the user's timeline.

chapter-06/LunchRoulette/src/test/book/glass/notifications/LocationsUpdateServlet.java

```
// create a key of the user's current location
Location location = locations.get( itemId ).execute();
String positionKey = String.format("%.3f,%.3f",
    location.getLatitude(), location.getLongitude());

DatastoreService store = DatastoreServiceFactory.getDatastoreService();
```

```
// Build a key to retrieve the image, and store that it's been shown
Key imageKey = KeyFactory.createKey("image", positionKey);
Key shownKey = KeyFactory.createKey(imageKey, "shown", userId);

String shownImageHash = null;
try {
  Entity shown = store.get( shownKey );
  shownImageHash = (String)shown.getProperty("hash");
} catch( EntityNotFoundException e ) {
  // the user has never seen this, carry on
}
```

chapter-06/LunchRoulette/src/test/book/glass/notifications/LocationsUpdateServlet.java

```
try {
  // Get the image data from the store
  Entity entity = store.get( imageKey );
  String contentType = (String)entity.getProperty("type");
  Blob blob = (Blob)entity.getProperty("image");
  String imageHash = (String)entity.getProperty("hash");
  byte[] image = blob.getBytes();
  ByteArrayInputStream data = new ByteArrayInputStream( image );

  // only show this if the image hash is different
  if( shownImageHash != null && shownImageHash.equals(imageHash)) {
    // nothing to do here, we've already see this
    return;
  }

  // create and insert a new timeline item with attachment
  TimelineItem timelineItem = new TimelineItem()
    .setText( "Compliments of Lunch Roulette" )
    .setMenuItems( Collections.singletonList(
        new MenuItem().setAction( "DELETE" )));
  InputStreamContent content = new InputStreamContent( contentType, data );
  timeline.insert( timelineItem, content ).execute();

  // save that this user has seen this image at this location
  Entity shown = new Entity( shownKey );
  entity.setProperty("hash", imageHash);
  store.put( shown );
} catch( EntityNotFoundException e ) {
  // there was no matching image found
}
```

Under the hood, this is a bit different from how timeline REST requests were built in Chapter 4, *Building the Timeline*, on page 41. Rather than simply posting a JSON object, it will POST a single multipart message that contains both the timeline-item JSON as well as the binary data for the image.

```
POST /upload/mirror/v1/timeline HTTP/1.1
```

```
Host: www.googleapis.com
Authorization: Bearer {auth token}
Content-Type: multipart/related; boundary="gc0p4Jq0M2Yt08jU534c0p"
Content-Length: 4096
--gc0p4Jq0M2Yt08jU534c0p
Content-Type: application/json; charset=UTF-8

{ "text": "Compliments of Lunch Roulette" }

--gc0p4Jq0M2Yt08jU534c0p
Content-Type: image/png
Content-Transfer-Encoding: binary

...lots of bytes...
--gc0p4Jq0M2Yt08jU534c0p--
```

Now if you take a photo and share it with Lunch Roulette, whenever any other Lunch Roulette users come within about 110 meters of that location, they will get your picture in their timeline. Glass will superimpose the "Compliments of Lunch Roulette" text over the image, as the following figure shows. We add the text to inform the user where it came from.

Figure 24—Showing an attachment with overlaid text

Wrap-Up

This chapter pulled together the remaining parts of the Mirror API. Contacts represent entities that we can share with, be they individual people, groups, or even Glassware. And the best things to share are often images, videos, and other attachments.

We now have all of the technical tools required to create any type of Mirror API Glassware. But as part of a positive user experience, your application's overall design is very important. We'll cover it next.

CHAPTER 7

Designing for Glass

Design is an evolving art in the best of times, though occasionally punctuated by sweeping changes in requirements. The printing press changed print design, allowing for a consistency unmatched by hand-copied manuscripts. Movable type further changed the game, elevating the field of typography. The advent of personal computers, and later the Web, caused designers—previously focused on magazines, posters, and the like—to re-evaluate their ideas of a fixed media, incorporating new layouts and new assets, such as video. Just as the web-design field began to mature, designers were forced to make another leap with the reduced size of mobile devices. Today, Glass is pushing the industry with a tiny, visually ever-present display and a timeline-based interface.

The difference between decent and great software is design, and Glass requires a different way of thinking about design. Our aim here is to cover some user-experience conventions, to get you into the heads of your users before you launch a Glass application. Following that we'll put on our design hats and dig into designing Glassware from a high level, down to the details of what user-interface designs you can and cannot reasonably accomplish. We'll end with some tools you can use to help craft your look and feel.

A Little UX

Design is more than making things look pretty. A good Glass *user experience* (UX) encompasses more aspects than *user interface* (UI) design. UX is about how a user ultimately perceives and feels about your software, and is not necessarily subject to hard and fast rules. But there are a few guidelines you can follow to create a better Glassware experience, such as following the principle of least astonishment,[1] presenting information that is relevant, and presenting information at the opportune time.

1. http://en.wikipedia.org/wiki/Principle_of_least_astonishment

Principle of Least Astonishment

The *principle of least astonishment* is a general design convention that promotes designing Glassware that works in a manner that makes sense. For example, when you share a card with a contact, you should share the timeline item, attachment, or some other aspect of its data. In Chapter 6, *Making Glass Social*, on page 75, we let a user share an image with other people in the same geographic location by creating a Lunch Roulette contact. Although you could technically do anything in response to a SHARE menu-item action callback, such as simply modifying the image, that would be unexpected and a little strange. Save custom behaviors for CUSTOM menu items.

Another case to keep in mind is limiting the number of timeline items that your app creates. Nothing is more maddening than shuffling through more timeline items than you can read, especially if they notify with *ding* sounds on creation. For example, rather than create a timeline item on every location callback, throttle new items to create at most two an hour. That may not seem like a lot, but it is if someone uses twenty Glassware apps.

Relevance

Adhering to the principle of least astonishment implies that results should be relevant. If someone signs up for an app that's supposed to display a joke every day, don't also include a card promoting another app. This veers dangerously close to spam, and is against the spirit of the Glass Platform Developer Policies.[2]

In short, be straightforward with your users about what your software is going to do, don't bother them too much, and always ensure that you deliver relevant results.

Right on Time

Timeline items are all about being displayed at the appropriate time—it's right there in the name. When choosing a displayTime to post you'll almost always use the default time, which is *now*, unless you have a compelling reason to choose otherwise. Running a batch job in advance of a specific time, however, is one case where posting a timeline item in advance of its time works.

There's rarely a good reason to post anything to the timeline in the past. Perhaps you can post a few minutes or possibly hours later if you're trying to keep in synch with a more real-time service. But posting a timeline item a

2. https://developers.google.com/glass/policies

day late doesn't do a user much good in practice. If you're running a service that executes in batch jobs, you should run them frequently enough to be closer to real time.

Design Layout

While UX is an important overarching concept always worthy of thinking about, what most of us think of when we hear the word *design* is *user interface* design. It's what our users see and interact with most readily. We've seen in previous chapters than HTML is the crux of timeline-item design, and Google gives you a lot to work with.

Although most HTML is valid as a Glass timeline card, to keep a consistent look and feel across Mirror apps, Google provides developer guidelines and suggested UI guidelines, which are a combination of standard HTML and CSS (cascading style sheets).[3] [4]

Column Layouts

In Chapter 5, *Tracking Movement and User Responses*, on page 59, we saw one method for creating a two-column layout: by setting a 360-pixel-high figure next to a section. The HTML render places them side-by-side due to space constraints and flow rules.

```
<article>
  <figure>
    <img src="glass://map?w=240&h=360&marker=0;${latitude},${longitude}&marker=1"
      width="240"
      height="360"/>
  </figure>
  <section>
    <h2>${ name }</h2>
    <strong>${ address }</strong>
  </section>
</article>
```

But we can be more intentional with our layouts by using two built-in CSS classes, layout-two-column and layout-three-column. You can probably guess what they do. Two-column is the more common of the two, for when you need to lay out information side-by-side. The following is a sample comparison of two nearby Mexican restaurants, with the cuisine type attached to a footer. The layout-two-column expects two child divs with contents.

```
<article>
```

3. https://developers.google.com/glass/guidelines
4. https://developers.google.com/glass/ui-guidelines

```
  <section>
    <div class="layout-two-column">
      <div>
        <h2 class="yellow">Chipotle</h2>
        <p>7003 SW Nyberg St Tualatin, OR USA</p>
      </div>
      <div>
        <h2 class="blue">El Sol</h2>
        <p>7028 SW Nyberg St Tualatin, OR USA</p>
      </div>
    </div>
  </section>
  <footer>
    <p>Mexican</p>
  </footer>
</article>
```

With a bit of color in their titles (yellow and blue classes), the two choices are clear, as you can see in the following figure. Adding a footer is a nice way to use a bit of extra space to pop in some minor info.

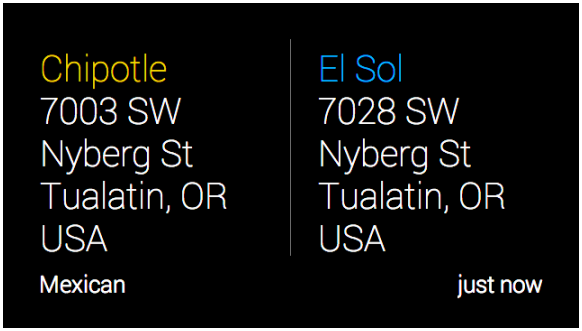


Figure 25—Two-column layout

Two-column tends to be more useful in practice than three-column. By the time you're venturing into three-column territory, you are either presenting data as a table or putting far too much information on a single card.

Full-Page Images

You may have noticed by now that all timeline items are wrapped in an article HTML tag. This is more than a semantically useful construct; it's also a way to include classes for new styles and behaviors.

One of the most useful classes is photo, which expects a child image tag. If you want to overlay any other HTML or text, you should follow the image with

a photo-overlay class on an empty div to dim the image slightly. This makes any following HTML readable.

```
<article class="photo">
  <img src="http://d1c739w2xm33i4.cloudfront.net/2.2/top_image.jpg"
    width="100%"
    height="100%">
  <div class="photo-overlay"/>
  <section>
    <p class="text-auto-size">Compliments of Lunch Roulette</p>
  </section>
</article>
```

You might notice that this looks identical to displaying text on top of an attachment. That's because the Mirror API generates something similar to the preceding. We are able to mimic the automatic resizing of plain text by using the text-auto-size class.

HTML Pagination

Each article is 640 pixels horizontal by 360 pixels vertical (640×360), with a default black background and white *Roboto* text. But sometimes that's not enough room to display all of the information you need, especially in cases where you can't or don't want to create multiple HTML pages or thread timeline items. Just like timeline-item bundles, a timeline item's HTML can flip though multiple cards. Every article is at least its own card that your users can flip through.

```
<article>
  <section>
    <h1 class="text-auto-size">List of Lunch Roulette Cuisines</h1>
  </section>
</article>
<article class="auto-paginate">
  <ul class="text-x-small">
    <li>American</li>
    <li>Burgers</li>
    <li>Chinese</li>
    <li>Dessert</li>
  </ul>
</article>
```

You can drive pagination further by using an auto-paginate class. A single list that could extend off the screen is a great use of auto pagination, although any large multicomponent object such as a long table can paginate.

```
<article class="auto-paginate">
  <ul class="text-small">
    <li>American</li>
```

```
    <li>Burgers</li>
    <li>Chinese</li>
    <li>Dessert</li>
    <li>French</li>
    <li>Greek</li>
    <li>Italian</li>
    <li>Japenese</li>
    <li>Mexican</li>
    <li>Pizza</li>
  </ul>
</article>
```

This will produce the following.

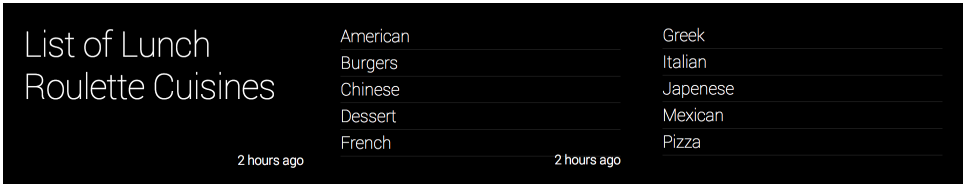


Figure 26—Showing all cards with HTML-based pagination

Look and Feel

We started with the high-level user experience, and then saw how to lay out data on particular cards and multicards. Now we'll get to the fine-grained control we need to give users a custom experience by using and overriding a particular CSS or creating custom menu items. Despite our ability to make varied and subtle designs, always keep in mind that this is a small optical display, so keep it simple.

Mirror CSS

The HTML of a timeline item follows similar rules to the HTML on a web page. But in the case of Glass it's codified implicitly by foundational styles.[5] Even if you don't ever set a style on your HTML, the cards will look pretty good.

Keep in mind, though, that the default style only works as designed by wrapping your HTML in an article element. Of course, you can write any valid HTML and style it yourself, but forgetting the article element will mean Glass isn't rendering a card for your data, and hence your information is not displayed, so be aware.

5. https://mirror-api-playground.appspot.com/assets/css/base_style.css

Colors

When choosing colors for Glass you should never forget that the display is transparent. This means you're always fighting with a background of unknown complexity or brightness. Users may be looking at a sunset or the Sistine Chapel or a cubicle wall. Your best is to always choose high-contrast colors. High contrast means simple and bold. Don't choose a subtle gray font on sage green, but instead keep to the standard white, or other basic colors, on black.

To help keep things standard, the default CSS provides six basic colors: red, yellow, green, blue, white, and gray/muted. There's rarely a good reason to stray outside of these colors for text. And there's rarely a good reason to set the background to any color other than black or an image. This doesn't mean you can't, but you should have a very good reason for choosing odd colors. Black doesn't simply make your app text more legible; it also makes it flow better among all of the other timeline items a user sees.

Style and Layout

It's fairly intuitive that the smaller a display is, the less text you can put on that display. But it's more insidious than that. The smaller a display is, the more whitespace you need to make a distinction between characters. The relationship between display size and text density can be sidestepped in some ways, such as by using high-information-density assets instead of text. If you can use visual cues like icons, or use an image, video, or map rather than a description, you can convey far more information in a small space.

If you aren't sure how you want to style or lay out your Glassware, you should take a look at the growing list of predesigned templates hosted on the Google Mirror API Playground.[6] It's easy enough to copy the HTML you want and make the changes you need.

Menus

Menu items can have a custom icon, which you should provide for a couple of reasons. One, it helps your menu item look like a regular menu item. Plain text stands out in a bad way. Second, it makes it easier for users to spot your item without having to read the text. When I use the Twitter app's Retweet menu item, I look for the small double-arrow icon, not the words.

If we add an icon to the Alternative custom menu item that we created in the last chapter, it should be a 50×50-pixel, simply shaped PNG image with a

6. http://developers.google.com/glass/playground

transparent background. The MenuValue that houses the item's state must point to an iconUrl that hosts the image.

```
MenuValue defaultMV = new MenuValue()
  .setState( "DEFAULT" )
  .setDisplayName( "Alternative" )
  .setIconUrl( "http://example.co/arrows.png" )
);
```

If we add that bit of code, our menu looks nicer. Keep in mind that simple icons are readily downloadable online, but you should always ensure you have permission to use them, (either via their license or by purchasing the usage rights). Additionally, host the icons yourself, both as a courtesy to other web hosts and to ensure they don't drop or replace the icon with something you don't want.

Wireframes and Mock-Ups

We've worked on our designs by writing raw HTML, but professional designers often prefer to work in a visual, or wireframe, environment. It's often easier and better to draw out the flow and how the cards will look before ever writing a line of code. In fact, drawing out your workflow can sometimes speed up the coding effort, since you have specific final goals to code against.

Being so new, Glass suffers from a dearth of design environments. You can, of course, use any existing graphic-design or web-design software to create wireframes and mock-ups. UXPin was the first to release a Glass design kit.[7]

UXPin (see Figure 27,*The basic UXPin page layout*, on page 95) is not an expensive suite, but you can sign up for a free trial to toy around. Let's use UXPin to mimic what we have in Lunch Roulette so we can see how simple changes might look. As for any nontrivial software, navigating the app can be hectic until you get used to its quirks. Add a new page and name it LunchRoulette. From there, choose an Elements library. From the long list of options, choose Google Glass.

What's neat about this design studio is that when you create a Glass canvas, it will put an image in the background (see Figure 28,*UXPin Glass canvas editor*, on page 95). This may feel jarring, but it helps you keep in mind that your users won't have a nice black backdrop in which to view your changes. In fact, even if you choose to design in something other than UXPin, this is a good trick to keep in mind. You can drag items around, add images, add text—anything you would expect in a WYSIWYG editor.

7. http://uxpin.com

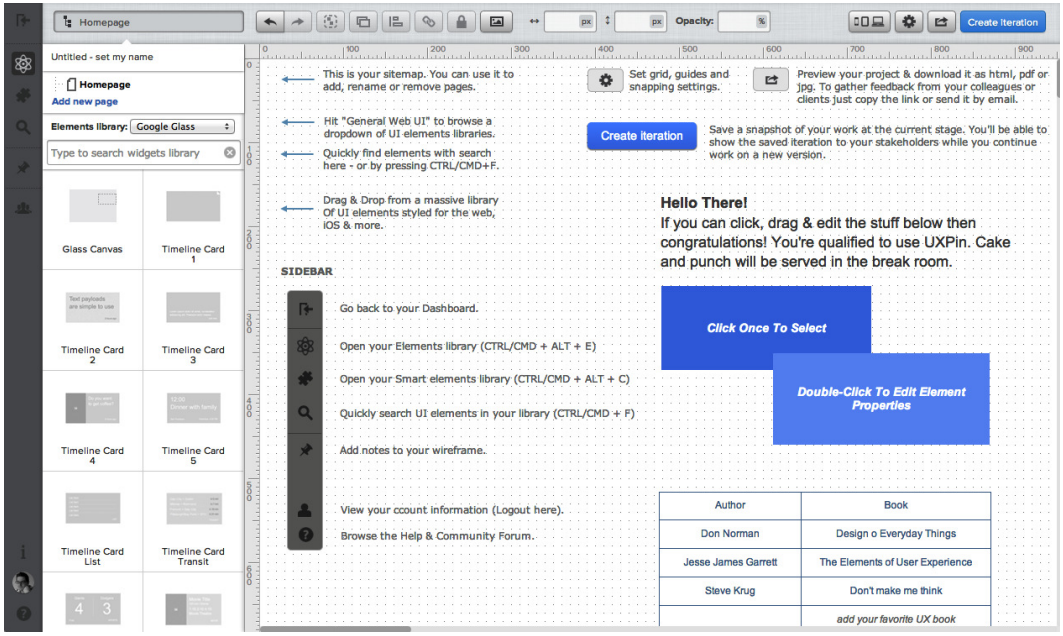


Figure 27—The basic UXPin page layout

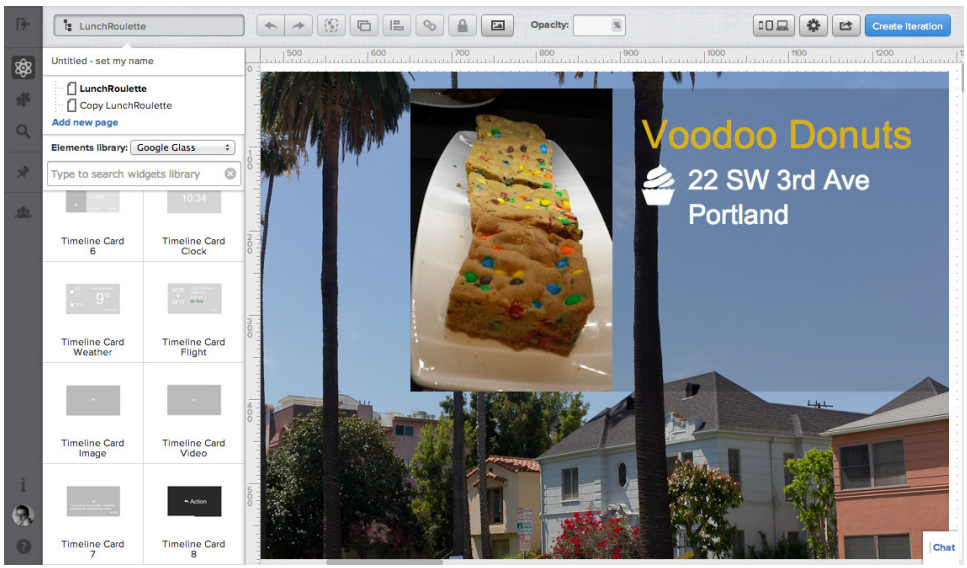


Figure 28—UXPin Glass canvas editor

If you want to design a workflow that requires multiple cards, you can add a new page. If it's a similar design or you just want to see what your card will look like with a menu on top, duplicate an existing card (see the following figure). Then it's a simple matter of dragging a menu element over the top of the card.

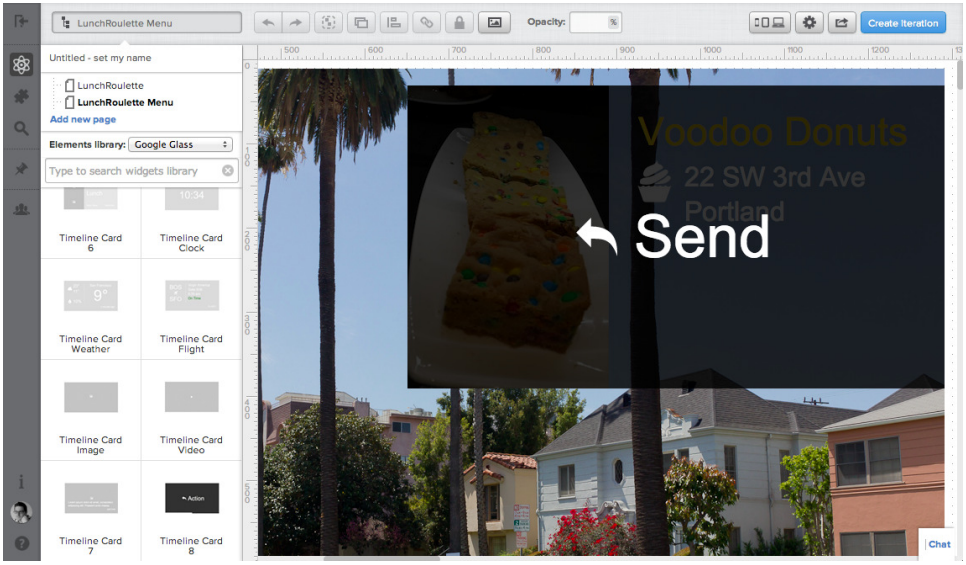


Figure 29—UXPin Glass canvas menu editor

Test on Glass

Throughout this book, we test timeline item creation by issuing a GET request to a /lunchroulette URL. This URL would never be visited in production, but it's useful for testing on Glass. Although it's possible to look at HTML in the Google Mirror API Playground or view raw JavaScript Object Notation by reading HTTP REST requests/responses, you should never launch an application to the public without testing it thoroughly on a real Glass device. Some layouts render slightly differently than you'd expect, some fonts don't work as you might hope, and colors can appear strange when viewed through a small translucent optical display.

That's pretty much all there is to it. With the selection of HTML, CSS, and images, you can make your designs as varied as you like—but it's a good idea not to go way off script. There's one last note to keep in mind when crafting mock-ups. Always outside of your control is the age of the item printed in the lower-right corner, such as *just now* or *2 days ago*. Keep some room down there, or even include a placeholder.

Wrap-Up

Glass is a very much a new device paradigm, and it could go any direction. While sometimes new technologies are an excellent prompt to reexamine fundamentals, they're also an excellent opportunity to seek out standard patterns. Glass is a different enough device with a novel enough interface without our having to toy around with abnormal fonts and color combinations.

On the other hand, we're entering uncharted waters. Although it's a good idea to stick to standard rules of thumb, you should not be shy about re-examining those topics and trying new ones. You may find that black text on a white background ends up easier to read than white on black in some cases. It may turn out that people really will like Comic Sans. No one knows, so experiment—just don't be too eager to throw out the collective knowledge we have.

CHAPTER 8

Turning a Web App to Glassware

Glass is a new kind of device, as different from a smartphone as a tablet is from a desktop computer. Although the world, including the Mirror API, runs on the Web, Glass isn't a web device. This means you'll have a bit of work cut out for you to enhance an existing web application to be Glassware, but it's worth doing to win more users.

Take heart! We can follow some general rules for expanding a web app to support Glass. In this chapter we'll upgrade a simple blog to function in Glass. We'll start with a look at the software's design and what actions a user can take. Then we'll break those actions into parts that we can and cannot implement in Glass. Finally, we'll dive deep into making the necessary changes so our blog can work as both a web app and Glassware.

ChittrChattr App

Blogging applications are common on the Web, so that's what we'll tackle for the broadest reach. We'll be converting a mini blogging app called ChittrChattr into Glassware. (In classic web-application form, we drop some vowels from the real words to make our app's name.)

Blogs are relatively easy to model, and we all understand how they work: you log in and make a post to a blog, and users can read those posts. For an extra bit of fun, ChittrChattr includes a *follow* feature. A user can choose to follow another's blog and view a listing of all posts made on all blogs she's chosen to follow.

The model of our blog software is basic, as you can see in Figure 30, *The blog model in UML*, on page 100. A user has a blog, and a blog has many posts. A user can follow many blogs, and get a listing of every post that the user follows in reverse chronological order.

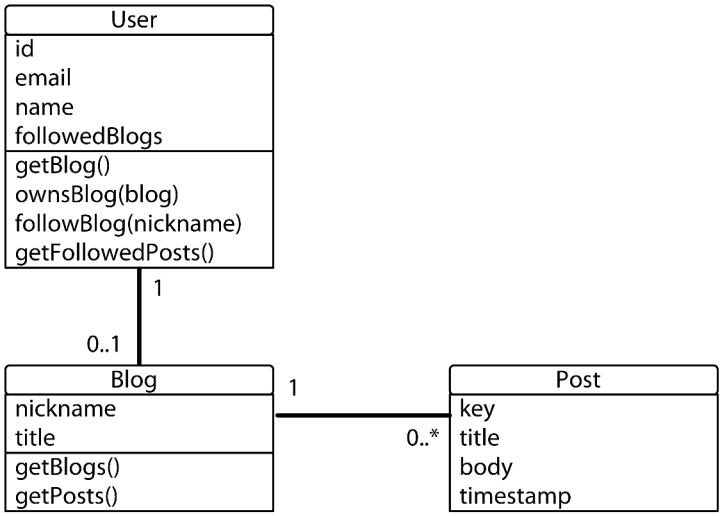


Figure 30—The blog model in UML

The rest of the code in our blog comprises servlets to support building or viewing instances of the model, and some helper code. A user can log in (OAuth2Servlet) or log out (LogoutServlet), create a blog (BlogServlet), create posts or view posts (PostServlet), and create or view followed posts (FollowedServlet).

The next figure shows what a GET request to FollowedServlet renders.

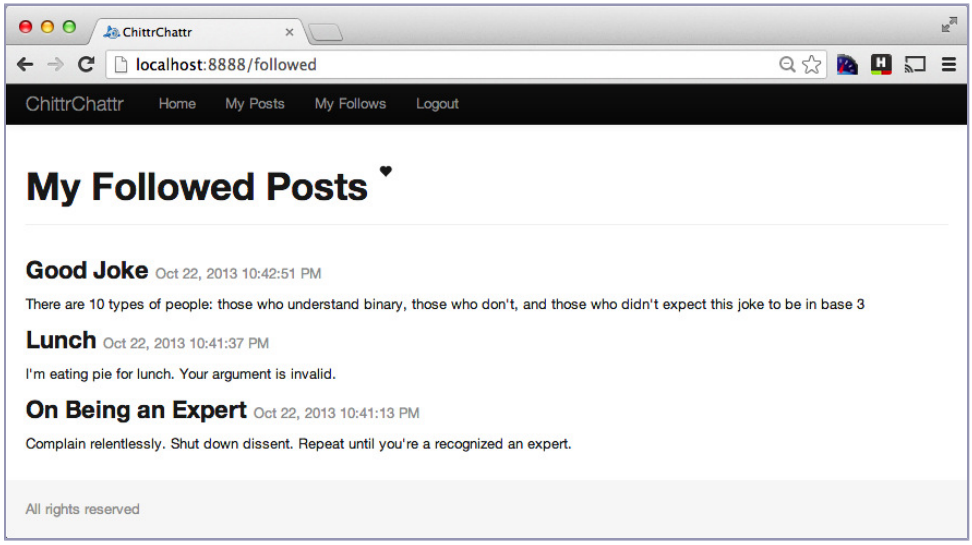


Figure 31—An example page of followed blogs

We also have a HomeServlet that lists the most recent new blogs. You can download all of this code with the book or on GitHub,[1] so I'll forgo posting any of it here. I recommend you take a look at the code so you can work through it alongside this chapter. It's not too complex.

List Actions

The first step in expanding our app from weblog to Glasslog requires figuring out which use-case actions we want to open up to Glass users. It may be possible to implement every one, but that may not be necessary or even desired.

We talked about the actions a user can take in ChittrChattr, but it's helpful to start by simply listing them out.

- Log in / Create user
- Create a blog
- Create a post
- List all blogs
- Choose a blog
- Read a blog's posts
- Follow a blog
- Read followed posts

We're looking at all of these actions from the user's point of view, and we aren't concerned with any implementation details yet. But the actions do have an order of operations, which you can map out as a sort of flow chart, shown in the figure here.

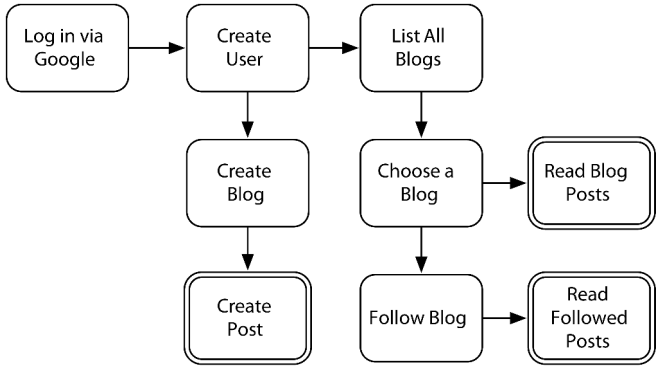


Figure 32—Blog actions flow chart

1. https://github.com/coderoshi/glass

Note that three components on this chart are double-outlined. We'll talk about why soon, but first let's prune any use-case actions we don't need to implement.

Remove Unneeded Actions

Not all actions must be ported to Glass. Some actions cannot be done on Glass at all due to technical restrictions, like lacking the access to hold a session cookie or type a password. Some actions are better done elsewhere prior to interfacing with Glass, often due to their rarity or complexity.

The major ChittrChattr action that *cannot be done* in Glass due to technical reasons is logging in, which creates a user. This is not really a surprise, since all Mirror Glassware requires logging in and authorizing through other means, such as the Web or a mobile app. More nuanced is figuring out which actions we could implement but we don't need.

First, let's filter any *rare actions*. Creating a blog is an action that is taken only once per use. Sure, we could implement this in Glassware, but it hardly seems worth it because a user can just create a blog when he logs in for the first time.

Next, we'll skip any actions that are too long or require complex steps. It can be hard to decide where to draw the line here. Examples include finding a new person to follow on Twitter, searching for a new friend on Facebook, or digging for a particular Tumblr. Anything that requires protracted navigation, such as looking through a very long list of blogs, could be skipped.

We must also consider removing use cases that cascade from actions we can't do (ignoring logging in, which is commonly a prerequisite). For example, before a user can follow a blog or read a blog's posts, he needs to list all blogs. Since that action is being skipped, we can consider skipping the successive steps unless we can come up with another path to those actions.

In our case, we won't bother implementing them, since we're left with two good solid actions that can be implemented in Glass. Recall the preceding flow chart of user actions. Notice anything? The actions we're left with are the double-outlined boxes on the chart. In effect, we end up with the final use cases that make our app handy. The rest is just navigation. And since the ability to read a blog's posts is covered by Read Followed Posts, we only need to choose a blog for our Glassware.

Next let's see how we can modify our two remaining use cases to suit Glassware.

Glassifying the Actions

Out of our initial list of web-based ChittrChattr actions, what remains are *create a post* and *read followed posts*. Since a web browser is an entirely different beast from the Glass timeline, it's not enough to simply copy a website's functionality directly. In this phase, we'll think about how to implement the use cases we require in a method that works best with Glass.

Create a Post

Since we've already decided to let our Glass users create a post, we should think about how to do that. A website would normally let you write up a post using a keyboard, but we need to investigate a different avenue here.

Glass has a robust speech-to-text interface we can use in our Glassware. But in order to use that, you'll have to add a ChittrChattr contact, in much the same way that you did for Lunch Roulette. The Mirror API provides for a contact to accept a `TAKE_A_NOTE` speech-to-text command.

This setting will add a new verbal command to the home card, which your users can activate by saying, "OK, Glass, take a note" followed by saying the contact that will accept the verbal note—in our case, the ChittrChattr app. Once you've chosen the app, you can speak what's on your mind, and Glass will convert that speech into text.

`TAKE_A_NOTE` functions are somewhat akin to custom menu items from a Glassware app's point of view. Our code must subscribe to the action of a user taking a note. Since that subscription will provide a `callbackUrl`, you also must create a servlet capable of accepting callback notifications.

Next, we need to look at our remaining action.

Read Followed Posts

Posting to a blog is great, but without an audience, you're just talking to yourself. The other half of our blog software is allowing a follower to read followed posts. While the web-based application requires a user to visit a web page to see a listing of posts, a more natural option for Glass would be to simply add new followed blog posts to the user's timeline.

This is a fairly passive activity from the user's point of view. One need only follow a blog to receive a new timeline item for each new post. Our effort here is to come up with a well-designed timeline item.

Clearly, a blog post must show the text that a user has posted. Since we're not placing a limit on how long a blog post can be, we should use the pagination feature in case it runs to multiple pages. There's no reason to change the text or background from its natural white on black; however, we can add a little custom icon to each item so that a user is aware this was created by ChittrChattr. It's a nice visual cue, and a bit of advertising to boot!

As a convenience, we should also include a few menu items. READ_ALOUD is a good choice since it lets a user listen to the blog post. Sometimes a blog poster might add a link to her blog post. If she does, a Glass user should have some way to visit that URL, so let's also add an OPEN_URI menu item. Finally, letting a user delete a post is always a nicety worth considering.

With our Read Followed Posts timeline item's design complete, we can come up with a complete list of tasks.

The Worklist

Taking everything we've considered so far, our blog Glassware code must implement five work items.

First, we must allow blog authors and/or followers to authorize with Google OAuth containing a glass.timeline scope. Of course, this is always a necessary step for any Glass app that wants to interact with a user's timeline.

Second, we must create a new ChittrChattr contact that can accept the TAKE_A_NOTE command. We can create this contact as part of the OAuth step, and we need only do it once.

Third, we have to subscribe to TAKE_A_NOTE actions. We also need to do this only one time, as a part of authorization.

Fourth, we have to create a servlet capable of receiving notifications that a new note is being added (thus creating a new blog post).

Finally, when a user creates a new blog post, everyone that follows the blog should receive a timeline item containing the new post. It shouldn't matter if the post was created through speaking to Glass or by typing into the web app—followers should always be notified of a new post.

We can tackle each of these items in turn, starting with the authorization changes.

The Mirror Code

With our list of tasks in order, now we get to implement the behavior we expect from our Glassware extension.

Login/Authorization

Adding Google Glass authorization requires that a user already have a Google account. All Glass users must have one for Glass to function, so luckily you'll never have to deal with the case where a user doesn't have an account. However, that doesn't mean your application employs any of Google's user-management systems, like OpenID or OAuth. When integrating Glassware into your existing app, you'll have to give some thought to how your own user-management system can associate its accounts with the necessary Google account.

One option is to have a user first log in to your own app, keeping her credentials in session as you normally would. Then, when she wants to activate your Glassware extension, she uses Google's OAuth mechanism. Your OAuth callback will then associate the Google user ID with the ID currently in session, and store those tokens together as a single user profile. However you choose to implement your user manager, you'll have to keep track of Google's user credentials to make use of the Mirror API.

If you aren't married to any particular login system, it's easiest to utilize Google's user profiles, which is exactly what we did with ChittrChattr. In that case, it's a simple matter of adding the `glass.timeline` scope to the `GoogleAuthorizationCodeFlow.Builder`, as we did in Chapter 3, *Authorizing Your Glassware*, on page 27.

A Contact and a Subscription

The next two items tie into the user-authorization step. Once a user has successfully authorized your Glassware, you're free to immediately start making Mirror requests. The first step is to create a contact object that will represent ChittrChattr, and the second is a create a new Subscription to inform Mirror that your app wants to be notified when a user issues a `TAKE_A_NOTE` command to ChittrChattr.

Before adding any Mirror code, you'll need the Mirror API client. Since our app is written as a Java web app in Android Developer Tools, you can add it easily enough: right-click the project, then choose Google -> Add Google APIs, search for *mirror* to find the Google Mirror API, select it, and click Finish.

We'll also be using MirrorUtils class code similar to that from previous chapters. It's just a collection of convenience functions for getting a Mirror object from any credentials or user ID.

Now let's look at the code we'll need to add. At the end of the OAuth2Servlet, before a user is redirected, add the following Contact-creation code. Note that you should change the PROD_BASE_URL value to be your own app's URL.

chapter-08/ChittrChattr/src/test/book/glass/auth/OAuth2Servlet.java

```
Contact contact = new Contact()
  .setId("chittrchattr")
  .setImageUrls( Collections.singletonList(
      PROD_BASE_URL + "/static/images/chittrchattr.png" ) )
  .setPriority( 100L )
  .setAcceptCommands(
      Collections.singletonList(new Command().setType("TAKE_A_NOTE")))
  .setDisplayName("ChittrChattr")
  .setSpeakableName("chitter chatter");

mirror.contacts().insert( contact ).execute();
```

The important thing to see here is the TAKE_A_NOTE value put into acceptCommands. This will let a user add a verbal note to the ChittrChattr contact. However, since all of these commands are verbal, Glass must know how to pronounce *ChittrChattr*. So we also add a speakableName field in natural English terms: *chitter chatter*. Without this level of specificity, Glass may have a hard time figuring out how to match what a user says with the contact's lexically awkward displayName.

Next, add the Subscription-creation code.

chapter-08/ChittrChattr/src/test/book/glass/auth/OAuth2Servlet.java

```
Subscription subscription = new Subscription()
  .setCallbackUrl( PROD_BASE_URL + "/postcallback" )
  .setVerifyToken( "a_secret_to_everybody" )
  .setUserToken( userId )
  .setCollection( "timeline" )
  .setOperation( Collections.singletonList( "UPDATE" ) );

mirror.subscriptions().insert( subscription ).execute();
```

This directs that timeline insertions trigger a POST with Notification payload to the /postcallback URL, which we'll create next.

Turning Notifications into Blog Posts

When a user says, "OK, Glass, take a note, ChittrChattr…" anything that follows will be converted into text. That text will then be sent as a payload to a servlet that we're about a create.

Before we create the servlet, we should take a minor detour. Since we want PostCallbackServlet to create a new Post in the same way that POSTing to the web app's PostServlet does, let's create a BlogHelper class that both servlets can call to perform the same createPost action. We'll dig into the details of this class later. Just know for now that it accepts at minimum a post body and a user, and creates a new blog post under that user's blog.

Our PostCallbackServlet is a standard HttpServlet that will accept a POST (don't forget to add the servlet to the web.xml file). Just as we've done in previous chapters, we create a notification object from Mirror's posted JavaScript Object Notation message body.

With our Notification in hand, we can start extracting values and check to ensure that this particular call is meant to be a blog post. First, we check that the call is a LAUNCH user action, which means the user launched the verbal command. If not, we return from the method.

chapter-08/ChittrChattr/src/test/book/glass/blog/mirror/PostCallbackServlet.java

```
// Get this user action's type
UserAction launchAction = null;
for (UserAction userAction : notification.getUserActions()) {
  if( "LAUNCH".equals( userAction.getType() ) ) {
    launchAction = userAction;
    break;
  }
}
// return, because this is the wrong kind of user action
if( launchAction == null ) return;
```

Next we want to validate that our ChittrChattr is an intended recipient of this verbal message. We do that because there's no reason why another application couldn't be the target of a TAKE_A_NOTE action. So we compare all contact recipient IDs to verify that our app's contact ID (chittrchattr) is in the list.

chapter-08/ChittrChattr/src/test/book/glass/blog/mirror/PostCallbackServlet.java

```
String userId = notification.getUserToken();
String itemId = notification.getItemId();

Mirror mirror = MirrorUtils.getMirror( userId );
TimelineItem timelineItem = mirror.timeline().get( itemId ).execute();

// ensure that ChittrChattr is one of this post's intended recipients
```

```
boolean belongsHere = false;
for( Contact contact : timelineItem.getRecipients() ) {
  if( "chittrchattr".equals( contact.getId() ) ) {
    belongsHere = true;
    break;
  }
}
// return, because this post wasn't intended for ChittrChattr
if( !belongsHere ) return;
```

Once we've made it through the verification gauntlet, it's a simple matter of retrieving the given text and posting it to the user's blog.

chapter-08/ChittrChattr/src/test/book/glass/blog/mirror/PostCallbackServlet.java

```
String body = timelineItem.getText();
User user = User.get( userId );
BlogHelper.createPost( null, body, user );
```

Notifying Followers of New Posts

To wrap this app up, we're looking to create a timeline item for every follower of any blog that has a new post. This is an inversion of how the web app operates. In the web application, a user loads the /followed web page, which queries for every post on all blogs that the current user follows. In other words, the local web-page user pulls the posts. However, in Glass that behavior is flipped on its head. Whenever a blog has a new post, that post should be pushed to every user that follows the blog.

Let's circle back around and look at some details of BlogHelper.

First, the createPost method we used in PostCallbackServlet (and the web app code, PostServlet) creates a new post object and stores it with the creating user.

chapter-08/ChittrChattr/src/test/book/glass/blog/BlogHelper.java

```
public static final Post createPost( String title, String body,
                                      User user, Blog blog )
  throws ServletException, IOException
{
  if( user == null || !user.ownsBlog( blog )) {
    throw new ServletException(
        "You must be logged in and own this blog to create a post" );
  }

  Post post = new Post( blog.getNickname(), title, body, new Date() );
  post = post.save();

  pushPost( blog, post );

  return post;
}
```

The last action creating a post takes is to push that post to every following user.

chapter-08/ChittrChattr/src/test/book/glass/blog/BlogHelper.java

```
private static void pushPost( Blog blog, Post post )
  throws IOException
{
  // find every user that follows that nickname
  List<User> users = blog.getFollowers();
  for (User user : users) {
    String userId = user.getId();
    Mirror mirror = MirrorUtils.getMirror( userId );
    TimelineItem timelineItem = new TimelineItem()
      .setText( post.getBody() );
    mirror.timeline().insert( timelineItem ).execute();
  }
}
```

For every user that follows, create a new timeline item with the blog post's body.

Wrap-Up

That's it! Converting an existing web application into Google Glassware is only a matter of figuring out which use cases you can implement, and which of those you want to implement. Then you take that list of use cases and build a game plan for them to work with the Mirror API.

This chapter wraps up Part One—but wait, there's more! Part Two of this book covers the GDK, the other major library for interacting with Google Glass. Where the Mirror API is how your web applications can interact with the Glass timeline, the GDK is about running code natively on the Glass device itself, accessing every cool component and sensor baked into Glass.

Part II

Glass Development Kit

CHAPTER 9

Introducing the GDK

When Glass was released for the first batch of beta users (Glass Explorers) in April of 2012, the only development option available was the Mirror API web service. This sentiment echoed the initial iPhone release, and Apple's insistence that the only viable iPhone apps were web apps. In both cases, a few months later, native development kits were released. As Mark Twain once noted, history doesn't repeat itself, but it does rhyme.

If Google Glass provided only a timeline of static cards populated by a web service, it would be a useful product in its own right. As we've seen in Part One of this book, you can do quite a bit with the Mirror API, from geolocation to sharing images or video with others, to interfacing with other web APIs like Twitter or Facebook. But Glass can be so much more, from translating text-to-speech in real time, to new forms of video games—that's what Part Two is about.

In Part One, we chose to write our Mirror API applications in Java out of convenience, since Mirror is a web service you could code in any language you wish. But Java knowledge is necessary for programming the GDK, since it's the *lingua franca* of Android, and hence, the Google Glass device.

This starts with learning when to use the Glass Development Kit (GDK) versus the Mirror API. From there we set up a development environment that can run and deploy GDK applications. We'll explore how Glass has many of the capabilities of any Android smartphone, as long as you know how to write code with the GDK. So let's bid *adieu* to the Mirror API, and open up to the GDK.

Choosing the GDK by Use-Case

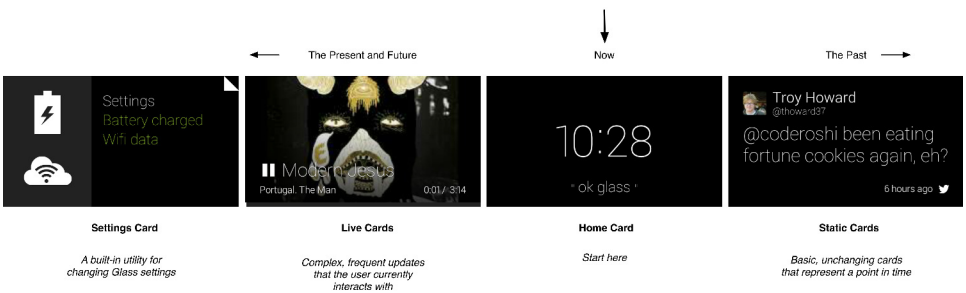
Before we dig too deeply into the details of the GDK, it's worth touching on what the GDK is, and how it differs from the Mirror API.

The most obvious difference is that GDK applications are installed on the Glass device itself, much like apps you might install from the Google Play store. Since Mirror API apps can only update the timeline when an Internet connection is established, the GDK is necessary if you need to create applications that don't require wireless access. Most of the time, however, the real differentiators will be other use-cases, such as video rendering, using Glass sensors, or fine-grained gesture recognition.

Google Glass provides three different styles of applications, which Google calls *UI elements*: *static cards*, *live cards*, and *immersions*. Let's review each one in kind.

Static Cards

We've seen plenty of what static cards can do in Part One. What they universally contain is information that is relevant for a particular point in time on the timeline. Just like a history book, things that occurred in the past are unchanging (unless you rewrite the book), aka, static. So you use the Mirror API for any application that deals in point-in-time events, like an email, a photograph, or an appointment. Static cards are placed to the right of the home card and live cards are placed to the left, as you can see in the following figure.



Static cards are strictly the domain of the Mirror API. You cannot create static cards with the GDK.

Live Cards

Live cards are akin to static cards, except they represent *now* rather than a point in the past. They run as applications until they are intentionally closed. Just like when viewing a static card, you can always swipe through the timeline to other cards, even multiple running live cards. Unlike static cards, live cards can change in real time.

A good example of a live card is a music-playing application that will continue playing even when the live card isn't currently on the screen, but when you wish to view the current song, jump to the next song, or change the volume, you interact with the live card. When you're finished with your music, you close the application through the live card's menu. A menu is always required in a live card, even if the only option is to exit the application.

Immersions

Immersions are Glass applications that are independent of the timeline. They are more akin to the Android applications you'd run on a smartphone, where the application takes over the current screen. Unlike live cards you do not continue to navigate the timeline while an immersion runs, but rather, the immersion overtakes the timeline experience. An obvious example of an immersion is a video game, where you play the game in the foreground, and close it when you're through. In this book we're going to create a few immersions, one being a QR Code reading app, as shown in the following figure.

Figure 33—An Immersion Lives Outside and On Top of the Timeline

In short: static cards are for the past, live cards are for now, and immersions exist outside the timeline entirely.

Setup a GDK Dev Environment

Before we can start coding our first GDK applications, we must first set up an environment. Like coding with the Mirror API, we'll use an Intellij-based IDE. If you don't already have the Android Studio (AS) IDE installed, you can download it from the Android SDK developer site,[1] and follow the site's installation instructions.

From the same URL where you downloaded and installed Android Studio, you can also install the Android SDK command-line tools. We'll be using both the IDE and command-line throughout the rest of this book. Once you've installed the command-line SDK, set the install directory as `ANDROID_HOME` and add the `platform-tools` and `tools` subdirectories to your `PATH`, for example:

```
$ export ANDROID_HOME="/Users/ericredmond/android/sdk"
$ export PATH=$ANDROID_HOME/platform-tools:$PATH
$ export PATH=$ANDROID_HOME/tools:$PATH
```

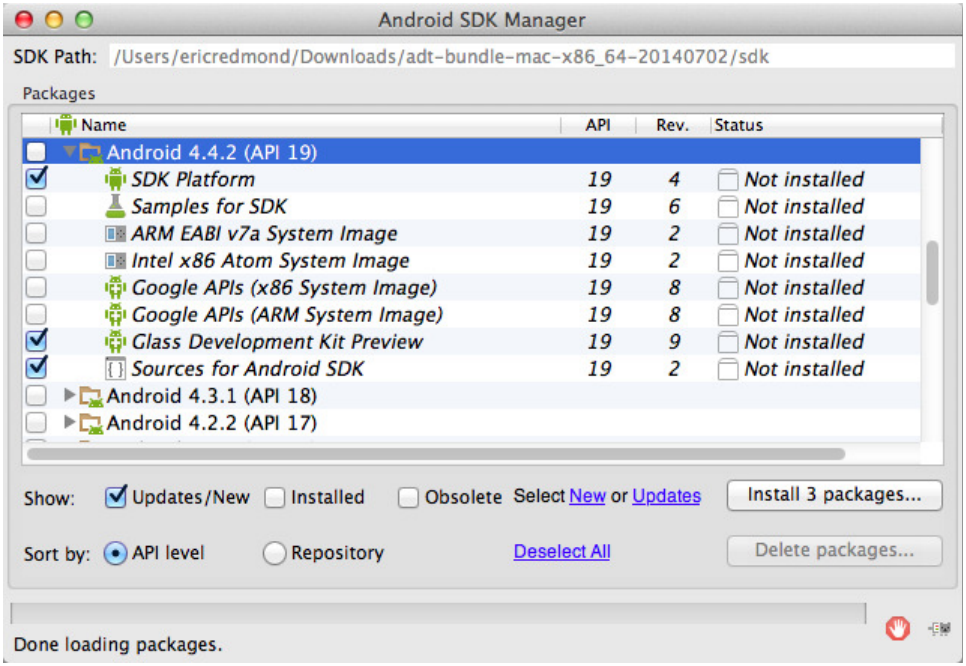
I added the above lines to my `~/.bashrc` file. Add them to whatever startup files you need for your shell.

Installing GDK

Unlike the GAE development tools which were Intellij-based, Android Studio (AS) is built on the Intellij IDE. AS is merely a set of plugins that add some new features to the standard Intellij Java development environment. After you start AS for the first time, click on *Configure -> SDK Manager*.

Google Glass runs a version of Android with API version 19, so scroll down to *Android 4.4 (API 19)*, and select *SDK Platform*, *Glass Development Kit Preview*, and *Sources for Android SDK*. Then click *Install 3 packages...*, as shown in the following figure, and accept the licenses.

1. http://developer.android.com/sdk/index.html



Wait a few minutes, and you'll be asked to restart AS. When it reloads, you'll be halfway to writing your first GDK-based Glassware. But first, you must set Glass to debug mode, which we'll walk through in the next section.

Reading Source Code

Books and documentation are great, but when learning a new library, I find it useful to look at actual code from time to time. Luckily for all of us, the Android codebase is open source. Since AS is based on the Intellij IDE, it comes with many general purpose Java programming tools. One of those tools is the ability to link binary JAR (Java Archive) files with actual source code. I highly recommend doing this with Android.

However, the GDK is still not open source, which leads me to a second recommendation for reading source code: the JD decompiler. JD converts binary bytecode into somewhat readable Java code. It has an Intellij plugin that works with Android Studio[a]. There will be times where you need to dig deeper into how the SDK or GDK works. Decompiling Java code can fill in the gaps of any libraries where you do not have the source code available. It's especially useful for keeping an eye out for upcoming features, which tend to get released in the toolkit (but deactivated) before wide release.

a. https://plugins.jetbrains.com/plugin/7100

Sideloading Glassware

Once you've developed your first Glassware app with the GDK, you'll want to test it on your Glass device. Or maybe you want to install in-development Glassware created by someone else. In either case, you'll need to open your Glass device to accept external packages from the USB port in a process called *sideloading*.

Sideloading is what it sounds like, you load your own software onto Glass by plugging the USB port from your Glass to your laptop, circumventing the standard My Glass store.[2] In order to do that, you must allow data to transfer by wire to your Glass by setting it to debug mode.

If you're not already wearing it, put on your Glass. Swipe left, all the way to the *Settings* screen. Tap it, and scroll to *Device info*. Then tap that, scrolling to and selecting *Turn on debug*. You should see something like the screens below.

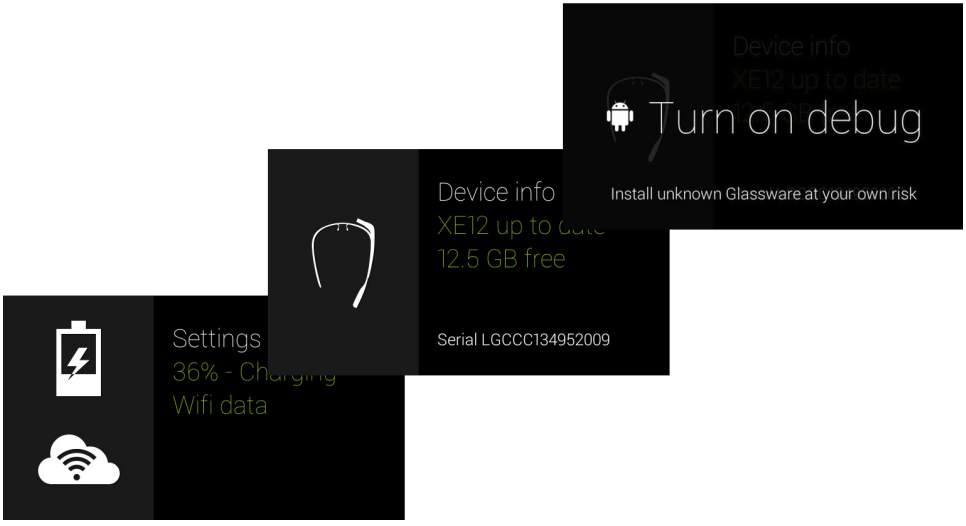


Figure 34—Putting Google Glass in Debug Mode

If it's not already connected, plug your Glass into your laptop's USB port so we can install an app. You can test that your Glass is ready to use in development by running `adb devices` on your command-line.

```
$ adb devices -l
List of devices attached
```

2. http://google.com/myglass

```
015DB75A0B01800C   device usb:1D110000 product:glass_1 model:Glass_1 device:glass-1
```

If your glass devices is not listed, some people have had to kill their adb service (adb kill-server) and try again. If you're running Windows, you may have to update your USB driver.

A Sideloading Test

Before we wrapup this section, let's take a moment and see what it's like to sideload an app from the command line. This is the method you'll use to install third party GDK Glassware. It will be a good test that your Glass device is prepared to accept custom software.

Since our example won't be touching any source code, but rather installing a third party application binary file, we won't use AS (which focuses primarily on building projects from source). Instead, we'll use the command-line application called ADB (Android Debug Bridge).

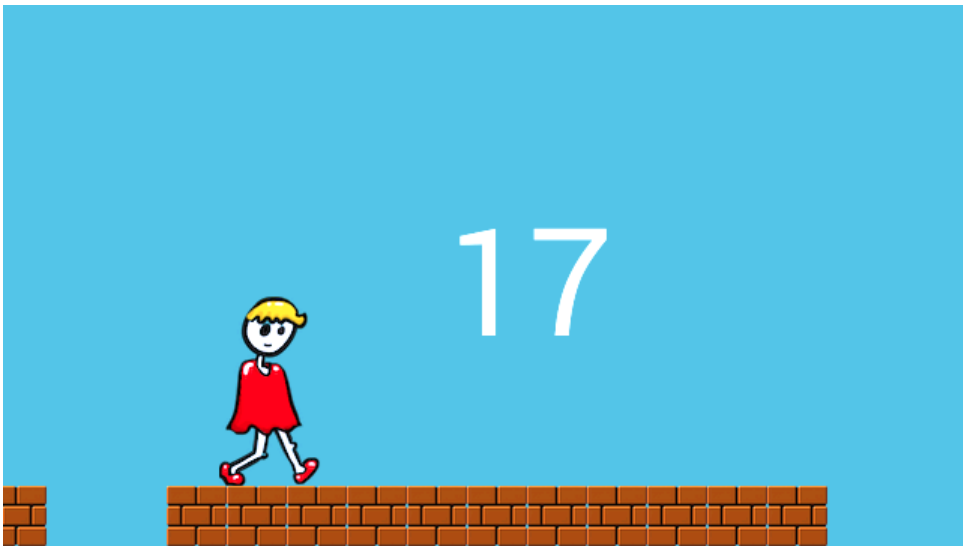
First, download the APK (application package) file named MissPitts.apk available with the book source code, or the book's git repository.[3] An APK is a binary package bundle for an Android app. Since GDK-based Glassware are Android apps, they too are packaged as APKs.

Next, execute the install command followed by the path to the MissPitts.apk file to sideload the app.

```
$ adb install MissPitts.apk
```

Now wear your Glass and say "ok glass, play a game" (followed by "Miss Pitts" if you have other games installed). A simple little side-scrolling video game should launch, looking something like this.

3. https://github.com/coderoshi/glass/tree/master/chapter-09



You'll be learning how to write this game in Chapter 14, *Creating an Immersive Game*, on page 219. Play! Have fun, then remove your Glass and let it rest a spell. We have some important business to discuss.

We've mentioned Android a bit, but it's time we learned a little more about Android. It's the underpinning of every GDK-based Glass app. The Glassware you write will be better for the knowledge.

An Android Primer

Google Glass is, from a hardware power point-of-view, a stripped down mobile device. It has a low-pixel view screen, lo-fi speaker, microphone, low-res camera, very simple touch interface, inertial sensors, and two buttons (black camera button up top, and power button inside the main body). But it does run the Android operating system, the most popular smartphone and tablet OS.

The benefits to the Glass community of Google's choice of Android are obvious: it has years of research and practice behind a rock-solid operating system, an army of developers already familiar with the basic tenets of its programming environment, a myriad of existing tools for design, development and debugging, and a huge library of existing applications waiting to be tweaked to work on Glass. We owe it to ourselves to take a peek behind the curtain of toolkits and IDEs, at the magic that makes the hardware come to life.

Be warned, if this is your first encounter with Android it can feel a little dense. But this knowledge will come in handy when you need to profile your applications for performance, debug, or potentially even customize the OS with new drivers or native libraries. We'll go into more detail about profiling and debugging in Chapter 15, *Preparing For the Real World*, on page 245.

A Little Bit About Android

Android is an OS designed specifically to run on small mobile devices, with a standard framework and library written in Java atop a modified Linux kernel. It leverages much of the power of open source Linux, such as proven stability, memory management, and security models. Many of the modifications to the kernel make sense for mobile devices, such as tight memory requirements, or that they only support a single user. Each Android application can run as its own Linux user, ensuring apps run strictly in isolation from one another. Android provides other additions as well, such as several native drivers and libraries, from power and audio services, to SSL and WebKit libs.

Dalvik

The Linux kernel is the lowest level of the Android operating system. From that point of view, the Android runtime is a higher level architecture. Probably the coolest aspect of Android as an OS is the fact that it runs each application with its own special Java virtual machine (VM) implementation named Dalvik.

Dalvik is a VM optimized for constrained memory and processor environments. After you write Android code in Java, it will compile that code into bytecode (`.class` files)—but it doesn't stop there. It will then convert the bytecode into a Dalvik executable (`.dex` files). This is a good thing to know if you ever crack open an APK, as you'll see plenty of `dex` files rather than so many Java `.class` files.

So to recap, the bottom layers of the Android OS are Linux with some drivers, services, and libraries. This layer is responsible for standard kernel work, like spawning processes, memory management, process permissions and security, hardware drivers, and other such low-level work. On top of that is the Android runtime, where each Android application (defined in an APK file) is spawned with its own DalvikVM. But what about the applications themselves?

Applications

Android applications are, basically, spawned as single threaded Linux processes. This process launches a DalvikVM, whose Java/Dalvik code is man-

aged and has access to a layer of common Java elements called the Application Framework, plus some core Java libraries. This layer is also where the GDK lives.

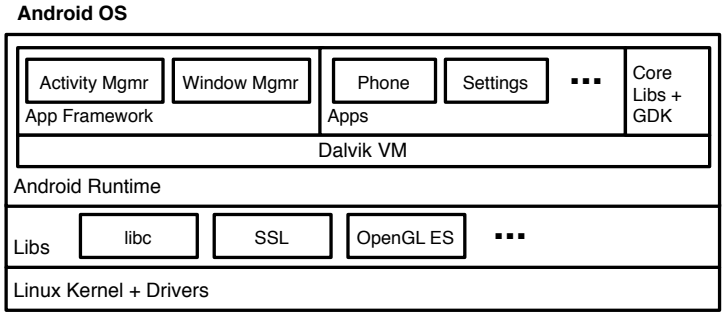


Figure 35—Android OS Layers

You may ask yourself, why go through all of this trouble to make Java the application manager/interface for an operating system? Why not just run native Linux apps on a mobile device? That's because Java, running on a virtual machine, allows Android applications to run on any hardware that supports the Android OS. The compiled Dalvik Executable (dex) and shared Java libraries are the same for every Android device, whether a smartphone, watch, tablet or Google Glass. Compile once, run anywhere.

So does this imply that you can run an unmodified smartphone Android application on Google Glass? In simple cases, yes. But you won't want to, since apps designed for a touch-screen smartphone won't work too well on Glass, which lacks a touchable screen. In Chapter 16, *Turning an Android App to Glassware*, on page 265 we'll take a look at how to convert an existing Android application into Glassware.

One final note about Android: the operating system used by Google Glass is Android proper, not the newer alternative OS for wearables called Android Wear.[4] That OS is currently used for some smart watches, but not Glass.

Wrap-Up

In this chapter, we covered the basic differences between the Mirror API in Part One, and the GDK in Part Two. We took a little peek at the two major UI elements that require the GDK: LiveCards and Immersions. We learned when you'd want to use the Mirror API, or when you'd need the GDK to render a

4. http://developer.android.com/wear/

LiveCard or an Immersion. We set up your dev environment to use AS, as well as sideload APK files using ADB. Finally, we took a brief tour of Android, the operating system of Google Glass.

We have the necessary runway to takeoff in the next chapter, where we'll cover the very basics of Android SDK application development—a required precursor to getting the most out of Glass and the relatively minor GDK extensions. If you want to know more about the large and complex ecosystem of Android proper, there are several books measured in the thousands of pages that dive down far deeper than this one.

With a general understanding of the Android stack, let's move on to build some very simple Android apps of our own.

CHAPTER 10

An Android Introduction on Glass

Any farmer will tell you, you can't grow a good crop if you don't understand the soil. The best seeds, tools, and tender care won't be much help if you plant soybeans in acidic ground. What lies beneath matters. In the case of our Glassware, we need to understand some components of Android, the operating system that allows your GDK application to flourish or decay. Since the GDK is merely the Android SDK with some extra bells and whistles for Glass, this chapter will raise some new concepts for all developers, whether you consider yourself an Android expert or not.

Before you can truly appreciate the new Glass Development Kit (GDK) extensions to Android, it helps to understand some basics of Android proper. If you are new to Android development, don't worry. The simplicity of Glass actually makes learning many parts of the Android SDK unnecessary, from complex UI tools like Fragments, to touch gesture processing. You can create rich Glass apps with surprisingly little overall Android knowledge if you leverage the enhancements that the GDK provides.

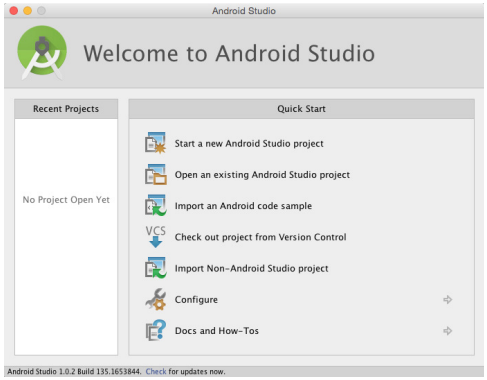
We'll start by building a simple Android project with the extended GDK library. You'll convert the application to a Glass Card that prints "Hello World" when it's launched. Finally, you'll expand the app to detect whether the Glass device is currently plugged into a USB port or not by hooking into some system events.

Generating a Simple Android App

A GDK-based Glassware project can be created in several ways. You can build the files from scratch, copy an archetype from the Internet, or use the Android Studio (AS) wizard. But our goal for this chapter is to see just how close Glassware is to any old Android project. So to start, we're going to generate a basic smartphone style Android application using the AS project wizard,

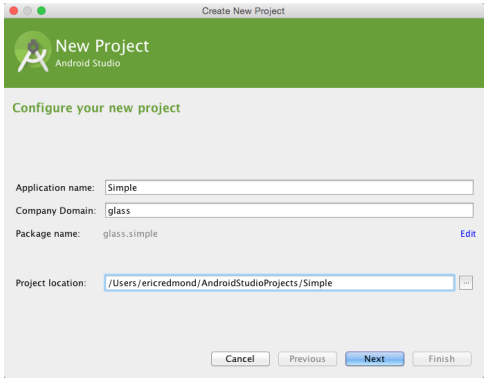
with no Glass-specific code. We'll poke and prod at this little example, and run it as a *Hello World* app on our Glass device.

When you first launch Android Studio, you'll be greeted by a project generator window. If you don't see this window, go to the Android Studio menu bar and select *File -> New Project*, which skips this first window and jumps right to the next.



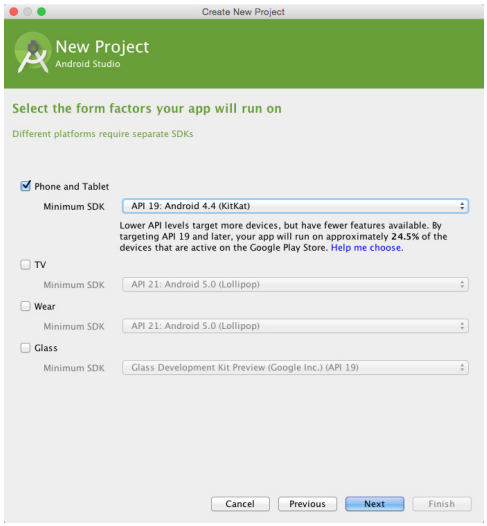
Next configure your new project. We named our application *Simple* with the Company Domain *glass*. The company domain will be the prefix of all generated Java packages, which by default are domain.application, so in our case, *glass.simple*.

You can let it generate the Project location, or change it to the directory in which you want the project to live, and continue on to the next screen.

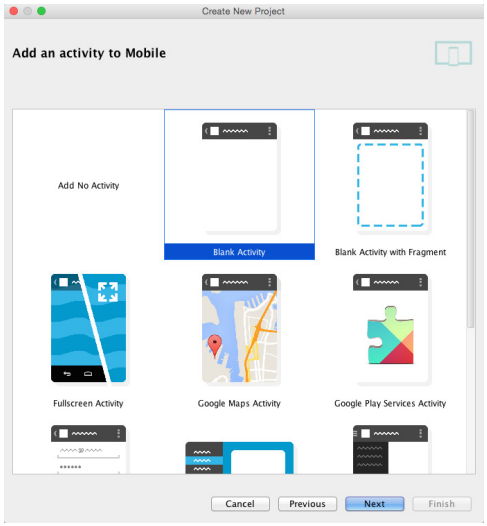


For this example, check Phone and Tablet (do not check Glass). We want to generate a standard Android project at this time. In the future we'll jump

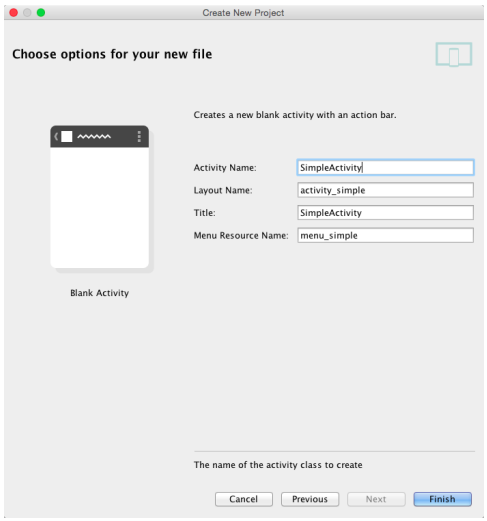
straight to Glass projects. Choose Android API 19 as the minimum SDK and click next.



Later in the chapter we'll cover what an Activity is, but for now, just choose *Blank Activity*. This will generate code that outputs "Hello World" to an Android screen and little else.



Finally, the project creator wants you to name the blank activity. Let's rename the activity from *MainActivity* to *SimpleActivity*. The other fields will automatically change to match.



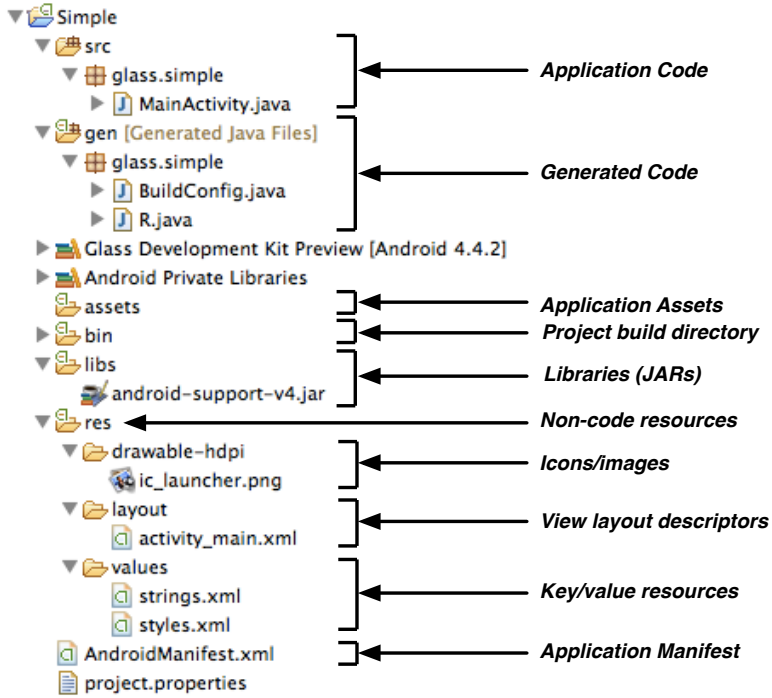
Several Gradle commands will run before launching your application. Gradle is the project building tool of choice for Android projects. You may be familiar with alternative build tools, like Make, Ant, or Maven. This is yet another one, but created by Google.

You have now generated a basic *Hello World* Android project that will launch a single screen when run on any Android platform, including Glass.

Double-click on the SimpleActivity.java file under app/java/glass.simple.

The Project directory structure that AS generated for you is similar for all Android apps, no matter how simple or complex those apps may be. The most interesting parts are the src directory where Java source code lives, the res directory where XML and icon resources are, and the AndroidManifest.xml configuration *manifest file*. There is also a gen directory that contains Java classes that are automatically generated based on the contents of the res XML and manifest files, namely, BuildConfig and R, which we'll use later.

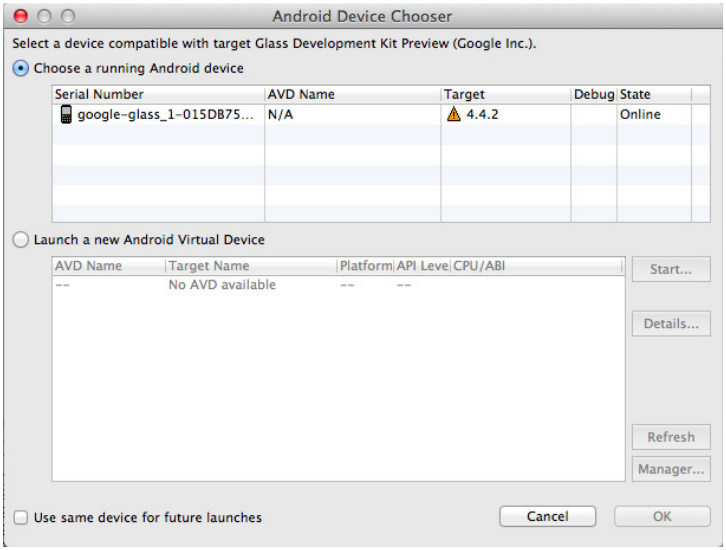
We'll cover other parts of this structure in more detail in later chapters. For now, you should see something similar to the following layout directory.



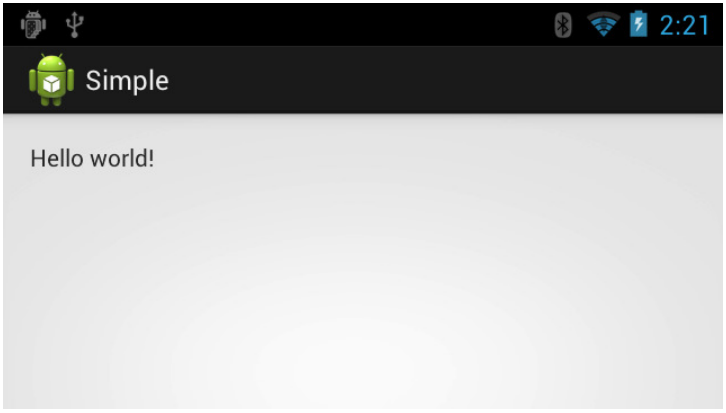
Note that your values and drawable directories may contain suffixes, which relate to specific API versions and Android screen resolutions (AKA, density buckets), for example, drawable-xhdpi or values-v14. However, Glass currently only supports one resolution and version. If you prefer removing unnecessary directories, you can remove the values suffixes entirely, and leave drawable-hdpi as the only supported drawable directory.

The code that will launch and run our *Hello World* application is also generated by the AS as sample code, named SimpleActivity.java. We'll tinker with that file in a moment. For now, let's just run what we have.

Plug in your Glass device, put it on, and keep the home card lit. If the Glass screen isn't active, the app won't run, since the trigger to launch will be missed by a sleeping Android. In AS click the green right arrow in the toolbar, or press ctrl-R. This will launch the *Android Device Chooser*, shown here.



Pick your device, and your app will run on your Glass screen. You should see a simple *Hello World* screen, similar to this:



The first thing you may notice is how the app doesn't look like a Google Glass card. It looks more like a mobile app. It has status and action bars, and Android icons for power, wifi, and a clock. It displays a text view that says *Hello World* (you can see where this layout is declared in the res/layout/activity_simple.xml file). But since you don't have a touchscreen or keyboard, you can't really interact with the text view, which is normally editable (try running this app on an Android phone if you want to see how it's intended to work).

Congrats! It may not seem like much, but you've just written, packaged, sideloaded, and run your first GDK Glassware. Let's look at some parts of the Android SDK, and get a handle on what our code just did.

Basic Android SDK+GDK

Android has a lot of moving parts. It's a full operating system that also provides a well-defined API for interacting with common smartphone elements. But its general-purpose nature explains why it's gaining popularity for non-smartphone uses, both bigger (like tablets and televisions) and smaller (like smart watches and Glass).

Android deals with this range of hardware elements and form factors with a few API design choices. One part of the design uses the AndroidManifest.xml file to describe to the Android runtime how an Application is structured, the resources it uses, and some details about how it's intended to run.

Another API design is that applications interact with Android components within an app, or externally between apps via messages called Intents. Intents are messages that either communicate to a UI element called an Activity, or a set of non-visible tasks called a Service, or they deliver a global message accepted by a BroadcastReceiver. Other application resources or supporting code are laid out in a predefined directory structure to help simplify development and packaging.

Don't worry if the preceding paragraphs felt dense with a lot of new terms, we'll spend the rest of this chapter unweaving each thread.

Once you understand these basic building blocks, you're a long way toward developing your own Android apps on Glass. Let's start with the most basic requirement for any Android app, the *Manifest*.

Application/Manifest/Resources

An App Manifest explains how an application hooks into the Android runtime. It's an XML file named AndroidManifest.xml, which sits at the root of the APK package. There are plenty of XML elements[1] to learn, but we'll cover them as needed throughout the book.

The Simple project we created with AS generated the following AndroidManifest.xml. All projects follow a similar form, but with varying degrees of complexity.

simple-files/AndroidManifest.xml

```
Line 1  <manifest
```

1. http://developer.android.com/guide/topics/manifest/manifest-intro.html

```
    xmlns:android="http://schemas.android.com/apk/res/android"
    package="glass.simple"
    android:versionCode="1"
    android:versionName="1.0">

    <uses-sdk
        android:minSdkVersion="19"
        android:targetSdkVersion="19" />

    <application
        android:allowBackup="true"
        android:icon="@drawable/ic_launcher"
        android:label="@string/app_name"
        android:theme="@style/AppTheme">
        <activity
            android:name="glass.simple.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>
</manifest>
```

Every value we must set is under the manifest element, as shown on line 1, where every field is under the http://schemas.android.com/apk/res/android XML namespace, generally named android. If you aren't familiar with XML details, this just means most fields are prefixed with android:. The versionCode and versionName fields values are up to you. You'll use them to increment later versions of a project every time you release your app as a new APK.

The minSdkVersion and targetSdkVersion fields of the uses-sdk on line 9 reflect the API version 19 choice we made in the *Android Application Project* wizard when we created out project. This is currently the only supported version of the API for the GDK.

Notice that several of the field values start with an @ sign—these are links to different resources under the res directory in your project. In some cases, @ prefixes a file name, such as the way @drawable/ic_launcher maps to res/drawable/ic_launcher.png. In the case of @style/AppTheme on line 15, it's mapped to a them (look & feel) name defined in res/values/styles.xml, which we'll cover later in this chapter. The final application field is @string/app_name, where @string is the name of the XML resource file under res/values/string.xml, and app_name is a string name defined as *Simple* in the file.

Tracing further down the file, the real interesting values are under application. These XML elements define Java classes which act as entry points into the

application. These classes come in several flavors called *components*. For now, just note that the activity on line 18 points at the SimpleActivity class that was generated as part of our *Hello World* app.

The intent-filter element on line 18 describes how the *Hello World* activity will be launched. When this app receives the given event—defined as action and category—Android will launch this app. The *event* in question is called an Intent.

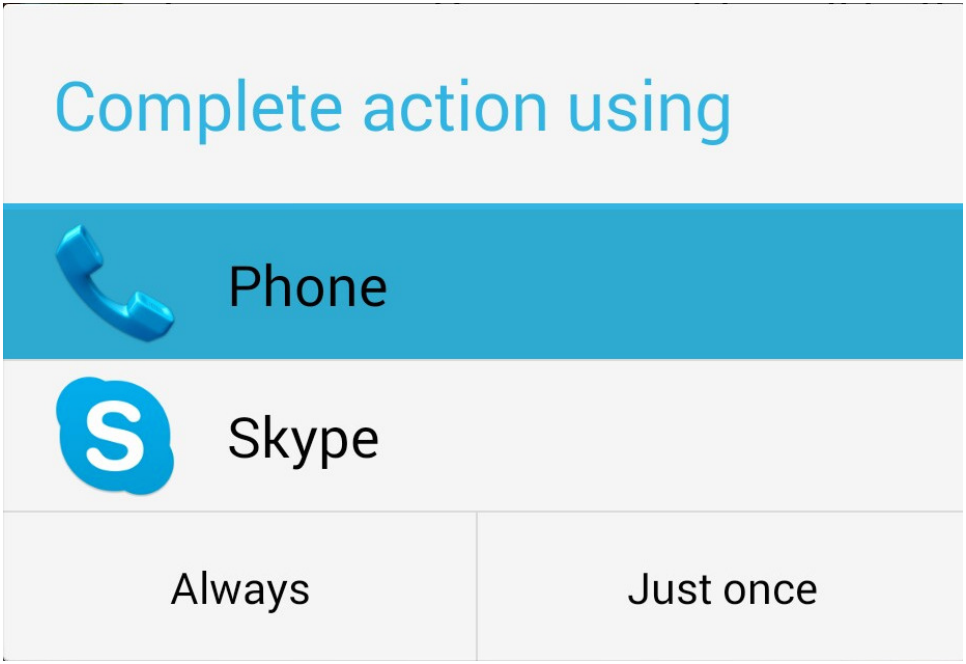
Intents

Intents[2] are signals from operations performed by a user, this application or another application, which trigger Android apps to react in some way. Intents are message objects composed of an *action* and a set of optional attributes, like *data* URI. You can think of Intents as parcels, with Android playing the role of the postal service that delivers them.

Every Android application is initially started in response to an Intent. You can think of Intents like this: when your friend mails you a birthday present, it's her *intent* to delight you. Whether you actually feel delighted depends on the kind of person you are, and the contents of the gift. You may decide to send a thank you card, or just post *kthxbai* on Facebook. In a similar way, when an application receives an Intent, it's up to that application to decide how to respond, if at all.

For a more technical example, consider an Android smartphone app that shows phone numbers as links. When a link is clicked, the app will fire an Intent with the *action* android.intent.action.DIAL, and the *data* as a phone number URI, such as tel:0019005551212. This message is broadcast to any application registered to receive the Intent to dial the number. By default this would be the built-in phone app, but other registered apps are given the chance to dial as well, like Skype or Google Voice. If there's a conflict between which applications should launch in response to a broadcast Intent, the user can choose the app they want from a list provided by Android via the *app chooser* as seen here.

2. http://developer.android.com/reference/android/content/Intent.html



Intents are Android's fundamental method for handling control flow within, and between, applications. There are four components[3] that app writers can choose to execute in response to an Intent operation: Activities,[4] Services,[5] Broadcast Receivers,[6] and Content Providers.[7]

Every Android application starts from an Intent. In a tablet or smartphone, this Intent is usually fired when a user clicks on an app icon. In Glass there are no icons, so apps are started verbally, or from an app menu. We see how to do this in Chapter 11, *Live Cards*, on page 147.

Back in our generated manifest you can see our app has a SimpleActivity. It's configured to run when the android.intent.action.MAIN action and android.intent.category.LAUNCHER category Intent are fired.

chapter-10/Simple/app/src/main/AndroidManifest.xml

```
<activity
    android:name=".SimpleActivity" >
    <intent-filter>
```

3. http://developer.android.com/guide/components/fundamentals.html#Components
4. http://developer.android.com/reference/android/app/Activity.html
5. http://developer.android.com/reference/android/app/Service.html
6. http://developer.android.com/reference/android/content/BroadcastReceiver.html
7. http://developer.android.com/reference/android/content/ContentProvider.html

```
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

When the app is packaged and sideloaded by Android Studio, the main/launcher Intent is automatically called. The app runs on the Glass screen because our app was configured to launch SimpleActivity. But we haven't yet discussed what an Activity is, so let's take a peek.

The Activity Component

An Activity is one of the four Android components that we're covering. It's also the most visible one. Other components cannot be seen by the user—they run in the background. When an application receives an Intent to launch, unless it's designed to act as a background process, it will usually be an Activity.

An Activity represents the screen that a user sees. In an Android smartphone, this could be an email application screen. In Glass, this could be an interactive Card. When users think of an application, it's generally the activity that they imagine. An application can have many Activities, which are tacitly related to each other. In Glass, one activity could be the main app screen, while another activity could be an app settings screen.

An Activity runs on a main Java thread, and follows a well-defined lifecycle (three lifecycles, actually, but we'll just deal with the *entire lifetime* for now). It's useful to have an idea of the states in this table, since they correspond to Activity methods that you implement.

Lifecycle State	This Activity is…
onCreate	being created, where you perform any setup
onRestart	being stopped, and then started again
onStart	becoming visible
onResume	now visible, or has been resumed
onPause	being paused, another Activity is taking precedence
onStop	now not visible
onDestroy	being shut down

When an Activity is first launched, Android calls the onCreate method. That method is ultimately responsible for setting the root View, which is what will be displayed to the user, via setContentView.

Referring back to the code in the generated SimpleActivity.java activity, notice that onCreate is implemented, and setContentView contains an integer defined at R.layout.activity_simple. If this method wasn't set, you'd only ever see a blank screen.

We've talked about setting a view, without discussing what a view is. Views are another piece of the visual interface puzzle.

Views

A View is what it sounds like—it's an object that displays something a user can see. Views are units of user interface, and are arranged as a tree with a single root. A View can be be a single interactive item, like a button, a text box, or a progress bar. Some Views, called ViewGroups, instead contain and manage multiple views, such as a full screen layout, or in Glass a Card view.

There are two ways to construct Views in Android: layouts and code.

Let's start with layouts. In our generated project you can find a file res/layout/activity_simple.xml. It's a layout file that contains XML that tells the Android runtime exactly how it wants to build a View hierarchy. In our simple case, the layout constructs a RelativeLayout—a type of ViewGroup—containing a TextView that prints out the contents of the the res/values/string.xml resource, which in this case is *Hello World*.

chapter-10/Simple/app/src/main/res/layout/activity_simple.xml

```
<RelativeLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:paddingBottom="@dimen/activity_vertical_margin"
    android:paddingLeft="@dimen/activity_horizontal_margin"
    android:paddingRight="@dimen/activity_horizontal_margin"
    android:paddingTop="@dimen/activity_vertical_margin"
    tools:context=".SimpleService" >

    <TextView
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="@string/hello_world" />
</RelativeLayout>
```

This layout is used by AS to generate a part of the class named R. This is an AS generated class you'll find under your project's gen/glass/simple/R.java file. You should see many classes and variables corresponding to resource files and values. These variables can be used in code to bind the values defined in your XML files, with compilable code, represented as classes and integer ids.

For example, the activity_simple layout becomes the integer R.layout.activity_simple. This is why our generated SimpleActivity's onCreate() function called setContentView(R.layout.activity_simple). The AS generated the code to use this layout for us.

The second way of generating a View is more obvious. You just write code. Let's create a card the easiest way, by using the GDK Card object. We populate the card text and footer, and then convert that Card into a View via getView(). The GDK will generate a new View object with a pre-defined layout.

chapter-10/Simple/app/src/main/java/glass/simple/SimpleActivity.java

```
public class SimpleActivity extends Activity {
    private View cardView;
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        CardBuilder card = new CardBuilder(this, CardBuilder.Layout.TEXT)
                .setText("Hello World")
                .setFootnote("This is from an Activity");
        cardView = card.getView();
        setContentView(cardView);
    }
    protected void onDestroy() {
        super.onDestroy();
        cardView = null;
    }
}
```

We can also remove onCreateOptionsMenu, since we won't use the menu bar that it creates (defined in res/menu/simple.xml). We'll cover creating menus in Glass in Chapter 11, *Live Cards*, on page 147.

Ensure that your home card is active again, and rerun the Simple project. This time it will look a bit different.

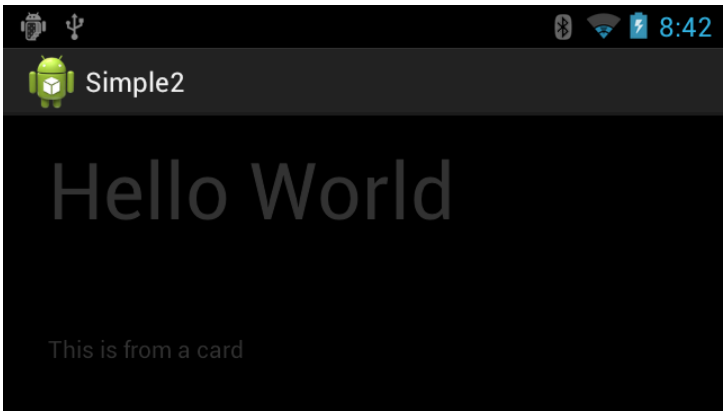


Figure 36—A Glass Card with a Weird Theme

It looks more like a Glass Card, but we're not quite done. There are still smartphone-style status and action bars overlaying the Card. To remove those we need to alter the application *theme* to be more Glass friendly.

Theme

A *style* or *theme* in Android is a resource that represents the look and feel of Views. The list of built-in style[8] choices is huge, and can be found in Android's source code.[9] These style and fields are often self-explanatory. It may take you some time to learn all of the specific nomenclature, but covering Android style fields in detail is also far outside the scope of this book.

The theme for an application or activity is set in the app manifest. Let's clean up the generated style with our own.

chapter-10/Simple/app/src/main/AndroidManifest.xml

```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/AppTheme" >
```

Like our other resources, this file lives under res. It's a set of values within res/values/styles.xml. Note that the AppTheme inherits from a parent AppBaseTheme. Since we chose no theme in the new Android app building wizard, the style defaulted to Android's basic android:Theme.Light.

8. http://developer.android.com/guide/topics/ui/themes.html
9. https://android.googlesource.com/platform/frameworks/base/+/refs/heads/master/core/res/res/values/styles.xml

We can make our styles line up with Google Glass by setting the AppTheme to inherit from @android:style/Theme.DeviceDefault.

simple-files/styles.xml
```
<resources>
    <style name="AppTheme" parent="@android:style/Theme.DeviceDefault">
    </style>
</resources>
```

Other projects we create in the future will start with this same basic style.

Now it's time to run our Simple app one more time and you'll see the following.

Figure 37—A Correctly Themed Glass Card

Success! This looks much more like a Glassware Card that we, and our users, should expect. If something is amiss, you can download and import the Simple source code and try again.

Service, Broadcast Receiver, and Content Provider

Although Activities and Views make up the visual aspects of an Android app, there are still three other components that we have yet to cover. Those are *Services*, *Broadcast Receivers*, and *Content Providers*

Services

Services[10] are components that run invisibly in the background. They can run for short or long periods of time, and can even remain running when the application is not currently active, unlike Activities. Like Activities components,

10. http://developer.android.com/guide/components/services.html

Services can be triggered through Intents. Let's use this behavior to make our *Hello World* Card behave a bit more like Glassware should.

We'll start by creating a new class called glass.simple.SimpleService. This class extends IntentService, which is a subclass of Service that makes it easier to create a service that merely reacts once to an event. This time, rather than our application only displaying a Card View while the Activity is on screen, this app will run for a brief period of time and display a short message to the user via an Android message tool called Toast.

chapter-10/Simple/app/src/main/java/glass/simple/SimpleService.java

```
public class SimpleService extends Service {
  @Override
  public int onStartCommand(Intent intent, int flags, int startId) {
    Toast.makeText(this, "Hello from a Service", Toast.LENGTH_LONG).show();
    return START_NOT_STICKY;
  }
  @Override
  public IBinder onBind(Intent intent) {
    return null;
  }
}
```

This also requires us to add a new element to our AndroidManifest.xml under application named service.

chapter-10/Simple/app/src/main/AndroidManifest.xml

```
<service
    android:name=".SimpleService"
    android:label="@string/app_name"
    android:clearTaskOnLaunch="true"
    android:launchMode="singleInstance"
    android:enabled="true" >
</service>
```

Unlike the activity element, this one doesn't contain an intent-filter. That's because we're going to launch it from the SimpleActivity. At the end of the onCreate method, add the following line.

```
startService(new Intent(this, SimpleService.class));
```

This is another way to fire an Intent, called an *explicit* Intent. It's explicit, since our code added the class it wants to launch as a service. This is opposed to the way we launch our SimpleActivity through the manifest with an action name and category, called in *implicit* Intent.

Now run the Simple app again, and you'll see something new. In addition to seeing the *Hello World* card, you should also see a message pop up when the SimpleService starts: "Hello from a Service".

Broadcast Receiver

A BroadcastReceiver class receives a broadcast Intent and asynchronously executes the onReceive method. You may be wondering if this is the same as a Service. In one sense, yes, because it runs invisibly. But in another sense it's not like a Service, since BroadcastReceivers block the main project thread as they run, and should only run for a very short window of time.

After the Intent is received, the BroadcastReceiver object is no longer valid. It exists only to be an Intent listener. This makes it a pretty good choice for creating simple events in response to global Intents, like short messages, or logs. You can even spawn Services from a BroadcastReceiver for longer running tasks.

Let's implement a BroadcastReceiver that displays a short message to the user on the screen whenever Glass's USB power cable is connected or disconnected.

When onReceive is called, we need only compare the action of the Intent with one of our two expected options, and tailor our message for the change in state. The two convenience constants that contain the Intent action strings we care about are Intent.ACTION_POWER_CONNECTED and Intent.ACTION_POWER_DISCONNECTED.

A big class inheritance difference between a BroadcastReceiver and an Activity or Service, is that a receiver does not implement Context. Context is the current state of the application and Android environment, and it is necessary to interface with the larger Android system.

If you recall in our SimpleService example in the previous section, we created a popup message with Toast.makeText(this,...), where this was the Service instance. We could use this, because all services implement Context (a requirement for makeText). But since receivers don't implement Context, we instead use the Context object used to call onReceive().

chapter-10/Simple/app/src/main/java/glass/simple/SimpleReceiver.java

```
public class SimpleReceiver extends BroadcastReceiver {
  @Override
  public void onReceive(Context context, Intent intent) {
    String message = "I shouldn't exist";
    if (Intent.ACTION_POWER_CONNECTED.equals(intent.getAction())) {
      message = "Yum! Power tastes good.";
    } else if (Intent.ACTION_POWER_DISCONNECTED.equals(intent.getAction())) {
      message = "Being unplugged makes me hangry.";
    }
    Toast.makeText(context, message, Toast.LENGTH_LONG).show();
  }
}
```

Since we're receiving both Intents, we want to output a different message depending on whether power is being connected or disconnected. The Intent message object's action can be compared against the action string, which is conveniently defined as a constant on the Intent class. You can read the android.content.Intent for a huge list of built-in Intents constants.

Then we simply insert the Card into the timeline and our Receiver action is complete.

Now we need to set the receiver to fire whenever the power state changes. Like the activity and service elements, a BroadcastReceiver component needs a receiver element in the app manifest. This time we'll add two intent-filter actions, one for each power state. Don't forget to comment out the service element.

chapter-10/Simple/app/src/main/AndroidManifest.xml

```
<receiver
    android:name=".SimpleReceiver" >
    <intent-filter>
        <action
            android:name="android.intent.action.ACTION_POWER_CONNECTED" />
        <action
            android:name="android.intent.action.ACTION_POWER_DISCONNECTED"/>
    </intent-filter>
</receiver>
```

Now, we run the project. Similar to our Service component, launching the app will not actually perform any visible actions. Instead, our app will patiently wait for its power cable to be removed or replaced before running. You can test your new app by unplugging, then replugging in your USB cable. Keep an eye on the message that pops up live on your home card.

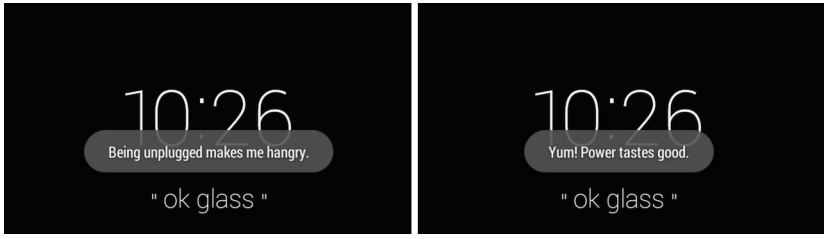


Figure 38—Toast Messages when the USB Cable is Unplugged then Plugged

We've covered three components that can be launched by Intents. There exists a forth component type called *Content Providers* that exists to store and access data between processes.

Content Providers

A content provider[11] acts as a CRUD (*Create*, *Read*, *Update*, *Delete*) interface to a data repository presented as tables, such as a database or even a web service. If you're familiar with classic n-tiered architecture, the content provider would be akin to the data access tier. The ContentProvider component is the simplest to understand, the most complex to implement, and likely the component you'll use the least.

The complexity of implementing a full and properly working ContentProvider is far beyond the scope of this book, but we do have an example implementation in code for Chapter 16, *Turning an Android App to Glassware*, on page 265. A majority of the time you'll consume, rather than create your own, ContentProviders.

A commonly used content provider is Android's Calendar provider. This provider acts as a system-wide component where all Android applications can share a user's calendar data. We're going to query your Glass's calendar for any upcoming events.

Since providers, well, *provide* access to some underlying datastore, your app must inform Android that it uses a special application permission. This is acquired under your app manifest's manifest sub-element named uses-permission.

chapter-10/Simple/app/src/main/AndroidManifest.xml

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

This is a very commonly called element in GDK apps.

While we're on the topic of requesting permission, you should note that there's a special permission for informing Glass that you're in development mode. You'll see this DEVELOPMENT permission quite a bit in our Glass applications. It's how we'll unlock custom voice commands and other catchall permissions.

```
<uses-permission android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

Let's continue with the Calendar provider. Although content providers open up a CRUD interface, users won't call the content provider methods directly. Instead, once a content provider is registered with Android, user call the provider they want to use through a universal interface called the ContentResolver.

The content resolver is responsible for executing the correct underlying provider based on a registered content URI authority. The calendar URI can

11. http://developer.android.com/guide/topics/providers/content-provider-basics.html

be found as a constant Calendars.CONTENT_URI, which resolves as the URI content://com.android.calendar/calendars. The root path com.android.calendar is called the content provider authority, and is how Android knows which provider you want to execute an operation on.

In our case, we want to call the read operation query, passing in the URI, along with some other arguments. If you're at all familiar with relational algebra, you may know the terms projection and selection. Projections are the list of fields you want a query to return, while a selection is the fields that you want to match against. In SQL, this query might look like *SELECT projection FROM uri WHERE selection*. The selectionArgs are the actual values that you populate the selection fields with.

So in our case, we want to match an calendar events' owner account with our own email address. The query will return a cursor, which we can iterate over all matching fields we requested in our projection: the event ID and the display name of the calendar.

chapter-10/Simple/app/src/main/java/glass/simple/SimpleActivity.java

```
public static final String[] EVENT_PROJECTION = new String[] {
    CalendarContract.Calendars._ID,                     // 0
    CalendarContract.Calendars.CALENDAR_DISPLAY_NAME  // 1
};
private static final int PROJECTION_ID_INDEX = 0;
private static final int PROJECTION_DISPLAY_NAME_INDEX = 1;
protected void onResume() {
    super.onResume();
    ContentResolver cr = getContentResolver();
    // The URI is "content://com.android.calendar/calendars"
    Uri uri = CalendarContract.Calendars.CONTENT_URI;
    String selection = "(" + CalendarContract.Calendars.OWNER_ACCOUNT + " = ?)";
    String[] selectionArgs = new String[] {"eric.redmond@gmail.com"};
    // Submit the query and get a Cursor object back.
    Cursor cur = cr.query(uri, EVENT_PROJECTION, selection, selectionArgs, null);
    // Use the cursor to step through the returned records
    while( cur.moveToNext() ) {
        // Get the field values
        long calID = cur.getLong( PROJECTION_ID_INDEX );
        String displayName = cur.getString( PROJECTION_DISPLAY_NAME_INDEX );
        Log.i("glass.simple", "calID: " + calID + ", displayName: " + displayName);
    }
}
```

This query is called whenever the SimpleActivity is resumed (which calls onResume), or generally, every time this app screen is made visible.

Wrap-Up

Glass runs Android, plus a few enhancements we call the GDK. There are many excellent books that cover Android[12] in great detail—this is not one of them. What this book does cover is the minimum amount of Android's SDK necessary to write Glass applications, and the GDK that helps expand and simplify development on Glass.

The only GDK specific item we've seen so far has been the Card class. Later chapters will cover more.

In this chapter we learned that Intent messages are Android's way to handle flow between components. Those components can be a visual Activity, a background Service, or a short-lived BroadcastReceiver.

An Activity requires a root View, which we can generate from a GDK Card object by calling its getView() method. The look and feel of this view can be defined by a style resource file.

Unlike the Mirror API, we cannot populate the Glass timeline with static Cards. As we learned in Chapter 9, *Introducing the GDK*, on page 113, the GDK exists to create *Live Cards* and *Immersions*. In the next chapter we're going to learn how to use a Service and a BroadcastReceiver to create a Live Card. We'll also throw an Activity into the mix to create a custom menu for our card. These are basic Android features, but as we're about to learn, they're necessary components for creating Glassware with the GDK.

12. http://developer.android.com/guide/index.html

CHAPTER 11

Live Cards

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.

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.

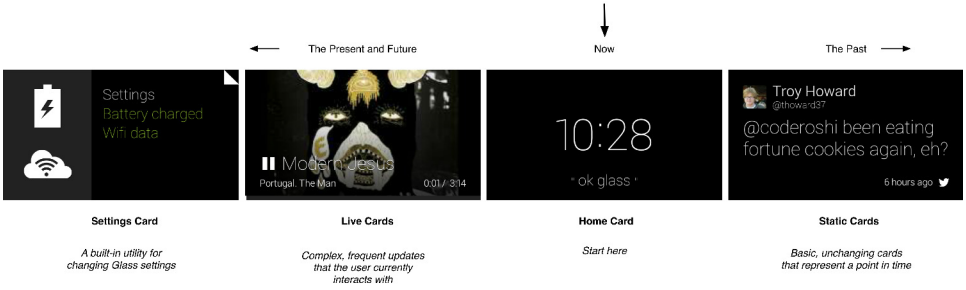


Figure 39—The Live Card is Left of the Home Card

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 167 using a screen callback object.

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.

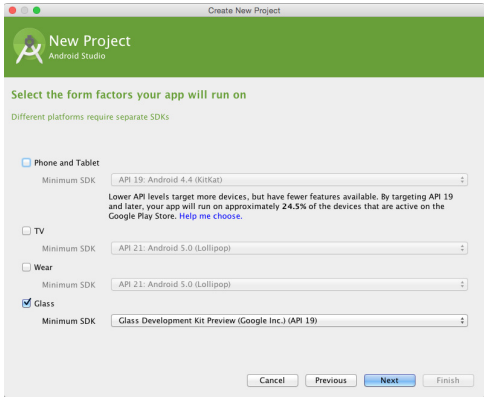
This is important because the Glass timeline is its own Android process, distinct from our application. So we create a RemoteViews object in our Service, then populate it with an XML-defined layout and whatever view values we'll want to see. These could be text view values, or colors, or images, or a 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.

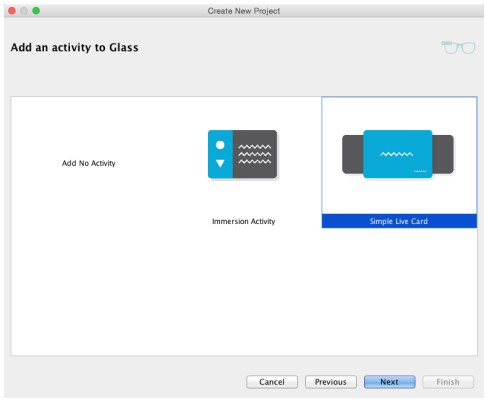
Generating a Glass Project

We generated a general Android project in the previous chapter. This time, we're going to generate a project named *Stats* as a Glass project using the GDK. Fire up Android Studio like before, but this time choose the *Glass* form factor as you see below.

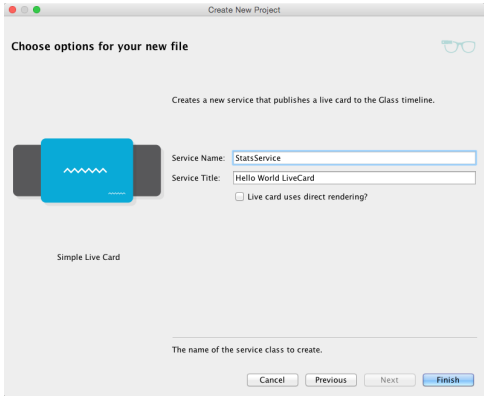
1. http://developer.android.com/reference/android/widget/RemoteViews.html



Just like an Android smartphone project, the builder will ask what kind of Activity you require. For this project, choose *Simple Live Card*.



Finally, name the class that will be generated StatsService and finish.



You can use this new project to build your own Stats application with what you learn from this chapter, or just copy the code from the public repository.[2]

Collecting Android Stats

The project we're going to investigate in this chapter creates a LiveCard that displays Glass device statistics. 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.

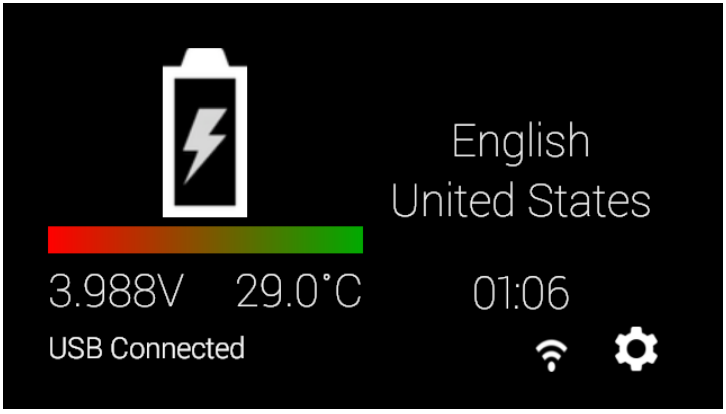


Figure 40—Our Live Card Stats Screen

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);
```

2. https://github.com/coderoshi/glass

```
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/app/src/main/java/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.[3] 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.

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 125, 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 167, but let's pretend like it doesn't exist, for now.

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

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/app/src/main/java/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/app/src/main/java/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 148 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/app/src/main/java/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 151.

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.

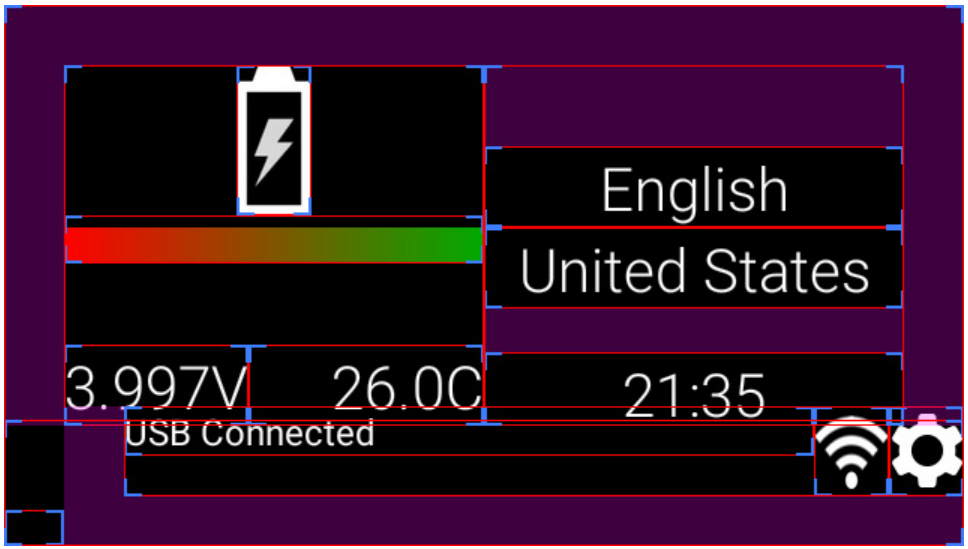
The following XML is a truncated version of the real stats layout XML, which can easily be pages long when you start including fields like android:layout_marginLeft.

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, Android Studio 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.

A Card Menu

Before we proceed, let's think a bit about what we have built. We've created a StatsService that will run in the background. When the service is created and started, it will create a LiveCard that will display a layout view that we defined, via RemoteViews. This view is part of the timeline process, so even if we leave the live card, it will still be there when we return.

But this raises a question: how can we stop the service from running? Your application must respond to a user action in some way. Providing a menu when your user taps on the LiveCard would be the least surprising choice. Then the menu should contain, at minimum, a choice to stop the Service.

LiveCard helps protect Glass programmers from accidentally creating LiveCards which can't die (I like to call these *Zombie Cards*) by requiring an action. A LiveCard *will not launch* if you do not set a pending intent in liveCard.setAction().

Launching the Menu

Returning to our StatsService.onStartCommand()method on page 153, you may have noticed that we skipped a method called buildAction() that populated LiveCard's setAction(). Let's first look at this method that returns a PendingIntent object, then discuss exactly what it's doing.

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

```
private PendingIntent buildAction() {
    Intent menuIntent = new Intent(this, LiveCardMenuActivity.class);
    menuIntent.addFlags(
```

```
            Intent.FLAG_ACTIVITY_NEW_TASK | Intent.FLAG_ACTIVITY_CLEAR_TASK);
    return PendingIntent.getActivity(this, 0, menuIntent, 0);
}
```

We're setting up an Intent that will fire when the LiveCard is tapped, thus launching a LiveCardMenuActivity. As you can probably discern from the name, the LiveCardMenuActivity is an Activity that displays a menu to the user from a LiveCard. The flags are required to let this Intent know it's ok for this activity to take over the screen.

Let's think about how this would operate from the user's point of view. The LiveCard is running, and the user taps on the side of their Glass's gesture bar. The LiveCard uses that gesture to fire this explicit intent, which creates a LiveCardMenuActivity and lays it over the card's view. Then, the user can either choose a menu option by tapping, or close the menu activity by swiping down, which will return control back to the LiveCard's views.

It's worth mentioning that PendingIntents define an Intent that will be performed later, potentially by another process. This is not unlike RemoteViews which define how a view will be constructed later by another process. This makes sense, since LiveCards act as a bridge to the timeline process.

Menu Activity

The LiveCardMenuActivity class that the LiveCard will run when tapped is not built in to the GDK. It's an Android Activity like any other, but we'll be using it in a slightly different way. Rather than setting a content view (via setContentView) to render a view when the Activity is created, we're instead going to immediately launch an options menu.

Every Android Activity has the capability of defining and launching its own menu. In a smartphone or table application, you'd be used to seeing this as a drop-down, or point-and-click set of menu items. In Glass, we expect it to be a side-scrollable list of options, similar to the look and feel of Mirror API generated menus.

The four steps to turning an activity into a menu are straightforward.

First, you have to inform the Activity that you want to create a menu by setting up a Menu object inside the onCreateOptionsMenu method, and returning true.

chapter-11/Stats/app/src/main/java/glass/stats/LiveCardMenuActivity.java

```
@Override
public boolean onCreateOptionsMenu(Menu menu) {
    getMenuInflater().inflate(R.menu.stats, menu);
    return true;
}
```

In Android, there are often multiple ways to perform some action, and menus are no different. We'll choose to the simplest. Here we populate the Menu object with items defined in an XML file res/menu/stats.xml, with the help of an Android object known as the MenuInflater.

chapter-11/Stats/app/src/main/res/menu/stats.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/action_stop"
        android:title="@string/action_stop"
        android:icon="@drawable/ic_stop" />
</menu>
```

Each item needs its own unique ID and description. The id is generated by the XML attribute android:id as R.id.action_stop. The description we use is set in the res/values/string.xml resources, which we labeled "Stop", and point to an icon bundled with our project.

Second, you have to open the menu item up. Since this LiveCardMenuActivity exists solely to provide a menu, we call openOptionsMenu as soon as this Activity is attached to the Android window (meaning, it's visible).

chapter-11/Stats/app/src/main/java/glass/stats/LiveCardMenuActivity.java

```
@Override
public void onAttachedToWindow() {
    super.onAttachedToWindow();
    // Open the options menu right away.
    openOptionsMenu();
}
```

Third, you have to respond in some way when a specific menu item is chosen. This is easy in our case, since we only have one item. When selected, the chosen item is passed by Android to the onOptionsItemSelected method. From there, you perform some action depending on the item. Since we've only added a stop item, when it's chosen we call stopService, which takes an Intent wrapping the Service class to stop, in our case we want to kill StatsService.

chapter-11/Stats/app/src/main/java/glass/stats/LiveCardMenuActivity.java

```
@Override
public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.action_stop:
            // Stop the service which will unpublish the live card.
            stopService(new Intent(this, StatsService.class));
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

Fourth, you have to turn off the Activity when the menu is closed. This is essentially the converse of step one. Since this Activity exists to display a menu, when the menu is gone, we finish the Activity component.

chapter-11/Stats/app/src/main/java/glass/stats/LiveCardMenuActivity.java

```
@Override
public void onOptionsMenuClosed(Menu menu) {
    super.onOptionsMenuClosed(menu);
    // Nothing else to do, finish the Activity.
    finish();
}
```

With that, the code for our LiveCard is in place. But we need to go about installing our app so it can be launched like built-in Glassware, such as by saying "ok glass, get directions to...".

Launching an App with Voice

Google Glass is undeniably voice driven. Standard Android apps have a desktop screen filled with icons, where you launch an application by clicking on its icon. This metaphor doesn't work so well in a device like Glass, since there's no good way to select one out of many items on a screen. So Glass forgoes the icon method of launching apps in favor of simply telling Glass which app to launch.

You start any verbal command in Glass with the attention words "ok glass" (the same words you used before "add a caption" on a static card in Chapter 6, *Making Glass Social*, on page 75), followed by the trigger keyword you've defined. When you launch an application via speech in Glass, the voice keyword tend to be conversational. Since our application is going to show someone some stats, "show me stats" is a good voice trigger.

The first step to setting up a voice trigger is to create an XML resource like res/xml/voice_trigger.xml. The file only contains a single element trigger with a keyword that is some word or short phrase.

chapter-11/Stats/app/src/main/res/xml/voice_trigger.xml

```
<trigger keyword="show me stats" />
```

It's always good form to place the actual keyword value into the values/string.xml resource. It makes translating your app into different languages easier in the future.

Build-in Voice Triggers

In this and other examples in this book, we're creating custom voice triggers by using the keyword element. However, you can choose to use one of the built-in system triggers listed under the VoiceTriggers.Command enum by changing the keyword field to command with an enum value, such as command="CHECK_ME_IN". We are only able to use custom keywords in development.

Currently, Google will disallow submission of any GDK applications that do not use one of the system keywords. The reason behind this is to specially tune the voice input to recognize the system words, in English and in other languages. If you desire that Google adds a new keyword, you can request its addition https://developers.google.com/glass/distribute/voice-form.

For more information about voice trigger, check out https://developers.google.com/glass/develop/gdk/starting-glassware.

With our voice trigger resource in place, now we finally can set up our Android manifest. Our application contains both a service and an activity, so we define them both. What makes this manifest different from what we've seen before is that the intent-filter is no longer a MAIN/LAUNCHER, but instead now the action is com.google.android.glass.action.VOICE_TRIGGER. We also set our voice_trigger.xml resource to com.google.android.glass.VoiceTrigger meta-data.

chapter-11/Stats/app/src/main/AndroidManifest.xml

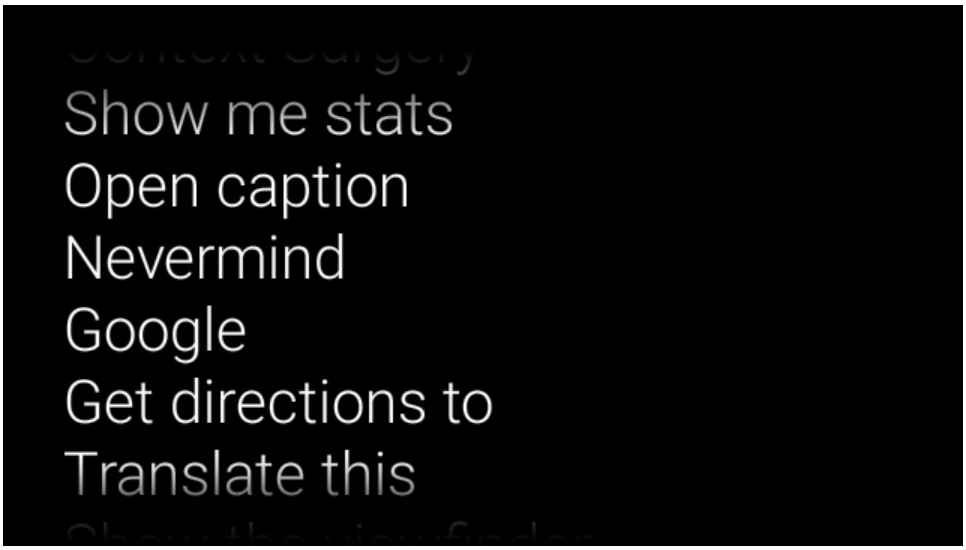
```
<service
    android:name=".StatsService"
    android:icon="@drawable/ic_glass_logo"
    android:label="@string/app_name" >
    <intent-filter>
        <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
    </intent-filter>

    <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/voice_trigger" />
</service>

<activity
    android:name=".LiveCardMenuActivity"
    android:theme="@style/MenuTheme" />
```

Only the StatsService needs to launch in response to a voice trigger. The LiveCardMenuActivity will be started when the LiveCard receives a gesture bar tap.

Surprisingly, an intent and a voice trigger resource are the only requirements for adding a new trigger in the launch list.



Adding an app with a voice trigger actually performs a double-duty, as it also adds your app to the non-verbal launch menu.

Now it's time to play with our app! As a reminder, click *Run As -> Android Application* and choose your Glass device. Once it's sideloaded onto your device, you can launch the app either verbally by saying *ok glass, show me stats*, or tap on the home card and scroll over to *Show Me Stats*.

You should be greeted by a Card that looks somewhat like Figure 39, *The Live Card is Left of the Home Card*, on page 148. However, it's incomplete. Some values, like battery voltage, are set to default values from the stats.xml layout. Also, none of the values change. We'll round out our application by updating the Stats LiveCard in response to changes in our Glass device data next.

For now, you can tap the side of your glass, and choose the menu item to *Stop* the application.

Updating RemoteViews

What we promised ourselves at the beginning of this chapter was a live card that would update itself in real time whenever device stats change. This requires tracking changes in our system state. Luckily, we already know how to do this from the Chapter 10, *An Android Introduction on Glass*, on page 125: by creating our own BroadcastReceiver that receives certain intents.

But unlike the last chapter, we aren't going to register this new component in the Android manifest. Instead, we're going to register our receiver in code. This will allow us to easily share the LiveCard and RemoteViews objects we created in the StatsService with our new receiver, and make updates whenever it receives a new Intent.

StatsReceiver

Our first task is to create a BroadcastReceiver named StatsReceiver—this rounds out our project rather nicely with three intent components we learned about in the last chapter, don't you think?

Like most receivers, its behavior is rather basic. It will accept a LiveCard and a RemoteView. When onReceive is called, it will check which intent was fired, and change the Card's view based on that information.

chapter-11/Stats/app/src/main/java/glass/stats/StatsReceiver.java

```
public void onReceive(Context context, Intent intent) {
  // set the current time no matter what
  rv.setTextViewText(R.id.time, StatsUtil.getCurrentTime(context));

  // if connection status changes, set it
  if (Intent.ACTION_POWER_CONNECTED.equals(intent.getAction())) {
    rv.setTextViewText(R.id.connected, context.getString(R.string.connected));
  }
  if (Intent.ACTION_POWER_DISCONNECTED.equals(intent.getAction())) {
    rv.setTextViewText(R.id.connected, context.getString(R.string.disconnected));
  }

  // if a battery value changes, update them all
```

```
  if (Intent.ACTION_BATTERY_CHANGED.equals(intent.getAction())) {
    rv.setTextViewText(R.id.battery_degrees, StatsUtil.getDegrees(intent) + "C");
    rv.setTextViewText(R.id.battery_voltage, StatsUtil.getVoltage(intent) + "V");
    rv.setProgressBar(R.id.battery_level, 100,
        StatsUtil.getBatteryLevel(intent), false);
  }

  if (WifiManager.NETWORK_STATE_CHANGED_ACTION.equals(intent.getAction())) {
    rv.setImageViewResource(R.id.wifi_strength,
        StatsUtil.getWifiIconResource(context));
  }

  // inform the livecard of the changes to the views
  liveCard.setViews(rv);
}
```

Simply updating the RemoteViews values is not enough. Remember that a RemoteViews is not an actual View, but rather a bundle of changes that can be passed to another process. You have to pass the object to liveCard.setViews() to display the changes on the live card, which is the last call of the onReceive() method.

Register the Receiver

Coding the StatsReceiver is half of the work, now we need to associate the receiver with some intents. Recall in the last chapter, we added the SimpleReceiver to the app manifest along with an intent-filter with two actions—ACTION_POWER_CONNECTED and ACTION_POWER_DISCONNECTED. We codify a similar association in a new StatsService method called buildReceiver, which constructs an IntentFilter object.

We also create the StatsReceiver object, passing it and the intent filter into the registerReceiver method. Now our StatsReceiver is registered to fire whenever a power cable is connected or disconnected.

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

```
private void buildReceiver(RemoteViews rv) {
    IntentFilter filter = new IntentFilter();
    filter.addAction(Intent.ACTION_POWER_CONNECTED);
    filter.addAction(Intent.ACTION_POWER_DISCONNECTED);
    filter.addAction(Intent.ACTION_CONFIGURATION_CHANGED);
    filter.addAction(Intent.ACTION_BATTERY_CHANGED);
    filter.addAction(Intent.ACTION_TIME_TICK); // update every minute
    filter.addAction(WifiManager.NETWORK_STATE_CHANGED_ACTION);
    receiver = new StatsReceiver(liveCard, rv);
    registerReceiver(receiver, filter);
}
```

Don't forget to actually call the buildReceiver method after the liveCard is constructed in StatService's onStartCommand.

```
public int onStartCommand(Intent intent, int flags, int startId) {
  if( liveCard == null ) {
    ...
    buildReceiver(rv);
  }
  return START_STICKY;
}
```

Now when you reinstall your app, you should see several values that will update every few seconds. Your battery gauge is measured in percentage from full, your battery temperature shouldn't be much higher than 30°C, and hopefully your wifi signal icon is at full power.

It's worth noting that you can alter this card to contain any system values you want to see. You could register compass sensor events that list the direction you're looking, geolocation events that display your current latitude/longitude, or Bluetooth signal health. The sky's the limit.

Frequency Rendering

In the beginning of the chapter I mentioned that there were three ways to render a view in a LiveCard, a simple way and two hard ways. It's more accurate to say a low frequency rendering and two high frequency rendering options. We investigate low frequency rendering in this chapter, where a RemoteViews object is constructed, then passed to the LiveCard which converts the assets into a View on the Glass timeline process. This is sufficient if the view need only be updated every second or so.

But for higher frequency rendering, you'll need to access the drawing surface directly. That will be the topic of the next chapter.

Wrap-Up

You'll rarely make a Live Card app with fewer than two Java classes: the Service context that launches and cleans up the LiveCard, and a menu Activity. We went for the trifecta by also adding in a BroadcastReceiver to change the view by updating RemoteViews. We also chose to make our lives simpler by defining our views using an XML layout, rather than writing a big view hierarchy in code.

The ability to update values in a single card in real time is a new behavior that you can only perform by using the GDK. Mirror API-based static cards could be updated, but they aren't meant to be used live—they are meant to

be a reference point in history. Unless you're a fan of rewriting the past, use LiveCards for displaying mutable states that always represent *now*.

So far, our card hasn't looked all that different from a static card. Sure it can update live, and the look is livened up a bit, the low frequency rendering via LiveViews can't support the kind of smooth, on the fly rendering you get when viewing a video. In the next chapter we're going to expand our LiveCard to support a more robust visual and UI experience.

CHAPTER 12

Advanced Rendering and Navigation

As we've seen repeatedly, Google Glass is primarily a visual device. Because of this, it's fair for users to expect a rich visual environment comparable to modern mobile devices. And while static cards, and low frequency rendering LiveCard RemoteViews are functional, there's nothing quite like seeing smooth scenes rendered before your eye. That's exactly what we're going to implement in this chapter, by digging into high frequency rendering. Beyond simply delighting users, high frequency rendering is necessary in some cases, like displaying video, or as we'll eventually see in Chapter 14, *Creating an Immersive Game*, on page 219, creating video games.

One other topic we'll look into in this chapter is supporting more complex interfaces. Up until now all of our interfaces have been pretty basic single cards with a menu. But Glass supports much more, and we'll make our apps more interactive with scrollable multi-card views. It can take a bit of code, but the concept is simple enough.

But before we get ahead of ourselves, let's take a peek at the project *du jour*.

High Frequency Rendering

To get this party started, we're going to design a live card that renders balloons floating up the screen (you can't have a party without balloons). We'll allow the user to change the number of balloons they see rendering on the scene by flipping through a selection of cards via a card scroll adapter and view. When the user selects a number, the scene will be recreated with the new balloon count.

Before we jump into the code, let's sketch out how we want our application to ultimately work. It doesn't have to be pretty, but a quick napkin sketch like the following figure can save us a lot of hand wringing in the future.

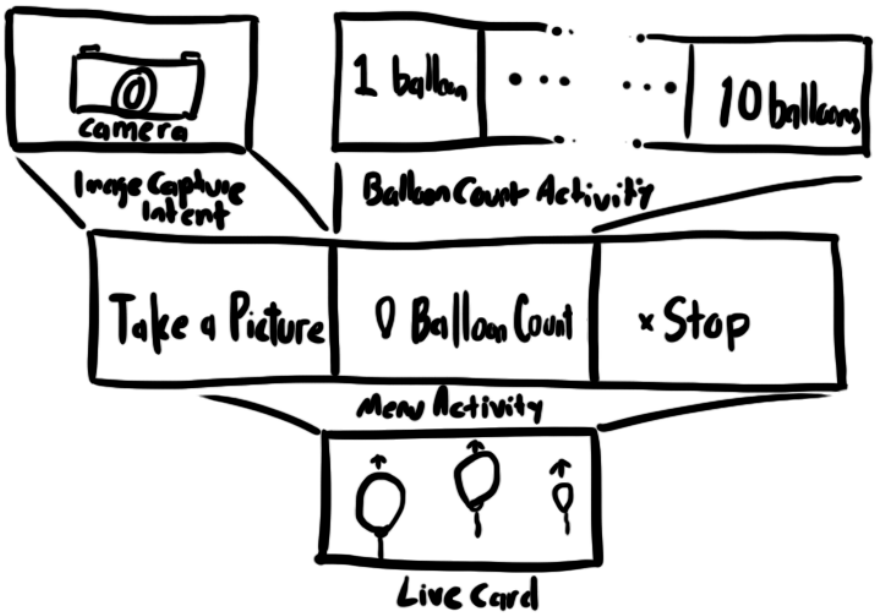


Figure 41—It's always a good idea to sketch-out the app

The simplest place to start is quite close to where our app left off at the end of Chapter 11, *Live Cards*, on page 147. Let's create a new application called Partify, that has a single service to manage our LiveCard, and a menu. We'll also need a voice trigger of `lets party` to start the application, a `menu.xml` with the required `stop` item to finish the application, and the `PartyService` and `LiveCardMenuActivity` defined in the `AndroidManifest.xml`. This is all pretty pedestrian stuff so far.

But we want to render balloons to smoothly float up the screen, and RemoteViews just aren't up to the task. Their refresh of once a second-or-so is simply too slow. So rather than calling `liveCard.setViews(remoteview)` in our service's `onStartCommand` method, we'll replace that line with the following.

chapter-12/Partify/app/src/main/java/glass/partify/PartyService.java

```
renderer = new LiveCardRenderer(this);
liveCard.setDirectRenderingEnabled(true);
liveCard.getSurfaceHolder().addCallback(renderer);
```

What we're doing here is flipping a switch in the liveCard to allow our application to draw pixels directly onto the card's surface, rather than providing a bundle of views to render remotely on its own. This gives us far more control

over exactly how and when to render, but it is far more complex and lower level than a RemoteView.

The last line adds a callback to the card's surface holder. A SurfaceHolder[1] is just a regular Android object that contains a reference to the Google Glass screen's drawing surface, and makes a few calls to the attached callback object. We're responsible for implementing this callback object, to inform the SurfaceHolder about the pixels we'd like to render, which we do in a class named LiveCardRenderer.

Getting App Assets

Before we define our LiveCardRenderer callback object, we need to have something worth drawing. For this, we'll finally use the assets directory that we see generated in every project. As you can imagine, the directory holds assets that the application will use: image files, data files, audio... anything binary. In our case, we have three balloon files named red_balloon.png, green_balloon.png and yellow_balloon.png.

To access these images, we'll create a class that represents a balloon that will float up the screen. A Balloon will contain a bitmap image of one of the three colors scaled to a certain size, which floats at a relative speed—far away things appear to be smaller and move slower.

chapter-12/Partify/app/src/main/java/glass/partify/Balloon.java

```
public class Balloon {
  enum Color {
    GREEN("green"), YELLOW("yellow"), RED("red");
```

1. http://developer.android.com/reference/android/view/SurfaceHolder.html

```
    // ...skipping basic enum methods
  }

  private Bitmap balloonImage;
  public Balloon(Context context, String color, double percentSize,
                 int initialLeft, int initialTop)
    throws IOException {
    // get a bitmap asset and resize it
    String filename = color + "_balloon.png";
    InputStream stream = context.getAssets().open(filename);
    Bitmap image = BitmapFactory.decodeStream(stream);
    int width = (int)(image.getWidth() * percentSize);
    int height = (int)(image.getHeight() * percentSize);
    balloonImage = Bitmap.createScaledBitmap(image,width,height,true);

    this.left = initialLeft;
    this.top = this.initialTop = initialTop;
    // restart when the balloon when it reaches past the screen
    this.endTop = -height;
    // give the balloon a speed related to size
    this.speed = (int)(10 * percentSize)+1;
  }
  // ...skipping other helper methods
}
```

You can download the source code for the book for a full view of the Balloon class.

The only Android-specific code here gets the bitmap image context.getAssets().open(...), and then decodes the image and resizes (*scales*) it. There's also a method called nextTop() that increments the top field at a certain speed until it reaches endTop, then starts back at the initialTop. Each frame will draw the balloon a bit closer to the top by a few pixels, creating the illusion the the objects are floating vertically to the top.

With that housekeeping out of the way, let's create our SurfaceHolder's LiveCardRenderer callback.

Drawing on the Surface

The LiveCardRenderer callback must implement the DirectRenderingCallback interface. This interface's methods provide hooks to draw your own pixels onto the Glass screen's surface. When your callback is registered, your code can react when the surface is *created*, *changed*, *destroyed* or *(un)paused*.

What Happened to SurfaceHolder.Callback?

> If you're a seasoned Android programmer, this new DirectRenderingCallback interface might come a surprise. In Android you would have implemented the SurfaceHolder's own Callback. Why the change? Because Glass is a small device with a small battery. To conserve power, it goes to sleep relatively often, and your app should *pause* rendering whenever the surface is not visible.

The DirectRenderingCallback methods are named: surfaceCreated, surfaceChanged, surfacePaused, and surfaceDestroyed. Our class also needs to keep the Context that created this callback object—in our case, PartyService. It's necessary to get our Balloon assets later. Next, we need the SurfaceHolder object passed into surfaceCreated to draw on later. Finally, we create a RenderThread object, which we'll cover shortly.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardRenderer.java

```
public class LiveCardRenderer implements DirectRenderingCallback {

    public static final int DEFAULT_BALLOON_COUNT = 3;
    private Context         context;
    private SurfaceHolder   holder;
    private DrawFrameThread thread;
    public LiveCardRenderer(Context context) {
        this.context = context;
    }
    public void surfaceCreated(SurfaceHolder holder) {
        this.holder = holder;
        this.thread = new DrawFrameThread();
            loadBalloons(DEFAULT_BALLOON_COUNT);
        this.thread.start();
    }
    public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) {
        // nothing to do here
    }
    public void renderingPaused(SurfaceHolder surfaceholder, boolean pause) {
        this.thread.pause(pause);
    }
    public void surfaceDestroyed(SurfaceHolder holder) {
        this.thread.exit();
        this.thread = null;
        this.holder = null;
    }
    // there's more code coming...
```

Most of the work in these methods is being farmed out to the RenderThread object that we'll see in a bit. This thread starts when the surface is created, is paused when the surface is paused, and exits when the surface is destroyed.

But what we want to focus on right now is the call to loadBalloons, which populates three Balloons of random size/speed, position, and color.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardRenderer.java

```
private Balloon[] balloons;
public synchronized void loadBalloons(int balloonCount) throws IOException {
    if( thread != null )  thread.pause(true);
    balloons = new Balloon[balloonCount];
    Random rand = new Random();
    for( int i = 0; i < balloonCount; i++ ) {
        // what color balloon
        Balloon.Color[] colors = Balloon.Color.values();
        String color = colors[rand.nextInt(colors.length)].toString();
        // what size should the balloon be
        double percentSize = rand.nextDouble();
        // how far left to does the balloon rise
        int left = rand.nextInt(640);
        balloons[i] = new Balloon(context, color, percentSize, left, 360);
    }
    if( thread != null )  thread.pause(false);
}
```

Notice that we pass in the context object to new Balloon, so that the Balloon constructor can access the image asset. Since this callback isn't an Android context itself, it has to borrow a lot of power from the Service or Activity that ultimately created it.

With our renderer poised to do *something*, and a handful of balloons, we're ready to start drawing…but how do we do that, exactly?

The Animation Loop

As you likely assumed, the RenderThread will actually extend Java Thread. We'll implement it as a nested class within LiveCardRenderer. This thread will run in the background, independent of the main thread that runs our application. This is a very crucial component of our drawing logic, and it's pretty easy to imagine why.

Because we're animating a scene, we're going to have to draw our balloons frame by frame, over and over. This requires an animation loop. But looping on the main thread won't work, since an unending loop will lock it from any future user actions. So we run our loop on a separate thread.

Fruity Loops

There are many ways to manage an animation loop in Android. If you look in other books, or other example projects, you'll likely run across a few. Some will employ a

Looper object, while others will use a Handler queue thread, with runnables that recursively post themselves to a handler.postDelayed. Whatever the method you choose, you just must take care that the loop doesn't get weird and fruity. The style in this book is the most basic Game Loop [a] style.

Whatever you prefer, your animation loop should execute in its own thread. It shouldn't block the user from performing other actions. Make sure it doesn't consume too many resources by running too often or creating objects. Last but not least, ensure it actually stops running (via join()s) when the application is finished, or you're going to have a bad time.

a. http://en.wikipedia.org/wiki/Game_programming#Game_structure

While the running variable is true, we keep looping. We're aiming for 24 frames a second (or about 42 milliseconds per frame). However, it's likely that our draw function will run faster than that. So we track the time it takes to draw a frame in frameLength, then cause the thread to sleep for the difference. Say that frameLength is 20 ms, then we pause the render thread for 22 seconds befor waking and drawing again.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardRenderer.java

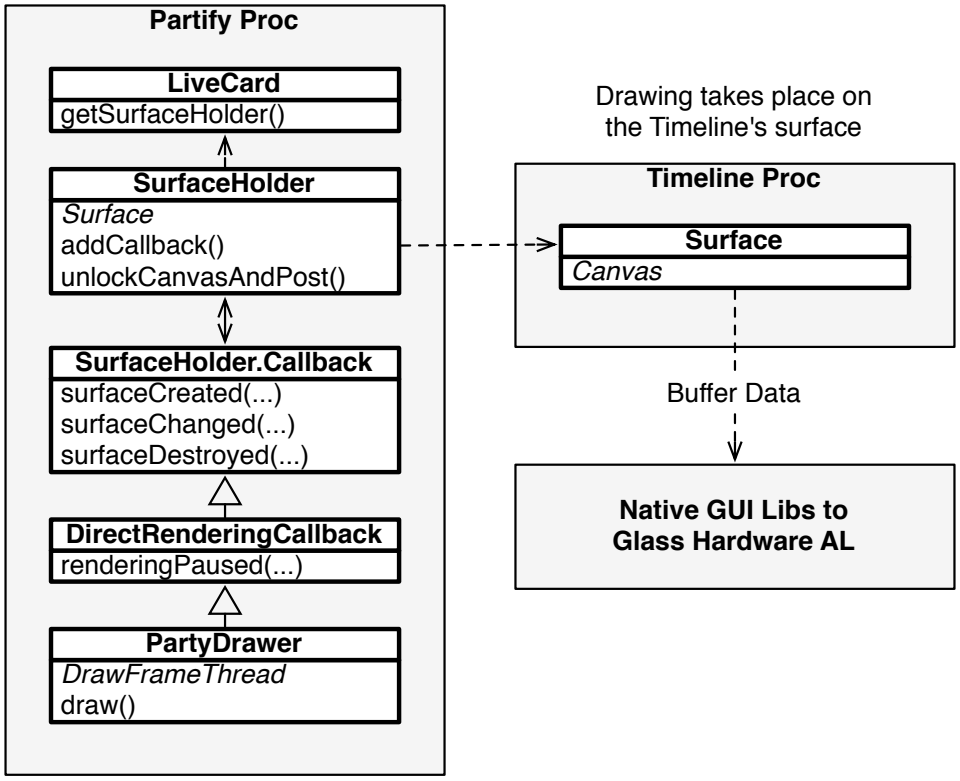
```
class DrawFrameThread extends Thread {
    private static final int FPS = 24;
    private boolean running;
    private boolean pause;
    public void run() {
        pause = false;
        running = true;
        while( running ) {
            long frameStart = SystemClock.elapsedRealtime();
            if( !pause )  draw();
            long frameLength = SystemClock.elapsedRealtime() - frameStart;
            long sleepTime = 1000 / FPS - frameLength;
            if (sleepTime > 0) {
              SystemClock.sleep(sleepTime);
            }
        }
    }
    public void pause(boolean pause) {
        this.pause = pause;
    }
    public void exit() {
        pause = true;
        running = false;
        join();
    }
```

```
}
```

When we're done with our thread, exit() will set running to false, and join() this draw frame thread back up with the main thread. This kills the animation loop.

Drawing

So far we've created some balloon assets, a DirectRenderingCallback object that the SurfaceHolder can call back against, and an animation loop that will run at 24 frames per second. But the one thing we haven't done yet is actually draw anything on the screen. Here is a partial UML of what is being called.



The RenderThread run() method calls draw() every frame, which we'll implement as a LiveCardRenderer method. Since we already have our balloon bitmap images, rendering to the holder is surprisingly straightforward.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardRenderer.java

```
private void draw() {
    if( this.balloons == null ) { return; }
    final Canvas canvas;
```

```
    try {
        canvas = this.holder.lockCanvas();
    } catch (Exception e) { return; }
    if( canvas == null ) { return; }
    synchronized( LiveCardRenderer.this ) {
        canvas.drawColor( Color.BLACK );
        for( int i = 0; i < this.balloons.length; i++ ) {
            Balloon b = this.balloons[i];
            if( b.getBitmap() != null ) {
                canvas.drawBitmap( b.getBitmap(), b.nextLeft(), b.nextTop(), null );
            }
        }
    }
    this.holder.unlockCanvasAndPost( canvas );
}
```

The magic happens with the holder.lockCanvas() and holder.unlockCanvasAndPost(canvas) methods. The SurfaceHolder object gives you a Canvas[2] object on which to draw. The Canvas class has many methods, but in the realm of drawing surfaces it's pretty basic. You can draw circles, lines, text, set colors, modify the pixels to rotate or skew, and just post a bitmap image.

Drawing on the canvas doesn't draw to the screen directly, but rather it describes how the surface *should* be drawn on. It's not until unlockCanvasAndPost is called that the instructions are drawn onto a bitmap that is then shown on the screen for a brief moment, until it's drawn over 42 milliseconds later at the next frame.

This is the last necessary method for our high frequency rendering to work. If you sideload the project and say *ok glass, let's party*, you should see three balloons in random positions, sizes, speeds, and colors floating vertically up your Glass screen, similar to the following figure.

2. http://developer.android.com/reference/android/graphics/Canvas.html

Scrolling Through Multiple Cards

Seeing a couple balloons float up the screen is fun indeed, but sometimes three just isn't enough for a party. Sometimes, it takes six, or maybe even a ten balloons! If we recall from Figure 41,*It's always a good idea to sketch-out the app*, on page 168, the design is to add a menu option that allows the user to scroll through cards to pick the number of balloons they see.

Rather than tackling this project with the menu item first, let's work backwards a bit, starting with how the LiveCardRenderer will be made aware of a change in balloon count. We already have the loadBalloons method, but who will be responsible for calling it? Since the PartyService created the renderer, it makes sense that it should do the honors.

chapter-12/Partify/app/src/main/java/glass/partify/PartyService.java

```
public void setBalloonCount(int balloonCount) {
    this.balloonCount = balloonCount;
    if (renderer != null) {
            renderer.loadBalloons(balloonCount);
    }
}
public int getBalloonCount() {
    return balloonCount;
}
```

Since the balloon count choice is going to be launched from the LiveCardMenuActivity, it will have to pass the value to the PartyService. The problem is, LiveCard-

MenuActivity doesn't have any reference to PartyService. We'll have to connect them somehow.

Binding the Service

Have you noticed that every time you create a Service class in the IDE, there's a method named onBind that we just let return null? Well, now we need it. The onBind method is how a Service lets other components communicate with it. All we have to do is implement a Binder object that performs whatever actions we want. Binders are a very flexible way to communicate between components, and these communications can even happen across processes (inter-process communication, or *IPC*).

But in our case, we just want to allow our local LiveCardMenuActivity component to access our PartyService instance, and the easiest way to go about that is just returning PartyService.this.

chapter-12/Partify/app/src/main/java/glass/partify/PartyService.java

```
public class GifferBinder extends Binder {
    public PartyService getService() {
        return PartyService.this;
    }
}
@Override
public IBinder onBind(Intent intent) {
    return binder;
}
```

The other side of this equation is the ServiceConnection, which is responsible for connecting to a service and doing something with the IBinder it receives. When our LiveCardMenuActivity is created, we'll call bindService, passing in an explicit intent for the PartyService component, and a ServiceConnection instance.

The bindService method will call PartyService's onBind method, and pass that IBinder to our ServiceConnection's onServiceConnected method. From there, it's a simple matter of getting the PartyService instance.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardMenuActivity.java

```
private ServiceConnection serviceConnection =
        new ServiceConnection() {
            public void onServiceConnected(ComponentName name, IBinder binder) {
                if (binder instanceof PartyService.GifferBinder) {
                    service = ((PartyService.GifferBinder) binder).getService();
                    if( hasWindowFocus() ) {
                        openOptionsMenu();
                    }
                }
                unbindService(this);
```

```
            }
            @Override
            public void onServiceDisconnected(ComponentName name) {}
        };

protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    bindService(new Intent(this, PartyService.class), serviceConnection, 0);
}
```

Notice in the onServiceConnected method, after we extract the PartyService instance we immediately unbind the service. That's because we only want the Partyservice object, not the binding. After we have what we want, we no longer need the connection.

With our service instance in hand, we can move on to adding a scrollable list of balloon counts.

The Balloon Count Adapter

Android has many built-in design patterns, driven by common use-cases. One of the more commonly used patterns is creating UIs where a list view is populated by some underlying list datasource. For example: a Twitter app that contains an infinitely scrollable list, and the list is populated by an array of data from the twitter web service.

Android splits this common case into two sides: the Adaptor which represents the data, and the AdapterView, which is a visualization of that collection. In plain old Android app (like the Twitter app), an AdapterView will likely be a scrollable ListView, or a Gallery of items.

Glass provides an implementation for each side of this pattern to scroll through a list of cards: a CardScrollAdapter to manage some data, and the CardScrollView to visualize it. Alongside the adapter pattern, we also need to create an Activity (called BalloonCountActivity) to not only display the CardScrollView, but also allow the user to swipe through the list of cards, then tap on the number they want. We'll track the user's motion events with a GestureDetector.

These classes are the critical ingredients to scroll through a list of cards, each with the number of balloons we'd like to see. Let's start with creating a CardScrollAdapter called BalloonCountScrollAdapter.

This is one of the simplest adapters you'll likely ever see. The thing to keep in mind is that an adapter has two jobs: to act as a bridge between a set of data and to provide a view for each data point. In our case, since we only

want to render a view to choose one of ten numbers (0-9), we only have to convert a card's (or item's) position into a number in the getPosition() method.

The adapter manager gets our objects from getItem(). An item could be anything, such as an element in an array of Strings. For us, the card position *is* the item object as an Integer object, so we need only convert the position to an Integer with Integer.valueOf(position).

chapter-12/Partify/app/src/main/java/glass/partify/BalloonCountScrollAdapter.java

```
public class BalloonCountScrollAdapter extends CardScrollAdapter {
  public final static int MAX_BALLOONS = 10;
  private final Context   context;
  public BalloonCountScrollAdapter(Context context) {
    super();
    this.context = context;
  }
  public int getCount() {
    return MAX_BALLOONS;
  }
  public Object getItem(int position) {
    return Integer.valueOf(position);
  }
  public int getPosition(Object item) {
    if( item instanceof Integer ) {
      int idInt = (Integer) item;
      if (idInt >= 0 && idInt < getCount()) {
        return idInt;
      }
    }
    return AdapterView.INVALID_POSITION;
  }
  public View getView(int position, View convertView, ViewGroup parent) {
    if( convertView == null ) {
      convertView = LayoutInflater.from(context)
          .inflate(R.layout.balloon_count, parent);
    }
    TextView countView = (TextView)convertView.findViewById(R.id.count);
    countView.setText(position + " balloons");
    return convertView;
  }
}
```

Most of the methods in the adapter are about getting items, by position or id or just a count of all items. But there is one method, getView, that converts an item in a certain position into a View. When getView is called internally by the CardScrollView, we set the text in a text view to be that position + " balloons", e.g. *8 balloons*. We could have created a view with code, but in this case we opted to create a balloon_count layout.

That's all it takes to make a scrollable view. It's not a lot of code, but it's a useful construct to understand if you want to make Glass apps with more complex interfaces than just a single card.

Gestures and Sound Effects

Our goal is to make a menu option where a user chooses from a list of numbers, then the Partify app renders that many balloons on the screen. We're halfway through that mission. Now have have to add the ability for our user to scroll through the list and pick a balloon count card. This requires detecting the user's gestures made on the Glass device. When a user swipes his or her finger back and forth across the gesture bar, we want to scroll the cards. When a user taps on the card scroller, we want to get the current item shown.

Happily, Glass has provided an object to capture swipe and tap gestures, called the GestureDetector.We'll create this detector in a new class BalloonCountActivity, which will manage displaying the card scroll view and detecting gestures.

The Balloon Count View and Activity

Before we write our Activity, we still have to code the other half of the adapter/view pattern. The BalloonCountScrollView will be our implementation of the CardScrollView, which is a specialized AdapterView.

Most of the work that the view will do is implemented in the CardScrollView base class itself, along with our BalloonCountScrollAdapter getView method. All we have to do is give the view a Context (our yet-to-be-written Activity), and a GestureDetector. Then we dispatch the scroll event to our own GestureDectector. Most CardScrollViews you'll ever implement will look like this.

chapter-12/Partify/app/src/main/java/glass/partify/BalloonCountScrollView.java

```
public class BalloonCountScrollView extends CardScrollView {
  // NOTE: this is not a android.view.GestureDetector class
  private GestureDetector gestureDetector;

  public BalloonCountScrollView(Context context, GestureDetector gestureDetector) {
    super( context );
    this.gestureDetector = gestureDetector;
  }

  public boolean dispatchGenericFocusedEvent(MotionEvent event) {
    if( gestureDetector.onMotionEvent(event) ) { return true; }
    return super.dispatchGenericFocusedEvent(event);
  }
}
```

With the easy View out of the way, we now get to write the Activity. Sticking with our *balloon count* theme, we'll name it BalloonCountActivity. Don't forget to add this activity to the AndroidManifest.xml.

You may immediately notice that, unlike other activities we've seen, this one implements GestureDetector.BaseListener. We'll get back to that in a moment. For now, we should see some familiar faces: onCreate, onResume, onPause. Most of the work is in onCreate, but it's hardly heavy lifting—we're just creating new objects.

chapter-12/Partify/app/src/main/java/glass/partify/BalloonCountActivity.java

```
Line 1  public class BalloonCountActivity
        extends Activity
        implements GestureDetector.BaseListener
{
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    audioManager = (AudioManager)getSystemService(Context.AUDIO_SERVICE);
    gestureDetector = new GestureDetector(this).setBaseListener(this);
    // configure the adapter/view relationship
    adapter = new BalloonCountScrollAdapter(this);
    scrollView = new BalloonCountScrollView(this, gestureDetector);
    scrollView.setAdapter(adapter);
    setContentView(scrollView);
  }
  public void onResume() {
    super.onResume();
    scrollView.activate();
    scrollView.setSelection(getIntent().getIntExtra(EXTRA_CURRENT_COUNT, 3));
  }
  public void onPause() {
    super.onPause();
    scrollView.deactivate();
  }
  // more code to come...
```

The interesting lines here are where we associate (setAdapter) the adapter with the scrollView on line 12, then set scrollView as the activity's content view (setContentView) on the following line. This is how we wire up our card scroll adapter/view pattern, and make it available as part of this Activity.

As for onResume and onPause, they just activate and deactivate rendering of the view, depending on the state of the activity. We'll talk about the last line of onResume later.

But what about the gestures and audio?

GestureDetector and AudioManager

Because we set this Activity to implement GestureDetector.BaseListener, it must implement a method called onGesture. BaseListener implies that this object will receive all gesture detection results, and that we want the opportunity to consume them.

The Activity informed the GestureDetector that it was going to handle gesture bar events when it called setBaseListener(this) in onCreate. We could have created a custom class to implement that one method, but overloading contexts to implement callback interfaces is actually a common Android practice, which we'll take advantage of for the remainder of this book.

But the GestureDetector isn't magic. It is only a wrapper for a lower level Android class called MotionEvent. Motion events are handled by the current activity, and are pretty generic things. Android was originally written with touchscreens in mind, but can support all kinds of input events, even joysticks. Swiping and tapping on the screen are but one kind of event.

What we need to do is access motion events captured from the activity, and pass it into the GestureDetector object.

chapter-12/Partify/app/src/main/java/glass/partify/BalloonCountActivity.java

```
public boolean onGenericMotionEvent(MotionEvent event) {
  return gestureDetector.onMotionEvent(event);
}
```

Now the GestureDetector has all of the information it needs to call the BaseListener's (in our case, our activity) onGesture method, in response to some user event.

chapter-12/Partify/app/src/main/java/glass/partify/BalloonCountActivity.java

```
public boolean onGesture(Gesture gesture) {
  if( Gesture.TAP.equals(gesture) ) {
    int balloonCount = scrollView.getSelectedItemPosition();
    Intent result = new Intent().putExtra(EXTRA_BALLOON_COUNT, balloonCount);
    setResult(RESULT_OK, result);
    audioManager.playSoundEffect(AudioManager.FX_KEY_CLICK);
    finish();
    return true;
  }
  return false;
}
```

We only care about a gesture if a user taps the gesture bar. Since the card scroll view is the only thing visible at this point, we take a tap event to mean the user has chosen a card he or she likes. All we need to do now is get the current selected item from the scrollView, and send some results through an Intent object.

Just to make the choice feel like a natural Glass interface choice, we play the key click sound effect, which is just as simple as calling playSoundEffect(AudioManager.FX_KEY_CLICK) on the audio manager object. It's not functionally necessary, but it is good UX.

Once the result has been set and sound effect has played, we can finish() this activity. But finish to what? And where are we sending this result to? We have one last bit of glue code before our app is complete.

Launching the Balloon Count Scroller from the Menu

We've written a fair bit of code in this chapter, introduced several new topics, and laid all of the pieces out on the board. Let's finish this puzzle by returning to the LiveCardMenuActivity. A menu option we've seen before is stop, which just calls stopService, passing an explicit Intent for the PartyService.

The menu is also where we will launch the balloon count portion of the application, according to our napkin sketch. This means we have to add a new menu item that will launch the BalloonCountActivity.

Android provides a simple mechanism for launching one activity from another, and waiting around for a result to return. Just like any time we want to make our intentions known, we populate an Intent object, and call startActivityForResult along with a *request code*.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardMenuActivity.java

```
private static final int  REQ_CODE_BALLOON_COUNT = 100;

public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.stop:
            return stopService(new Intent(this, PartyService.class));
        case R.id.balloon_count:
            Intent balloonCountI = new Intent(this, BalloonCountActivity.class);
            balloonCountI.putExtra(
                    BalloonCountActivity.EXTRA_CURRENT_COUNT,
                    service.getBalloonCount());
            startActivityForResult(balloonCountI, REQ_CODE_BALLOON_COUNT);
            waitingForResult = true;
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

We've given the request code the number 100, but as long as the number is unique within our application, it's a valid code. This request code is important when launching an activity, as we're about to see.

When the activity called in startActivityForResult sets a result with the *result code* RESULT_OK and an intent, Android calls the onActivityResult. This allows you to respond to whatever information the called activity sent back. The *request code*, *result code*, and data stored in the resulting Intent are all provided to the method.

Generally, if the result code isn't RESULT_OK, we just finish the menu activity. Something went wrong. Then, depending on the request code, we perform some action. In the case of REQ_CODE_BALLOON_COUNT, we extract the EXTRA_BALLOON_COUNT value, and use it to set the balloon count on our service.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardMenuActivity.java

```
public void onActivityResult(int requestCode, int resultCode, Intent intent){
    if( resultCode != RESULT_OK ) {
        finish();
        return;
    }
    switch( requestCode ) {
        case REQ_CODE_BALLOON_COUNT:
            int balloonCount =
              intent.getIntExtra(BalloonCountActivity.EXTRA_BALLOON_COUNT, 3);
            service.setBalloonCount(balloonCount);
            waitingForResult = false;
            finish();
}
```

Where did the BalloonCountActivity.EXTRA_BALLOON_COUNT value come from? Recall BalloonCountActivitys onResume method, after the scroll view was activated? We didn't talk about this line.

```
scrollView.setSelection(getIntent().getIntExtra(EXTRA_CURRENT_COUNT, 3));
```

What this is actually saying is: get the intent used to launch this activity, and from that retrieve the extra value that it set—if you don't find one, return 3. Set that value as the current selection. This is an easy way to send state upstream to an activity you're launching.

You may also wonder what the purpose of the waitingForResult boolean value was. This is a trick to keep the menu activity alive until the result has returned.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardMenuActivity.java

```
public void onOptionsMenuClosed(Menu menu) {
    if( !waitingForResult ) {
```

```
            finish();
        }
    }
```

With that last bit of code in place, you'll have a new menu item called *Balloon Count*. Clicking on that menu item will bring up an activity with a scrollable card view, each card with a number on it. It feels almost like a miniature timeline. If you tap on the item you like, you'll hear a click sound effect, and then that value will be returned to the menu. The menu passes the number to the service and ultimately to the LiveCardRenderer, which will generate that number of Balloons, and immediately start animating them to float up the LiveCard surface.

That's quite a lot going on for a seemingly simple application. But are we finished? Can we take a breath? Not just yet, we have one more addition in store.

Taking Photos with the Camera

Balloons floating up against a black background makes a pretty weak party. It would be much more exciting if those balloons floated against a more interesting background. Sure, we could provide one, but it would be more fun to let users take their own background photo. To do this, we'll leverage the Glass camera.

Our basic design with be like this: to start, we add a new menu item to *Take a Picture*. When that menu item is selected, the Android camera will be launched and take a photo. Once the photo has been captured, we pass that image into the PartyService, which in turn sends the image to the LiveCardRenderer. The renderer will include the background image as part of it's draw method instead of just a black background.

Let's start with the menu.

Add The Menu Request

We've added menu options before, so this is all old hat. The first step is to add a menu item to menu.xml entitled *Take a Picture*. As before, we then have to add this item to onOptionsItemSelected. And once again, we're callind startActivityForResult with an intent and a unique request code.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardMenuActivity.java

```
private static final int  REQ_CODE_TAKE_PICTURE  = 101;

        case R.id.take_picture:
            Intent captureImageIntent = new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
```

```
            startActivityForResult(captureImageIntent, REQ_CODE_TAKE_PICTURE);
            waitingForResult = true;
            return true;
```

The interesting part here is the intent action we choose to start. By using the MediaStore.ACTION_IMAGE_CAPTURE action, we're firing off a request for the Glass's own built-in camera activity.

When the activity returns a result, we'll have the following code in onActivityResult to use this new image.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardMenuActivity.java

```
case REQ_CODE_TAKE_PICTURE:
    String picFilePath =
            intent.getStringExtra(Intents.EXTRA_PICTURE_FILE_PATH);
    final File pictureFile = new File(picFilePath);
    final String picFileName = pictureFile.getName();
    // set up a file observer to watch this directory on sd card
    observer = new FileObserver(pictureFile.getParentFile().getAbsolutePath()){
        public void onEvent(int event, String file) {
            if( event == FileObserver.CLOSE_WRITE && file.equals(picFileName) ) {
                service.setImageFileName(pictureFile.getAbsolutePath());
                stopWatching();
                waitingForResult = false;
                finish();
            }
        }
    };
    observer.startWatching();
    waitingForResult = false;
```

The camera action has populated CameraManager.EXTRA_PICTURE_FILE_PATH, which contains the file path to the image. But there's a bit of a problem, and that's timing. See, it takes a bit of time for the camera to process the image file. We can't just pass the file directly to the service, because it won't be available yet.

So we have to watch the file and set the filename after it has been written. We do this with a FileObserver. The FileObserver is an Android hook into the Linux kernel's inotify change notification subsystem. When the file changes state in any way, inotify will cause the onEvent() method to be triggered with a code of the event. When the event matches FileObserver.CLOSE_WRITE, the file has been completely written.

Once the file has been observed, we can set the image file name, stop watching, and close the menu.

Rescale and Render the Image

Getting the picture is one half of the process. Now that we have a filename, we need to get the bitmap file. We already know how to do that, but the image resolution that Glass saves is much higher than the resolution of the Glass screen. So we compensate by resizing the image to the display of the screen.

chapter-12/Partify/app/src/main/java/glass/partify/PartyService.java

```
public void setImageFileName(String fileName) {
    if( renderer != null ) {
        Bitmap background = BitmapFactory.decodeFile( fileName );
        DisplayMetrics dm = getResources().getDisplayMetrics();
        int width = dm.widthPixels;   // 640px
        int height = dm.heightPixels; // 360px
        background = Bitmap.createScaledBitmap( background,width,height,true );
        renderer.setBackgroundImage( background );
    }
}
```

We take that resized image and send it to the renderer. Just like we called canvas.drawBitmap on each of the balloons, we need only draw the bitmap of the background image before drawing the balloons on top of it, for each frame.

chapter-12/Partify/app/src/main/java/glass/partify/LiveCardRenderer.java

```
public void setBackgroundImage(Bitmap background) {
    this.backgroundBitmap = background;
}

private void draw() {
        // ...after we paint the canvas black with drawColor...
        if( this.backgroundBitmap != null ) {
            canvas.drawBitmap( this.backgroundBitmap, 0, 0, null );
        }
        // ...now render the balloons overtop the image...
}
```

That's it! Now it's time to try out your fully functioning Glassware that converts any scene you take into a party, with some caveats. The camera image takes some time to load, but when it does, the menu disappears. You may have noticed other ways that this application is somewhat inefficient. Clicking on the menu can cause a bit of a delay. Loading multiple balloons can force garbage collection. We'll cover how to investigate and resolve these (somewhat intentional) design flaws in the debug chapter.

Wrap-Up

You may have found this to be a tough chapter, which it is. In one application, we've covered most of the new GDK enhancements to the Android SDK: live cards, direct rendering, gestures, card scrolling, and sound effects.

The power of Live Cards may be wielded for a range of purposes. Many of the built-in applications of Glass are LiveCards, such as Google Play's Listen app. But if you need more punch in your app, such as crafting a video game, we'll learn in the next chapter how to break out of these confines by making our own immersive environment.

CHAPTER 13

Voice and Video Immersions

Immersions are simple Android Activities that are launched and managed as Glassware, but are independent of the timeline. In other words, they immerse the user in your Glassware's specific environment. This is perfect for apps that don't make sense within the constraints of timeline, heavy multimedia apps, and those requiring complex gestures. You can probably think of a few cases, such as interactive games or audio/video applications.

This chapter and the next will investigate the wide open world of immersive glassware. It's the freest form of expression for Glassware authors, but also the most complex, so we'll start slow. This is an ensemble chapter where we'll build three simple immersion apps. We start with a simple, and surprisingly useful, activity called *Nevermind* that lets you verbally back out of your voice menu. Following that, we'll build an immersion we call *Open Caption* that converts speech into static cards. Just for fun, we'll add geolocation to pinpoint where we heard the voice. Finally, we'll create a more complex immersion that hooks into the Glass camera's video screen, scanning for—and launching—*QR codes*.

An Immersion App

Experiencing an *immersion* is pretty much how it sounds. Your immersive application completely takes over the Glass device, immersing the user in its own world—visually, aurally, and interactively. Technically, immersions are just Android activities that are run in the foreground on top of the regular Glass timeline, as shown in the following figure.

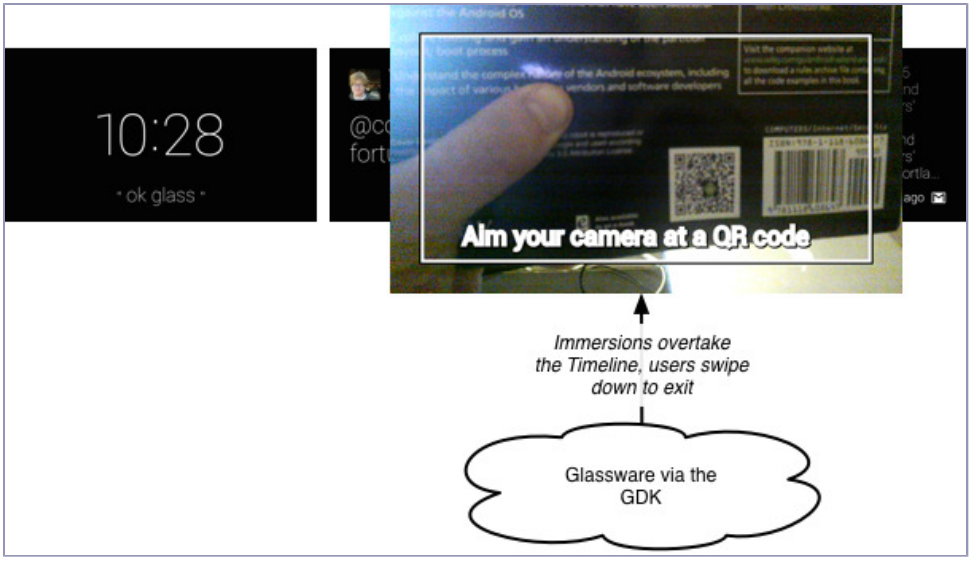


Figure 42—Immersions are Placed On Top of the Timeline

Immersions are launched like any other GDK application, through the home card app menu.

All of the projects we've created so far have interacted with the Glass timeline in some way. The Mirror API exists to populate a user's timeline with static cards through a web service. Using the GDK's Card object we explored another avenue for creating static timeline cards. Then we moved onto LiveCards, which allow us to interface with a special timeline card's rendering surface—either indirectly via `RemoveViews`, or directly through a `SurfaceHolder` callback or `GlRenderer`. Immersions are the third and final type of Glassware that the GDK provides, and the most flexible with respect to the control we have over its environment. Some of our most powerful examples will require an immersive environment, but first, let's start with something simple.

The Nevermind App

If you've used Glass for any stretch of time, you may have run into an occasional problem. Sometimes Glass thinks that it hears the words *ok glass* out of a din of background noises, and launches the application menu in response. Surprisingly, although you can launch the app menu verbally, you can't close it. So we'll rectify the situation with our first immersion called *Nevermind*.

Nevermind is the simplest app that you'll ever create, yet one you may end up using often. When you launch the app menu, then say *nevermind*, the

application will immediately close itself and thus close the menu as well. It's the Seinfeld[1] of applications—it's an app about nothing.

Most of the work to create an immersion is handled by the GDK itself. You can turn an activity into an immersion by adding the android:immersive="true" field in the AndroidManifest.xml. This tells Android to launch this Activity in the foreground as an immersion, distinct from the Timeline interface. Our voice trigger keyword is Nevermind.

```
<activity
    android:name=".CancelActivity"
    android:label="@string/app_name"
    android:immersive="true">
     <intent-filter>
         <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
     </intent-filter>
     <meta-data
        android:name="com.google.android.glass.VoiceTrigger"
        android:resource="@xml/voice_trigger" />
</activity>
```

The CancelActivity is one of the simplest Activity classes you can write. When onCreate() is called at the start of the activity lifecycle, we call the finish(), which immediately closes the activity.

chapter-13/Nevermind/app/src/main/java/glass/nevermind/CancelActivity.java

```
public class CancelActivity extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        finish();
    }
}
```

All the user will see when running this app is the menu disappearing, and control returning to the home card.

If this application seems far too simple, then that's a very good sign. Immersions are simple in the realm of Glass development, because they dispense with much of the complexity of dealing with LiveCards and the timeline. All this app really does is add a noop menu item, like in the following figure.

1. http://en.wikipedia.org/wiki/Seinfeld#Theme

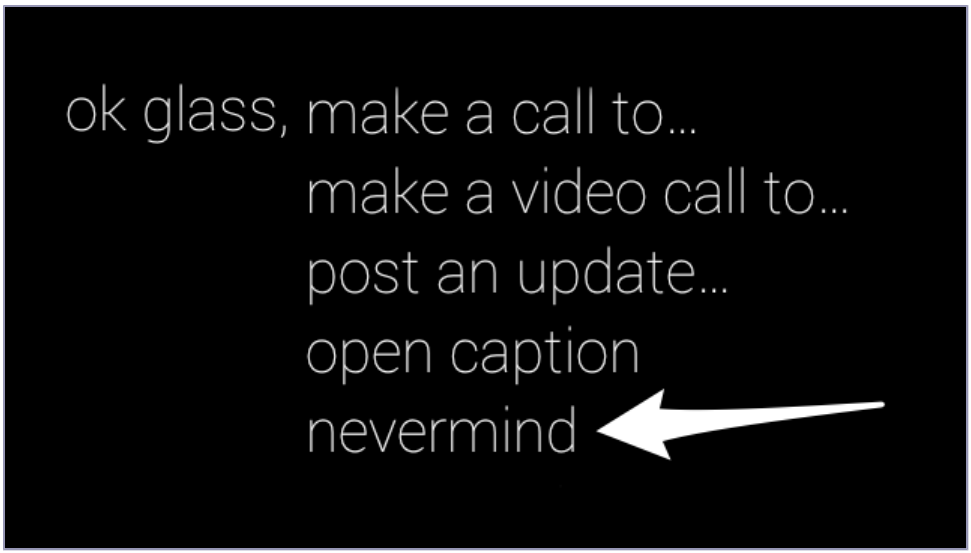


Figure 43—The Nevermind App is Little More than a Menu Item

But what about some of the cooler features of Glass, like voice inputs, camera video, gelocation, or loading third-party libraries, like a QR reader? The good news is that with immersions, you can access the full spectrum of Android application development without the constraints of LiveCards or the static timeline.

A Speech-to-Text Caption App

In many countries broadcast television supports a speech-to-text (STT) option called Closed Caption. Verbal speech or descriptions of sounds are transcribed to text on the screen for the deaf and hard of hearing. This is fine if you're watching television, but what if you are just out in the world? Or what if you just want your own personal transcription of things said?

We can take advantage of Glass's built-in voice input and speech recognizer. In the realm of STT research it's not bleeding-edge, but it's pretty good for a popular consumer option, and easy to use in our applications. We'll avail ourselves of Glass's voice input in two ways: through the application menu, and launched within our application. These two uses will give our app different flavors, and we'll start with the first one.

Voice Input from the Menu

The first immersion we create will be a simple speech-to-text application that will attempt to convert any speech collected by the microphone into text, and then store that text on a static timeline card for safe keeping.

When we attempt to launch *Open Caption*, it will first expect some input text. This voice input is part of the application launching. While this menu is up, our activity has yet to launch. Requiring voice is part of the launching process, like the following figure.

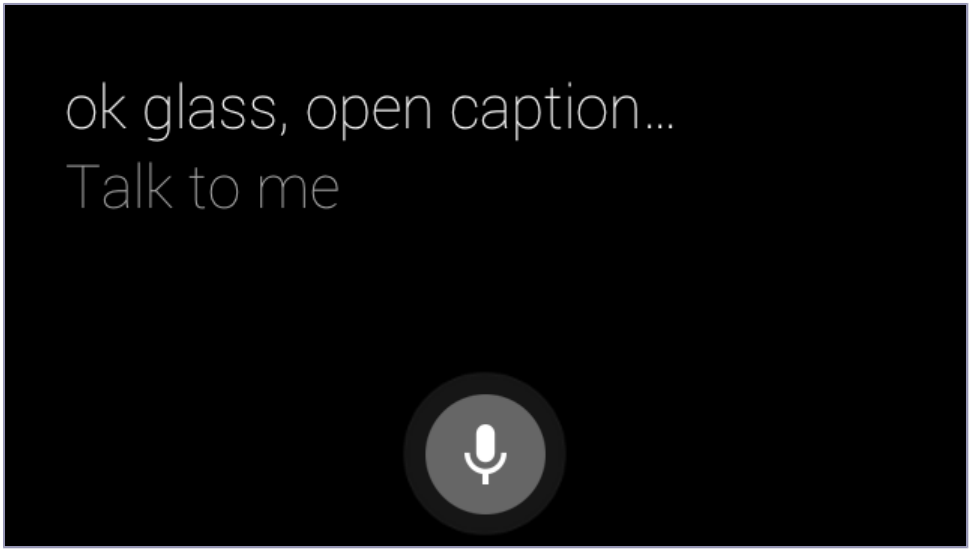


Figure 44—Voice Input from the Menu

We saw that creating an immersion is done by simply applying a configuration field (remember `android:immersive="true"`). Making the menu require a voice input is equally as simple, you just need two new elements in our `voice_trigger.xml` settings: `input` and `constraints`.

First, we want to tell Glass to disallow this app from being launched if the microphone is not available. We do this by adding a `constraints` element with a `microphone` field set to true, meaning, a microphone must be available to use this app. There are three trigger constraint fields: `microphone`, `camera`, and `network`. They aren't all required, but you can set any combination of constraints.

If a constraint isn't met for some reason, this app can't be launched. It'll also be grayed out in the home card menu list. The microphone is built-into Glass and always on, so this is more of a future-proofing exercise. Some day a ver-

sion of Glass may be available without a microphone, or possibly a setting will be added to deactivate it.

Second, we want to tell Glass that we expect a voice input along with the app launch, by adding an input element. We can optionally set a custom voice prompt, which is just some text to let the user know it's time to talk: like *Speak your message* or *Talk to me*.

```
<trigger keyword="@string/voice_trigger">
  <constraints microphone="true" />
  <input prompt="@string/voice_prompt" />
</trigger>
```

Finally, we need to actually implement our OpenCaptionActivity class. It's pretty simple. When onResume is called, we get the data from the Intent that created this activity. We take that data which was stored as an extra data string under the RecognizerIntent.EXTRA_RESULTS name, and insert a new static card with the string, and finish the activity.

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/OpenCaptionActivity.java

```
protected void onResume() {
  super.onResume();
  messageText = new StringBuilder();
  List<String> results =
      getIntent().getStringArrayListExtra( RecognizerIntent.EXTRA_RESULTS );
  if( !results.isEmpty() ) {
    // save the spoken text, we'll do something with it later
    messageText.append( results.get(0) );
  }
  finish();
}
```

That's all we need in order to get data from the app menu trigger. If you want to try it out, just say *ok glass, Open caption* followed by anything you want to say. Glass will convert that speech to text as well as it can (which is actually pretty good). When you've finished speaking, it will trigger our Activity lifecycle that uses an Intent object, just as we would if we were to start our own Activity. This is why our code calls getIntent(), and pulls the extra data stored inside the intent message.

Voice Input from Code

The app works, but it's cumbersome. Accepting voice input from the menu is fine for apps that only require input once, but we're asking a lot from our users if they have to relaunch the application every time they want some speech converted to text. It would be so much cleaner to continuously accept

speech until the user closes the app. For this, we'll have to get voice input in a different way.

We need to rethink our app design.

The voice input in Glass has two flavors. It has a built in menu interface, which we've seen. It also provides an activity which you can launch by starting an android.speech.action.RECOGNIZE_SPEECH intent action (also defined in the RecognizerIntent.ACTION_RECOGNIZE_SPEECH constant).

First, let's remove the input element from our voice trigger.

```
<trigger keyword="@string/voice_trigger">
  <constraints microphone="true" />
</trigger>
```

Next, remove the onResume() method. Since we no longer have a menu voice input, we have no need to interrogate the calling intent. Instead, we need to launch the voice input activity ourselves.

We've launched activities before, so you know the drill. First, we call startActivityForResult with an Intent containing the activity action. We also need to give the activity a unique request code, but it's all we're calling, so we'll start with 1.

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/OpenCaptionActivity.java

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    messageText = new StringBuilder();
    startSpeechPrompt( 1 );
}

private void startSpeechPrompt( int speechRequest ) {
    Intent intent = new Intent( RecognizerIntent.ACTION_RECOGNIZE_SPEECH );
    intent.putExtra( RecognizerIntent.EXTRA_PROMPT, R.string.voice_prompt );
    startActivityForResult( intent, speechRequest );
}
```

Once the voice activity is called and completed, it will return an activity result—so our next step is to implement onActivityResult. Assuming the result is ok, we do what we did in the onResume method before, which is to extract the EXTRA_RESULTS. The speech recognizer activity looks a bit different than menu voice input, as you can see in the following figure.

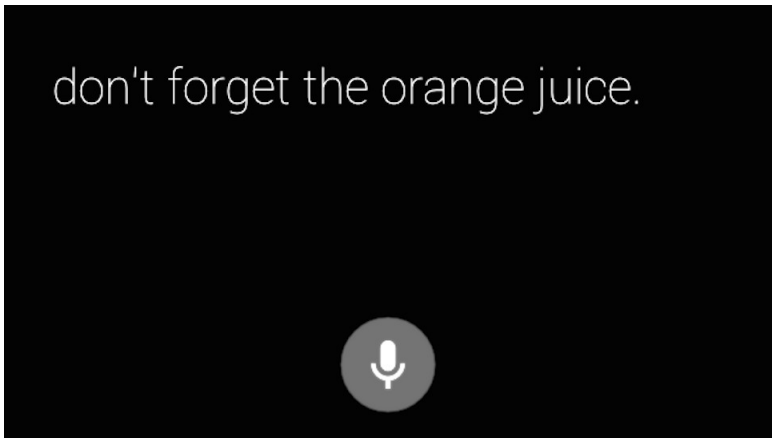


Figure 45—Running the Speech Recognizer Activity

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/OpenCaptionActivity.java

```
protected void onActivityResult(int requestCode, int resultCode,
                                Intent intent) {
    if( resultCode == RESULT_OK ) {
        List<String> results =
                intent.getStringArrayListExtra( RecognizerIntent.EXTRA_RESULTS );
        if( !results.isEmpty() ) {
            // save the spoken text, we'll do something with it later
            messageText.append( results.get(0) );
            messageText.append( "\n" );
        }
        // run recognizer again, increment the next request code
        startSpeechPrompt( ++requestCode );
    } else if( resultCode == RESULT_CANCELED ) {
        // end this activity
        finish();
    }
    super.onActivityResult(requestCode, resultCode, intent);
}
```

Since we are responsible for creating the speech recognizer activity, we should stop it, before starting a new one. We also increment the request code to keep it unique for this particular immersion. This loop is how we can continuously accept speech inputs until the user chooses to end this immersion.

When the user decides to close the app, they can swipe down in the speech recognizer activity. This will send a `RESULT_CANCELED` code, whereby we simply finish our `OpenCaptionActivity` and don't start another speech recognizer.

That's all we require to create an immersion that loops and gets Glass voice inputs strings.

So far, we've seen two different ways to access spoken text in Glass: via the voice trigger prompt, and explicitly starting the speech prompt activity. But what are we to do with this speech-to-text once we have it?

Sending Data to a Web Service

Our OpenCaption application technically works, but it's also not very interesting from an end-user point of view. They can read what they say as it happens on the screen (it's built into the speech-to-text translation screens in Glass), but if we want to save the text for later we have to do something with it. Let's keep things basic, and email these notes to ourselves when the app finishes.

Setting up SES on AWS

OK, email isn't a hot new technology, but in our case that doesn't matter. What matters is that we're connecting to a web service to send the email. We could use any web service, from storing a file to Dropbox to making a Facebook wall post.

For this example, we're using Amazon Web Service (AWS) Simple Email Service (SES). If you've never used AWS before, you can find plenty of information online to open an account, and create access key security credentials (aka, an *access key ID* and a *secret access key*). Be sure to add your email address to the *Verified Senders* list.

Next we want some Java client libraries that simplify our ability to connect to the web service. AWS provides some Android clients for each of their services, which you can get by adding `com.amazonaws:aws-android-sdk-core` and `com.amazonaws:aws-android-sdk-ses` compile dependencies to the Gradle build system's `build.gradle` file. If you're adding the version `2.1.+` jars, your `dependencies` section should now look like the following.

```
dependencies {
  compile fileTree(dir: 'libs', include: ['*.jar'])
  compile 'com.amazonaws:aws-android-sdk-core:2.1.+'
  compile 'com.amazonaws:aws-android-sdk-ses:2.1.+'
}
```

Gradle will now take care of downloading the jars and adding them to your project.

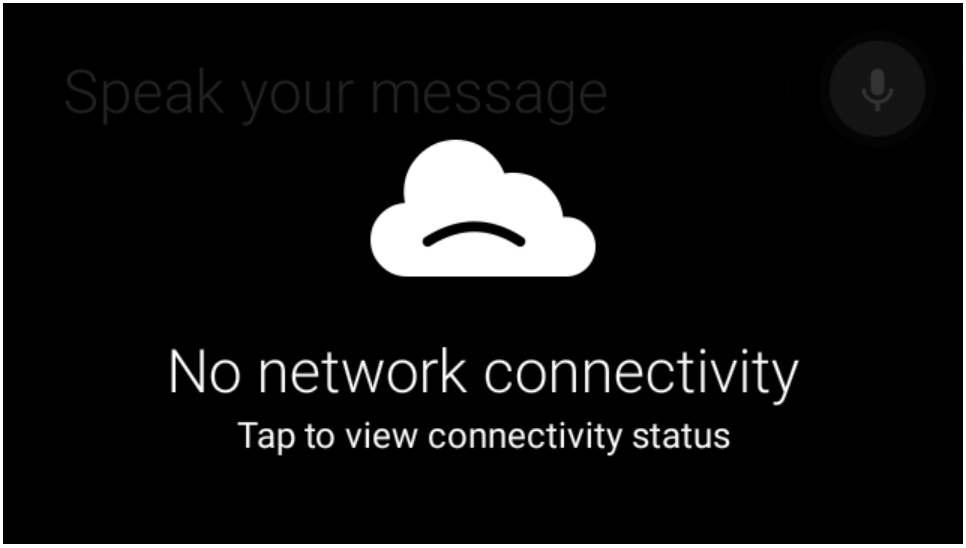
All of the setup we've done so far is to configure the the Amazon Web Service side of our application. Now we're getting to the Android/GDK side of things.

Internet Permissions and Constraints

Since our Glass device must transmit information over the Internet in order to connect to the AWS SES, you need to give the OpenCaption application permission to to so. The permission is rather obviously named android.permission.INTERNET, which you add to the manifest.

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

This actually plays a couple interesting roles. It informs Android that it requires Internet access just like any other permission. But in a slight twist from other permissions (such as WRITE_EXTERNAL_STORAGE), if Glass is not currently connected to the Internet, a card will inform the user that it is not connected. You can see the card in the following image.

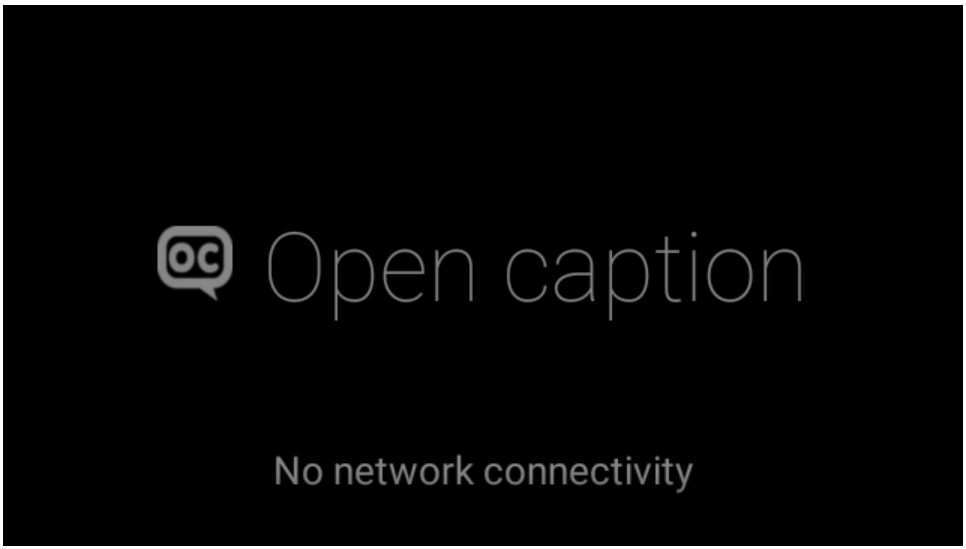


If your users tap on the message, they'll be taken to a live card that shows the source of their connection problems, shown here.

While this card will inform users that Internet access is necessary but unreachable, an ever better option would be to prevent users from launching the app at all without Internet access. If you recall the voice trigger constraints we talked about earlier, one of those constraints was network. Let's set the network field to true in voice_trigger.xml.

```
<trigger keyword="@string/voice_trigger">
  <constraints microphone="true" network="true" />
</trigger>
```

Now, if a user's Glass isn't currently connected to a network, Glass will suppress the menu option used to launch the app, and instead show the following card.



As well as a working microphone, our app also an Internet connection or it just won't work.

Making a Network Connection

We have our AWS SES service setup, and all of the client libraries, configurations, permissions, and constraints. Now is the moment of truth, where we write the code that actually connects to SES along with our speech-to-text messages. There's one more dark secret in Android network communication we've been avoiding until now.

The code to send an AWS email request is easy enough, as we'll soon see. But if we were to simply attempt to send an email directly from the `OpenCaptionActivity`, Android would throw a dreaded `NetworkOnMainThreadException`, crashing our application. If you want to perform any network operations, such as open a Socket or make an HTTP request, Android doesn't want you to run it on the main UI thread.

The main UI thread loop is what we've used for most of our example code operations (except in the case of Chapter 12, *Advanced Rendering and Navigation*, on page 167). This thread runs what the user sees and interacts with in our applications. The problem with allowing Internet connections to occur on this thread is that they are slow and unreliable. If a network connection ran in this thread, the user experience would be blocked. Users would have to wait until the network connection closed in order to continue interacting with their app. Google's engineers were so certain that opening connections

in the main thread was abusive, they only allowed network operations in a worker thread. The easiest way to create a worker thread in Android is by extending AsyncTask.

We stored the AWS access key and secret keys in the res/values/strings.xml resource, along with the SES verified email address. Normally storing secret tokens and passwords in this way is bad practice, but let's overlook that for now. In a production application you'd provide your own backend that manages individual user accounts, similar to what we did with OAuth.

The rest of the code is just AWS client code that sends the message. When a task's execute() method is called, it will pass in any number of parameters to doInBackground() in a separate worker thread. When the process is complete, the thread is closed.

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/EmailWebServiceTask.java

```
public class EmailWebServiceTask extends AsyncTask<String, Void, Void> {
    public EmailWebServiceTask( Context context ) {
        String accessKey = context.getString(R.string.aws_access_key);
        String secretKey = context.getString(R.string.aws_secret_key);
        fromVerifiedAddress = context.getString(R.string.aws_verified_address);
        toAddress = getAccountEmail( context );
        AWSCredentials credentials = new BasicAWSCredentials(accessKey, secretKey);
        sesClient = new AmazonSimpleEmailServiceClient( credentials );
    }
    protected Void doInBackground(String...messages) {
        if( messages.length == 0 ) return null;
        // build the message and destination objects
        Content subject = new Content( "OpenCaption" );
        Body body = new Body( new Content( messages[0] ) );
        Message message = new Message( subject, body );
        Destination destination = new Destination().withToAddresses( toAddress );
        // send out the email
        SendEmailRequest request =
                new SendEmailRequest( fromVerifiedAddress, destination, message );
                sesClient.sendEmail( request );
        return null;
    }
```

One aspect that we want to handle separately is accessing the current Glass device's associated Google email account. We can get that by grabbing the first matching *com.google* account type.

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/EmailWebServiceTask.java

```
private String getAccountEmail( Context context ) {
    Account[] accounts =
            AccountManager.get(context).getAccountsByType("com.google");
    Account account = accounts.length > 0 ? accounts[0] : null;
    return account.name;
```

```
}
```

Now that we've finished up our email task, we need to call it. We could choose to send an email at the completion of every speech prompt in OpenCaptionActivity. But this is asking for a flood of emails if you're a compulsive verbal note taker. Instead, we'll execute our task whenever the app is paused, then empty the string builder buffer.

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/OpenCaptionActivity.java

```
protected void onPause() {
    super.onPause();
    // email the spoken text to the Glass owner
    new EmailWebServiceTask(this).execute( messageText.toString() );
    messageText.setLength(0);
}
```

We're on a roll, so how can we make OpenCaption cooler? What other technologies make sense here? Sometimes users forget where they were when a note was made. Let's add a location note to the bottom of the email, and hopefully trigger a spark of recognition in these forgetful souls.

Adding Geolocation

One of the selling points of Google Glass is that it performs many of the same operations as other mobile devices. Geolocation has become a standard requirement in many applications, so much so, that some laptops even support location detection—though often only a guess based on an IP address.

Glass supports a form of geolocation, as we saw in the Mirror API, though technically it contains no geolocation hardware. It instead accesses the location of a paired mobile device's GPS or other passive location providers, like a cell tower. The good news is that you can write code for Glass as though it had it's own built-in hardware. Just like any other Android application, you listen for location changes by registering with the LocationManager[2] service.

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/OpenCaptionActivity.java

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    messageText = new StringBuilder();
    setupLocationManager();
    startSpeechPrompt( 1 );
}

private void setupLocationManager() {
    locationManager = (LocationManager)getSystemService(Context.LOCATION_SERVICE);
```

2. http://developer.android.com/reference/android/location/LocationManager.html

```
    Criteria criteria = new Criteria();
    criteria.setAccuracy( Criteria.ACCURACY_COARSE );
    // criteria.setAltitudeRequired(true);

    List<String> providers = locationManager.getProviders(criteria, true);
    for (String provider : providers) {
        locationManager.requestLocationUpdates(provider, 15*1000, 100, this);
    }
}
```

We get the location service, set the accuracy to coarse with Criteria.ACCURACY_COARSE. This criteria is important, because it must match a setting we add to the AndroidManifest.xml.

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

The other option is setting Criteria.ACCURACY_FINE with the android.permission.ACCESS_FINE_LOCATION permission.

Next, we need to register an object that will be notified whenever a change in location is discovered by a given location manager provider. There are a few provider options (*gps*, *network*, *passive*) but we really don't care which one Glass uses. So we request location updates for all of them, every 15 seconds within 100 meters.

The last parameter of the requestLocationUpdates method must implement the LocationListener interface. We can either create a new class, an anonymous class inline, or just take the easy way out and have our OpenCaptionActivity implement it—which is what we'll do.

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/OpenCaptionActivity.java

```
    public void onLocationChanged(Location location) {
        if( this.locationMessage == null ) {
            Toast.makeText(this, R.string.location_found, Toast.LENGTH_LONG).show();
        }
        String msg =
                location.getLatitude() + "," +
                        location.getLongitude();
        setLocationMessage( msg );
    }
}
```

Whenever the location changes, the location manager will call onLocationChanged. We want to store the latitude and longitude to display later in a little locationMessage getter/setter.

Also, the first time a location is detected we also want to let the user know by displaying a small Toast[3] message. It will detect your location asynchronously, so the message can pop up at any time, like in the following figure.

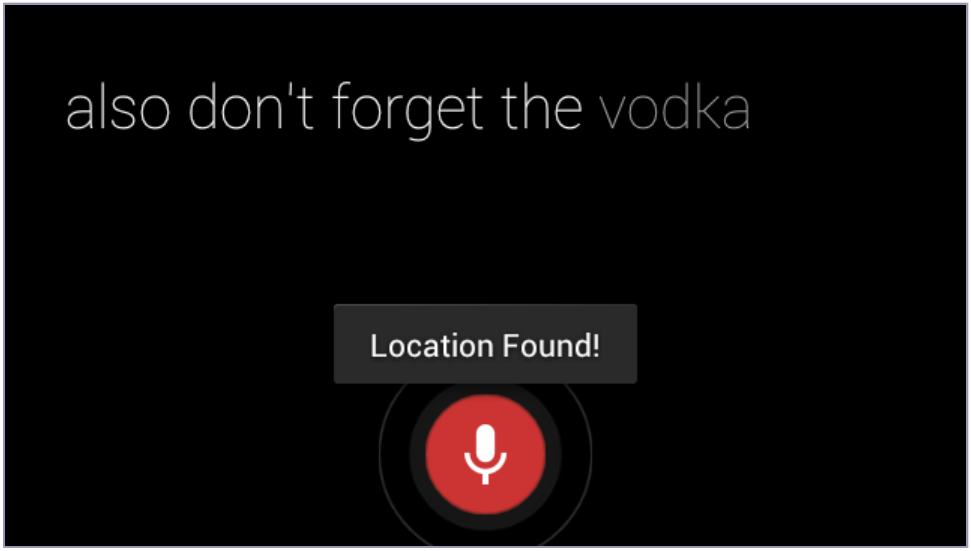


Figure 46—Geolocation is Found then Displays a Toast Message

Calling the getLocationMessage() method is an easy way to get the location, but to what end? Let's add the location to the end of every email before it's sent out.

chapter-13/OpenCaption/app/src/main/java/glass/opencaption/OpenCaptionActivity.java

```
protected void onPause() {
    super.onPause();
    // email the spoken text to the Glass owner
    messageText.append( "\nLocation: " );
    messageText.append( getLocationMessage() );
    new EmailWebServiceTask(this).execute( messageText.toString() );
    messageText.setLength(0);
}
```

It would be a small matter of code to send the geolocation to a third party service, like Google Places, and hazard a guess at a street address, instead of a pair of numbers. But we have another interesting application to write instead, which takes advantage of the Glass video camera, as well as a popular QR code library.

3. http://developer.android.com/reference/android/widget/Toast.html

A QR Code Reading Camera

Over the past two decades, QR codes have grown into the defacto method of information exchange from print to mobile devices. QR codes are popular in marketing, saving users from typing in URLs—they merely scan a poster, and the website is launched. QR codes have other uses too. Many airlines rely on them now to scan tickets at a gate, and more recently they've been used to exchange currency through Bitcoin.

What is a QR code?

QR codes are two dimensional matrices that codify data as an image label. QR stands for Quick Response, and were designed to speed up item tracking in the Japanese automotive industry. They're meant to be scanned, not unlike the UPC bar codes you see on grocery store items, but can hold a higher density of varied information, and don't require specialized hardware—any camera will do. They can be small, and withstand many kinds of degradation and smudges, which helped fuel their popularity in other fields. And because QR codes can work with an image, they've become a common way to transmit information from print to mobile devices.

Let's round out this chapter with one more immersion, using one more important Glass aspect: video. Unlike the previous two we've written this chapter, this application will manage its own views rather than closing immediately or pawning off its work to another activity. But first, let's think about how we want our app to function.

A Video Immersion

When launched, our app is going to be a single screen displaying a video. Our users will point their camera toward a QR code, which will show up in the video screen. To help the user know what to do, the app will have a rectangle overlay with text telling them to point at a QR code, like in Figure 47,*A Scan QR Codes Using the Camera and Screen*, on page 206.

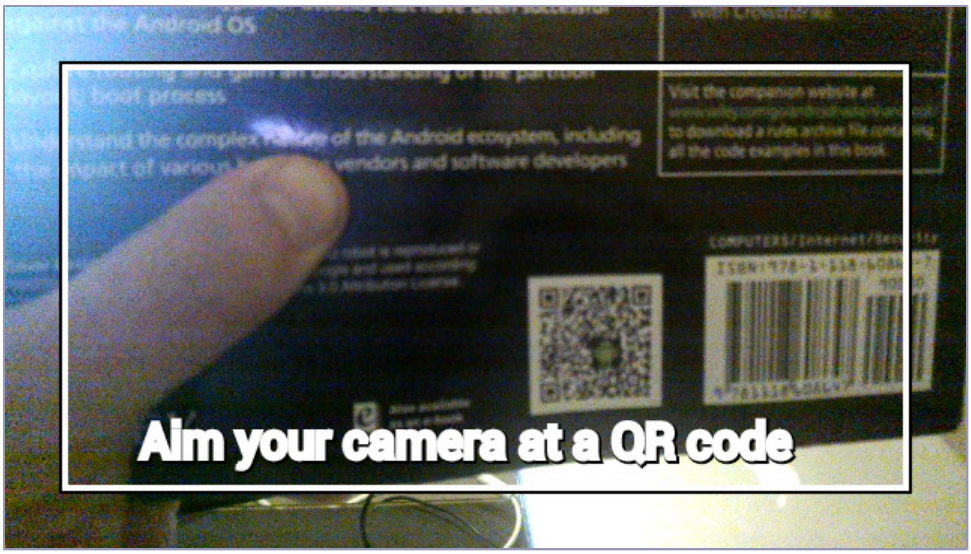


Figure 47—A Scan QR Codes Using the Camera and Screen

As soon as the app detects a QR code on the screen, it will launch a related intent, depending on the data stored in the code. If the data is a URL (starting with http:// or https://) it will launch a web browser. If the QR code is a latitude/longitude (a pair of floating point numbers separated by a comma) it will launch a Glass navigation Activity to the given geolocation. If the code contains any other text, it will display the string as a Card.

We also want to add a menu to our application so the user can choose to close the immersion. This isn't strictly necessary, since swiping down will close the immersion anyway, but adding a stop menu option is good UX when creating immersions.

To start, we create our application like the others, with a voice trigger (*get QR code*) and an activity with android:immersive="true" set in the app manifest. Let's name the activity QRCameraActivity, and jump right into the onCreate method.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraActivity.java

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    audioManager = (AudioManager)getSystemService(Context.AUDIO_SERVICE);
    cameraView = new QRCameraView( this );
    overlayView = new OverlayView( this );
    layout = new FrameLayout( this );
    layout.setKeepScreenOn( true );
    layout.addView( cameraView );  // order is important here
    layout.addView( overlayView ); // the last view is on top
```

```
    setContentView( layout );
}
```

We get the system's AudioManager object, so we can play a click sound when the app menu is opened, which we'll deal with later. The interesting part of this method is how we create the immersion view.

In regular Android development, a FrameLayout is a useful view group that fills up the entire screen. Since the class extends ViewGroup, it's a simple way to layer views. In our case, we want to create our own QRCameraView that will display a preview of the Glass camera's input. On top of that, we want to generate an OverlayView to help properly direct the user. As a nicety to our users, we also call setKeepScreenOn(true), which will stop the Glass from timing out as long as the frame layout is active.

Let's take a look at the implementation of the two views, starting with QRCameraView, then we'll quickly cover the OverlayView.

Camera View

The QRCameraView is the heart and soul of our application, and pulls triple duty.

First, we extend a type of view called a SurfaceView. A SurfaceView is the same sort of View implementation that Glass uses to render a LiveCard in high frequency mode, but we're just creating one outright. A SurfaceView gives access to its underlying drawing surface via a SurfaceHolder, an object we talked about in Chapter 12, *Advanced Rendering and Navigation*, on page 167. This simply provides the surface that our camera will draw upon.

The second duty our view performs is to display a camera preview based on the state of the underlying drawing surface, which we attach by calling getHolder().addCallback(this). Like our example in the last chapter we add a callback to the holder in order to know the state changes of the surface. We could create a new class to do this, but it's much easier and common practice to have the SurfaceView also implement DirectRenderingCallback. When the surface changes or is unpaused, we need to open up a camera, and start a preview. When the view is destroyed or paused we need to release the camera.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraView.java

```
public class QRCameraView
  extends SurfaceView
  implements DirectRenderingCallback
{
  public final static String TAG = QRCameraView.class.getName();
  public static final int    FPS = 30;

  private Camera camera;
```

```
public QRCameraView( QRCameraActivity context ) {
  super(context);
  getHolder().addCallback(this);
}
```

Finally, our view will setup the code that scans a camera frame for a QR code, by using the ZXing library.[4] Like the AWS library we added, you can get the ZXing core jar by adding this to your build.gradle (Module: app) file.

```
compile 'com.google.zxing:core:2.3.+'
```

The details of this scanning action is in a camera callback implementation called QRPreviewScan, that we'll dig into soon. But first, let's look at what our DirectRenderingCallback does.

Direct Rendering Callback

Our implementation of the DirectRenderingCallback interface is, from a high level view, rather simple. Recall from Chapter 12, *Advanced Rendering and Navigation*, on page 167 that we must implement surfaceCreated, surfaceChanged, surfaceDestroyed, and renderingPaused. We don't need surfaceCreated this time around, so let's focus on the remaining methods.

When the surface changes or returns from a pause, we need to open the Android camera object and start the preview. The SurfaceHolder we pass into openCamera will associate this view's holder with the preview display, so that the camera preview will draw on our QRCameraView's surface. Our view's holder object is passed into DirectRenderingCallback methods, because we asked it to be in the constructor by way of getHolder().addCallback(this).

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraView.java

```
public void surfaceCreated(SurfaceHolder holder) {}

public void surfaceChanged(SurfaceHolder holder, int format,
                           int width, int height) {
  this.camera = openCamera(holder);
  this.camera.startPreview();
}

public void surfaceDestroyed(SurfaceHolder holder) {
  releaseCamera();
}

public void renderingPaused(SurfaceHolder holder, boolean paused) {
  if (paused) {
    releaseCamera();
```

4. https://github.com/zxing/zxing

```
  } else {
    if (holder.getSurface().isValid()) {
      this.camera = openCamera(holder);
      this.camera.startPreview();
    }
  }
}
```

When the surface is paused or destroyed, stop the camera preview and release the camera resource. Let's take a look at the details of opening and releasing the Glass camera.

Opening and Releasing the Camera

Glass has only one camera, but like other Android devices, can run multiple applications at once. In order to share this single camera resource across multiple processes, an application has to *check out* the camera by calling the Android static method Camera.open(). If the camera isn't currently being used, you'll get a camera object that represents the hardware. With the camera object, we set some parameters we need, such as refresh rate and preview size in pixels.

Two settings that are very important for our application to function are setPreviewDisplay and setPreviewCallback. The first is where we pass in the QRCameraViews SurfaceHolder object, to ensure that our camera preview will display on this view's surface. This is the crux of our view's visual functionality. The second method is how we retrieve the contents of each frame of the preview. This callback object must implement the Camera.PreviewCallback class onPreviewFrame method. This method will be called by the camera preview mechanism once for each frame of the incoming video stream. We'll take a look at the implementation of this callback in the next section, after we look at releasing the camera.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraView.java

```
private Camera openCamera(SurfaceHolder holder) {
  if (this.camera != null)
    return this.camera;
  Camera camera = Camera.open();
  try {
    // Glass camera patch
    Parameters params = camera.getParameters();
    params.setPreviewFpsRange(FPS * 1000, FPS * 1000);
    final DisplayMetrics dm = getContext().getResources().getDisplayMetrics();
    params.setPreviewSize(dm.widthPixels, dm.heightPixels); // 640, 360
    camera.setParameters(params);
    camera.setPreviewDisplay(holder);
    QRCameraActivity activity = (QRCameraActivity)getContext();
    camera.setPreviewCallback(
```

```
      new QRPreviewScan(activity, dm.widthPixels, dm.heightPixels));
  } catch (IOException e) {
    camera.release();
    camera = null;
    Toast.makeText(getContext(), e.getMessage(), Toast.LENGTH_LONG).show();
    e.printStackTrace();
  }
  return camera;
}
```

The flip side of using a single shared system resource is that you *must release the camera*. If your application fails to release the camera, your app will continue holding onto the resource, tying it up when other applications wish to use it.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraView.java

```
private synchronized void releaseCamera() {
  if (this.camera != null) {
    try {
      this.camera.setPreviewDisplay(null);
    } catch (IOException e) {}
    this.camera.setPreviewCallback(null);
    this.camera.stopPreview();
    this.camera.release();
    this.camera = null;
  }
}
```

All we need to do to release the camera is wipe out the callbacks, stop the preview, and call release() on the camera object.

There's one more case when we must release the camera. Whenever Glassware opens the camera object, it must be ready to release that object whenever the small black camera button is clicked on the Glass device. If we don't take care to release the camera, the build-in *take a photo* action will be unable to secure the object, and crash. This violates one of the Glass design principles: *Avoid the Unexpected*. Users should always safely expect that clicking the camera button will function as normal.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraActivity.java

```
public boolean onKeyDown(int keyCode, KeyEvent event) {
    if (keyCode == KeyEvent.KEYCODE_CAMERA) {
        // Release the camera by pausing the cameraView
        cameraView.renderingPaused(cameraView.getHolder(), true);
        return false;  // propgate this click onward
    }
    else {
        return super.onKeyDown(keyCode, event);
    }
```

```
}

protected void onResume() {
    super.onResume();
    // Re-acquire the camera and start the preview.
    cameraView.renderingPaused(cameraView.getHolder(), false);
}
```

All we have to do is override the Activity class's onKeyDown method, and capture a KeyEvent.KEYCODE_CAMERA keyCode. When the button is pressed, we first call cameraView.renderingPaused with true to release the camera object, before returning false, which allows the click to propagate onward to the built-in Glass photo application.

When the button-triggered camera action is completed, Android will give control back to our immersion, and call onResume. All we have to do is call cameraView.renderingPaused with false to unpause the view and reacquire the camera instance.

Scanning for QR Codes

Thus far, we've created an Activity that hosts a SurfaceView, which uses the Glass Camera and displays a preview of what it sees. And to be a good Glassware citizen, it relinquishes its hold on the camera if the application is ever paused, or the camera button is pressed. We've heretofore made a nice app for displaying live video from the camera, but it doesn't scan for QR codes yet. Nor does our app yet launch any intents based on the QR code content, like a browser for URLs.

When we opened our camera object, we assigned a class called QRPreviewScan to setPreviewCallback. This class implements Camera.PreviewCallback, which calls onPreviewFrame with an array of bytes that represent a specific frame of the video. From there, we call a method called scan that will do the real heavy lifting with the frame data.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRPreviewScan.java

```
public class QRPreviewScan implements Camera.PreviewCallback {
  public void onPreviewFrame(byte[] data, Camera camera) {
    // Only scan every 10th frame
    if( ++framesSinceLastScan % (QRCameraView.FPS / SCANS_PER_SEC) == 0 ) {
      scan(data, widthPixels, heightPixels);
      framesSinceLastScan = 0;
    }
  }
```

The scan function is going to perform a lot of heavy work. In order to reduce the workload on our system, we track the number of frames that have passed

since we last scanned, and only call scan three times a second, or every 10th frame. There's nothing special about this number, it's just a way of reducing the amount of resources our application will have to perform, while still remaining somewhat realtime.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRPreviewScan.java

```
private void scan(byte[] data, int width, int height) {
  PlanarYUVLuminanceSource luminanceSource = new PlanarYUVLuminanceSource(data,
      width, height, 0, 0, width, height, false);
  BinaryBitmap bitmap = new BinaryBitmap(new HybridBinarizer(luminanceSource));
  Result result = null;
  try {
    result = multiFormatReader.decodeWithState(bitmap);
  } catch (ReaderException re) { // nothing found to decode
  } finally {
    multiFormatReader.reset();
  }
  if (result != null) {
    Intent intent = new QRIntentBuilder(result.getText()).buildIntent();
    activity.launchIntent(intent);
  }
}
```

With our scan data in hand, we convert the YUV (an old video color space)[5] encoded byte array to a BinaryBitmap image object from the ZXing library, then use the library's MultiFormatReader to scan the image for any results. If a result is not found, we continue on to the next camera preview frame. However, if a result is found, we extract the encoded text—we use a helper class called QRIntentBuilder to decide what the text represents, which generates a relevant intent.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRIntentBuilder.java

```
public Intent buildIntent() {
  switch(getType()) {
  case URL:
    return new Intent(Intent.ACTION_VIEW, Uri.parse(qrText));
  case LOCATION:
    return new Intent(Intent.ACTION_VIEW, buildNavUri());
  default:
    return new Intent().putExtra("text", qrText);
  }
}

private Uri buildNavUri() {
  return Uri.parse("google.navigation:ll=" + qrText + "&mode=mru");
}
```

5. http://en.wikipedia.org/wiki/YUV

It finally send that intent back to the QRCameraActivity for further processing.

The QR Code is Buggy!

I've purposefully placed a performance bug in this QR code reading app. Although everything technically functions, we're placing a very heavy decoding method in the main UI thread, which is not the most efficient thing to do. In chapter 13, we're going to track this bug down and fix the problem. But if you can't wait that long, the solution is commented out in this chapter's downloadable code.

Launching Intents based on QR Data

After a QR code has been scanned, and the launchIntent method has been called, we now have to decide what to do with our new found intent. After we pause the camera view, most of the time we'll just pass the intent on to startActivityForResult and let the Android OS decide what to do with an http or google.navigation URL. In the case where we only have plain text, we want to create a Card view and display the contents of the QR code to the user.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraActivity.java

```
public synchronized void launchIntent(Intent intent) {
    cameraView.renderingPaused(cameraView.getHolder(), true);
    if (intent.hasExtra("text")) {
        // no intent to launch, just show the text
        CardBuilder card = new CardBuilder(this, Layout.TEXT)
                .setText(intent.getStringExtra("text"));
        layout.removeAllViews();
        layout.addView( card.getView() );
    } else {
        startActivityForResult(intent, REQUEST_CODE);
    }
}

public void onActivityResult(int requestCode, int resultCode, Intent intent) {
    switch (requestCode) {
        case REQUEST_CODE:
            finish();
            break;
        default:
            break;
    }
}
```

If the web browser or navigation apps are launched, they'll continue running until the user closes them. When they're finished, our onActivityResult implementation will just finish this immersion.

Overlay View

There's one more view we have to create to give our users the best experience. Since our immersion is intended to scan for QR codes, we should inform the user how to go about using it. Simply displaying a realtime video feed from the Glass camera by itself isn't a very good app design. We need to make it clear that this is a QR code reading app, and not just any old video screen. We do this by drawing a view over the top of the live video view.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraActivity.java

```
static class OverlayView extends View {
    private Paint paint;
    private String overlay;
    public OverlayView(Context context) {
        super(context);
        this.paint = new Paint( Paint.ANTI_ALIAS_FLAG );
        this.overlay = this.getResources().getString(R.string.qr_camera_overlay);
    }
    public void draw( Canvas canvas ) {
        // draw drop shadows
        paint.setColor( Color.BLACK );
        drawBox( canvas, 1 );
        drawText( canvas, 1 );
        // draw box and text
        paint.setColor( Color.WHITE );
        drawBox( canvas, 0 );
        drawText( canvas, 0 );
    }
    private void drawBox( Canvas canvas, int offset ) {
        paint.setStrokeWidth( 6 );
        paint.setStyle( Style.STROKE );
        canvas.drawRect( 40-offset, 40-offset, 600+offset, 320+offset, paint );
    }
    private void drawText( Canvas canvas, int offset ) {
        // paint.setTypeface( Typeface.create("Roboto", 0) );
        paint.setTextSize( 32 );
        canvas.drawText( this.overlay, 90+offset, 300+offset, paint );
    }
}
```

Drawing onto a view is a pretty basic Android operation. We needn't use any fancy specializations, so simply extending `View` makes a sufficient transparent drawing surface. Anything beneath this view that we don't specifically draw upon will still be visible. We only need to implement the `draw` method, and draw a rectangle and text on the given Canvas. We draw twice, once offset black and another white, to create a shadow effect.

We won't go into any details about canvas draw functions, as there are plenty of Android books and online resources[6] where you can learn to draw all sorts of shapes and images.

Immersion Menu

Technically, our application works just fine. We could stop here, and the user could launch the app, scan for qr codes, or swipe down to exit the immersion. However, it's often good Glassware design to supply a menu when a user taps on a card, whether it's a static Mirror API card, a Live Card, or an immersion. Happily, creating a menu inside an immersion shares some similarities with a LiveCard menu, but is actually simpler to implement.

All you have to do is add this code to your `onKeyDown` method. A Glass gesture bar *tap* is actually registered in Android as a `KeyEvent.KEYCODE_DPAD_CENTER` keycode. When that code is used, call `openOptionsMenu`, and add a click sound effect for good measure.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraActivity.java

```
else if (keyCode == KeyEvent.KEYCODE_DPAD_CENTER) {
    audioManager.playSoundEffect(AudioManager.FX_KEY_CLICK);
    openOptionsMenu();
    return true;
}
```

From there, it's nothing we haven't done a few times before. Override the standard menu methods directly in your immersion's `QRCameraActivity`, and call it a day.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRCameraActivity.java

```
public boolean onCreateOptionsMenu(Menu menu) {
    MenuInflater inflater = getMenuInflater();
    inflater.inflate(R.menu.qr, menu);
    return true;
}

public boolean onOptionsItemSelected(MenuItem item) {
    switch (item.getItemId()) {
        case R.id.stop:
            finish();
            return true;
        default:
            return super.onOptionsItemSelected(item);
    }
}
```

6. http://developer.android.com/guide/topics/graphics/2d-graphics.html

Now, whenever the user taps on their Glass gesture bar when the immersion is running, they can choose to stop it from the menu list. One last tip: don't forget to add the menu XML resource.

With our QR Code immersion complete, go ahead and test that it works by scanning any of the following codes.

Address of Voodoo Donuts in Portland, OR

URL of the Pragmatic Bookshelf website

An important secret message

Wrap-Up

This chapter covered immersions, the last UI element for Glass alongside static and live cards. Where the other two exist within the Glass timeline, immersions live their lifecycle outside the timeline after they are launched like other Glassware, in the Glass app menu.

Immersions are technically simpler than LiveCards, when looked at from the perspective of number of moving parts. LiveCard applications require a background service to launch and manage the LiveCard, an activity to manage the required menu launched by pending intents, and a remote view parcel or manually drawing via surface holder callback. Immersions are a mere Android Activity, which as we saw in the Nevermind App, doesn't have to do much of anything.

If immersions are simple, why didn't we cover them first? Because they aren't easier. The complexity of immersions lie in the fact that they are free roaming Android Activities, with all of the power and responsibilities that implies. They don't live in the nice warm cocoon of the Timeline, and thus require more explicit design.

You can convert an old Android applications into immersions without much change, but it may not make sense as a Glassware. Although much of the Android ecosystem is at your disposal, you may have a hard time finding a good home for built-in resources, such as seek bar wigits, or qwerty keyboard

intents. Immersions allow your applications the ability to take over a user's Glass environment, but you have to carefully design what you should do.

CHAPTER 14

Creating an Immersive Game

Video games can be excellent tests of a computer system's abilities. They can push the limits of CPU/GPU, memory, display, audio, inputs, and in some cases of networked games, bandwidth. In the previous chapter, we dipped our toes into Glass immersions. But now it's time to dive in, headfirst, and see just how deep an immersive experience can go by creating a Google Glass video game. We already have many of the tools necessary to make any form of Glassware, so we'll create a side-scrolling adventure game called Miss Pitts. This is a jumping-style game inspired by the endlessly annoying the Flappy Bird genre.

In this chapter we're constructing this game, designed as an immersion activity running outside the timeline, with a simple splash screen that shows the game's high score. From there we'll design how the user interacts with the game by using the gesture bar and 9-axis sensor inputs, followed by rendering graphics, sound effects and music. Finally, we'll fill in the rest of the game with the objects and game engine logic required to make the whole thing go.

A Note On Code in this Chapter

There's a lot of code in this chapter. Throughout this book we've forgone printing certain lines of Java code in the interest of saving space. For example, we always skip `package` and `import` statements. This game constitutes even more code, far too much to print out. So we'll be skipping most obvious class field definitions, trivial constructors, and required `try` statements with empty `catches`. More than with any chapter before, I'd suggest you download the code and follow along.

Start The Game with a Splash Screen

The start of any good video game design is explaining to your users why they're here, and what they're trying to achieve. A backstory, no matter how basic, is a good launch point. Here's our story:

Our heroine, Miss Pitts, lives in a post-apocalyptic two dimensional world, where huge sinkholes have appeared in the only street that actually exists. If she falls down a pit it's game over. Luckily, she can run to the right, jumping at varying heights, allowing her to clear differently sized pits. Let's keep the scoring and game play simple: You receive one point per pit jumped.

It's a good idea to sketch out our game before we start coding. Happily our game is pretty easy, and can be described in a single screen, like so:

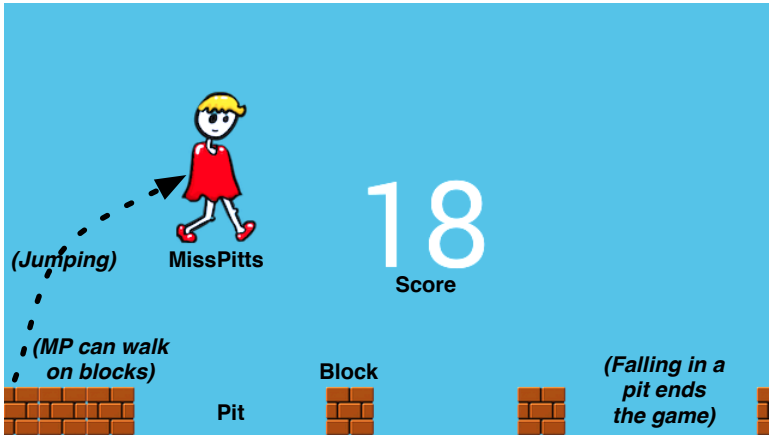


Figure 48—Designing MissPitts in One Screen

Since this is a standard immersion, we need to start with an activity which we'll just give the deeply clever name GameActivity. Don't forget to set android:immersive="true" in the Android manifest.

Voice Commands

There's one minor deviation from how we've created voice triggers so far in this book. In MissPitts, We're going to use the built-in voice command PLAY _A_GAME. In our voice_trigger.xml, rather than giving the trigger a keyword, we're instead giving it the built-in command.

chapter-14/MissPitts/app/src/main/res/xml/voice_trigger.xml

```
<?xml version="1.0" encoding="utf-8"?>
<trigger command="PLAY_A_GAME" />
```

You can get the entire list of commands from the VoiceTriggers.Command class.[1] Here are a few examples:

ADD_AN_EVENT	CHECK_THIS_OUT	EXPLORE_THE_STARS
CALCULATE	CONTROL_MY_CAR	FIND_A_BIKE
CALL_ME_A_CAR	CONTROL_MY_HOME	FIND_A_DENTIST
CAPTURE_A_PANORAMA	CREATE_A_3D_MODEL	FIND_A_DOCTOR
CHECK_ME_IN	EXPLORE_NEARBY	FIND_A_FLIGHT

...and so on.

These don't require that you have the com.google.android.glass.permission.DEVELOPMENT permission that we've always set in the manifest up to now. Only these commands are officially sanctioned by Google, so if you have plans of someday deploying your application in the Google play store, you'll have to use them. The benefit of using these commands is that they are automatically internationalized into all languages supported by Glass.

An Activity with a Splash Screen

Our GameActivity is not too different from other activities we've written. There's an onCreate method that populates all of the fields that our app will use. We'll talk about UserInputManager, frameBuffer, GameEngine and RenderView in due course. For now, let's focus on the loadingCardView() method and the Handler object.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameActivity.java

```
public class GameActivity extends Activity {
  public void onCreate( Bundle savedInstanceState ) {
    super.onCreate( savedInstanceState );
    setContentView( loadingCardView() );
    inputManager = new UserInputManager( this );
    final Bitmap frameBuffer = Bitmap.createBitmap( 640, 360, Config.RGB_565 );
    engine = new GameEngine( this, inputManager, frameBuffer );
    renderView = new RenderView( GameActivity.this, engine, frameBuffer );
    // After 3 seconds, switch the content view to the rendered game
    Handler handler = new Handler();
    handler.postDelayed(new Runnable() {
      @Override
      public void run() {
        setContentView( renderView );
      }
    }, 3000);
  }
```

1. https://developers.google.com/glass/develop/gdk/reference/com/google/android/glass/app/VoiceTriggers.Command

First, let's look at the loadingCardView. We want to create a splash card that displays the name of the game (and the high-score, which we'll cover later) for a few seconds before starting the game. This allows the player a few seconds to get into the right frame of mind before starting the game, and it's also a good way to convey other information, like the high score, or maybe how to control the game.

Our splash screen will look something like this.

We're using a reference drawable called loading.png as a full background image, and overwriting the last high-score over top of it. We use the Card class as a convenient way to build a card view, but we could have just as easily made a layout with an ImageView and some text.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameActivity.java

```
private View loadingCardView() {
  Card loadingCard = new Card( this );
  loadingCard.setImageLayout( ImageLayout.FULL );
  loadingCard.addImage( R.drawable.loading );
  loadingCard.setText(
      getResources().getString(R.string.high_score) + getSavedScores() );
  View cardView = loadingCard.getView();
  cardView.setKeepScreenOn( true );
  return cardView;
}
```

Before returning the view generated by the card, we set setKeepScreenOn(true) for good measure. This keeps the screen from dimming/pausing within the three seconds it takes for the game to start. We set this view at the start of onCreate.

If you look closely, the code calls setContentView twice—once immediately, and again inside of an anonymous Runnable class.

A Handler in Android manages the processing of messages in a queue. This queue can either act as a scheduler to run some code at a future date, or it can be a way to offload work to a different thread. We're using it in the first way, since we didn't pass in another thread's Looper.[2]

This is a low-tech way of creating a timed splash card. Our application is still loading in the background, and for three seconds the user will only see the view constructed by the loadingCard method. Once the three seconds have passed, however, renderView becomes the new content view, overwriting the splash card.

With our basic application shell and splash screen underway, let's dig into some of those fields we skipped, starting with UserInputManager which, as you may have guessed from the name, manages user inputs.

Gesture and Sensor Inputs

A game drawing itself and playing music without user interaction is, frankly, just a cartoon. A game must provide an interface so a player can interact with the environment in some way. Moreover, this interface must be easy and natural. Otherwise, no matter how good your game's story or graphics may be, discomfort will stop anyone from playing it. So let's investigate our user control options, with an eye on playability.

For simplicity, we'll handle all player inputs through a class called UserInputManager. This class is in two parts—the first handles inputs that cause Miss Pitts to jump, the other handles making her walk, run, or stop.

One of the cooler features of Glass is its variety of inputs. We must forego the kind of hand-held controller you get in game consoles, or the variety of keyboards/joysticks on a PC. Let's consider the two character movements in our game: jumping and walking.

Tap Gesture

We'll start with making our character jump. You could interface with a variety of inputs, each with their own strengths and weaknesses. Voice is always an option, simply saying out loud the word "jump." It's hands free, but also, limits where you play the game. You can play in the privacy of your own home, or risk annoying people by talking aloud to yourself in a public bus. Voice also always has a lag, and reaction time between talking and an action happening is slower than other muscle movements, like twitching a finger.

2. http://developer.android.com/reference/android/os/Looper.html

Another option is to take advantage of the Glass sensors, where a player could flick his or her head to cause Miss Pitts to jump. That would also be hands-free and silent, but asking users to whip their head up repeatedly is a recipe for a hurt neck.

Instead, let's have the player tap the gesture bar to trigger a jump. We can actually track how many fingers are used to tap, so let's take advantage of that fact where one finger represents a short jump, two finger taps for higher jumps, and three for the highest.

We'll send gesture events onto our UserInputManager that implements GestureDetector.BaseListener, which requires this class to override onGesture(Gesture). We'll go over the SensorEventListener methods later.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/UserInputManager.java

```
public class UserInputManager
  implements GestureDetector.BaseListener, SensorEventListener
{
  enum GameEvent {
    JUMP, JUMP_HIGHER, JUMP_HIGHEST,
    WALK, RUN, STOP
  }
  public UserInputManager( Context context ) {
    this.context = context;
    events = new ArrayList<GameEvent>(4);
    eventsBuffer = new ArrayList<GameEvent>(4);
    gestureDetector = new GestureDetector( context ).setBaseListener( this );
  }
  public synchronized List<GameEvent> getEvents() {
    events.clear();
    events.addAll( eventsBuffer );
    eventsBuffer.clear();
    return events;
  }
  public boolean dispatchGenericFocusedEvent( MotionEvent event ) {
    return gestureDetector != null && gestureDetector.onMotionEvent(event);
  }
  @Override
  public boolean onGesture( Gesture gesture ) {
    switch( gesture ) {
    case TAP:
      eventsBuffer.add( GameEvent.JUMP );         return true;
    case TWO_TAP:
      eventsBuffer.add( GameEvent.JUMP_HIGHER );  return true;
    case THREE_TAP:
      eventsBuffer.add( GameEvent.JUMP_HIGHEST ); return true;
    default:
      return false;
    }
```

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

Some of this might look familiar, like creating a new GestureDetector and setting this as it's BaseListener. There's also the dispatchGenericFocusedEvent method. Both of these are similar code to Chapter 12, *Advanced Rendering and Navigation*, on page 167, where we captured gestures via BalloonCountActivity, and shared those events with BalloonCountScrollView via dispatchGenericFocusedEvent.

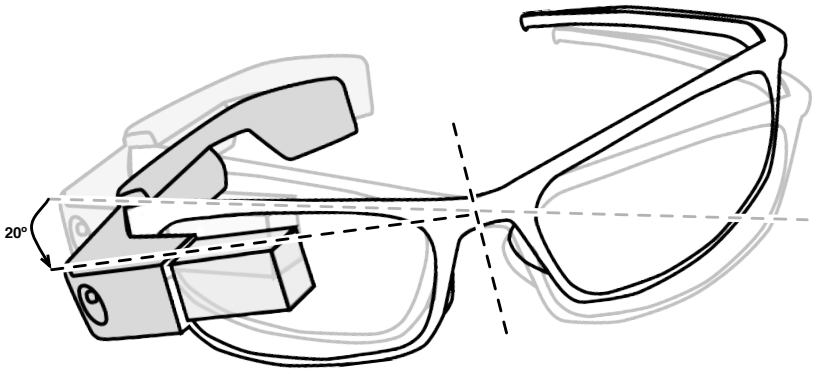
The difference here is that we capture gestures, and rather than react immediately to them, we store corresponding game events in a buffer. So, for example, when a player makes a TWO_TAP Gesture (meaning, with two fingers), we save that as a JUMP_HIGHER game event.

Later, we'll see how the GameEngine periodically translates these events into game actions by calling getEvents. In this method, we synchronously copy the buffer into an unmodifiable list and flush the buffer. This allows us to continue accepting inputs from the player into the event buffer while the GameEngine iterates on new events.

Notice that our enum GameEvent has placeholders for all three jump heights. There are also values for walk, run, and stop. But how do we get those events?

The 9-Axis Events Sensor

Just like the jump option, we have a few input choices for walking. One natural choice might be to have the player actually walk, and each step makes the character walk in our game… but that's exhausting, and potentially dangerous if you're walking toward a street. Another option is voice, but it has the downsides we just discussed. We could use swipe gestures, but that wouldn't be very interactive—swipe down to walk, up to stop? No fun. How about if the player tilts his or her head to the right, as shown in the following figure, Miss Pitts will walk in that direction, and if they tilt their head even further, she runs?



Glass contains what's called a 9-axis sensor. The *9* in 9-axis comes from adding three sensing elements that each operate in three dimensions: 3-axis accelerometer, 3-axis gyrometer (gyroscope), and 3-axis magnetometer (compass). This means that every sensor knows the acceleration, orientation, and magnetic north of the device.

If everything is working properly, you should be able to measure exactly how Glass moves in three dimensional space. With these sensors, along with the geolocation service (Context.LOCATION_SERVICE), you can detect a glass device's position and movement on Earth.

You can access these values directly from Android by registering a SensorEventListener object on the proper sensor, which you get from the sensor manager via getDefaultSensor. You can access the values of the three 3-axis hardware sensors by getDefaultSensor on one of the three Sensor class constants.

- TYPE_ACCELEROMETER (accelerometer)
- TYPE_GYROSCOPE (gyrometer)
- TYPE_MAGNETIC_FIELD (magnetometer)

We get the sensor service from context.getSystemService(Context.SENSOR_SERVICE).

chapter-14/MissPitts/app/src/main/java/glass/misspitts/UserInputManager.java

```
public void registerSensor() {
  // register gravity sensor for player movement
  sensorManager = (SensorManager)context.getSystemService( Context.SENSOR_SERVICE );
  sensorManager.registerListener(this,
      sensorManager.getDefaultSensor( Sensor.TYPE_GRAVITY ),
      SensorManager.SENSOR_DELAY_GAME);
}
public void unregisterSensor() {
  sensorManager.unregisterListener( this );
}
```

When values or accuracy change, the sensor service will call the registered SensorEventListener's onSensorChanged or onAccuracyChanged.

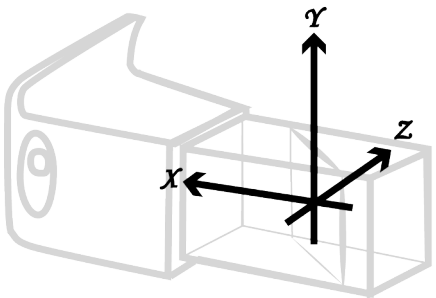
If you noticed, we didn't call any of the three listed sensors. We registered our object to Sensor.TYPE_GRAVITY. An accelerometer reports the acceleration of the Glass device, but if we recall Einstein in our college physics class, acceleration and gravity are indistinguishable. An accelerometer measurement can't help but include the force of gravity's (g) pull on the device ($g = 9.81 m/s^2$). Glass performs some physics calculations against the accelerometer's value in order to isolate the force of gravity. There's no specific gravity sensor; Sensor.TYPE_GRAVITY is known as a virtual sensor or synthetic sensor.[3]

Glass supports[4] the following virtual sensors. These use a combination of hardware and software to act as sensors in their own right.

- TYPE_GRAVITY (gravity)
- TYPE_LINEAR_ACCELERATION (acceleration with gravity removed)
- TYPE_ROTATION_VECTOR (device rotation from a fixed point of reference)

In addition to the 9-axis and virtual sensors, Glass has a seventh sensor, named TYPE_LIGHT. It uses Glass's photometer hardware to determine the illumination surrounding Glass. This is used to make the display brighter/dimmer relative to ambient light.

All of the motion sensors return values in x,y,z Cartesian coordinates, where x is the right-left axis, y is up-down, and z is backward-forward.



When the player tilts their head to the right 10-degrees the character walks, tilts 20-degrees she runs, and stops when the player holds their head level to the ground.

3. http://developer.android.com/guide/topics/sensors/sensors_overview.html
4. https://developers.google.com/glass/develop/gdk/location-sensors

With a bit of math we get the left-right tilt of the Glass device from it's face up resting position. If you don't follow why the math works, that's ok, we'll end the chapter with a simpler option. For those of you who enjoy this kind of thing, we need to convert from Cartesian to spherical coordinates, specifically, we're looking for the polar θ (theta) angle.

We don't care all that much about the (y,z) plane, since that represents tiling the head forward and back. What we want is a left-right angle (x). So we first collapse the (y,z) plane into a plane with radius R, with the help of Pythagoras ($y^2 + z^2 = R^2$). Then we only have a simple polar conversion to get theta (arctan(x / R)) for a single value that represents a left-right tilt angle in radians. Vertical is zero, and the more you tilt your head to the right, the closer the number gets to -π/2 radians (if you could keep turning your head completely upside-down, you'd reach π radians, or 180º).

Then it's a simple matter of converting from radians to degrees. This isn't strictly necessary, but makes the values easier to reason about if you're used to dealing in degrees. Combining these steps yields:

$$\tan^{-1}\frac{x}{\sqrt{y^2+z^2}} \times \frac{180}{\pi} = \theta^o$$

We can simplify the conversion from radians to degrees by using the built-in Math.toDegrees() method. Then for each range of degrees we choose to add a STOP, RUN or WALK event.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/UserInputManager.java

```
public void onAccuracyChanged(Sensor sensor, int accuracy) {
  // nothing to do here
}
public void onSensorChanged( SensorEvent event ) {
  if( event.sensor.getType() == Sensor.TYPE_GRAVITY ) {
    // convert Cartesian system to the polar value in spherical coords
    float x = event.values[0], y = event.values[1], z = event.values[2];
    // theta is the tilt of the Glass from a flat resting position 0 in radians
    // then convert to negative degrees for simplicity
    double thetaDegrees = -Math.toDegrees(
        Math.atan(x / Math.sqrt(Math.pow(y,2) + Math.pow(z,2))));
    if( thetaDegrees < 10 ) {       addEventChange( GameEvent.STOP ); }
    else if( thetaDegrees >= 20 ) { addEventChange( GameEvent.RUN );  }
    else if( thetaDegrees >= 10 ) { addEventChange( GameEvent.WALK ); }
  }
}
private void addEventChange( GameEvent event ) {
  if( event != lastMovement ) {
    lastMovement = event;
    eventsBuffer.add( event );
```

```
  }
}
```

If the player's head is tilted greater than 20-degrees we signal that the player wants Miss Pitts to run. With a tilt of greater than 10-degrees she walks, and anything less than 10-degrees she stops. To ensure that we don't fill up our event buffer, we only add an event to the buffer if it's different from the last time we measure.

Integrating InputManager with GameActivity

So we've created an input manager, but how do we get the user's gestures to trigger the UserInputManager's dispatchGenericFocusedEvent? We did this before on page 182, by overriding an activity's onGenericMotionEvent(event) method.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameActivity.java

```
public boolean onGenericMotionEvent( MotionEvent event ) {
  // Send generic motion events to InputManager's GestureDetector
  return inputManager.dispatchGenericFocusedEvent( event );
}
```

To ensure we don't expend unnecessary energy on a paused activity, we'll call the UserInputManagers register method in onResume, and unregister in onPause.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameActivity.java

```
public void onResume() {
  super.onResume();
  inputManager.registerSensor();
  renderView.setKeepScreenOn( true );
  renderView.resume();
}
public void onPause() {
  super.onPause();
  inputManager.unregisterSensor();
  renderView.setKeepScreenOn( false );
  renderView.pause();
  if( isFinishing() ) {
    saveScores();
    // engine.releaseAssets();
  }
}
```

Notice that renderView is also called in these methods? We're about to learn why in the next section.

Rendering Visuals and Playing Audio

We now have an Activity that accepts inputs. But without visuals, and some audio to add depth, our game won't be all that interesting... no application

is. In this section we're going to craft the underlying structures necessary to draw on and play audio, although the actual job of drawing and playing audio will fall to the *Game Logic* section.

The Render View

The Glass viewport is projected as a floating, translucent, 25-inch screen at a distance of 8 feet. But the differences in a Glass display compared to the kind we're used to (television, monitor, hand-held) we have to keep a few points in mind:

First, the Glass display is transparent. You're always wrestling with the world behind the screen in any Glass app, even more so if you have fine details, or low contrast in your application. To deal with this shortcoming, we'll craft our objects with high contrast colors and shapes.

Second, the display is a lowly 640 x 360 resolution, which isn't much better than a Super Nintendo from 1990, so you can give up on the idea of making a hyper realistic environment. We're better off with cartoony or blockish figures.

Third, since the display is close-up and monoptic, it can cause the player some strain to stare at the screen for any extended time. So we should make the gameplay short—in our case, we'll just make the game really difficult.

Also, we have to accept that some players of our game will be outside their home. We don't want to make the game distracting—it should always be easy to reconnect with the real world. In our case, we'll just forgo drawing any unnecessary images, keeping the background plain and distraction-free.

If we remember the components of Glass applications that we've seen before, we have many of the tools required to implement this game. Similar to how we rendered smooth animations in the Partify `LiveCardRenderer.RenderThread` class in Chapter 12, *Advanced Rendering and Navigation*, on page 167, we'll draw directly on the surface in a class called `RenderView`.

Our class extends `SurfaceView`, a view with access to a `SurfaceHolder` object.

48 fps is two times faster than our Partify app's surface rendering. 48 is a good framerate for a side-scrolling game on Glass. Standard videogame fps are 60, but low-resolution side-scrollers don't require quite so high a frame count, and it also has the benefit of relaxing 20% of the rendering strain on the CPU/GPU.

With a goal of render 48 fps, each frame must be render within about 21ms. If it routinely takes longer to render a scene, say 35ms, our game will run too slowly, making our game look like some sort of child's flip-book animation.

Base rendering on time, not frame counts. Frame rate is variable, meaning you can never be certain what framerates will be on any system. On a low power system like Glass, long delays between frames are even more likely.

In order to assist the RenderView in it's task of drawing on the surface, we create a Bitmap of the exact dimensions of our Glass screen. It's named frameBuffer, and acts as buffer that we can populate in the main application thread, while the buffer's previous contents are drawn to the RenderView's canvas. This object is created in the GameActivitys onCreate(), and passed into both the renderView, as well as the GameEngine object that we'll see in the next section.

Using a frame buffer that's separate from the buffer being drawn on the screen is an old game rendering technique called *double buffering*, where you draw on one buffer with the CPU, the other is painting to the screen with the GPU. It ensures that your application is always working the CPU/GPU to capacity, rather than overloading one while the other waits.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/RenderView.java

```
public class RenderView extends SurfaceView implements Runnable {
  public static final int FPS = 48;
  public static final int MSPF = 1000 / FPS;
  public void run() {
    Rect dest = new Rect();
    while( running ) {
      if( !holder.getSurface().isValid() ) continue;
      long startTime = System.currentTimeMillis();
      // update the game engine, paint the buffer, and draw it on the canvas
      engine.update();
      engine.paint();
      Canvas canvas = holder.lockCanvas();
      canvas.getClipBounds( dest );
      canvas.drawBitmap( frameBuffer, null, dest, null );
      holder.unlockCanvasAndPost( canvas );
      // fill in the difference between render time and frame timing.
      // this normalizes the framerate to about 48/sec
      int delta = MSPF - (int)(System.currentTimeMillis() - startTime);
      if( delta > 0 ) {
        try {
          Thread.sleep( delta );
        } catch (InterruptedException e) {}
      }
    }
  }
```

We're familiar with the basic draw loop from the LiveCardRenderer.RenderThread in Chapter 12, *Advanced Rendering and Navigation*, on page 167, but we've introduced a few differences. One is that we call update and paint in the GameEngine object. This is what's responsible for altering the game physics, then drawing on the current frameBuffer object. We'll see how that works soon enough. We open the canvas, flush the frameBuffer to it, and call to drawBitmap.

Finally, we compute the time difference between the start of the render action and when it completes, and sleep for that amount of time. This puts the brakes on our renderer. In order to get a smooth result, we only need about 48 frames per second for this game, or approximately one frame per 21 milliseconds. This pauses the thread for the amount of time left after the frame has been rendered to stay as close to that total 21 ms time. Without this pause, a future Glass with a very fast CPU/GPU might cause the game to run far too fast.

RenderView implements Runnable because we want this object to manage its own thread. The GameActivity onResume and onPause will call these methods, respectively. Just like the UserInputManager, the renderView thread doesn't need to run at all when the game isn't active.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/RenderView.java

```
public void resume() {
  running = true;
  renderThread = new Thread( this );
  renderThread.start();
}
public void pause() {
  running = false;
  renderThread.join();
}
```

Notice too that we hang setKeepScreenOn(true) onto the view when it's resumed. For the same reasons that we kept the loadingCard view on, we don't want to turn off a game screen just because a user hasn't made a jump in a few seconds.

A Theme Song

To add some depth to our game, we employ background music and sound effects. So we'll create a Music class that controls starting, stopping, and releasing the resource of a pleasant song in the background. Most of that code will wrap an Android API called MediaPlayer. We'll handle the sound effects differently by employing a rather straightforward Android service called SoundPool.

Including background music in a GDK application is not complicated. We employ the Android MediaPlayer object. It's a state-based object with a strict order of operations regarding initialization, preparation, pausing, playing, stopping and so on. There's detailed state diagram on the MediaPlayer API page,[5] and it's partly because of this complexity that we're wrapping it up into a Music class with simple play, stop, and release methods.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/Music.java

```
public class Music implements OnPreparedListener, OnErrorListener {
  public Music( Activity activity, String fileName ) {
    mediaPlayer = new MediaPlayer();
    isMusicPrepared = new AtomicBoolean();
    initialize( activity, fileName );
  }
  public void play() {
    if( !isMusicPrepared.get() ) {
      mediaPlayer.prepareAsync();
    } else {
      mediaPlayer.start();
    }
  }
  public void stop() {
    if( isMusicPrepared.compareAndSet(true,false)) {
      mediaPlayer.stop();
    }
  }
  public void release() {
    stop();
    mediaPlayer.release();
  }
  // ...
```

All of the real work is done within initialize. Here we load the music file as a file descriptor and pass it into the mediaPlayer object as a data source. Since we want the music to play continuously, we set looping to true. We also set this object as a callback, which implements OnPreparedListener.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/Music.java

```
private void initialize( Activity activity, String fileName ) {
  AssetFileDescriptor afd = null;
  afd = activity.getAssets().openFd( fileName );
  mediaPlayer.setDataSource(afd.getFileDescriptor(),
                            afd.getStartOffset(), afd.getLength());
  mediaPlayer.setLooping( true );
  mediaPlayer.setOnPreparedListener( this );
}
@Override
```

5. http://developer.android.com/reference/android/media/MediaPlayer.html

```
public void onPrepared( MediaPlayer mp ) {
  isMusicPrepared.set( true );
  // when the music is prepared, just start playing it
  mediaPlayer.start();
}
```

When we first try to play() the music, we call prepareAsync(). When to file has been loaded and prepared it will call onPrepared, where we in turn set isMusicPrepared to true, and then start playing the music. Successive calls to play() will not attempt to prepare again, unless the music was stopped.

Initiating, playing, and stopping/releasing the music is straightforward, and we'll call it as part of the GameEngine logic.

```
Music themeMusic = new Music( activity, "theme.mp3" );
themeMusic.play();
themeMusic.release();
```

Sound Effects

Have you ever watched a movie and noticed the good work of the foley artists, those folks who craft realistic sound for a movie, like the sound of a horse trotting? Probably not. It's a thankless necessity, but like the bassist in a heavy metal band, you notice when they're gone. Good video game sound effects suffer a similar fate.

For convenience we're putting this code in the GameEngine class, which we'll cover in more detail in the next section. For now we're more interested in the mechanism that we employ to play those effects, called the Android media SoundPool. The SoundPool keeps a set of simple, short audio files resident in memory, so it's imperative that you keep the files small, and relatively few.

If there wasn't much work involved in adding music to a GDK app, there's even less to include sound effects. We need to load the sound asset as a file descriptor object, and load it into the SoundPool.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameEngine.java

```
private int loadSoundPoolId( String fileName ) {
  try {
    AssetFileDescriptor assetDescriptor = activity.getAssets().openFd( fileName );
    return soundPool.load( assetDescriptor, 0 );
  } catch (IOException e) {
    throw new RuntimeException("Failed loading sound fx " + fileName);
  }
}
```

Before we can use the soundpool object, however, we must first initialize it. The constructor values are the size of the pool, the audio stream type, and a

sound quality value which should always be 0, since it's not used anyway by Android.

We want to create two sound effects: jumping and a sound to play when the game is over (when poor Miss Pitts meets her inevitable end).

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameEngine.java

```
soundPool = new SoundPool( 4, AudioManager.STREAM_MUSIC, 0 );
jumpSoundId = loadSoundPoolId( "jump.wav" );
endSoundId = loadSoundPoolId( "end.wav" );
```

Like any good programmer, we end with releasing resources back to the operating system. We handle releasing the theme music as well as the sound pool effects all at once when the game ends.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameEngine.java

```
public void releaseAssets() {
  themeMusic.release();
  soundPool.unload( jumpSoundId );
  soundPool.unload( endSoundId );
}
```

This is the last component we needed to create, play, and release music assets. Next we'll put everything together into a working video game.

Game Logic

Finally we reach the guts of the game, the game objects and the game engine. This section isn't about Glass per se, but a description of how you can turn the Glass-focused aspects above into a video game. The benefit of our design is that, with minimal changes, the code in this section could be used in a comparable game on an Android phone, or even Android TV.

Everything we've done up until now involves rather generic functions. We create an immersion with a render view that displays animations. We accept user inputs by tapping a gesture bar and capturing gravity events. We even play music and allow for sound effects. When we put these parts together, managed by the game rules we're about to create, only then do you have a working video game.

These components are represented in our design as implementations of an interface called GameObject, and a class named GameEngine. Let's start with our game objects.

Game Objects

Let's take an inventory of the game object requirements, that we'll turn into Java classes.

We can't have a game without a protagonist, so we'll need a MissPitts character class. Since she can fall into pits, this implies surfaces exist that aren't pits. Let's channel established 80's side-scrollers, and call these surfaces Blocks. We'll call a sequence of pits/blocks a Level, which we'll randomly generate as a never-ending series as the character maneuvers through them. We also need a class to keep track of the player's Score, which will display mid-screen.

You can see all of these components in Figure 48,*Designing MissPitts in One Screen*, on page 220.

All four game object types can update their internal state, and also be painted on the screen. So let's standardize them by making a GameObject interface that requires objects that implement update() and paint().

We'll pass the bufferFrame around, but wrapped in a Canvas object. Canvas contains many convenience methods, that will write commands to the underlying buffer, like drawBitmap. It saves every object from having to manipulate the bufferFrame image manually.

Game objects take care of painting themselves on the screen for each frame, simplifying the work of the GameEngine, and separating the logic into each specialized class.

Let's take a quick tour of the game object classes: MissPitts, Level, Block, and Score.

An Example Game Object, Starring the MissPitts Class

Let's start with the MissPitts object which represents the character that the player controls. She's the player's avatar in the game world. The GameEngine is responsible for interpreting the commands sent by the player, and changing the state of MissPitts. The MissPitts object itself is responsible for the two things that every GameObject is: ensuring values sent to it by the GameEngine keep are consistent with its own rules (like physics), and drawing itself.

MissPitts contains values for her current speed in x and y coordinates, the center of her position in x and y coordinates, and a bounding box that corresponds to her shape on the screen used for collision detection.

Collision Detection

You've likely heard the word before, even if you're not familiar with the concept. Collision detection is pretty much exactly as it sounds. It's the practice of detecting if two objects have collided. There are many ways of going about it, and the more detailed the shape of the object, and the more objects in play, the more processing power (aka crazy math) it takes to decide if two objects on the screen are intersecting or not. But we're taking the easy way out by simply drawing a rectangle on Miss Pitts, and rectangles on the blocks, and scanning for intersection. If a collision has been detected, stop her from falling. If not, let her keep falling until she hits a block, or falls right off the bottom of the screen (aka, into a pit).

Here is an example of the MissPitts class's update() and paint() methods.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/objects/MissPitts.java

```
public class MissPitts implements GameObject {
  public MissPitts( Score score ) {
    this.score = score;
    boundary = new Rect();
    centerX  = 100;
    centerY  = 100; // drops from the sky
    // animates steps for a quarter of a second
    animatedWalking = new MissPittsAnimator( 250 );
  }
  public void update() {
    if( speedX < 0 ) centerX += speedX;
    if( centerX <= 200 && speedX > 0 ) centerX += speedX;
    centerY += speedY;
    speedY += 1;
    if( speedY > 3 ) jumped = true;
    // The player's bounding rectangle for collision detection
    boundary.set( centerX - 35, centerY, centerX, centerY + 70 );
    // update the animator tick to create an animation illusion
    animatedWalking.update( RenderView.MSPF );
  }
  public void paint( Canvas c ) {
    // choose which player image to show
    final Bitmap sprite;
    if( jumped ) {            sprite = MissPitts.bitmapAsset2; }
    else if( !movingRight ) { sprite = MissPitts.bitmapAsset1; }
    else {                    sprite = animatedWalking.getImage(); }
    c.drawBitmap( sprite, centerX - 61, centerY-HEIGHT/2, null );
  }
```

Since we want to create the illusion of Miss Pitts walking, we employ a little animation class that simply alternates her two images which are drawn with feet together and feet apart every 1/4 of a second. When animated, she looks to be walking. If she's jumping we display the image where her legs are apart,

and when standing, legs together. We get quite a few movements out of only the two images that follow.

The other GameObjects conform to their update own game logic paint methods, which we won't print in this book. You can download the code and follow along. To wrap-up this section, let's cover the remaining GameObjects.

The Remaining Game Objects

Our game has three more GameObjects, namely, Level, Block, and Score.

In our game we don't have levels that a user can see, but rather a Level class represents a set of blocks and pits. We render these a level at a time, and draw them on screen as the player passes through each level. Each level generates a random set of blocks and pits, ensuring that gameplay is unique each run through. When a player reaches the middle block of a level, we signal that the GameEngine should generate the next one.

Next, we use the Block class to represent a count of one or more 40 pixel blocks, or a pit. Just like MissPitts, blocks track their x and y positions, although y is always the same vertical position of 40 pixels, since levels in this game are flat. With a small amount of coding, you could place blocks vertically too, giving them random heights.

Each block/pit scrolls to the left as MissPitts moves to the right, providing an illusion of a camera panning along with the character.

Blocks also check if the player has collided with a block. If so, we stop the player from falling. We also track if a player has landed on a block at the far end of a pit. If so, they Score a point.

The Block paint() method draws as many block images as the blocks count. It draws on the Canvas passed in from the Level paint() Canvas, which itself is called from the GameEngine.

The Score is our final, and simplest, GameObject. Score needs only update and keep the player's current score, and then paint that value onto the middle of the screen.

Game Engine

Think of the components we've implemented so far like stone blocks arranged in an archway. There's one important stone that holds the weight of the entire structure, called a keystone. The GameEngine class is the keystone of the entire game. It's responsible for loading assets like images and audio, it manages game objects and resources, it handles settings, physics, current game state, tracks the score, and paints the frameBuffer. Without a game engine, you don't have a game.

Coordinating the Game Objects

All of the game objects will paint themselves, but are coordinated by the GameEngine. It updates each object on the screen based on user input and game physics, and informs objects to draw the frameBuffer for the RenderView.

The GameEngine is responsible for creating all of the game objects, as well as tracking overall state like current levels, whether the game is RUNNING or GAME_OVER, and playing the theme music.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameEngine.java

```
public class GameEngine {
  enum GameState { RUNNING, GAME_OVER }
  public GameEngine( Activity activity, UserInputManager inputManager,
                     Bitmap frameBuffer ) {
    this.activity = activity;
    this.inputManager = inputManager;
    canvasBuffer = new Canvas( frameBuffer );
    initGameAssets();
    score = new Score();
    missPitts = new MissPitts( score );
    currentLevel = new Level( 0, 0, missPitts );
    nextLevel = currentLevel.next();
    state = GameState.RUNNING;
    themeMusic.play();
  }
```

It's also responsible for loading up any external assets, such as the images that represent Miss Pitts, or the bricks image that make the blocks of our pit-laden roadway. It also loads the theme music and any sound effects.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameEngine.java

```
private void initGameAssets() {
  MissPitts.bitmapAsset1 = loadImage("pitts1.png");
```

```
  MissPitts.bitmapAsset2 = loadImage("pitts2.png");
  Block.bitmapAsset      = loadImage("block.png");
  soundPool = new SoundPool( 4, AudioManager.STREAM_MUSIC, 0 );
  jumpSoundId = loadSoundPoolId( "jump.wav" );
  endSoundId = loadSoundPoolId( "end.wav" );
  themeMusic = new Music( activity, "theme.mp3" );
}
private Bitmap loadImage( String fileName ) {
  InputStream in = activity.getAssets().open(fileName);
  return BitmapFactory.decodeStream( in );
}
```

The following updateFrame() method may look innocuous, but it's the core of our game. It ties together the gesture events that the player makes, with updates to the MissPitts character object and related sound effects. It also updates the two active levels (current and next), which, if we recall the Level update() method, cascades updates to each of the blocks/pits in the level. Finally, it checks if the game is over because MissPitts has fallen off of the bottom of the screen, i.e. down a pit. If so, we play a sad song and change the game state from RUNNING to GAME_OVER.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameEngine.java

```
private void updateFrame() {
  List<GameEvent> gestureEvents = inputManager.getEvents();
  // Handle player events, jumps/walks
  for( int i = 0; i < gestureEvents.size(); i++ ) {
    switch( gestureEvents.get( i ) ) {
    case JUMP:
      soundPool.play( jumpSoundId, SFX_VOLUME, SFX_VOLUME, 0, 0, 1 );
      missPitts.jump( MissPitts.SINGLE_JUMP );     break;
    case JUMP_HIGHER:
      soundPool.play( jumpSoundId, SFX_VOLUME, SFX_VOLUME, 0, 0, 1 );
      missPitts.jump( MissPitts.DOUBLE_JUMP );     break;
    case JUMP_HIGHEST:
      soundPool.play( jumpSoundId, SFX_VOLUME, SFX_VOLUME, 0, 0, 1 );
      missPitts.jump( MissPitts.TRIPLE_JUMP );     break;
    case WALK:
      missPitts.moveRight( MissPitts.WALK_SPEED ); break;
    case RUN:
      missPitts.moveRight( MissPitts.RUN_SPEED );  break;
    case STOP:
      missPitts.stopRight();  break;
    }
  }
  // Update the player position and relative movement of the levels/blocks
  missPitts.update();
  checkGameOver();
  currentLevel.update();
  nextLevel.update();
```

```
}
private void checkGameOver() {
  if( missPitts.getCenterY() > 500 ) {
    // fell down a pit :(
    soundPool.play( endSoundId, SFX_VOLUME, SFX_VOLUME, 1, 0, 1 );
    state = GameState.GAME_OVER;
  }
}
```

The updateFrame() method is called as long as the game is running, since RenderView calls GameEngine update() in a loop. When the game is over, all of the music/sound assets are released, which we saw in Rendering Visuals and Playing Audio on page 229. Then the activity is marked as finished, the sensors are unregistered, the renderView thread is stopped, and the high score is potentially saved (GameActivity onPause()).

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameEngine.java

```
public void update() {
  switch (state) {
  case RUNNING:
    updateFrame();
    break;
  case GAME_OVER:
    releaseAssets();
    activity.finish();
    break;
  }
}
```

Finally, we reach a method that can only be described as a sheep in wolf's clothing. It's responsible for clearing the frameBuffer, and redrawing every object in the game in their current state.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameEngine.java

```
public void paint() {
  // clear the buffer with a blue sky color
  canvasBuffer.drawColor( 0xFF22AAAA );
  // draw the blocks
  currentLevel.paint( canvasBuffer );
  nextLevel.paint( canvasBuffer );
  // draw Miss Pitts
  missPitts.paint( canvasBuffer );
  // Paint the score mid screen
  score.paint( canvasBuffer );
}
```

Each and every frame in the game is drawn by this innocent looking bit of code.

Saving the Final Score

When the game is over, we save the score when the GameActivity is paused. But we only need it saved when the Activity is being finished, so we check isFinished().

```
public void onPause() {
  // ...
  if( isFinishing() ) {
    saveScores();
    // ...
```

Saving simple key/values is easy by using the Android SharedPreferences object. You need only give it a unique name to represent the preferences store (we call it *scores*), and then save the *highscore* with the SharedPreferences.Editor.

chapter-14/MissPitts/app/src/main/java/glass/misspitts/GameActivity.java

```
private static final String PREF_SCORES = "scores";
private static final String PREF_HIGH_SCORE = "highScore";
private void saveScores() {
  SharedPreferences sp = getSharedPreferences(PREF_SCORES, Context.MODE_PRIVATE);
  int lastScore = engine.getScore().getValue();
  int highScore = sp.getInt( PREF_HIGH_SCORE, 0 );
  // update the shared pref value if this is a new high score
  if( lastScore > highScore ) {
    sp.edit().putInt( PREF_HIGH_SCORE, lastScore ).commit();
  }
}
private int getSavedScores() {
  SharedPreferences sp = getSharedPreferences(PREF_SCORES, Context.MODE_PRIVATE);
  return sp.getInt( PREF_HIGH_SCORE, 0 );
}
```

One last thing. In order to write anything to the Glass SSD, you must add this permission to the Android manifest.

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

It requests that our app is allowed to write to the SSD, be it by SharedPreferences like we did above, or streaming a file to the storage device.

Wrap-Up

In this chapter, we created an entire world, which is no small feat. We designed how it looks, how it sounds, and how it reacts to user interaction. But beyond creating a video game, we learned more details about interacting with Glass's gesture bar and 9-axis sensors to track head movement. By leveraging these sensors we we able to track head gestures for movement, and made our game work naturally for Glass in a way that wouldn't be the same in any other form

factor. We also employed a bit of music and sound effects to give our application depth.

We also learned a few rules along the way about Glass game design, such as not to design games that require a large time commitment. Anything more than a few seconds to a minute, and you're going to make players nauseous. We dealt with this problem by simply making the game difficult.

This was a big project, but the end result is fun. Our separation of inputs and assets from objects and the game engine provides a general purpose framework for any number of Google Glass side-scrollers. Feel free to play around, and make any modifications and improvements to customize your own game.

As this book has progressed, we've increased the complexity of our projects. But we've also toyed with lower level operations. In the next chapter we continue this trend, and look at how to interact directly with the Android operating system, play with native code and hardware, and even learn how to hack our Glass device—in the interest of science, of course.

CHAPTER 15

Preparing For the Real World

This chapter is meant to serve as a map to help you brave the wilderness of bringing Glass applications from your IDE, to the real world. Programming well is a difficult endeavor, and contrary to your best laid designs, code doesn't always work how you might hope or expect. But you can save yourself plenty of headaches later on if you invest time upfront learning how to test, debug, and profile your code.

We're going to begin by learning how to construct and run tests and other tools for debugging and performance analysis. We then take a closer look at some command-line tools, where you can write automated tests for your Glassware. Finally, we'll put the finishing touches on packaging an APK for use by others, by securely signing the package. Once you've done that, you're free to share your project with the world.

Testing Glass Apps

One of the cornerstones of delivering professional quality applications includes writing plenty of tests. More tests can only be a good thing, and in some cases, you may find that you're writing more test code than actual application code. In a vernacular of quality assurance, there are two major varieties of test: *white box* and *black box*. Both styles are made available to Android/GDK.

White box testing is concerned with the correct operation of specific internal components, and is sometimes called *unit testing*. Android provides a testing framework that leverages a combination of JUnit[1] (a popular Java unit testing framework), instrumentation to hook into the Android lifecycle, helper libraries to more easily manipulate or mock values, and tools to run the tests.

1. http://junit.org

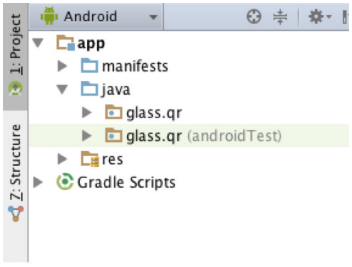
Black box (a.k.a. functional, integration) testing focuses on testing only the points that an end user can operate, such as tapping a card, or swiping a touchbar. Android blackbox testing is most easily done with a bundled Python execution environment named *monkeyrunner*.

Testing API Fundamentals

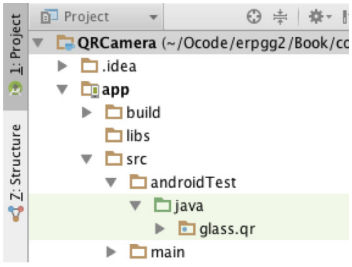
A big benefit of building Glass on Android OS, is that many of the Android testing tools are immediately available for Glass apps. Much of the work required to setup and use the Android Testing framework is integrated into either Android Studio or the SDK tools—so it's easy to get started. To create a test in AS requires first creating a few test classes. These test classes are separated from the project code, we don't have to worry about test binaries being bundled with our app's APK.

There's one bit of bad news, however. While the Android test framework can run against a virtual device that mimics the hardware of a mobile device, no such virtual device exists for Glass. In order to run these tests, you'll have to actually run them one your physical Glass device. So make sure your Glass is plugged in, fire up Android Studio, and we'll get started with the *QRCamera* project from Chapter 13, *Voice and Video Immersions*, on page 189.

You've probably noticed by now that every project we generate in Android Studio has created two java packages, the second is annotated with *(androidTest)* in the Android view. In the Project view you'll see that the package is under the src/androidTest/java directory, as you can see here.



AndroidTests in *Android* View



AndroidTests in *Project* View

You can safely delete the ApplicationTest class—we're going to create our own simple unit test. Let's test something easy, like the QRCamera project's QRIntentBuilder class. This was a class that accepted a String, and converted it to one of several kinds of Intents: a web ACTION_VIEW Intent, a geolocation ACTION_VIEW Intent, or a plain Intent object with a text string attached.

Since this is a plain Android unit test, we'll create a class named QRIntentBuilderTest extending AndroidTestCase. AndroidTestCase extends the JUnit version 3 TestCase, so all tests you write need to be public void methods that begin with test. In our case, since we want to test that the QRIntentBuilder class properly built a web ACTION_VIEW Intent, we'll name our method testWeb().

chapter-13/QRCamera/app/src/androidTest/java/glass/qr/QRIntentBuilderTest.java

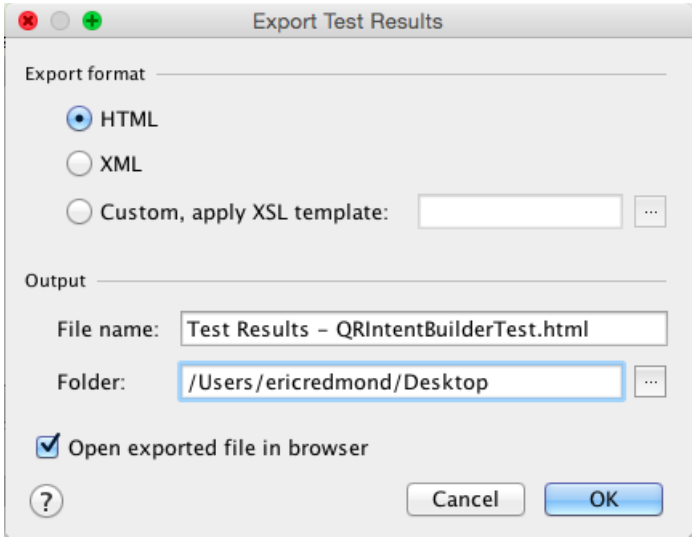
```
public class QRIntentBuilderTest
  extends AndroidTestCase
{
  public void testWeb() {
    QRIntentBuilder ib = new QRIntentBuilder("http://pragprog.com");
    Intent intent = ib.buildIntent();
    assertNotNull( intent );
    assertEquals( Intent.ACTION_VIEW, intent.getAction() );
    Uri uri = intent.getData();
    assertNotNull( uri );
    assertEquals( "http", uri.getScheme() );
    assertEquals( "pragprog.com", uri.getHost() );
  }
```

Just like any JUnit, we run the code we want to test with a specific input, and verify that response is what we expect with a series of JUnit assert* methods.

Let's run our test project against the QRCamera APK. It's as easy as right-clicking the QRCameraActivityTest file, *Run -> QRCameraActivityTest...* (Android Tests. If all goes well, you should see output like the following in your launch console.

```
Waiting for device.
Target device: google-glass_1-XXXXXXXXXXXXXXXX
Uploading file
  local path: ./QRCamera/app/build/outputs/apk/app-debug.apk
  remote path: /data/local/tmp/glass.qrcamera
No apk changes detected. Skipping file upload.
Uploading file
  local path: ./QRCamera/app/build/outputs/apk/app-debug-test-unaligned.apk
  remote path: /data/local/tmp/glass.qrcamera.test
No apk changes detected. Skipping file upload.
Running tests
Test running startedFinish
```

But the console doesn't say anything at all about whether the test was successful. To find out the result of your test, you should see an icon above the Test Results window labeled *Export Test Results*. You can generate html and open the file, which looks like this.



To fully test QRIntentBuilder requires more tests, such as testLocation(), and testText(). But let's move on to testing the QRCameraActivity immersion.

Testing Immersions

There is no special Glass immersion testing framework. However, since immersions are Android Activities, there does exist a special test case named ActivityUnitTestCase. It's still a JUnit-style test, but with a few methods that allow us to more easily hook into our QRCameraActivity lifecycle. Let's call this test QRCameraActivityTest.

Before each test method is run, JUnit first runs setUp() if it exists. We'll use that method to create an Intent that starts the immersion Activity, and pass it to startActivity(intent, null, null).

chapter-13/QRCamera/app/src/androidTest/java/glass/qr/QRCameraActivityTest.java

```
public class QRCameraActivityTest
  extends ActivityUnitTestCase<QRCameraActivity>
{
  private QRCameraActivity activity;
  public QRCameraActivityTest() {
    super( QRCameraActivity.class );
  }
  @Override
  protected void setUp() throws Exception {
    super.setUp();
    Intent intent =
        new Intent(getInstrumentation().getTargetContext(), QRCameraActivity.class);
    activity = startActivity(intent, null, null);
```

```
    assertNotNull( activity );
  }
```

Our test is rather basic, simply checking that the QRCameraActivitys content view contains our QRCameraView, and that the view is visible on the screen.

chapter-13/QRCamera/app/src/androidTest/java/glass/qr/QRCameraActivityTest.java

```
public void testContentLayout() {
  // Get the activity's content view
  FrameLayout cv = (FrameLayout)activity.findViewById( android.R.id.content );
  FrameLayout frameLayout = (FrameLayout)cv.getChildAt(0);
  // Get the QRCameraView and overlay
  assertEquals( 2, frameLayout.getChildCount() );
  assertTrue( frameLayout.getChildAt(0) instanceof QRCameraView );
  assertTrue( frameLayout.getChildAt(1) instanceof QRCameraActivity.OverlayView );
  // test that QRCameraView is displayed on the screen
  ViewAsserts.assertOnScreen( cv, frameLayout.getChildAt(0) );
}
```

We could continue testing deeper and deeper functionality, but let's move on. You should strive for tests that run as many parts of the application code as possible.

The percent of the total code count that was run is called the *code coverage*, and while 100% coverage is not always possible, a higher percentage is usually desirable to be certain you've tested all parts. We'll see how to check our coverage in the next section.

Testing LiveCards

Since there are two kinds of GDK applications, immersions and LiveCards, it's time to cover the latter. LiveCards correspond to a Service component, so we need to see how to build a Service test case. Let's create another test project, this time for *Stats* in Chapter 11, *Live Cards*, on page 147, named *StatsTest*.

Similar to our ActivityUnitTestCase, we need to create a service test that extends ServiceTestCase. This sort of test gives us service lifecycle support. Like an Activity test, you'll set up the service first.

If we wanted to customize the Service Context for more control over the test scenario, we could have replaced getContext() with a custom android.test.mock.MockContext object. We could mock the Application object as well with android.test.mock.MockApplication. We don't need to mock anything for our test case, however, so in this example we're building the Intent that launches the Service with the test's getContext().

chapter-11/Stats/app/src/androidTest/java/glass/stats/StatsServiceTest.java

```
public class StatsServiceTest
    extends ServiceTestCase<StatsService>
{
    private StatsService service;
    public StatsServiceTest() {
        super( StatsService.class );
    }
    @Override
    protected void setUp() throws Exception {
        super.setUp();
        Intent intent = new Intent( getContext(), StatsService.class );
        startService( intent );
        service = getService();
        assertNotNull( service );
    }
```

Our test is simple. We just want to ensure that our LiveCard is published after the startup of StatsService.

chapter-11/Stats/app/src/androidTest/java/glass/stats/StatsServiceTest.java

```
    public void testLiveCardIsPublished() {
        LiveCard liveCard = service.getLiveCard();
        assertNotNull( liveCard );
        assertTrue( liveCard.isPublished() );
    }
}
```

When you run this test case, you'll see two other tests are included that we didn't write: testAndroidTestCaseSetupProperly and testServiceTestCaseSetUpProperly. These tests merely check that our test has a Context object, and that the Service was successfully launched in setup, respectively.

Clean Projects with Lint

While we're on the topic of verifying our code by unit testing, let's take a short detour into static analysis. Android provides a tool named lint that analyzes your Android project settings and structure, and provides hints on how you can improve the project quality.

To run lint in AS, simply right-click a project, and select *Analyze -> Inspect Code*. Let's try it with the whole *Stats* project.

A new view will appear with a list of warnings. One will start with *Not targeting the latest versions of Android*. The Stats AndroidManifest.xml file highlights the lint issue with a suggestion for change. Since we can't control what version of the SDK that Glass supports, you can fix this by adding tools:ignore="OldTargetApi" to uses-sdk.

```
<uses-sdk
  android:minSdkVersion="19"
  android:targetSdkVersion="19"
  tools:ignore="OldTargetApi" />
```

Google's Android suggests fixing every Lint warning, but not every Lint issue can or should necessarily be fixed. Generally, fix all that you can, and exclude the rest. It helps pinpoint when new Lint issues creep up.

Our fingers are now firmly on the pulse of testing Immersions, LiveCards, and cleaning up our Android code in general. Let's move on to another side of testing Glass: debugging code and profiling the runtime.

Debugging and Profiling

Any good project should be well tested, but sometimes no number of tests can stop something from going wrong. In those cases, you have to roll up your sleeves and get your hands a little dirty by manually debugging or profiling your running code.

Debug mode

The classic brute-force method of debugging code is to use a debugging tool. Debuggers allow you to run live code, but create break-points where code execution stops, allowing you to step through portions of the code, even a line at a time if you like.

The Intellij tool that AS is built from provides a Java debugger service which works with Glass. Make sure your Glass is plugged in and the project you wish to debug is installed. You can then launch the Glass app as normal, and click the *Attach debugger to Android process* icon.

Now let's pick a point in our application we want to debug. Let's say that the QRCameraView is displaying the output from our camera incorrectly. One thing we might want to check is that the camera parameters actually contain the values we expect. The first thing we do is choose a break-point, or a location where the code should stop running. In AS you click to the left of the line you want to break at, which sets a little ball icon.

```
// Glass camera patch
Parameters params = camera.getParar
params.setPreviewFpsRange(FPS * 10(
final DisplayMetrics dm = getConte
params.setPreviewSize(dm.widthPixe
camera.setParameters(params);
camera.setPreviewDisplay(holder);
QRCameraActivity activity = (QRCam
```

With the AS debugger attached to your running Glass program, and a breakpoint set in the project source, the app will freeze when the camera loads.

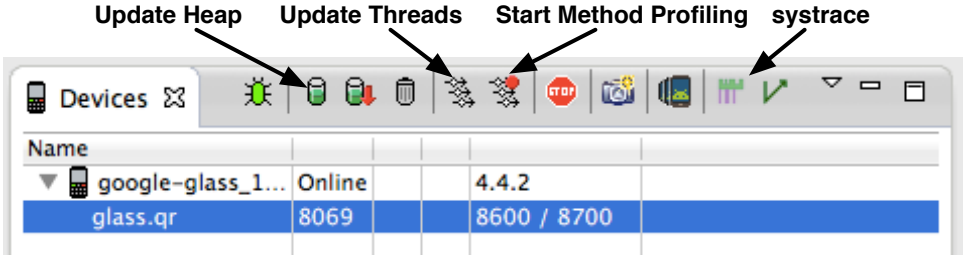
If you hover over a variable, you will see a popup of it's current state. The DisplayMetrics shows you Glass's default values. If you check it out you'll see that widthPixels and heightPixels are respectively 640 and 360. That is the Glass screen resolution. So we know that the camera params preview size is correct, allowing us to move on and look for our hypothetical problem elsewhere.

Let's move on from debugging to profiling, which is another important aspect of runtime analysis.

Dalvik Debug Monitor Server

An important aspect of improving application efficiency is by profiling the running application. This will give us a window into several metrics, such as visibility into active threads, network usage, running methods, stats about the Java heap and CPU load. The Android SDK provides an integrated set of views for doing this kind of analysis called the Dalvik Debug Monitor Server[2] (DDMS), which is accessible from the Android Device Monitor by calling monitor from the command-line.

By default, DDMS views don't collect much data, beyond basic system information. So you must activate which profiles to run, as you can see here.



2. http://developer.android.com/tools/debugging/ddms.html

We have many profiling options. Let's run the application, and select *start method profiling*. Then wait a few moments, and press the button again to *stop method profiling*. This populates Traceview, which gives us an overview of the methods called per thread in a timeline, and a profile of which methods are called and how much time is spent there. You should see something like the left half of the following figure.

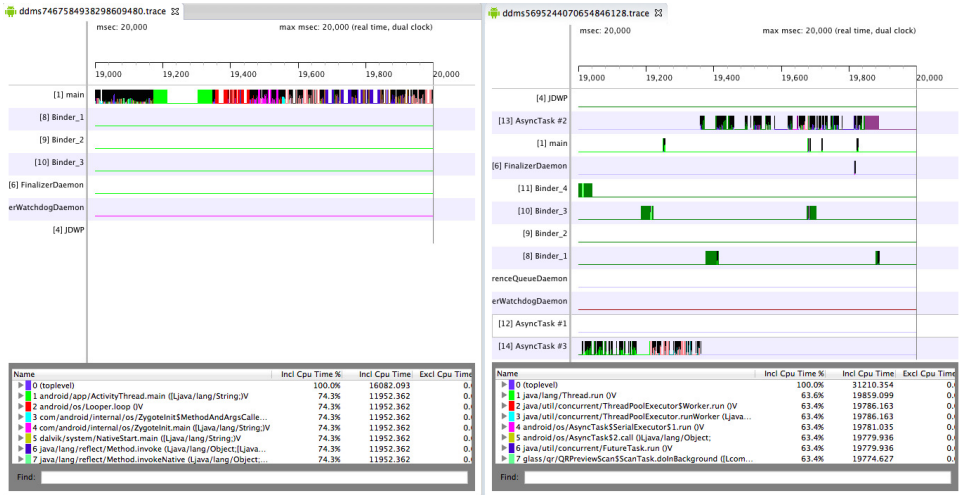
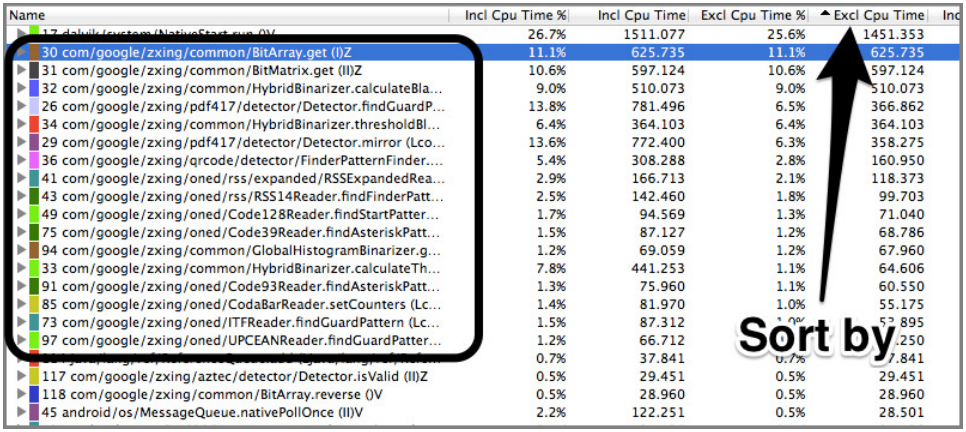


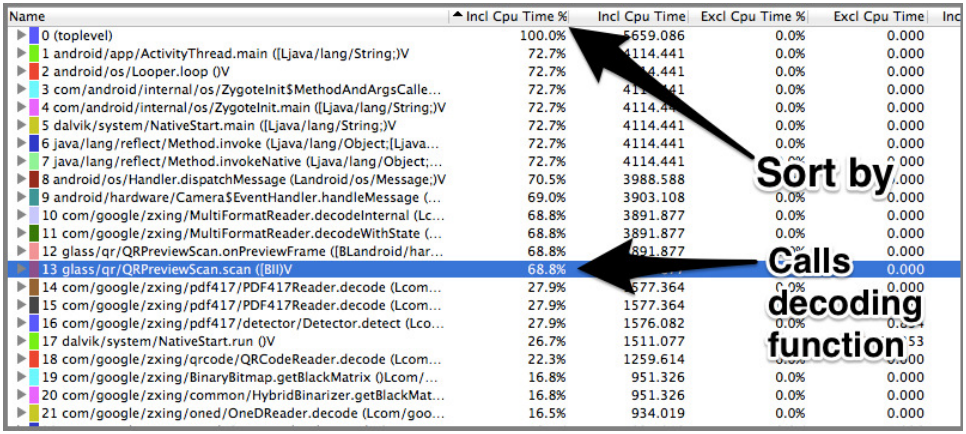
Figure 49—Profiling Pre and Post Code Change

Notice how densely packed the main UI thread is. While not necessarily bad, a busy UI thread can cause delays when the user needs to interact with the application—user interactions always occur in the main UI thread. If you tap to open the menu of the running app, you might notice a delay before the menu opens, as your request is queued behind our other code running on the same thread. We can fix this by offloading some of the heavier lifting to helper threads. But first we need to find where in the code we spend the most time, so we know what part to offload. Let's see if the profile panel provides a clue.

The profile panel allows us to sort by several dimensions. The default sorting is *Incl CPU Time %*, which sorts the method calls by how much total time the method has spent running, including the time of all method calls made by it. If you want to see how much time was spent running a specific method in a vacuum, excluding any sub method calls, sort by *Excl CPU Time %*. As you can see from the following figure, it's pretty clear what part of our app is the heaviest.



The ZXging library we use to scan incoming images for QR codes are all the top offenders. Sort again by *Incl CPU Time %* and you can see right where the biggest dropoff happens, at `QRPreview.scan` like in the following figure.



If you need to extract a block of code into a worker thread, the easiest option in Android is to pull the code into an `AsyncTask`.

chapter-13/QRCamera/app/src/main/java/glass/qr/QRPreviewScan.java

```
private static AtomicBoolean scanning = new AtomicBoolean(false);
class ScanTask extends AsyncTask<PlanarYUVLuminanceSource, Result, Result> {
  @Override
  protected Result doInBackground( PlanarYUVLuminanceSource... params ) {
    if( scanning.getAndSet(true) ) return null;
    PlanarYUVLuminanceSource luminanceSource = params[0];
    BinaryBitmap bitmap = new BinaryBitmap(new HybridBinarizer(luminanceSource));
    try {
      return multiFormatReader.decodeWithState(bitmap);
```

```
    } catch (ReaderException re) { // nothing found to decode
    } finally {
      multiFormatReader.reset();
    }
    scanning.set(false);
    return null;
  }
  @Override
  protected void onPostExecute(Result result) {
    if( result != null ) {
      Intent intent = new QRIntentBuilder(result.getText()).buildIntent();
      activity.launchIntent(intent);
      scanning.set(false);
    }
  }
}
```

With the new ScanTask subclass in place, you can change the code in the scan() method to the following.

```
private void scan(byte[] data, int width, int height) {
  PlanarYUVLuminanceSource luminanceSource = new PlanarYUVLuminanceSource(data,
    width, height, 0, 0, width, height, false);
  new ScanTask().execute(luminanceSource);
}
```

This will offload the heavyweight QR decoding into one of a pool of task threads. If you now reinstall and run the method profiler again, you should see a marked change, represented by the right half of Figure 49,*Profiling Pre and Post Code Change*, on page 253. More importantly, you should see that your app reacts to user inputs (specifically, launching the menu) without delay.

There are many other kinds of profiling options for Glass. You should certainly play around with these settings, to improve your code performance and user experience. There are as many kinds of profilers as aspects to investigate, many of which are documented on the Android developer site.[3] Most of these tools are available in Android Studio,[4] as well as the command-line.

Command-line Power Tools

The Android toolchain is not limited to Android Studio. In fact, many Android developers don't use an IDE at all. All of the tools you've used thus far in this book are available as command-line options. For example, much of the

3. http://developer.android.com/tools/debugging/index.html
4. developer.android.com/sdk/installing/studio.html

interaction between the AS and Glass is actually done via the adb command, wrapped in Intellij-based plugins to make your life easier. It's worth knowing a bit about the command-line tools for a few reasons.

First, you may not want to bother with the overhead of an IDE on lower power computers, or you may wish to SSH into an Android build environment. Second, with the command-line, you can more easily write automated scripts to perform some actions, such as integrating Android builds into a continuous integration service. Third, there are more options in most of the command-line tools than are available in their visual interface counterparts.

In this section, we'll dig into setting up and using some choice Android command-line options.

ADB

If you recall in Chapter 9, *Introducing the GDK*, on page 113 we used the adb (Android Debugger) command-line option to sideload our first APK application onto Glass. What we didn't discuss, is that adb can do so much more than install applications. In fact, it's the crux of almost all operations between your development machine and Glass.

If you didn't do it in Chapter 9, open up a command-line shell, and change directories to $ANDROID_PATH/sdk/platform-tools/. You can optionally add this directory to your PATH environment variable. If you run adb (or ./adb if you didn't add to PATH) you'll see a long printout of arguments and command options.

With your Glass plugged in, run adb devices -l. Your Glass should be listed, along with any other Android devices you may have plugged in.

```
$ adb devices
List of devices attached
015DB75A0B01800C   device usb:1D110000 product:glass_1 model:Glass_1 device:glass-1
```

If you see nothing, you may need to kill the adb server process (eg. adb kill-server) and try again.

Knowing that your Glass is connected to the adb service, we'll now take a look at some adb command-line options useful to Glass development.

logcat

No discussion of debugging and other analysis can be complete without knowing how to dig through log files. Logcat[5] is a view we introduced in

5. http://developer.android.com/tools/help/logcat.html

Chapter 9 used to list logs from the android.util.Log functions and other outputs. The Logcat AS view is populated by an adb option of the same name. It's actually a shortcut command to listing and filtering the contents of the /dev/log buffers: main application log, events in the system, radio or phone information, and system for low-level messages.

Running adb logcat outputs main and system by default. Use the -b option if you need to read the contents of another log buffer, such as the events buffer. The following is a sample of some system event log items. Press ctrl-C to end the log.

```
$ adb logcat -b events
I/netstats_mobile_sample(  543): [0,0,0,0,0,0,0,0,0,0,0,0,1412865389094]
I/am_create_service(  543): [0,1112346480,.WatchdogService,1000,1138]
I/am_destroy_service(  543): [0,1112346480,1138]
I/am_create_service(  543): [0,1112868608,.WatchdogService,1000,1138]
I/am_destroy_service(  543): [0,1112868608,1138]
I/am_create_service(  543): [0,1110176696,.GcmReceiverService,10023,1150]
...
```

Note how the first character is an I, which is called the log priority. The list of priorities from lowest to highest order: V (Verbose), D (Debug), I (Info), W (Warning), E (Error), F (Fatal), S (Silent). Following that is the log message tag, followed by the process id and the message contents.

You can filter only specific logs by appending tag:priority; wildcards (*) are supported. All matching tagged logs higher than the given priority will print, which means you can filter out any non-matching output with *:S—meaning display only tags with a priority greater than silent (which no log entry is ever given).

To see logged events when Glass sent a checking message to Google (yes, Glass checks in regularly), run this command.

```
$ adb logcat -b main CheckinTask:I *:S
I/CheckinTask( 1150): Sending checkin request (12238 bytes)
I/CheckinTask( 1150): Checkin success: https://android.clients.google.com/checkin \
  (1 requests sent)
```

It's good to know that Glass has checked in, but when did it happen? The time was logged, it just isn't output by default. You can control the format of the log output with the -v option followed by one of: brief, process, tag, raw, time, threadtime, long. Feel free to play around with different outputs, or check the docs online for details.

To see the time that your Glass last checked in, along with all log information, use -v long. We also use the silent shortcut option -s, which is the same as suffixing the filter with *:S.

```
$ adb logcat -b main -v long -s CheckinTask:I
[ 10-09 22:29:25.845  1150:20856 I/CheckinTask ]
Sending checkin request (12238 bytes)

[ 10-09 22:29:26.275  1150:20856 I/CheckinTask ]
Checkin success: https://android.clients.google.com/checkin (1 requests sent)
```

Snoop onto them, as they snoop onto us.

uninstall

In Part Two of this book, we've installed many sample projects. But after you're done, you may want to remove them. We saw that installing applications using adb is a one-line command that requires the APK file you wish to install. Uninstalling requires some investigation, because you don't uninstall an APK file, but rather, you uninstall an application package.

If you aren't sure of the package name, you can search the list with adb shell pm list packages, filtering with grep if you have a general idea of the name.

```
$ adb shell pm list packages -f | grep -i MissPitts
package:/data/app/glass.misspitts-1.apk=glass.misspitts
```

The pm command is run on the Glass device through the shell client. It outputs the APK filename and path, followed by the app's package name. You use this name to uninstall the application.

```
$ adb uninstall glass.misspitts
Success
$ adb shell pm list packages -f | grep -i "glass.misspitts"
```

You can run pm list packages again to verify that the project was removed.

shell

Android is an operating system based on Linux. Few actions can drive home this point than by running the adb shell. We can open up a shell client to Glass, similar to how telnet or ssh connect to a Linux server.

We used the shell command in one of the previous section's uninstall steps. But if you don't pass in a command to run with adb shell, you'll be logged in and greeted by a command line that looks like this.

```
shell@glass-1:/ $
```

You can run any number of standard Linux commands (ls, cd, cat, etc.) alongside some Android commands (pm, am, input, etc.). You can navigate the filesystem, as long as you have permission to poke around (some things, however, require root permission to access or modify). For example, if you want to see the list of pre-installed Glassware, you can run:

```
$ ls /system/priv-app/*.apk
```

You'll get some obvious apps such as GlassMusicPlayer.apk or GlassPhone.apk, and a few non-obvious ones like FusedLocation.apk. Let's get a bit more information about the Glass Music Player.

Investigating Packages

It's easy to get the package name if you know the APK file. We simply use the same pm list command we just used to find the MissPitts package, but this time we know we'll find something.

```
$ pm list packages -f | grep /system/priv-app/GlassMusicPlayer.apk
package:/system/priv-app/GlassMusicPlayer.apk=com.google.glass.musicplayer
```

So the music player's package name is com.google.glass.musicplayer. We can now issue commands against the installed application, if only we had a hook into the app. But we do! That's exactly what Intents are for. We can get the name of the intent with a little more sleuthing.

Run the following command, then open your Glass Music Player app. This will look for any new log info lines that contain the package name.

```
$ logcat -v long *:I | grep com.google.glass.musicplayer

START u0 {act=com.google.android.glass.action.VOICE_TRIGGER flg=0x10000000 \
  cmp=com.google.glass.musicplayer/.ListenToQueryActivity (has extras)} \
  from pid 903
START u0 {cmp=com.google.glass.musicplayer/.ResultsActivity (has extras)} \
  from pid 23909
Displayed com.google.glass.musicplayer/.ResultsActivity: +291ms (total +625ms)
```

The VOICE_TRIGGER line is what we're looking for. Now we know the name of the activity launched when we start up the app. Close the music app on Glass.

Launching Intents

With the package and activity name in hand, we can launch this APK's activity ourselves from the command line. Just use the am start command like so, with the home card active.

```
$ am start -n com.google.glass.musicplayer/.ListenToQueryActivity
```

The music player should launch. When you're done, you can stop the app.

```
$ am force-stop com.google.glass.musicplayer
```

Part of the argument for using the command-line is making it easier to automate some tasks, but launching an Intent is only part of automation.

Inputs

We also require the ability to send input events to Glass, such as tapping on the gesture bar, or clicking the camera button.

Running `input` alone will print its full Android usage doc. But for Glass, it's mostly useful for sending key events to the D-pad. Internally, Glass maps certain gestures as D-pad key events. `KEYCODE_DPAD_CENTER` is a gesture bar tap, `KEYCODE_BACK` is a swipe down. You can open and close the app menu from the home card like this.

```
$ input keyevent KEYCODE_DPAD_CENTER
$ input keyevent KEYCODE_BACK
```

Another useful keycode to send is `KEYCODE_CAMERA`. If the Glass screen is active, sending that event will take a photo, as though you pressed the top black camera button.

There are plenty more shell commands that can be used for a variety of purposes. We went through a bit of investigation and showed how to simulate user actions, like a high-level integration testing. But there's a better way.

Monkeyrunner

Scripting the launch of an APK and clicking around can get you pretty far for testing, but it's not optimal. What we need is an integration testing tool for Android that was design specifically for black-box testing. Enter Monkeyrunner.[6]

Monkeyrunner is a command-line interface and a set of Python classes that allow you to write automated functional tests. You don't have to be a Python expert, since the commands are fairly straightforward. There are three classes that represent different aspects of a test: MonkeyDevice,[7] MonkeyImage,[8] and MonkeyRunner.[9]

6. http://developer.android.com/tools/help/monkeyrunner_concepts.html
7. http://developer.android.com/tools/help/MonkeyDevice.html
8. http://developer.android.com/tools/help/MonkeyImage.html
9. http://developer.android.com/tools/help/MonkeyRunner.html

The following code loads the QRCamera APK on the Glass device, simulates a tap on the gesture bar. If you recall, tapping the gesture bar launches the menu. The monkeyrunner test then saves a snapshot of a the menu open on the screen to a local file named menu.png, before completing. Snapshots are a great way to verify that tests not only functionally pass, but look how you expect. You can even keep the snapshots between runs, and compare the images against a base expectation.

chapter-15/qr_test.py

```
# Imports the monkeyrunner modules used by this program
from com.android.monkeyrunner import MonkeyRunner, MonkeyDevice
# Connect to Glass and return with a MonkeyDevice object
device = MonkeyRunner.waitForConnection()
# Install QRCamera APK, presuming it's built
device.installPackage('../chapter-13/QRCamera/bin/QRCamera.apk')
# Run the Activity component and wait for launch
package = 'glass.qr'
activity = '.QRCameraActivity'
device.startActivity(component="%(package)s/%(activity)s" % locals())
MonkeyRunner.sleep(2)
# Open Exit Menu and wait for launch
device.press('DPAD_CENTER', MonkeyDevice.DOWN_AND_UP)
MonkeyRunner.sleep(3)
# Keep a screenshot of the Exit menu
result = device.takeSnapshot()
result.writeToFile('menu.png','png')
# Stop the app process
device.shell('am force-stop ' + package)
```

To run the test, execute monkeyrunner qr_test.py. Once the test has completed its run, it calls am force-stop glass.qr in the shell. This is a more absolute way of completing the test than a graceful shutdown (say, tapping the exit menu item). Depending on what you're testing, this may or may not be appropriate.

Avoiding MonkeyRunner.sleep

It's worth noting that 'MonkeyRunner.sleep' is a last resort, when you have to wait for some event to occur, but you can't detect it. Sleeping in tests can cause problems because sleep is a timed action in response to a queued event with no timeout. This can cause nondeterministic results if, for example, you sleep for four seconds while waiting for a menu to launch, but it takes five seconds for the launch to actually happen due to slow hardware.

There are a few options around our use of sleep, and it depends on what you're trying to test. None of the options are pretty. If you're waiting for the launch of an Activity, you can scan logcat for a message that the Activity has been launched. If you're waiting for a certain screen to appear, you can take snapshots until the result you expect populates (to some upper bound). The point is, avoid sleep if you can. Sometimes

writing deterministic tests can be more difficult than writing the application itself, but it's worth the peace of mind knowing that your tests are always testing properly.

Signing your Glassware

The last command-line tool we'll cover is a fitting end to this list. In all of Part Two of this book we've covered how to write Glass applications, but given no mention on how to go about building an APK that you can share with other users. In development, APKs are unsigned, but for a non-developer to install your Glassware, you must securely sign your application. This requires three command-line tools: keytool, jarsigner, and zipalign.

First, you must generate your own private key and keystore using keytool. You'll use this keystore file and key alias to securely sign your APK.

```
$ keytool -genkey -v -keystore ~/.android/glass.keystore \
  -alias glass -keyalg RSA -keysize 2048 -validity 10000
Enter keystore password:
Re-enter new password:
What is your first and last name?
...
[Storing /home/ericredmond/.android/glass.keystore]
```

Next grab the APK you need to sign, and run the jarsigner tool, with your keystore and key.

```
$ jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 \
  -keystore ~/.android/glass.keystore MissPitts.apk glass
```

The jarsigner will sign the APK inline. You can check that the sign was successful with the -verify option.

```
$ jarsigner -keystore ~/.android/glass.keystore -verify MissPitts.apk
jar verified.
```

A final trick to make your app ready for prime-time, is to run the zipalign command on your signed APK. It aligns data files that will be loaded on app launch, which reduces RAM consumption in the Android app.

```
$ zipalign -v 4 MissPitts.apk MissPitts-prod.apk
Verifying alignment of MissPitts-ready.apk (4)...
      50 META-INF/MANIFEST.MF (OK - compressed)
     603 META-INF/GLASS.SF (OK - compressed)
     ...
Verification successful
```

You can now share MissPitts-prod.apk with other Glass owners, who can side-load your MissPitts game. We covered the command-line methods here, so you can easily automate this process as part of a continuous integration environment.

Wrap-Up

This chapter may have seemed a bit scattered, but it all gathered around a common theme: how to prepare your application to be used in the real world. We did this by learning how to improve your Glass application using tools provided for Android.

We started with testing Glass apps, first with white-box unit testing, and later as black-box functional testing. We also learned a bit of static quality analysis using Lint. Next, we dove into the deep end by learning how to walk through running code to debug problems, followed by several runtime analysis choices for improving thread usage, method timing, CPU and memory profiling. Finally, we broke out of the IDE into the command-line, where true power-users live. Leveraging the command-line opened up a world of options around fine-grained control of our environment, ending with learning to package real APK applications for a mass audience.

Now that we know the fundamentals of GDK development, and how to build professional quality Glassware, we'll put those skills into practice. In the next chapter, we're taking an existing Android application designed for a smartphone, and converting it to Glassware.

CHAPTER 16

Turning an Android App to Glassware

So far in Part Two of this book, we've focused on writing GDK Glassware from scratch. However, you may have an existing Android application that you want to convert into Glassware. If you're familiar with standard Android applications, you've no doubt noticed by now that the GDK is different enough that you must create a new Glassware-specific project. But that doesn't mean you can't reuse large portions of your existing codebase.

Glass and mobile apps provide very different user interfaces, and no matter how much code you can reuse, transitioning your app to Glass will require fundamental interface changes. All hope is not lost, however. While it's likely that your application will offer a truncated set of options in one domain (such as the lack of a keyboard input), you can expand its offerings in another (providing realtime updates to the user over simple alerts).

We ran through a similar exercise in Chapter 8, *Turning a Web App to Glassware*, on page 99, where we converted a web application to also support a Glass interface via the Mirror API. We'll see a few overarching similarities compared to that process: we outline necessary use-cases, then redesign those cases for Glass, before finally implementing the Glassware. But the experience of converting an existing Android app to a GDK Glassware is different enough to require this dedicated chapter. You see, the Mirror API has no overhead on Glass to speak of. No matter how CPU, memory, or network intensive your web server application may be, it won't translate to a drain Glass's resources. But GDK code, since it runs exclusively on the device, can drain resources.

Design will be a bit different, too. Web apps are generally CRUD (Create, Read, Update, Delete) interfaces to some backend service—hence the popularity of REST APIs. Those operations map well to Glass as a producer and consumer

of messages. You take a photo with Glass and send it to a server via the Mirror API (Create), or receive an update from the service (Read).

Mobile smartphone and tablet applications often have complex features, such as multi-touch, OpenGL graphics rendering, and other actions that don't clearly map to CRUD. Navigating a map by pinching and swiping is technically a read operation in the CRUD world, but is not as straightforward of a read as, say, scanning through a Flickr blogroll.

This chapter is focused on steps you can follow to convert an existing Android app to Glassware. We'll use a simple note-taking Android application called Notoriety.

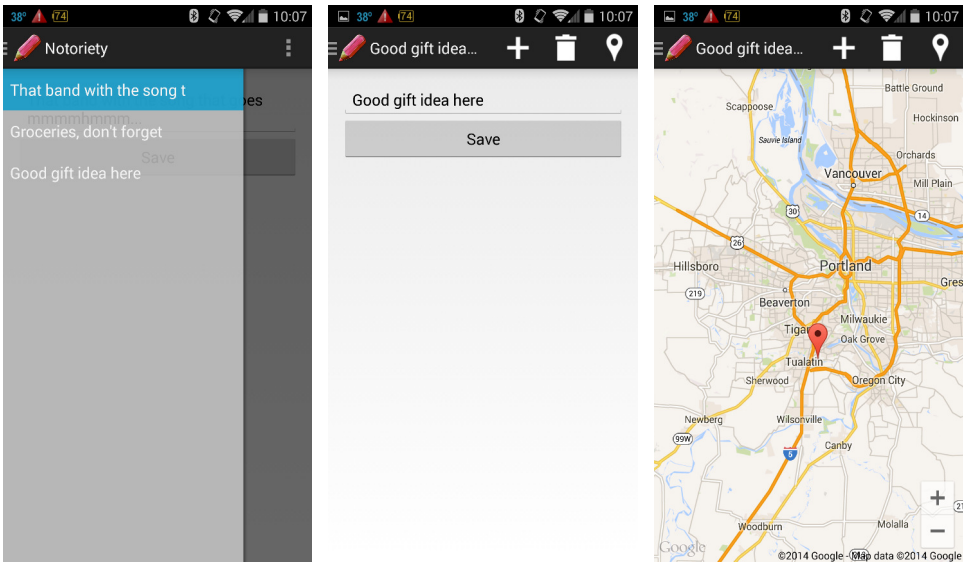
Notoriety App

Notoriety is a simple app for Android phones and tablets. It's built to let a user to take a quick textual note. The note grabs the user's current geolocation, and tags the note with both the creation time and place. The notes are displayed in a list, sorted by the datetime created, and can be opened for more detail. You can optionally view a map pointing to where the note was made. This let's you make short cryptic notes that only make sense with a location, such as "found a nice gift here" in a bookstore, or "this cake was great" at a reception hall.

To build and run the application with the Map API requires the Google Play Services (GPS) library in your Android Studio project, as well as GPS installed on your Android device. This is so you can view a map of where a given note was taken. You'll also need to create an API project in the Google console, and obtain a Google Maps API key. All of these steps are in the Google Android Maps API documentation.[1] If you wish to just try out the app on your Android device, you can download a demo APK[2]—just ensure your device is in development mode.

Here are three screenshots of the app.

1. https://developers.google.com/maps/documentation/android/start
2. https://github.com/coderoshi/glass



Notoriety is a basic Android design, with a drawer fragment[3] that lists all notes. The drawer can be accessed by swiping the screen from the left, where it overlays the currently viewed note. You can choose a list item to view the note, and then edit and save any changes. In the top bar, there are three buttons: one to create a new note (the plus-sign icon), one to delete the current note (trashcan), and one to view the location of the current note in a map (map pin icon).

The Android Code

The code for Notoriety is fairly simple. The classes under the app.notoriety package are the project's main activity, and a couple Android Fragments for drawing the notes list drawer and current note view. Fragments are common in standard Android development, where they have their own view trees and lifecycles (similar to Activities), but are not full-fledged components (we skipped any deep discussion of fragments in this book, since they are not used in GDK development).

The remaining code represents our Note and NoteList models, and a custom ContentProvider (the fourth component) which stores the user's notes in a SQLiteDatabase. Here are the packages and files that represent all of the Notoriety project code.

- app.notoriety

3. http://developer.android.com/training/basics/fragments/fragment-ui.html

  - MainActivity.java *(the project's main activity)*
  - NoteFragment.java *(a fragment to view/edit a note)*
  - NotesDrawerFragment.java *(a fragment that lists all notes as a drawer)*

- app.notoriety.models
  - Note.java *(represents a single note)*
  - NoteList.java *(represents a list of notes, interfaces with NotesProvider)*

- app.notoriety.providers
  - NotesProvider.java *(stores notes in a SQLiteDatabase)*

Another way to look at this code is as a Model-View-Controller (MVC) design. Note, NoteList and NotesProvider are the model layer; NoteFragment and NotesDrawerFragment and corresponding layouts (in the code you can download) represent the views; while the MainActivity plays the role of a simple controller. This can be a useful abstraction when we later decide what code can be reused.

Coming Up with Actions

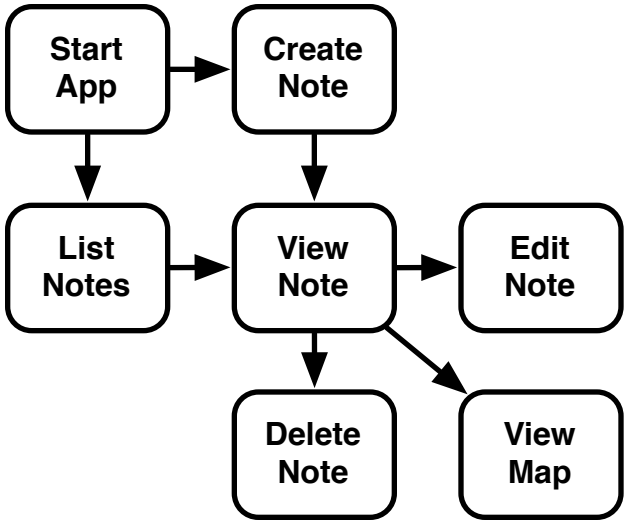
Our first task when porting an existing app to Glassware is the same as what we'd do when designing a brand new application. We start with a list of use-cases. Since we're porting an existing app, this is an easier job. We can start by dissecting the use-cases of Notoriety, and come up with a set of features that are portable to Glass. The steps to convert an Android app to GDK Glassware are not dissimilar from converting a webapp to Mirror API Glassware (Chapter 8, *Turning a Web App to Glassware*, on page 99), which are as follows:

- List existing Android app actions
- Remove features that don't work with Glass
- Add new features that do make sense

We start with a list of the existing Android app actions. Like a webapp in the Mirror API, it may be possible to implement every use-case, but that may not be necessary or even desired. The current list of Notoriety actions are:

- Create a note
- View a note
- Edit a note
- Delete a note
- View a map of where the note was created
- List all notes

We can show these in a flow chart of actions that a user can take.



Next, we scower this list and eliminate any features that must be removed due to technical limitations, or simply don't belong because they're too complex for Glass.

One use-case in this list that might be awkward for Glass users is editing an existing note. Since we don't have the benefit of a keyboard, imagine trying to replace a single word in a paragraph via voice commands. The best we can hope to achieve in a single note is to the replace a note entirely, or append to an existing one. Let's just agree it's easier for users to create a new note, and delete notes they don't want.

We haven't started designing yet, but with a bit of forethought, we can presume that our *list all notes* will follow common Glass design practices. This mean displaying notes as a scrollable list of note cards. Unlike the Android app's drawer that truncates notes, a card is capable of displaying an entire short note. So let's kill the *View a note* action, too.

Although there are facets of Glass that don't play well with Android smartphone designs, Glass also opens up new opportunities. The fun part of converting an existing Android app to Glassware is figuring out what new features you can provide. One that jumps out, is having Glass read a note aloud to the user.

We've decided to implement: List all notes, create a note, read a note, delete a note, and view a map. This seems like a good start. Let's hop into designing the Glass UI to handle these use-cases.

Designing a GDK Glassware UI

With our list of desired user actions in hand, our next task it to craft a design that makes sense for our uses-cases. Unlike the Mirror API, which is much more targeted in its capabilities, designing GDK Glassware requires a bit more consideration of user experience—putting yourself in your user's shoes, so to speak.

As we've seen throughout Part Two, the sky's the limit in GDK design. But we also know we should adhere to the Principle of Least Astonishment,[4] and keep our apps with a basic set of design constraints. Boiled down to a practical rule of thumb:

Always prefer a card-focused and menu driven design, leveraging existing components, with a glanceable interface. Stray from this rule only when you have a compelling reason to do so.

Let's create a checklist for converting our Android app into GDK Glassware.

- Choose the interface (live card or immersion)
- Design Glass workflow
- If you reuse assets/icons, simplify them
- Choose voice inputs

The first item on our list assumes you already need to use the GDK, as opposed to the Mirror API like in Chapter 8, *Turning a Web App to Glassware*, on page 99. But we know there are two interface choices, Live Cards and Immersions. Choosing one can be tricky or straightforward, depending on the application.

Choose the Interface

The first question we need to answer is what kind of UI make the most sense for your app. Should we create a live card or immersion? This question is often more easily answered if we rephrase it as: does your app run in the background, or in the foreground?

Apps that run in the background are a strong candidates for Live Cards, since they are presumed to be available as ever-present cards. Live Cards live in the timeline and represent now, but like any timeline card they don't always need to be visible. Once you've launched them, however, they are always running until the user explicitly stops them. Strong examples for Live Cards

4. http://c2.com/cgi/wiki?PrinipleOfLeastAstonishment

are: stock tickers, timers/clocks, music players, Tamagotchi games, bicycle performance apps, and so on.

On the other hand, apps meant to run in the foreground for some brief period of time should lean toward immersion activities. But, sticking with our goal of brief, glanceable, micro-interactions, we don't want the user to spend too much time in the immersive world. Immersive examples are: short video games, QR code scanners, photograph editors, video clip players, or anything else that you'll launch, use exclusively, and then close.

With these ideas in mind, it can still be hard to make a choice. Do we really want to make Notoriety available at all times in the background in case someone wishes to take a note? Or should it be a one-off deal where a user creates a note, or wants to read old notes? I find that, when in doubt, go with an immersion. They tend to open up more capabilities, reducing your odds of being caught flat footed when you want to add a new Glass feature like playing videos. And since immersions are activities, they tend to ease some conceptual burden of porting existing Android activities to Glass.

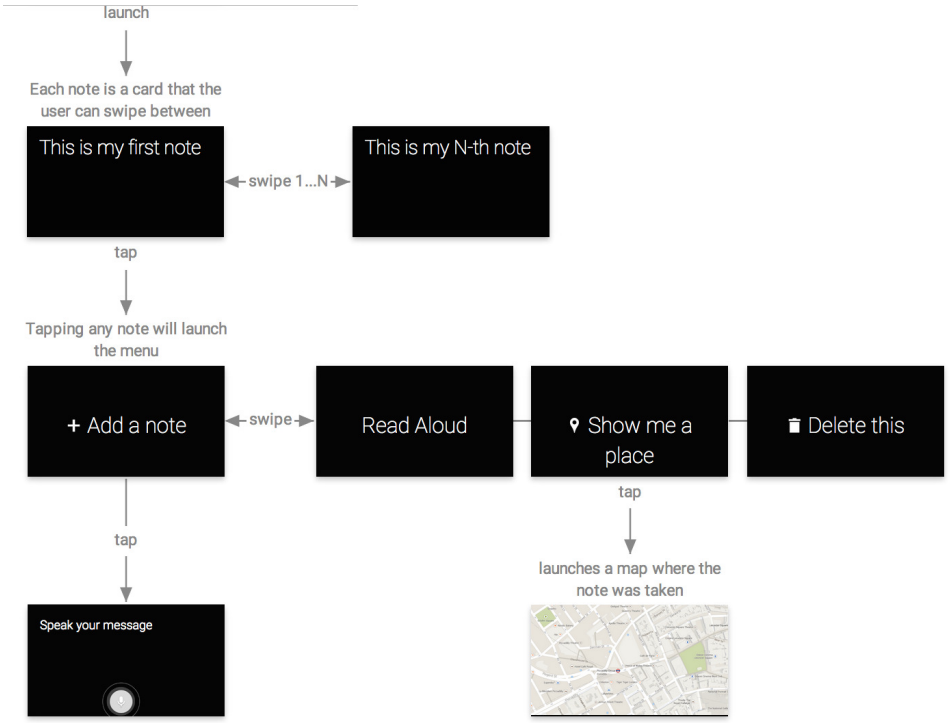
Let's make our app immersive, and move on to designing a workflow.

Design Glass Workflow

Knowing that we're creating an immersion frees us up in some ways, and constraints us in others. Since we know we want to display a list of notes, it only makes sense to create a card scroller, with one note in each card. In a Live Card we can't scroll without first opening a menu option (because otherwise, we'd just be scrolling back and forth through the timeline). But in an immersion, we can set our immersive activity's content view to a `ScrollView`, and save a step.

To help us visualize all of the parts of our new app, we're going to leverage Google's Glassware Flow Designer.[5] The app launches like any other directly into a scrollable list of cards in order of datetime.

5. https://glassware-flow-designer.appspot.com/



Listing all notes is represented by the first couple of cards at the top of the diagram. Tapping on any active note will launch the next row of menu options, where we can create a note, read a note, view a map, and delete a note. So far so good.

Since we don't have a keyboard, adding a note should leverage the speech-to-text action we used in Chapter 13, *Voice and Video Immersions*, on page 189, which will open the speech recognizer activity. Our code will have to capture this input and create a new note card.

Showing the map of the current note's geolocation is also something we've done in Chapter 13 as part of the QR Code reading app. Again, we'll leverage an existing action built into Glass by launching the map Activity.

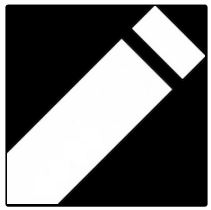
The other two actions don't launch any further activities. The *Read Aloud* menu option will play the given note's text through the Glass speaker, while deleting the current note will simply remove it from the list.

Simplify App Icons

Our Notoriety app was designed and written for a mobile phone and tablet environment which—sad but true—has far clearer resolution than Google Glass. This means that standard Android icons and assets can be colorful and intricately designed.

Glass icons, on the other hand, shouldn't fight for your attention. This means that you'll need to convert any iconography to simple, white on black, silhouetted logos. Our Android Notoriety app had a nice detailed image of a pencil, but needed to redesign the logo for Glass, as you can see here.

Original Logo



Glass Logo

Menu items should also use icons, to help users more quickly pinpoint the menu items they're looking for. Happily, Google actually provides many icons[6] for use in your Glassware.

Choose Voice Inputs

Since you'll (probably) want to publish this application on Google's MyGlass store, you must launch the app using one of the existing voice triggers.[7] Google won't accept any applications that have custom voice triggers or menu items. This means that any contextual menu items must also appear on the approved menu list.[8]

If the voice trigger or menu lists don't contain constants that you *really want* in your application, you can submit suggestions[9] to Google.

But in our case, there's a perfect voice trigger command for launching our app: `VoiceTriggers.Command.TAKE_A_NOTE`. If there are multiple apps installed that

6. https://developers.google.com/glass/tools-downloads/downloads
7. https://developers.google.com/glass/develop/gdk/reference/com/google/android/glass/app/VoiceTriggers.Command
8. https://developers.google.com/glass/develop/gdk/reference/com/google/android/glass/app/ContextualMenus.Command
9. https://developers.google.com/glass/distribute/voice-form

use this same trigger, Glass will list out the apps by name before continuing, for example: Notoriety, Evernote.

Although it can sometimes be a pain to find an existing voice trigger that encompasses your application, finding a set of menu items tends to be easier. They may not be perfect, but they're pretty close for our needs. All of these are defined under ContextualMenus.Command: ADD_A_NOTE, READ_ALOUD, SHOW_ME_A_PLACE, DELETE_THIS.

With voice constants in hand, we have everything we need to implement Notoriety on Glass, except the code.

The GDK Code

We programmers like to believe that coding is paramount, when we know deep in our hearts that it's but one step out of many. But it's certainly one of the most fun steps, which it's finally time to tackle. In order to keep the Android and GDK projects distinct in our heads, the Glass project will be called *Gnotoriety* (the "G" is silent).

Let's begin Gnotoriety by generating a new Glass application, and creating/configuring a MainActivity as the immersion. Since this activity will have to open a menu, it needs to implement GestureDetector.BaseListener to capture taps to the gesture bar. Furthermore, let's go ahead and implement LocationListener to track Glass's geolocation, and implement TextToSpeech.OnInitListener, which we'll use to initialize a TextToSpeech (TTS) object.

```
public class MainActivity
extends Activity
implements GestureDetector.BaseListener,
           TextToSpeech.OnInitListener,
           LocationListener
{
  private GestureDetector    mGestureDetector;
  private TextToSpeech       mTTS;
  private LocationManager    mLocationManager;
  private AudioManager       mAudioManager;
  private Location           mCurrentLocation;
  private boolean            mTTSInit;
  // required by GestureDetector.BaseListener
  @Override
  public boolean onGesture(Gesture gesture) {
    switch( gesture ) {
    case TAP:
      mAudioManager.playSoundEffect( Sounds.TAP );
      openOptionsMenu();
      return true;
```

```
    default:
      return false;
    }
  }
  // required by TextToSpeech.OnInitListener
  @Override
  public void onInit(int status) {
    mTTSInit = mTTSInit || status == TextToSpeech.SUCCESS;
  }
  // required by LocationListener
  @Override
  public void onLocationChanged(Location location) {
    mCurrentLocation = location;
  }
  @Override
  public void onStatusChanged(String provider, int status, Bundle extras) {}
  @Override
  public void onProviderEnabled(String provider) {}
  @Override
  public void onProviderDisabled(String provider) {}
}
```

With the required functions in place, we can track location changes, verify when TextToSpeech is initialized, and open the menu when a user issues a TAP gesture. We also added the manager fields (GestureDetector, TextToSpeech, LocationManager) that will use this Activity as a callback object, and an AudioManager to play sound effects.

All of these managers need to actually be initialized, where this implements their required callbacks. Let's put this code into a helper method called setupManagers().

chapter-16/Gnotoriety/app/src/main/java/glass/notoriety/MainActivity.java

```
private void setupManagers() {
    mGestureDetector = new GestureDetector(this).setBaseListener(this);
    mTTS = new TextToSpeech(this, this);
    mLocationManager = setupLocationManager();
    mAudioManager = (AudioManager)getSystemService(Context.AUDIO_SERVICE);
}
```

As usual, you need to implement onCreate(). Here we'll call setupManagers(), create a NotesList model (covered in the next section), and use it to populate a list of CardBuilder objects. Finally, we need a simple CardScrollAdapter (backed by the CardBuilder list) and CardScrollView set as the content view.

chapter-16/Gnotoriety/app/src/main/java/glass/notoriety/MainActivity.java

```
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setupManagers();
```

```
    mNotes = new NoteList( this );
    mCards = new ArrayList<>();
    for( Note note : mNotes ) {
        addNoteCardToAdapter( note );
    }
    mAdapter = new NotesScrollAdapter();
    mAdapter.notifyDataSetChanged();
    mScrollView = new CardScrollView( this );
    mScrollView.setAdapter( mAdapter );
    mScrollView.setHorizontalScrollBarEnabled( true );
    mScrollView.activate();
    setContentView( mScrollView );
}
```

Except for the TextToSpeech class, we've seen all of these manager objects before. TextToSpeech, once initialized, is how we'll read our notes out loud via a method called speak(...).

Don't Forget the Permissions

This is also a good place to remind you not to forget to set the necessary permissions in the Android Manifest. Tracking location will require android.permission.ACCESS_COARSE_LOCATION, for example. Forgetting to set the required permission of a resource that your application is using can be a nightmare to debug, and is not always obvious. If you suspect you're missing a permission, try setting com.google.android.glass.permission.DEVELOPMENT in your app manifest, and see if that works. If so, you'll need to track down the missing permission.

But what about the NotesList model object? This is all code we can copy from the Android Notoriety app to the Gnotoriety Glass app. In fact, the entire model layer, including the SQLiteDatabase backed NotesProvider can be reused.

- app.notoriety.models
  - Note.java *(represents a single note)*
  - NoteList.java *(represents a list of notes, interfaces with NotesProvider)*
- app.notoriety.providers
  - NotesProvider.java *(stores notes in a SQLiteDatabase)*

Check the API Version

Before you can reuse any existing Android code, ensure that code is valid in API 19. This means, in cases where you're using older than API 19, you'll have to upgrade your code. But the converse is also true, where you may have to downgrade usage of any objects and methods added in API 20 or greater.

Let's take a quick look at the model layer before going back to our Glassware interface.

The Model Layer

The model layer consists of three classes: Note, NoteList, and NotesProvider. Note is a simple POJO (plain-old Java object) with a unique id, the body text of the note, a created at date, and latitude/longitude for the note's geolocation.

chapter-16/Gnotoriety/app/src/main/java/app/notoriety/models/Note.java

```
public class Note implements Serializable {
  String id;
  String body;
  Date createdAt;
  double latitude;
  double longitude;
}
```

The most interesting class is the NotesProvider, which stores note data in a SQLiteDatabase, wrapped in a ContentProvider component. This is the only kind of the four Android components (recall from Chapter 10, *An Android Introduction on Glass*, on page 125) whose implementation we haven't covered. We won't now, but feel free to walk through the book's code.

In a nutshell, NotesProvider creates a database named *Notoriety* with a table named *notes*. It uses SQL to implement insert, update, query, delete methods (a.k.a. CRUD operations) that are stored in a lightweight database on the Glass device.

The content provider is defined in the Android manifest's application, just like any other component, like activity or service. Our provider element specifies three fields.

chapter-16/Gnotoriety/app/src/main/AndroidManifest.xml

```
<provider
    android:name="app.notoriety.providers.NotesProvider"
    android:authorities="app.notoriety.providers"
    android:exported="false">
</provider>
```

Like other components, android:name is the classname of the provider implementation. Next is the android:authority name of the provider, accessible as a URI, such as content://app.notoriety.providers/. This separates this provider from other provider datastores, like the com.android.calendar provider we used in Chapter 10, *An Android Introduction on Glass*, on page 125. Then set android:exported to *false*. This tells Android that we don't want to expose this provider outside of

our application—if you want other apps to access your user's notes list, you could set this to *true*.

Finally, we come to the NoteList. It is, in enterprise design pattern terms, a data-access object (DAO). This means it's a helper object that wraps the complexity of communicating with the underlying NoteProvider datastore, and translates the NoteProvider content values to and from Note model objects. For example, here is the code to add a new Note object to the database:

chapter-16/Gnotoriety/app/src/main/java/app/notoriety/models/NoteList.java

```
public void addNote( Note note ) {
  Uri insertUri = Uri.parse("content://app.notoriety.providers/notes");
  Uri newNote = mContext.getContentResolver()
      .insert(insertUri, noteToContentValue(note));
  List<String> segments = newNote.getPathSegments();
  note.setId( segments.get(1) );
}

private ContentValues noteToContentValue( Note note ) {
  ContentValues noteVals = new ContentValues();
  noteVals.put( "body", note.getBody() );
  noteVals.put( "createdAt", note.getCreatedAt().getTime() );
  noteVals.put( "latitude", note.getLatitude() );
  noteVals.put( "longitude", note.getLongitude() );
  return noteVals;
}
```

So to create a note we merely need to call noteList.addNote(note).

With the model layer out of the way, let's wrap this up by implementing our remaining requirements: launching a menu to add a note, read a note aloud, show a map, and delete the current note.

Voice-Driven Menu Actions

We know how to create menus in an immersion, but let's make things interesting. Nearly every interaction has had a verbal option, from launching the app, to taking the notes. However, the Glassware we've designed requires that a user touch the gesture bar to open the menu and choose an item. Wouldn't it be nice if the menu could be launched verbally as well? It's an easy change to make.

We start with requesting a feature from Glass under onCreate() named FEATURE_VOICE_COMMANDS. This setting will overlay your activity view with "ok, glass" at the bottom.

```
getWindow().requestFeature( WindowUtils.FEATURE_VOICE_COMMANDS );
```

For text-based menus we've had to employ two methods. The onCreateOptionsMenu() is responsible for populating the menu, while onOptionsItemSelected() handles actions when a menu item is selected. Voice menus require a minor change.

Rather than relying on those text-menu methods, we instead override their higher-level counterparts, responsible for all types of methods, including both text and voice. While onCreatePanelMenu() populates the menu in response to either touch or voice, the onMenuItemSelected() method responds to the chosen menu item. In both cases, they should only execute if the featureId variable is WindowUtils.FEATURE_VOICE_COMMANDS or Window.FEATURE_OPTIONS_PANEL.

chapter-16/Gnotoriety/app/src/main/java/glass/notoriety/MainActivity.java

```
public boolean onCreatePanelMenu(int featureId, Menu menu) {
    if( featureId == WindowUtils.FEATURE_VOICE_COMMANDS ||
            featureId == Window.FEATURE_OPTIONS_PANEL ) {
        getMenuInflater().inflate( R.menu.main, menu );
        return true;
    }
    return false;
}
```

In order to view the menu that we're inflating, we need to craft a menu resource. We've already chosen predefined ContextualMenus.Command constants that have been approved by Google. We put those constant names in the menu title field. We add some icons to help with glanceability.

chapter-16/Gnotoriety/app/src/main/res/menu/main.xml

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item android:id="@+id/action_new_note"
        android:title="ADD_A_NOTE"
        android:icon="@drawable/ic_plus_50"/>
    <item android:id="@+id/action_read_note"
        android:title="READ_ALOUD"
        android:icon="@drawable/ic_plus_50"/>
    <item android:id="@+id/action_map"
        android:title="SHOW_ME_A_PLACE"
        android:icon="@drawable/ic_location_50"/>
    <item android:id="@+id/action_delete_note"
        android:title="DELETE_THIS"
        android:icon="@drawable/ic_delete_50"/>
</menu>
```

The cool thing about this is that our app is using built-in menu commands. Our Glassware's menu options can be translated to different languages without further changes to the code itself. If you run Glass in Japanese, all of the built-in menu items will be displayed in Japanese.

Our pins are all set up, now it's time to knock them down by implementing the actions.

Add and Remove Notes

Since our actions are driven by the menu, the arbiter of these actions all take place under onMenuItemSelected() based on the chosen item. The first action in our list is creating a new note. This is easy, and we've done it before. We need only use the RecognizerIntent.ACTION_RECOGNIZE_SPEECH Intent action to launch the speech recognizer activity.

chapter-16/Gnotoriety/app/src/main/java/glass/notoriety/MainActivity.java

```
public boolean onMenuItemSelected( int featureId, MenuItem item ) {
    if( featureId == WindowUtils.FEATURE_VOICE_COMMANDS ||
        featureId == Window.FEATURE_OPTIONS_PANEL ) {
        Note currentNote = getSelectedNote();
        switch( item.getItemId() ) {
            case R.id.action_new_note:
                Intent intent =
                        new Intent( RecognizerIntent.ACTION_RECOGNIZE_SPEECH );
                startActivityForResult( intent, SPEECH_REQ_CODE );
                break;
            // ... other menu action cases...
        }
    }
    return super.onMenuItemSelected(featureId, item);
}
```

When the user has spoken his or her note, the action will return the text result. We capture this result with a request code constant we defined, named SPEECH_REQ_CODE. From there, we create a new note with the given spoken text, and also grab the current geolocation. Finally, you store the new note, and update the scroll list of notes.

chapter-16/Gnotoriety/app/src/main/java/glass/notoriety/MainActivity.java

```
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if( resultCode == RESULT_OK && requestCode == SPEECH_REQ_CODE ) {
        List<String> results =
                data.getStringArrayListExtra( RecognizerIntent.EXTRA_RESULTS );
        if( !results.isEmpty() ) {
            String body = results.get( 0 );
            Note note = new Note().setBody( body );
            if( mCurrentLocation != null ) {
                note.setLatitude( mCurrentLocation.getLatitude() );
                note.setLongitude( mCurrentLocation.getLongitude() );
            }
            mNotes.addNote( note );
            addNoteCardToAdapter( note );
            mAdapter.notifyDataSetChanged();
```

```
        }
    }
}
```

Deleting a note is even more straightforward. You merely delete the current note from the backend and remove the note from the scroll list. The model layer is doing most of the work here.

chapter-16/Gnotoriety/app/src/main/java/glass/notoriety/MainActivity.java

```
case R.id.action_delete_note:
        int selectedNotePos = mScrollView.getSelectedItemPosition();
        mNotes.deleteNote( currentNote.getId() );
        mCards.remove( selectedNotePos );
        mAdapter.notifyDataSetChanged();
```

Two actions down, two to go.

Reading a Note Aloud and Viewing a Map

The final two actions are actually very complex procedures. However, since we're able to leverage the built-in capabilities of Glass, they end up being the simplest of all to implement. They don't make any modifications to the backend, and they don't change the scrollViews adapter mCards list at all. They merely perform their action, then complete.

Reading a note is surprisingly simple. Since we've already created the TextToSpeech manager, we need only verify that it's successfully been initialized, then call the speak(text, queueMode, params) method. We get the text from the note's body. And TextToSpeech.QUEUE_FLUSH means to play the last chosen note only—don't queue up multiple texts to read if the menu item is chosen multiple times.

chapter-16/Gnotoriety/app/src/main/java/glass/notoriety/MainActivity.java

```
case R.id.action_read_note:
    if( mTTSInit && currentNote != null ) {
        mTTS.speak(currentNote.getBody(), TextToSpeech.QUEUE_FLUSH, null);
    }
    break;
```

That was painless. The next one is even easier, since launching a map is something we've done before in Chapter 13, *Voice and Video Immersions*, on page 189. Using the built-in Glass map activity we need only build an intent with a URI scheme of google.navigation. Using the ll URI query value, we'll also add the note's latitude and longitude.

chapter-16/Gnotoriety/app/src/main/java/glass/notoriety/MainActivity.java

```
case R.id.action_map:
        String nav = String.format("google.navigation:ll=%s,%s",
```

```
        currentNote.getLatitude(), currentNote.getLongitude() );
Intent mapIntent = new Intent( Intent.ACTION_VIEW, Uri.parse(nav) );
startActivity( mapIntent );
```

So much power, so little code.

And that's it! We've now ported the fundamental features from *Android Notoriety* to *Glass Gnotoriety*. While the interface may look and feel different, it's hard to deny that these apps perform the same function. And that's what supporting an app in different forms is all about.

Wrap-Up

This chapter was about porting an existing Android smartphone or tablet application into Glassware. This is meant as a starting point for your new Glassware, but by no means should you end there. One of the great, exciting opportunities of Google Glass is that it's a new way of interacting with software, and the world in general. This is the proverbial wild west, and the rules are always up for reinterpretation. Go explore.

This chapter wraps up Part Two, and the book as a whole. Just keep in mind that Glass is an emerging technology. That means that things are changing fast every day. This book, along with all of the references provided by Google and other community resources[10] should be all you need to implement any idea that the Glass is capable of.[11] The rest is up to your creativity. Now is your time to get in on the ground floor, and take the first steps in shaping the future of the computer-human interface.

10. http://github.com/coderoshi/glass
11. https://developers.google.com/glass

APPENDIX 1

HTTP and HTML Resources

This appendix contains a listing of the valid HTTP resources and their valid values, based on the Mirror API version 1, Glass platform XE11.

Every resource body is represented in HTTP as a JavaScript Object Notation (JSON) object with various fields. Every resource object body has a unique `id` that represents the object, and a `kind` that represents a resource type prefixed by `mirror#` (such as `mirror#location` for Location resource JSON objects). The rest of the fields are either writeable or read-only, or are objects that themselves are field values.

The base URL for the Mirror API is (usually) `https://www.googleapis.com/mirror/v1`. For timeline resources, there are two base URLs for the Mirror API: Mirror and Mirror Uploads. For most of the Mirror API the base URL is `https://www.googleapis.com/mirror/v1`. However, to upload larger objects the base is `https://www.googleapis.com/upload/mirror/v1`.

The following resource tables will be marked *default* for the base URL, *upload*, or *both* if it supports both.

Timeline

The timeline is the core resource for manipulating or gaining information about a Glass device's timeline. This is the resource you'll interact with most commonly.

HTTP Methods

The timeline resource supports all major CRUD (create, read, update, delete) operations as RESTful HTTP methods.

Description	Method	Path	Base
insert a new timeline item.	**POST**	/timeline	**both**
get the timeline item with the given ID.	**GET**	/timeline/{itemId}	**default**
list a user's timeline items accessible by this app.	**GET**	/timeline	**default**
update the timeline item with the given ID.	**PUT**	/timeline/{itemId}	**both**
patch the timeline item with the given ID as a partial update.	**PATCH**	/timeline/{itemId}	**default**
delete the timeline item with the given ID.	**DELETE**	/timeline/{itemId}	**default**

Read-Only Fields

Don't bother setting these; the Mirror servers populate them for you, often in response to some action your Glassware or users have taken.

Value	Description
id	The unique identifier string for this object,
kind	mirror#timelineItem. The type of JSON object this is, set by the mirror service resource,
isPinned	If true, this item is pinned to the left of the home card (rather than being in the main timeline to the right of the home card). This is set if a user taps a TOGGLE_PINNED menu item (more on that soon),
pinScore	The higher the number, the closer the pinned item is to the home card. This is automatically set by the TOGGLE_ PINNED callback,
isDeleted	This is set to true when the DELETE action is called on the timeline item, or the DELETE menu item is tapped.
inReplyTo	The parent timeline item ID, if this item was created in reply to that parent. Think of this in terms of a nested email conversation, where this item is a reply to a previous one.
etag	Similar to an HTTP etag, this autogenerated entity tag acts as a version of this timeline item. You can use the HTTP If-None-Match header field for a potentially cached value.

Value	Description
selfLink	This item as a gettable link; for example, https://www.googleapis.com/mirror/v1/timeline/1234567890 (it still requires an Authorized header, though).
created	A timestamp set when the item was created.
updated	A timestamp that's updated whenever the item changes.

Common Fields

The following are common optional timeline items you can set.

Value	Description
title	A readable title of this item, largely useful for debugging.
text	Outputs the given text to the card.
html	Renders the given HTML to the card.
speakableText	Similar to text, but rather than being printed on a item to read, this text is read aloud to a user when the READ_ALOUD menu item is selected (more on menu items later).
speakableType	A short, relevant string that describes the type of timeline item this is. Before text or speakableText are read aloud to a user, this will be read to give the user some context.
displayTime	An RFC 3339–formatted date that places this item in the timeline at that instant/position.
canonicalUrl	A URL that points to a more full representation of the data on this item.
sourceItemId	A custom ID you can use to link this item with some other data that means something to your system, such as a username.
bundleId	A unique ID to group together cards into a bundle.
isBundleCover	Set to true if this is the cover card for a given group with a common bundleId.
creator	A JSON object representing the creator of this item, which accepts a contact object detailed in the Contact section later in this appendix.
recipients	An array of JSON objects representing the contacts that are meant to receive this item, useful in the REPLY_ALL user action.

Value	Description
menuItems	An array of JSON objects representing menu options that a user can select. The details to populate these objects are under the MenuItem Object Fields section later in this appendix.
notification	A JSON object that sets how notifications are generated for this timeline item. It accepts a deliveryTime and a notification level (which only accepts DEFAULT).
location	Represents the location of this timeline item, populated with the same resource-body values defined in the Locations section later in this appendix.
attachments	An array of JSON objects representing attachments, populated with the same resource-body values defined in the Attachments section later in this appendix.

MenuItem Object Fields

These are the fields you can set as one of the menuItem object values under the greater timeline item body.

id	The unique ID for this menu item.
action	The action that this menu item takes when tapped. It can either be one of the values under Menu Action, or CUSTOM.
payload	Some data whose meaning changes depending on the menu action, such as a URL for OPEN_URI or PLAY_VIDEO.
removeWhenSelected	If true, remove this item from the menu if it is selected (used only when the action is CUSTOM).
values	An array of JSON objects representing menu-item appearances, according to state, whose values are under the Custom Menu Values section later in this appendix (used only when the action is CUSTOM).

Menu Actions

When creating a menu item, you must choose one from the following list of actions. The Required column shows a timeline item field that is also required to make this menu action work properly, or a dash if there are no required

fields. For example, you cannot use a REPLY_ALL menu action without a list of recipients who will accept that item.

Value	Description	Required
REPLY	Initiates a reply to send this item to the card's creator.	creator
VOICE_CALL	Calls the card creator's phone number (creator.phone_number).	creator
REPLY_ALL	Initiates a reply to send this item to the card's recipients list.	recipients
DELETE	Deletes this card.	—
SHARE	Shares this card with available contacts.	—
READ_ALOUD	Uses Glass text-to-speech to read either the speakableText (if set) or text field aloud, prefixed by speakableType.	speakableText or text
NAVIGATE	Opens a map showing your current location; the destination is the card's location.	location
TOGGLE_PINNED	Pulls this item outside of the normal timeline, making it an item to the left of your home card. It's a great shortcut for important or atemporal cards.	—
OPEN_URI	If the menuItem's payload value is a URL, opens that website in a browser window.	payload
PLAY_VIDEO	If the menuItem's payload value is a streaming video URL, plays that video.	payload
CUSTOM	A custom action defined by your code (more on what this means in the next section).	—

Custom Menu Values

When a CUSTOM menuAction object is set, populate the values field with an array of objects containing these fields.

Value	Description
state	This value applies only for the given state (DEFAULT, PENDING, CONFIRMED).
displayName	Text to display for this menu item.
iconUrl	A 50×50-pixel icon image (transparent-background PNG works best) to display alongside the menu text.

Timeline Attachments

The timeline resource has a subresource to manage attachments, sometimes known on the web as *timeline.attachments*.

HTTP Methods

Description	Method	Path	Base
insert a new attachment to a timeline item. Set the uploadType parameter to resumable to break a large upload into multiple requests. Otherwise, set this to media for a simple upload.	**POST**	/timeline/{itemId} /attachments? upload-Type={ut}	**upload**
get the attachment under the timeline item with the given IDs.	**GET**	/timeline/{itemId} /attachments/{aId}	**default**
list the attachment under the timeline item with the given IDs.	**GET**	/timeline/{itemId} /attachments	**default**
delete the attachment under the timeline item with the given IDs.	**DELETE**	/timeline/{itemId} /attachments/{aId}	**default**

Attachment Fields

Your timeline-item HTML can display these attachments by either their index in the timeline-item attachment array (for example, show the first in the list <img src="attachment:0">) or by the attachment ID (such as <img src="cid:1234567890">).

Value	Description
id	The unique ID for this attachment.
contentType	The content type of this attachment, in MIME format.
contentUrl	When the attachment is processed, this is the available URL for the content.
isProcessingContent	When the attachment is processed, this will be set to true, meaning the contentUrl can be accessed.

Locations

A location represents the physical location of a Glass device, or an object representing the location of something such as a business. This is usually set as a named latitude and longitude.

HTTP Methods

The locations resource is used to access specific location objects by ID or the latest. There are no location-creation/modification requests via the Mirror API.

Description	Method	Path
get the location by ID or latest.	**GET**	/locations/{locationId}
list a Glass device's most recent locations.	**GET**	/locations

Read-Only Fields

Value	Description
id	The unique identifier string for this object.
kind	mirror#location. The type of JSON object this is, set by the mirror service resource.
timestamp	An RFC 3339 timestamp set when the location was created. Unlike for the timeline, there's no created or updated field, since a location object is immutable.

Writeable Fields

When you make a request for a Glass Location, the latitude, longitude, and accuracy will be populated. But you can create your own location objects for other purposes, such as setting a timeline item's location field.

Value	Description
latitude	Degrees latitude, where positive is north of the equator and negative is south (–180 to 180)
longitude	Degrees longitude, where positive is east of the prime meridian and negative is west (–180 to 180)
accuracy	The accuracy of Glass's location, in meters
displayName	The name of this location (if named), such as a business name
address	The complete address, including street, city, state, country, zip, and so on

Subscriptions

The subscription resource supports most CRUD operations as RESTful HTTP methods, except for getting a single subscription.

HTTP Methods

Description	Method	Path	
insert a new subscription	**POST**	/subscriptions	default
list a user's subscriptions accessible by this app	**GET**	/subscriptions	default
update the subscription with the given ID	**PUT**	/subscriptions/{subscriptionId}	default
delete the subscription with the given ID	**DELETE**	/subscriptions/{subscriptionId}	default

Read-Only Fields

kind	mirror#subscription
updated	A timestamp that's updated whenever the subscription changes

Writable Fields

callbackUrl	A required URL that is called when a matching event happens; must begin with https://
collection	The required event source, either timeline or locations
operation	An array of actions that triggers this subscription's callbackUrl; can be one or more of the following: INSERT, UPDATE, DELETE
verifyToken	A token used to verify that Google created this notification
userToken	An optional token you can set to identify the user that originates a notification

Notification Objects

Notification objects aren't part of the subscription resource per se; these are the fields that are populated as a notification JSON body when a subscribed event has taken place and Mirror has posted to your callbackUrl.

collection	The required event source, either timeline or locations.
operation	That action that triggered this notification: INSERT, UPDATE, DELETE. Unlike the mirror#subscription type, this field is not an array, but a single operation string.
verifyToken	A token used to verify that Google created this notification.

userToken	An optional token you may have set during subscription to identify the user that originated the notification.
itemId	The timeline item or location ID this notification refers to.
userActions	An array of objects that each contain a type, such as PIN, and optionally a payload field, which is usually the ID of a selected CUSTOM menu item. These are one or more of the values in the next section of this appendix.

User Action Types

An array of userActions is part of the notification object.

Type	Description	Operation
REPLY	Replied to the creator of the item, which created a new timeline item	**INSERT**
REPLY_ALL	Replied to all recipients of an item, which created new timeline items for each recipient	**INSERT**
DELETE	Deleted this item from the timeline	**DELETE**
SHARE	Shared the item with available contacts	**UPDATE**
PIN	Pinned the item to the left of the home card	**UPDATE**
UNPIN	Unpinned the item back into the main timeline	**UPDATE**
LAUNCH	Updated the timeline item via voice command	**UPDATE**
CUSTOM	Chose a custom menu item, and accompanies a payload	**UPDATE**

Contacts

Contacts represent individuals (often humans), groups, or Glassware that can be shared with.

HTTP Methods

The contacts resource supports all major CRUD operations as RESTful HTTP methods.

Description	Method	Path
insert a new contact.	**POST**	/contacts
get the contact with the given ID.	**GET**	/contacts/{contactId}
list a user's contacts created by this app.	**GET**	/contacts
update the contact with the given ID.	**PUT**	/contacts/{contactId}

Description	Method	Path
patch the contact with the given ID as a partial update.	PATCH	/contacts/{contactId}
delete the contact with the given ID.	DELETE	/contacts/{contactId}

Read-Only Fields

id	The ID for this contact.
kind	mirror#contact.
source	Represents the Glassware application that created this contact. The Mirror API will always set this for you.

Writable Fields

type	Notes whether this contact is an INDIVIDUAL or a GROUP.
displayName	The name of this contact, be it a person's name or a group label.
speakableName	This should be a phonetic representation of a contact's name, which Glass will use when you speak the name. Similar to timeline items' speakableText, but used for input rather than output.
imageUrls	A list of up to eight URLs that represent this person or group. One is usually sufficient for a single person.
phoneNumber	A phone number for this contact. Usually a cell phone, since who uses landlines anymore?
priority	An optional number used to sort contacts when displayed in a list. The higher the number, the higher chance the contact will be at the top of the list. It's best to keep these numbers low, such as 0 for lowest and up to 10 for highest, lest you end up in an arms race against your own number system.
acceptTypes	A list of MIME types that this contact will be capable of receiving, displayed as a contact when attempting to SHARE an item of with one of those types. For example, if your sister is in your contact list with the image/* type, her name will be shown as a potential recipient of photographs taken with Glass.

acceptCommands	A similar idea to acceptTypes, but for voice commands. This is a list of voice commands that a contact can accept. Each list item takes an object that accepts one field type, and can be either TAKE_A_NOTE or POST_AN_UPDATE.
sharingFeatures	When sharing an item with this contact, this is a list of extra actions that the user can take. Currently, the only valid feature is ADD_CAPTION, which lets a user add a caption to an image.

Map Parameters

The parameters for glass://map are a subset of the parameters you would use from Google's Static Maps API.[1]

w	The width of the map in pixels.
h	The height of the map in pixels.
center	Center the map at this location, either with a comma-separated pair (latitude,longitude) or some unique address (for instance, Voodoo+Donuts,+Portland,+OR). If center is chosen, zoom is also needed.
zoom	Represents how closely the map is zoomed in on the center location, as a number from 0 (the entire world) to 21+ (building level). If zoom is set, center is also needed.
marker	A pin on the map, defined by a type or location. If the type is 1, indicate Glass's latest location. Otherwise, pin a latitude-longitude pair. Without center and zoom defined, the map will try and encompass the markers. You can include more than one marker.
polyline	A line on the map, generally between markers. Optionally, you can begin with a line width,color (for example, 6,0000ff), then a semicolon (;), followed by a series of comma-separated latitude,longitude points to represent a path.

HTML

Throughout this book we've skimmed over what counts as a valid HTML element. The Glass timeline-item HTML renderer works just like any other web

1. https://developers.google.com/maps/documentation/staticmaps/

browser and uses standard HTML5, with straightforward styles, but does not support the entire W3C spec.[2]

Valid HTML

Each of these elements will work as you would expect them to according to the spec.

- *Block elements*
  - *Basic text:* p, h1-h6
  - *Lists:* ol, ul, li
  - *Other:* blockquote, center, div, hr
- *Inline elements*
  - *Phrase elements:* em, strong
  - *Presentation:* big, small, strike, b, i, s, u
  - *Span:* span
  - *Images:* img
  - *Other:* br, sub, sup
- *Tables:* table, tbody, td, tfoot, th, thead, tr
- *Semantic elements:* article, aside, details, figure, figcaption, footer, header, nav, section, summary, time
- *Other:* style

Although Glass imposes its own default CSS upon these elements, the style element and style attributes allow you to override those defaults.

Invalid HTML

Any of these elements will be ignored, including all of the data inside.

- *Headers*: head, title, script
- *Frames*: frame, frameset
- *Embeds*: audio, embed, object, source, video, applet

Any other HTML element will be stripped out, but the content will be printed. For example, <address>1234 Main St</address> would just output 1234 Main St.

2. http://dev.w3.org/html5/spec/

Index

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

This Book's Home Page
https://pragprog.com/book/erpgg2
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
https://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
https://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
https://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: *https://pragprog.com/book/erpgg2*

Contact Us

Online Orders:	*https://pragprog.com/catalog*
Customer Service:	*support@pragprog.com*
International Rights:	*translations@pragprog.com*
Academic Use:	*academic@pragprog.com*
Write for Us:	*http://write-for-us.pragprog.com*
Or Call:	+1 800-699-7764