Early praise for Programming Google Glass

Eric’s writing captures the spirit of Glassware development with honesty, wit, and charm—peppered with a healthy dose of insider tricks and techniques for quickly building effective wearable computing services that coders of all levels will appreciate.

➤ Jason Salas
Coauthor, Think for Glass: Discover, Design, Develop

This insightful book opened my eyes to the potential of Google Glass in a big way. Glassware is going to be huge, and this book gives you the tools to make it.

➤ Jim Wilson
Author, Node.js the Right Way

A great read and learning resource. You start with a small project and build it into a full-fledged Glassware application, learning all about the Mirror API along the way.

➤ Steven Mitchell
Glass Explorer

I really enjoyed your book; you successfully convinced me to buy Glass and start learning as much about it as I can. It’s not like the tech is going to disappear if I ignore it!

➤ Marissa Anderson
Principal Developer, Electrozoic

This is a great introduction to the Google Glass Mirror API. Any serious Google Glass developer will want to get a copy of this book.

➤ Mark Billinghurst
Director, HIT Lab NZ, University of Canterbury
Programming Google Glass
The Mirror API

Eric Redmond

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

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

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

The team that produced this book includes:

Jacquelyn Carter (editor)
Candace Cunningham (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Copyright © 2013 The Pragmatic Programmers, LLC.
All rights reserved.
Contents

Acknowledgements . . . . . . . . . . . . . . . . . . . . vii

Preface . . . . . . . . . . . . . . . . . . . . . . . . . . ix

1. Wrapping Your Head Around Glass . . . . . . . . . . . . 1
   Getting to Know Glass 1
   Learning to Navigate 2
   Glass Hardware 7
   Glass Software 8
   Wrap-Up 9

2. The Google App Engine PaaS . . . . . . . . . . . . . . . . . 11
   Setting Up GAE 12
   Making a Web App 14
   Deploying to the Web 18
   Fancy Templates 20
   Wrap-Up 22

3. Authorizing Your Glassware . . . . . . . . . . . . . . . . . 23
   Activating Your Mirror API 24
   A Short Primer on OAuth 2.0 26
   Applying OAuth to Create Glassware 30
   Wrap-Up 36

4. Building the Timeline . . . . . . . . . . . . . . . . . . . . . . . 37
   Mirror HTTP Requests 37
   Timeline Items 41
   Multicards: Bundles and Paginating 47
   Menus 50
   Cron Jobs 52
   Wrap-Up 54
All books, even short ones like this, require efforts that extend far beyond a single author. Although I get to put my name on the cover, I want to thank everyone who helped turn a collection of drafts and ideas into a book we all hope you’ll enjoy.

Thanks first go to my indefatigable editor, Jackie Carter. Thanks, too, to Susannah, Andy, Dave, and the rest of the crew at Pragmatic for keeping this ship sailing straight.

I’m thrilled to thank my eagle-eyed friend Jim Wilson, and the piles of notes from a cadre of talented Glass Explorers: Jason Salas, Steven Mitchell, Marissa Anderson, Mark Billinghurst, Elmer Thomas, and Rahul Ravikumar.

Finally, thanks to my patient wife Noelle for enduring another book, and a preemptive thanks to Miss Wiggles for tolerating the next one.
Google Glass is the new wearable computer that everyone is talking about. Not only does Glass offer a head-mounted optical display and touch interface, but it’s also programmable in two ways: by creating native applications using the Glass Development Kit (GDK), and by programming Glassware (Glass applications) using the HTTP-based Mirror API. This book is about the latter.

You’ll get a glimpse of what Glass is and what it is not, and how users can interface with Glass. Then 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. 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?**

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. Google Glass has beaten that mark by a year.

Although Glass is sometimes wrongly 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 reliant on an external paired smartphone for some actions, such as geolocation or continuous Internet access. Internet access is a necessary component for using Glassware with the Mirror API, in the same way that Twitter is usable only if you have online access.

This is a powerful requirement, since it means your Glassware can connect users and allow them to share assets. You can store and retrieve each of your
Glass user’s information in the cloud, and easily connect the same data through Glass, smartphones, and the Web.

**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 use this book, but it can help to know your way around the syntax and Eclipse editor. You may also get more out of this book if you’re familiar with Google App Engine, although you can use any Platform as a Service (PaaS) or host your own Glassware applications.

**What’s in This Book?**

This book is about programming Google Glass by using the Mirror API, but there’s more information to know beyond the technicals of one API. We start with an overview of the Glass environment in Chapter 1, *Wrapping Your Head Around Glass*, on page 1. From there we set up our development and deployment environments in Chapter 2, *The Google App Engine PaaS*, on page 11, and follow the OAuth 2.0 steps for Chapter 3, *Authorizing Your Glassware*, on page 23.


We wrap up the book with an eye on design in Chapter 7, *Designing for Glass*, on page 83, and a look at basing Glassware on a web app in Chapter 8, *Turning a Web App to Glass*, on page 95.

This book is intended to be read linearly, from the first to last chapter. It does cover most of the Mirror API, but it’s not designed to be a reference book. However, there is a reference in Appendix 1, *HTTP and HTML Resources*, on page 107.

---

1. [https://github.com/coderoshi/glassmirror](https://github.com/coderoshi/glassmirror)
Online Resources

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

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.

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.

---

2. pragprog.com/book/erpgg/programming-google-glass
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. 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

---

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.

![Figure 1](image)

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*. 
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](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 37.

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, the blog post “Sensors on Google Glass,” the GDK documentation, and Glass tech specs. 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: 640x360 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

---

3. [http://thecodeartist.blogspot.com/2013/05/sensors-on-google-glass.html](http://thecodeartist.blogspot.com/2013/05/sensors-on-google-glass.html)
4. [https://developers.google.com/glass/develop/gdk/](https://developers.google.com/glass/develop/gdk/)
5. [https://support.google.com/glass/answer/3064128?hl=en&ref_topic=3063534](https://support.google.com/glass/answer/3064128?hl=en&ref_topic=3063534)
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 4.0.4 (API level 15), 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 11) and authorizing a user (covered in Chapter 3, *Authorizing Your Glassware*, on page 23).

**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!
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 12. 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
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.

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 this book we’ll be using the Eclipse-based Android Developer Tools (ADT) as our integrated development environment (IDE). ADT 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 ADT for your operating system.\(^1\) All examples in this book are based on the ADT that uses Eclipse version 4.2 (Juno).

The first time you launch ADT 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 ADT 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 ADT, on page 15.

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 ADT 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

---
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 16.

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
your setup looks like the one in Figure 6. Your setup should look like this, on page 17.

You should now have a project in your ADT 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 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.

```java
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.
public static String getRandomCuisine()
{
    String[] lunchOptions = {
    };
    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.

![Hello App Engine!](image)

**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](https://developers.google.com/glass/community)
3. [http://stackoverflow.com/questions/tagged/google-mirror-api](http://stackoverflow.com/questions/tagged/google-mirror-api)
Download the GAE-compatible binary FreeMarker .jar, and drag it into your project’s war/WEB-INF/lib directory.

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.

```java
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);
   }
}
```

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.
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.

```html
<!--chapter-2/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 ADT (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 37. 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.
In Chapter 2, *The Google App Engine PaaS*, on page 11, 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.

Steps for activating are as follows:
1. Click the Create Project button in Google’s Cloud Console. Provide a project name and set the project ID to match your GAE subdomain. My project looked like the preceding figure.

2. Select the APIs & Auth tab, and toggle the Google Mirror API on.

3. Select the Registered Apps tab, and click the Register App button. Name your Glassware as Lunch Roulette and ensure that Web Application is the selected platform, as you see in the following figure.

![Register new application](image)

Figure 9—Registering a new application

4. Expand the OAuth 2.0 Client ID section, and fill out the Web Origin URL to be the same HTTPS hostname as your appspot Glassware from last chapter (Chapter 2, The Google App Engine PaaS, on page 11). My test domain was https://glassbooktest.appspot.com.

5. Fill out the Redirect URI field to be the same prefix, with the path oauth2callback, such as https://glassbooktest.appspot.com/oauth2callback. Then click Generate.

6. While still under the OAuth 2.0 Client ID section, we need to add two more URLs to test our Glassware. Click the ‘+’ sign (to add a new URL) next to the Web Origin URL and set https://mirror-api-playground.appspot.com. Then add a new URL for Redirect URI to http://localhost:8888/oauth2callback.

---

1. https://cloud.google.com/console
(assuming you’ll be running your test server on the default port of 8888). Click Generate again to save, so you end up with something similar to the following figure.

![Figure 10—OAuth 2.0 Client ID content screen]

7. Still under the OAuth 2.0 Client ID 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. Expand the Browser Key section, and note the API key. We’ll use this at a later point.

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.

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.

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](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:

---

² [https://developers.google.com/oauthplayground/](https://developers.google.com/oauthplayground/)
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).

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).

```java
public class AuthFilter implements Filter {
    public void doFilter(ServletRequest req, ServletResponse res, FilterChain fc) {
```

```java
} // end AuthFilter
```
The filter must also be registered with the web application, which we'll do by adding to the GAE Glassware’s **web.xml** file.

```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 11.

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.

```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.

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


    if( authCode == null ) {
        return flow.newAuthorizationUrl().setRedirectUri( callbackUri ).build();
    }

    GoogleTokenResponse tokenRes = getTokenRes( flow, authCode, callbackUri );

    String userId = getUserId( tokenRes );

    SessionUtils.setUserId( req, userId );

    flow.createAndStoreCredential( tokenRes, userId );

    return "/";
}
```
Like with any new servlet, we map its URL path by adding its information to `web.xml`.

```xml
<chapter-3/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.

```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`:

```xml
<chapter-3/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.

```java
chapter-3/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.

```java
chapter-3/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](http://google.com/myglass) or the Android Glass app, and at the bottom of the list of applications you’ll see Lunch Roulette.

![Lunch Roulette](image)

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.
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 \textit{June 18, 1815: Napoleon loses at Waterloo}, followed by a point that reads \textit{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 \textit{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.

\textbf{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](image)

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.

```java
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 23](#) (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`.

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

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

```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.

```json
{
    "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/Ur85r0qylBQ0rj5CxBM9xX7-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.

```java
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](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](image)

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() method to send the new timeline change to the Mirror API; otherwise, you’ve just populated the object locally.

```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.

```json
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.

```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 107. 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.

```java
chapter-4/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 107).

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.

```java
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.

```java
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",
```
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.

```java
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 11 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().

```java
chapter-4/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 11, but without the html, head, and body tags that Mirror would strip out.

```html
chapter-4/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 47.

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.
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.

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.

```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](image-url)
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 49.

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 83.

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 107. 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.
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.
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 popu-
lated 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).

```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.

```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

---

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.

```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 107.
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.

```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": "2013-09-09T22:09:45Z",
  "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`.

```java
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. This is in the
same API console where you activated the Mirror API in Chapter 2, The Google
App Engine PaaS, on page 11. 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.

```
// 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/
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](image)

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.](image)

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 107.

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.

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 62). 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.

---

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 107. 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 `glassbook-test.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.
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.

```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 107.**

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 107.**

`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, 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.

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

---

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-5/src/test/book/glass/LunchRoulette.java

// Set up two menu values for DEFAULT and PENDING
List<MenuItem> menuValues = new ArrayList<MenuItem>(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.

```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.

```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.
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 through 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 imageUrl. 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-6/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" )
        .setDisplayValue( "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 calls the creator’s phoneNumber.

```
chapter-6/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-6/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 55. 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-6/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 107.

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.

```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-6/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-6/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.

![Lunch Roulette](image)

**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.

![Google App Engine logs](image)

**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](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?
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

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

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

```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.

```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-6/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 37. 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
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.

![Compliments of Lunch Roulette](image)

**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.
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.

**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 71, 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.


\(^2\) [https://developers.google.com/glass/policies](https://developers.google.com/glass/policies)
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 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).  

**Column Layouts**

In Chapter 5, *Tracking Movement and User Responses*, on page 55, 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.

```html
<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

---

3. [https://developers.google.com/glass/guidelines](https://developers.google.com/glass/guidelines)
nearby Mexican restaurants, with the cuisine type attached to a footer. The layout-two-column expects two child divs with contents.

```html
<article>
  <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](image)

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.

```html
<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.

```html
<article>
  <section>
    <h1 class="text-auto-size">List of Lunch Roulette Cuisines</h1>
  </section>
</article>
```

```html
<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>Japanese</li>
    <li>Mexican</li>
    <li>Pizza</li>
  </ul>
</article>

This will produce the following.

![List of Lunch Roulette Cuisines](image.png)

**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. 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](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. 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

---

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

```java
MenuValue defaultMV = new MenuValue()
    .setState( "DEFAULT" )
    .setDisplayNames( "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.¹

UXPin (see [Figure 27, The basic UXPin page layout, on page 91](http://uxpin.com)) 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 91](http://uxpin.com)). 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.

---

¹. [http://uxpin.com](http://uxpin.com)
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.
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.

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.

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.
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**

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 96*. 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.
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.
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, 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.

![Blog actions flow chart](image)

**Figure 32—Blog actions flow chart**

---

1. [https://github.com/coderoshi/glassmirror](https://github.com/coderoshi/glassmirror)
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 ChittrChatr 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 ChittrChatr 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 ChittrChatr 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 23.

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.
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.

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 to 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.

```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.

```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.
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.

    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.

    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 );
            // push the post to the mirror
        }
    }
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.

Glass is emerging technology. This book, along with all of the references provided by Google and other community resources birthed daily should be all you need to implement any idea that the Mirror API is capable of. 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.

2. https://developers.google.com/glass
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.
<table>
<thead>
<tr>
<th>Description</th>
<th>Method</th>
<th>Path</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert a new timeline item.</td>
<td>POST</td>
<td>/timeline</td>
<td>both</td>
</tr>
<tr>
<td>get the timeline item with the given ID.</td>
<td>GET</td>
<td>/timeline/{itemId}</td>
<td>default</td>
</tr>
<tr>
<td>list a user’s timeline items accessible by this app.</td>
<td>GET</td>
<td>/timeline</td>
<td>default</td>
</tr>
<tr>
<td>update the timeline item with the given ID.</td>
<td>PUT</td>
<td>/timeline/{itemId}</td>
<td>both</td>
</tr>
<tr>
<td>patch the timeline item with the given ID as a partial update.</td>
<td>PATCH</td>
<td>/timeline/{itemId}</td>
<td>default</td>
</tr>
<tr>
<td>delete the timeline item with the given ID.</td>
<td>DELETE</td>
<td>/timeline/{itemId}</td>
<td>default</td>
</tr>
</tbody>
</table>

### 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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The unique identifier string for this object,</td>
</tr>
<tr>
<td>kind</td>
<td>mirror#timelineItem. The type of JSON object this is, set by the mirror service resource,</td>
</tr>
<tr>
<td>isPinned</td>
<td>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),</td>
</tr>
<tr>
<td>pinScore</td>
<td>The higher the number, the closer the pinned item is to the home card. This is automatically set by the TOGGLE_PINNED callback,</td>
</tr>
<tr>
<td>isDeleted</td>
<td>This is set to true when the DELETE action is called on the timeline item, or the DELETE menu item is tapped.</td>
</tr>
<tr>
<td>inReplyTo</td>
<td>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.</td>
</tr>
<tr>
<td>etag</td>
<td>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.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>-----------</td>
<td>---------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>selfLink</td>
<td>This item as a gettable link; for example, <a href="https://www.googleapis.com/mirror/v1/timeline/1234567890">https://www.googleapis.com/mirror/v1/timeline/1234567890</a> (it still requires an Authorized header, though).</td>
</tr>
<tr>
<td>created</td>
<td>A timestamp set when the item was created.</td>
</tr>
<tr>
<td>updated</td>
<td>A timestamp that’s updated whenever the item changes.</td>
</tr>
</tbody>
</table>

### Common Fields

The following are common optional timeline items you can set.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>A readable title of this item, largely useful for debugging.</td>
</tr>
<tr>
<td>text</td>
<td>Outputs the given text to the card.</td>
</tr>
<tr>
<td>html</td>
<td>Renders the given HTML to the card.</td>
</tr>
<tr>
<td>speakableText</td>
<td>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).</td>
</tr>
<tr>
<td>speakableType</td>
<td>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.</td>
</tr>
<tr>
<td>displayTime</td>
<td>An RFC 3339–formatted date that places this item in the timeline at that instant/position.</td>
</tr>
<tr>
<td>canonicalUrl</td>
<td>A URL that points to a more full representation of the data on this item.</td>
</tr>
<tr>
<td>sourceItemId</td>
<td>A custom ID you can use to link this item with some other data that means something to your system, such as a username.</td>
</tr>
<tr>
<td>bundleId</td>
<td>A unique ID to group together cards into a bundle.</td>
</tr>
<tr>
<td>isBundleCover</td>
<td>Set to true if this is the cover card for a given group with a common bundleId.</td>
</tr>
<tr>
<td>creator</td>
<td>A JSON object representing the creator of this item, which accepts a contact object detailed in the Contact section later in this appendix.</td>
</tr>
<tr>
<td>recipients</td>
<td>An array of JSON objects representing the contacts that are meant to receive this item, useful in the REPLY_ALL user action.</td>
</tr>
<tr>
<td>Value</td>
<td>Description</td>
</tr>
<tr>
<td>------------</td>
<td>--------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><code>menuItems</code></td>
<td>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.</td>
</tr>
<tr>
<td><code>notification</code></td>
<td>A JSON object that sets how notifications are generated for this timeline item. It accepts a <code>deliveryTime</code> and a notification level (which only accepts <code>DEFAULT</code>).</td>
</tr>
<tr>
<td><code>location</code></td>
<td>Represents the location of this timeline item, populated with the same resource-body values defined in the Locations section later in this appendix.</td>
</tr>
<tr>
<td><code>attachments</code></td>
<td>An array of JSON objects representing attachments, populated with the same resource-body values defined in the Attachments section later in this appendix.</td>
</tr>
</tbody>
</table>

**MenuItem Object Fields**

These are the fields you can set as one of the `menuItem` object values under the greater timeline item body.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>id</code></td>
<td>The unique ID for this menu item.</td>
</tr>
<tr>
<td><code>action</code></td>
<td>The action that this menu item takes when tapped. It can either be one of the values under Menu Action, or CUSTOM.</td>
</tr>
<tr>
<td><code>payload</code></td>
<td>Some data whose meaning changes depending on the menu action, such as a URL for OPEN_URI or PLAY_VIDEO.</td>
</tr>
<tr>
<td><code>removeWhenSelected</code></td>
<td>If true, remove this item from the menu if it is selected (used only when the action is CUSTOM).</td>
</tr>
<tr>
<td><code>values</code></td>
<td>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).</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
<th>Required</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLY</td>
<td>Initiates a reply to send this item to the card’s creator.</td>
<td>creator</td>
</tr>
<tr>
<td>VOICE_CALL</td>
<td>Calls the card creator’s phone number (creator.phone_number).</td>
<td>creator</td>
</tr>
<tr>
<td>REPLY_ALL</td>
<td>Initiates a reply to send this item to the card’s recipients list.</td>
<td>recipients</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes this card.</td>
<td>—</td>
</tr>
<tr>
<td>SHARE</td>
<td>Shares this card with available contacts.</td>
<td>—</td>
</tr>
<tr>
<td>READ_ALOUD</td>
<td>Uses Glass text-to-speech to read either the speakableText (if set) or text field aloud, prefixed by speakableType.</td>
<td>speakableText or text</td>
</tr>
<tr>
<td>NAVIGATE</td>
<td>Opens a map showing your current location; the destination is the card’s location.</td>
<td>location</td>
</tr>
<tr>
<td>TOGGLE_PINNED</td>
<td>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.</td>
<td>—</td>
</tr>
<tr>
<td>OPEN_URI</td>
<td>If the menuItem’s payload value is a URL, opens that website in a browser window.</td>
<td>payload</td>
</tr>
<tr>
<td>PLAY_VIDEO</td>
<td>If the menuItem’s payload value is a streaming video URL, plays that video.</td>
<td>payload</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>A custom action defined by your code (more on what this means in the next section).</td>
<td>—</td>
</tr>
</tbody>
</table>

### Custom Menu Values

When a CUSTOM menuAction object is set, populate the values field with an array of objects containing these fields.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>state</td>
<td>This value applies only for the given state (DEFAULT, PENDING, CONFIRMED).</td>
</tr>
<tr>
<td>displayName</td>
<td>Text to display for this menu item.</td>
</tr>
<tr>
<td>iconUrl</td>
<td>A 50×50-pixel icon image (transparent-background PNG works best) to display alongside the menu text.</td>
</tr>
</tbody>
</table>
Timeline Attachments

The timeline resource has a subresource to manage attachments, sometimes known on the web as `timeline.attachments`.

**HTTP Methods**

<table>
<thead>
<tr>
<th>Description</th>
<th>Method</th>
<th>Path</th>
<th>Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
<td>POST</td>
<td>/timeline/{itemId}</td>
<td>upload</td>
</tr>
<tr>
<td>get the attachment under the timeline item with the given IDs.</td>
<td>GET</td>
<td>/timeline/{itemId}</td>
<td>default</td>
</tr>
<tr>
<td>list the attachment under the timeline item with the given IDs.</td>
<td>GET</td>
<td>/timeline/{itemId}</td>
<td>default</td>
</tr>
<tr>
<td>delete the attachment under the timeline item with the given IDs.</td>
<td>DELETE</td>
<td>/timeline/{itemId}</td>
<td>default</td>
</tr>
</tbody>
</table>

**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">`).

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The unique ID for this attachment.</td>
</tr>
<tr>
<td>contentType</td>
<td>The content type of this attachment, in MIME format.</td>
</tr>
<tr>
<td>contentUrl</td>
<td>When the attachment is processed, this is the available URL for the content.</td>
</tr>
<tr>
<td>isProcessingContent</td>
<td>When the attachment is processed, this will be set to true, meaning the contentUrl can be accessed.</td>
</tr>
</tbody>
</table>

**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.

<table>
<thead>
<tr>
<th>Description</th>
<th>Method</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>get the location by ID or latest.</td>
<td>GET</td>
<td>/locations/{locationId}</td>
</tr>
<tr>
<td>list a Glass device’s most recent locations.</td>
<td>GET</td>
<td>/locations</td>
</tr>
</tbody>
</table>

Read-Only Fields

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>The unique identifier string for this object.</td>
</tr>
<tr>
<td>kind</td>
<td>mirror#location. The type of JSON object this is, set by the mirror service resource.</td>
</tr>
<tr>
<td>timestamp</td>
<td>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.</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>latitude</td>
<td>Degrees latitude, where positive is north of the equator and negative is south (–180 to 180)</td>
</tr>
<tr>
<td>longitude</td>
<td>Degrees longitude, where positive is east of the prime meridian and negative is west (–180 to 180)</td>
</tr>
<tr>
<td>accuracy</td>
<td>The accuracy of Glass’s location, in meters</td>
</tr>
<tr>
<td>displayName</td>
<td>The name of this location (if named), such as a business name</td>
</tr>
<tr>
<td>address</td>
<td>The complete address, including street, city, state, country, zip, and so on</td>
</tr>
</tbody>
</table>

Subscriptions

The subscription resource supports most CRUD operations as RESTful HTTP methods, except for getting a single subscription.
HTTP Methods

<table>
<thead>
<tr>
<th>Description</th>
<th>Method</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert a new subscription</td>
<td>POST</td>
<td>/subscriptions</td>
</tr>
<tr>
<td>list a user’s subscriptions accessible by this app</td>
<td>GET</td>
<td>/subscriptions</td>
</tr>
<tr>
<td>update the subscription with the given ID</td>
<td>PUT</td>
<td>/subscriptions/{subscriptionId}</td>
</tr>
<tr>
<td>delete the subscription with the given ID</td>
<td>DELETE</td>
<td>/subscriptions/{subscriptionId}</td>
</tr>
</tbody>
</table>

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.

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>REPLY</td>
<td>Replied to the creator of the item, which created a new timeline item</td>
<td>INSERT</td>
</tr>
<tr>
<td>REPLY_ALL</td>
<td>Replied to all recipients of an item, which created new timeline items for each recipient</td>
<td>INSERT</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deleted this item from the timeline</td>
<td>DELETE</td>
</tr>
<tr>
<td>SHARE</td>
<td>Shared the item with available contacts</td>
<td>UPDATE</td>
</tr>
<tr>
<td>PIN</td>
<td>Pinned the item to the left of the home card</td>
<td>UPDATE</td>
</tr>
<tr>
<td>UNPIN</td>
<td>Unpinned the item back into the main timeline</td>
<td>UPDATE</td>
</tr>
<tr>
<td>LAUNCH</td>
<td>Updated the timeline item via voice command</td>
<td>UPDATE</td>
</tr>
<tr>
<td>CUSTOM</td>
<td>Chose a custom menu item, and accompanies a payload</td>
<td>UPDATE</td>
</tr>
</tbody>
</table>

---

### 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.

<table>
<thead>
<tr>
<th>Description</th>
<th>Method</th>
<th>Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert a new contact.</td>
<td>POST</td>
<td>/contacts</td>
</tr>
</tbody>
</table>
### Description | Method | Path
---|---|---
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}
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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
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

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
</table>
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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>w</td>
<td>The width of the map in pixels.</td>
</tr>
<tr>
<td>h</td>
<td>The height of the map in pixels.</td>
</tr>
<tr>
<td>center</td>
<td>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.</td>
</tr>
<tr>
<td>zoom</td>
<td>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.</td>
</tr>
<tr>
<td>marker</td>
<td>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.</td>
</tr>
<tr>
<td>polyline</td>
<td>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.</td>
</tr>
</tbody>
</table>

---

¹ [https://developers.google.com/maps/documentation/staticmaps/](https://developers.google.com/maps/documentation/staticmaps/)
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 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/](http://dev.w3.org/html5/spec/)
Open GL and Android on Android

Dive into OpenGL on Android, and script your Android device right on the device.

OpenGL ES 2 for Android

Android is booming like never before, with millions of devices shipping every day. It’s never been a better time to learn how to create your own 3D games and live wallpaper for Android. You’ll find out all about shaders and the OpenGL pipeline, and discover the power of OpenGL ES 2.0, which is much more feature-rich than its predecessor. If you can program in Java and you have a creative vision that you’d like to share with the world, then this is the book for you.

Kevin Brothaler
(346 pages) ISBN: 9781937785345. $38

Developing Android on Android

Take advantage of the open, tinker-friendly Android platform and make your device work the way you want it to. Quickly create Android tasks, scripts, and programs entirely on your Android device—no PC required. Learn how to build your own innovative Android programs and workflows with tools you can run on Android itself, and tailor the Android default user interface to match your mobile lifestyle needs. Apply your favorite scripting language to rapidly develop programs that speak the time and battery level, alert you to important events or locations, read your new email to you, and much more.

Mike Riley
(232 pages) ISBN: 9781937785543. $36
Tinker, Tailor, Solder, and DIY!

Get into the DIY spirit with Raspberry Pi or Arduino. Who knows what you’ll build next...

Raspberry Pi

The Raspberry Pi is a $35, full-blown micro computer that runs Linux. Use its video, audio, network, and digital I/O to create media centers, web servers, interfaces to external hardware—you name it. And this book gives you everything you need to get started.

Maik Schmidt
(149 pages) ISBN: 9781937785048. $17
http://pragprog.com/book/msraspi

Arduino

Arduino is an open-source platform that makes DIY electronics projects easier than ever. Even if you have no electronics experience, you’ll be creating your first gadgets within a few minutes. Step-by-step instructions show you how to build a universal remote, a motion-sensing game controller, and many other fun, useful projects. This book has now been updated for Arduino 1.0, with revised code, examples, and screenshots throughout. We’ve changed all the book’s examples and added new examples showing how to use the Arduino IDE’s new features.

Maik Schmidt
(272 pages) ISBN: 9781934356661. $35
The Joy of Math and Healthy Programming

Rediscover the joy and fascinating weirdness of pure mathematics, and learn how to take a healthier approach to programming.

Good Math

Mathematics is beautiful—and it can be fun and exciting as well as practical. *Good Math* is your guide to some of the most intriguing topics from two thousand years of mathematics: from Egyptian fractions to Turing machines; from the real meaning of numbers to proof trees, group symmetry, and mechanical computation. If you’ve ever wondered what lay beyond the proofs you struggled to complete in high school geometry, or what limits the capabilities of the computer on your desk, this is the book for you.

Mark C. Chu-Carroll
(282 pages) ISBN: 9781937785338. $34
http://pragprog.com/book/mcmath

The Healthy Programmer

To keep doing what you love, you need to maintain your own systems, not just the ones you write code for. Regular exercise and proper nutrition help you learn, remember, concentrate, and be creative—skills critical to doing your job well. Learn how to change your work habits, master exercises that make working at a computer more comfortable, and develop a plan to keep fit, healthy, and sharp for years to come.

*This book is intended only as an informative guide for those wishing to know more about health issues. In no way is this book intended to replace, countermand, or conflict with the advice given to you by your own healthcare provider including Physician, Nurse Practitioner, Physician Assistant, Registered Dietician, and other licensed professionals.*

Joe Kutner
(254 pages) ISBN: 9781937785314. $36
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
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://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
http://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: http://pragprog.com/book/erpgg

Contact Us

Online Orders: http://pragprog.com/catalog
Customer Service: support@pragprog.com
International Rights: translations@pragprog.com
Academic Use: academic@pragprog.com
Write for Us: http://pragprog.com/write-for-us
Or Call: +1 800-699-7764