Extracted from:

Programming Google Glass

The Mirror API

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Pragmatic exPress

Programming Google Glass

The Mirror API

Eric Redmond

Edited by Jacquelyn Carter

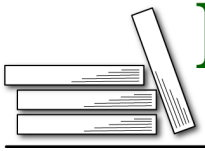
Programming Google Glass

The Mirror API

Eric Redmond

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

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)

Printed in the United States of America.
ISBN-13: 978-1-937785-79-6
Encoded using the finest acid-free high-entropy binary digits.
Book version: P1.0—December 2013

CHAPTER 4

Building the Timeline

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

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

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

Mirror HTTP Requests

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

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

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

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

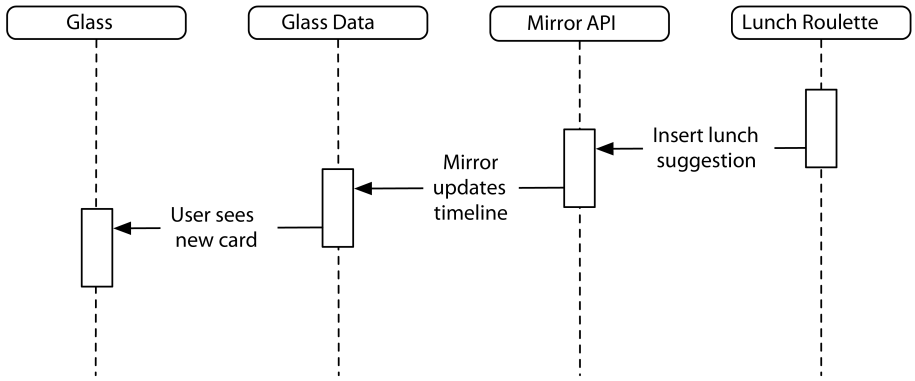


Figure 13—Flow of Glassware to Glass

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

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

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

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

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

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

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

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

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

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

chapter-4/src/test/book/glass/MirrorUtils.java

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

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

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

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

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

chapter-4/src/test/book/glass/LunchRoulette.java

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

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

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

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

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

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

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

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

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

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

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

Figure 14—Lunch Roulette card

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

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