Extracted from:

Programming Google Glass, Second Edition

Build Great Glassware Apps with the Mirror API and GDK

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

Programming Google Glass

Second Edition

Build Great Glassware Apps with the Mirror API and GDK

Eric Redmond

Edited by Jacquelyn Carter

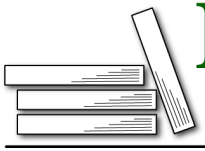
Programming Google Glass, Second Edition

Build Great Glassware Apps with the Mirror API and GDK

Eric Redmond

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina



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*.

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

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

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.

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

OAuth Location Scope

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

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

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

chapter-05/LunchRoulette/src/test/book/glass/auth/AuthUtils.java
```
public static final List<String> SCOPES = Arrays.asList(
  "https://www.googleapis.com/auth/userinfo.profile",
  "https://www.googleapis.com/auth/glass.timeline",
  "https://www.googleapis.com/auth/glass.location"
);
```

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

One Location

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

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

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

chapter-05/LunchRoulette/src/test/book/glass/LunchRoulette.java
```
Location location = mirror.locations().get("latest").execute();

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

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

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

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

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

Many Locations

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

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

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

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

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

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

```
   "accuracy": 49.0
  }
 ]
}
```

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

Using Location

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

Navigating to a Location

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

The code necessary to search Google Places for a nearby cuisine is wrapped up into a PlaceUtils class as part of the book's downloadable code. The getRandom() method chooses a random restaurant by searching for a nearby cuisine at a given latitude and longitude location.

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

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

```
// get a nearby restaurant from Google Places
Place restaurant = getRandomRestaurant(latitude, longitude);
// create a timeline item with restaurant information
TimelineItem timelineItem = new TimelineItem()
```

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

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

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

Figure 19—Getting directions

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

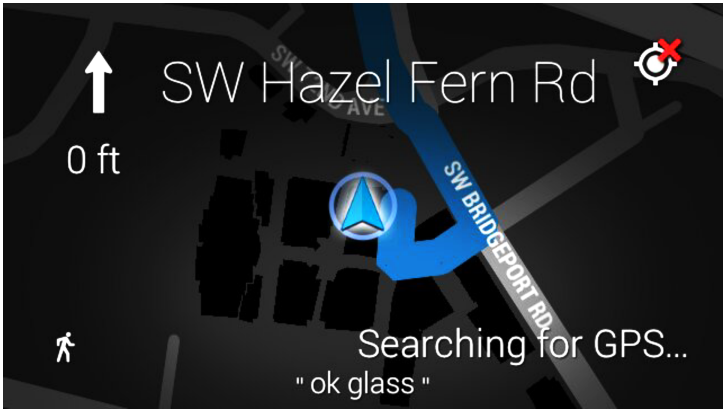


Figure 20—Navigation shows a map.

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

Showing Your Own Map

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

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

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

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

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

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

When you rerun the project, you should notice a dramatic change on your Glassware card (see Figure 21, *Displaying a map alongside a restaurant*, on page 10). No longer are we stuck with plain text, or slightly less-plain HTML.

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

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

Figure 21—Displaying a map alongside a restaurant