

Designing & Developing for Google Glass

THINKING DIFFERENTLY FOR A NEW PLATFORM

Allen Firstenberg & Jason Salas

Designing & Developing for Google Glass

Creating apps for Google Glass is more involved than simply learning how to navigate its hardware, APIs, and SDK. You also need the right mindset. While this practical book delivers the information and techniques you need to build and deploy Glass applications, it also helps you to think for Glass by showing you how the platform works in, and affects, its environment.

In three parts—Discover, Design, and Develop—Glass pioneers guide you through the Glass ecosystem and demonstrate what this wearable computer means for users, developers, and society as a whole. You'll learn how to create rich functionality for a consumer technology that's radically different than anything currently available.

- Learn the Five Noble Truths of great Glassware design
- Understand the Glass ecosystem and learn why it's different
- Sidestep Glass's societal concerns in your projects
- Learn how Glass adapts to the user's world, rather than the other way around
- Avoid poor design by identifying Glassware antipatterns
- Build cloud services with the Google Mirror API
- Use the Glass Development Kit to develop client applications
- Submit your project for review in the MyGlass directory

“Few in this developing field can be called masters, but Allen and Jason are worthy of the label. Going far beyond Google Glass as a specific product, this book captures the essence of designing for smart glasses of all varieties.”

—Eric Redmond

Author of *Programming Google Glass*
(Pragmatic Bookshelf)

Allen Firstenberg, Senior Project Engineer at Objective Consulting, Inc., has been instrumental in creating websites and mobile apps for several companies and organizations, including the National Science Foundation. Allen is a Google Developer Expert for Wearables.

Jason Salas is a software developer, marketer, broadcaster, sportswriter, and filmmaker. In the past few years, he's been concentrating on Google Glass, Hangout extensions, HTML5 games, and Chromecast apps.

PROGRAMMING/MOBILE/WEARABLES

US \$49.99

CAN \$52.99

ISBN: 978-1-491-94645-9



Twitter: @oreillymedia
facebook.com/oreilly

Praise for *Designing and Developing for Google Glass*

“For the first time ever we have a piece of technology that is designed for the user instead of a screen. Google Glass has changed the landscape of technology and is forcing us to rethink not only the way we develop, but just as importantly the way we design apps. Allen and Jason have captured both design and development of Glass in their book expertly. Equal emphasis has been placed on both important aspects of Glass and has made this book an asset to the tech community.

Upon completion of the book the reader will be fully engaged in how to think for Glass, the new way to compute.”

— *Katy Kasmai*
founder of UbiTech.co

“Jason and Allen are trailblazers when it comes to development and design for Google Glass. There aren’t many authors with the level of experience as Jason and Allen. The unique approach to designing for wearable electronics makes this book required reading for any aspiring and proficient developer looking to take the plunge in designing and developing for Glass.

I am honored to call them mentors and friends.”

— *Noble Ackerson*
Cofounder/CEO of LynxFit

“Allen and Jason are both talented developers as well as experts on Google Glass. This is a must-read for anyone interested in working on a Glass app.”

— *Libby Chang*
Society of Glass Enthusiasts San Francisco / Cohost of
Wearables Weekly

“Few in this developing field can be called masters, but Allen and Jason are worthy of the label. Going far beyond Google Glass as a specific product, this book captures the essence of designing for smart glasses of all varieties.”

— *Eric Redmond*
Author of *Programming Google Glass*

Designing and Developing for Google Glass

Allen Firstenberg and Jason Salas

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'REILLY®

www.allitebooks.com

Designing and Developing for Google Glass

by Allen Firstenberg and Jason Salas

Copyright © 2015 Allen Firstenberg and Jason Salas. All rights reserved.

Printed in the United States of America.

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (<http://safaribooksonline.com>). For more information, contact our corporate/institutional sales department: 800-998-9938 or corporate@oreilly.com.

Editor: Rachel Roumeliotis

Production Editor: Nicole Shelby

Copyeditor: Charles Roumeliotis

Proofreader: Kim Cofer

Indexer: Ellen Troutman

Cover Designer: Ellie Volckhausen

Interior Designer: David Futato

Illustrator: Rebecca Demarest

December 2014: First Edition

Revision History for the First Edition:

2014-12-08: First release

See <http://oreilly.com/catalog/errata.csp?isbn=9781491946459> for release details.

The O'Reilly logo is a registered trademark of O'Reilly Media, Inc. *Designing and Developing for Google Glass*, the cover image, and related trade dress are trademarks of O'Reilly Media, Inc.

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 O'Reilly Media, Inc. was aware of a trademark claim, the designations have been printed in caps or initial caps.

While the publisher and the authors have used good faith efforts to ensure that the information and instructions contained in this work are accurate, the publisher and the authors disclaim all responsibility for errors or omissions, including without limitation responsibility for damages resulting from the use of or reliance on this work. Use of the information and instructions contained in this work is at your own risk. If any code samples or other technology this work contains or describes is subject to open source licenses or the intellectual property rights of others, it is your responsibility to ensure that your use thereof complies with such licenses and/or rights.

ISBN: 978-1-491-94645-9

[LSI]

This book is dedicated to every Glass Explorer who dared to take the plunge.

Allen Firstenberg

Thirty years ago, my parents gambled on getting me a computer, thinking it might play a part in my future. Little did they know where it would lead, and I would like to thank them for the opportunities they set in motion all those years ago. For my closest friends and colleagues, who always reminded me that I can do it, even when I doubted myself the most, I am thankful for the support you've given me over the years. Finally, I'd like to dedicate this book to my son—as I watch you explore the world, I'm thankful I can share those moments with you through Glass, and I hope you never lose that sense of discovery.

My special thanks to Jason, my coauthor, collaborator, benevolent taskmaster, and friend, for his work in getting the book off the ground and for serving as the sounding board for all of my design ideas. It has been my honor to work with you, and I look forward to the day when we can meet in the same time zone.

Jason Salas

For Sharon Strandskov, whose strength and spirit have inspired me in more ways than anyone else ever could; for Will Ymesei, who's endured more of my spur-of-the-moment, rapid fire "I have an idea!" conversations than anyone ever should; for Mom and Dad, who've shown me more support than anyone ever would; and for Ali Bales, whose encouragement, patience, friendship, and love couldn't be more constant or genuine—and provided all the tools I ever needed to get this project done.

Table of Contents

Prologue.....	xiii
Preface.....	xv

Part I. Discover

1. The Revolution Will Be Wearable.....	3
Forging Glass	3
Wearable Computing	4
What Does It Mean to Think for Glass?	5
2. The Glass Ecosystem: What It Is and How It Is Different.....	9
What You See and What You Get	9
Glass Is a Platform, Not a Product	10
The Glass Application Model	12
Mirror API	12
Glass Development Kit (GDK)	13
Actions, Not Apps	14
Reinventing Human–Computer Interaction	16
The Science Behind the Projection	18
How Glass Gets Audio into Your Ear	20
Using the System	20
The Camera: Photos, Videos, and More!	22
Glass Is a Great Listener	23
Content Creation in a POV World	28
Which Hue Is for You?	29
Welcome to Wearable Computing!	29
3. Societal Issues with Glass and How to Avoid Them in Your Projects.....	31
Issue #1: Privacy	32

Where Are We Now?	33
Think for Glass	34
Issue #2: Facial Recognition	34
Where Are We Now?	35
Think for Glass	36
Issue #3: Using Glass While Driving	37
Where Are We Now?	37
Think for Glass	38
Issue #4: Aesthetic Appeal—Is Glass Fashionable?	38
Where Are We Now?	39
Think for Glass	40
Issue #5: Augmented Reality	40
Where Are We Now?	40
Think for Glass	41
Issue #6: Glass Analytics	42
Where Are We Now?	42
Think for Glass	43
Issue #7: Regulatory Environment—Glass and Public Policy	43
Where Are We Now?	44
Think for Glass	45
The Business of Producing Glassware	45
What Glass Isn't	46

Part II. Design

4. Thinking for Glass: How Glass Is, and Should Be, Personal.....	51
Glass as Personal Technology	51
Best in Show	52
Twitter	53
Gmail	56
Google+	58
Winning Glassware Design Takeaways	61
Designing with the Think for Glass Mindset	63
Vignettes	64
Google Now	65
Google Search	68
Glass for Gaming	71
Design for the Cloud	74
Are You Starting to Think for Glass?	74

5. The Five Noble Truths of Great Glassware Design.....	77
Noble Truth 1: Design for Glass	78
Targeting Microinteractions	78
Tactical Wearable Design	80
Don't Neglect Audio	81
Delete Versus Dismiss	82
Provide Web-Based Configuration	83
Noble Truth 2: Don't Get in the Way	84
Notify Responsibly	84
Less Is More	86
The Exceptions Make the Rule	87
Noble Truth 3: Keep It Relevant	87
Context FTW	88
How Soon Is Now?	89
Noble Truth 4: Avoid the Unexpected	90
Content	90
Performance	91
Don't Be a Bandwidth Hog	91
Permissions	93
Error Handling	94
Synchronization Across Platforms	96
Surprises Should Be Pleasant Surprises	97
Noble Truth 5: Build for People	98
Advocate Multitasking	99
Glass Is Naturally Social	99
So...That's It? Really?	101
6. Glassware Antipatterns: Avoiding Poor Design.....	103
Improperly Implementing Ideas for the Glass Experience	104
Treating Glass Like Any Other Mobile Device	105
Overloading the System AND the Wearer	106
Think in Actions, Not in Apps	107
Stick to the Prefab Templates and Styles	109
Don't Use the Prism Display as a Stage for Complicated Reports	110
Glass Isn't Necessarily Bound to Your Phone	110
Unrealistic Expectations for Augmented Reality and Gaming	111
Don't Deviate from Default: Using Categorical Voice Commands	112
Not Fully Utilizing Cloud Computing	113
Choosing the Wrong Development Framework for Your Glassware Project	114

Part III. Develop

7. Overview of the Mirror API.....	121
Life on the Server Side	122
Events: The Building Blocks of the Glass Timeline	125
The High-Level View	125
How Your Server Talks with Google	128
Components of the Mirror API	130
Preparing Your Project	131
The Glass Ceiling—Your Project’s Quota	134
The Even Bigger Picture	136
8. Security and OAuth.....	137
Event Security: Google the Bouncer	137
OAuth: IDs and Secrets	138
Will You Come and Join the Dance?	139
Who Are You? I Am the New Number Two	145
What Do You Want? Information	146
Disabling (and Reacquiring) Permissions	149
Who Is Number One? You Are Number Six	149
WAKE UP!!!	151
9. Working with Timeline Cards.....	153
“Hello, World!”, Glass-Style	153
HTML: Even More Style	158
What About Images?	170
Working with Mosaics	174
Rendering an In-Card Map	178
Simple Audio	181
Bundles of Fun	183
Going Beyond the Playground	193
Media Matters	196
Oh, CRUD...	197
10. Card Actions and Subscriptions.....	199
Simple Event Actions	199
Listen Up!	202
collection	204
userToken	204
verifyToken	204
callbackUrl	205
Responding to Subscription Pings	205

Simple Callbacks and How to Handle Them	206
Custom Menu Items	209
Keeping in Contact	216
11. Sharing Resources with Glassware.....	221
The Share Menu Item	221
Share Contacts	222
Voice Commands	230
12. Context Is King: Using Location and Other Signals.....	233
Enabling Location	234
Where Do You Think I Am?	234
Location as Part of Timeline Events	236
Setting Things Straight	237
Location Becomes Localization	238
Subscribing to Locale Changes Saves API Calls	239
Other Contextual Signals	240
Context and the Future	241
13. The GDK.....	243
Installed Apps Running on Glass	243
What Is the GDK?	245
How the GDK Differs from the Mirror API	247
User Interface Elements of GDK Apps	249
Live Cards	249
Two Flavors of Live Cards	250
Immersion	253
More Tools for Rapid Design	255
It Was Native All Along!	257
The GDK Object Model	259
Packages	259
System Intents	259
On-Head Detection Halts Running Apps, Too	260
Hybrids: The Ultimate Glassware Challenge (and Experience!)	261
Authentication	263
Writing Native Code for Glass	265
Testing Native Glass Applications	267
A View to a Card	268
Basic Text Formatting	269
Creating Rich Text	272
Ellipses and Excess Content	274
Columnar Layouts and Mosaics	275

Using Icons	276
Other Neat Templates	279
When You Have No Choice—Doing It Yourself	283
Configuring Voice Commands	284
With Voice Commands, Google Has the Final Word	287
Updating Releases, Versioning, and Crash Reports	287
Porting Existing Apps to Glass: DON'T	288
So Which Framework Is for Me?	289
14. Getting on MyGlass: Glassware Submission, Review, and Distribution.	291
Making Your Awesome Glassware Even More Awesome	291
The Objective of Glassware Review	292
What MyGlass Gets You	292
Prereview Activities	293
Things to Think About Before You Submit	294
Submitting Your Glassware	296
Gotchas	298
Submit!	299
The Review Process	301
Timing Your Release	303
Categorical Listings	303
App Analytics	303
Marketing Channels	304
Monetization	304
15. Reflections on the Future.	307
Corporate Glass	307
Streamlining Operations	308
Glass in Medicine and Education	309
Accessibility	311
Home Integration	311
Chromecast and Home Entertainment	312
Android Wear	313
Hardware Hacking and the Internet of Things	313
Peripherals/Accessories	314
In Closing	315

Part IV. Appendices

A. Glassware Done Right: Case Studies from the Field.	319
--	------------

B. Hacking Glass.....	349
Index.....	357

Prologue

Most logical people would probably come to the conclusion that attempting to write a book for a product not yet released to market and with next to no public information available about it is, well, how can we best say this...stupid. But that's precisely the challenge we took on when proposing this book to O'Reilly, as an extension of our faith and optimism about the exciting new realm of wearable computing and the implementation of it within Google Glass.

Writing this material was a particularly challenging project, to say the least. We were tasked with not only authoring a book within an incredibly compressed timeframe, but with extremely little information about the new platform with which to work. Google's steadfast commitment to not letting details about the product leak out made getting details about timelines, specifications, and API capabilities quite tough. But our symbiotic optimism about Glass and our unique dichotomy—Allen being a coder who can write, Jason a writer who can code—carried us through this labor of love.

By the same token, the staggered way that information was disseminated as the product neared its glorious launch made writing this book a constant torrent of rewrites, tweaks, updates, and deletions. Written over a period of nearly two full years, the book has been expanded from its original 12 chapters to as many as 19, then pared back down to 15.

Due to numerous evolutions, several of the chapters have effectively been completely rewritten numerous times. And with information made public teetering back and forth between topics governing marketing, engineering, technical specs, and product philosophy, it made for a wildly anachronistic writing process.

But that, as they say, was half the fun.

At the end of the day, we wrote this book as much for our own edification as we did for the Glass community, in its current membership and with scores more to come.

For just as Googlers (in)famously have their “dogfooding” practice of internally testing new technologies, we wanted to ensure we knew exactly what Glass was all about and what it can do for us as well as for society and bring that to the people.

We hope you thoroughly enjoy this book as much as we did putting it together!

Allen and Jason

A New (R)evolution in Computing

Google Glass has taken the popular imagination by storm like few other technologies have, potentially being the most transformative consumer device since the iPad...*and it did so long before its commercial release*. Scores of people have shown ardent interest in learning as much as they could about the mysterious machine and the ecosystem that encapsulates it, and many flocked to develop software for it in order to capitalize on its next-gen model of information delivery. Many producers of mobile services beat down the door to learn how to repurpose their existing data and codebases to establish presence for their applications in this new space.

And a significant number of people were hesitant to embrace Glass due to concerns with privacy, legality, health issues, and the personal image using it conveys.

This book effectively addresses all these issues by presenting a simple philosophy we've been heavily promoting to Glass enthusiasts: *Think for Glass*. While many forthcoming books will discuss coding conventions, design patterns, and development idioms, and will do so very well, our work presents a holistic view of Glass not only as hardware, but as a proper computing platform requiring the adoption of a new mindset for users to truly appreciate the product and with which architects can build maximum value.

Like many of you, we were captivated by our first glimpses of Google Glass in 2012. We immediately knew that Glass was going to be different, more than just a cell phone on our faces, and we wanted to help others understand how it would change our lives. In a series of Hangouts, at first, and later through presentations and helping in the various communities, we shaped our core philosophies of Glass. In this book, we hope to share those philosophies with you.

As two passionate members of the Google development community, each of us an experienced programmer, writer, and Glass enthusiast, we hope this book will serve as a primer to the device, an overview of the platform's guiding principles, and a broad introduction to writing Glassware.

What We’re Bringing to the Table

What we need to state up front is that this isn’t just a “How to code for Google Glass” book—we wholeheartedly believe it’s much much more than that. Rather than merely copy and paste Google’s API documentation and run through the general environment for programming, testing, and distributing Glass applications, we seek to educate you with a philosophy centered on awareness of the Google Glass ecosystem—getting you to Think for Glass. By this merit, you’ll have a better understanding about the approach necessary to get the most out of this revolutionary new computing platform.

We’ll be looking at what Glass means for users, developers, and society as a whole.

This book is organized to be flexible to your interests while still presenting the material in a logical fashion to teach you all about Glass. We’ve essentially organized the material as a three-act play. While some of the more technical sections build upon concepts that preceded them in earlier chapters, the topics are largely arranged in such a way that they can be consumed completely out of order without losing focus, giving you the freedom to jump to the discussion that intrigues you most, or skip over stuff that doesn’t pique your interest. New Glass owners (or those of you contemplating taking the leap) will particularly enjoy the first few chapters, which deal with how to properly *Discover* the platform.

Maybe you just want to be one of the cool kids and talk to your peers with deep expertise about what Glass is and how they can use it. Living on the bleeding edge is totally awesome, and we’ve written [Chapter 2](#) just for that reason.

If you’re a designer who wants to create for the hottest UI to hit the consumer tech industry in years, you’ll want to make sure you read the *Design* section. We’ll be talking about the Glass aesthetic and how to compose effective designs that maximize data in a minimal display, how the experience is fundamentally different than designing for other platforms, and what we mean when we say you should “Think for Glass.” Before you dive into code, you’ll want to consider [Chapters 4](#) through [6](#) to make sure that what you want to do works well with the people who use Glass.

If you’re a programmer who’s heard the buzz about wearable computing, you’ll then want to continue to the third part of the book, *Develop*. In this section, it may behoove you to read [Chapters 7](#) through [15](#) in order, as several of the concepts we’re presenting have a certain sequence. We begin that ambitious quest in [Chapter 7](#), where we drill down into program structure, syntax, and the architecture that governs how RESTful Glassware works with the Google Mirror API. We also help those of you maintaining existing software platforms if you’re keen on integrating with Glass to get a leg up on the other guys. If you’re an Android developer, and you’re interested in exploring the similarities and differences when programming for Glass with Android framework code, head on over to [Chapter 13](#) where we talk about the Glass Development Kit.

Or if you've been itching to create a slick mashup with some third-party web APIs and want to corner a very emerging space, we've got you covered, too—check out the latter chapters for insight.

And simply starting from page 1 and running through the entire book will give you a well-rounded Glass education in order to understand the Glass ecosystem and build effective Glassware. Being able to Think for Glass means that you have to understand the system in total. You'll truly grasp what the system is all about and how to best make it work for you once you're done reading.

And at the end of the book, we've included a helpful appendix that focuses on hacking Glass. We also feature “Glassware Done Right,” a series of case studies that highlight noteworthy third-party services and applications that are top-shelf examples of programs written specifically for the Glass experience that really get the job done. Some are simple, some are complex, but they're all outstanding and all really fun to use, with each illustrating a positive lesson to follow about how to implement a specific software idea or challenge with this new model.

What we want to avoid at all costs is presenting you with content that just lets you rip code and not really comprehend what the Glass experience truly is. We don't want to give rise to a generation of sloppily crafted apps that wind up being force-fit versions of existing services that don't fully utilize the best aspects of the platform the way they were intended. Glass succeeds beautifully on both sides of the product development coin—form and function—so having mastery of this new toolset is critical.

As you'll see, Glass isn't purely another client-server web system or just the latest fork of Android. The input system, UI/UX, networking capabilities, and application framework—it's an entirely different animal than what you've been used to. Everything about the ecosystem is a radical departure from traditional methods of distributed computing. A new paradigm requires a new mindset—and that's precisely what we're giving you. Even experienced developers should appreciate the effective simplicity of Glass's mechanics for data delivery. This *personal area network* of sensors and displays provide the wearer with the information they want in exactly a form that's extremely convenient, contextually relevant, and highly effective for them as they go about their day with their digital lifestyle, as well as doing things in the real world.

We feel that empowering you with the ability to explain, explore, and exploit Glass, confidently and competently, is invaluable because then you can teach others, too! Pay it forward, baby. Spread the gospel. Share the love.

Maybe you can charge people for your expertise as a Glass consultant and recoup your investment for buying this book. *Viva capitalism!*

How This Book is Organized

Discover

Chapter 1, The Revolution Will Be Wearable

We explore the fundamental question, “Why Glass?” What is Google trying to solve by creating the product, and why do they think people will want to buy it? What underlying technologies are going into it? What is new or different about Glass, and what is the same as what we have seen before? Why is this book a good way to learn all these things? Most importantly, we set the groundwork for the three major sections of the book that follow and start to guide the different types of readers into which parts might be most interesting.

Chapter 2, The Glass Ecosystem: What It Is and How It Is Different

This chapter is the core of educating people to what Glass currently is and the foundation for exploring how Glass apps should be designed. We go over the specifications of Glass. We highlight why the hardware represents a balance between user needs, aesthetics, and design constraints. In particular, we will focus on how this is a product unto itself—not a peripheral or accessory to something else. In conjunction with the hardware, we start to delve into how people use Glass, the timeline, and the software driving it.

Chapter 3, Societal Issues with Glass and How to Avoid Them in Your Projects

It is important, in both educating people about how to use Glass and what to expect from Glass applications, to make sure we are clear about what Glass isn’t and what it can’t do...yet. Although we will go into detail about this as part of designing and developing apps, it is important to take some time out and discuss false impressions that have already built up. In particular, we will discuss some of the controversial issues surrounding Glass—privacy, control, intrusiveness, aesthetics, and whether this is true augmented reality.

Design

Chapter 4, Thinking for Glass: How Glass Is, and Should Be, Personal

We dive into the meat of the aesthetic philosophy for Glass. Much like the change in thinking that took place when people moved from text to graphic programs, then later to web design, and still later when moving from web to mobile apps, writing for Glass requires a totally new way of thinking. Glass is, in some ways, more personal and intimate than a smartphone, and people will be interacting with Glass very differently. Where existing examples do not exist, we will hypothesize other illustrative apps, showing how they can (or in some cases can’t) have counterparts in Glass and how they needed to be rethought.

Chapter 5, *The Five Noble Truths of Great Glassware Design*

We solidify the design for Glass with The Five Noble Truths, based on the initial guidelines presented by Google, and how they were ultimately expanded. Each of these guidelines will be complete with good and bad design examples that follow the guidelines.

Chapter 6, *Glassware Antipatterns: Avoiding Poor Design*

The natural inclination of people will be to think about developing for Glass as either being “just like” developing for other mobile platforms, or expect it to be tied to a mobile device. We’ll discuss why these perceptions may cause problems. We’ll also reiterate some of the design constraints in the system and how it constrains some of the applications that are available.

Develop

Chapter 7, *Overview of the Mirror API*

We present the prerequisites for developing for Glass using the RESTful Google Mirror API. We will also discuss how the Mirror API is language-agnostic but requires an understanding of how to build and host web apps. We’ll touch on Google App Engine as a container while emphasizing other available options. Finally, we’ll discuss the infrastructure for how Glass will communicate with your app and how we will present each of the concepts.

We talk about concepts, with a couple of language-specific code examples in some cases, and make sure people understand how those concepts will map to the Google-provided documentation and the language-specific APIs. Think of this entire section as providing a way to help you translate between Google’s documentation and your language experience.

Chapter 8, *Security and OAuth*

We talk about how to authenticate users with OAuth 2.0 and authorize services to communicate with Glass on their behalf.

Chapter 9, *Working with Timeline Cards*

The fundamental UI aspect of Glass is the timeline card, which we’ll have seen illustrated in earlier chapters. We’ll now learn more details about these cards, how they are bundled together, how Google helps you manage them, and how you can format them. Some of the tools that Google provides to help design them will also be reviewed, along with the various parameters and procedures involved in creating, editing, and deleting them.

Chapter 10, *Card Actions and Subscriptions*

It wouldn't be a very interesting program if it couldn't handle input from users. We'll discuss how cards can have actions attached to them and how your program needs to define what actions are available, accept these actions, and verify that they are authentic responses to your cards.

Chapter 11, *Sharing Resources with Glassware*

Beyond attaching actions to your own cards, you can also allow photos, videos, and other resources captured through Glass to be shared with your application. We'll discuss how you need to handle this data and what Google does to help you with it.

Chapter 12, *Context Is King: Using Location and Other Signals*

The curious topic of the use of geodata within the Glass ecosystem will be detailed here, alongside other forms of input for contextual computing, like identity, time, and activities. Particular attention will be given to the way Glass handles location data updates via the Mirror API.

Chapter 13, *The GDK*

We provide an overview of the SDK libraries that constitute the Glass Development Kit that makes writing Android framework code possible, giving Android programmers the ability to create installed applications for Glass.

Chapter 14, *Getting on MyGlass: Glassware Submission, Review, and Distribution*

Getting seen on MyGlass should be the destination you aim for after building an amazing product, as it yields the greatest rewards. You'll learn how to prepare for Google's review of your projects, how to be approved quickly, and then how to distribute and promote your Glassware to the masses.

Chapter 15, *Reflections on the Future*

This discussion addresses some more advanced topics and issues relative to the progression of the platform, how Glass might be used from an organizational standpoint, and how the Glass ecosystem can be extended.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

Constant width

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

Constant width bold

Shows commands or other text that should be typed literally by the user.

Constant width italic

Shows text that should be replaced with user-supplied values or by values determined by context.



This element signifies a tip or suggestion.




This element signifies a general note.



This element indicates a warning or caution.

Safari® Books Online

 **Safari**® *Safari Books Online* is an on-demand digital library that delivers expert **content** in both book and video form from the world's leading authors in technology and business.

Technology professionals, software developers, web designers, and business and creative professionals use Safari Books Online as their primary resource for research, problem solving, learning, and certification training.

Safari Books Online offers a range of **plans and pricing** for **enterprise, government, education**, and individuals.

Members have access to thousands of books, training videos, and prepublication manuscripts in one fully searchable database from publishers like O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit

Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology, and hundreds **more**. For more information about Safari Books Online, please visit us **online**.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (in the United States or Canada)
707-829-0515 (international or local)
707-829-0104 (fax)

We have a web page for this book, where we list errata, examples, and any additional information. You can access this page at http://bit.ly/design_develop_glass.

To comment or ask technical questions about this book, send email to bookquestions@oreilly.com.

For more information about our books, courses, conferences, and news, see our website at <http://www.oreilly.com>.

Find us on Facebook: <http://facebook.com/oreilly>

Follow us on Twitter: <http://twitter.com/oreillymedia>

Watch us on YouTube: <http://www.youtube.com/oreillymedia>

Acknowledgments

We'd like to thank O'Reilly Media for picking up the project and believing in us and our concept for a book to help educate, inspire, and motivate people to get involved with this exciting new movement. Mike Hendricksen and Rachel Roumeliotis have been incredibly gracious, informative, patient, and available throughout. The Glass team at Google[x] and several Googler friends of ours have also gone above and beyond with guidance and motivation, so a tip of the cap to Jonathan Beri, Angela Chien, Timothy Jordan, Jenny Tong, Sarah Price, Dori Storbeck, Alain Vongsouvanh, Natalie Villalobos, and Teresa Zazenski for their input and constant nudging-on to make this community contribution a reality.

And even though we mention them and their work more than a few times, we'd like to especially recognize our friends and fellow Glass early adopters—those endlessly talented people that have been part of this unforgettable experience with us from Day 1,

producing Hangout shows and media appearances, coding Glassware, leading discussions in community groups, organizing meetups, writing about Glass, being critical of the platform, and sharing our interest in making it a success. Thanks to Cecilia Abadie, Keith Achorn, Noble Ackerson, Libby Chang, Ethan Bresnick, Mike DiGiovanni, Katy Kasmai, Israel Knight, Dan McLaughlin, Gerwin Sturm, Daniel Ward, Andrew Pritykin, Abraham Williams, the members of UbiTech NYC, and all the Glass Explorers from all over whose diverse contributions to our community are outweighed only by their boundless passion for Glass.

They continually lent their vast knowledge and experiences to this book's production, so we're eternally grateful for having kindred spirits whose insight and enthusiasm matched our own, and who so readily shared it with us.

— Allen and Jason

My special thanks to Jason, my coauthor, collaborator, benevolent taskmaster, and friend, for his work in getting the book off the ground and for serving as the sounding board for all of my design ideas. It has been my honor to work with you, and I look forward to the day when we can meet in the same time zone.

— Allen Firstenberg

I also thank my coauthor and friend for collaborating with me on this book from half a world away and for providing the vision throughout its production. While I came up with the idea on a whim during a Hangout that we write a book about Google Glass, Allen supplied the creative direction. I may have raised the sail, but he steered the ship. It's been an absolute blast working with you, partner!

— Jason Salas

About the Authors



By day, Allen Firstenberg is a Senior Project Engineer at **Objective Consulting** where he has been instrumental in creating websites and mobile apps for companies and organ-

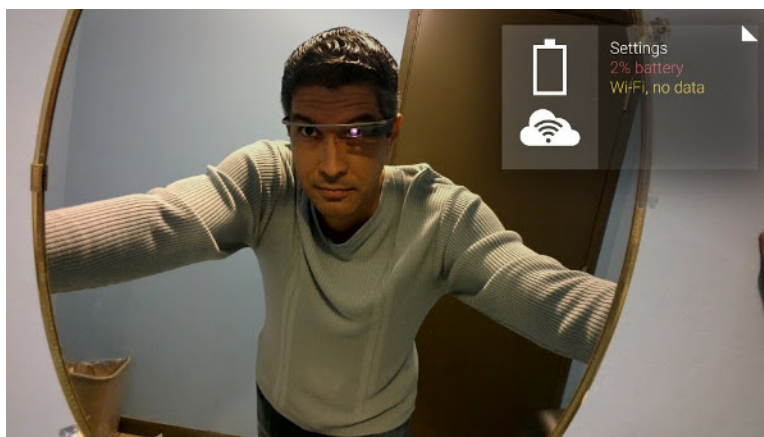
izations from the American Booksellers Association to the National Science Foundation. By night he dons his super-coder cloak and creates tools, software, and tutorials to help people share their stories and improve their digital lives.

Allen is a **Google Developer Expert**, recognized by Google as one of the leaders in the developer community for Google+ and Google Glass, and is a Glass Explorer and Pioneer, having attended the New York Glass Foundry event run by Google in early 2013. Appearing on the YouTube shows **Our Android Week** and **Google Plus Week** (which he cohosts), he is one of the first to heap praise on good ideas, one of the most brutal in criticizing problems, and is relentlessly focused on helping people understand the grand vision. In the early days of the World Wide Web, he created one of the first collaborative websites, Adventure, in an effort to push the boundaries and explore what could be done with the new media that was beginning to evolve.

Allen holds a BS in Computer Science from Rensselaer Polytechnic Institute.

He was inspired to join the Glass Explorers program when he realized how intimate and personal Glass would be, yet at the same time be inherently social. His goal is to help people find the experiences with Glass that can be done by nothing else on the market today, to watch people look at each other as they discover new ways to play games together with Glass, and to watch (and record) his son's face as the two of them roll in laughter during a tickle fight.

Allen is on **Google+** and **LinkedIn**. He blogs about life and other four-letter words at iaflw.com.



A software developer, marketer, broadcaster, sportswriter, and filmmaker, Jason Salas enjoys perpetual summer living on the island of Guam. He fell in love with the Web the moment he first got online in 1994, seeing it as the world's biggest toy, and began helping people embrace the Internet before shifting his focus to content development and ultimately to designing data-driven systems. In the last few years he's been concentrating

on Google Glass, Android Wear, Hangout extensions, HTML5 games, and Chromecast apps. Since 1999 he's worked at [KUAM News](#), where he runs the company's R&D group and also co-anchors the nightly news (*no, really*).

Jason was twice named a Microsoft MVP for his involvement with the ASP.NET community, is a four-time winner of the Edward R. Murrow Award for cutting-edge web development in the news industry, and is constantly interacting in Google+ Communities for Glass users and developers. He's served as technical editor for a book on Microsoft Hailstorm, has written for .NET Magazine, MSDN, and Wrox, and authored an ebook about the trials and tribulations of a semipro football league. He holds a bachelor's degree in marketing from the University of Guam and an MBA with emphasis in technology management from the University of Phoenix. He's also an adjunct professor of MIS and business at the University of Guam and a member of the Football Writers Association of America.

He wants to make good on his public promise to be the first man on the planet to do three distinct things with Glass: host a Hangout On Air from Epcot Center, call play-by-play for a game while broadcasting his POV video to give viewers a behind-the-scenes look at a live sports production, and direct a stage version of *Hamlet* with all actors performing the tragedy of the troubled Prince of Denmark through their own first-person perspectives.

Jason is online at jasonsalas.com.

In this introductory part we give you a primer on the Google Glass ecosystem and all it entails as a wearable computing platform. We also introduce the Think for Glass philosophy, and begin to touch on how you should apply it to how you approach Glass as a user and an architect of Glassware.

- Chapter 1, *The Revolution Will Be Wearable*
- Chapter 2, *The Glass Ecosystem: What It Is and How It Is Different*
- Chapter 3, *Societal Issues with Glass and How to Avoid Them in Your Projects*

The Revolution Will Be Wearable

Forging Glass

Even in a world when smartphones, tablets, and laptops are becoming faster each month and shipping with tons of memory and eye-popping HD displays, the hardware that runs in Glass is impressive, putting the capabilities of Glass on par with that of a Galaxy Nexus. As you'll find, the lightweight model of the ecosystem ensures that the infrastructural demands for Glassware, the applications that run on Glass, don't generally require a supercomputer or eye-popping specs.

First, let's consider the Glass form factor. Gadgetry driving processing capability, hard disk storage, and network connectivity that used to take up entire server rooms is now condensed within the arm of a headset frame less than a half-inch thick and not even weighing a full pound. Put into perspective, this is more raw computing power than was on the first space shuttle *and* which initially ran the New York Stock Exchange. (*OK, we're totally making this part up...but it's probably true. Someone look it up and get back to us.*) At the same time, it is important to understand how the final physical design reflects a set of compromises between what is possible, what is visually appealing, and a wide range of other diverse requirements.

From the software side of things, Google engineered the *Glass sync* component of the application framework that manages cloud-based services to do all the heavy lifting—calculations, string manipulation, data transformation—in the cloud, not on the device itself, reducing the need for expensive on board components to handle such processing locally, meaning all that's transferred over the wire is extremely small payloads of HTML markup. Glass sync manages the complex messaging aspect for you—delivery, queuing, retries for users whose batteries might be dead or without Internet access, etc. Timeline cards, which are the atomic units that encapsulate data on Glass, akin to web pages within a web browser or screens in a mobile app, arrive to the wearer in near real

time with information that's geographically aware, contextually accurate, and user specific.

When you do need to run apps locally, Glass is still capable of executing Android applications written specifically for its user experience. Processing, memory, and graphics cater to *microinteractions*—a streamlined, perfectly suited form of application interactivity that doesn't require lengthy usage sessions or attention. Aggressive competition by hardware manufacturers has driven the price for consumer technology down considerably, and innovations in materials engineering have allowed the physical properties of the Glass head-mounted display (HMD) to be light and flexible, yet durable and sturdy. The display prism reflects bright colors and rich video. The embedded camera captures remarkably clear images and video.

It's truly an elegant system.

The one thing to keep in mind as you approach your own adventure with Glass as an application architect is that at its core, this technology isn't a new idea. Glass is the result of a concept that's been floating around advanced computer science circles and egghead thinktanks for the past two decades: *wearable computing*. This is some pretty hardcore stuff, so buckle up.

Wearable Computing

Straight from the pages of science fiction and echoing episodes of *The Jetsons*, wearable computing has been envisioned for years—a utopian era when devices would be small and lightweight enough to literally be part of your person and actually mesh with your outfits (there's a reason Glass ships in five distinctly attractive colors—Charcoal, Cotton, Shale, Sky, and Tangerine). This is the next progression in how we use computers, going from desktop towers tethering us to fixed locations, to portability with laptops and netbooks with wireless connectivity, and then the mobility advantage smaller devices like smartphones and tablets achieved, to finally wearables with intelligent software.

Relatives of Glass in this emerging space are products from Telepathy One, Microsoft, Recon Jet, the IndieGogo-backed GlassUp, Chinese search provider Baidu, and even automaker Nissan; Vuzix's M100 Smart Glasses; smartwatches like the Samsung Galaxy Gear, Sony SmartWatch, Qualcomm Toq, and the Kickstarter-born Pebble, along with offerings from Apple, LG, and Google; and even do-it-yourself electronic components like sensors that you can sew into your clothing with special conductive thread and program to do all sorts of neat things.

Being designed from the ground up as a platform that frees you from having to constantly fuss with controls or type input or force refreshes, keeping content and the delivery of it out of your way, *Glass exists to sit on your head in order to put your life back in your hands. Don't you just love the irony?*

What Does It Mean to Think for Glass?

When you begin to Think for Glass, you begin to think in this new way that Google is beginning to forge. It means thinking about social interactions, personalized delivery, and streamlining interactions—all at the same time. It means thinking about more than just one application, it means thinking about how everything works together to create something greater than the sum of their parts. It means thinking about new ways for how your program should behave, how information flows between services and devices, and how the interaction experience is optimized.

Being able to Think for Glass keeps technology omnipresent yet nonintrusive—a lofty goal achieved only by delivering high-impact usability with minimal cruft. This is a dramatic reinvention of how we've come to perceive mobile communications.

Here's the basic idea: *Glass is designed to live in your world, not for you to live in the Glass world. It is meant to adapt to your life, not for you to adapt to how it does things. Your apps are expected to behave the same way.* Everything else stems from this basic idea. To be sure, it is an ideal—but that doesn't mean it isn't the ideal to strive for. It also suggests that just because something can be done with Glass doesn't mean that it is a good idea to do so.

It means that anything that works through Glass should be secondary to the world around the person wearing Glass, and that an app should never expect otherwise. With few exceptions, this is what the user expects, and violating those expectations will only cause problems and a poor user experience.

Software that works best with Glass are those programs that require no more than about five seconds at a time. *This is the essence of microinteractions.* If users are taking more than that time, they will be making a conscious decision to break out of the world they're in and enter the Glass universe. Sometimes users will choose to do this—to review photos and share them, for example. But even these tasks are oriented toward what users want to do (find a picture and share it) rather than how an app wants them to do it (open their app and process a photo). Apps should accept a wearer's commands—not narrow a person's choice into what the app wants you to do.

Of course there are menus and voice commands that limit what a person can do, but app developers should be as aware and open as possible to the full set of commands that are available by default and how people may wish to use the data they provide. It takes a lot of extra thinking, and we'll be discussing the designs required to do this successfully.

Consider, for example, the desire that everyone has to get on the Glass home screen, or to give their cards persistent presence on a user's timeline. This is an app-centric focus—we want our applications prominent for the user. But it forces users into our world instead of trying to understand the world that they want to be in. They don't want to sort through dozens of “pinned cards,” as the Glass terminology refers to it, to find the

one they want. They want us to deliver information to them when, and only when, they want it. They want to be able to easily ask for new information. They want to quickly find the data they know is there...somewhere. They don't want clutter. (And if you're confused about what this all means, you'll understand by the end of [Chapter 5](#).)

Sometimes, the boundary between “the Glass world” and “the human world” isn't clear. Consider an app for running enthusiasts, for example. The runner wants to be able to quickly, easily, and probably frequently see his progress, heart rate, course, and other important information. But, all too easily, this information can become distracting if constantly presented...and the runner could stop paying attention to the physical world he is are trying to traverse, risking physical injury. Perhaps this is a case where a runner can set predetermined alarms (heart rate too high, another mile traversed in the course, etc.) to notify him for important events, but also allow for an on-demand summary if the runner wishes.

This event-driven angle is at the heart of Glass. Glass needs to be a mind reader—delivering the information about events exactly when you want, before you ask for it. It needs to respond, immediately, to the events you wish to capture in the world around you and turn them into online representations of those events.

We have to remember, also, that the world around us isn't the same as that of Glass, or even the wearer's world, and Glass tries to address that. As we write Glassware, we need to remember that the public perception of our world isn't going to adapt to Glass just because we may want it to. This is why Glass requires specific actions to take a picture and why it lights up when recording video—the world around us has certain expectations that Glass tries to meet. Our software needs to respect the social norms, while at the same time nudging it forward. Software that violates these norms (for example, software that does facial recognition or that uses material Google deems objectionable) is discouraged, even if the features it provides could be tremendously beneficial.

Glass is different. Glass is more personal than any computer we've had before. Glass is technology that tries to get us away from technology and into our world. Our applications can't try to drag a user away from that.

By the end of this book, we'll empower you with the knowledge to Think for Glass so that you respect the following virtues about optimal Glassware construction from three perspectives:

1. Usage

- Understand personal technology
- Appreciate the value of personal area networks
- Know the ecosystem
 - Hardware, firmware, cloud infrastructure, Glassware, administrative tools

- Realize Glass doesn't replace, but complements, your other gear
 - Smart devices and mobile platforms still have their place alongside wearables
- Get involved
 - Ask questions, share knowledge, submit feedback, participate in the community
 - Demand Glassware from your favorite services
- Live in the physical world
 - Respect the privacy of those around us
 - Use Glass to be part of the moment, not divorced from the moment

2. Design

- Aim for simplicity and brevity
 - Target supporting microinteractions
- Respect the Five Noble Truths
 - Design for Glass
 - Don't get in the way
 - Keep it relevant
 - Avoid the unexpected
 - Build for people

3. Development

- Leverage the software-as-contact model
 - The evolution of sharing
- Work judiciously with frameworks
 - Mirror API versus GDK
- Build distinctly for Glass
 - Don't "lazy port" existing Android or web apps
 - Leverage the Glass UI and control system
 - Exploit event-driven existences
 - Context powers content
 - Let users live in the moment

The Glass Ecosystem: What It Is and How It Is Different

When Glass was first unveiled in 2012, the developer community was both put on notice and challenged, with Google producing a new application client that's radically different. But Glass is more than just a set of hardware specifications or the apps that are available by default, it is a new way of interacting with your computer and with the real world.

Our main goal isn't to give you the information just to produce Glassware—*we want you to produce GREAT Glassware*. Your success is our success. So let's hit the ground running and set you off on your journey to thoroughly understanding the Glass ecosystem and becoming a Glassware-producing superstar.

What You See and What You Get

As you can tell from [Figure 2-1](#), the maturity of the Glass design scheme has come a long way in about two years. The headset appears as a minimalist pair of glasses that have been bulked out with some additional hardware. On the one hand, the additions are sleek and almost futuristically styled. On the other hand, they are clearly noticeable and perhaps a bit bulky. Some are made even more visible because of the color choice.



Figure 2-1. The evolution of Google Glass (image courtesy of Google)

But contained in this device is a small modular bundle of technology:

- A battery module
- A micro-USB port that serves as power source, headphone jack, and data port
- A bone conduction transducer speaker
- A trackpad that detects forward, backward, and downward swipes, as well as one-, two-, and three-finger taps and long presses
- Assorted sensors that can detect head tilts, head turns, and eyelid movement
- WiFi radio and Bluetooth module to communicate with either a phone or directly to the Internet
- A fixed-focus camera that has roughly the same field of vision as your eyes
- A microphone tuned to pick up the wearer's voice
- And, of course, the characteristic display that gives Glass its name

We fully detail the full technical specs for Glass later in this chapter.

Glass Is a Platform, Not a Product

It's important in learning how to Think for Glass to realize that the product known to the world as "Google Glass" isn't just a device you wear on your head. It's a full ecosystem—a synergy of hardware, software, applications, APIs, and a backend environment. It's

also an opportunity to create new peripherals and accessories for things like custom frames, sticker designs, display sleeves and covers, and other modular add-ons.

Like most of Google’s product line, Glass is a platform, not a product (Figure 2-2). The hardware that makes up Google Glass itself is an achievement of industrial design. The product has conquered environmental challenges of size, weight, and sturdiness, impressively addressing obstacles like power consumption, networking requirements, and heat dissipation. On a functional level, it seeks to accomplish a single type of experience—to cater to *microinteractions*, allowing the wearer to utilize technology while not being taken out of the moment.

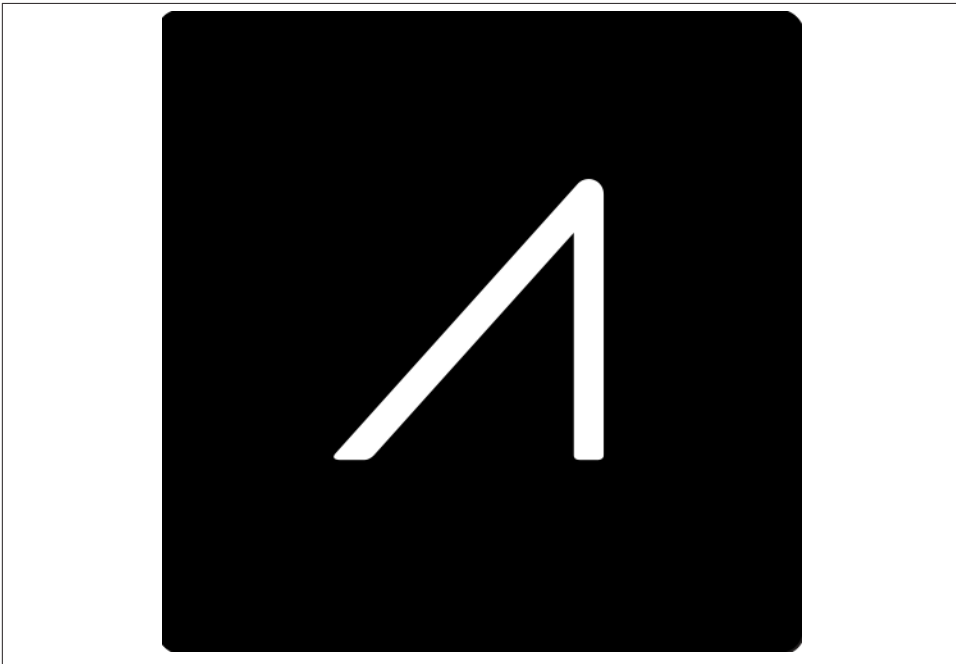


Figure 2-2. The Google Glass logo (image courtesy of Google)

The hardware, the system software, and the application environments represent a balance between user needs, aesthetics, and design constraints and are extremely elegant in their approach. Glass embodies a huge shift for Google from a design standpoint. The company that gained global notoriety for its intentionally sparse and minimal user interface elements with its web systems to capitalize on speed has invested heavily into assembling a team of designers, materials experts, human–computer interaction savants, and scholars of personal computing to come up with a design concept that’s pleasing to look at, comfortable to wear, inherently self-promotional (“Hey, check it out! That

guy's got Google Glass!”), and functional. The slender shape of Glass, functionally wrapped around your head, is sleek and modern, not clunky and cliché.

That’s a fairly tall order for any device, much less one that’s worn on your face, and weighing less than a pound.

The components are packaged into a thin form factor that sports cosmetic appeal and handles the connectivity for Glass and communications demands of an Internet-aware device. Glass was built with modularity in mind, and the arm containing the electrical components can be removed from the frame for aftermarket modifications like fitting them to custom frames.

The Glass team was beyond obsessed with getting the form factor right. Isabelle Olsson, lead industrial designer for Google Glass said at Google I/O 2013, “If it is not light, you’re not going to want to wear it for more than ten minutes,” adding, “We care about *every gram*.”

The space-aged aesthetics of the headset with its extremely pliable titanium-trimmed frame, matte finish, and prism display make Glass comfortably symmetric on either axis—the headset doesn’t slide too far forward on your face, being counterbalanced by the rear-seated battery sitting behind your ear; and it won’t awkwardly sag sideways, despite the impression that all the gadgetry is lopsided to one of the arms.

On the software side of things, the Android-based firmware that runs Glass features an extremely responsive UI sitting on top of the timeline concept that’s easy to master with a slick multi-input control system, is highly performant, and properly handles multimedia like other mobile platforms. The UI uses a simple card-based metaphor (increasingly used across many of Google’s products, and one that we’ll go into great detail about later) with simple head gestures like nods and voice commands to control it. This lets most of the common tasks be run with minimal CPU usage, and thus minimal drain on the battery.

The Glass Application Model

Glassware, those programs running on Glass, is available in two distinct flavors: those built using the Google Mirror API and those written with the Glass Development Kit. Each has its own approach to deliver a consistent experience. Both of these frameworks are thoroughly examined in third part of the book, *Develop*, but let’s take a look at them briefly here.

Mirror API

The Mirror API is a RESTful interface, handling server-side programs with all computation done in the cloud before they get sent to your device, inserted into your timeline,

and rendered as cards. All you get is the finished product, thoroughly cooked and ready to eat. These types of services resemble traditional client/server web apps.

Because Mirror API programs run purely in the cloud, the processor on Glass is left free to work on other things and not worry about calculations, string manipulation, on-the-fly interpolation, working with binary data locally, or client-side presentation mechanics. The payload is pure HTML and CSS style rules (but no JavaScript, at least not at the moment), both lightweight and simple to handle. Multimedia can also be included with minimal additional overhead, and video is streamed on-demand, not downloaded.

The Mirror API framework is based on the publish/subscribe model that enables push notifications, and avoids repetitive operations between clients and servers. As opposed to the traditional method of installed programs, this keeps downstream payloads to a minimum. To receive updates and interact with the Glassware, users authorize their Google accounts to register with the Glassware without needing to install executables on the device. This eliminates a lot of unnecessary network roundtrips, which again, reduces radio activity (via WiFi or Bluetooth tethering to a mobile device), bandwidth, and battery use.

This is significant to the optimal performance of the system. There's no requirement for managing a massive internal storage hard disk to store application components like embedded databases, configuration settings, and data caches. This also results in the development time for Mirror API projects being incredibly rapid—you can get a complex project up and running in a couple of hours. This speaks directly to the flexibility of the Glass ecosystem.

Mirror services do have two big requirements: being cloud-aware, they obviously need connectivity. Also, services need to be registered by using an OAuth provider, which typically means having to do so in a browser. There's no executable file you can just send to someone.

For services where network access is optional, you need access to the Glass sensors, the default UI of Glass needs to be extended, or you'd like a more flexible means of distributing your programs, Glassware can be written using the GDK.

Glass Development Kit (GDK)

For those who want more granular control of their application and the environment in which it runs, Glassware built using the Glass Development Kit, an extension to the Android SDK for Glass, are programs that are written in Java and installed on your device. The GDK extends the standard libraries used for Android programming with Glass-specific features, resulting in an app that can be distributed easily and installed directly on the device.

GDK Glassware goes functionally further than the Mirror API, in directly accessing the hardware and being able to run offline. In applications programmed with the GDK, the

same lightweight, microinteraction model is enforced via the timeline and cards UX, but this can be expanded upon, or even diverted from completely. It's possible to program immersive native experiences on Glass, letting you create UIs that stay resident in the prism and are meant to be used over longer sessions.

Examples of immersions are games and applications using the camera.

Actions, Not Apps

This approach to the application model leads to a different way of thinking about how to treat apps with Glass. As we will see, the most natural way to use Glass is to think about what you want to do, not what app you want to run to do it. In fact, we would be so bold to say that if your mindset is about apps and not what you want to do with Glass, *YOU'RE THINKING ABOUT IT WRONG*. Glass doesn't have a launcher in the same way a phone or tablet does, and the home screen invites you to issue instructions to it. The results from Glass commands are displayed as part of the timeline, mixed together with the results of other notifications, updates, and commands. Although each of these cards is generated by a program, we don't normally need to think about which program generated them.

Similarly, Glass changes the way that we as users interact with wearable computing devices through the various input mechanisms it supports—swiping/tapping on the trackpad, using voice commands, using head movement gestures, and even winking to instruct Glass to take a picture. All of these manipulate what is shown to us, but none of them appear to “launch” an application directly. Instead, Glass represents them as verbs—actions that are doing something, rather than objects that do it.

In the future, it may even be possible to see other physical actions associated with activities on Glass. There has been a lot of work in extending the ways to trigger actions on the system—Google was granted patents to allow a user to make hand motions out in space, which the company hinted might ultimately be used to endorse a post on social networks (like, +1, heart, star, thumbs-up, mood, etc.), by making the popular heart hand gesture. And the company **Remotte** aimed to further make interacting with Glass seamless, launching a Kickstarter campaign to sell its remote control that communicates with Glass over Bluetooth. Users can keep their hands in your pockets and just control the HMD with simple button clicks while out for a walk, never needing to reach up and fiddle with the trackpad.

A point of reference about the “actions, not apps” idea is the voice commands available to launch Glassware. These aren't just arbitrary trigger phrases picked because they sound neat or roll off the tongue nicely or are terse enough to work in a crowded room—they use active voicing with strong verbs and truly capture the essence of interacting with the wearable software. They lay out very simply and without ambiguity what single

action you need to take to get an app to conduct its main purpose. And that's what you need to target.

The projection unit actually shares space with other key hardware in the area of the device officially known as the *optics pod*, the module that houses three very powerful sensors on Glass—the accelerometer, the gyroscope, and the magnetometer.

The outside of the brick (the side perpendicular to your face) is covered with two-way reflective material so as not to let ambient light in or out. Projected content bounces off the angled piece, refracting the light down through the flat side of the prism facing the user's eye, and projecting it onto the wearer's retina, but giving the impression of information being projected out in space in front of them. The end result isn't unlike most commercial HUDs commonly used in some automobiles.

Because the HMD is strapped to your head, the display perfectly follows whatever direction you're facing, consistently and without lag. It's always there.

The timeline UI and the cards within it appear as semitransparent images, allowing the user to look through the content set against the real world as a background stage. The projector can also be adjusted manually, as the section containing the unit is on a hinge, and can be manipulated for better visibility. Glass sends the content directly at you at a perceived scale that's neither too big nor too small. To achieve a natural and comfortable depth perspective, Glass tweaks the appearance of projected material out in space by way of a clever optical illusion (see the sidebar [“Calling upon an Old Hollywood Trick” on page 19](#)), all while being mere centimeters from your periphery. Many first-generation Glass users have said looking at content in Glass is akin to holding your smartphone screen at arm's length.

As a pro tip to see the actual difference between the projected display to really appreciate the display while wearing Glass, try this: wake up Glass so that some sort of content is showing, and then look at yourself in a mirror. While the display as it appears to you is relatively large and legible, the actual dimensions of the display are actually miniscule.

That's the effect it achieves and the magic behind how it gets it done.

A couple more helpful things that the prism does for its wearer automatically are in terms of the amount of available rays. The prism is made of a photochromatic lens that reacts to an increase in ultraviolet light, applying tinting so that direct sunlight doesn't blur information or become magnified and blind you while you wear it. On the opposite end of the spectrum (pun certainly intended), if you have trouble reading the content being projected indoors in artificially lit rooms, try pointing Glass for a moment toward a lamp, a TV, or some other highly illuminated object. One of the sensors that the device has on board (which we cover in [Chapter 13](#)) measures light and self-brightens the display if the environment is too dim.

This Feels Vaguely Familiar

One of Allen's most poignant observations about the product's design is something he's told many people since using Glass, saying, "If I didn't tell you it was made by Google, you'd swear it was an Apple product."

And he's not alone.

Scores of people have given the design of Glass high praise by saying its feel rivals the legendary product designs a certain tech company in Cupertino, California is known for. Cecilia Abadie, a Southern California developer who created **Genie** ("The Swiss Army Knife of Glassware") and one of the original Explorers, says Glass "is one of the most carefully designed environments we've seen in a long time." We won't challenge that conclusion.

When both of us first got Glass, we agreed we haven't been this excited to build cool things for a platform in quite some time.

Reinventing Human–Computer Interaction

Another subject that got a lot of attention before Glass rolled out was exactly how much independence it would have as a communications device. People constantly debated the autonomy of the wearable computing apparatus—one camp assumed that it would merely exist as a peripheral or accessory, linked to a phone in order to communicate, and ostensibly dead in the water without it; others surmised it would be a first-class computer, complete with its own cellular data connection.

Glass thankfully is a hybrid of sorts, a self-contained device with its own WiFi radio and not explicitly requiring an accompanying smartphone, but enhanced by tethering to another smart device for network connectivity and for telecommunications services like text messaging and voice calls. This mercifully relieves you as a user of the need to have multiple data plans with a carrier, reducing your total cost of ownership. Administering your Glass profile requires only the occasional peek at the helpful MyGlass mobile/web app to manage Glassware subscriptions and manage contacts.

The main theme, and a prime objective of how the system was designed, is to require as little user input as possible to negotiate the system. It's a hands-free, ears-free, and wires-free means of staying connected and interacting with others online.

Glass is also able to audibly recite text content to you. Glass additionally becomes a telephony device when tethered to a smartphone, facilitating text messaging and voice calls, and chat sessions and multiuser calls through Hangouts.

As long as Glass has a network connection, you're plugged in (metaphorically) and ready to run with core services like search, participating in Hangouts, obtaining directions,

and getting real-time directions, as well as using the growing number of third-party Glassware applications the community is building. You could get by with just WiFi and the occasional visit to a desktop web browser to manage your profile.

And even if you go offline because you're out of cellular range, your phone's battery dies, your WiFi time at the library expires, whatever the reason—Glass is still perfectly capable of taking pictures and shooting video and storing files locally, which can be synced to the cloud when you regain connectivity.

And in case you were wondering about any dangers about waves being emitted from a computer that sits resident right next to your brain for potential extended periods of time, because the radio communication is short range, being Bluetooth tethering or WiFi, the radiation that Glass gives off, [according to Google](#), is “significantly less than a cell phone.” Additionally, Glass is designed to force heat generated by the processor to flow away from the user, so even in times when Glass may be heating up due to extended use or with applications that require a significant amount of processing, the heat can be felt on the outside of the touchpad opposite your head, but you won't feel it against your skull.

What Glass brings to the table is a rapid-response mechanism for you to stay in the moment. And this means not requiring you to look down, negotiate input controls, type frantically, and navigate through menus and complex user interfaces to do what you want. Its hands-free, ears-free, wires-free design liberates you in being able to interact with objects, places, and people as you normally would without fidgeting with a device and manipulating a screen to perform operations like editing photos, seeing where people are in relation to you, joining a Hangout, or sending a message.

Glass doesn't accomplish anything you can't already do with existing technology, *it just does it without so much effort*. What's critically important to realize is that Glass was designed so people would interact with it very differently than other types of computing devices. “But,” you're probably saying, “isn't this what I'm already doing with my smartphone?” Absolutely. The product doesn't intend to take any of the shine away from its more tenured cousins—the smartphones, laptops, and tablets you've been using for years. Rather, it wants to be a valuable, contributing member of the family and take that interaction to the next level.

This isn't just wearable technology, this is personal technology. Really let that statement sink in for a few seconds, because it's key. Understand that this isn't personal like “personal computer”—in some ways, it is, but it's also so much more. This is a new dimension of connectedness with data, and a new form of intimacy—both for you with your social connections, and for you with your device.

Technical Specifications (as of the Explorer Edition)

- Weight: 1.5 ounces (.9 lbs)
- Operating system: Android 4.4 (KitKat)
- Memory: 682 MB of free memory, possibly 2 GB total, on Elpida mobile DRAM
- Chipset: Texas Instruments OMAP 4430 SoC 1.2Ghz dual-core (ARMv7) CPU
- Display resolution: 640 x 360
 - Slightly less than VGA
 - HTML-based timeline cards have a recommended CSS gutter to limit text from bleeding off the visible area, but images and video are displayed at full resolution
- Display focal depth: similar to viewing a 25-inch high-definition display from 8 feet away
- Embedded digital camera: 5 MP images, 720p video
- Audio: Bone conduction transducer
- WiFi: 802.11b/g
- Bluetooth: Bluetooth Low-Energy 4.0 (BLE) module
- Touchpad: Synaptics touch module driven by T1320A touchpad controller
- Disk space: 16 GB flash memory via Sandisk
 - 4 GB for system software and drivers, 12 GB usable via cloud-synced storage
- Battery: Nonremovable 570mAh cell providing several hours' charge with normal use

The Science Behind the Projection

One of the biggest curiosities pundits sought resolution on as the initial *Project Glass: One day....* concept video and the follow-up video, *How it Feels (through Google Glass)* made their way around the Internet and into societal consciousness was the device's display. This fervor ramped up in intensity as shots from various angles of the early prototypes immediately went viral. Great interest ensued for insight about the little sliver of clear glass's resolution, focal length, clarity, HD capacity, and ability to share what the wearer was seeing on other displays via mirroring. It wasn't too long until the projection system was detailed, revealing some very impressive engineering that delivers remarkable quality in an incredibly small space.

People pondered the possibilities of the system and what radical new technology might be at play, and whether text, images, graphics, and video in Glass would be floating as an overlay in some sort of flat monitor with content sitting out in space for a user to

gaze at, or whether data would be projected directly onto an owner's eyeball. A **technical teardown** of the hardware by Catwig.com published in June 2013 notes the “the pixels are one-eighth the physical width of those on the iPhone 5's retinal display.” This for a while was one of the most hotly contested topics in Glass forums, in media reports, in social posts, and around water coolers.

The answer is—it's both!

Glass displays information by using its signature prism display in concert with a sophisticated projector system. The projection unit is mounted at the front of the arm just behind the embedded camera, casting its image within the display brick, which houses a second piece of glass at a 45-degree angle, creating a prism. What you may not have immediately noticed is that the Glass logo, which is also its social avatar, features a slanted “A” in the product name, which is a clever mnemonic device reminding the wearer about how Glass achieves its overall experience (it admittedly took the both of us months to realize this). Pretty sneaky, huh?

Calling upon an Old Hollywood Trick

The impression that the content you're seeing is further away than its actual distance is managed by a variation of an optical illusion technique used in movie making for decades. To achieve a spatial feel much larger than the actual dimensions, set builders in the film and theatre industries often employ forced perspective, which tricks observers into thinking an object is of much different proportions than it really is by scaling or skewing it in various ways.

This technique is used extensively at Disneyland. For instance, Sleeping Beauty Castle, the centerpiece of the theme park, was constructed so that its unmistakable tower gets narrower as it gets taller. Observed from ground level, this creates the effect that the building appears to be a skyscraper, when in reality it's just 77-feet tall. A similar technique is applied by optometry equipment for eye tests.

The Glass team employed this same principle in creating depth of field for tweaking the prism's focal length in relation to your eye and your perspective, so that images projected to you would seem to be a comfortable distance away in the prism, sitting out in space rather than slammed up against your face. The projected image also appears much larger than its actual dimensions—the effect of viewing a 25-inch high-definition display from 8 feet away. In reality, the images are being projected onto your eyeball. This preserves clarity, makes graphics stand out, and ensures text is legible.

Show business, baby!

How Glass Gets Audio into Your Ear

Bone conduction transducers aren't exactly a new concept, but their use in Glass is going to be most people's maiden voyage with the technology at the consumer level. Devices built on the technology have been available as hearing aids for the hearing impaired or the elderly.

Ears-free audio in Glass is achieved by converting audio signals to vibrations, which are then sent through the speaker inside the arm of the frame on Glass and vibrate against the user's skull behind the earlobe and into the inner ear, rather than by broadcasting soundwaves directly into the user's eardrum. The audio quality and clarity is remarkably good, comparable to a good pair of headphones.

But for the more traditional user, Google makes a set of micro-USB earbuds designed specially for Glass. You get the best of both worlds!



Battery Life

Glass isn't meant to be used as a perpetually-on device. The use of camera-centric services such as recording video for long periods of time, in addition to applications that cause the Glass projector to stay on and tax the processor, like games and turn-by-turn navigation, will more dramatically drain a battery's charge than allowing the device to go to sleep after a few seconds of nonuse. This shouldn't come as a surprise—most mobile devices that excessively use a camera, display, or data communication won't have tremendous usage time. (The Amazon Kindle is one of the few exceptions.) Glass isn't any different in that regard.

Building apps in such a way as to not kill the user's battery is a major principle of program design for any mobile developer, but it's especially true for Glass when considering the low-intrusion goal that we'll cover thoroughly in [Chapter 13](#) when we talk about the GDK. But just know that it is possible—and highly encouraged—to design your native apps in such a way to leverage Glass gracefully going to sleep on its own while having your program continue to run in the background.

Using the System

Information in Glass is presented through a simple concept but controlled in a variety of ways. Everything in Glass exists around the *timeline*. Your home screen—the simple UI element with the current time and “OK Glass” underneath it—serves as your timeline's anchor as its center point, as indicated in [Figure 2-3](#). Much like windows are the main visual elements that let you control a graphical operating system, a user's timeline is the interface through which she receives information, gets notified of new content,

interacts with subscribed services, sends the system user input, and makes changes to her customization settings. It's a snapshot of your activity and notifications.

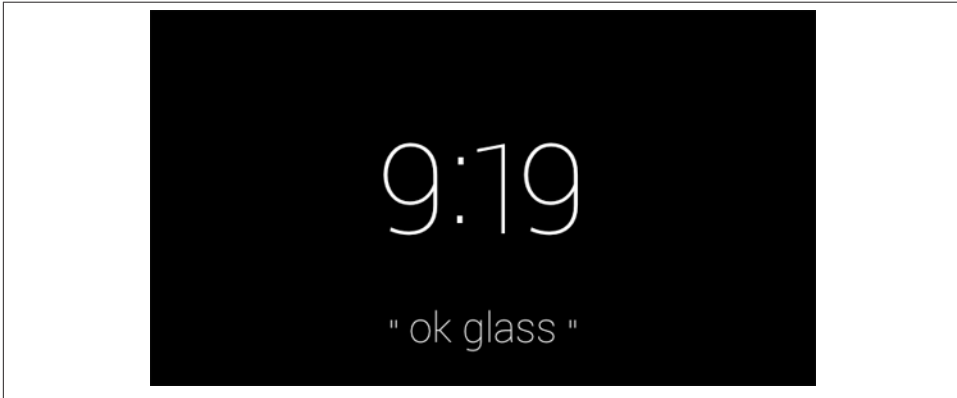


Figure 2-3. The Glass home screen

A timeline consists of *cards*—units of information that support text and multimedia, spanning everything from system settings and status messages, to games, tweets, chat messages, and email messages, optionally organized into groups known as *bundles*. Additionally, cards may be *pinned* so that they sit close to the home screen and are available for quick reference, like bookmarking.

Difference Between Glass and Android Wear

While Glass and its cousin within the scope of wearable computing, Android Wear, share the timeline card as their UI container, the element that contains cards is slightly different. On Glass we've got the timeline that presents information chunks about those events that will happen in the future or are happening now, and those that have already taken place—separated by the home screen. Android Wear-driven applications are organized within the context stream, a similar chronology but all within one continuous feed based more on each item's relevance to the user at that moment, not merely on when it was posted.

It's all about the details.

Each user's timeline has two parts, each consisting of information and events—what's coming up and what's already happened, navigable by single-finger swiping back or forward on the touchpad. System settings, pinned cards, and Google Now cards for upcoming events like calendar entries, to-do items, sports scores, weather forecasts, and stock prices are accessible by swiping back on the touchpad toward your ear (to the left

of the home screen). Also found left of the home screen are any pinned cards, as well as running programs for native apps—both types of cards give the feeling of more dynamic content, and play into the idea of things that are happening at the current moment.

Swiping forward toward your eye (to the right of the home screen) moves over the collection of older items for a running log of your activities. If cards have additional functionality like a series of menu items for a picture, tapping the trackpad to iterate through the available commands for selection is analogous to a mouse click on a desktop computer. Swiping downward goes up a level, so that if you're in a menu item, it goes back to the card with that menu, for example. If you're at the top level, it turns the display off.

Glass turns itself off after a few seconds of inactivity. To turn it back on, you can either tap the trackpad to wake it up, or enable a setting that lets you tilt your head upward and wake Glass up. Glass additionally supports optional features, such as automatically waking from sleep when it's being worn, or enforcing a screen lock consisting of user-defined gestures. Powerful voice controls as we're accustomed to in recent versions of Android are available, both in interpreting specific commands from the main screen for things like messaging—the canonical “OK Glass” menu—as well as for spoken input when replying to a message.

The Camera: Photos, Videos, and More!

Glass is pretty much a point-and-shoot device. A shutter button on top of the camera manually initiates photo taking or video recording. Framing can be a bit awkward because the lens is shifted to the right, but the widescreen nature of the pictures generally captures everything that is in your field of vision. The Viewfinder application acts as a preview monitor, and some Glassware projects are looking to expand the camera's capabilities to be on par with many commercial DSLRs so you can have a better handle on getting the shot you want.

If you've enabled Google+, media files are uploaded to a private album via Auto Backup. This also means that pictures shot in rapid succession along with videos can be assembled by Google+ as Auto Awesome movies. (The ability for an app to compile and edit video footage into a prepared presentation is a concept that Glassware like **Perfect** provide, taking your raw footage and assembling time-lapsed mini-movies.)

Once you've taken a picture, you can immediately share it by saying, “OK Glass, share with...” to share the image with a contact or a piece of Glassware (this concept is big and we'll get to in **Chapter 11**). You can also choose the Send menu item to upload the image to a contact to start a Hangout chat, or to an ongoing conversation with a group.

Video recording by default is for 10 seconds. To record longer clips, tap the trackpad and choose Extend video or just press the top button again. To stop, tap and select Stop

recording or press the top button once more. You can also upload recorded video to the YouTube channel associated with your Google account. Just enable the YouTube Glassware and you'll see the options to share the video as Public, Private, or Unlisted.

However, while Glass can take impressive five-megapixel pictures that look especially good in natural light, don't expect to throw out your existing gear or anticipate never buying another camera again. Because of the intent for the embedded camera to work quickly and capture things in the moment, it doesn't have auto-focus or zoom. It also lacks the types of controls that cameras on other Android devices have, with the focal perspective being fixed. In this regard, the camera is a compromise between several different needs in order to handle the types of situations in which you'll want to quickly capture life happening, at the expense of having a full-feature photography rig.

This shouldn't be seen as a handicap of the system, but the key driver of it. This is what it means to Think for Glass.

Some of the initial natural limitations with the device are evident in the framing of a shot (with the lens right-of-center on your head), having to holding the camera still (meaning not moving your head), and shooting subjects from an appropriate distance to capture them legibly (which varies depending on the situation). So, some early criticism was that the Glass camera generally isn't suitable for doing QR code processing, but some app developers have found clever ways around this.

Limitations Lead to Opportunity

Glass in a few instances may be limited, but it's also extensible. While reading the previous section you might have been inspired to improve upon the camera's capabilities—specifically the lack of controls the camera has out of the box.

You might be able to use the GDK to write native Glassware that takes control of the camera feed and hooks into trackpad gestures to feign features that don't ship with the system. An example of this could be applying Instagram-like filters to photos.

Just some food for thought to get you started. Refuse to accept the status quo! Get creative!

Glass Is a Great Listener

You'll probably be using voice actions a ton to interact with the system. Glass wakes up either through you tapping the touchpad or using a head gesture (if enabled in the Settings bundle). Once active, announcing the hotwords "OK Glass..." triggers the full list of voice actions available to you to be displayed, which is made up of the default system commands for Glass plus any trigger phrases for any apps you may have installed or services to which you've subscribed. This is essentially the same voice-driven tech-

nology used in Android search and in Microsoft's Xbox One. You can speak the command or tilt your head vertically to scroll through the choices and say the command to select.

Voice actions (as of the Explorer Edition):

- “OK Glass, Google + <QUERY>”
 - Open-ended searches return a list of cards as results, which may be matching images and/or web pages. You can also force a filtered search of Google's index for images by saying “*images of...*” and for video clips by saying “*videos of...*”.
 - As far as browsing the sources for search results, Glass provides you with an embedded instance of a browser so you can view pages on the Web. A set of slick tools are available to let you navigate pages.
 - Search on Glass taps Google's Knowledge Graph and uses **the same natural language syntax as Google Now**, and works best with simple informational queries based on factual information like the following:
 - “*Who wrote The Canterbury Tales?*”
 - “*How tall was Wilt Chamberlain?*”
 - “*Who were Mel Brooks' wives?*”
 - “*Stock price of General Electric*”
 - “*When was Bon Jovi's Slippery When Wet released?*”
 - “*Convert 3 US Dollars to Yen*”
 - “*What will the weather be like in Columbus, Ohio on Thursday?*”
 - “*How old was Bruce Lee when he died?*”
 - “*What states make up The Four Corners?*”
 - “*What's the population of Winter Garden, Florida?*”
 - “*How long does it take to get a PhD?*”
 - “*How far to Hershey, Pennsylvania?*”
 - “*How many homeruns did Roger Maris hit in 1961?*”
 - “*What is breaking the fourth wall?*”
 - “*Cast of Dude, Where's My Car*”
 - “*What is the seating capacity of Michigan Stadium?*”
 - “*Definition of subcutaneous*”
 - “*What is Gene Simmons' real name?*”

- Asking something more vague like searching for something abstract such as “*Why did Francis Ford Coppola feel the need to make The Godfather: Part III?*” or “*Will I ever find love?*” won’t be as accurate. Glass isn’t that smart...yet.
- “OK Glass, take a picture” / “OK Glass, record a video”
 - You can also press and release the shutter button to take a picture, or hold the shutter button down to initiate recording.
 - If you’ve optionally enabled and set up the Wink feature in the Settings bundle, you can take pictures with Glass by winking, even if the device is idle and the screen is locked.
- “OK Glass, get directions to + <LOCATION or ADDRESS or GENERIC PLACE>”
 - Returns a list of known places. Selecting a place launches turn-by-turn-navigation.
 - Tapping on a directions card also lets you swipe through a series of travel choices, giving you directions for driving, walking, biking, and transit.
 - Location searches are best when using a formal name of a place like “McDonald’s” or a generic type of location like “gas stations” (which give you a list of matches), the full address of a location such as “1600 Pennsylvania Avenue,” or user-defined names like “School,” “Work,” and “Party Spot.”
- “OK Glass, send a message to + <CONTACT’S NAME or CIRCLE or HANGOUT GROUP>”
 - Create a message for the specified recipient via voice dictation. Once you stop speaking, Glass waits for one second before delivering the message and during that time gives you the option to swipe down to erase it and start again.
 - Glass uses a **specific hierarchy** of services for Messaging, with Hangouts sitting atop the food chain. If you have the Hangouts Glassware enabled, that service will be the default delivery mechanism for all contacts. If you’ve set up Google Voice on your phone and are tethered via Bluetooth and have paired Glass with the MyGlass mobile app, the message will arrive from your Google Voice number. Otherwise, Glass defaults to using your phone’s SMS number with your carrier. If your recipient only has an email address listed in his or her contact information or you’re not running MyGlass, voice action messages use email delivery.
 - Glass also sets a phone number associated with your contacts as their Calling information, which is what’s dialed when selecting them for a voice call.
 - You can still configure default messaging services on a per-user basis—just click/tap on their card in MyGlass and you’ll be able to assign a service to them as their preferred delivery method—email, SMS, or Hangouts. Thus, when you use “*Send a message to...*”, the selected service is used.

- Group conversations are also supported through Hangouts. If you're part of a group chat, that conversation will be available for you to send messages and photos to. The structure of group messaging on Glass typically requires you to add multiple users via the desktop, browser, or mobile versions of Hangouts, which are then available on Glass. This makes the process more akin to email—not everyone in the group will have Glass or may be designated as a sharing contact, so you will need to add them to the chat manually on another device (just like addressing an email message). It's an extra step, but it really does pay off in the final wash.
- MyGlass also manages SMS threads intuitively, forwarding outgoing text messages from a connected smartphone to Glass and displaying full threaded conversations as card bundles.
- “OK Glass, make a call to + <CONTACT’S NAME or CHOOSE FROM LIST>”
 - If you've paired Glass to a Bluetooth-enabled phone, you can initiate a voice call and receive incoming calls. All hands-free, or with tappable menus.
- “OK Glass, post an update to,” “OK Glass, take a note with,” “OK Glass...<invoke native Glass app>”
 - While the GDK gives you a little more leeway into what voice prompts are accepted to launch an installed application, the Mirror API allows you to use two custom voice commands to invoke your service. “Post an update to” and “Take a note with” can be used as gateways into your Glassware for sharing images, videos, or voice data transcribed to text with your contacts. The other wildcard that the voice commands menu provides is as a gallery of your installed applications, written with the GDK. Native programs that you've enabled through MyGlass or sideloaded can be launched hands-free—each has a key hotword phrase, such as “Play a game” or “Translate this” or “Start a timer,” appearing in order of their most-recent use among other trigger phrases and the system commands mentioned previously.
 - The chance that two or more applications will want to use the same trigger phrase is inevitable. To resolve these naming conflicts when someone utters a shared command, Glass displays a secondary menu to disambiguate the command between apps. For example, the Mini Games Glassware is actually a collection of five titles, which all share the same “Play a game” hotword phrase, but so does the social spelling game Spellista, in a completely different Glassware package. Saying the phrase displays a second-level menu that lets you choose between the games, as Figures 2-4 and 2-5 demonstrate. So Glass handles what could be potentially confusing same-name issues for you right out of the box.
 - In case you're wondering how the trigger phrases get chosen, here's the lowdown: all voice commands that stem off “OK Glass” are subject to approval by Google. While Glassware developers working with the GDK may be able to add their

own spoken text to launch their apps during testing, programmers have to choose from a finite set of approved voice commands, as is the case with Mirror API Glassware, as mentioned earlier. This helps cut down on abuse and provides categorization.

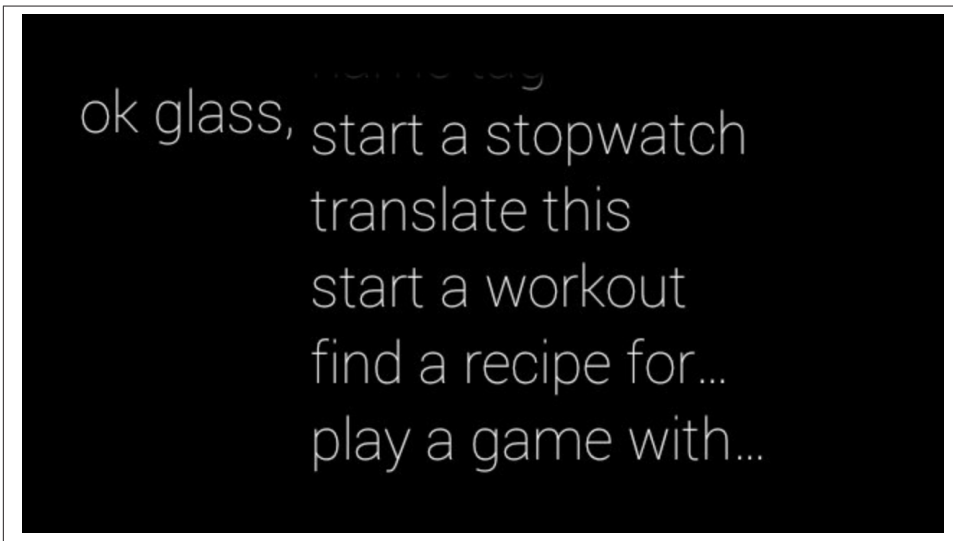


Figure 2-4. Launching Glassware via voice commands

What Didn't Make the Cut

“OK Glass” has become a pop culture phenomenon. It’s achieved that notorious goal of not only being functional within the scope of the product itself, but also gaining top-of-mind awareness to a broader audience, including those who may not own or have never used the device. When you become the subject of countless memes, you’re doing something right.

Google has confirmed that the list of candidate hotwords included “Listen up Glass,” “Hear me now,” “Let me use Glass to,” “Clap on,” “Device, please,” “3, 2, 1...,” “Glass alive,” “Pew pew pew,” and “Go Go Glass.”

What would you have preferred?

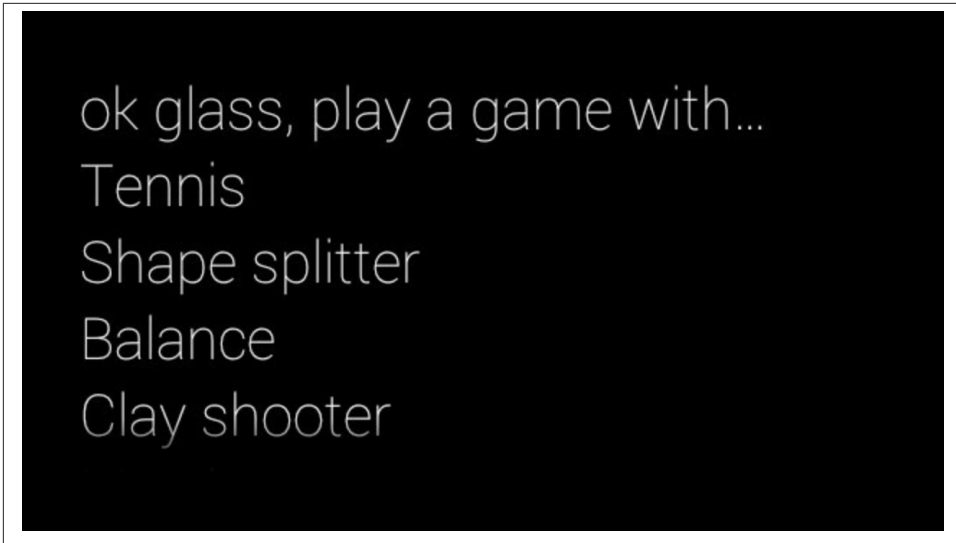


Figure 2-5. Voice commands should be categorical, not Glassware-specific

It's fully expected that future system updates will include more voice actions.

Content Creation in a POV World

The future of content generation is going to take some very interesting turns with Glass and other platforms like GoPro and Action Cam rigs that emphasize the first-person perspective. Whether in telling a story, documenting history, or accidentally stumbling across a significant moment in time, first-person video will invariably change the premium that we as content creators put on the use of body language and gesticulation to create effective media when standing in front of the camera—the emphasis is now shifting to POV cinematography and ad-lib narration.

Even the most seasoned media-savvy presenters may find this transition a little odd, being devoid of the physical traits of stressing points mid-dialogue and relying on shot selection. Jason, who's got a ton of experience as a TV presenter under his belt spanning everything from anchoring newscasts to calling live sports to doing storm coverage to moderating political debates to hosting beauty pageants, is really excited about some of the new opportunities this creates. In this light, we're becoming one-man bands—wholly independent camera crews. And the prospect of someday being able to broadcast video via Hangouts On Air live from Glass from the field means no satellite trucks and other cost-prohibitive infrastructure. Some very interesting and creative opportunities will come from this very quickly.

And the implications for video from the perspective of the user aren't a new thing. The passionate uptake in first-person view videography by the remote control and unman-

ned aerial vehicle communities will simply leave you speechless. Using off-the-shelf and hobby shop components, with cameras that feature amazing stabilization control mid-flight, it's possible to remotely control quadcopter drones and capture and monitor high-definition video that's of uncanny quality. In years past, this required the type of gear and expertise that made it available to largely only professional film crews commissioned for special projects. Now, practically anyone can get involved.

If you'd like to learn more about this exploding space, check out former *Wired* editor-in-chief Chris Anderson's company, [3D Robotics](#).

User-driven media is already changing in neat and very creative ways, so have fun being part of the new generation of filmmakers!

Which Hue Is for You?

How many points has your blood pressure increased when trying to pick out that perfect paint job when buying a car? It expresses your personality and makes a statement about your style. Glass is available in five colors—white (Cotton), powder blue (Sky), primed black (Charcoal), beige (Shale), and orange (Tangerine). Don't sell color selection short—consumer electronics like the iMac, iPod Mini, Toyota Prius, and Nintendo 3DS have made billions of dollars using a varied palette of pastels, endearing themselves to their owners' personalities based solely on aesthetics. We don't dare attempt to quantify the volume of posts throughout social networks crying over why someone's favorite shade of green or purple or red or yellow or pink or the millions of other RGB combinations got no love.

As proof of the level of devotion people have about their selected shade of frame, if you peruse Google+ Communities groups focused on Glass, you'll notice the friendly rivalry between the ardent supporters of each color—Team Sky, Team Tangerine, etc. People really get passionate and align themselves with this stuff.

We'll discuss some of the fashion concerns about Glass in [Chapter 3](#), and opportunities for expansion kits, accessories, and even alternate frame designs in [Chapter 15](#).

Welcome to Wearable Computing!

As a new Glassware developer you join a select community—you're one of the first wave of people adopting (and we hope passionately embracing) wearable computing in your digital lifestyle. Most testimonials that recount their first time using Glass admit it feels awkward, but then is really cool. It really changes how you multitask. Mastering the control system isn't difficult, and adjusting system settings to make Glass your own gives you a true sense of ownership. The overall feeling is one of liberation, putting you in charge of your digital existence.

Some people say they feel right at home using Glass, others feel pressured to prove to themselves they can make it work. The best way to use Glass in your daily life is just that—take it around and give it a spin in your normal routine. Put it on, go run some errands, hang out at the mall, take it into the office, fill out a report, take it to the basketball court for a shootaround, or go get your taxes and/or hair and/or nails done. It's always there when you need it. Do what you'd normally do as a connected individual, but use Glass instead of the tools you've always used. Enjoy the freedom of not having to constantly consult your smartphone for news updates, location-specific check-ins, and social posts in all forms. And watch data literally appear as it happens. Take some pictures, shoot some video, share some neat stuff.

So...off you go. Tell Glass what you need while you take a walk and notice how you can be more of an active part of the world around you, no longer looking down at a screen. And then *take note of how Glass quietly slips out of the way, not bothering you until you need it the next time*. That's the essence of the Google Glass experience and your first step toward being able to Think for Glass. **Chapter 5** gives developers a proper outline of how to design services to best fit this method of delivery.

Welcome to the revolution!

Societal Issues with Glass and How to Avoid Them in Your Projects

Glass makes for a fascinating case study in consumer psychology. In the spring of 2013, it seemed like everyone at some point had an opinion on the then-unreleased and uber-mysterious Google Glass—most loving it, many diametrically opposing it, others deathly afraid of it. One of the biggest challenges Glass faced was that everyone tried to figure it out, pigeonhole it, and create perceived stigmas around it—again, even before it came out. Even after the Explorer Edition was released, many people who hadn't even tried it attempted to pass final judgment on what was essentially still a closed beta.

When word of the release of Google Glass starting getting more of a buzz with the mainstream press, several distinct issues surfaced and rapidly created mammoth global speculation and what renowned tech sector pundit and journalist Jeff Jarvis deemed unnecessary “technopanic” about health concerns, safety, privacy, and wearers' appearance to others. Throughout its history, albeit brief, the window to innovate introduced by Glass is constantly being juxtaposed against the supposition of nefarious actions by some entity.

Because of the intent for Glass to be used by practically anyone, anywhere, and at any time, it's been not only the talk of the programming community, but also a lightning rod for controversy based on assumed impacts and ill-informed perceptions dealing with everything from personal safety of the wearer and those around them to concerns about limitations on personal freedom; debates that have escalated from occasional online musings to arguments being brought to the highest levels of government. And whether as architects or as entrepreneurs looking to cash in on the growing intensity of the global craze for wearable electronics, a litany of issues await you that will have an impact on your Glassware, requiring that you shape its design in ways that avoid these pitfalls to ensure its success. While you're completely able to go the indie route to build, distribute, and support your Glassware all on your own, most projects will want to be

approved by Google and listed as official Glassware in the [MyGlass directory](#). Our goal, again, is to help you craft *great* Glassware, and this means being listed in the official distribution channel and getting maximum exposure—so in this chapter we’re helping you get past the velvet rope and in the club.

Let’s run through some of the more prevalent and lingering societal concerns, their current state, and then present techniques for how to apply the Think for Glass philosophy so you can build services and applications that avoid falling into what could be a very uncomfortable trap. Tell a friend.

(In short, don’t build things that add fuel to the fire.)

Issue #1: Privacy

Early adopters, whether they like it or not, bear the initial burden of being asked probably several times a day, “Hey, is that Google Glass you’re wearing? Cool! I’ve heard a lot about it! That looks awesome! Waitaminnit...ARE YOU RECORDING ME?!? IS THIS GOING ONLINE?!?” Rooted in a natural paranoia about privacy, the first major wave of criticism about Glass dealt with the assumed capability of Glass to take pictures, and record and/or broadcast live video of people to a watching Internet without their knowing and against their will.

The question of whether Glass would include a tally light—an embedded LED that would sit next to the front-facing camera and illuminate to indicate that the user is actively recording/broadcasting—was brought up, which also led to discussion of the fact that the video image being captured is shown in the prism during recording, clearly visible to onlookers, acting as a *de facto* playback monitor for the user, hopefully a visual key for those within range. But for people concerned that they were being surveilled, even this wasn’t enough.

The concerns about the impacts of Glass on privacy went all the way to Congress, as representatives inquired about the potential for the platform to infringe on a user’s existence and possibly involuntarily reveal personal data about them. It was later speculated based on the vagueness of the privacy issues and with the limited official information about Glass that several places would be naturally hesitant (or it would be morally questionable) to let people use the potential recording device in their establishments. These include:

- Banks and ATMs
- Government buildings
- Military installations
- Going through customs at the airport
- Casinos

- Courts of law
- Movie theaters, playhouses, and concert halls
- Places with children or the elderly present
- Public restrooms
- Financial exchanges
- Shareholder meetings
- Clinics, hospitals, triage units, and other medical facilities
- Confessional booths, temples, synagogues, or other places of worship
- Certain types of retail establishments
- Strip clubs
- During job interviews
- On first dates

Where Are We Now?

Overall, worries about privacy continue to be the most-discussed topic surrounding Glass. But this isn't a new argument. This is the latest iteration of the longstanding "nothing is sacred" conspiracy theory that predates Glass by several decades. What's important to understand is that *Glass isn't spyware*. It is not a perpetual surveillance apparatus. Properly designed Glassware informs those nearby when it's working. Nevertheless, its use—especially out in the open—calls for tact and courtesy.

While Glass does enable everyone using it to be an active documentarian of human experience, as did smartphones and tablets before it, there's still some distrust over the fact that everyone is being watched. Just now, the implication isn't that we're all being watched by Big Brother—it's that we're all watching each other.

The concerns are valid in varying degrees and worth noting, but also keep in mind the history behind the general argument. The very same fear was vocalized when handheld camcorders were first showing up in the market and then again when cellular phones with embedded cameras and Internet access were widely available.

Those same people who fear how they'll look with Glass in public are probably the same that pooh-poohed the notion of the idea of a device made specifically for electronic reading of publications, and the very same crowd that knocked the idea that people could interact with video games with body gestures. And the Kindle, Xbox, and Nintendo Wii have done just fine. And those people more than likely came around, too, once they saw how amazing they were.

And by extension, the onus of responsibility is on you as a Glassware developer to encourage people to use the platform the right way.

Think for Glass

It's critical to adopt design and development patterns that don't add to the paranoia. Chief among these for privacy concerns is to *always use the Glass prism display as a preview monitor when the camera is in use* by using a **SurfaceView** in a GDK app. This activates the projector unit and at least gives those around the wearer some basic indication that the device is in use and doing something. The **Glass Platform Developer Policies** specifically state that Glassware to be approved for official listing in MyGlass must meet the following guideline: "Don't disable or turn off the display when using the camera. The display must become active when taking a picture and stay active during a video recording as part of your application."

Several unapproved projects in order to save battery life over extended use like in time-lapse photography applications allow Glass to capture images without the display being illuminated. In most cases you'll want to avoid doing this, lest someone take legal action against you for allowing your users to record them against their knowledge—whether the wearer intended to, or not. Let your Glassware clearly indicate that it may be capturing the moment.

In terms of the societal impact of Glass, there will invariably be pushback based on naive distrust from nonusers that we'll have to endure, but with enough work that will subside if we demonstrate proper and courteous use of the product in public, specifically with capturing photos and recording video.

Issue #2: Facial Recognition

Dovetailing off the previous concern, biometric systems that use facial recognition to identify a person on-the-fly and relay fragments of information about them have long been a dream for advanced computing applications, way before people flocked to theaters to ooh-and-ahh at the futurist predictions laid out in *Minority Report* or see the Terminator acquire a target. So as Glass caught on, more and more people and developers began thinking creatively about instances where being able to implement the technology to recognize someone in real time based on images captured of them could be applied.

Within Google's own product line, facial recognition has been in use with great success. Using powerful machine learning algorithms, Google+ asserts tagging of images of users and other members in their circles with fairly consistent accuracy (with the occasional hilarious misclassification), and it also achieves impressive accuracy with Google Image search. Google Goggles has additionally achieved a strong reputation for strikingly accurate results for general object detection...but not for people. Still, the practice remains

a touchy subject because of security and liability implications. This is the kind of thing politicians just love to legislate into oblivion before it ever sees the light of day, and it's the kind of headline that can cause a stock price to nosedive.

As such, Google put an abrupt end to any speculation about it possibly supporting facial recognition on Glass at Google I/O 2013, when Google Glass project director Steve Lee stated that the company would not be implementing such features in its own Glassware or those programs developed by third parties until privacy concerns were effectively addressed. Nevertheless, outside of Google interest never waned. Two companies in particular, FacialRecognition.com with its NameTag native Glassware app, and Lambda Labs with its Face API, came up with compelling propositions. NameTag maintains its own index of data curated from public web APIs and municipal resources to cross-reference scans of faces. Lambda Labs has been supporting its platform for some time, performing both facial and object recognition.

And in academic halls, researchers at Duke University announced their work on developing a system that sought to identify people in a crowd known by a user by analyzing submitted samples of what those friends usually wear and then using a spatiogram to classify patterns and textures of their outfits.

Where Are We Now?

At the federal level, no statutes are in place to regulate the electronic identification of people. And while not officially sanctioned by Google with its own firmware on Glass, there is a market for facial recognition. Those objecting to the concept of electronic identification have strongly voiced a two-pronged concern: that they'll be scanned against their will, and that information they'd rather not have revealed would be put out there. Advocates of facial recognition technology see it as a means of streamlining activities like electronically exchanging business cards, or a utility for finding out about people they just met or providing advanced customer service or human resources actions through the personalization that ascertaining someone's identity delivers.

NameTag indexes known systems like municipal criminal databases and the National Sex Offender Registry, along with data from public profiles on Twitter, LinkedIn, and Facebook. Profiles that aren't public aren't indexed—because they can't be. The only exceptions are public figures like celebrities and government officials, along with criminals who by their actions have lost their rights to privacy.

The demand for facial recognition apps and interest in innovating within this space are clearly there, but the pushback is likewise substantial. Expect this to be an interesting venue for Glass with many use cases to be introduced—and challenged—every step of the way. Short of Google outright blocking apps from Glass or such applications being found to be illegal and those producing them prosecuted, the interest in creating services that solve complex problems, despite their controversy, will continue to flourish.

Think for Glass

Our friends in Mountain View aren't blurring the lines at all on how they treat Glassware that employs facial recognition or voice print analysis: *Google won't approve such programs for official listing*. The **Developer Policies state, in part**, "Don't use the camera or microphone to cross-reference and immediately present personal information identifying anyone other than the user."

But you don't have to scan and identify humans to provide a valuable Glassware service that uses object recognition—there are Glassware projects that use object recognition in different ways, like **Preview**, which scans movie posters, identifies them, and launches the Glass media player to stream their trailers. Imagine further how helpful a system that identified breeds of dog, species of bird, or types of flower might be.

Elsewhere, offshoots of the idea of facial recognition are gaining momentum for Glass. **Emotient**, which bills itself as "facial expression," seeks to let Glassware determine what a person is feeling when a Glass wearer is looking at them, introducing sentiment analysis to wearables. This could have huge implications for advertising, marketing focus groups, public speaking, and medical fields, among scores of other applications.

The worry against the technology is certainly justified, but Glass indirectly handles the problem of someone being scanned and identified involuntarily—the form factor could be seen as quality control. The camera requires a subject to be at near point-blank range and standing still for at least a few seconds to get a decent scan, so in nearly all cases to get the system to work the subject will probably be aware of and will have consented to the process. You won't be able to identify someone standing in a crowd from 50 feet away and determine if their relationship status is "single and ready to mingle."

To the concern about what becomes known about people who are scanned, Glassware tapping social data means systems are based on data where members explicitly published content available to the world and already being indexed by search engines and harvested by other APIs. This isn't anything you can't find out already. There isn't the ability to scan someone and access their grades from school, Social Security numbers, or bank accounts—the entire experience of being online is essentially opt-in, based on a user's membership to a social platform. It's unlikely you'll be able to find someone's home address, their shopping history, and blood type. Users are in control of what's available about them.

See the case studies in **Appendix A** for the technical details if you're interested in this space.

Being identified by strangers electronically tends to creep some folks out...but there are clever variations on the theme that you can investigate and safely implement.

Issue #3: Using Glass While Driving

As awareness of Glass spread, musings about safety at the public policy level bubbled up, with West Virginia legislator Gary G. Howell proposing a bill that would outlaw the use of Glass while driving due to concerns about distractions, in line with the Mountain State's statute prohibiting drivers from texting while operating motor vehicles or driving without a hands-free device. (To his credit, Rep. Howell praised Glass overall, but was concerned about it taking away from a driver's ability to concentrate on the road.) Similar legislation was also later proposed in Delaware, Illinois, Missouri, Wyoming, New Jersey, and New York.

Months later, the Glass community's jaw collectively dropped after Explorer Cecilia Abadie was pulled over and ticketed for speeding by a California Highway Patrol trooper in San Diego, who gave her an additional citation for using Glass in a moving vehicle. (A section in the [California Vehicle Code](#) states that "a television or video screen" cannot be actively displaying material—with the exception of vehicle information content, reverse-gear cameras for backing up, navigation devices, and GPS-powered mapping tools.) A court commissioner ultimately dismissed the charges due to a lack of evidence indicating that Glass was ever in use at the time Abadie was driving.

This doesn't necessarily mean Glass users are 100% in the clear for using the device while using a motor vehicle, in California or anywhere else, or that Glass would be applicable as a display device under the statute.

Where Are We Now?

While most people don't critically need stock price updates, shipping confirmations for Amazon orders, and birthday reminders for those in their Google+ circles while they do 65 MPH down the interstate, there are still some very valid use cases where live data being pushed to a driver would be advantageous, like construction alerts, road closures, gas prices, and messages from fellow motorists about route conditions.

There's certainly merit in encouraging the safe use of Glass and any other product, but calling for an all-inclusive prohibition on Glass in moving vehicles may be a bit extreme. Having helpful and relevant timeline cards appear in a driver's periphery while users keep their eyes on the road is arguably less intrusive than, say, constantly glancing down at a speedometer, engine light, or car audio controls.

Government is also warming up to the concept of vehicle-to-vehicle communication in order to prevent accidents by rapidly transmitting data between each other. Might Glass become a node in this network?

Think for Glass

Clearly, you don't want to endanger the lives of your users or those around them. [Google's FAQs for Glass](#) stress the need to follow the law and pay attention to the road. As a matter of fact, Glassware projects have been launched based on the concept of actively incorporating Glass in the driving process to help people drive *better* and be *more defensive* out on the roads. [DriveSafe](#) helps keep driver attention sharp by using sensor data to get a feel if a vehicle operator might be falling asleep at the wheel. If so, turn-by-turn navigation kicks in and directs the wearer to the nearest rest stop.

There are many use cases where Glass in vehicles would be helpful, but you should always adopt patterns that keep drivers as focused on the road as they can be, and that means providing a hands-free experience. The default navigation does an effective job of staying out of the wearer's way, even while providing them with turn-by-turn directions in real time.

But with the issue being distracted driving, eliminating it should be your objective. Use voice commands to launch Glassware with microinteractions all the way, using other custom vocal triggers to invoke actions (even if just detecting the presence of a certain sound to perform a certain action, as some Glass games do), keeping interaction and notifications to a minimum, and providing spoken text as output. Don't require a driver to look at the display.

You also don't want to create an app that's either too noisy or too persistent. You might consider reading the accelerometer and only deliver cards to the user when the vehicle is at a complete stop, or provide an application setting that allows the wearer to mute notifications while in transit and turn them back on once they've arrived.

Again, there's a lot of room for development here, but safety first.

Issue #4: Aesthetic Appeal—Is Glass Fashionable?

While more of a personal insecurity about being typecast in the face of one's peers as a cyborg, dork, dweeb, nimrod, spazz, etc., there has been more than just a bit of pushback by the mainstream against the implied aesthetics of wearing Glass. Connectedness and having the ability to conveniently access real-time data and interact notwithstanding, people want to avoid looking awkward when out in public and when dealing with the unwashed masses. (Interestingly enough, the GoPro camera rigs that are so popular for shooting sports video in first-person view tend to be a lot more “camera crew-ish” when worn, but don't face the same scathing criticism.)

Anticipating this hesitation, a deliberate design decision was made by Google to make Glass as visually appealing as possible and position the product as a fashion statement. To drive home this point, Glass has been worn at events by notable names in the fashion community, including designer Diane von Fürstenberg at New York Fashion Week, and

modeled by attractive people in promotional photos that made their way around the Web. When Google unveiled the Titanium Collection of custom-built frames for people who wear prescription glasses, this gave Google a direct inroad for the device to be used by the rather large segment of the fashion-forward population who wear faux frames merely because they look good doing so.

The way Glass has also been marketed across different channels is likewise interesting. Whereas posts from the Glass team on Facebook and Google+ emphasized the device's technical capabilities and applicability in a wide range of use cases, on Instagram Glass was almost exclusively exhibited and positioned largely as a tasteful, artsy fashion statement and content creation tool.

Whereas naysayers lashed out against Glass as dorkwear, Google played up its role as a must-have item of accoutrement for geek chic.



Oh, Peanut Gallery...

One of our favorite observations-qua-snipes about the design of Glass is from [Fast Company's Anne Cassidy](#), who suggested that Glass might have “All the sex appeal of orthodontic headgear.”

That's funny. And we know funny.

Where Are We Now?

Fortunately, if you're into Glass (and judging by the fact that you're reading this book, you probably are), this isn't a concern for you. You've likely already taken several shots of yourself and your friends wearing the HMD and set those as your avatars on social systems. Poring over which color your Glass frame would be was likely not an easy decision, picking between Shale, Cotton, Charcoal or Sky—because who in their right mind would be caught dead with that garish Tangerine on their noggin...right?

(RELAX, we're kidding.)

At the end of the day, the dork factor implication is purely in the eye of the beholder. You love the platform, and that's what counts most. Wear Glass with pride; heck, you paid for it. Rock your gear in public and show others what it can do for them, too. This is our way of life, and we stand together as a community, unashamed of who we are and what we enjoy. We wear our computers as badges of honor, and we're better off because of it. It makes using applications more sticky.

When it comes down to it, can you think of anything cooler than being able to have the entire Internet on your face? We can't.

Think for Glass

Art is subjective and fashion is ambiguous. People are going to have polarizing opinions on the visual appeal of Glass whether you like it or not, and you likely won't be able to change their minds anyway. All you can do is be friendly, honest, and positive. It's the *usage* of it that'll likely be a make-or-break situation and determine how classy or trashy the product appears. As a superstar Glassware creator, don't force a wearer to bark out awkward voice commands that might make them or people around them uncomfortable. "OK Glass...make my alimony payment," or "OK Glass, order hemorrhoid ointment," or "OK Glass, Yankees suck!" aren't the best thing to say at full volume when at the bank, at school, in a religious ceremony, or standing out in the street in the Bronx. Excessively extreme head gestures for program control, being fixated on the display, fiddling with interactivity, or being forced to squeal like a stuck pig won't lend well to someone using the Glassware in public.

This is proper usability planning that we fully detail in the *Design* section.

Issue #5: Augmented Reality

Is Glass true AR? This is actually a pretty easy one to answer: *No*. Yes, there are a lot of things that Glass borrows from augmented reality. And yes, in 2 or 5 or 10 years, the Glass that is available then may have more features that are based on AR. And sure, there are people who are working on applications for Glassware who are doing some nifty things with Glass that certainly augment the world around us. But is Glass itself AR? It seems pretty clear it isn't.

Augmented reality seeks to add layers of information (or remove layers of obfuscation) about the world around us, in real time, in a way that keeps us engaged with that world.

In a way, this sounds a lot like what Glass is trying to do, doesn't it?

Where Are We Now?

Underneath this issue, however, are two much better questions: why isn't it, and what is it?

We can only speculate on why Google didn't try to create an immersive augmented reality system that occupies your full vision. After all, many other companies are developing systems that do exactly this. One likely reason, however, is that the technology just isn't there to do this sort of thing comfortably. Even with decades of work and research in this field, the human eye is still very sensitive to the environment—most schemes cause a great deal of eye strain and potential motion sickness when used for too long. Glass is intended to be something that is always with us, and it wouldn't work too well if we could only stand to wear it for an hour at a time.

And that leads to what Glass actually is. It keeps the wearability and personal nature that AR systems are intended to bring, and it shares the desire to provide information when you want it, but it goes one step further by saying that it stays out of your way when you don't. Using Glass is always done on your terms—you're in the driver's seat.

The design of Glass then takes these two aspects one step further—the design seeks a way to make it comfortable and mostly out of the way by not giving you a choice in the matter. Glass ends up being comfortable to use for just a few seconds at a time: micro-interactions instead of deep immersions. This not only makes it easier on your eyes, but also encourages you to get back to the world around you.

But it does no more than encourage you. Already, we're seeing people using Glass as a platform to experiment with AR systems. More power to these Explorers!

Think for Glass

AR is one of the most active topics of discussions in mobile development circles right now, so it was inevitable that Glass would somehow adopt it. It's expected that Glass will be a leading contender as an AR client once the platform is in its commercial release. Google's **Field Trip** Glassware is a fine example of how to use AR while still being a microinteraction medium—notable places of interest only appear in users' displays as static cards when they're nearby, not as a constant video stream they have to pay attention to. And that's the sort of experience you need to emulate, not something that requires constant attention and interaction.

The fundamental elements of most AR applications are the user's location (where they are on Earth), bearing (the direction they're facing), and an image surface (typically everything seen through the camera, upon which to display pertinent information). Field Trip works for Glass, although it might be considered a “diet” version of pure augmented reality as it produces static cards instead of graphics or text overlaid on whatever backdrop the wearer is looking at. It works for the paradigm.

While supporters and rejectors of augmented reality continue to squabble about how well suited a stage Glass is for AR long term, it undoubtedly will see a lot of traction in terms of Glassware efforts built for it—some bad, some good. If you're interested, check out some of the samples by **Layar**, **Blippar**, or other frameworks for inspiration. Download their SDKs and see how you might blend their functionality with the mindset we're giving you.

The commercial implications for targeted advertising and helpful contextual information are expected to see some interesting concepts, so you'll want to get in early, take notes, tinker a bit, and stake your claim.

Issue #6: Glass Analytics

Developers naturally hope Glass will emerge as a profit center, which means extracting knowledge about how their applications are being used, and which areas are hit the most. There are a few pressing quantitative areas we anticipate architects and marketers asking for with measuring Glassware:

- How many times Glassware-generated screens have been viewed
- Number of users having installed and/or subscribed to Glassware
- Crash reports and logs
- The percentage of video clips that are streamed before the wearer dismisses them
- Ability to see various forms of Glassware interaction (taps, swipes, choices selected) for specific screens and menus
- Most applicable traditional mobile metrics, and any new ones relevant to track, given the uniqueness of the Glass UX

Where Are We Now?

Glass has the potential to redefine traditional metrics and perhaps introduce some exciting new ones. These may be slightly different than the analytics you're used to seeing with tools like Google Analytics and comScore, but we should be able to gain insight into how wearers are making use of our services. One example of this kind of variation on a theme could be the bounce rate for Glass content—if you send bundles of cards to wearers or have long content paginated over several cards, how many of them do users swipe and/or tap through before exiting and moving on to something else? Could referrer logs indicate if Glass cards were generated from an associated web application (like Evernote), a Chrome extension, another Glassware, a separate device, or some other source? We also hope to see tools to assess campaign effectiveness based on information delivered via Glass.

But with the Mirror API being a server-side technology, all of the metrics we derive will stem from that side of the equation. Glass doesn't let you inject JavaScript into cards (at least not for the moment), which means we can't tap the rich well of client-side data that's possible with Google Analytics like tracking exactly where the user tapped the trackpad and how fast they scroll. Third-party tools for analytics and crash reporting are allowed...and encouraged!

Still, we also fully expect to see some new and interesting things about measuring subscription pings. And we should still be able to get basic geographic/demographic reach data and peak-time usage reports like we always have for web apps.

Think for Glass

While at the time of this writing Google hasn't announced a roadmap for how development teams will be able to monetize their Glassware and the displaying of any advertising is prohibited, the implication is that the Glass ecosystem will include a structure to have it be a viable mobile revenue stream for those who produce services and apps for it.

This ostensibly will involve heavy integration for [Google Play Services](#) for GDK installed apps and [Google Analytics](#) for cloud-based services written with the Mirror API, which means tying in usage data for measuring the aggregate activity your work gets and the size of your userbase. Read up on these systems, send Google your feedback, and keep an eye on the Glass developer docs as more is released.

Issue #7: Regulatory Environment—Glass and Public Policy

One of the things that stood out from Google I/O 2012 was how Sergey Brin, upon announcing the Glass Explorer program, said that the effort initially would be available only to software developers, more particularly those in the United States ([Figure 3-1](#)). “We’ve still got a lot of regulatory issues to iron out,” Google’s cofounder stated at the time. The company’s since applied for a number of patents for various technologies relative to wearable computing, and to the Google Glass initiative specifically.

As is the case with many of Google’s products on an international scale, clearly much work remains for the Glass team for getting the product ready for a global audience, specifically with getting it up to standard to work in many countries for wattage and wireless communication standards (in Europe the standard for the amount of radiation a communications device can emit is [slightly higher](#) than the standard allowed in the US). Getting consumer electronics and digital goods globally available is hard work. Google Voice isn’t available everywhere at the moment, and fluctuations in copyright law and licensing in addition to payment processing and shipping issues have long prevented music, videos, and the Nexus line of smart devices in Google Play from being completely available for purchase in all countries.

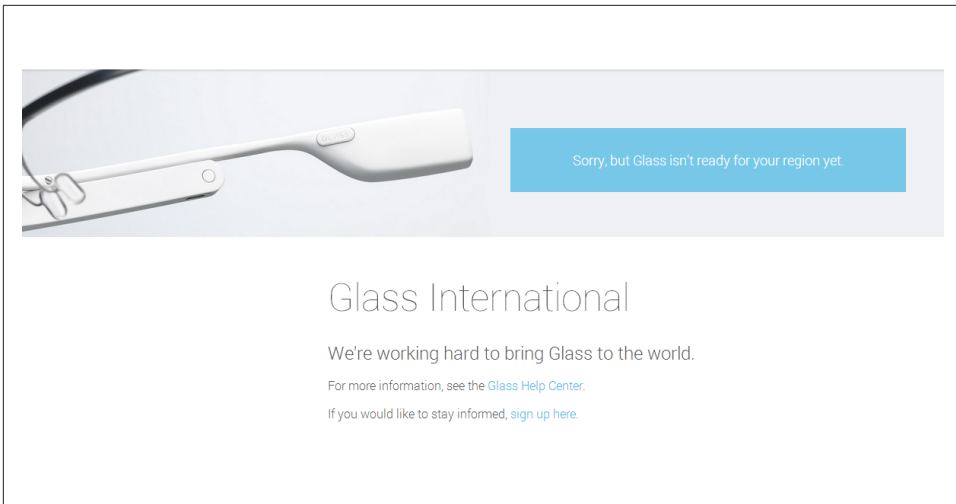


Figure 3-1. Glass was initially only available in the United States

External to Google, politicians in several states as well as across the pond in Britain proposed legislation banning the use of Glass while operating a moving vehicle, as noted earlier. At least five states determined that, in line with current policy, banning computers and recording devices within casinos and shows for fear of cheating at gambling or unauthorized recording, wearing Glass would be prohibited in such establishments.

Google itself has had to react to and modify its own policies—sometimes swiftly, sometimes clumsily—on what Glassware it will allow. The aforementioned decision to not ship Glass with facial recognition technology in lieu of appropriate privacy controls was a major step toward international compliance, arguably at the expense of the feature set. Google also announced a sweeping change to its stance on acceptable Glassware content following MiKandi's launch of its Tits & Glass service that let users vote up/down and comment on adult images; Google modified its policies to ban any content having to do with pornographic content—being defined as “nudity, graphic sex acts, or sexually explicit material.”

Google executive chairman Eric Schmidt has maintained that preemptive legislation is dangerous and that society will naturally adapt and embrace this and new technological concepts. But this remains to be seen.

Where Are We Now?

If politics and legal matters aren't your thing and you just want to enjoy Glass, that's totally understandable. Most geeks put a premium on innovation over legislation anyhow. Google's been no stranger to fighting the battle of innovation-over-legislation, and

as efforts by state regulators mounted, Google began lobbying officials to cease their work in proposing policies that would ban using Glass when operating a motor vehicle.

If it's not ready for your region just yet, rest assured work is being done to get it to you and ensure you'll be able to use it safely and legally. Glass perfectly demonstrates the value of network effects. It is imperative that to maximize the usefulness of the device that it be available to as many users as possible.

Making Glass available worldwide remains the ultimate goal, and that means working diligently to have it be a safe, compliant, and well-understood platform.

Think for Glass

Needless to say, you're going to want to make sure that the Glassware you write is within the scope of the rules and regulations where you live and where people will use it, and as laid out by Google itself in the [Glass Platform Developer Policies](#), which apply to both the Mirror API and the GDK. Make sure to thoroughly review the guidelines to ensure your Glassware is compliant. If you're so inclined, you can take the reins get involved and educate your policymakers locally about what Glass really is and is not capable of, so they don't craft ill-advised legislation that might hamper it...or outlaw its use, even partially.

You can also reach out to a Glass user group nearby to work together on the issue and communicate your feelings to public sector leaders with a unified voice. The last thing our community needs when addressing perceptions is a voice that's not unified.

The Business of Producing Glassware

We've got little doubt that your developer mind is already thinking several steps ahead and wondering about three fundamental components of software distribution: what the strategy is for *archival* storage of your Glassware in a public centralized repository; *discovery*, so that all of the available Glassware can be categorized, indexed, searched, and perused like Google Play, Apple's App Store, or the Amazon Appstore; and *monetization*—what methods you can employ to profit from the value you create.

Not everyone is going to want to make their services and apps available as freeware, and the more complex the Glassware, the deeper the integration with other platforms, the longer it takes to churn out, and the more effort it requires to support it, which means money. And those costs, which can be formidable for things like games and telephony apps, are going to have to be recouped somehow.

Some architects may be considering applying value propositions such as the freemium model, opting to meter usage of their services on users' data transfer (like Evernote or Pandora) or the amount of data stored (like Dropbox). They might also be thinking about leveraging in-app advertising or in-app purchases, or use rate-limiting or metered

API calls. Others will surely be looking to extend paywalls for existing platforms for membership access, or even possibly the a la carte method for pay-as-you-go use. These are big marketing decisions that can dramatically impact the success of your Glassware, and could determine if your work winds up being a cash cow or a money pit. They may also prove to be the difference between them possibly being yet another obscure needle in the mobile program haystack or attaining the rarified air of being the next Fart Sound Generator or Angry Birds franchise.

But rest assured that a very sound strategy for finding, accessing, sharing, and capitalizing on Glass development will be available to bring it all together and create not only great wearable software, but good business models.

What Glass Isn't

One more thing we need to cover is that Thinking for Glass doesn't mean swimming only in the optimistic end of the pool all the time. Sure, we wrote this book to help you keep a positive mental attitude about your investment in wearable technology and get maximum value out of it, but we're also offering you an honest view of what Glass is—and that means recognizing the brutal truth of what the Glass ecosystem simply was not meant to do (at least, not yet). To try to force it to accomplish tasks for which it's not intended is missing the point of why it was invented.

So to give you a proper well-rounded perspective, let's consider the “glass is half-empty” approach. *See what we did there?* That isn't to say that people aren't pushing the boundaries of what is possible, but there needs to be a basis in reality. Pragmatism is a good thing. Still, it's the nature of innovation to test limits—and forward-thinking programmers are doing so everyday. At the same time, you obviously don't want to endanger your users, either directly or indirectly.

It's important to realize that *Glass isn't meant to be a replacement for your other technology tools*—desktop PC, laptop, smartphone, tablet, television, or gaming console. It works wonderfully on its own as a content creation utility and communications platform, and is able to function perfectly alongside your current crop of consumer technology gadgets, tightly integrating so that you stay an active part of the connected community. If you want an analogy, you can compare Glass and your other computing devices the same way you'd compare a motorcycle and an automobile—there are some similarities between the two devices, but sometimes one is better suited than the other. (For the record, we think Glass is the motorcycle in that contrived example.)

The truth of the matter remains that while there are lots of things the Google Glass ecosystem—the hardware, the software, the application model, the administrative controls, the peripheral development potential—does very well, greater still is the volume of things it can't pull off at the moment. But so as not to get your users, and by association yourself, into hot water, it's best to implement best practices and patterns that sidestep

the issues altogether. As you'll see in this book, knowing what works and what's still a touchy subject is a major part of designing great Glassware.

And the other big thing we'd like you to remind yourself of, the most significant "isn't" of them all: *Glass isn't finished*. Far from it. Not by a long shot. Remember, this is Google—the company that proudly touts its innovations being in perpetual beta. The possibilities for how the system can improve, expand, and extend are huge.

We've cited the [Glass Platform Developer Policies](#) on more than one occasion in this chapter, and it's crucial that you review these terms thoroughly at the onset of your work. As someone coming up with ideas for Glassware, it's critical that you have a solid understanding of what Google won't approve. You could still distribute your service on your own as an unofficial service or app, but you won't benefit from the support and marketing benefits that Glassware listed in MyGlass enjoy. You'll also be operating outside of the communications pipeline for new additions and features, so you'll be on your own to come current when things change and this might break compatibility between your Glassware and the Glass firmware.

As early adopters and developers, it's up to us—all of us together—to make sure the technology isn't ill-fitted into situations it wasn't intended for, and isn't subjected to unfair comparisons. We want the technology to live up to its potential. Hopefully we've given you a mental springboard you can use to avoid getting into trouble and find help.

PART II

Design

This second part of the book deals with assembling layouts for both timeline cards and applications themselves. Numerous examples are cited, highlighting winning examples of wearable programs that are cosmetically pleasing, consistent with that of other Glassware vendors, and logical in conveying their data in methods that achieve the Glass goal of being quick, convenient, and nonintrusive. We also detail the core design guidelines for building great Glassware and differentiate how designing wearable software isn't just like designing for another mobile platform.

- Chapter 4, *Thinking for Glass: How Glass Is, and Should Be, Personal*
- Chapter 5, *The Five Noble Truths of Great Glassware Design*
- Chapter 6, *Glassware Antipatterns: Avoiding Poor Design*

Thinking for Glass: How Glass Is, and Should Be, Personal

Glass as Personal Technology

By bringing technology closer, Google Glass keeps technology out of the way. There. That's Chapter 4...all of it. Hope you enjoyed it. Go back and read it aloud. And don't blink, because you'll probably miss it. Good thing we're not getting paid by the word.

Oh...still here? Good, because the devil is really in the details. Read on.

Cheap laughs and literary parlor tricks aside, that simple statement stands as the core underlying theme for Glass. And we're using this to kick off the *Design* section of the book. And nonintimidating as it may be, it's often the hardest thing for people to grasp when thinking about the platform. Isn't Glass supposed to be an always-on device that's built for microinteractions, sending us stuff as it happens and letting us capture events around us? And by that merit, coupled with the fact that it's sitting on our heads, doesn't that make it omnipresent? Yes, absolutely. But there's so much more than that.

And, appropriately, so much less.

As we've seen, Glass is a platform that makes for an extremely lightweight solution—figuratively and literally—for staying connected and interactive while experiencing the world around you. The goal of Glass is to put you in control of your life by not taking you away from living in the moment. Glass isn't just another mobile device, and getting the most out of it means thinking about technology in a completely new and different way—probably more than you ever have before.

And when it comes to building Glassware, this means thinking about structuring information and experiences in new ways, being more terse than you're probably accustomed to, and really using the timeline and cards to your advantage. The secret is making the user's atomic world part of the digital UI, not just basing their interaction with the

system on controls, input, data, and bandwidth. This means designing software that plays up the benefits of the technology being personal.

In this chapter, we're going to cite several examples from Glassware vendors that do it right. We'll highlight what we feel are the best of the best, and show what you can learn from them in putting together your own UIs and features.

Best in Show

Let's examine some amazing services that in our opinion really separate themselves from the pack in terms of their winning design and the experience they deliver. As developers, we appreciate beautiful engineering and design approaches that are logical and creative, and there's always something you can learn in positive ways with any project. But it's no surprise that those that get the nonexistent *Think for Glass Blue Ribbon for Awesome Glassware Design* are household names, known for their form-and-function excellence across all platforms. On Glass, they enforce proper branding so you know who sent which cards, and have really put thought into making their presence on wearables the best it can be.

These top-shelf productions are services that provide perpetual communication ability, but on Glass do so in a way that's not bludgeoning the wearer with nonstop alerts about incoming information, each figuring out a creative way to deal with what can be huge inbound conversation streams. And they also get the nod for fulfilling the promise to make content creation easy—users are able to initiate new posts and interact with their social connections in mere seconds, not making the act of staying connected a laborious chore.

By the same token, nobody's perfect and even the creme of the crop has the occasional area where it might need revision. This, too, is a learning experience.

So let's take a look at three examples of truly outstanding Glassware: *Twitter*, *Gmail*, and *Google+*, discovering what design decisions make their brands stand out. What's important to keep in mind as you read these is that like Glass, these aren't just software products, they're entire platforms with ecosystems of their own. They each feature massive streams of inbound data, very complex UIs, and memberships in the hundreds of millions. But by using the Glass paradigm the right way, and not trying to force older models, they've managed to create really usable services and create powerful Glassware.

Twitter



As we went to press, Twitter removed its Glassware for new users (but you can still get notifications and interact with others through their mobile app connected to Android Wear). We're including this discussion about the now-defunct Mirror API-based service in this section, however, because we continue to think it is a good example of Glassware design.

Twitter's Glassware supports full read/write capabilities for a user's timeline using a creative solution (Figure 4-1). The Glassware empowers users to share images they've taken with Glass, as well as receive mentions and direct messages and retweets for their own account. For getting updates, Glass users can enable mobile notifications on Twitter.com for select profiles they follow, which are pushed to Glass as their authors update their feeds. Tweets can be replied to via voice, retweeted, and favorited. It's a proper translation of its core service that curates a member's broader timeline, without inundating the Glass wearer with a constant barrage of notifications that would kill the battery. As Figure 4-2 shows, you can also enable mobile notifications from Twitter's web client that get pushed to Glass.

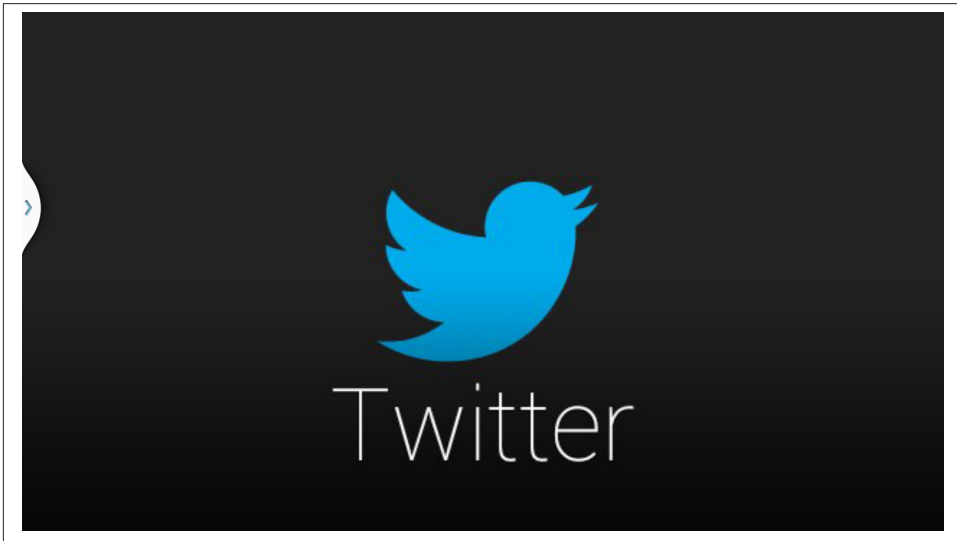


Figure 4-1. Twitter's Glassware

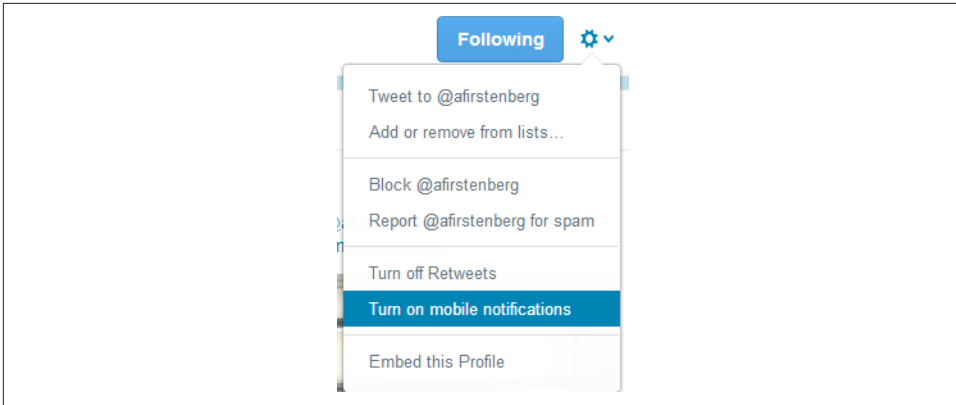


Figure 4-2. Get your followed tweets on Glass

Twitter also happily puts into practice a design pattern that many people don't realize at first—your Glassware can have as many custom menu items as you wish, not just one, and it does this to translate its own commands to Glass, providing Reply, Retweet, and Favorite right alongside a custom Delete command, all wired up to its own API so the experience is what you'd expect and what you would want from a head-mounted Twitter client. The tweet body itself is front and center, while still giving space to see the user's real name and handle. Direct messages are clearly marked with a "DM" in the left side of the footer in cards so you know when something isn't public, along with the timestamp in the right corner of the footer. Since it seamlessly implements its own functions on Glass, Twitter doesn't use any of the stock Glass actions, so it's its own self-contained platform. Path is a great example of using multiple custom menu items, too, letting members assign an emotion to a friend's post.

In contrast, Evernote deviates from this route, opting rather to implement the Mirror API's stock SHARE built-in menu item, which uploads a resource to its system and stores it in the generic *All Notes* notebook with the default title "Note from Glass." The hierarchy it uses for organizing its notebooks would be complex if replicated on Glass and likely require multiple nested menus, so they went with a simple solution that really works for them.

Twitter also demonstrates a clever usability technique—keeping track of state for menu items. If you've previously retweeted or favorited a juicy tidbit or hilarious zinger, the menu items from that point on use a different icon to appear colored (instead of their default white) and read "Retweeted" or "Favorited" as noted in Figures 4-3 and 4-4, just like they do across Twitter. This is not only good UX practice, but it deters users from taking the same actions repeatedly because they have no idea if their action went through the first time—which could potentially devour the number of calls against the Mirror API your project is allotted from Google daily. We'll be discussing this quota and how to manage it in [Chapter 7](#). (Oddly though, Twitter's menu items can be tapped even after

they've already been selected. They don't do anything because their state has already been changed, which is more a gotcha of the Mirror API since all menu items are tap-able and you're unable to nullify that behavior. We'd like to see this changed in the future release.)



Figure 4-3. Twitter's Favorite menu item



Figure 4-4. Menu items change based on their state

As far as letting members compose and publish tweets, Twitter’s Glassware handles images shot on Glass by letting you share to Twitter, as well as handling text input from the “*Post an update*” voice command. However, as of the time of this writing you can’t share video from Glass with it, so don’t expect to have a clip appear with a shortened URL and get tweeted out as a Vine post. We hope this will soon be resolved somehow by Twitter or a third-party tool.

You also can’t see the total social engagement statistics for a tweet, like other mentions in a threaded conversation or who else has retweeted it. This is deliberate—some things are more appropriate on other more powerful platforms. It’s intelligently not a crude port of the more complex web and mobile layouts, but an effective redesign of the data to properly fit the parameters of Glass—it retains all of the properties of a tweet with interactivity features while keeping things very lean and light.

Design takeaways:

- Multiple custom menu items
- Tracking state changes in menu items
- Web-controlled curation of user’s social stream
- Not a straight port of the more complex web UI, but an effective redesign

Gmail

The Gmail Glassware delivers your electronic mail to you just as it would on a mobile app or desktop web client, with the twist that you can reply to messages via voice dictation, freeing your hands and eyes to work on other things (Figure 4-5). To keep the load light and your notifications at a manageable level, your subscription delivers messages only from your *Important* folder, which Gmail algorithmically calculates based on recipient frequency, and which can always be tuned by the user by moving messages around to other folders—it learns about you as you use it.

This is really helpful and is an object lesson for applying a custom filter for data so that the entire stream isn’t imposed on a Glass user. And since items in *Important* are the result of machine learning algorithms listing only those people you correspond with the most, you won’t have to worry about spam on Glass. (Now would be the appropriate time for you to jump for joy.)

Gmail also uses a message structure we’ll see in many other scenarios—a mosaic of avatars on the left of the card lets you quickly assess who a message is from, so you can determine if you want to take further time to read the message, have it read to you, reply to it, or if this is important enough that you should pull out your phone. Like Twitter, Gmail also applies several of its own custom actions as menu items for *Archive* and

Star, along with the Glass stock versions of *Read more* and *Read aloud*, and custom versions of *Reply*, *Reply all*, and *Delete* (Figure 4-6).

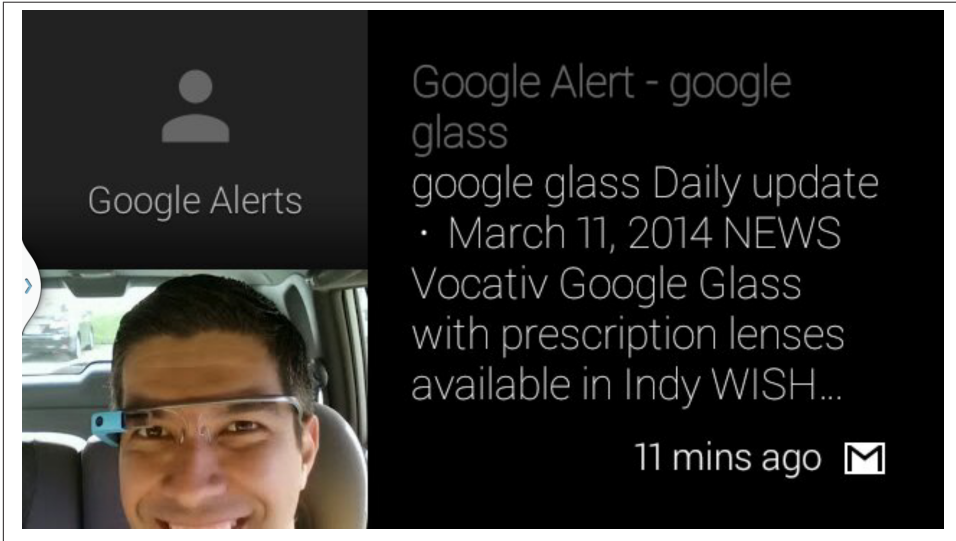


Figure 4-5. An inbound Gmail message



Figure 4-6. Gmail's Archive menu item

It makes use of the ability for longform text to be automatically paginated over several cards, as an entire message can be displayed in addition to organizing conversation threads into their own distinct bundles. The cover card is also different from other cards within its bundle, noted by the subject header being on the cover card and the use of an ellipsis to indicate more text. Gmail doesn't insert new cards any time there's a reply to an ongoing conversation, but more intelligently updates existing ones in place and as such reseats a bundle to the front of the timeline with the new message as its cover card. This is a critical best practice for Mirror API programming you always need to keep in mind, and one we'll be talking about in Chapters 9 and 11 when we get into the mechanics of the timeline and sharing.

Additionally, Gmail demonstrates a key part of Glassware design: *adding value by omission*. The service doesn't include a menu item that allows you to forward a message to someone else on the Internet, which makes sense because with the current composition of your contacts, you'd only be able to do so with 10 people, and even mighty Google's voice transcription powers can't dictate even a simple email address. Really consider this as a lesson in how you apply your logic to Glass, realizing that some things just can't be done.

One area where Gmail on Glass could do a better job is in attachments. Currently there's no visual indication within a Gmail card that it includes a file at all—which would, of course, prompt users to use another device or a laptop to review what's been sent. We can't load PDFs in Glass or Excel spreadsheets on Glass, but maybe being able to view an image would be a nice touch to an already amazing application.

Design takeaways:

- Updating bundles, using pagination over multiple cards to reflect conversation threads
- Effective layout of headers and message body
- Custom menu items
- Filtered data streams

Google+

Google's social layer pushes cards to Glass only for posts that the wearer is directly sent or mentioned in or for conversations that were posted with Glass, which like Twitter uses a subset of the user's normal data stream, not the entire shebang. It's a really neat way to use Google+ across devices to share content and communicate with people. The simple card UI shows how a system with lots of moving parts can be adopted within the static prism display—Google+ on Glass doesn't look anything like Google+ does on the desktop web, in mobile web browsers, or on tablets. And it shouldn't.

The Glassware also gives users the ability to engage in posts—reading and adding comments to conversations, as well as +1’ing them. Google+ on Glass accepts text, photo, and video posts as well as link shares, and was the first Glassware service that really let you involve your social graph and keep you in touch with your connections. Google+ achieves this by registering itself as a recipient for most types of multimedia formats as well as links to places on the Web. It opens up its full range of engagement as a sharing contact—it lets you share content with the people you’ve added in MyGlass, entire circles of people, and Communities groups you belong to. This is a new way of thinking about how data exists, as data isn’t just shared to an app like on other mobile platforms, but to people and services alike.

The Glassware also does a masterful job with funneling notifications. There are a ton of interactions that Google+ could alert you to, but only the most appropriate make it to Glass. If your profile is tagged by someone, you get notified once on Glass with the card shared with you, but then not again unless you jump in the conversation, just as Google+ behaves on other clients. If you share a photo or link with a Google+ connection of yours from Glass, your posts will append the hashtag #throughglass—which is a great design tactic to consider for self-promotion for your platform—and then also register you to receive notifications on Glass, in an intuitive way. Your headset will only sound the Glass alert tone when a new comment has been added to the post, not when someone has +1’ed it or shared it. But whenever anyone on Google+ engages with you, the original card for that post gets updated with the current +1 count and number of comments, represented visually in the footer (Figure 4-7).

It’s very well done and makes good use of the cardspace without feeling forced or cramped.



Don’t Sleep on Using the Footer

The footer area in cards is a great place to put tiny fragments of information. By convention, the right side of the footer is reserved for an icon identifying your service and for timestamp data. But on the left side, you’ve got carte blanche. As we’ve seen with Twitter and Google+, helpful information like direct message indicators and social metrics, respectively, can be included with only a few of characters or an icon and can make a huge difference in describing content.

You can also get creative by using the footer in cover cards for bundles, as the Hangouts Glassware does by including the name of the user chatting with you.

Be wise though—don’t just slap something in there just because the space is available. Supplementary information that helps describe the main body content is always best.

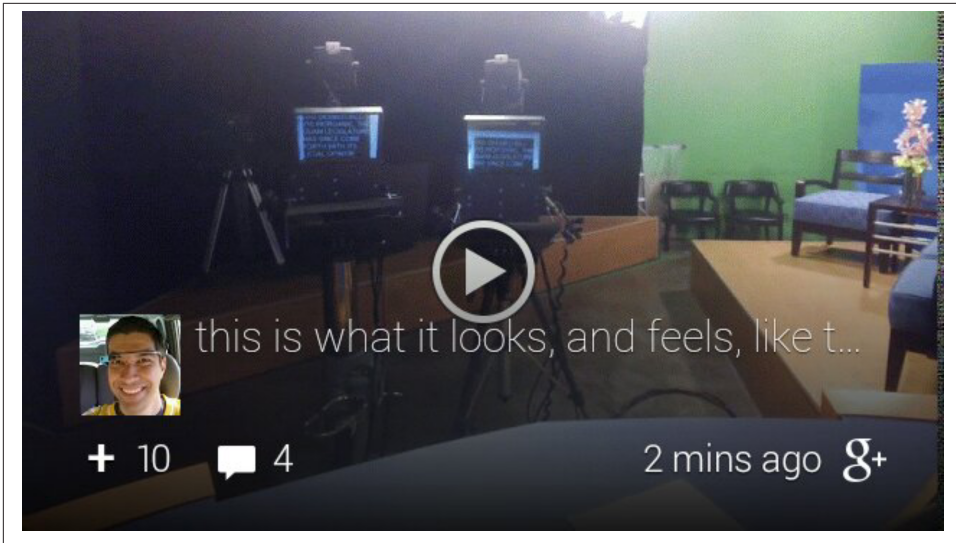


Figure 4-7. A Glass video shared with Google+'s Glassware

However, the biggest room in the world is the room for improvement, and Google+ isn't without its minor oopsies. It's an elaborate application that has different view conditions that aren't always displayed as you might anticipate. For example, if you're mentioned or tagged in a post with a video, you see a card with the clip attached to it, with the `PLAY_VIDEO` menu item available to trigger playback. But this also means you're unable to tap and swipe through the comments that are available for that thread, because whereas you'd normally tap on what would be a card bundle and iterate through comments, tapping with an attachment brings up the menu items for that card. In that case, comments are unavailable.

Additionally, early builds of the Glassware had an issue when encountering internationalization within posts, converting UTF-8 characters to ASCII, due to the way that Google+ supports foreign character sets on its web and mobile clients to allow comments by people typing in non-Roman alphabets like Cyrillic, Farsi, Hiragana, or Greek, which drives the helpful Translate feature. For example, if a post including the line *I'm going to say "I'm finished with dinner & I'm ready for dessert!"* would appear in cards with URL-encoded text, it would be `I'm going to say "I'm finished with dinner and I'm ready for dessert!"`, which would not only be a nightmare to read visually, but also would come out horribly when the Read aloud action was selected, announcing every escaped character verbatim. Make sure that if your Glassware incorporates multiple languages, you test and retest and properly use your web development framework's internationalization features to avoid this.

These are minor gotchas, but we have full confidence the Google+ team will continue to work on it and find a creative way to incorporate much of the feature set.

Design takeaways:

- Reporting engagement metrics
- Social notifications
- Support for text, photos, videos, and link shares
- Platform promotion
- Internationalization
- Use of the footer
- Custom layout that fits the form factor

We Literally Have Google+ to Thank for This Book

A funny anecdote we share is that Google+ was, in no stretch of hyperbole, the driving force behind this book. It allowed us to meet and become friends, it's how we collaborated on writing it and endlessly debated what approaches we would take, and it provided the infrastructure that galvanized the entire effort. After we outlined the chapters and started laying out the initial material, we still needed a hook—that one thing that readers could glom onto to really get what it means to Think for Glass.

So while driving home late one Saturday night from giving a talk about Glass, Allen had an epiphany. He suddenly was hit with the creative vision about what our philosophy meant, and hurriedly recorded a video of himself dictating our concept that “Glass is designed to live in your world, not for you to live in Glass’ world. It is meant to adapt to your life, not for you to adapt to how it does things. Your apps are expected to behave the same way.”

Still on the road, he thanked his muse and shared this gem of a revelation using the Google+ Glassware with Jason, who was writing a chapter on the other side of the world on Guam and began responding on Google+ on his tablet. We had a series of exchanges that solidified the concept we’ve been presenting to you—cross-platform, collaboratively.

Pretty cool, huh?

Winning Glassware Design Takeaways

Now let’s look at some other notable design instances from the initial salvo of Glassware vendors that leverage the Glass experience (Tables 4-1 and 4-2). Many are existing web brands and were challenged with transposing their in-place ideas to the Glass UX. Check them out on Glass yourself and pay attention to both what they do and how they do it.

Table 4-1. Winning Mirror API Glassware takeaways

Glassware	Genre	Design takeaways
Evernote	Productivity	Integration with existing APIs, send-to-Glass from web
Path	Social	Rapid development (a full-stack port to Glass took a few weeks), multimedia resource sharing
The New York Times	News	User configuration, delivery batching, photo captioning
Elle	News	User configuration outside of Glass, web-based reading queue
CNN	News	Web administration, user-defined content selection, time-based delivery
Facebook	Social	Sharing with varying social scope, integration with APIs
Tumblr	Social	Read/write access, integration with API
Ice Breaker	Game	Using Glass as a casual gaming client

Table 4-2. Winning GDK Glassware takeaways

Glassware	Genre	Design takeaways
Spellista	Game	UI outside of the timeline, game controller driven by accelerometer data
Compass, Stopwatch, Timer	Utility	Long-lived live cards updating content constantly, in the same way that Google Now cards work; real-time sensor data
Strava Run, Strava Cycling	Fitness	Capturing motion information, real-time visualizations, social integration; provides sporadic feedback only at the most important times
Word Lens	Communications	Real-time processing of video data

The main thing to remember about these examples is that they exemplify one of the key aspects of how to Think for Glass—each is a well-designed service specifically written for the idiosyncrasies of the platform, not just force-fitting an existing mobile website or a clumsily ported native app or ramming an RSS feed at the user. They conform to the Glass UI restrictions, emphasize minimal user interaction, leverage the system’s low-bandwidth ideals, exploit the platform’s numerous hardware capabilities, and take into consideration the wearer’s behavior during use.



How Are Pinned Items Ordered?

Since live cards in GDK apps will appear when Glass wakes up instead of the home screen, how will you know which pinned item a user sees if they’ve got more than one live card running? Just like the timeline items that sit to the right of the home screen, the ordering of pinned items uses LIFO (last in, first out), with a twist: live cards always take precedence over any static cards, regardless of when they’re pinned.

And the most impressive thing about these services is that they’re not all carbon copies of each other laid out across different industries. Each demonstrates something distinct about the Glass ecosystem. They’re built for maximum effectiveness by emphasizing

some of the best aspects of the platform through playing up the features of the Google Mirror API and emphasizing positive minimalism, or leveraging the low-level control and system capabilities with native code via the GDK. Despite the first-glance restrictions of its display, Glass is extremely flexible as a stage for third parties to build upon.

And that's the point: understanding what Glass is and is not and what it can and cannot do, and repurposing platforms for that model. These examples prove that if done right, a service can be extremely useful—but more than that, they demonstrate that Glassware can achieve something in interesting, helpful, convenient, and fun ways!

Designing with the Think for Glass Mindset

Let's revisit the rather heavy concept we floated in [Chapter 1](#), in which we laid out what it means to Think for Glass. Quoting ourselves:

Glass is designed to live in your world, not for you to live in Glass' world. It is meant to adapt to your life, not for you to adapt to how it does things. Your apps are expected to behave the same way. Everything else stems from this basic idea. It means that anything that works through Glass should be secondary to the world around the person wearing Glass, and that an app should never expect otherwise.

This philosophy means approaching program design in a way that not only remains cognizant of the user's surroundings—it actively incorporates them. Central to this is keeping the experience personal to the wearer. You need to be able to read your user's mind—which is no small task. The Glass experience was meant to tightly involve contextual signals, so involving sensors, location, real-time data delivery, and one's social graph is part of what constitutes the user interface. Most important is tailoring your design around users' environments (the people, places, and things surrounding them) at the moment that they use the application. Modern mobile applications took design into a new stratosphere by getting architects to base their UIs around accelerometers, gyroscopes, and GPS data—Glass now flies even higher, bringing the full range of context into the picture.

Glass progresses human–computer interaction in that the environment of the user is a top priority for the design decisions you'll make when creating Glassware. Situations that the person wearing Glass encounters may be ripe scenarios in which to use your program. Or, they may be absolutely inappropriate. This is the delicate balance you'll need to keep in mind when using control mechanisms like voice commands. Imagine the trouble you'd get some poor unsuspecting user into if they had to yell out “*Fire!*” to play a game while in a movie theater in which you shoot rockets at aliens...if they happened to be near federal agents. Not good.

If this need for clairvoyance is intimidating or at the very least confusing, don't worry. Let's consider a couple of illustrations from the Glass system software that reinforce this idea.

Vignettes

Glass includes the incredibly popular ability to create vignettes. This simple idea lets a user take a screengrab of whatever timeline element or application screen is currently being displayed in the prism, and then sets that image against whatever backdrop the user is looking at in real life (Figure 4-8). The overlay is composited with the background image and saved on the timeline, which can then be shared with contacts or Glassware like any other applicable resource. With only two input actions (holding the shutter button down to take the screengrab, then selecting the Make vignette menu item), the user has captured the moment from his unique perspective of the world, and maybe added a funny remark about the environment around him, or created something neat without escaping the moment.

The wearer is using his own interpretation of the world as the backstage for the application. It's the key element in how the feature is used. He could snap a photo and include that picture within a picture, or take a screengrab of any item on his timeline and include it on top of a scenic shot of wherever he happens to be. And whatever the situation, it didn't require the user to meticulously manage a complex application menu or look down and meticulously negotiate controls. With vignettes, you literally never look away from what you're doing.

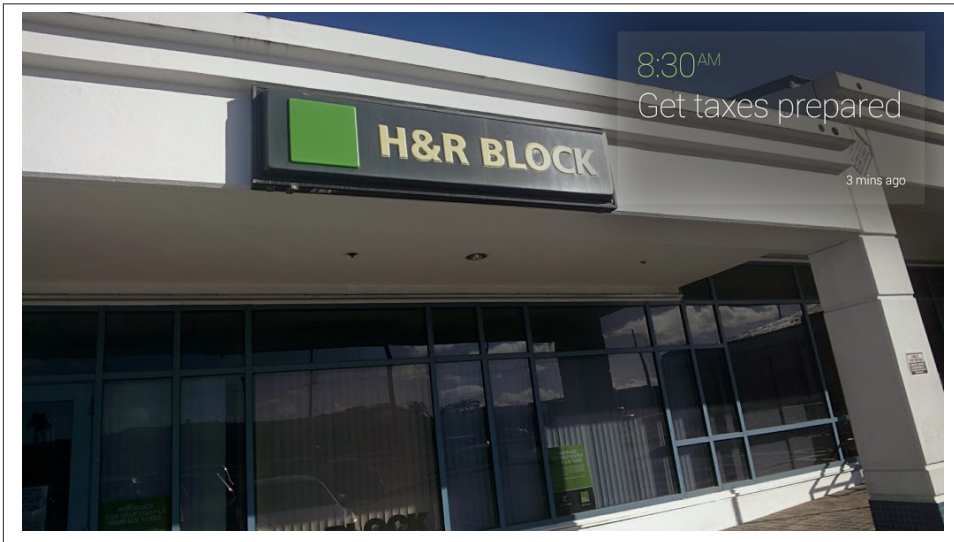


Figure 4-8. Vignettes on Glass

Vignettes are system software, available by default to nearly any piece of Glassware, even those from third parties, and even in native apps. A third-party service for Glass, **Vignette Postcards**, even jumped on this idea and allows wearers to apply seasonal greeting

card designs to their images. People have gotten extremely creative in combining their Glass content with real-life views using vignettes. This is a stellar example of how both worlds merge for a seamless experience with a personal touch. And it only took a couple of seconds to snap-and-tap to capture.

Google Now

We'll be riffing on contextual information a lot throughout this book, so strap in. Using a number of signals relative to the user is a major aspect in making Glass content distinct, relevant, and valuable. And nowhere is this more evident than in the platform's integration with Google Now as noted in Figures 4-9 through 4-13. The feature is touted as "the right information at the right time," based on your physical location, the time of day, what you've searched for in your web history, appointments you've made in Google Calendar, messages you've received in Gmail, and various other preferences about the world around you.

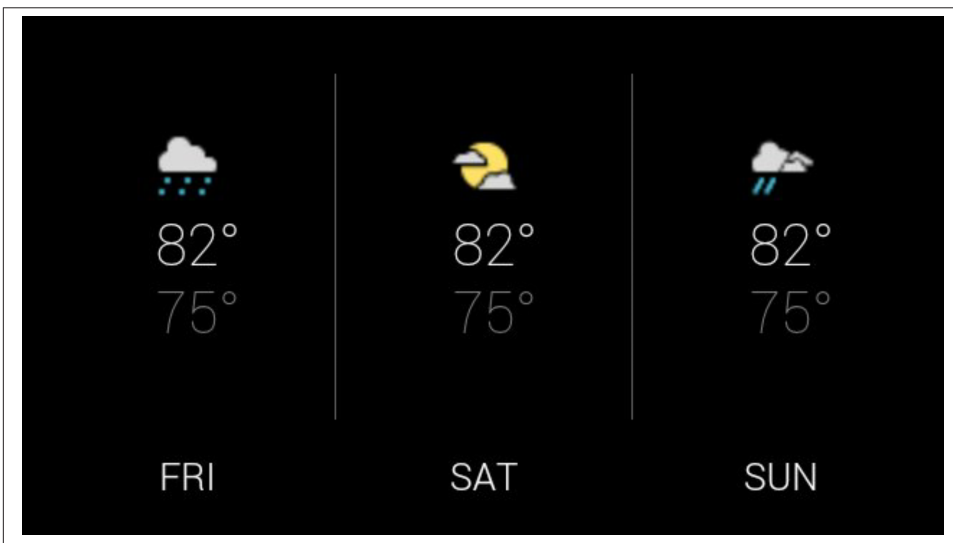


Figure 4-9. Google Now's weather card

Google Now keeps track of your activity and learns about your behavior, generating cards based on patterns. Its Glassware uses a variety of templates to format the various types of data it displays, not sticking to a single one for all types. The stock price card isn't laid out the same as the sports scores card, which doesn't have the exact same formatting as a card with a map, or one with an upcoming appointment. This demonstrates flexibility within a single application.

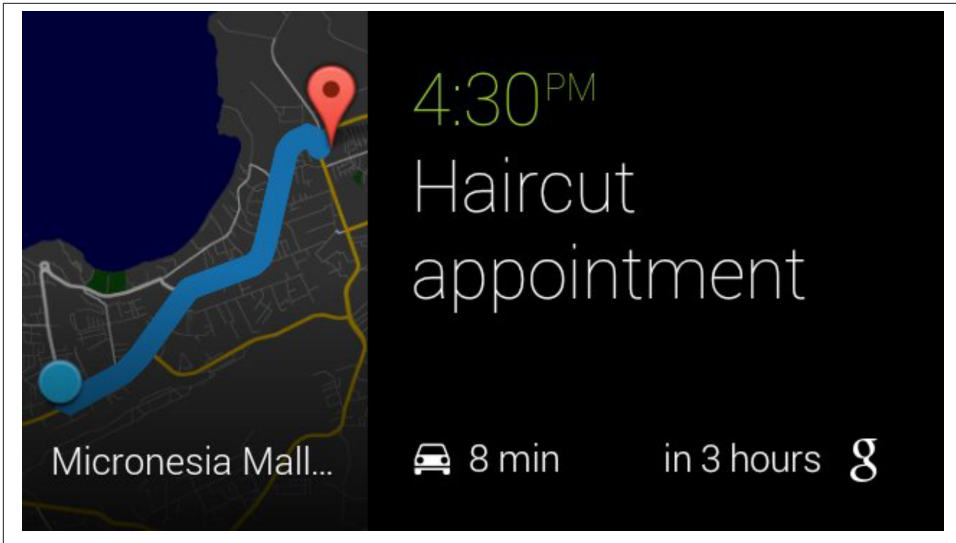


Figure 4-10. Appointment reminders from Google Now

The cards that Google Now generates are also tied into the larger scope of Google Now on mobile devices and the desktop via Chrome notifications, which illustrates another example of great design: the ability to dismiss items on one platform and have them cascade across other clients. When you dismiss a card about a flight delay on Glass or on your phone, it syncs with other platforms to not appear there. This is a best practice you should emulate in your own projects, too.

Figure 4-12 is a great illustration of how Glass can pull in data from different sources you've expressed interest in and present them in a useful way. The most recent stock prices you searched on Google Search are listed as cards within a bundle you can drill down into. This is, perhaps, one of the best examples of contextual computing.

And, most importantly, the cards are created and delivered when in the appropriate context—again, with sensitivity to the user's time, location, and things happening around her, and in terms of events she is participating in, people with whom she's connected, or occasions she's keeping tabs on. This is another example of making the user's real-life activities a core part of the application experience.

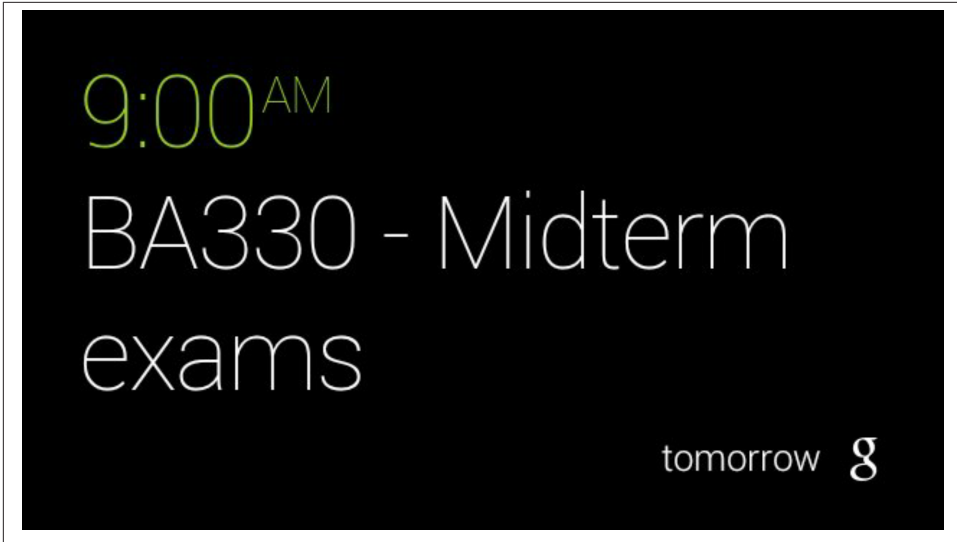


Figure 4-11. Upcoming events

TSLA	+7.08	(+3.02%)	241.49
ZNGA	+0.14	(+2.48%)	5.79
FB	+0.78	(+1.11%)	70.88
TWTR	+0.48	(+0.89%)	54.50

g

Figure 4-12. Tracking stock prices

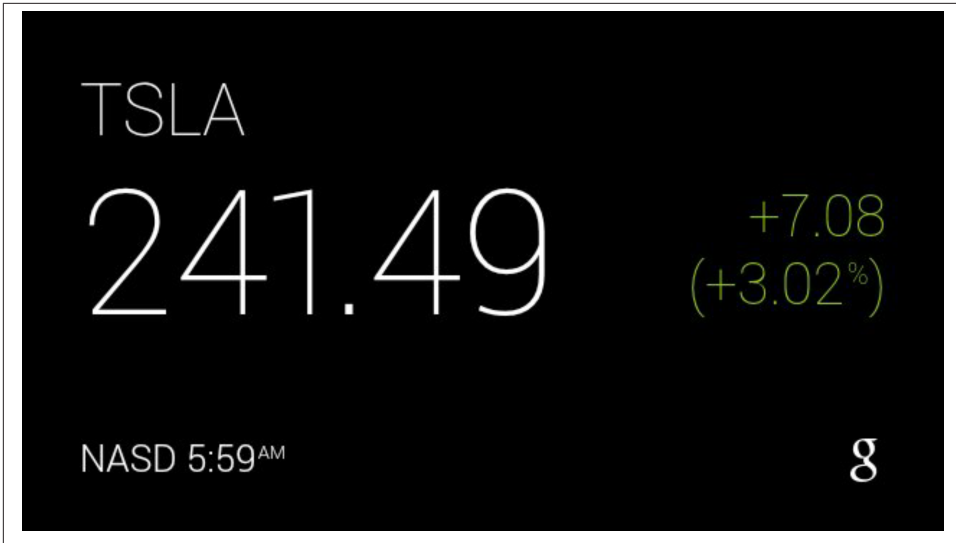


Figure 4-13. Individual securities

Google Search

The canonical way to use Google also has a strong presence on Glass, and serves as a great lesson on how to handle the very complicated task of fetching and showing results from web-wide queries, which is intensive from processing and interface standpoints. Conducting voice-driven searches requires connectivity and is comprised of two steps: preparing the query and displaying search results.

Try doing a search on Glass yourself, and take note that the “working” status bar at the bottom of the card by the Google Search app then returns a result set back to Glass. The use of iconography and visual cues about what steps to take are well laid out, using the familiar animated microphone when input (or background noise) is detected.

As good defensive programming, Glass enforces a timeout if no vocal input is detected within a few seconds and terminates the job; and if the connectivity isn’t good enough, it times out and shows a card asking the users if they want to check on their connection settings. Both are good visual cues about the work being done behind the scenes and provide the application with some time to do the heavy lifting, which in this case is talking over the Internet. And this all happens in mere moments.

Most information retrieval services won’t be anywhere near this fast, so check out [Appendix A](#) for a couple of ideas you could play around with while the results are being compiled if you need search features for your project.

For this example in Figures 4-14 through 4-16, we’ll perform a search on Glass for “sleestaks.”

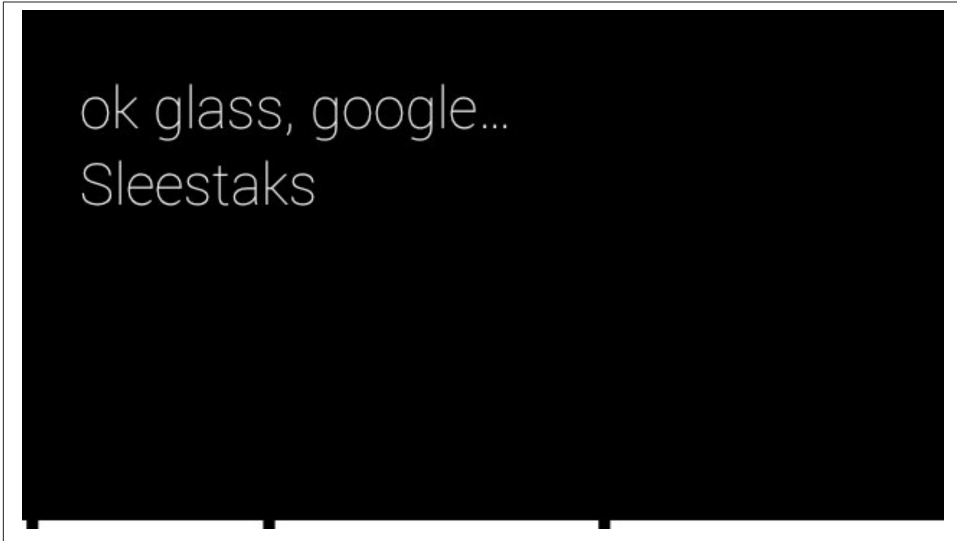


Figure 4-14. Voice searches

So **Figure 4-14** is the querying-and-assembly phase. What about displaying the results back on Glass? By default Google Search lists a small collection of matches, and in cases where the topic has a match in Wikipedia or Google’s Knowledge Graph, reads the first item back to you automatically.

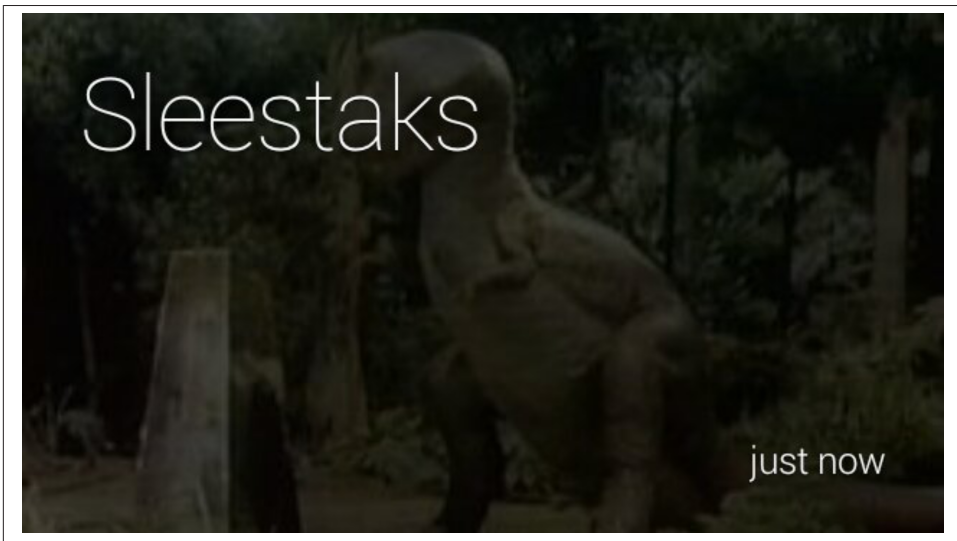


Figure 4-15. Matches to the sleestaks search

For archival purposes, the search results persist on your timeline as a bundle of cards so you can refer to them later, not unlike [Figure 4-16](#). This is a big win in terms of usability and system optimization, saving the user from having to go across the network to review past searches. At scale, this adds up.

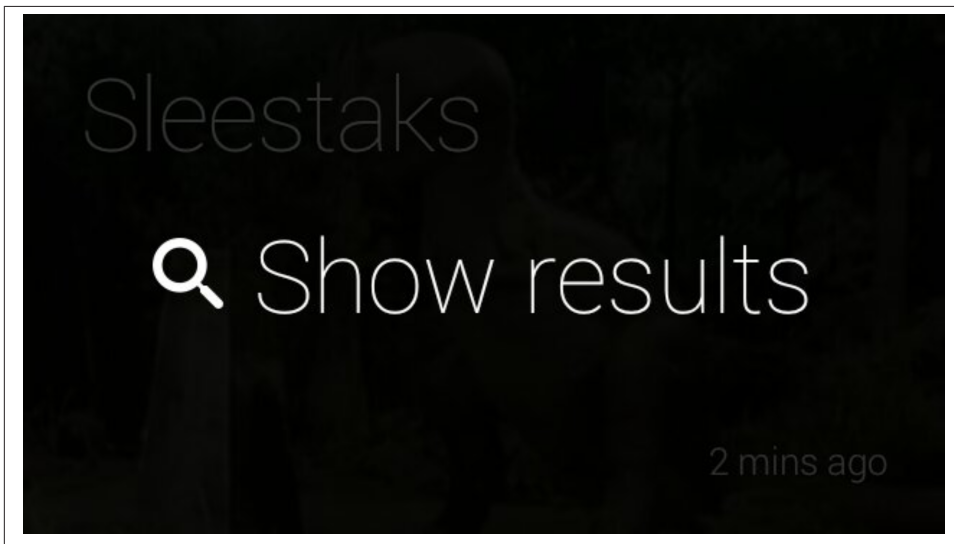


Figure 4-16. Your searches are preserved

But a possible UI challenge remains in how to dig into more search results than just those provided. Even the most obscure search won't have just a handful of matching URLs, and many people like getting into the weeds. There's currently no ability to see more results beyond those provided to you. Play with this idea and see what you could come up with in your own projects.

But before you set off on this little thought experiment, keep in mind that the number of results that are returned is capped for a reason. Can you guess when it is? It's our favorite word beginning with an "m"—*microinteractions*! Other mobile platforms still handle the job of showing tens of pages of search results just fine, so there's no shame in deferring more work to a device that was meant to do it. It's a well-executed and deliberate UX trade-off between letting users access information quickly, while not spending excessive time on Glass searching and thus taking them away from what they're doing in the real world. Use your best judgment and see what tests best with your users.

We're coming back to this again—the integral part of the Glass experience is getting technology to be high-impact and low-intrusion. Let the users quickly search, find results, review them, then continue to do what they were doing without ever taking them

out of the moment. Google Search on Glass is a very lean frontend, which highly encourages multitasking.

These types of features typify the program design ethos we're bringing to light by not preventing the users from taking part in their world, and always letting them return to it at a moment's notice. This is active engagement.

Whereas critics of Glass have said its input mechanics are limited, being only virtual menu items, voice commands, and gestures, we prefer to see this as a great opportunity with tremendous flexibility. And you should, too. We're far beyond just having a d-pad, a keyboard, and a couple of buttons at our disposal for cutting-edge program control. Again, having clairvoyance about the user's activities in the real world is a big advantage to coming up with great wearable usability. It's not easy, and not the way we're used to planning software projects—but it will pay off.

The Greater Goal of Wearable Design

Let's detour for a moment and talk about wearable design not just for Glass, but for all wearable devices. One app we have great respect for due to its design approach is Tinder, the social dating app. It was one of the first applications to create an extension for Android Wear, bucking the convention of many other related services that tend to use a complex heuristic based on wonky math to determine compatibility between two people. That type of functionality often leads to complicated UIs.

Tinder is much more (and admittedly) superficial—letting the user subjectively determine if another person is attractive. That's basically emulating real life. This drives Tinder's fluidly simplistic design for its mobile app, and lends to its streamlined user experience on wearable computing devices. Its mechanic of swiping left on an image of a potential match to ignore them or swiping right to connect with them is becoming the idiomatic standard for many other apps throughout industry. On Wear devices, this swiping action is captured perfectly by the smartwatch app, and makes for a perfect lesson in what a microinteraction should be.

It's so stupidly simple, based on a users-can't-get-it-wrong UI that relies on contextual computing (other users within a certain proximity), and social endorsement. We hope to eventually see this on Glass, too.

It's crude and crass, but it really works.

Glass for Gaming

One topic that's gaining a lot of momentum is how Glass will be used for gaming. Gaming has always been an application of technology that drives many other uses of software and hardware, and this isn't a lesson that's lost on the Glass ecosystem. While

the very respected craft of good video game design tends to lean toward high-end proprietary systems to accommodate the rich UIs for graphics and sound and create addictive gameplay, there's a lot we can learn and implement within the wearable space. The gaming market is expected to be enormous with Glass, both for playing directly on the HMD and in using it to display auxiliary information while a user is playing a game on a completely separate platform, ostensibly all connected through a common socket.

The **Mini Games** Glassware package developed by Google proves that casual gaming has a place on Glass; it is a series of titles that only take less than a minute to play each, yet can still be challenging, captive, and won't dramatically drain the battery. They all use user movement, sound, and tapping as controls. And each is complete with tutorials for beginners, scoreboards, and varying levels of difficulty. They're complete ideas—proper video games, not disappointing diluted afterthoughts or incomplete translations.

Several other early adopters put together some very interesting game concepts, concentrating on using the gesture capabilities of Glass as controller mechanisms to play sidescrollers like **GlassCopter**, or the parallax clay pigeon shooting fun of **Glass Hunt**—both of which are GDK apps. **GlassFrogger**, which won first place at the Breaking Glass hackathon in San Francisco in 2013, is a Mirror API Glassware service that was built in Dart in just two days and uses players' motion for program control—having players literally jump in place to move the little frog across the busy highway in a throwback to the classic 8-bit title.

(We've yet to see anyone apply the old *Up-Up-Down-Down-Left-Right-Left-Right-B-A-Start* Konami code trick yet. Hopefully by the time this book goes to print, that'll be a thing without putting Glass users in traction from whipping their heads in all directions.)

Speaking of reinventing the classics, imagine applying Think for Glass design to timeless titles. Think for example of a version of the classic game Battleship (which was, coincidentally, an idea BrickSimple implemented with its **GlassBattle** concept), using the turn-based model for wearable gameplay. This might be a little easier to pull off across platforms like Glass, PCs, portable gaming devices, console, the Web, and even other wearables, as opposed to something like a clone of Breakout or an MMORPG or real-time strategy title whose gameplay requires input from all involved without latency of any sort (though those other genres still may be possible).

Or consider a low attention span version of Monopoly using your actual neighborhood as the gameboard, with houses you define in geofences as properties. Players nearby could “land on” various locations in your real-life neighborhood and “rent” space. Or, how about the timeless Operation where you perform virtual medical procedures on a friend also wearing Glass? What Ice Breaker proved, which led to it winning Google's first Glass hackathon in San Francisco in 2012, is that the fun factor is achieved by blending data with real-world interactions, so keep that in mind.

Further, how might the popular augmented reality game Ingress be enhanced? Imagine the interesting applications that may arise by playing a real-world version of hide-and-seek, with players subscribed to the same game instance moving around in space, with Glass letting them track each other, while still not demanding long sessions, typically a minute or two.

Someone Create a Virtual Pet for Glass... Please???

Here's an idea to mentally chew on about using Glass for a great gaming client: a Tamagotchi clone. You know, a virtual pet that would remind the user to engage in a series of recurring routines to keep your digital companion from going hungry, being bored, staying awake past its bedtime, getting enough exercise, and so on. The design considerations and gameplay are fairly simple—recurring time-sensitive user interaction with sparse user input handling and only doing so within a certain daily timeframe (sounds an awful lot like microinteractions, doesn't it?), all for the reward of earning the love and loyalty of a happy pet.

A player would only need to interact with the program for a few seconds each time, a few times per day, to stay in the game. The challenge of building the backend to maintain update alerts for potentially tens of thousands of players caring for pets on staggered schedules makes this something worth thinking about for both front- and backend architectures.

So we're invoking executive authority as your authors to see if someone out there can build this. It'd be SWEET.

Glassware and apps driven by Android Wear may breathe new life into legacy social games, too, by giving them an added dimension as another outlet to receive and respond to in-game notifications—an extended interactive stage. Imagine applying this to a title like **Happy Aquarium** by Crowdstar, a Flash title that incorporates a player's Facebook friends. It routinely generates bonus gifts, such as the elusive and valuable baby turtle, which a beneficiary must accept on their Facebook wall within a few minutes, or the chance to add it to their fish tank expires. Facebook notifications fire to inform users about such gifts.

Happy Aquarium could push a card to Glass and/or a smartwatch, which would have only required a simple interaction to confirm the action—tap, gesture, or voice command—and accept the gift. The player wouldn't necessarily have to be at their desktop browser and in the game environment to stay an active part of it.

By introducing microinteractions, a social game's overall play can be enhanced by keeping users engaged with an ongoing game, even if they're not actively engaged with it at the moment. Simple and effective!

Games existing solely on and for Glass have already demonstrated their worth, concentrating on being the best they can be within the scope of the platform. On the opposite end of the spectrum, our community's history has also witnessed several examples of designs that don't work—both in terms of the visual appeal they project and the way they apply gameplay. Glassware that forces the user to pay attention for extended periods of time (typically more than 60 seconds), keeping the display active, and being excessively noisy with notifications are largely examples of what you don't want to do.

Design for the Cloud

Another high-level mindset you need to have when thinking of the architecture for your application is: *design for the cloud, not a particular platform*. You should adopt a practice of creating your own ecosystem around your idea and not be limited to a single client. Right from the start, you're building a distributed multitier application. This approach may have some familiarity to it, as many of the web services you might use are probably based on this structure. And from a customer's perspective, your users might expect to be able to access your slick Glassware on a browser and on their phone, too, and not just to customize settings.

We've now got a touch-aware computing solution in Android, Chrome for the Web, and Android Wear and Google Glass for wearables. To support those tiers and integrate across them you need to craft RESTful services with decoupled components, such as with [Google Cloud Endpoints](#). You can then add frontends to your logic, each with their own idiosyncratic UIs and capabilities.

Check out the [Google Cloud Platform](#) to see what scalable infrastructure and resources you can use to build, host, and run your application. Again, the Mirror API supports any cloud infrastructure, including existing ones.

Are You Starting to Think for Glass?

Let's conclude this chapter by coming full circle, back where we started with the simple concept of why Glass exists. And it bears repeating to emphasize the advantage to knowing how to Think for Glass, now that you've been enlightened with the knowledge of exactly what that entails.

The key is implementing these ideas within the constraints of the form factor using the principles we detailed at the onset of this chapter. Prioritize what the wearers are doing at the time they use the Glassware. Design not with system capabilities or the physical layout and arrangement of controls, but the user's behavior in the real world and how the software fits into that event. It's not simply porting existing frontend code to a new client—it's retrofitting entire ideas and applying new design approaches, while maximizing the experience.

So again, here's the concept about what not just Glass, but its ecosystem, truly is: *by bringing technology closer, Google Glass keeps technology out of the way.*

Makes a little more sense now, doesn't it?

The Five Noble Truths of Great Glassware Design

What does it take to build effective Glassware that users will thoroughly enjoy and use often? Great frameworks. Google started out by providing developers with the cloud-based Mirror API and an accompanying set of **core design principles**, initially with four guidelines and then later expanded to add a fifth member that came with the rollout of the Glass Development Kit (GDK). Imagine the Beatles or Metallica adding a keyboardist.

(OK...that's a terrible analogy, but still.)

By understanding exactly what Glass is as a revolutionary technology and what it brings to the table, these concepts dictate the pace for assembling services that are in line with the overall theme for Glass: *keep technology out of the way when it's not needed*. The guidelines, as described on the [Google Developers site](#), introduce a dramatic shift in the types of experiences software creators have been accustomed to building. The upside is that each rule is interwoven with the others, so there's perfect synergy. Once you conform to building your service according to one, the others naturally fall in line. That's not to say that applying one relieves you from needing to use the others, but their interdependence makes going from one to the next much easier.

We present them here as *the Five Noble Truths*, your means to understanding program design in the Glass world. They are collectively and individually essential in order to achieve the ultimate experience—a nirvana, if you will—for usability, convenience, aesthetics, flow, efficiency, and timeliness: the Glass UX. The Think for Glass philosophy is an extension of and complement to these guidelines. And as an abstract framework to guide you in building great Glassware, the lead that can't afford to be buried is that *the Five Noble Truths aren't bound to any particular programming language, software engineering methodology, or coding idiom*. You're free to use the tools and techniques

you're most comfortable with on top of the ideals of great design for wearable computing applications.

As a Glassware architect you're dealing with a new dimension of human-computer interaction, designing your application primarily around the user's active participation in the real world. This has a direct impact on usability. We've not been challenged with this concern before, historically needing only to worry about aesthetics, the arrangement of user interface components, networking, and performance impacts. And you want to do this because this in turn will create designs that encourage more use out of the convenience and fun factor they provide. Stay true to the Glass UX; this involves using larger fonts, understanding constraints within the display, and encouraging microinteractions—smaller, lightweight activities requiring bursty attention spans and minimal user input.

Just like the experience of using Glass itself the Five Noble Truths are short, simple, and to the point. And knowing them and how to apply them in your design of Glassware is essential in being a successful Glass developer.

Noble Truth 1: Design for Glass

Keeping in mind that Glass is an entirely new computing experience is critical to how you design Glassware. You can't just port old code or massage legacy libraries, or point resources to some server and simply hope they work. Likewise, you can't merely force-feed existing fragments of raw text and HTML or RSS feeds at a subscriber, or just spend time jamming on CSS. You need to apply all of these techniques to some degree, but you need to do them a little differently than you're probably used to.

The output may be pure hypertext (in the case of Glassware written with the Mirror API), but this isn't the mobile web, and it's certainly not WAP. And it's not merely re-scaling an iOS app that works on iPod Touches and iPhones up to dimensions so it looks good on an iPad. Perhaps the most poignant insight on this theory was made from Gene Reddick, who founded KitchMe.com, which has Glassware offering recipe searches and a catalog of meal preparation lessons. He said, "Simplicity isn't easy to get right."

You need to approach this as a completely new way of working with information, because that's precisely what it is.

Targeting Microinteractions

So far we have, and will continue to, emphasize the importance of microinteractions as the core unit of usability for the Glass experience. Cards on the timeline are an expression of this, but it goes much deeper. The thing to remember is that *well-designed Glassware doesn't apply only to situations where low intrusion is the norm*. Apps and services that are effective with wearable users actually shift their use cases toward microinteractions.

You can see this with the layout of the cards available from most Glassware—graphics on the left let you identify who or what the message is about or with, bolder headers provide a subject summary, and footer captions can quickly identify a full-bleed background image; all of these help users quickly determine if they wish to have the full text of the event read aloud to them, or if this is an item that should be dealt with later or via another device.

The **Android Wear design principles**, which aren't too different from the Noble Truths for Glass, refer to this phenomenon as “the Glanceable UI,” noting how the metaphor of cards on a timeline is meant to be quick. The message is short and sweet: get in, get out, and get on with your life. We sometimes like to point out that Glass is uncomfortable to look at for more than a couple of seconds, so if the user is going to, it had better be for a good reason.

Remember, *if you manage an existing application don't try to do a seamless feature-by-feature translation of it, simply because it's probably not going to work*. Glass is admittedly feature-reduced—and that's actually a good thing! In giving your platform a new home on the wearable computing landscape, keep commands simple. More menus mean more timeline cards to page through, which means more swipes for the wearer, which means less time spent in the real world, which means a violation of the Five Noble Truths, which means a lousy user experience, which means unsubscribers. Try to prioritize only the most pertinent functionality and build interactivity around that.

You wouldn't want to watch entire newscasts, but streaming just the highlights would be nice. **OVGuide Videos for Google Glass** does this and does it very well for entertainment news. Conferences, speeches, and debates can often be long (and boring) events, so how about getting a snapshot of just the most meaningful soundbytes? Hulu already features a massive library of shortform videos like trailers and clips, so this might be a possibility—another way a service can creatively adjust its core competency to embrace the platform and drive activity toward its own much broader ecosystem.

A major part of this Noble Truth for Mirror API Glassware is structuring your cards intelligently—their data, the menu items they use (and the careful ordering of them to make sure the most useful or most frequently used menu items are listed first to ensure single-swipe access), and any sharing capability they might contain. Some don't even use a UI at all. YouTube's Glassware is a one-way write-only uploader utility, and as such has no visual presence whatsoever—it's just a sharing service.

But it does what it does very well, because it works within the UI/UX parameters of the platform, and achieves minimal user input in a short amount of time (a microinteraction).

Tactical Wearable Design

As far as the cosmetic appeal of your services is concerned, most UIs will be a candidate for an “addition by subtraction” overhaul. *Thinking for Glass emphasizes minimalism in design and data brevity, but overall high impact.* Depending on the goal you’re trying to achieve, you may find this a refreshing and fun change...or a daunting challenge. As we touched upon earlier, working to achieve a simple result is often a huge amount of work.

Because of spacing gutters and the box model, plan on having a 427 x 240 viewport available as usable space for content—about 67% of the prism display. **Google strongly encourages** as a best practice using its **official templates** to design your timeline cards. They rely on the Roboto typeface, bright text on a dark background, and use spacing to give content some breathing room within the prism display.

There are also several free tools you can use to rapidly design specifically for the Glass experience, including the **Mirror API Playground** and the **Glassware Flow Designer**. Additionally, integrated development environments (IDEs) like Eclipse and Android Studio include pre-fab projects you can learn from and modify to build effective applications.

You’ll also need to drop the tendency to think of application design in terms of controls—no form elements, spinners, calendars or date/time pickers, menus, context menus, etc. Glass shouldn’t be viewed as a data entry medium. Information can be submitted by sharing resources like images, photos, and videos and other posts, but text input is extracted from text-to-speech, which means it isn’t going to be very much. Your UI design should emphasize sensory controls—sight, sound, and touch—with low-friction usability.

You want a wearer to have a minimum number of input steps to interact with your service. Work within the scope of timeline cards. Try to keep everything one level deep—that means at most one trackpad tap or one swipe, with only a few menu options. This isn’t the Web with infinite link depth or a mobile app that can have menus within menus within menus, and options upon options within those. Glass doesn’t use any modal dialogs. If you already have a complex UI on other platforms, you may need to rework it or leave some control mechanisms out. Prioritize the core of your platform. Having reduced features doesn’t mean a bad experience, because the overall result is still a nonintrusive way to use your product.

Just like the Noble Truths, the patterns on which to base your UX have solid foundations and are always expanding as the APIs mature. Check out the **official documentation for design patterns** to see the latest best practices for creating effective interfaces that are consistent with the rest of the community.

Don't Neglect Audio

One of the most popular features of Glass, and one that you get for free right out of the box, is the ability to have the text content of timeline cards read aloud to a wearer. It's a simple menu item that requires only a couple of lines of code—just hook it up to a property one time and you've got a great feedback medium that gives your Glassware rich usability. Audio on Glass is a valid, powerful, and flexible user interface, and is an important format for output. It greatly enhances immersive experiences like games, multiplies the usability of static items so the user can do other things, and is a standalone medium when reading content to the user without any accompanying visual elements.

Keep in mind that *microinteractions are measured in terms of the required attentiveness of the wearer, not just how long the wearer engaged with the system*. And with audio you've got a bit more leeway than with visual media, since microinteractions naturally don't mandate staring at them and since Glass will gracefully go to sleep if there's no interaction with the device even while the wearer is actively reading an item. The Glassware for Google Play Music streams songs and talkshow content without constant ocular interaction. Native cards can sound-off in the middle of other running operations, like when getting directions during turn-by-turn navigation. This enables users to have the freedom to multitask in the real world—they can walk, drive, shop, fold clothes, exercise, pay bills, or any number of other IRL activities.

However, there are some requirements when designing how to use audio. With the Mirror API, you can't set automatic reading of content with the built-in Read aloud functionality, so this is 100% an on-demand feature, like any other static menu item. Also, while Glass will continue to announce card content even after it's returned to an idle state, the user has to stay put on the card that contains it. If Glass wakes up mid-reading, the timeline's position will still be on that card, but swiping away from this card in either a horizontal direction or downward cancels the operation. There's also no concept of a playhead like there is with the Glass video player, so any interrupted reading will need to be restarted from the beginning. Upon completion, the timeline returns to the home screen.

You also want to be careful not to neglect poor uses of audio. The Mirror API by default audibly recites whatever value has been set in the `Timeline.speakableText` property (we get into the Timeline object model in [Chapter 9](#)). If that property isn't set, the API reads the value of the `Timeline.text` property, which is the visual content populating a card, never null, and normally short bits of content like a headline. So don't get lazy and fail to set some speakable text, because that's bad design. With a wearer having to glance at card before selecting the Read aloud menu item, Glass would read a user the blurb he literally just looked at. And that's, well, stupid.

These factors should provide an idea of how you should design audible output into your Glassware. Like everything else involved in Thinking for Glass, you want to implement it sparingly and tastefully. And that means anticipating the user's situation first and

foremost and catering to that, within the scope of a microinteraction. Figure out the comfortable range of how much audible content is acceptable, and how often it should be read back.



Your Content Out Loud—Shakespeare It's Not

The text dictation service in Glass is strikingly accurate and fast... but don't expect James Earl Jones-caliber enunciation, inflection, and emphasis for your content when it's read back to a user. Punctuation for dramatic effect like commas, semicolons, and hyphens are largely ignored, so a pause won't be translated very well.

That said, there is opportunity to improve upon this! **Umano built Glassware** featuring professional voice actors who narrate news items with a little more pizzazz than the staccato female tone Glass uses by default.

(But if you happen to get Morgan Freeman to do voicework for your Glassware, do let us know.)

Delete Versus Dismiss

One other thing that Glass users tend to be especially fidgety about is keeping vtheir timelines clean. Users often want a way to remove items to preserve disk space, and this is possible for many (but not all) items on your timeline with the Delete menu item. Traditional timelines and streams like those seen on social systems theoretically never end—but with the Glass timeline, you should almost never need to do anything to it administratively. Glass manages it for you, deleting older items routinely and automatically. *Google's server manages the timeline by keeping only the last 7 days worth of cards resident, or the most recent 200 cards, and also purges any items older than 30 days.* People can almost always safely ignore older posts.

Lastly, understand the difference between *deleting* and *dismissing* within your Glassware. The purpose of each is very clearly defined by Google, and your Glassware and the naming of your menu items and voice commands need to explicitly reflect how actions will affect their data or use of the program. *Delete* deletes the card and you should delete any related item on your end. *Dismiss* should just remove the card (and it isn't really necessary since the timeline will automatically purge items, as noted). See [Figure 5-1](#).

Google also has rules pertaining to deletion of data on its [Developer Policies page](#), so review those.

Will choosing a certain command cancel an action or eliminate a resource completely? Are there recovery methods? While Glass does provide the ability to let a user swipe downward to cancel an in-process action like saving or sending, which is analogous to pressing the “back” button on an Android handheld device, the system doesn't enforce

modal dialogs. Good Glassware design strongly discourages using modal dialogs, as doing so is very un-Glass-like.

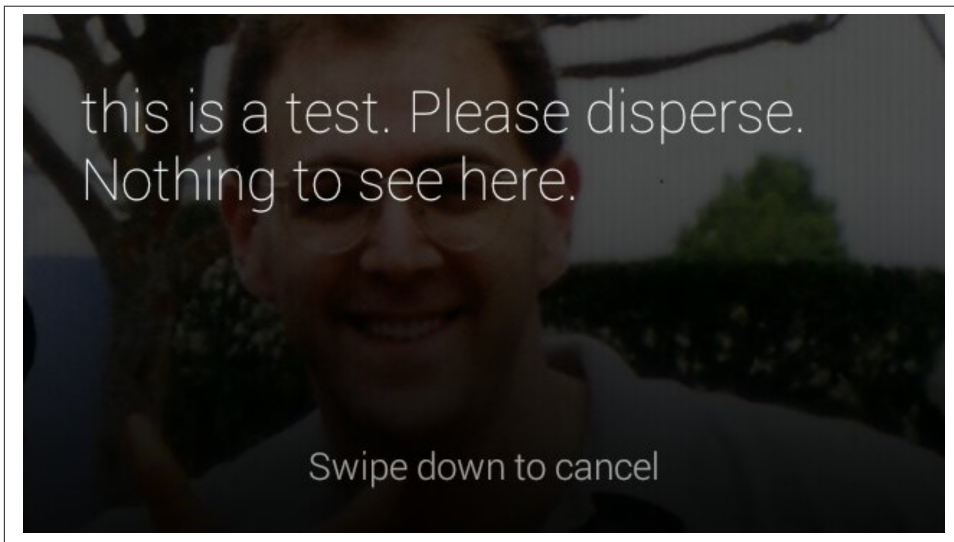


Figure 5-1. Cancelling actions in cards

We're all just one tap or swipe away from losing what could be a valuable piece of data, so be up front and clear about what impacts the choices and input controls your Glassware makes available will have.

Provide Web-Based Configuration

You would also be wise to keep configuration tasks like choosing certain categories of content or picking what time to not receive notifications from a service away from Glass. Sending users to more capable web UIs in a browser interface allows form elements to be used to their fullest, including some of the newer HTML5 controls like datepickers, without trying to implement workarounds that would be tedious to use on Glass.

Persistent values that span multiple sessions such as user configuration settings, and application preferences should be stored on the server in a database, not directly within timeline cards, even for native apps built with the GDK, which we detail in the *Develop* section of this book. The Glassware developed by CNN, Elle, and LynxFit each feature browser-based configuration so that complex forms, web menus, and categorical choices aren't forced into the prism. And administration of your service should be relegated to a web-based interface and accessible via desktop or mobile browsers, and/or an accompanying mobile app.

Herein lies one of the main advantages of Glassware—Mirror API services run on HTTP servers, so they talk to other HTTP services easily. Just write a web form to configure user settings for your Glassware and wire it up to your Glassware backend to associate it with your user database. This makes using your stuff inherently multiplatform, which is a major victory for building your brand.

Noble Truth 2: Don't Get in the Way

The second Noble Truth of Glass service creation is essentially a “No Loitering!” policy for third-party programs, upholding the nonintrusive nature of how Glass was built from the ground up. Put it this way: the system software doesn't barge in on its wearer, so neither should your Glassware. This isn't to say that you shouldn't make your presence known; just don't linger around when the user's done using your stuff, and don't make the act of using it time-consuming. In wearable computing the top priority is the wearer, so the objective is to let her get on with her life.

Notify Responsibly

Another big area of designing your service is respecting the product's ambition to stay the heck out of the user's way. You may have a really popular social application across multiple platforms, or while reading this book you may have been dropkicked by your muse or guardian angel and came up with an amazing idea—but recognize the fact that even though you may have a ton of data the user may want, Glass is still meant to be a low-intrusion medium.

Despite having one of most voluminous archives of content in the world, the *New York Times* Glassware only generates headlines once per hour, and deviates from this workflow only when pushing breaking news items when they occur. The *Times* sticks to its intelligent distribution schedule for the day's major events through batch processing, which is predictable and expected and thus does not interfere with the users' daily activities as they go about their day. This is also efficient in terms of the backend; the timeline card bundles that the *Times* delivers contain one or more stories, but generating them involves just a single sync operation for Glass, and one notification.

A wise design decision is to avoid bombarding your subscribers with constant notifications, and knowing how to reshape your data into a manageable load that respects the ongoing activity of the Glass user is a powerful skill to hone. A big culprit for this will be social software applications, which are challenged to find a way to condense their users' data, consisting of their graphs and accompanying messages, into a concise format. Twitter has done great work to this end with its Glassware, not forcing people to ingest the entirety of their timeline, as we detailed in [Chapter 4](#).

You may very well have a veritable firehose of updates to offer to an audience, but knowing what pulses are acceptable across desktop, web, mobile, and Glass applications,

and knowing how to properly spread them around with each of those platforms, is the sign of a champ.

Even in cases where batch processing might mean receiving a whole slew of updates at once, it's probably not necessary to send a notification for each one. **Winkfeed**, a Mirror API–built service that lets users subscribe to multiple information channels, gets around this problem by issuing bundles of cards during times when two or more sources are pushing alerts at a user, rather than pushing multiple timeline cards independently (**Figure 5-2**). It even goes a step further by laying out a helpful summary of the bundle's sources within the cover card. What could be a very noisy problem at scale is handled beautifully, respecting the user's bandwidth...and sanity. Your customers will appreciate it.



Bundle or Die

One of the most consistent patterns in use by the best Glassware is that services publishing multiple timeline cards do so by organizing them into bundles, where appropriate. So keep this in mind if you want your stuff to make it into the official channel and get top exposure.

We all share the same space, so let's not drown the user with new timeline cards every eight seconds.

Remember that while other developers may in practice be your competitors, as part of this community they're designing for Glass, too. Your goods actually become more useful when not in the user's face all the time. People will appreciate the breathing room.

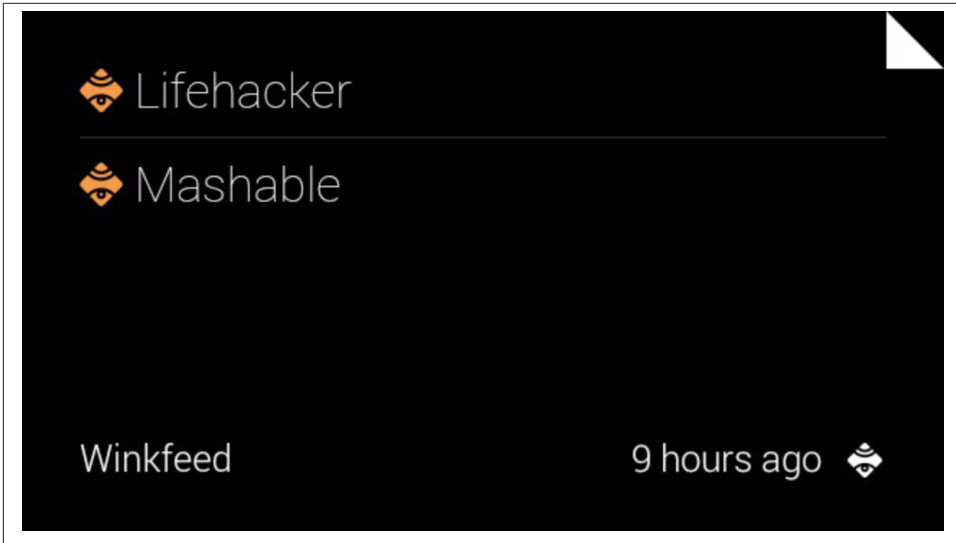


Figure 5-2. Winkfeed uses bundles

Less Is More

An example of not getting in the wearer's way while still providing great usability is Strava's fitness apps—Strava Run and Strava Cycling. They deliver great usability by giving measurements during your workout, comfortably placed in a card that pins itself to the left of the home screen that shows running stats while you work out, and slips quietly into the background when Glass goes to sleep rather than stay present in your periphery with the display on, possibly distracting you and draining the battery. The apps are always accessible at the moment's notice, completely hands-free, by waking up Glass with a head nod. There you'll get real-time measurements and progress reports read to you audibly every half-mile about your pace and progress, even if the device's display isn't awake—the audio is delivered as you hit your milestones. The same feature is available with turn-by-turn navigation when getting directions to a destination.

This is a technique that's easy to do regardless of whether you're working with the Mirror API or the GDK, and proves how a major part of a high-impact user Glass interface sometimes means having no visual element at all. *The user's interaction in the real world takes precedence over the user's interaction with the system.* This could even include in some cases that your target viewer may not get your notification, may never see your item on the timeline, or may just ignore you completely—and you have to deal with that. Good Glassware design means having some contingency.

Strava's Glassware also uploads your progress to your profile as you're working out, so your friends can see and compete against you. But you also get the ability to control

certain aspects, such as marking splits with menu items. It's an effective mix of automated and manual input that doesn't take you out of the moment.

The Exceptions Make the Rule

The sole exceptions are immersions for GDK apps, which completely take over the Glass UI and keep the user constantly captive and engaged. While we've stated the overall goal to facilitate microinteractions, Glass is capable of handling larger, more demanding UIs, too, where appropriate—and what's more, *it isn't appropriate* for most things. Some situations in life are more critical than others and might require constant attention for longer periods of time, like surgeries, air traffic control towers, baby monitoring, or safety inspections.

Turn-by-turn navigation scenarios are an example of an immersive application—having a map dynamically display and update your position in relation to a destination, and making this resource available as a live card is significantly better usability than having to swipe to a specific point in the timeline to find a static map with location data updated only every few minutes.

Glass is designed to let people interact better with those things surrounding them, and Glassware needs to make sure it honors that. With regards to your service, this means getting rid of UI elements quickly and hiding long-running operations, so as not to unnecessarily draw the user's attention. The Timer Glassware proceeds with user-defined countdowns quietly and in the background, but a live card app is pinned to the left of the home screen for the duration of its use and becomes immediately available and the first thing the user sees when Glass is woken up. The app is still a microinteraction-friendly conduit, not taking significant attention away from the user.

There are facilities to allow processes like backups and silent updates to run in the background, so leverage those if you need them.

Exhibiting altruism toward your fellow developers is also important, as well. Yours more than likely isn't the only Glassware users are running on their headset, so keep notifications light and grouped logically. Rather than seeing this in terms of modern mobile applications where installed applications wage a perpetual battle for their owner's eyeballs and time, Glass promotes respectful coexistence.

Noble Truth 3: Keep It Relevant

Interacting with Glass is very much helping you stay in the now. Doing things that don't contribute to that will lead to negative experiences with your service. Again, you're not trying to take the users away from the real world, you're keeping them in it by having technology be a transient presence. And with Glass being a contextual platform, providing information and interactivity capabilities in real time for variables like location, time, and any associated events from other applications is a big opportunity.

To illustrate the point, say you maintain Glassware that tracks the variance of stock prices throughout a day's trading. Rather than provide a constant stream of new timeline cards each time a security's price changed, you would want to initiate a fresh pinned card to start off, and then update that same card when certain criteria are matched (e.g., percentage or price fluctuations above/below a set threshold), as opposed to generating a series of new cards each and every time an event occurs and flooding the user's timeline.



It's worth noting that this Noble Truth was modified from its original dictum of "Keep it timely." Emphasizing relevance, of which time is inclusive, speaks more to making information truly contextual based on multiple signals, as well as both time-sensitive and location-aware, which dramatically increases its value. This is the greater good and gives a better sense of what the user is doing at that precise moment.

Consider, for example, the desire that everyone has to get on the "OK Glass" voice command main menu, or to have their Glassware's cards pinned. This is an app-centric focus—we want our applications prominent for the user. But it forces the users into our world instead of trying to understand the world that they want to be in. They don't want to sort through dozens of pinned cards to find the one they want. *They want us to deliver information to them when, and only when, they want it.* They want to take action, not start an app. They want to be able to easily ask for new information. They want to quickly find the data they know is there...somewhere. They don't want clutter.

Context FTW

It's also vitally important to note that when Thinking for Glass, relevance applies not only to *time*, but also to *space*. This goes back to the point we can't stress enough that Glass is a conduit for contextual data, which these days is multidimensional. Where the user is, what those conditions are (i.e., weather and traffic), and who is in his or her proximity are among the additional signals you can use to add richness to your Glassware and make it more meaningful. Take for instance **Field Trip**—after the Glassware has been enabled, helpful information pops up wherever you are right at the moment when it makes the most sense, without being invoked by you.

Additionally, weather monitoring applications are always going to be big, and **eyeFlame built Glassware** that tracks tropical storms. It has a neat twist: it lets you customize the geographic zones you wish to keep tabs on (including your own) to receive timely storm data.

We can also predict seeing a similar service that might push card bundles containing the local time, currency exchange rate, news headlines, and common phrases to ask when a wearer gets off the plane in a new country. It could even hail taxicab owners for the area. Or, how about a more complex app that would illuminate your prism display

with factoids about the movie or TV show you're currently watching? Further, let Glass talk to your DVR and send you recommendations based on who's starring in a particular show. Or, consider a home automation application that uses the Glass sensors to display suggested cards that would let you wirelessly adjust thermostat, air conditioning, and lamp settings based on the surrounding temperature and available sunlight. Or maybe Glassware suitable for a hotel would save road-weary guests from the arduous task of standing in line to check in, getting their room key, seeing about any messages, ordering a pay-per-view movie and room service, booking a seat for the dinner show later that night, and then queueing up the next morning's wakeup call.

Or you could even build an app having nothing to do with Glass but geared toward its users—a bar or club owner could define a geofence at their establishment that would alert friends within the same Google+ circle or Path clique of each other's presence there on their devices when each arrived, and when 10 or more of them arrived, they all got a free round on the house.

See what we mean? Context rules!

How Soon Is Now?

So the stage is set for Glass to relay the freshest content, the most relevant information, and the most up-to-the-second data available. Impressed with the responsiveness that Glass offers, noted tech critic Steve Gillmor [said of the Glass experience](#), “The whole world is not only watching but feeding the real time stream. Social meets mainstream.” Nowadays people online expect their applications to reflect the latest information. And they will with Glass. Any message a user gets she'll expect to have just happened.

But again, with respect to the first Noble Truth, this doesn't mean blasting inbound notifications at subscribers nonstop. A bit of restraint needs to be applied in shaping the contextual value of your message and exactly how quickly you need it to be delivered.

Don't rush into this, really give it some thought. How critical is your information? Is it time-sensitive? How in sync is it from the source in relation to the person using Glass? For example, Glassware from a government utility agency informing 90,000 subscribers that they've got a power outage to look forward to nine hours from now is a lot different than an early-warning weather system informing 90,000 subscribers that a tsunami alert was declared three minutes ago and to seek high ground immediately, which is a lot different than an auto dealer informing 90,000 subscribers that a brand-new Lamborghini Murciélago is on sale for only \$100 and ends in nine minutes. Get it? It's all about context.

Timeline cards generated with the Mirror API are not real-time entities—even though data on an existing card can be modified on-the-fly and will change as a user is looking at it. Just update a card that's already been inserted and if Glass is awake and that card

has focus, the user will see the new data without having to refresh. *Only the GDK is capable of producing items for the timeline that are truly zero latency*, based on sensor readings and programmatically using live cards as a frontend.

The Glass sync operation for Mirror services is driven by a special build of Google Cloud Messaging, and while the system does deliver messages incredibly quickly, message delivery isn't guaranteed as soon as an event fires (it may attempt a series of retries or queue the message temporarily). Also, Glassware using the Mirror API that uses search tends to take longer than Google's search feature on Glass (which is a native app). Search features in Glassware from KitchMe and Fancy are known to take as long as a minute before returning matching listings when queried for recipes and shopping, respectively.

So with information being delivered online as it happens—more or less—these are very exciting times, and this is a big part of the thrill of writing Glassware.

(Even if the numbers are fudged just a little for dramatic effect.)

Noble Truth 4: Avoid the Unexpected

Any respectable book on software development will stress the importance of defensive programming. And that's certainly true as you begin and progress through your adventure in Glass development...with a twist. Because of the intimacy Glass creates between itself and its owner, unexpected or out-of-context behavior with any aspect of the system disrupts this bond. As such, bothersome actions feel significantly more disturbing than with other distributed platforms. So maintaining this relationship should be of utmost concern to you.

Content

If you're writing a Glass service that's content-centric, there are a few gotchas to be on the lookout for. If your service spits out preformatted hypertext, like that generated by a content management system, you may need to scrub your HTML through a regular expression to filter out unsupported tags and attributes.

The complexity and length of your content is also a biggie. You might want to extend a multiplatform CRM app you manage, which is great, but keep the complicated stack charts out of Glass. If you're building something with social integration, temper how your user input and interface elements are applied within timeline cards to fit the microinteraction motif. Emoticons? Absolutely. Presence indicators? Sweet! Geolocation and check-ins? Knock yourself out. Image tagging? Maybe. Long status updates? Perhaps. Feature-length articles? Not so much. We stress again the need for terseness.

Even with features that let cards handle longer chunks of text like bundles and automatic pagination, your optimal goal should still be to shoot for a single screen. People anticipate the body of an electronic post to be the main course, and especially for a new

medium like Glass, probably won't assume multiple pages. The best microinteraction is the one that requires as little input as possible. If a user gets an alert from you and with a simple head nod is able to get all the content on a single screen, that's a perfect situation. Even with hands-free, aim to be completely hands-off.

Even if you've got nothing but text in a timeline card and use the full space available to you, there's not a whole lot of room to work with. Keep an eye on excessive information within them and spread extended messages out over multiple cards, or utilize ellipsizing or truncating material or having it bleed out of the display. This highlights the need to manage expectations—using the various visual cues that are available to indicate there is more text that isn't visible. Less is more, once again.

Performance

There are a number of things that programmers can do to cause Glass to quickly heat up. These mostly apply to those using the GDK, as direct programmatic access to the hardware, running long loops, or running computationally expensive background services tax the processor, which results in it generating excessive heat, which also quickly drains the battery. A failsafe feature in the Glass OS blocks any Glassware from running in cases where the device is overheating, letting the system naturally cool down for a couple of minutes and giving the system a much-needed break. We're still aiming at small usage, so even custom-built applications shouldn't be used for more than a few minutes to ensure the reliability of the system.

Generating cards in obnoxiously large loops can be dangerous overall and you don't want to tax the system. Even if you send cards in bundles, try not to do too many. Instagram could easily generate hundreds of cards with images for its high-end users that follow 10,000 profiles; and while that works for a smartphone or tablet, it's not a positive Glass experience. It requires lots of user attention, and lots of interaction to page through them, which forces the display to stay on, causes alerts to constantly go off, and drains the battery.

Judicious management of the camera and processor and not keeping the projector unit active for long periods of time is therefore a must, because you don't want the system overriding what your service and the user are trying to do, stalling what should be productive time with defensive procedures in what is essentially the Glass version of a fire drill. Simply stated: *when you're good to the system, the system's good to you.*

Don't Be a Bandwidth Hog

And also, dear reader, while animated GIFs are supported within timeline cards by specifying their URL in the "src" attribute in `` elements, you'll want to steer clear from using those haphazardly assembled files that take movie scenes and turn them into

45-second miniclips based on rasterized JPGs. The end result is file sizes that are insanely hard to work with, take longer to download, and make the system crawl.

(Mercifully, the Mirror API caps media uploads at 10MB.)

Further, this is bad content design and a card that appears like in [Figure 5-3](#) would certainly be considered unexpected behavior, due to the fact that a user isn't able to visually discern between a broken image and one that's loading in the background. The timeline doesn't use any sort of "loading" icon or status bar for large images like Google + has for animated GIFs within streams, or Vine or Instagram use for embedded videos—so a Glass user won't know if an image placeholder icon in a card is simply a mis-referenced image, one that's loading, or severed connectivity, as [Figure 5-3](#) demonstrates. If a file is tremendously large, nothing may load as the card's background anyway, and they'd not see anything, except for perhaps a confusing caption with no context.

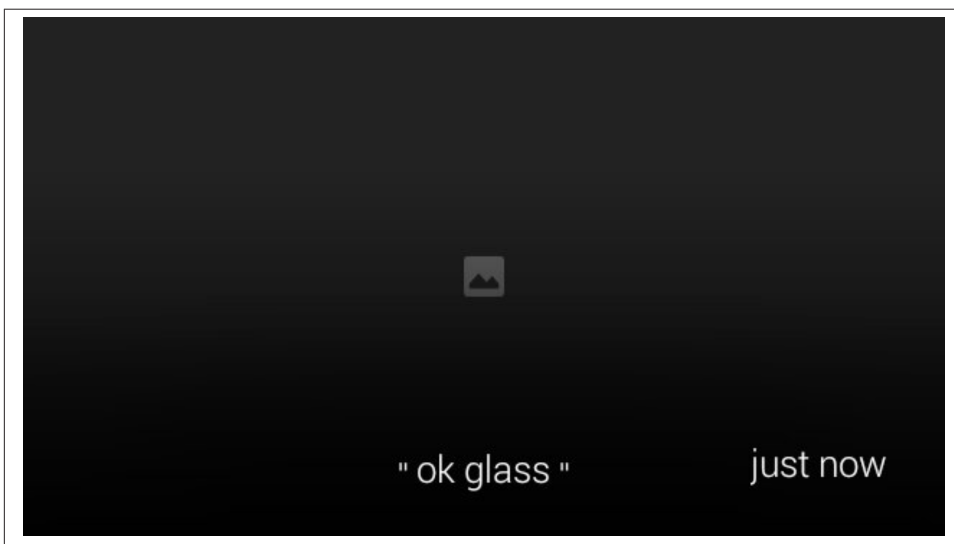


Figure 5-3. A nonloaded image

A broken image icon within a card when connectivity is spotty...the users can't tell what's going on with the image, just that it's not loading. We'd bet that most of the time they'd just swipe on through—they've got better things to do than wait for images to load. Avoid these situations at all costs.

The best practice in our community is to defer to the native Glass media player, which streams video playback from the source. If you really want to have moving pictures with your card, include a snapshot and link to a movie file, or reference a clip and let Glass handle it gracefully with its own dedicated player. Don't try to be sneaky and force an

animation to loop within a card. Thinking for Glass means being responsible, and a good neighbor.

And if you see someone using them, make a Netizen's arrest under the charges of *Not Cool Design*.

Permissions

Another area where not forcing unanticipated behavior upon the user is crucial is in service-level permissions. If you've developed some neat functionality you've put in a custom action, make sure the user knows about it. Mirror API services follow the same sandboxing model that Chrome extensions and server-side applications do in that they can't access and/or manipulate each other's data, but this doesn't mean you should just go merrily along without thoroughly informing the user about what your stuff's capable of. Don't be cheap and fly under the OAuth 2.0 radar and assume that it'll provide blanket coverage for you working with the user's data.

Fear not, we'll fully get into permissions and authorization in [Chapter 8](#), so you've got that to look forward to.

In Glass development, the operations that should be declared up front are working with a user's location, writing data to a cloud storage service like Google Drive or Dropbox, doing work when the user isn't using the system, incorporating her contact entities, or using her credentials for another system like a social network such as Google+. To do otherwise might seem nefarious to the end user, and there's no better way to earn unsavory ratings and reviews for your products than being shady.

This speaks to a big part of development on any platform, but it's something that has particular relevance to Glass, since context is the order of the day: user expectations. We mentioned wisely futureproofing earlier, but this doesn't mean including permissions and OAuth scopes you're not currently using. All Mirror API Glassware will need read/write access to the user's timeline; optionally, the Glassware may need to ascertain location data, and possibly even profile information about the wearer. [Figure 5-4](#) highlights Ice Breaker, which requires the user to confirm granting several such permissions. Be up front and state in simple terms what your Glassware does and what data it requires. Some people get skittish about nonmap apps using location data (recall the conspiracy theories from [Chapter 3](#)).

Honesty truly is the best policy here, so make sure to do the right thing.

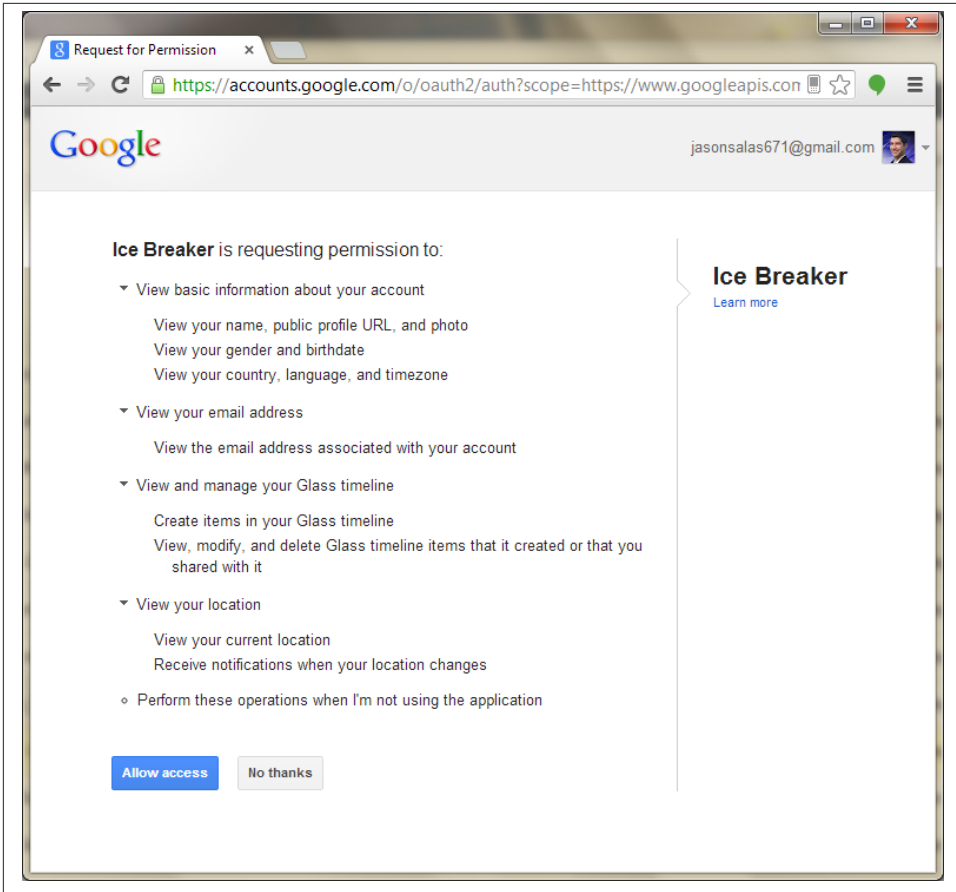


Figure 5-4. Permissions for Glassware

Error Handling

Simply stated, error handling is different when you Think for Glass. Robust an operating system as Android is, things will go wrong. As opposed to web programming where astute practices normally involve recording an error, handling it in a graceful manner, and displaying some sort of friendly message within the client app (or as friendly as it can be to say, “Oops, we messed up”), in the Glass world just sidestep anything visual altogether. You’ll want to avoid sending wonky error codes to the user and waste a sync cycle and do not want to uselessly generate or update a timeline card. Have some facility that logs an error code and informs you as the developer that something went awry (SMTP, SMS, IM, etc.) so that you can quickly take action to remedy the situation. Including timestamps on your timeline cards is helpful in this regard, as a sly reference to let the user (and you) know when the last successful sync occurred.

For native Glassware, you might consider implementing a messaging feature where crash reports could be logged and sent back to you, complete with stack trace information.

Figure 5-5 shows how Fancy’s Glassware handles issues with searches against its database, and Figure 5-6 is a default error card inserted into the timeline when GDK Glassware experiences problems installing on the device.

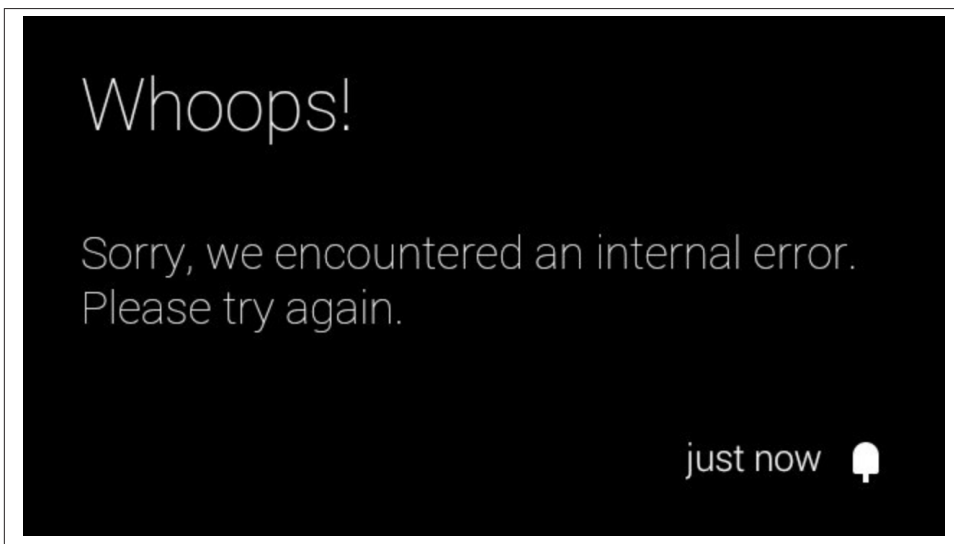


Figure 5-5. Error reporting

Because the service you’re writing is in the cloud, there’s no need to inform the user that there’s some new feature or the all-encompassing “various bug fixes” and then push an update. Assuming you don’t apply a change to your application code that requires one or more new permissions, you just update your service and let the changes silently roll out to subscribers naturally the next time they sync with the cloud or run the app. This again borrows from one of the prime advantages of web development and having your codebase sit on and be served from an HTTP server: you just hit “save” and deployment happens. You also don’t have to worry about client compatibility and fragmentation issues or whether your Glassware will work—it will.

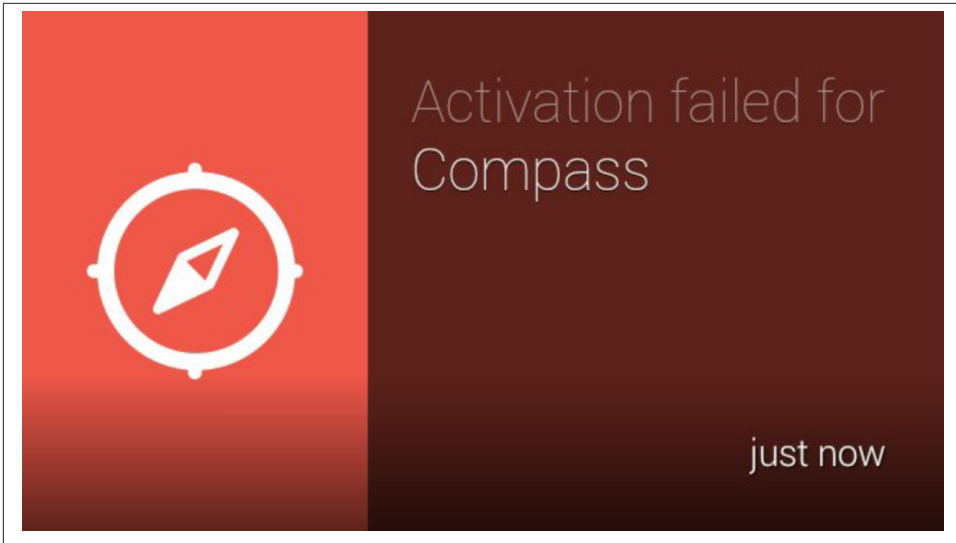


Figure 5-6. Installation glitches

But for those unfortunate occasions when you have to suffer through periods of system maintenance, unforeseen downtime, getting hacked, a younger sibling downloading malware that locks up your server, or your dog chewing on your CAT 5 cable (all of which we sincerely hope never happen to any of you), just don't send anything. Make system uptime status reports accessible on the Web and in a mobile app and tweet out updates, and make sure to note this in your Glass service documentation. A good application these days is multiplatform, so utilize the ecosystem you've built.

For a straight-from-the-horse's mouth approach to error handling, consider Google's best practices documentation about how to handle glitches when [using the Google Mirror API to upload media](#).

Synchronization Across Platforms

A final tip for strategic design relative to keeping things clean is that you take care to follow a pattern of replicating notifications across as many devices as the user is signed into with his Google account as they occur. But this is only half the battle. The bigger concern at your application's backend is that you should take care, especially when dealing with GDK Glassware that's able to tap into cron-like alarms, to not let notifications linger across other devices after they've been dismissed elsewhere. As we embrace more screens in our daily lives, it gets more tedious to have to chase notifications around those screens, or swipe off a notification on your tablet or Glass that you dismissed on your smartphone three hours earlier.

Android has done this incredibly well since its Jelly Bean version (v4.3), and you shouldn't neglect using the various synchronization tools that the Android SDK provides. Even for systems that rely on RESTful APIs, good multiplatform design dictates that notifications read on one device or OS shouldn't hang around on another as if they've never been viewed. It's causing more repetitive work for the user and goes against the microinteraction model.

An example of Glassware that does this really well is Hangouts. Imagine that you're sitting at your desk and chatting with friends on your laptop using the desktop Hangouts widget in Chrome, while your phone and tablet sit idle on your coffee table, and Glass sits asleep on your head. As you post instant messages to your recipient(s), the other devices tend not to go off with the "new chat message arrived" tone whenever a participant responds as long as the group chat widget has focus in the current browser tab. Should you switch to another tab or minimize Chrome completely and someone sends you a message, one of the other devices in some indeterminate order will sound its alert.

The Hangouts Glassware does a really good job of not buzzing the bone conduction transducer each and every time during a lengthy chat session if another connected device has the focus. It's very helpful and aids the overall experience by not inundating you with notifications across every single device and platform you are signed in to. While you should notify users about activity and cleanup alerts that the user has dismissed, Hangouts goes a step further to achieve this objective by suppressing excess noise.

In a similar fashion, Gmail's Glassware lets you mark messages as archived or starred, which sync online to mobile and desktop clients within seconds. If you build a system where users are able to set their mood in text or graphically, make sure to apply updates across any listening channels, whether Glass was the client that applied the change or just is a connected destination that pulls down the new data.

This type of feature is common to popular web APIs, and pulling it off isn't the same for everyone. It may be as simple as setting flags in your system as to the state of certain messages or as complex as concurrency and managing race conditions, so get creative here. The point is that you'd be wise to keep things tight if you exist on two or more platforms.

Work to bridge the digital divide!

Surprises Should Be Pleasant Surprises

Before moving on to the final Noble Truth, keep in mind that sometimes a little surprise can be a good thing. Not all unexpected behaviors result in negative experiences, and you should balance the amount of usability and functionality that your customers anticipate with innovation that they'd probably never ask for based on your domain expertise and creativity. Maybe gift them with a coupon from a partner based on how

many times they've interacted with your app, or apply some gamification and set up a leaderboard based on the number of times people share your stuff. Perhaps a card-based tooltip for power users might be just the thing they need. Or plant the occasional funny easter egg just to show them you've got a sense of humor.

But this, of course, should complement, not supplant, rock-solid engineering and intelligent design—and basic common sense. You don't need to have a master's degree in marketing to know that sending people time-sensitive discount vouchers for a big sale at the department store isn't practical during times when they're asleep or if they arrive during periods when they can't make use of them. Several Glassware programs let the users define their time zone for things like notifications. A time-sensitive buffer to ensure the wearer will actually see an alert for things like incentives is a great idea for campaigns like this.

Maybe you could even structure your Glassware so that over time it learns how each user interacts with it and adjusts the ordering of its menu items to be those that the wearer uses most frequently. This wouldn't hurt your image and would plant the seed in the user's mind that you're looking out for his best interests, that your development team is really good.

They certainly won't object to that.

Noble Truth 5: Build for People

The newest member of the bunch, the fifth principle reinforces the need to maintain the experience that Glass is a personal technology, by designing your Glassware in a way that adjusts to a user's life and lifestyle. And let's face it—the average human being is busy, scatterbrained, and lazy. So base the blueprint of your Glassware in a way that takes this into account. Exploit mobility, incorporate contextual input signals, leverage external configuration, emphasize gestures and voice as primary program control mechanisms, use lightweight data payloads, and lean toward limited attention demands.

As Glass gains in popularity and market share and the ecosystem expands, people are going to see the true benefits of integrating the device into their daily lives, so don't outline a service that demands their full attention for long periods of time and requires constant manipulation.

Voice is a powerful component both as a user control and also as output. Because Glass is fully intended to be used in all sorts of situations and scenarios, inside and outside, supplement text in cards with the Read more built-in menu item action. Transcribe content and let people enjoy it at their leisure. In this sense, you're offloading any need for a UI whatsoever!

This is a new way of presenting data and organizing an application's user interface that you're likely not used to, so don't save this one for last or do it on the cheap. Really take

some time to consider how your data will be delivered, how your application will be interacted with, how someone can easily share and receive resources while they're on the go, and how someone would want to use your Glassware all the time. The blissful simplicity of Glass makes for an interesting design challenge.

As mentioned in **Chapter 4**, this involves predicting what specific activities the wearer will be doing when using your Glassware, and using that as the base for proper design. It's the mother of all ambiguous statements, but *in Glass the wearer's environment is an active part of the UI*.

Zynga's achievements with its line of video games brought to light the fact that gaming didn't have to be an all-encompassing experience; players could interact with them while in line at the supermarket. Its titles not only leaned heavily on being *social*, but also on existing as *casual* activities. This is a great lesson that speaks to the heart of the Glass experience, not just within the gaming space but also with much broader implications. To flip the script, you might design produce-shopping Glassware you could comfortably use while waiting at the video game store.

Advocate Multitasking

You also want to put a premium on supporting multitasking as a core part of your Glassware's usability. This can involve three key areas, depending on your implementation and which development framework you're using:

Managing microrinteractions

You do not want to impose too much effort in either interacting with the system or consuming content.

Integrate the real world with Glassware

You want to make the users' actual experiences part of the Glassware for things like photo taking, time-sensitive news updates, and geo-accurate coupons and offers.

Using the timeline while your Glassware is running

This is a feature that's promoted for designing great live cards with the GDK, as it allows the user to have a running service while browsing other content that's arrived.

Glass Is Naturally Social

One other thing to mention is that the development model for Glassware is very conducive to writing social software. The mechanism for sharing a resource practically begs you to do it. Libby Chang, a Silicon Valley-based environmental engineer and a leader in both the Society of Glass Enthusiasts chapters for San Francisco and Asia, says a key factor in the Glassware design phase is "**designing for community**", noting the ease with which resources can be Liked, retweeted, +1'ed, approved, and exchanged on social graphs and platforms, and ultimately distributed between them.

In designing great Glassware, it's important to provide a solid feedback loop so Glass users know the actions they took on their device actually happened on your system. Two great examples that epitomize this effect for the social crowd are, again, Google+ and Twitter. When users want to endorse a post shared to their timeline, Google+ features a dedicated +1 menu item, and Twitter has its familiar Favorite and Retweet menu items. When a user taps "+1," an animated status bar fills from left to right and sounds an alert, confirming the response. Twitter's goes about this somewhat differently.

Choosing its Retweet or Favorite selections in a card immediately shows the "working" icon in the footer of the tweet, which then disappears after a few seconds. The card the user is endorsing is then copied and placed at the front of the user's timeline to the right of the home screen, and an alert tone is sounded. Both services use the Mirror API's provided tools with different implementations to achieve the same effect. The user knows her actions were posted, visually and audibly, and that her choices are now reflected on any other online platform. Their design tactics took different routes but delivered the same end result.

Evernote is also among several services with a neat feedback mechanism, allowing items to be shared from its web UI to Glass. Then when wearers use the Read aloud built-in action in their timeline items, the Glassware prefaces the content with "Evernote..." just to remind the user what service the item came from. It's very simple and extremely helpful. This is a pattern encouraged for items to identify their source.

These examples are use-case specific, but demonstrate the flexibility of the platform while providing a new way to confirm actions. These same ideas can and should be applied for any type of Glassware.

The Power of Community

An important part of the Glass ecosystem is also the communities it's spawned, like the [Society of Glass Enthusiasts \(SoGE\)](#), user groups that are completely independent and community-run. There are more than 25 chapters, spanning five continents throughout the United States, Canada, South America, Europe, Asia, and Australia. Some are regional, some are statewide, some are within a city. There's even a SoGE chapter dedicated to Glass filmmakers! And [UbiTech NYC](#) is a thriving and eclectic group of wearable computing designers, engineers, and entrepreneurs in the northeast US.

Most user groups have frequent meetups, hackathons, and photowalks, and all focus on networking, listing events for the Glass community, exchanging ideas, sharing tips, and experiencing their surroundings through Glass.

If you can't find a chapter near you, start your own! The SoGE logo templates are open source and freely available from [Noble Ackerson's GitHub repository](#) so you can come up with your own design and charter your own local membership.

You can also get in touch with the numerous Google Developer Groups (Google the topic to find the closest one to you), or communities like [UbiTech NYC](#), run by Katy Kasmai.

So...That's It? Really?

From soup to nuts, Glass is a totally new way of working with data, users, and their environments. And as far as designing effective software solutions, that's what you need to concentrate on. *The Noble Truths ring true regardless of which developmental framework for Glass you're using—but depending on whether you're using the Google Mirror API or the Glass Development Kit, you do have some challenges and opportunities.*

Because of the general streamlined model that follows from everything we've talked about in this chapter, keeping all five of the Noble Truths always in mind as you create the next great Glassware will make sure it honors the Glass experience and makes your users happy. Keeping these truths in mind is important, but not always so simple. Glass is so new that it is easy to fall into old habits that don't work so well in our new world.

We'll be looking at those traps next.

Glassware Antipatterns: Avoiding Poor Design

As we discovered in [Chapter 4](#), an unbiased overview of Glass reveals that while it's a wonderful and powerful communications platform that's rapidly expanding with Google and the developer community creating a ton of real-world uses, it quite frankly isn't the solution to many existing problems—and wasn't intended to be. A further issue is the ways that Glass can create new problems and be devalued by misapplying it in different scenarios.

As a Glassware architect (either on the design or development side) and even as an end user, you need to know what Glass was meant to do and do well, and not try to cram the Think for Glass philosophy into every situation under the sun. And programmers and designers that spend their days and nights crafting innovative Glassware need to be aware of and respect the implied boundaries and design recommendations, allowing users to get the most use out of the platform in terms of maximizing battery life, minimizing notification frenzies, and keeping the input required from hitting excessive levels. It's a virtuous cycle: responsible program design emphasizes ease of use and efficient use of system resources, which breeds proper use, resulting in happy users, leading to a huge userbase, meaning favorable reviews, a positive reputation, and a profitable project.

The brutal truth is that at least for its initial incarnation, Glass isn't very accommodating to purposes for which it was not intended. In that regard, it's quite inelastic. Smartphones still have the edge here. Some of the design constraints in the system limit some of the applications that are available. Therefore, a simple rule to adopt whenever you find yourself struggling is *always return to the Five Noble Truths*, as presented in the previous chapter. Let the simplicity of the guiding principles show the way and get you back on track.

We're going to look into some of the more prevalent potholes that you might run into when beginning to Think for Glass, and how to make sure you understand the ecosystem and can work your way around them. We want to help get Glass all over the world and give rise to a community that's equally passionate and competent, so this chapter's focus is on some final design considerations to keep in mind to get that done as we begin to transition into the *Development* section of the book. And as you'll see, wise integration trumps sheer force. The really meaty chapters on Glassware coding are right around the corner, so let's run through some of the major snafus that people tend to make when beginning their journey with Glass, so we can get to the good stuff.

See how many you're guilty of, and how many you were able to figure out on your own.

Improperly Implementing Ideas for the Glass Experience

Not every idea is naturally a perfect fit for the Glass application model. As a Glassware developer, you're going to have to get really creative and think hard to figure out how to carve out its place within the ecosystem. Media, particularly video, is an area that's a big part of Glassware, but the dimension limitations of the prism display are the primary challenge. Netflix and Hulu for streaming video and Zynga for web games are organizations whose products don't exactly translate well within the confines of the prism as they exist on larger displays, but there should certainly be clever workarounds or helpful integrations that still let members use Glass to interact with their systems, even if not with full-stack functionality.

We fully expect to see managerial access to instant queues and remote control features for separate monitors like Google TV, or rich second-screen data from IMDb that pops up in the user's HMD while streaming programming on a separate monitor, triggered by timing hints synced with the programming—which several cable networks do currently with tablet apps, notably AMC for *The Walking Dead* and TBS for Conan O'Brien's show. We're already seeing a scenario where Glass can be integrated with Chromecast via the [Google Cast API](#) by the fitness Glassware LynxFit and its [LynxCast](#) feature. Could this possibly mean Glass would be aware of a video that the Chromecast is streaming and display supplementary information cards, personalized to the wearer and contextually sensitive to the content? The sky's the limit!

And relative to the financial services industry, a utility to manage one's investments is an obvious candidate for Glassware, along with a service for banking/lending companies. While a bank's mobile app and web presence let customers do the normal range of transactions, it's largely still a cumbersome process spanning numerous screens and clicks. We're curious to see how personal banking institutions could streamline actions like funds transfers, deposits, and balance queries using the Glass control set of trackpad swipes, voice commands, and gestures, or using the camera as input. Imagine being able to capture a picture of a check and share it to your bank's Glassware, which could then apply OCR software to identify various aspects of the check for denomination and

account, similar to the feature **Concur's Glassware provides** for scanning receipts, and automatically credit it to your balance. Such applications exist today for mobile platforms, so we're excited to see how they might embrace the Glass experience, and how they will start to Think for Glass.

Turkey's Garanti Bank **has Glassware approved and listed** in the official directory, delivering convenient ways members can see their account balances, credit card limits, and much more. Additionally, PayPal developed an app for Android Wear that facilitates payments and bridges notifications about completed transactions.

But obviously paramount to this type of application is member safety. You wouldn't want a program that determined amounts to be transferred between accounts based on voice input to misinterpret the audio because of background noise and accidentally overdraw from a member's balance.

The important thing to recall, especially for those of you working with existing brands and looking to expand your reach to another device, is that *Glass is different*. You need to apply your idea as Glassware in a way that respects the user's real-world activities *and* the Glass experience.

The only thing worse than not having Glassware for a particular product is having bad Glassware.

Treating Glass Like Any Other Mobile Device

Some people are of the mind that since the firmware that Glass is running is based on Android, everything Glass is and everything Glass does should copy the phones and tablets that Google and its licensed partners produce. All aspects of the ecosystem, they purport, needn't deviate from the current mobility model. All Glassware should be listed in the Google Play Store alongside Android apps and Glass should be thought of as just another Android device.

Google's had an interesting history with forking its mobile OS—Google TV and the Nexus Q may share the same source code as devices running on Android proper, but they weren't necessarily lumped in with other existing devices. Forking a platform maintains its roots while also introducing new personality. By this merit, Glass is its own entity and needs to be viewed as such.

It's part of what makes this new realm so interesting...and a bit of a challenge to break new ground in. As we've seen, proper Glassware design means thinking about what situations the users could be in when using your Glassware, beyond just whether they have decent bandwidth. Are they walking or jogging? Are they outside in an open environment where GPS is optimal or within a building? How is the ambient noise level surrounding the wearer? And how about their lighting? Does the environment in which the user would likely invoke your Glassware permit the use of voice commands? Most

of the great Glassware that sprung up as first-generation examples assume some range of activities by the user, and this directly affects the available input controls and how content is laid out (individual cards or bundles as opposed to custom UIs).

One thing that drives this new paradigm is obviously the form factor—there are tons of digital devices around the world running the Android OS in various ways, but nothing like Glass. The headset’s physical design and supported hardware components occupying such a tiny space with enhanced portability make for some very compelling concepts to try as applications. But challenging this is the amount of raw computing power Glass has...which is admittedly less than its Android contemporaries. The battery and processor don’t stack up to other forms like tablets and set-top boxes, as they are the equivalent of a mid-range smartphone.

How Does Glass Fit into Android Wear...or Does It?

The role that Android plays in supported computing platforms continues to broaden, and Glass isn’t just another frontend running on top of Google’s mobile operating system. The rollout of **Android Wear** and Google’s partnerships with smartwatch manufacturers in early 2014 demonstrated the depth of the company’s commitment to supporting wearables.

Riding the momentum (and controversy) of Glass, an entirely new era of mobile computing and multiscreen experiences that bring with it wildly creative new applications, new UIs, and contextual data opportunities for all sorts of situations and environments is upon us, subsequently demanding a new set of tools. Google has carved out its own niche for Glass within the Android umbrella brand—it’s based on a unique user experience that stands apart from anything else the company produces, and writing smart software for Glass demands specific skills.

Sure, if you’re a native programmer there’s going to be crossover and integration, and the design pages of the two systems are strikingly similar. But Android is not Android is not Android anymore. Even within the wearable space, Glass stands alone.

Overloading the System AND the Wearer

Jason works in the broadcasting industry, and like many of his contemporaries he’s considered using Glass as a teleprompter for his live on-air duties. While this would be a really obvious and cool application, it’s highly impractical to the Glass experience because it forces the user’s attention on Glass over an extended period of time, and by the nature of the app Glass would need to stay on and illuminated, which causes the battery to rapidly discharge. It’s still a doable idea, but the core concept needs to be massaged a bit to work with the system constraints of Glass, while still delivering the intended experience. Services like **Glassentation** have proposed neat workarounds to

this idea, so keep an eye on where these go. This very clever Glassware lets users send content from Microsoft PowerPoint slide decks to Glass, to use as a visual cheat sheet when giving a speech—not reading material verbatim, but getting just quick notes to elaborate on.

Glass renders material for the timeline and handles its own version of small-chunk data communications payloads extremely well, but it isn't meant to load apps even written for other Android builds. The Glass version of Android is very adept, but also very specialized. The fact that Glass overheats when running certain native apps is a well-documented concern. The battery life also isn't what you might consider to be top of the line, and will drain quickly under certain usage conditions like prolonged use of the projector or the camera, and processing-intensive operations.

This is another reason the focus should be on terse usage patterns with reduced information, smaller amounts of data sent across the wire, and minimal user input...or user input that actively uses the wearer's surrounding or actions as a staging environment. A canvas, if you will. Where they are, who is around them, and what they're doing become a key part of the application's user interface. Glass goes beyond the computing platforms you've been used to. It's not stationary like desktop PCs, set-top boxes, and gaming consoles; it's portable like smartphones and tablets, but without demanding so much attention from the operator as handhelds do, as to remove them from living in the moment.

Remember, *we're always aiming for microinteractions*. All of us.

We Messed Up, Too

When Glass was but a blip on the radar of the mainstream media and prior to the Glass Foundries, both of us had spent months thinking about how Glass might shape up, leading us to co-organize a Hangout On Air with several members of the community, attempting to prognosticate how the Glass ecosystem would all play out. A few friends from the Glass team were watching in the background, giving us the supportive nod and saying, "You guys really nailed a couple of points...and we think you'll be pleasantly surprised at how off you were on some others."

It's the most fun we've had being absolutely wrong.

Think in Actions, Not in Apps

We introduced this concept earlier: a major way to design your Glassware, as we've stressed, is thinking in terms of what the users will be doing when invoking a certain action, not what screen they may be looking or the arrangement of submenus. This is a shift in computing UIs because it puts the emphasis on the user actively doing some-

thing in addition to starting an app or making a selection. These design decisions are important, but have largely been decommissioned with the simplicity of the timeline model. Usability in the Glass world is equal parts input controls and the activities they're engaged in, such as swiping while strolling down the sidewalk, or barking out "Take a picture!" while taking on a sharp curve at full speed while on a go-kart. Or, what [Allthecooks Recipes](#) does—allowing users to tilt their head up to quickly glance at a secondary set of instructions that list the ingredients for a dish while they're cooking!

Another example of this mindset is the native Timer application. Because it exhibits the capabilities of live cards, it's bound to the parameters of the timeline such as the menu systems and gestures, so it uses a series of custom actions and swipes to invoke its functions. But while it's a good demonstration of what's possible with native coding, we also feel it serves to show what can happen when you put *too much* into a control system. It takes at least seven swipes and taps to create and start a countdown, which is excessive. Clearly, this is an application whose primary choices need to be driven by vocal input, but that's missing entirely. In contrast, the Stopwatch app is incredibly intuitive, as you'd expect. Once launched, it starts a three-second countdown then begins tracking the time, and can be stopped by bringing up the Stop menu item. Just two taps.

Play with these and other apps to see what we're saying. And as part of your routine design planning, chart out the number of input steps it takes you to perform each function your Glassware offers, and see if they can be streamlined or optimized in any way in terms of eliminating unnecessary steps or replacing any actions with other types of input.

You'll also see a tip in [Google's developer docs](#) about not using pinned cards as application launchers (regardless of which Glass framework you used to build your Glassware), but many people don't really grasp what it really means until they experience why it's important—and it's too late. Pinning can be a helpful and strategic way of keeping your Glassware present within the mind of the user, like an application shortcut. For this reason, you may be tempted to pin a card to help your users find your program and to get a leg up on the competition, but this will come back to haunt you, trust us.

The purpose of pinned cards, just like all the other items to the left of the home screen, is to display items that are upcoming, or happening now (items on the right are events that have already taken place). Thus, you should use this space for *data*, *not apps*. Items that are perfect for pinning are static cards about an appointment reminder or a "Don't forget!" occasion; or live cards that update every few minutes (in-game sports scores), every few seconds (store inventory), or even multiple times per second in real time (some sort of animation, like the Compass Glassware).

As part of its maintenance, Glass ensures the timeline doesn't contain stale items and purges any cards older than a week, which includes pinned items. If you pin a card, it will disappear eventually. Pinning can be a helpful tool *for the user* as a digital sticky

note or bookmark, but *as a developer* it's not something you want to use as a normal part of your feature set. Allow it as a menu item, but don't bet the farm on it.

The primary entry points for your Glassware should be voice actions and menu items, both of which are accessible with just a single tap of the home screen or in contextual menus within cards. Either developmental framework lets you set a voice trigger and icon to launch Glassware; Mirror API Glassware can use the preset voice commands “*Post an update*” and “*Send a note.*” And more are coming!

Stick to the Prefab Templates and Styles

You needn't reinvent the wheel when we've got two fantastic resources at our disposal for coming up with great UIs for Mirror API Glassware. The [Google Mirror API Playground](#), which we fully examine in [Chapter 9](#), is replete with a gallery of flexible templates for all sorts of uses, not only provide a guideline for how to structure content within cards, but also get you in the habit of capping the amount of data per card, which makes for better readability. And the [base style rules](#) provide a consistent look and feel to your cards through Cascading Style Sheets (CSS), freeing you up to concentrate on manipulating data and not formatting or positioning HTML elements.

This inherits from the themes used in native toolkits like those available for iOS and Android. It's best to use the same typefaces and sizes for text that everyone else is using so as not to confuse the users or make them weary with radically different designs as they fly through their timeline. Consistency is critical here. When users aren't shifting their focus based on different color schemes and fonts and readjusting for layouts that vary from card to card, they can concentrate more on the information they contain.

The pitfall here would be applying a custom design and/or layout that's too far outside the scope of the recommended style. This might be a tough pill to swallow for established brands with signature colors and fonts, but you can find a happy midrange. LynxFit is a great example of an app that applies the general patterns while still tailoring its feel to give it a unique appeal. The software has its own personality while working within the parameters of known templates.

But if you ever do need to apply a UI that's dissimilar to the timeline motif, like when creating a custom game experience, you use GDK *immersions*, which take users outside the timeline, and thus put them in the frame of mind that they're in a different environment completely. We'll cover immersions and the other types of native interface elements in the GDK chapter in the *Develop* section.

We've mentioned before the importance of being a responsible Glassware designer—adopting a look and feel that's consistent with what other developers are using. This flow makes the overall experience less intimidating for users. See if the groundwork that's already been laid for you does the trick.

Don't Use the Prism Display as a Stage for Complicated Reports

While we're considering the topic of consistent UIs, it's also important to note that Glass generally shouldn't be used for complex graphical data. Enterprise uses are clearly a market that will thrive for wearables in general, but several early adopters have attempted to using timeline cards to hold entire business reports or statistical data. This isn't anathema, but should be implemented very carefully. Tabular data has a place on Glass, but spreadsheets don't translate well. Simple graphs can be effective even in a small display, but not complicated stack charts.

The Glass templates mentioned previously provide a tabular layout structure that works well, and excessive data should be paginated over multiple cards or in bundles. Doing sight analysis on business intelligence data is tough enough...don't impose the additional eye strain of having to check out regression trends in a complicated chart. Maybe use generalized text with color to imply variance, like the stock price template does.

Some companies are already creating Glassware around data analysis, and have managed to find a happy and effective balance—**Dawn Data** is a Mirror API service that assembles next-day traffic reports for Google Analytics profiles already attached to your account that are sent to your headset. And Australian data science company Loves Data has done **some impressive work** in merging Google Analytics with the Mirror API's voice commands to measure Glass usage. They're both very creative implementations that are worth checking out for inspiration.

Glass Isn't Necessarily Bound to Your Phone

There may be third-party services that require you to have a paired device, such as using Glass as a viewfinder or controller, but as long as you have a network connection *Glass is a standalone smart device, and becomes even more valuable by enabling telephony services when you pair it with your phone.* Any network-centric activities that Glass does pass through its associated phone or WiFi hotspot. And even without a network connection, Glass still makes a very capable camera with a ton of onboard drive space. It's likewise a great client for running apps using sensor data like Compass and Stopwatch, and for playing casual games.

You can't transfer files or stream videos and music from your smartphone to Glass, or vice versa. GDK apps, running locally, can exist autonomously without needing any outside help. And there's a lot of room for growth here as far as what we might see with the creative use of offline access. And once you do hook up to a network, Glass sync will download any Mirror API content automatically.

As public WiFi continues to become more widely available and secure, as portable connectivity solutions like MiFi cards become more affordable, and as the software written

for Glass by Google and by outside developers evolves in terms of being feature-rich and utilitarian, Glass may begin to evolve as a standalone computing unit.

As we led off with in [Chapter 4](#), Glass is just getting started.

Unrealistic Expectations for Augmented Reality and Gaming

While a major ad hoc selling point was Glass being the first big commercial product to make augmented reality available to a wide consumer audience and easy to work with for developers (many AR frameworks are based on complicated C++ libraries and custom systems), AR isn't making a huge dent in Glass just yet. The GDK does make these opportunities available to the many programmers interested in creating such experiences natively.

Nonetheless, AR vendors are taking a long look at Glass to determine its viability for their platforms. Opportunities are clearly there to use perspective, geofencing (physically entering a predefined location space like a park or a library, triggering some sort of action within an app), and surroundings as the stage for applications—factoring the user's world as the base of your app is the design mindset we've been discussing in the last three chapters. Wikitude, an Austrian company, has extended its developer SDK, which includes image recognition and location-based features, to support Glass.

The video game Glassware packages *Spellista* and the Mini Games collection show how GDK immersions can be used as gaming stages for the platform. As each uses various components like the various motion sensors, they tend to cause Glass to heat up, but still provide clever examples of casual, sub-45-second gaming, and animated content running in immersions. It'll be interesting to see if these one-player titles, where the user competes against a computerized opponent, are ultimately expanded upon to allow multiplayer and have Glass users from across the globe battle each other.

Also keep in mind the turnaround time for a game project. At the San Francisco hackathon that unveiled the GDK, attendees asked about timelines for getting a gaming title from conceptual idea to in-market product. While months earlier it took three developers working as a project group a few hours to come up with *Ice Breaker* by using the Mirror API, Glu Mobile reportedly spent around four months building *Spellista* by working with the GDK—but capitalizing on its much more robust toolset for sensor access, OpenGL graphics, and rich audio.

Early hackers proved they could install *Ingress* and get it to load in Glass and got it on record, but also managed to shine light on the fact that the game couldn't adapt to the slightly slower chip and reduced memory of Glass (although it is still a very capable computer), as opposed to running the same app on a media device built from the ground up with a quad-core processor and 12-core GPU to handle graphically intense gaming

experiences like a Nexus 7 or Samsung Galaxy S4. Just because you can doesn't mean you should.

Still, Ingress is a perfect example—one of many existing titles—of a game that needs a conceptual rework for Glass to fit the microinteraction model.

Don't Deviate from Default: Using Categorical Voice Commands

Almost immediately after the Glass docs were posted, the community, programmers, users, and the media began speaking out that the stock voice actions weren't enough. The media said more choices were needed, users wanted the ability to define their own custom actions, and developers wanted to expand the set to include audibly driven commands for their Glassware. This resulted in sometimes heated back-and-forth on [“Issue 6” on the Mirror API Issue Tracker](#), with developers passionately advocating for and against giving third parties the ability to add their own custom voice triggers to launch Glassware.

Keeping the number of voice actions to a small level was an intentional design decision by Google, to maintain the platform's light-yet-capable feel—admittedly smaller than the number of voice commands used for Google Now (although that continues to rapidly narrow as more and more cards are added to Glass with each firmware update). The list may slowly expand to accommodate more Glassware ideas, so check back with [the list of supported voice triggers](#), and make sure to review Google's [voice command checklist](#) if you submit a custom trigger phrase. But before you start brainstorming about a cool catch phrase and go off the deep end, let's keep things simple. And to do this, we all need to think at a higher level.

Think categorically, not in terms of plugging a product name. A rule of thumb here is a repeat from our introduction of the Noble Truths: *issue commands, don't start apps.*

Still, a case can certainly be made for why the list needs to be extended. It will invariably hit the wall at some point, and as people get more creative and complex, Glass will need to accommodate that ingenuity, with *“Post an update to...”* and *“Take a note with...”* unable to fit the bill. Fortunately, Google is listening and willing to entertain trigger phrases. GDK Glassware is able to submit a dedicated custom voice command for Google's consideration to use as a trigger phrase to launch the app. You could request a command of your own, or piggyback off of an existing phrase available to you and others in your genre.

The key is to try to select a phrase that clearly conveys the functionality your Glassware delivers, but at the same time is generic enough so that the phrase could be used down the road as a higher-level category to accommodate other products similar to yours.

Again, one of the major aspects of Thinking for Glass is being altruistic. The developer documentation for Glass features **some great tips** on selecting a trigger phrase that fits with the convention of others being used, so check it out.

Not Fully Utilizing Cloud Computing

One type of misclassification we're seeing in these early days is developers not properly using the Internet as a computing engine for their data on Glass. It's very important to understand exactly how cloud computing is implemented on the platform, with data being accessed, stored, and shared from remote servers instead of solely on the device itself. This is a powerful means of enabling multiplatform access, which extends the platform and greatly increases the value of your Glassware, and it's worth looking into, as long as you understand the requirements.

Essentially, *you don't need to use the cloud in your Glassware, but you can get some advantages if you do—and use Mirror if you do.* This itself should lead you to down the right path when deciding whether to build a server-based solution or a native app.

Incorporating online communications into an application is a powerful thing, but what you need to remember as a Glassware designer is how this affects the system in terms of the immediacy of the UI informing the user something is happening, and the impacts it may have on the hardware. The Mirror API's architecture is such that simple messages and commands are fired off to much more robust backend infrastructures, which do all of the heavy lifting and then at some point are able to return simple data fragments to the client. However, you lose a little in terms of the ability to facilitate true real-time communication due to data roundtrips. In contrast, the GDK can wholly utilize resources on the device for storage and processing if the response loop needs to be more immediate, and also call remote APIs. And in both cases, the obvious requirement is for your Glassware to have connectivity.

Fancy and Sherwin Williams' ColorSnap Glass, two Mirror services, both use the clever tactic of color-as-query when letting wearers do searches, as opposed to open-ended text. They analyze the hues of colors from photos the user captures and shares with their respective Glassware services, then return items from their databases matching those shades. Both take considerable processing and offload them to more dedicated resources remotely, with Fancy even displaying a "Searching" card to let the user know the operation is working and matches are on the way soon. The end result is a simple HTML payload that gets returned a few moments later. Glass here is a frontend client to their off-site database. And similarly, Preview, which recognizes movie posters and plays back their trailers, lets Glass wearers scan images of those posters, which it sends to its servers; it then returns a YouTube clip.

And Allen's Glassware, **Vodo**, which he built with the Mirror API, facilitates real-time collaboration for users working in Google Drive on documents, whether they have Glass

or not. Any changes made to the information to the affected files is pushed up to the cloud and reflected in Drive, which others (including Glass users) can see nearly instantly.

Other cloud-centric applications might be image processing, updating databases, communicating with other peers on a system, most multiplatform social systems, or storing progress data for a game.

Knowing when and how to use the cloud capabilities of Glass is a very valuable skill to have, so really think about what your project would be best served by as far as applying the right combination of web API calls alongside local computations and storage. Don't just architect cloud computing because seemingly everyone else is doing it. If it's the right fit for your concept and the experience you're trying to create, make it work.

Choosing the Wrong Development Framework for Your Glassware Project

We're going to segue more into technical development now and cover some fallacies having to do with coding architecture. One of the big headaches that we saw was people demanding additional functionality, hardware, software services, and backend architecture without really taking the time to try to understand the system as designed out-of-box, or giving it a shot—especially developers. The Glass cloud application architecture (see [Chapter 9](#)) is quite comprehensive and handles a lot of issues and challenges that people bring up already.

Perhaps not surprisingly, there were lots of Glassware projects launched that took a design and developmental slant that was ill-fated. We saw many efforts haphazardly implement their ideas as native apps because of the perception that the Mirror API, being RESTful, was too limited. But for many ideas, this does just fine. The result was projects that reeked of overkill—not performant, not fully secure, and not delivering what should have been a true Glass experience—because they thought the Mirror API to be too inferior in terms of capability. They generally looked to the wrong framework on principle to get their things done and it hurt them.

One of the most frustrating things we've seen is when developers naively default to using the GDK to write apps that could have very well existed as Mirror API services, or vice versa. Each has its own capabilities and roles, and it's important to know how to best leverage them to the success of your projects. The deciding factor on the development framework upon which to design your Glassware for many people will come down to skill set—whether they're comfortable working with the Mirror API, which gives you the pick of the litter as far as using a programming language and server-side environment, or the Java-centric demands of the native GDK.

In deciding which framework to base your Glassware project on, the checklist of questions you should be examining should include these:

- What specific type of user experience is the Glassware trying to create?
- Under what real-world situations will someone use this Glassware?
- Is sensor data a crucial part of the feature set?
- Is the Glassware dependent on data/notifications in real time?
- Is it a service whose goal is for publishing, notifications, and sharing?
- Does the Glassware need total control of the camera, or just photos and videos captured by it?
- If location is a feature, how often should the user's geolocation be updated?
- Does network connectivity play a role or can it exist offline?
- Can the Glassware be expressed on the timeline through cards or does it require a custom UI?
- Can the expected experience exist as static timeline cards or does it rely on rapid, frequent updates over an extended period of time, meaning it should be a live card?

As a Glassware developer, you need to know that just because it's a webby framework, the Mirror API isn't a junior varsity-level platform. It's a powerful messaging system that's very capable and can be coupled with other types of platforms to do amazing things. A great example of using the Mirror API where some people might think it's a native app is [OKDoor](#), Glassware from Brivo Labs that demonstrates "social authentication" by sending images of people trying to get into a building to a Glass user in a card, who is then able to tap a menu item to unlock the door and let them in. It uses Brivo's SAM API to control other devices and machines.

And for some truly cool implementations of contextual machine-to-machine hardware hacking, check out the tinkering of Justin Ribeiro from Stickman Ventures, who's done tremendous work using Glass as a frontend for devices like 'net cams, Arduino boards, uninterruptible power supplies, 3D printers, and platforms like Google Drive. He's done some pioneering work for the community based primarily on [Message Queue Telemetry Transport \(MQTT\)](#), a publish/subscribe vehicle for Internet of Things systems. And what's most impressive is that Justin's project uses the Mirror API, and doesn't require native code.

The bottom line is that both the Mirror API and GDK have well-defined purposes, and you should always choose the right tool to do the right job in the right way, in concert with your skill set. And a major part of this is understanding the full range of capabilities that each provides.

Unifying the Camps

Given that web developers and native developers have long “enjoyed” a natural friendly rivalry, an angle that wasn’t properly appreciated with the “RESTful API versus native SDK” debate for Glass was that at least catering to both camps lets either side finally code for the same platform. Traditionally web programmers stuck to the browser while native coders worked with devices. It was always a house divided—Chrome or Android (which, ironically, is literally Google’s organizational division).

So while still not functionally the same, the availability of the Mirror API and the GDK might be the first time both groups wrote for the same system. And that’s significant, and we’re excited to see some crossover or merging between them, and possibly some new bridging solution by Google to facilitate communication between the two platforms.

So while it may look like forcing the Hatfields and the McCoys to sit down at the dinner table together, some good could come out of this by having web and native programmers work alongside each other. And that’s to be commended.

Developers will need to choose a path, given the needs and requirements of their Glassware. The Mirror API certainly isn’t without its constraints, as coders who have used the web stack for years may feel a bit jaded with the lack of JavaScript support in timeline cards.

The GDK route isn’t perfect either. Many mobile developers will wish for the total control native coding provides and will want their code to communicate directly with Glass, cutting out the intermediary that is Google’s cloud-backed system. Direct access to the hardware is good, but could lead to a negative user experience—particularly if the battery heats up too much or drains too quickly. Google has optimized its software to work efficiently on Glass, and you’re going to need to do the same. You may not want to go this route if you don’t need it. It’s more work in exchange for greater command.

Once you’ve spent a sufficient amount of time making these critical decisions, your choice of framework and which path you’ll take should be clear—cloud or native...or possibly both. Programs can easily exist as *hybrid Glassware*, leveraging the best of both frameworks. This lets you use the Mirror API to generate static cards as a frontend that launch an Android app built using the GDK for more detailed processing. Think about a turn-based game like tic-tac-toe. When a friend makes a move, you’d be pushed a static card informing you that he did so, which would include a link that would launch a dedicated activity that contained the game environment. Swiping down would exit the game and return the user to the timeline. This wouldn’t require the game to be running perpetually in memory, but would still let either player know quickly when the other had taken their turn to avoid long waits. Additionally, you could extend the game

to be accessible via the Web, too, giving people more surfaces to play on, with the use of the game being only a few seconds. This is a microinteraction at its finest!

Don't think of it as which horse you should you bet on long term; think about it more generically as a well-rounded Glassware developer.

Done deal.

OK, let's get coding! The most sound advice we can give for properly adopting and adapting the Think for Glass mindset is to thoroughly learn the system and its capabilities first. Whether as an end user, a Glassware designer, or a system architect, getting up to speed with what Glass can do should be your main goal.

If you've read the chapters in order up to this point, we've gone through what Glass is/ is not and how to design Glassware in ways that keep with the key themes of the mission of Glass to make technology available the moment when you need it and then keep it out of your way when you don't. You're now aware of what the aptitudes and limitations are of both the Mirror API and the GDK so you don't have to unnecessarily reinvent the wheel and can choose the most appropriate tool for your work.

In a tech world where few things are certain, keeping an open mind and attitude about Glass while staying pragmatic will let you get the most out of it. That we can guarantee.

In this final part, we examine the steps necessary to put working Glassware together, both as a cloud-based service with the Mirror API and as an installed application with the Glass Development Kit. For Mirror Glassware, the chapters walk you through the sequence of decisions to be made for setting up a new project, obtaining authorization credentials, establishing communications with Google, working with menus, managing subscriptions, and sharing resources with entities. We conclude with some thoughts on how the Glass community and industry at large is already applying Glass technology in real-world scenarios.

- *Chapter 7, Overview of the Mirror API*
- *Chapter 8, Security and OAuth*
- *Chapter 9, Working with Timeline Cards*
- *Chapter 10, Card Actions and Subscriptions*
- *Chapter 11, Sharing Resources with Glassware*
- *Chapter 12, Context Is King: Using Location and Other Signals*
- *Chapter 13, The GDK*
- *Chapter 14, Getting on MyGlass: Glassware Submission, Review, and Distribution*
- *Chapter 15, Reflections on the Future*

Overview of the Mirror API

This is the chapter in which we start to detail how Glassware works with the Mirror API and how to build it. This is a process that has several moving parts and a few different technologies, and requires several decisions to be made in a particular sequence, so we use this discussion to introduce the mechanics of the Mirror API framework and how you make it work for you to architect cloud-based services for Glass. So welcome!

Again, we emphasize that we're taking things a step further as part of our dedication to teaching you how to Think for Glass: we don't merely want to show you how to write Glassware, *we want to show you how to write great Glassware*. And as we've demonstrated thus far, such a talent for maximizing the platform involves amassing deep knowledge and embracing pragmatism.

So, as this is the first scene in the third act of our three-act play on Thinking for Glass, let's wax pessimistic for a moment and get some housecleaning duties out of the way first.

This isn't going to be an exhaustive dissection of the Mirror API's various method signatures and expected parameters. The online documentation does a fantastic job of that already, and like Glass, is changing all the time. Our goal is to arm you with a weapon that has far more shelf life and help you understand the documentation and the Glass environment. And appropriately, this is the level of abstraction that the Mirror API provides you with that you're going to need to embrace.

Let's say that again because it bears repeating (and we'll repeat it again): *we want you to understand what the Mirror API and Google's documentation is talking about, not just provide you some cookie-cutter code fragments*. This isn't a read-and-rip section—it's a lot more.

Life on the Server Side

With its development model based on client-server web programming, the Mirror API lets us use tried-and-true coding techniques that provide several advantages over native development:

Incredibly flexible and very easy to get started with

You can use practically any programming language, developmental framework, and hosting environment you wish. The usual suspects pop into mind, so you can use the Java, PHP, Python, Ruby, Perl, Go, Dart, C#, VB.NET, Node.js, or even Cold Fusion know-how that you've accrued over the years, along with their respective frameworks for desktop, console, web, and mobile platforms—anything that speaks HTTP. You can even change them later as your Glassware evolves! If it can run on a web server and if it can issue web requests, you can use it for your Glassware. That's just about every programming language these days. There are some requirements you'll need to follow (such as having a public HTTPS URL), but they're not onerous requirements. Although we'll be talking about Google App Engine later, this isn't required, and if you're familiar with Tomcat or IIS with .NET (or any other web application environment), you should be able to use them with ease. Play to your strengths and use what you know!

Rapid application development

The compile-debug cycle is shortened because you don't need to load the app onto an emulator or device.

Rapid application deployment

Unlike native apps, you won't need to push an entirely new version of your Mirror API Glassware and have users download/install it. Being web server-based, you need only save the changes, which then become effective the very next time the system automatically syncs. It is highly test-friendly—Glassware is heavily event-based (which we'll discuss shortly), and each event naturally translates into unit and regression tests.

Easy debugging

Tests are a good start, but sometimes you need to debug what is going on. The Mirror API provides ways to query for the state of every event your application has access to. We can do at least some design, testing, and debugging using our own HTTPS commands with our own JSON and HTML content. We can either send test events to Google's servers, to see what it may look like on Glass, or we can mock up events to send to our own web hooks.

Measurement

Web analytics have become highly evolved over the years. We can leverage these tools for our Glassware.

Battery-safe/processor-friendly programming

Because information is delivered in quick cards, there's very little, if anything, you can do with the Mirror API that will cause the device's battery to rapidly discharge or the processor to work so excessively the headset heats up.

While the Mirror API will handle a tremendous number of use cases, it can't do everything. So keep in mind some of the following pitfalls about the Mirror API:

Real-time data delivery

The whole concept of "real time" can be a misnomer these days. Glassware that uses the Mirror API operates as a low-latency vehicle, which is to say it distributes its payload to its intended target on the order of a few seconds, but there is going to be some network lag. If your project mandates immediate notification, you'll need the GDK.

Connectivity is required

One of the big selling points for native Glass development is the ability to handle offline situations. Glass will queue sync operations for when the network returns and you don't have to worry about it, but the Mirror API expects most of the heavy lifting to be handled off-device.

No sensor or hardware access

The full range of sensor APIs are programmatically available only through the GDK for things like light detection, motion, and velocity. Access to the camera, microphone, and GPS are more passive with the Mirror API and usually rely on the user taking an explicit action.

Purely event-response driven

There are just some things you can't do when you're talking about discrete events. Streaming video (either sending it from Glass or watching it on Glass) just can't be done frame by frame, for example.

But fear not. Most of these issues are addressed with native development and the GDK, although we'll see it has its own set of pitfalls. And don't dismiss developing with Mirror out of hand—there is still lots you can do with it that is far easier than with the GDK.

Mirror API development, as we noted, is incredibly liberating in the sense that any server-side language can be used to write a Glass service and tap the RESTful endpoints that the Google Mirror API provides. This means you can easily build a server in a web document, a desktop application, for a mobile OS, or even one that runs at the command line or via SMS. You can do whatever is most comfortable to you to build, most convenient to use, and most appropriate to satisfy the use case. Client libraries in a rapidly expanding array of languages are available for download at the [Google Developers Glass site](#).



Servers Versus Apps

We're starting our look at developing for Glass with a thorough look at the Mirror API, which provides a network-service perspective toward writing Glassware. Those of you already familiar with Android may feel left out or want to skip ahead. Native development with the GDK will be covered in a few chapters. GDK programs are more like traditional apps—running on Glass itself instead of sending all the events to a remote service.

We strongly suggest, however, that you read through and become familiar with the Mirror API as well. Many tasks, particularly those that will send things over the network, are particularly well suited for the Mirror API, and understanding how it works will help you write better GDK apps.

And here's a little tip—the Mirror API and GDK can even work together. (We'll look at how later.) This makes a case where the result is truly greater than the sum of its parts.

But what may not be so evident is that this also means you're free to use other frameworks like Google Apps Script and integrate with the Glass backend to communicate across platforms for things like writing to and reading from files in Google Drive.

A Funny Thing Happened on the Way to the Hackathon...

Allen based his hackathon project built at the New York City Glass Foundry on the concept of building Glassware with two technologies that at the time didn't have much documentation or examples relative to Glassware programming (and largely still don't)—coding up a to-do service using Google Apps Script (GAS) that would interface with Google Drive, an early version of what would ultimately become **Vodo**. This allowed items to be viewed and marked as completed, and be stored in a spreadsheet, or would track changes to word processing documents. Worksheets represent individual lists, which could be shared with others. To facilitate communication between GAS and the Google Mirror API, Allen wrote a simple proxy handler in Java. He says, "The key thing to understand is that each interface was designed for the environment. I didn't try to duplicate a spreadsheet on Glass—but I was able to get Glass to show a good representation of a list."

He explains his idea in full detail in a [YouTube video](#).

If you choose to develop with the Mirror API, you also have your choice of IDEs to work with—whether you enjoy the rapid coding atmosphere of the Google Developers Console (formerly the API Console) with Chrome's Developer Tools or the visualization for timeline cards in the **Mirror API Playground**; or if you favor a full-blown editing and

debugging environment of something along the lines of Eclipse or Visual Studio; or if you're a vi, Emacs, Notepad++, TextMate user or have Sublime Text savvy; or if you prefer web-based coding with something like Cloud9 or ShiftEdit, or like to go old school and just use good old Notepad. It's completely up to you.

You're also free to move around with hosting, with the cloud model being completely platform-agnostic. You can use affordable commercial web hosts, or enterprise-level scalable providers like Google App Engine, Amazon Web Services, and Windows Azure, or even host services yourself out of your room on a commodity machine. (We don't recommend the latter, however, and we'll explain why shortly.)

Events: The Building Blocks of the Glass Timeline

We have mentioned and will continue to mention events throughout this book, especially in the *Development* section. If you're a seasoned programmer, you might have figured out what we're talking about, but you might be a bit surprised. One of the things we realized as we began working with Glass is that it shows us things that have happened, will happen, or are happening. Similarly, most Glassware either notifies the wearer of things that are occurring, or receives messages from Glass about things that are taking place.

In this chapter, we'll explore a little bit more about what we mean by an event and how these events get between your Glassware and Glass (and vice versa). Since privacy and security of events are important, we'll look into this a bit in [Chapter 8](#) and discuss what role OAuth plays. We go into (a lot!) more details about sending events to the wearer in [Chapter 9](#), as we discuss the timeline and the timeline cards that represent our events. The tables are turned in [Chapter 11](#) and [Chapter 12](#) as we hear how our users can generate events of their own by replying to cards or generating their own cards with powerful built-in tools the Mirror API provides like subscriptions and sharing. We round out the Mirror API in [Chapter 12](#) as we discuss a special type of event, the Location event, and how it ties into other events as well.

We feel using events is a great way to think about what goes on with Glass, but the notion is a little different from what you might be used to. Don't worry—it may take some time for the concept to sink in, but once you have that “A-ha!” moment, you'll see how everything works based on this simple premise.

So let's dive in and take a look at how Google handles these events, where your Glassware will fit into the picture, and why you should care in the first place.

The High-Level View

We've talked about Glassware, and we've talked about Glass itself, but how do the two actually talk to each other? If you've done any web development at some point in the

past 15 years (and probably if you've done any other Internet-based programming in about that time), you'll find **Figure 7-1** somewhat familiar. It represents a device, working through a proxy, communicating with a server. In this case, Glass is the device, Google acts as a proxy for Glass, and the proxy is communicating with your server that runs Glassware.

The Mirror API builds on this legacy in many ways. The communication channel between Google's server and you is HTTP (actually HTTPS, but we'll discuss that more in a bit). You will use standard REST HTTP commands such as GET and POST to update the events sent to Glass. The contents of the HTTP payload are JSON objects. Unlike normal HTTP, however, the communication is bidirectional—your Glassware will register a web hook with the proxy, and Google will use HTTP to send events to you via these interfaces. We'll look at what these messages look like in just a moment.

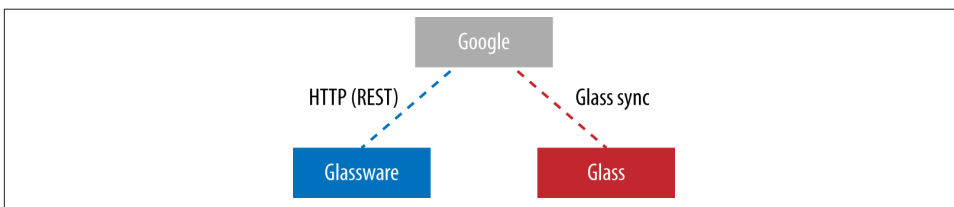


Figure 7-1. The Mirror API architecture (image courtesy of Google)

What about that part on the right of the diagram labeled “Glass sync”? How does Glass communicate with the mothership? Magic. Well, no, not really...you can see the upcoming sidebar for more about how the synchronization component of the ecosystem works, but this is an aspect that we generally don't need to concern ourselves with when we're working with the Mirror API, and this is a good thing. Technically speaking, as far as how information flows between users, your Glassware, and Google, wearers never talk directly to your server.

When users invoke commands through voice, gestures, and menu items, when they share a resource with your Glassware, send data to it, or when they subscribe to receive notifications from your program, they send these requests to Google, which then sends the messages in expected data structures in JSON to your server for processing. Any confirmation messages flow back in reverse from your server to Google, and then to Glass clients. The converse applies, too, for situations where your Glassware has new data to push. Any subscription information is sent to Google, which then delivers the messages to clients efficiently. *Your Mirror API-driven Glassware never talks directly to any Glass headset, no matter the scale.*

As **Figure 7-2** shows, Google is the ultimate middleman, a friendly intermediary that keeps everything having to do with messaging in line. Google handles all the complex situations of batching multiple events into a single job and queuing them up to make

sure they are delivered over a network that cannot always be relied upon. Glass sync ensures that once we hand an event off to Google for delivery, it will eventually be delivered to the correct Glass device; once an event takes place on Glass, it will eventually be delivered to us as well.

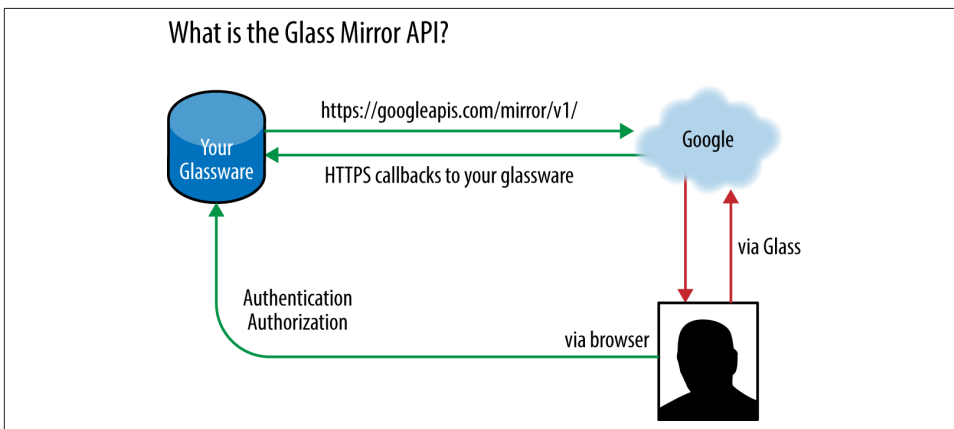


Figure 7-2. Data flow in the Mirror API

Under the Hood of Glass Sync

The Glass sync component is the phase in which Google manages all of the messaging operations for delivering the messages and synchronizing them with their intended headsets, such as queuing, batching, and optimization, as well as handling situations where the user's headset may be out of range of connectivity or isn't powered up at all. At the time of this writing, this part of the system runs on a modified build of **Google Cloud Messaging for Android** (formerly known as Android Cloud to Device Messaging), which is the same system that runs Google Play's over-the-air install feature for mobile apps. This infrastructure is subject to change at any point, and access to it is prohibited to external developers anyway.

While we as third-party software developers don't have to worry about the intricacies of this side of the Mirror API framework, it means we also don't have the ability to configure several cool aspects of GCM. We hope that someday we'll be able to tweak some settings in that regard for more direct control of messaging because there are some neat flags we'd like to enable.

How Your Server Talks with Google

Returning our focus to the lefthand side of [Figure 7-1](#), what does the Mirror API look like? As noted, our Glassware will be making HTTPS calls to Google’s API server. Each request will contain an authentication header that has resulted from a user authenticating themselves using OAuth2. As with all HTTP-based requests, we will be issuing REST-like commands against a URL that represents the object we will be working with (usually an event). If we need to specify values (and what is the point if we’re not specifying values at some point?) we’ll use either values in the URL, URL parameters, or JSON objects in the HTTP body. If we are sending media, such as a picture or video, we will send them using a multipart body, part of the HTTP standard. The text displayed on our user’s screen on Glass can be formatted using a simplified HTML markup, with some predefined CSS classes and other CSS styles we can define.

Data comes back to us in one of two forms. If we are issuing a query to Google, for events that meet a particular criteria for example, then the HTTP connection will contain the response, usually in the form of a JSON object. But for events that take place on Glass, we shouldn’t need to open up a connection to Google just to see if there is anything new. Instead, we will register a callback as our aforementioned web hook, which is a URL on our server that Google can contact when there is something to tell us. We’ll get an HTTPS connection to our server from Google, delivering a JSON object containing the event information. In either case, media is delivered to us in the form of a URL to fetch, instead of the full chunk of data.

If you’re not familiar with REST, you can generally think of it as a way to reference certain atomic database operations on objects. So if we are talking about events, we would be interested in doing tasks such as adding a new event to Glass, searching for events that meet specific criteria, fetching an event by a unique identifier, deleting an event, sending all new data about an event, or updating just some fields in an event. (But this is a spoiler...for [Chapter 9](#).)

If you’ve used JavaScript, you’re probably at least vaguely familiar with JavaScript Object Notation. (JSON, get it? Totally unrelated to one of this book’s authors.) This is a simplified, yet easy-to-read, way of distributing a data object, composed of attributes and values, where the values may be numbers, strings, Booleans, other objects, or an array of any of these. If you’re new to JSON or have only worked with formatting content in HTML, an extremely helpful post is available [on O’Reilly Answers](#) giving details, but here is a quick taste so you get the idea:

```
{
  "question": "string value",
  "numeric_example": 42,
  "is_this_the_answer": true,
  "nested_object": {
    "nested_attribute": "totally exciting",
    "nested_boolean": false
  }
}
```



```

    },
    "possible_questions": [
      "How many roads must a man walk down?",
      "What does our amp go to?",
      40
    ]
  }
}

```

In the preceding code example, we see a very basic, yet illustrative, object represented in JSON. The whole thing is surrounded in curly braces, indicating it is an object, and the nested object attribute uses this brace notation as well. Arrays of values are similarly marked by square brackets, and can contain mixes of types. Attributes of the object are surrounded by double quotes, to the left of a colon. Values are to the right of the colon. Attribute/value pairs are separated from each other by a comma. All values are constants—you can't call functions or do math formulas to compute a value.

If we were documenting this (and this is how Google documents it), we might describe the resource something like so:

```

{
  "sample_string": {string},
  "numeric_example": {integer},
  "is_this_the_answer": {boolean},
  "nested_object": {
    "nested_attribute": {string},
    "nested_boolean": {boolean}
  },
  "possible_questions": [
    {string or integer}
  ]
}

```

And then document it in table form as in [Table 7-1](#).

Table 7-1. Object model for a JSON resource

Property name	Value	Description
sample_string	string	A sample string, representing the ultimate answer.
numeric_example	integer	A possible numeric answer to the question.
is_this_the_answer	boolean	Is this the answer or not?
nested_object	nested object	An object representing a nested resource.
nested_object.nested_attribute	string	The attribute for this nested resource.
nested_object.nested_boolean	boolean	Random Boolean for the nested resource that was included.
possible_questions	list	A list of possible questions for this answer. May be string or numeric.

We've stressed one of the advantages of the Mirror API being your ability to program Glassware in whatever language you choose, so why are we going into so much detail about HTTPS, REST, and JSON? Because we need a lingua franca that everyone can

understand to illustrate the concepts behind how the Mirror API works and this is in line with how Google’s documentation reads as well. Whether you’re just starting out with Glassware development or are new to programming entirely, it’s helpful to have a consistent lexicon. The fact that the technologies are also open standards means that everyone else using them will refer to them consistently.

Don’t fret, however—Google’s documentation also talks about many common languages, and the wrappers they use to translate these concepts into ones you are probably more familiar with in your specific language. We’ll be pointing you at those language-specific references when the time comes, but we will be talking about the concepts more than the syntax.

Components of the Mirror API

Speaking of concepts, this seems like a good time to introduce you to the components of the Mirror API itself. We’ll be going into details over the next few chapters, but you’ve been such great readers so far that you deserve a little bit of a preview. In a nutshell, whenever your Glassware communicates with the Mirror API, it does so in terms of the collections that the API supports. This is the common grammar used by the JSON messages you’ll send to and receive from Google, essentially Mirror’s object model.

There are five collections you’ll be using, with each having its own distinct set of properties and expected values:

- Timeline
- Timeline.attachments
- Subscriptions
- Contacts
- Locations

The easiest and most basic thing you’ll be working with are events related to the timeline. You’ll be inserting, deleting, and updating cards there by sending these commands to Google’s servers. There are a lot of properties to set on a timeline item—ranging from the text to display on a card to menu items attached to each one. And speaking of attachments, you can also attach images or video to be shown along with each card. We’ll go into all of the options (and there are a lot of them), how to use them, and what commands you’ll be sending to the server in [Chapter 9](#).

As we can see from [Figure 7-1](#), however, our communication with Google is a two-way street. So in [Chapter 10](#) we’ll learn a bit about the menu items that we can attach to a timeline item and how we get notified when someone selects one. To do this, we’re going to have to provide a callback URL to a subscription that we send to Google, and we’ll learn about how to manage our Subscriptions and what it looks like when we get the

callback event. We'll also learn about what we might do with our timeline item once a menu on it has been called.

A special type of event occurs when someone shares a timeline item with us. We will need to create one or more contacts, register them with Google, and these may show up in the list of people or things that people can share a card to. We'll get the callback via the same Subscription method, but [Chapter 10](#) will go into details about what additional information we can provide with a contact, what it looks like when something gets sent to one of our contacts, how to get the timeline item that was created just for us, and what we should do with it.

Finally, Google will send us a specific event notification if we subscribe to location changes. [Chapter 12](#) will show us a bit about location information that may be attached to Timeline events, how to create maps for our cards, and how we can get user location to help keep our events timely. (You remember that was a Noble Truth, don't you? If not, reread [Chapter 5](#).)

Preparing Your Project

With this high-level view of what happens on the left side of the diagram, and a quick overview of what messages are exchanged between our Glassware and Google, it is time to start zooming in on what you actually need to do to use the Mirror API. You probably have an idea for a project. Maybe even more than one. We're not quite at the point of building it, but we're getting ready to set it up. To do so, you're going to need to have a few things ready.

First, you're going to need a server. Easy, no? Well, there's more to it. Don't forget you'll need a web server on it, and you'll probably want a data store of some sort, as well (i.e., a database, XML file, spreadsheet, flat file, etc.). Google will need to contact your server, so it has to have a public IP address—the machine in your house probably isn't enough. Since Google will be contacting your server via HTTPS, you're going to need an SSL certificate. And, with all your hard work, your Glassware is going to be a big success so you'll have to be prepared to scale this up. Is this sounding like a lot of work yet?

Would you like somebody to do all of that work for you? Fortunately for you, somebody is. If you'll be developing in Python, Java Platform Enterprise Edition, Go, or PHP, you can use Google App Engine (GAE) to host your project. GAE takes care of providing the hardware, a working web server that supports HTTPS, the necessary SSL certificate, and even a hostname that you can use for your Glassware. It is free to set up a basic GAE instance, so you can get familiar with it and begin developing your idea, and you can transparently add more instances as your needs increase. You can find out more about GAE and the rest of Google's Cloud Platform at <http://cloud.google.com>.

Once you have a server set up, it's time to set up a project with Google. If you're using GAE, or any of the other Google Cloud Servers that we just covered, you'll already be

familiar with the Google Developers Console. If you're not, we'll be using the Developers Console to set up access to the Mirror API. The console allows us to do several things: create a project, indicate that it will be using the Mirror API (and any other APIs we may need), and get a Client Key and Client Secret that we will use as part of the authentication and authorization process. You'll also use it to find out helpful trends about your quota usage, such as what time(s) of day your API calls are being made most. (We'll talk about the quota in a moment.)

API Console Versus Developers Console

If you've used Google's API Console in the past, you may be wondering what this Developers Console thing is, and why we're pointing you at it instead of the tried-and-true API Console. Google appears to be shifting all its future efforts to the Developers Console, and although it isn't fully functional yet, it seems destined to replace the API Console in the (relatively near) future. Now's the time to start getting used to it and sending in feedback about the up-and-coming hotness.

Oh, and why call it "Developers Console" if it does more than control access to Google's Cloud Services? Your guess is as good as ours.

We're going to create a new project that we'll use for the next couple of chapters to get us started with the Mirror API. Our journey begins at <http://cloud.google.com/console> where we will either log in to our Google account or create a new one if we've somehow managed to avoid this before. If we've never set up a project, we're greeted with a welcome message, as shown in **Figure 7-3**.

If you have created a project in the past, you'll be greeted with the list of your projects and the same red Create Project button.

As you might have guessed, we'll be clicking that button to create a new project. A window will pop up prompting us to enter some basic information. The Project name should be set to Glass Playground for reasons that will someday become clear. You can keep the Project ID to one that they suggest, or come up with something clever on your own—but it has to be unique across everyone's projects, so you might want to just accept what they suggest or incorporate something with your name (**Figure 7-4**). Finally, you'll check the box saying you've read and accepted the Terms of Service (you have, haven't you?) and click the Create button.

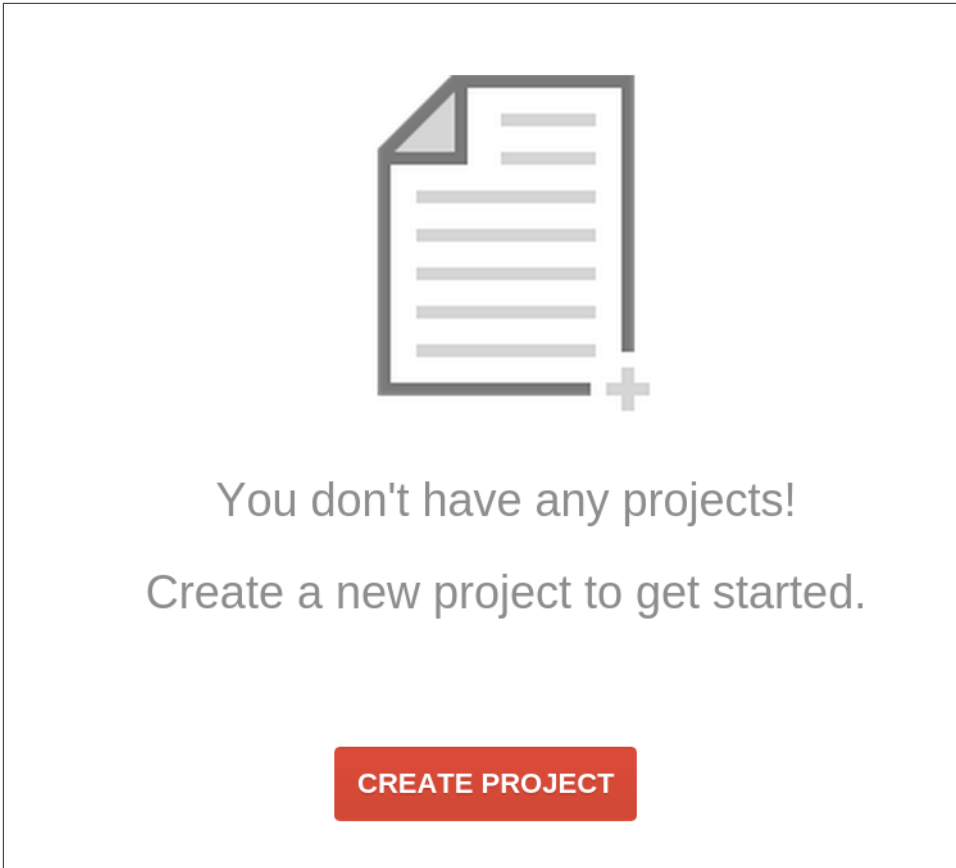


Figure 7-3. A new Developers Console project

It may take a few moments, but once the system has created the project for you, it will place you on the overview screen for your Glass Playground project. You can explore some of this, if you wish, but we'll be going straight for setting it up so we can use the Mirror API. If you select the APIs & auth menu item on the left, it will open up the API submenu and show you a (long) list of Google APIs that you can enable. Some of them may already be turned on—you can safely turn them off since we're not interested in any of the cloud APIs right now. Turn them off by clicking the green ON button, then scroll down until you see the entry for Google Mirror API (they're arranged alphabetically) and click the OFF button to turn it on, as shown in [Figure 7-5](#). Intuitive, right? You'll be prompted to acknowledge the terms of service. Once you have, the API will be turned on and moved to the top of the list.

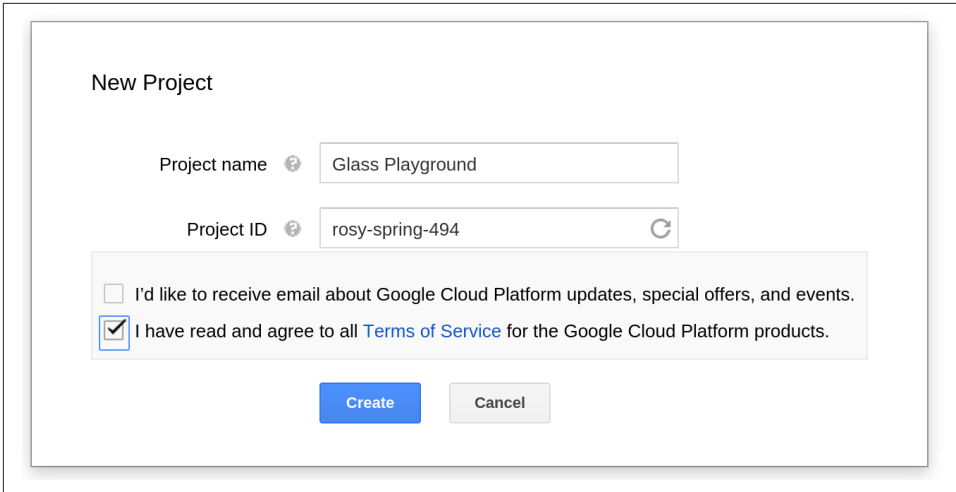


Figure 7-4. Name your project

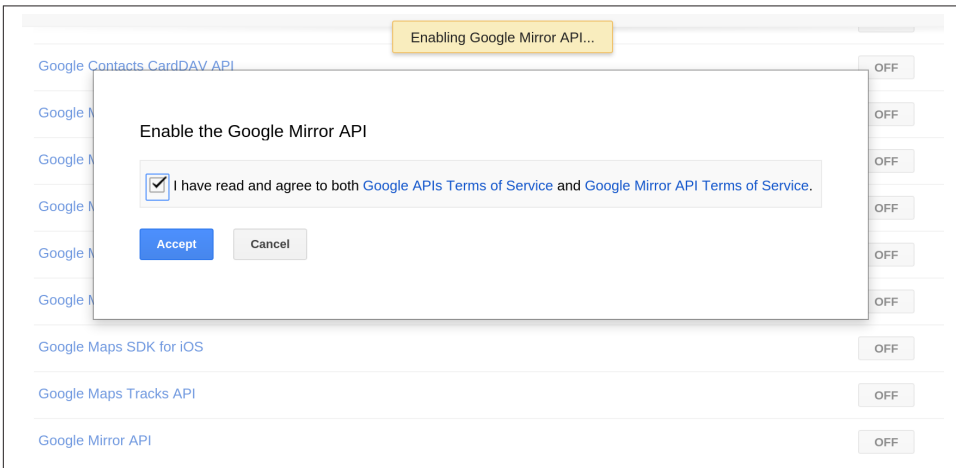


Figure 7-5. Flip the switch on the Mirror API

We're all set! Well, not quite.

The Glass Ceiling—Your Project's Quota

The console also gives you access to some basic analytic information. We can see how many calls we've made to each API we've authorized, as well as how many calls per day we're permitted to make. *Waitaminnit...* we're limited in how many calls we can make?

Yup—Google limits how much we can use each API to prevent abuse, particularly while developing new applications with it.

Initially, you're allowed 1,000 calls for your project each day. This resets at midnight Pacific Time (and sorry—no rollover minutes). You can see your quota, current, and historic usage for your use of the Mirror API by clicking from the APIs & auth menu page and then clicking the Google Mirror API link. Links for quota and reports are available at the top.

A thousand calls per day is a pretty good level for initial development—it lets you get your feet wet while you learn, and it prompts you to think a little bit about making sure you're making the most efficient use of the Mirror API. (We'll share some efficiency tips as we go along, but don't stress too much about it.) It should even be enough for a few test users to help you get going before you're public. As you widen the program, you can request more quota from Google, as long as you justify the increased usage. Once you go live, of course, you can ask for an even bigger quota—but we'll be covering that later.

We're getting ahead of ourselves a little bit, but what counts against your quota? Every action that *you* take for *each* user by calling the API counts as one call. Batching calls won't help—a batch of 10 API calls, all sent at one time, still counts as 10 calls. If we look at [Figure 7-1](#), the arrow from your Glassware to the Google Cloud indicates what counts against the quota. The arrow from the Google Cloud to your Glassware, however, doesn't. Similarly, actions taken with just the GDK don't count against the quota.

Metrics on Wearables

It is important to make sure we know how our users are actually using our apps, and modern toolkits like Google Analytics, comScore, and WebTrends generate powerful reports about users browsing your website in real time. They can also be applied to native application tracking (like with the Google Analytics SDK for Android) to gauge which screens users are viewing, the length of their sessions, the effectiveness of integrated marketing campaigns, response to tweets with embedded links, devices and user agents, error logs, crash reports, and API calls. It makes sense to want all of this for our Glassware too, right?

At the moment, we've got solid metrics available in Developers Console that analyze how many calls we make to Google's API servers. But since the Mirror API is a server-side technology, we can't implement the frontend JavaScript libraries like we can with Google Analytics. We can, however, track messages being passed to and from the server in response to the various Glass events like menu item selection and data sent across the wire. The [Google Analytics API](#) lets you define interactions that can be tracked by adding events when you get a callback or when your Glassware inserts or updates a card on the timeline.

Over time, you should keep an eye on the console to make sure you're not reaching your quota limit.

The Even Bigger Picture

This is good for starters. But a real project will involve much more. If you're using Google App Engine to run your projects, you'll be using the console to control your instances and to manage the project and its financing.

Most Mirror API projects will be making use of other Google APIs. At the very least, you may want the [Google+ API](#) so you can get information about the users when they log in. You may wish to use Drive, [Cloud SQL](#), or [Cloud Storage](#) to store configuration information for your users. If you'll be using any of Google's other services, this will be the place (and time) to enable them. But we won't need to for the Glass Playground.

What we will need, however, is to set up the security credentials. We'll also be doing this using the console—but we'll explore why and how in the next chapter.

Security and OAuth

Before we get into the details of how we handle events, the moments that are generated by the Mirror API and expressed as cards, whether on the right of the home screen (events that have already taken place) or to its left (events that are upcoming), we need to take a moment to focus on security. Since Glass is such a personal and intimate device, as we've discussed, Google takes the notion of security very seriously. As we've seen, when you set up Glass you tie it to a Google account, and you're not expected (or allowed!) to share your Glass with someone else. Google expects you, as a Glassware developer, to take security at least as seriously.

DON'T PANIC.

The Mirror API has security tightly integrated into it, so you almost can't avoid doing the right thing for your users. We'll start here and see how security is handled within events. Additionally, it is built on top of OAuth2, and many of the libraries you'll be using take care of the OAuth2 details for you.

The major fact you need to know about Glassware and authorization is that any Glassware built on top of the Mirror API has to authorize the wearer's Google account for the Glassware to receive updates or be able to share resources like messages and multimedia with it. GDK Glassware, on the other hand, being installed applications that run locally on the device, only need to use authorization if access to certain APIs requires it.

There are some things you will need to know, but the basics aren't that difficult, and we'll cover all this later in this chapter.

Event Security: Google the Bouncer

The security model for events in Google Glass is pretty simple, straightforward, and important—*every card is permitted to exactly one person and exactly one piece of Glass-*

ware that they've approved to run on their behalf. So the picture you've taken on Glass is “owned” by the Photos Glassware representing you. The message you've received is owned by the SMS or email Glassware and represents you. Rogue Glassware can't suddenly tap into this and get access to your messages or pictures. Unless, of course, you explicitly share it with that Glassware.

But when you share a card, Glass can't just permit your Glassware to own that card. Remember the rule—one card, one program (representing one person).

What Glass does, instead, is create a whole new card with a copy of the same thing. If you're paying attention to your timeline, you'll even see the card being duplicated. Once the new Glassware is told about this card, it can do anything it wants with it. Rewrite it...delete it...anything at all. But the original card remains untouched.

What about messages that need to be sent to more than one person? Each person needs to get their own card about it. So if you're chatting or emailing two other friends, and you send a message, a total of three cards are generated—one for you and one for each of your friends. They may all have the same data on them, but they're different because each person can treat it differently—we may delete the message, while you keep it (or share it with someone else, generating a few more events and cards).

This may seem trivial and unimportant, but it will save you a lot of frustration as you're trying to write Glassware that deals with events shared with you or that you expect the user to share with others. So always keep it in mind—if your Glassware didn't create it, or the user didn't explicitly share it with you, it doesn't exist.

But this raises a very good question. How does Google know that your Glassware is... well... your Glassware? For that matter, how does it know that one user is a particular user when your Glassware is acting on their behalf? This is the job of OAuth, application credentials, and the Google authentication system.

OAuth: IDs and Secrets

Some of this sounds a little theoretical, but understanding it will help when we actually implement OAuth in our projects. Bear with us a little—we'll get to details in a little bit, and understanding some of the basics will pay off when you're trying to diagnose authorization problems.

OAuth started as a standard way for network clients to request permission to use a resource on behalf of a user, and for those resources to indicate the client is authorized to take the requested actions. It has since grown (some would say overgrown) from those modest beginnings, but still has authorization at its core. Google uses this later version, known as OAuth2 (also expressed as “OAuth 2.0”), for most of its web services including the Google Mirror API, and this is what we'll be talking about when we

use the term OAuth. Other providers use different versions; most notably Twitter uses OAuth 1.0a, and we won't be discussing the differences here.

OAuth2 is more a framework than a specific protocol, so you'll see some variations between different OAuth providers—we're going to be focusing on Google's implementation, but the concept should be familiar if you've used OAuth from other companies.

The general idea behind OAuth is that services, such as Google's Mirror API servers, can't fundamentally trust client programs to behave. If left unchecked, an unscrupulous programmer could create an app that claims to represent a user and then do all sorts of mischief that the user never expected. OAuth acts as a check on one part of this—applications must request permission to do certain activities, known as *scopes*, which users must explicitly approve.

But how is the user going to give this approval? OAuth can't tell the application itself since the server isn't going to trust the application. Instead, OAuth specifies how the application sends the user over to a site controlled by the server where the server can get permission directly from the user. Once the user has granted that permission, OAuth specifies how that approval has been sent back to the application so it knows it has the permissions it has asked for.

There are a number of different ways to do this depending on your needs. Apps that run on a mobile device, for example, have different needs and requirements than those that are running on a web page where the user stays on the page. And these are different needs than servers that will have to act on behalf of the user when the user isn't at the keyboard. We'll be focusing on this last one, but you can get full details about all of them at the [OAuth2 documentation](#).

All of this may seem a little abstract, but with a base understanding of the background, it is time to start getting into some of the details. The good news is that the details aren't that bad. The even better news is that the programming languages that Google supports all come with OAuth libraries to make some of this more straightforward. So let's go through the dance moves OAuth requires.

Will You Come and Join the Dance?

Let's assume, for the moment, that a user has spotted your Glassware on the MyGlass website or mobile app. Since we're talking about Glassware, this is a pretty good place to start. They'll turn the switch ON in MyGlass for a service or application by selecting its card and then clicking the toggle switch...and this opens a new window that starts an elaborate dance in their web browser between you and Google's systems.

Step 1: Redirect to Google for authorization

MyGlass opens this window and redirects it to a URL on your website to handle user authorization. The first thing you'll do is redirect the user to the OAuth URL, providing some information as part of the parameters to the URL. The base URL at Google for this is <https://accounts.google.com/o/oauth2/auth> and we'll add the parameters as query string parameters. Make sure you use HTTPS—regular HTTP connections are rejected.

There are a number of different parameters controlling some specifics, but for your services, you'll be focused on just five:

`response_type`

This needs to be set to the string value `code`.

`access_type`

This should be set to the string value `offline`. This will enable you to get a refresh token (a what? we'll explain shortly) so the user doesn't need to be at the keyboard and authenticating your Glassware every hour.

`scope`

These are the permissions you are requesting. You can request more than one, and they should be separated by a space. For access to most Mirror API services, you'll need to request the specific timeline scope: <https://www.googleapis.com/auth/glass.timeline>. We'll talk about scopes a little more in a moment.

`client_id`

Each application requires its own unique identifier, and the `client_id` is how we tell Google what our identifier is. Remember in the last chapter we set up an application using the Developers Console? In a little bit, we'll be using it again to find out what our ID is.

`redirect_uri`

Once the user is done on Google's site (which we'll cover in a moment), Google will redirect back to us. But we need to tell it where to redirect to, and this is how. We also need to give the Developers Console a list of valid URIs to redirect to, for security purposes. You guessed it...we'll talk about this shortly.

So, five parameters. Two of which you don't need to change, and three we'll be covering later. Clear as mud, right? Trust us, it will make sense soon.

So for some hypothetical Glassware that we've written, we might redirect our guest to something like the following. We've broken it up into multiple lines to make it easier to read, but note that it should be all one line and should be URL encoded (which we've also done):

```
https://accounts.google.com/o/oauth2/auth?  
response_type=code&  
access_type=offline&
```

```
scope=https%3A%2F%2Fwww.googleapis.com%2Fauth%2Fglass.timeline&
client_id=1234567890&
redirect_uri=http%3A%2F%2Fexample.com%2Fauthinfo
```

So, we've sent our users back to Google. What happens there?

Step 2: Authorization (and maybe authentication)

In most cases the users will have logged in and authenticated themselves already. After all, you need to log in to MyGlass in the first place. In a couple of scenarios, users may be prompted for their login info (or at least account) again. But most of the time they'll proceed to the authorization step.

This will be a screen prompting them with information about the various scopes you've requested, but in a format that is a little more straightforward to understand. **Figure 8-1** conveys this screen. **Figure 8-2** shows you the details about each permission if you click the information icon (the circle with the "i" in the middle). It will also show them information about your app so they can try to get a sense if they can trust you.

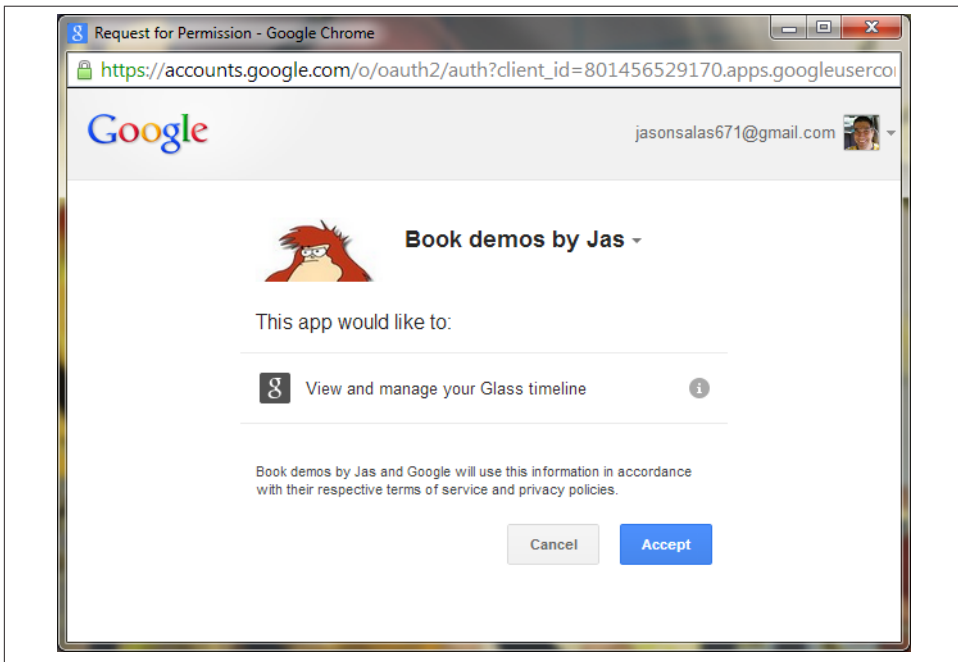


Figure 8-1. Google's OAuth provider screen for Glassware

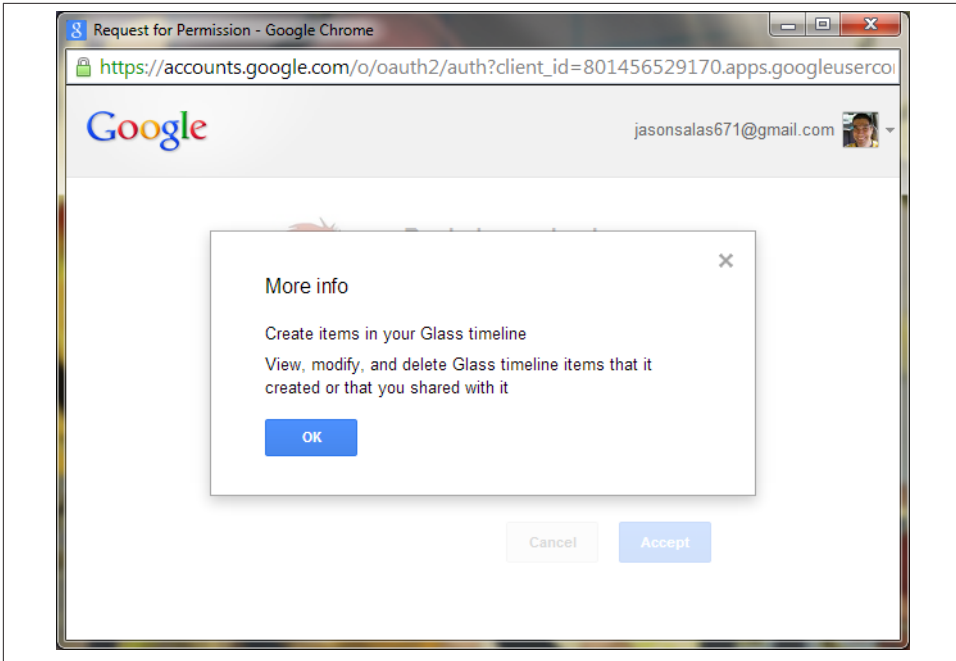


Figure 8-2. Permission information

Once they approve the scopes, Google will redirect them back to your website, at the URI you requested with some additional information. This takes us to:

Step 3: Process authorization code (or error)

Let's handle the bad news first. If the user doesn't trust you, or if something else went wrong, you'll get an "error" parameter with some (but not much) information about the rejection. Sorry, maybe it will work out better next time.

In most cases, however, you'll get back a query parameter with a code, sometimes called the *authorization code*. You don't need to save this code, and it wouldn't do you much good if you did—typically it is only good for a few dozen seconds, at the most. Instead, we're going to exchange it for a more long-lived access token and an even longer-lived refresh token.

Why This Middle Step?

It isn't immediately obvious why OAuth2 goes through this particular dance step. Why can't it just return to you the access token and refresh token directly through this redirect? Why is it giving you this intermediate code instead? The answer lies in who is doing

the work, how much you can trust them now, how much you can trust them in the future, and how secure your communication channels are.

Up to this point, we've been using the user's browser as a go-between for us to communicate with Google's servers. We need to do this because we need the users to approve the permissions, and they need to do it directly with Google. The code being sent to us contains proof that they have, so it is pretty valuable! And yet, the code is being handed back to the user to hand to us. While the way we're doing it is pretty secure, it isn't perfect, and there is a chance that the browser might store this URL for a long time. So we limit the window where this code is good. If someone gets it outside of that window, they can't do anything with it.

From here out, however, we don't need to go through the user's browser anymore. We're going to talk to Google's servers directly ourselves. So we can use a token that will live for a little bit longer—an hour by default.

We will do this exchange directly, opening an HTTPS connection to a Google URL and doing a POST operation to the same URL (<https://accounts.google.com/o/oauth2/auth>) with the code and some additional parameters to help prove who we are. What parameters this time? Well, most of them will look pretty familiar:

`grant_type`

The string value `authorization_code`.

`code`

The code that was just sent to us.

`client_id`

The same client ID we specified before. Really, we'll explain it shortly.

`redirect_uri`

This is the same redirect URL as before, too. This is just used to help prove that we know where this code came from—it won't try to redirect here or anything this round.

`client_secret`

This is a secret that we get from the Developers Console when we set things up. We'll cover it when we cover the client ID. Any day now.

What we'll get back in the HTTPS reply is a JSON object with our access token and some other information. Most of the OAuth libraries treat the object returned as a single creature—you shouldn't try to break it up or store different parts of it separately. If you're going to store it (and you probably will), we suggest you store it as a single unit, as well.

The object contains several fields:

`token_type`

This should be the string value `Bearer`.

`access_token`

This is the important bit. We'll be providing this token along with all our calls to Google's API.

`refresh_token`

A token that can be used in the future to get new access tokens. This token never expires (although it may be revoked), so guard it carefully!

`expires_in`

The number of seconds that `access_token` is valid for. This is generally about an hour, but you shouldn't count on it. The token may actually become invalid before this time, particularly if it was revoked. We'll talk about this a bit more... you guessed it...shortly.

There may be other fields that are returned, so you shouldn't rely on just these four and no others.

Step 4: Use (and refresh) the access token

We now have completed the major steps to our dance and we have an access token! Time to celebrate!

At this point, we can probably start doing things with the user's Glass—like sending a welcome message, listening to updates from Glass, registering contacts, and sending out updates if we want. Users should be able to configure your service, and this would be the time to do that. When all that is done, you can close the window, thus completing the initial login phase.

The access token that we have is our key to doing all of this. We need to include it as part of the HTTPS Authorization header, which might look something like this:

Authorization: Bearer 1234567890

You won't see us include this in further documentation, however, for two good reasons. The first is that we'll assume you actually read and remembered this paragraph. The second is that you'll be issuing most commands through a library, and the library will take care of adding the authorization information for you.

After an hour, our access token won't be good anymore. Trying to use it will generate an "Unauthorized" error. In order to get a new access token and keep working, we'll need to do another POST to the same URL (<https://accounts.google.com/o/oauth2/auth> just so we remember) with slightly different parameters again:

`grant_type`

The string value `refresh_token`.

`client_id`

The same client ID we've specified every time so far.

`client_secret`

The same client secret as before.

`refresh_token`

The refresh token we saved earlier. Clever, yes?

We'll get back a JSON object that contains the new access token and most of the same things as last time. We won't get a new refresh token, however—the one we just used will continue to be good.

Rather than trying to time the expiration of the access token exactly, a common scheme that the libraries tend to use is to execute a call with the access token. If it fails due to authorization problems, they get a new access token with the refresh token and try again. If it still fails... well...you're just out of luck. We suggest you make use of the libraries when you can, and use this same trick when you can't.

You may notice that in everything that we've sent, and everything we've received, there is one thing that is quite conspicuously missing. We have absolutely no idea who it is that we're talking to.

Who Are You? I Am the New Number Two

It may seem strange that OAuth doesn't provide us with any identifying information about the user, but this is somewhat deliberate. When it comes to OAuth, we need to specifically get permission from the user to be able to do anything, and that includes getting information about who they are.

So how do we get a user's identity? It depends on what we need, exactly. In many cases, we just need a unique identifier from them—some way to know what user configuration we should be using when the user returns to set up or change their settings.

In other cases, however, we may want to personalize the users' experience on our site a little more, and be able to send them nice personalized messages on Glass as well. We may want their name. It may be a good UX to present them with their profile photo to make sure they are using the right account. You may want to give them the opportunity to invite friends or share things with them. We may want access to their email addresses to send them notifications.

As we suggested earlier, and as you might have guessed, each of these options is controlled by a different scope.

What Do You Want? Information

You can think of a “scope” as corresponding to a set of permissions. We’re already familiar with the `glass.timeline` scope (which is expressed as a full URI), which we use to get permission to write to and read from the user’s Glass device. We’ve also gotten hints that we will need permissions to access some user information. All of Google’s services are governed by scopes of authority, and we need to consult the API documentation to find out what authorization scopes will be required for each operation.

Scoping Out the Entire World

You’re also able to bundle multiple scopes into the same project from different authorization servers. We mentioned earlier in [Chapter 6](#) about the expected surge in mashup Glassware, which will be based on wearable programs existing just to leverage the functionality of other APIs, using them as the backbone for their features. This is essentially what made Twitter such a worldwide hit. Winkfeed, for example, does this by authenticating a Glass user with its own permissions and then also using Pocket as a reading queue.

This isn’t necessarily as easy as it sounds, however. Often when writing Mirror API Glassware, you’ll need access to the timeline, location, and profile permissions, meaning you’ll need multiple scopes. So, Google requires your application to pass a string of three space-delimited URLs. Not all providers handle scope in exactly the same fashion in their authorization servers.

While the OAuth spec recommends multiple scopes to be case-sensitive and space-delimited, Facebook requires that multiple scope URLs be separated by commas. Instagram uses plus signs to delimit two or more scopes passed to its authorization servers. And LinkedIn’s implementation goes a different route—prefacing scopes with “r” for read access or “w” for write access, then an underscore, and then the scope URL, with all scopes separated by a space.

This knowledge may come in handy if your Glassware uses other web APIs, so when accessing a remote system, don’t assume they all behave the same, and consult their documentation.

Given this, it sounds awfully tempting to just request permission for everything we might possibly need. This is a bad idea for two reasons:

1. Users are increasingly aware and savvy of what permissions they’re granting you. Unless you give them a good reason why you’re asking for some, or so many, rights, they may just reject your service as being overly intrusive.

2. You can request the bare minimum now and request additional permissions later as you may need them.

The second point is important—we may start out by requesting just basic permission to Glass to get user information, and if we later add a feature that lets users send information to Google+, we can prompt them to turn this on in their settings and request permission at that time. This is known as *Incremental Authorization* and is a more advanced feature in Google’s OAuth implementation.

So what, exactly, is the bare minimum right now? We’ve talked about the `glass.timeline` scope. There is another Glass scope that we’ll be talking about later that requests location information, and if you’re going to use it, asking for it up front is best. Finally, as noted previously, we should probably get some information about the user’s identity. And here we run into a small pile of options:

profile

This is the most basic identity scope you can request. It gives you access to the UserID and some other profile information through the Google+ API.

<https://www.googleapis.com/auth/plus.me>

Similar to the *profile* scope, the biggest difference is that some profile information (such as the user’s name) won’t be available if they haven’t upgraded their account to Google+.

<https://www.googleapis.com/auth/plus.login>

A much more advanced scope, this gives you the same read access as *profile*, but also lets you see some social information (if users have permitted it to you explicitly), manage app activities through Google+ (an interesting subject, but way out of the “scope” for this book, so to speak), and shows your Glassware on the Google+ app list. Unlike the *plus.me* scope, if your user hasn’t upgraded her account to Google+, she’ll be unable to continue unless she does so.

email

An additional scope that you’ll likely request along with one of the previous scopes, this provides you with the email address for this account. Be careful about requesting or using this scope—not every user will want you to know their email address, and this is not necessarily a good address to send them email.

<https://www.googleapis.com/auth/plus.profile.emails.read>

Similar to the *email* scope, this is one that you will request with another profile scope. In addition to the email address provided by the email scope, however, it also lets you access all the other public and verified email addresses attached to this profile.

<https://www.googleapis.com/auth/userinfo.profile>

You may see documentation referencing this scope as a good way to get user information, but Google has deprecated this scope and stopped supporting it in September 2014. This provided access to the same information as *profile*, but using a different API (the *UserInfo* API instead of the *Google+* API). Use *profile* and the *Google+* API instead of this scope.

<https://www.googleapis.com/auth/userinfo.email>

Similar to *userinfo.profile*, this is an older scope that provides the same information as email through a different API. It has also been deprecated and was removed in September 2014.

So once we request the right scope, we're just given all this information, right? Well, no...not exactly. All requesting the scope does is give us permission to get the user's info. We still have to actually request that information. For all the recommended scopes, you should be using the *Google+* API to do so. The nitty-gritty of the API is outside of the scope of this book, but for a primer on how to use it, we suggest our good friend Jenny Murphy's excellent work on the subject, *Developing with Google+*, as well as Google's full documentation online at <https://developers.google.com/+/api/>.

Since this really is an important bit to have, however, we're going to take a quick look at getting the info by checking out the *people.get* method. Consider it a warmup for working with Glass.

The gist of using the *plus.people.get* API is that we're going to send a GET request via HTTPS, and the JSON object that is returned will contain the information we need. This particular URL is in the form <https://www.googleapis.com/plus/v1/people/userid> where *userid* is the user ID of the person we want to get information about. "But wait!" we hear you calling, "We don't know their user ID! That's the whole problem!" Fear not. The special user ID of "me" can be used to get the information (including the user ID) of the person we're acting on behalf of. And since we have their auth token, that's exactly what we're doing.

So if we have requested the correct scopes and make a call to <https://www.googleapis.com/plus/v1/people/me> we'll get back a JSON object containing a representation of a person. This includes fields for their user ID, name, possibly a profile photo, and other information. We need to be careful with this information, however—users may choose to change or delete it, so we can't assume it is correct indefinitely.

Google's Terms of Service control how long we can retain the information, so we might want to go back and refresh the info periodically. We might be shocked, however, to discover that we're no longer permitted access to it. What happened? The user might have cut us off and revoked our permission.

Disabling (and Reacquiring) Permissions

How dare they! How can they do that? Removing permissions... don't they realize we're doing this for their own good? They may, but the sad fact is that we have to remember that not all Glassware works for everyone. That's OK.

Most users will use the MyGlass app or page to disable our access. The same switch that was used to turn on our Glassware can be used to turn it off. When this happens, two major things happen:

1. The access and refresh tokens are revoked. Any attempt to use them will get an “Unauthorized” error.
2. The subscriptions and contacts that we've registered on their Glass (covered in [Chapter 10](#)) will be removed.

One thing that *doesn't* happen, however, is any sort of notification to our Glassware. We have no opportunity to send a final farewell message and clean the place up after they've gone. All we will know is that when we try to do something in the future, it will fail. We'll talk about some ways to handle this issue later in [Chapter 9](#).

Never lose hope, however. Just as quickly as they turned off your Glassware, they may turn it back on!

Who Is Number One? You Are Number Six

Speaking of never losing hope. We bet you forgot all about that `client_id` and `client_secret`, haven't you? Well, we didn't! And now is finally the time to discuss it.

OAuth relies on being able to trust every transaction it makes and being able to clearly associate it to both a user and an application. It tracks that application through the `client_id` (you're a client to the OAuth server) and verifies you're the client you say you are through the `client_secret` (the equivalent of a password). We'll get those through the Developers Console, which we first learned about in [Chapter 7](#).

Let's return to the project we began setting up: Glass Playground. We'll open up [the Google Developers Console](#) and scroll down the project list until we see it. We'll then open up the APIs & auth tab on the left and then the Credentials item underneath it. We'll see a section for OAuth credentials, and may already have Compute Engine and App Engine credentials created for us. We can't use these, so we'll create a new one by clicking the Create New Client ID button.

A dialog box comes up, prompting us for some information. Most of the time, we will want an Application Type of Web application—users will be accessing us initially through a web page, even if we're going to be using offline mode for most of what they

do. The next two sections can be changed later, but we'll set them to some initial values for our use now...after we explain them a little bit.

The Authorized JavaScript origins field, as noted in [Figure 8-3](#), contains a list of protocols and hostnames where we might be redirecting *from*. So the hostname that they land on is back a few pages at Step 1. We need that protocol and hostname here. We can list more than one, but we have to be careful—the exact protocol (HTTP or HTTPS) should be listed, and the exact hostname should also be listed. So if people might be hitting either *https://www.example.com* and *https://example.com*, we need to list both here. We don't, however, need to list the rest of the URL—no paths are necessary.

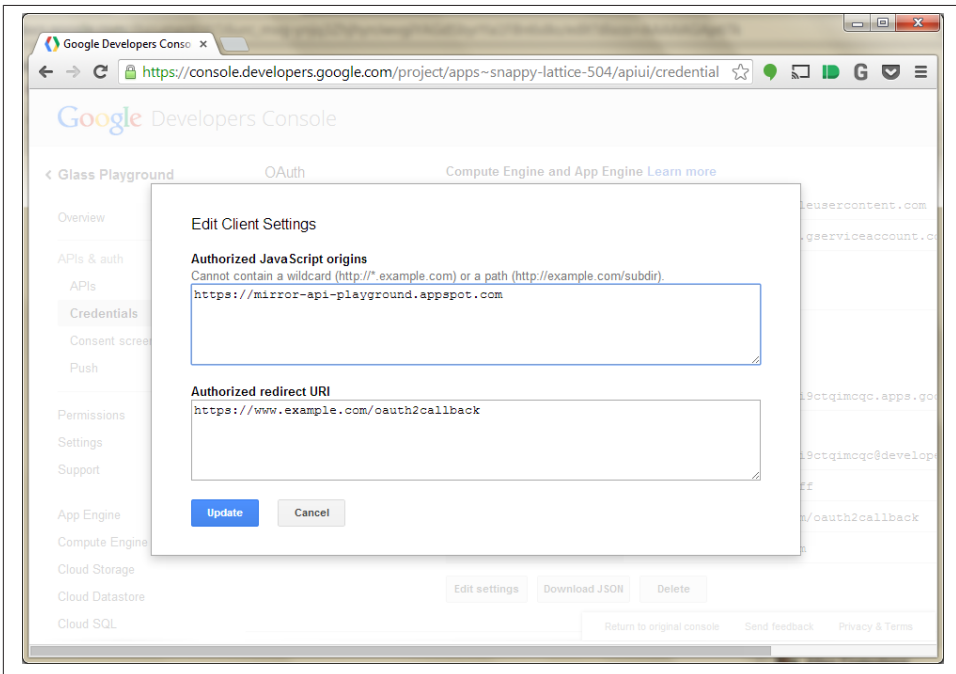


Figure 8-3. Assigning JavaScript origins and redirect URIs

For this playground, you should set this to *https://mirror-api-playground.appspot.com* (you'll find out why in the next chapter, we promise!) and delete the example.

Similarly, the Authorized Redirect URI field contains the pages on our site where we might have Google redirect the user *to* at the end of Step 2. We need to include one of these as the `redirect_uri` parameter in Step 1, and all of them must be valid HTTPS URLs. By default, it will pick a URL that corresponds to the origins in the previous field, although we can edit this if our URL paths are different. Go ahead and accept the default it provides.

Once we have all this set, we can click the Create Client ID button like [Figure 8-4](#) demonstrates. It will spin for a few moments, and you'll see a new section titled Client ID for web application. In the future, if you need to edit the Origin or Redirect fields, you can click the Edit settings button. Most important to us now are the Client ID and Client Secret fields. Why are they important to us? That would be telling. (We'll spill all soon.)

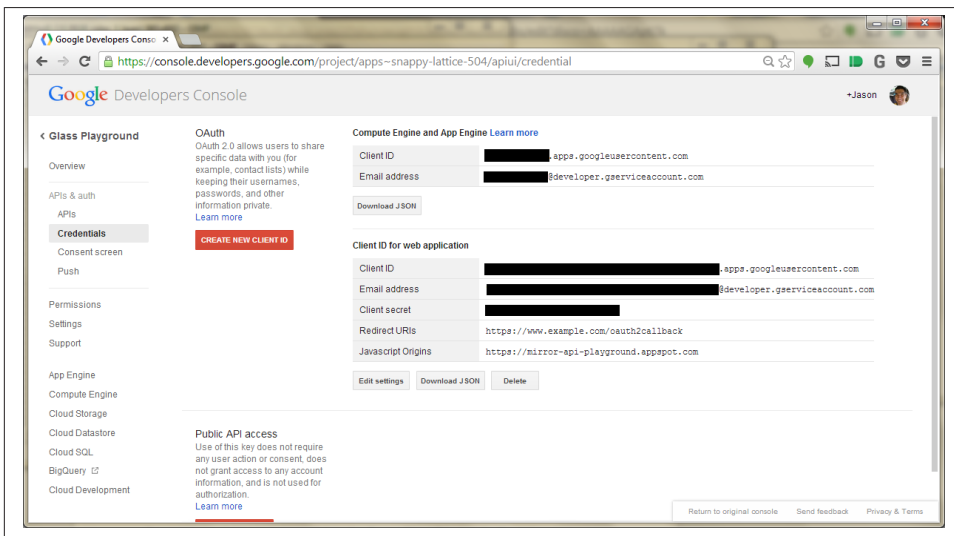


Figure 8-4. Obtaining Client ID information from Developers Console

WAKE UP!!!

We know the previous steps and configuration activities seemed long and boring, but it really was some essential groundwork before we can begin actually working with Glass. You need this critical information to allow your Glassware to communicate with Google, because again, your server will never directly talk to users, and your users will never talk directly to your server. As Jason likes to say, paraphrasing a fairly popular bumper sticker: *Know OAuth, know Glassware. No OAuth, no Glassware.*

But now that we understand card security, we have credentials for our playground, and we know what happens when we log in... we are (finally!) ready to do just that and start sending some data to Glass.

Working with Timeline Cards

The time has finally come. We've talked about effective Glassware design that properly conforms to the Five Noble Truths outlined in [Chapter 5](#), we've created a new Glassware project in Developers Console in [Chapter 7](#), and we've obtained the necessary OAuth credentials in [Chapter 8](#). Now, let's focus specifically on *presentation*. We have created a Glassware project. We have the client ID and secret. There's only one thing left to do—actually create a card and send it to Glass. You may be surprised that building a static card can be as simple as the concept of a card itself, but contained in this incredibly simple concept are many settings that give you lots of power over what you're sending to a user and how it will look.

We're going to start simple, learning some basic tools for designing the cards and getting a few onto Glass itself. We'll then start exploring some of the more advanced formatting features and options that are available as we generate cards. The next chapter continues talking about cards as we learn how users can interact with these compact creations of ours and, almost as importantly, how we find out that they've done so.

Just a reminder (we warned you), you're not going to see a lot of code in here. We're going to point you to Google's documentation for the actual code itself while we help make sure you understand the concept. We already know you're an awesome programmer—here's where you prove it.

“Hello, World!”, Glass-Style

Let's dive in! We'll be doing much of our work for this section using a tool from Google, the *Google Mirror API Playground*. This lets us create some cards for the timeline and send them to Glass so we can play with them. We can't do everything using the Playground, but this will help you learn the basic components of how to create a card. When you move to developing your Glassware, you'll return many times to the Playground to test out new card layouts, so this is a great place to start.

You can get to it from [the developer pages](#) by following the Tools section and then the Playground. (You can also [go there directly](#), but it helps to learn to navigate your way around the documentation.) The first thing you're greeted with is a prompt for your Client ID. Good thing we spent the past two chapters setting up a project and creating the OAuth Client ID! Go ahead and enter it here and click the Authorize button. You'll go through the OAuth flow to approve access to the timeline, and then you should be all set to use it.

The Playground is separated into several areas. Along the bottom are some sample templates, which we will be working with and adapting. Another tab gives us access to cards that we have already added to the timeline, so we can fiddle with them and update them as appropriate. Above the templates is a larger preview area on the left and the entry area on the right. We'll be able to enter data either using text, HTML, or JSON, and we can switch between the two (which comes in handy). [Figure 9-1](#) shows the main interface for the Playground.

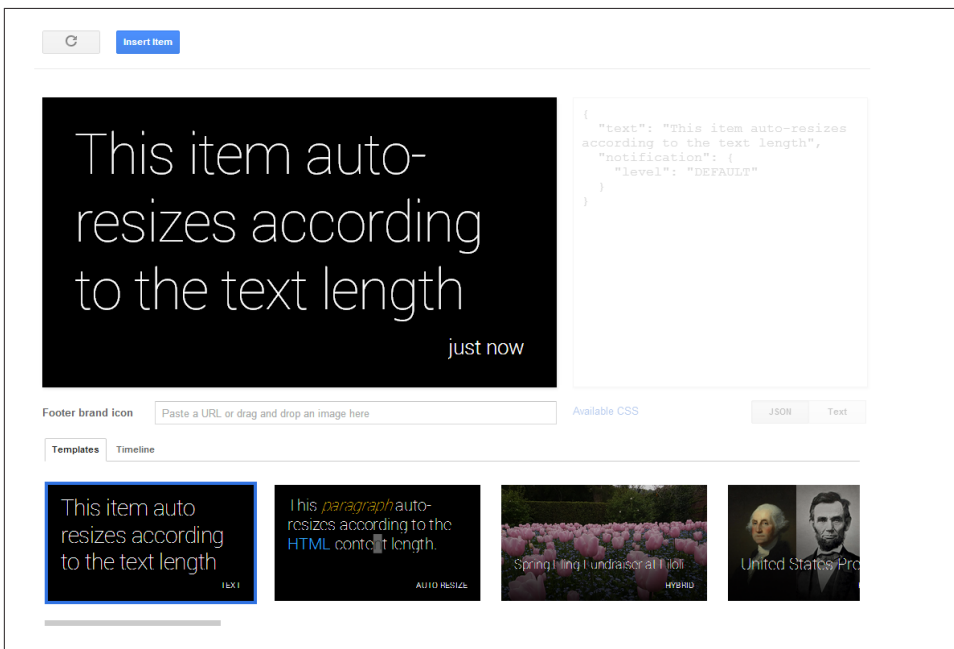


Figure 9-1. Testing prototypes in the Google Mirror API Playground

Poke around a little if you wish, but when you're ready, select the Text template, which is the first one all the way to the left. Then select the Text button beneath the entry area and replace the text by typing:

Hello Glass

As you're entering the text, you'll see the preview area in [Figure 9-2](#) change to reflect what you've typed—this holds true for both text and HTML. You can also click the JSON button to see the representation that will be sent to Google. We'll get to this in a moment.



Figure 9-2. A basic vanilla card

Once it looks good, give it a try on Glass—click the blue Insert Item button above the preview area. Several things should happen: you'll get an audible signal on Glass that you have a new card, and the Template area switches to Timeline view to show you what cards this app has in its timeline. If you were looking, you'd even have seen that the JSON entry area was updated to show you the response from the server with its representation of the item.

You just wrote your first Glassware...really!

You can even update this card. This time, click the JSON button and see what the JSON representation looks like. In the middle of everything, you will see a line that looks like:

```
"text": "Hello Glass",
```

Let's change this so we're a little more verbose, so it looks like:

```
"text": "Hello there, from Glass!",
```

...and then click the blue Update Item button, shown in [Figure 9-3](#). You'll see the Preview and Text Entry areas update as you go, and once you click the button, you'll see the Timeline area at the bottom changes, and your Glass will chirp alerting you that the previous card has been updated.

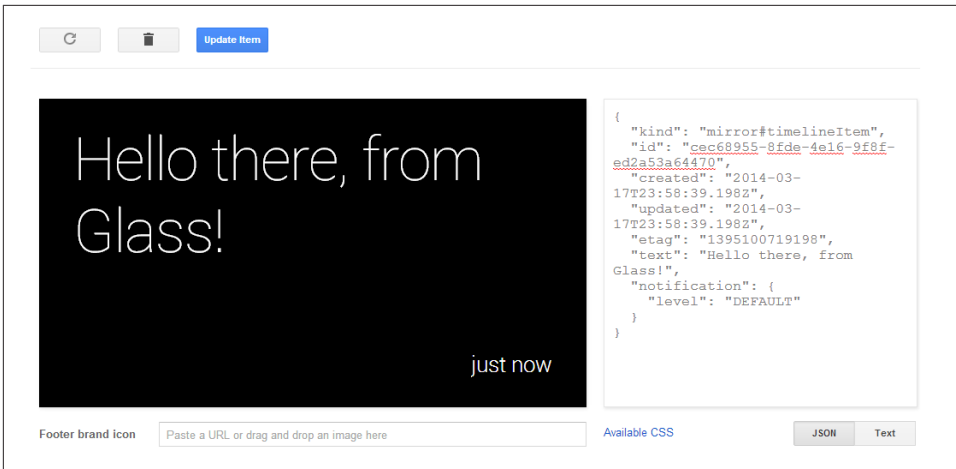


Figure 9-3. Updating existing items

Play with the text a bit. Try hitting Return so you force a new line in the display and see how it is represented in the JSON data, or try some very long text and watch how the text resizes itself dynamically. See if you can figure out how many lines it will eventually condense to. Wonder if you can edit the text on the preview window? (*Hint: yes!*)

When you're done playing, let's take a quick look at the JSON representation and learn a bit about these fields. To do so, we're going to consult the [documentation for a Timeline object](#). Go ahead and bookmark this URL—we'll be visiting it frequently. We use this object to send values to Glass, and the object that is returned is a fully populated version with more information. We'll cover most of these items through this and subsequent chapters, but some won't get much coverage—we've found them not so useful. Feel free to read and experiment with them here.

So we sent an object looking something like so:

```
{
  "text": "Hello Glass",
  "notification": {
    "level": "DEFAULT"
  }
}
```

and got back a fully populated object something like this:

```
{
  "kind": "mirror#timelineItem",
  "id": "cec68955-8fde-4e16-9f8f-ed2a53a64470",
  "created": "2014-03-17T23:58:39.198Z",
  "updated": "2014-03-17T23:58:39.198Z",
  "etag": "1395100719198",
  "text": "Hello there, from Glass!",
```

```
"notification": {  
  "level": "DEFAULT"  
}  
}
```

So what do these fields mean? Let's consult the table of property names and descriptions, which are arranged alphabetically. For the values we set when creating the card:

- *text*—Seems simple enough, this is the text that appears on the card. It is a string object and the notes indicate that the property is writeable. (Good thing, since we already wrote to it.)
- *notification*—This is a nested object that contains other properties, each of which is described in its own entry in the table. This is what tells Glass that it should chirp when the card is inserted or updated.
- *notification.level*—Another writeable property—the documentation as of the time of this writing says the only valid value right now is the string “DEFAULT”

Seems straightforward enough. What about the values we get after we do the insert?

kind

This will always be set to the string `mirror#timelineItem`. We see this sort of namespacing pattern for all objects that come from a Google REST API—we will never need to set this value explicitly, but this will always indicate the type of the object returned to us. Most of the time, our host languages will take care of this for us, but on occasion it may be useful for debugging.

id

A unique identifier, sometimes called a “UUID.” This card is ours. There may be others like it, but this is how we know this one is special. We’ll be using this ID later when we update or modify a card. Glass will always set this value for us—we can’t set it ourselves.

created and updated

These are datetime objects representing when the card was (you guessed it) created or last updated. These two times are initially the same, because the card was last updated when it was created. If you make changes, you’ll see that the updated time will change, but the created time (as well as the ID) won’t. We can’t set either of these values, but we’ll see in a moment how we can control the time on the card. The format of a datetime object is fairly simple. While it is probably easy to figure out from the example, you’ll see it is always represented with a four-digit year, a two-digit month, day, hour, minute, and second, followed by three digits containing the thousandths of a second. Fields are separated by hyphens, with a “T” separating the date portion and the time portion. It will always represent the UTC time, and we know this by the “Z” at the end. Your language’s library will probably have

converted this to whatever format dates and times are handled in your particular programming language.

etag

The details of what an ETag are can be complex, but you can think of this as a revision indicator for the ID. So while the ID represents a card, no matter what may have changed on the card, the ETag will let you know if the information on the card has changed in some way. It doesn't tell you what has changed or why—just that something has changed. This will come in useful later, when we read the timeline after we're told about updates, but we'll ignore it for the moment.

The notes of the documentation indicate which fields are writeable by us. No notation indicates that Glass will set it for us, and we might not even be able to set these fields using our language's library.

A Method to Our Madness

We've mentioned concepts of "inserting" or "updating" timeline items without really explaining how you'll do this in your own code. We're going to stick with the Playground for a while since it makes it easier to understand the concept before you get into syntax, but this is a good place to point out just what commands are, conceptually, possible. Under the list of object properties in the documentation you'll see a list of methods available with the Timeline object. You can click each if you really want to explore them now, but we suggest you hold off and focus on designing cards with the Playground first.

So we've welcomed ourselves to the wonderful world of Glass development. We're all set to code now, right? Well...the text looks OK, and may be useful for some basic things, but you'll notice that the good Glassware that we've shown you so far isn't just a wall of text—there are some clear visual elements that help us quickly grasp what the card is about. To do this, we're going to rely on many of the other templates in the Playground, and these are all based on HTML.

HTML: Even More Style

Let's take a look at what other templates are available. To our utter shock and amazement (not), we notice that many of the other examples look similar to much of the Glassware we've seen over the past few chapters. These design patterns come up over and over *because they work*. (And because they were in the Playground, and the Glassware developers used the Playground to help come up with their own cards.) Even with the similar design, there is plenty of room for creativity.

All of these layouts use HTML markup to provide structure and semantics for the cards, and supplement this structure with a little bit of CSS styling rules to provide visual

clues. If you're familiar with both, you may need to do some unlearning—many tags don't work, or don't work as well as you'd like, others should be avoided because they don't work as well on Glass, and other features we normally associate with HTML and CSS (namely JavaScript and its associated dynamic functions) just don't exist at all.



Your Own Style

If you're HTML and CSS savvy already, you may be tempted to skip this section and read the list of styles available through the “Available CSS” link on the Playground. We suggest you go through the chapter first and then supplement your knowledge with the details available there. (And beware—some classes aren't in the CSS rules.)

You can, of course, use CSS to add your own styling where appropriate (using your own classes or via a “style” attribute), but we strongly recommend that you don't go overboard—the style rules in Glass are there to help create uniform behavior. You can also use other HTML besides those that we'll be talking about here, but Glass doesn't let you do everything you may expect. Check out the documentation of the Timeline object, and the `html` property specifically, for the list of what tags are accepted, what tags are removed with contents preserved, and what blocks are removed completely.

If you're not familiar with HTML and CSS—don't worry. We're not going to teach you everything, but we'll cover enough for you to understand what you need to know.

We'll start by selecting the Auto Resize template and click the HTML button to see what the settings look like:

```
<article>
  <section>
    <p class="text-auto-size">
      ...
    </p>
  </section>
</article>
```



JavaScript's There... Just Not for Us

The secret about Glass is that while JavaScript code isn't supported for developers, it is running under the covers. If you read through [the base styles for CSS on Glass](#), you might notice in the comments notes about how the size of some text selectors is automatically resized by JavaScript. Sneaky, Google!

Just looking at the framework, we see a structure of nested blocks defined by tags. The outermost `<article>` tag block, a `<section>` block inside this, and a `<p>` block inside

this. The `<p>` block (for paragraph) has a class attribute associated with it that provides additional information about how to handle the contents (we'll document this as `p.text-auto-size` in the HTML code block that precedes [Figure 9-4](#), meaning a paragraph block that has `text-auto-size` set as its "class" attribute). What do these tags and classes mean?

article

Defines a timeline card visually. There are exceptions (and we'll cover them in a bit), but you can consider that everything defined under the article will appear on a single screen in the timeline. This contains all the other components of the card, which we'll be going into.

section

This is the main body of our card. You can think of this as where we're going to put the important part of our message.

p.text-auto-size

A single paragraph of our message that will auto-resize based on how much content it has. We can have multiple paragraphs, and it will make sure each starts on a new line.

Try this bit of HTML out and you should get a card like [Figure 9-4](#):

```
<article>
  <section>
    <p class="text-auto-size">
      Hello there
    </p>
    <p class="text-auto-size">
      Explorer
    </p>
  </section>
</article>
```

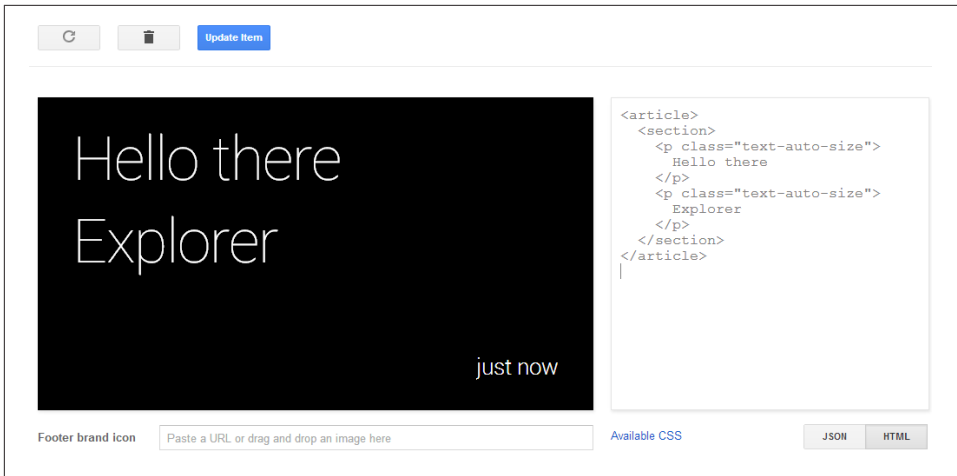



Figure 9-4. Working with content in the Mirror API Playground

Then play around a bit. What happens when you put lots more text in just one of the paragraph blocks? What happens if you put more in both? What happens if you put in more text than will fit? You should see that each resizes separately from the other, but you may see some oddities about when it resizes and how it looks when you do. *A note of caution: it may appear differently (and usually better) on Glass itself since the Playground doesn't perfectly emulate what Glass does.*

Auto-sizing is fine, but we usually want to be a little more in control of our layout by specifying the size. Change our preceding sample so we're specifying the size on the first paragraph and leaving the second paragraph at whatever default size Glass picks for us:

```
<article>
  <section>
    <p class="text-xx-large">
      Hello there
    </p>
    <p>
      Explorer
    </p>
  </section>
</article>
```

The `text-xx-large` class is good for just a couple of words (Figure 9-5). If you play with it, you'll see that it really dominates the screen. Glass defines a few sizes for you—play around with each and see how they work:

- `text-xx-large`
- `text-x-large`

- text-large
- text-normal
- text-small
- text-x-small



Figure 9-5. Applying text-xx-large

You may wonder which is best—using a fixed-size font or auto-sizing. Fixed sizes are more predictable and present fewer formatting surprises if a user is glancing at the screen. On the other hand, short text can certainly be read easier in a larger font, so it may be useful to see a short message quickly, while making it easier to determine you don't want to read a longer message. See what is best for your specific application.

Different sized text is great, but what about bold, italics, and colors? You're covered. Take this chunk of HTML for a spin, which should produce a card not unlike [Figure 9-6](#):

```
<article>
  <section>
    <p>
      You can do <b>bold</b> (or <strong>strong</strong>)
      and <i>italics</i> (or <em>emphasized</em>)
    </p>
    <p>
      You can even do colors like <span class="red">red</span>
    </p>
  </section>
</article>
```

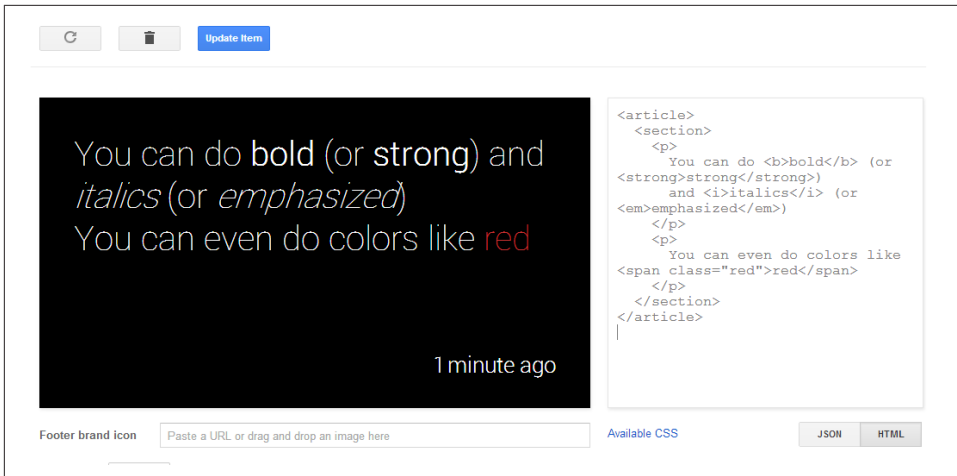


Figure 9-6. Formatting options

Simple text formatting is pretty straightforward, particularly if you've used HTML before. HTML folks call them inline blocks, and we have the following ones available:

b or strong

Bold.

i or em

Italics.

`span`

Inline element without formatting, but can be used to contain the color formatting classes without creating new block-level content like with the `<p>` and `<div>` tags.

You can apply these color classes to any of the inline blocks (see the documentation about color for the specific hues applied):

- red
- yellow
- green
- blue
- white
- gray
- muted



Mix Colors Instead of Font Sizes

One of the things that will drive you batty as a designer on any platform is changing the size of fonts to infer importance, precedence, or immediacy within your content. On Glass, you'll inevitably run into this because of the need to draw attention to text while making sure it stays visible within the available card space. You may want to highlight a headline so that it stands apart from other text, but you want to make sure it doesn't push other content away.

A cool trick is to use a set font size for all your textual content, while changing up your use of color. Make prima donna text stand out with bright colors like yellows and greens, while demoting other text with grays and the appropriately named muted class. Since Glass is a glanceable, short attention span medium, you'll preserve the screen real estate while establishing a hierarchy for your stuff.

A lot of the HTML that you're already familiar with should just work, although some of the tricks you're used to won't. Lists, for example, display fine, but are formatted differently:

```
<article>
  <section>
    <ul class="text-x-small">
      <li>Vanilla ice cream with nuts and chocolate syrup</li>
      <li>Chocolate ice cream with peanut butter topping</li>
      <li>Tutti-Frutti on a bed of bananas</li>
    </ul>
  </section>
  <footer>
    <p>Available Sundaes</p>
  </footer>
</article>
```

The result should appear like [Figure 9-7](#).

You'll notice that instead of bullets to the left of each item, there are subtle separator lines in between each list. Each item is confined to a single line by default, and nested lists are discouraged since font size isn't enough of a distinguisher. We can get rid of the separator line by adding the `no-border` class, but we strongly suggest you don't use it—the separator helps break up the items in a subtle way:

```
<article>
  <section>
    <ul class="text-x-small no-border">
      <li>Vanilla ice cream with nuts and chocolate syrup</li>
      <li>Chocolate ice cream with peanut butter topping</li>
      <li>Tutti-Frutti on a bed of bananas</li>
    </ul>
  </section>
```

```

<footer>
  <p>Available Sundaes</p>
</footer>
</article>

```

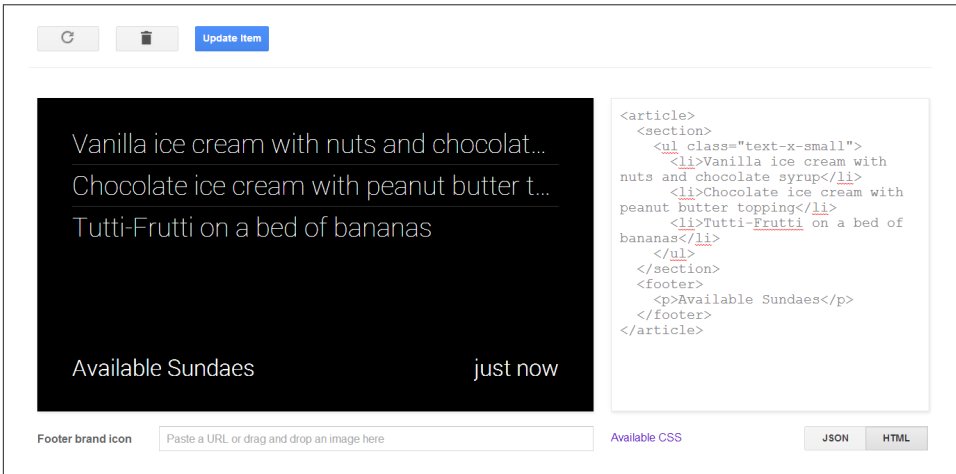


Figure 9-7. Working with lists

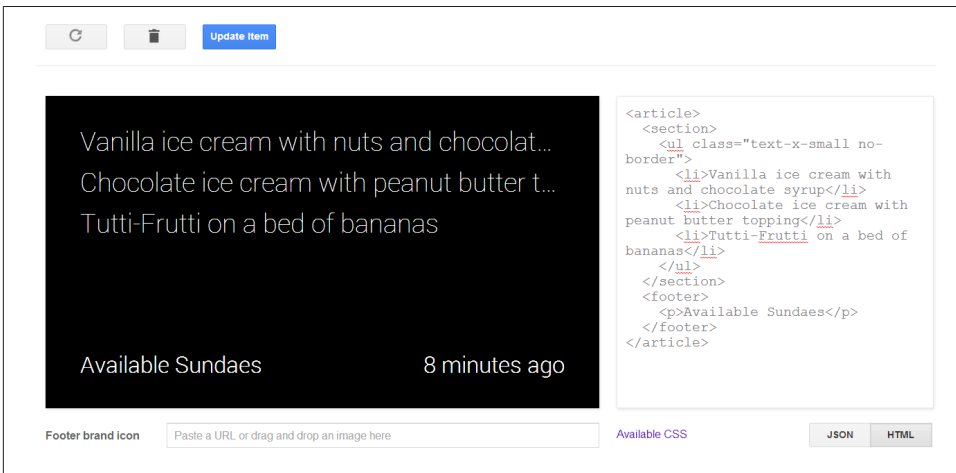


Figure 9-8. Working with tabular data

Tables also work well, and you have access to the `no-border` class here if you need it (Figure 9-8). The `align-justify` class does a little fancy work with tables—leftmost cells are left-justified, rightmost are right-justified, and all the rest are centered:

```

<article>
  <section>
    <table class="align-justify">
      <tbody>
        <tr>
          <td>email</td>
          <td>---</td>
          <td class="red">down</td>
        </tr>
        <tr>
          <td>web</td>
          <td>2ms</td>
          <td class="green">OK</td>
        </tr>
        <tr>
          <td>db</td>
          <td>20ms</td>
          <td class="yellow">slow</td>
        </tr>
      </tbody>
    </table>
  </section>
  <footer>
    Server Status
  </footer>
</article>

```

Figure 9-9 shows the output.

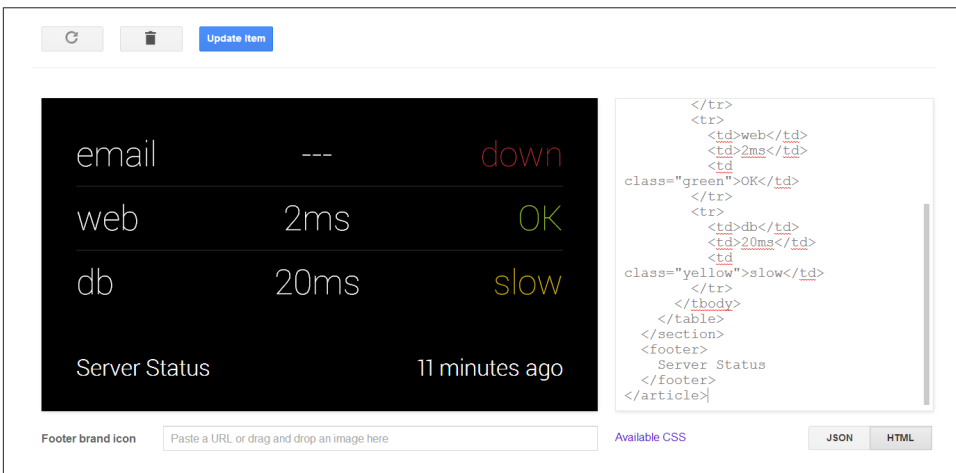


Figure 9-9. Mixing formatting in a card layout

We should caution you, however: *don't try to use tables to do absolute positioning for columns*. That way lies pain. There are, however, classes that you can apply to a `<div>`

inside a section, and nested `<div>` elements will be broken up into columns, as shown in [Figure 9-10](#):

```
<article>
  <section>
    <div class="layout-three-column">
      <div class="align-center text-x-small">
        <p>Mama</p>
        <p>too small</p>
      </div>
      <div class="align-center text-large">
        <p>Papa</p>
        <p>too big</p>
      </div>
      <div class="align-center">
        <p>Baby</p>
        <p>just right</p>
      </div>
    </div>
  </section>
  <footer>
    <p>Bears</p>
  </footer>
</article>
```

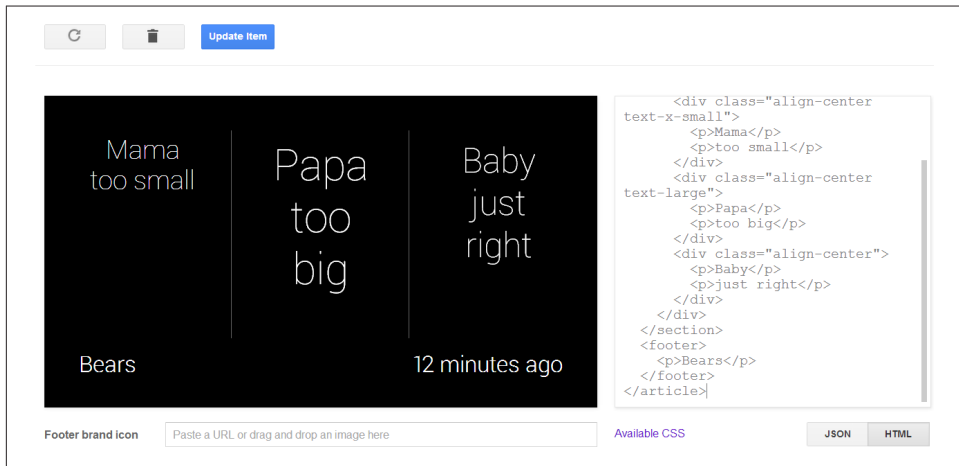


Figure 9-10. Applying the built-in CSS classes

In this illustration, we see `layout-three-column` doing just that. Experiment a little and see what happens when we use these other layout classes:

`div.layout-two-column`

Provides two equal columns.

`div.layout-three-column`

Provides three equal columns (clever, we know).

`div.layout-figure`

Provides two columns where the leftmost column is slightly narrower than the rightmost. This will be the same width as the `<figure>` block we describe later.

If you play around with HTML a lot, you'll discover what works and what doesn't. You'll probably be surprised at how much of the basic markup continues to do the right thing. Even things like horizontal rules, superscript, and subscript work correctly (Figure 9-11):

```
<article>
  <section>
    <p>1<sup>st</sup></p>
    <p>2<sup>nd</sup></p>
    <hr/>
    <p>A<sub>n</sub> = B<sub>n-1</sub> + C<sub>n-2</sub></p>
  </section>
</article>
```

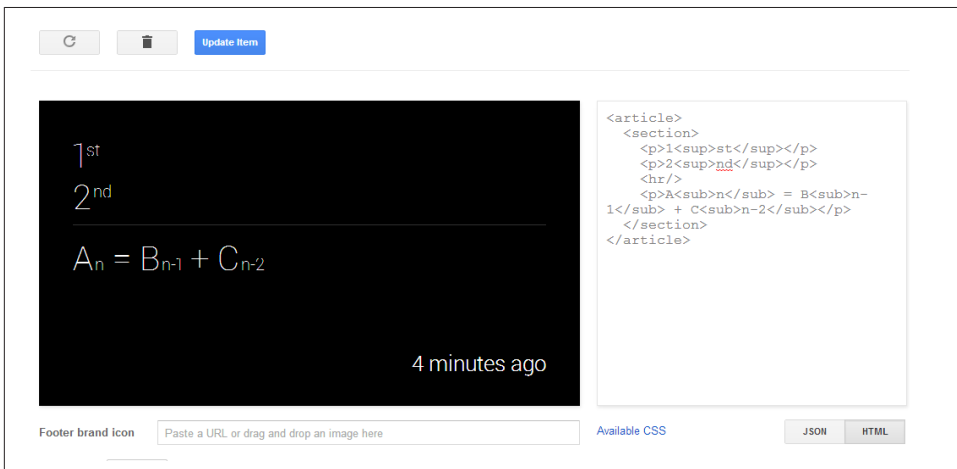


Figure 9-11. Formatting for scientific notation

Glass introduces some convenience classes for text as well. For example, this illustrates how you can easily align text (Figure 9-12):

```
<article>
  <section>
    <p class="align-right text-x-small">Right</p>
    <p class="align-left text-x-small">Left</p>
    <p class="align-center text-x-small">Center</p>
    <p class="align-justify text-small">
```



```

    Or make it so that both margins line up perfectly.
    But try to justify that action.
  </p>
</section>
</article>

```

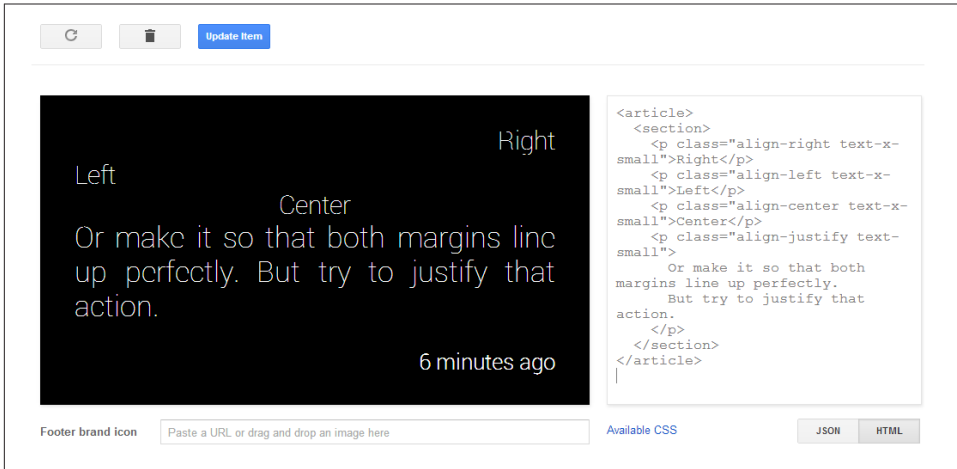


Figure 9-12. Alignment for text

Up until now, we've always had a `<section>` as the only thing directly inside an `<article>` block. You may have been wondering why, if we were always going to have the main content present, we don't just include it right under the article. The answer is that there are other components, besides section, and each gets its own block. One of the most significant ones is the footer block.

Every card automatically has a footer, and the timestamp is always on the right side of the card. Glassware that is distributed through MyGlass can also get its icon next to the timestamp. (And we can simulate this through the Playground by providing a URL for the Footer Brand Icon.) But we can also specify footer content that will appear on the left side of the card:

```

<article>
  <section>
    <p>
      The main text is here.
    </p>
  </section>
  <footer>
    <p>
      The footer is down here
    </p>
  </footer>
</article>

```

This renders the card in [Figure 9-13](#).

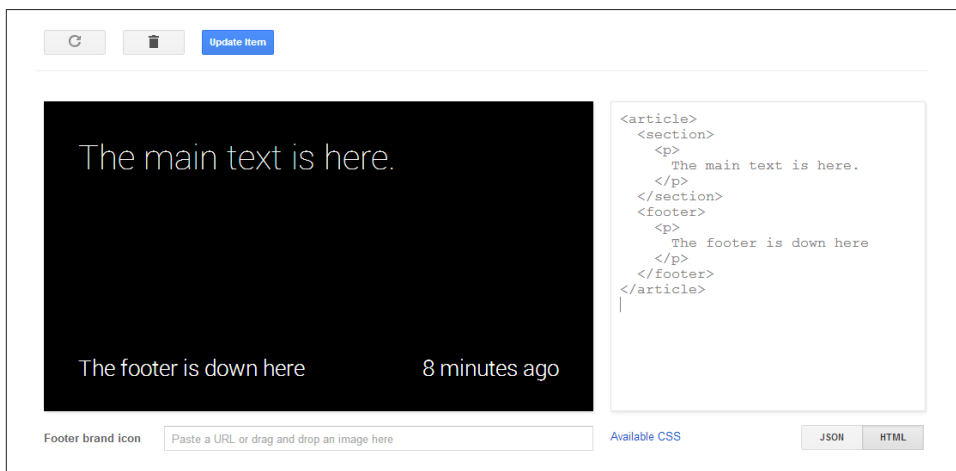


Figure 9-13. Working with the footer

This kind of footer is useful to provide additional secondary context for the main body of your text—perhaps to indicate a filename or additional source of the information. Consider, for example, the Sports template in the Playground, which uses the footer to give actual context to the timestamp it shares a line with.

If your Glassware is getting the information from another location, it makes perfect sense to use the footer to indicate that source. For example, we saw how Winkfeed uses this to indicate which RSS feed a card comes from. We shouldn't use the footer to simply specify the name of our Glassware, however, since this is amply provided by the icon on the right and doesn't provide additional information. If your footer always contains the exact same text—it might not be appropriate as a footer.

What About Images?

We have four primary ways we can use imagery in timeline items:

1. As icons accompanying text. We can see examples of these with the Flight and Transit templates.
2. With some special formatting in a header author block.
3. In a background photo such as demonstrated by the Hybrid template.
4. On the left side of the screen as a figure, illustrated by the Image List template.

We'll go over each one in some detail, since even the basic images are a little different than standard HTML.

Regular icons are pretty straightforward and use the `` tag, just like with standard HTML. You can even use height and width attributes to force an icon to be resized to meet your needs. One tool to help with this, however, is the `icon-small` class, which defines a 30 x 30 pixel graphic to be used with the `text-small` or `text-x-small` classes on a paragraph element. To illustrate, we can see this with the card in [Figure 9-14](#). See what happens if you remove the `img.icon-small` class on the selector or change the size of the text on the `<p>` tag:

```
<article>
  <section>
    <p class="text-small">
      
      Honest Abe
    </p>
  </section>
</article>
```

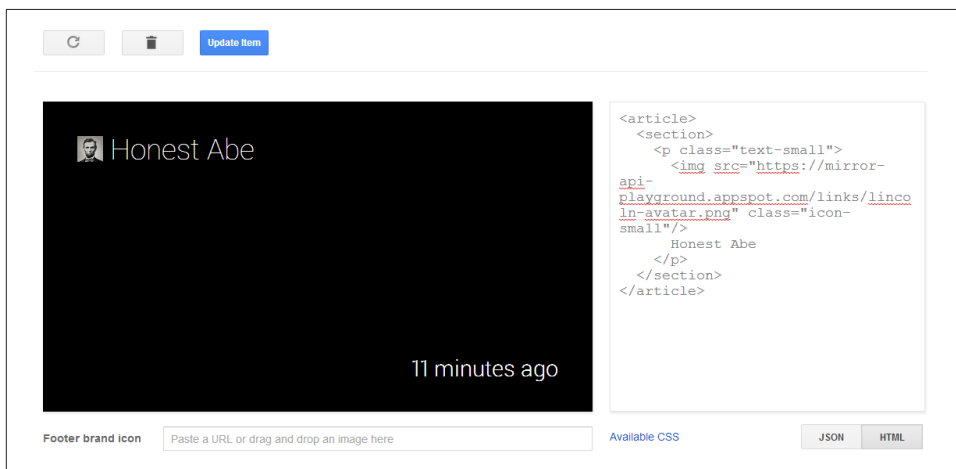


Figure 9-14. Working with small images

The header of a card combines a number of interesting formatting tags that are all enabled if the article has an author class associated with it. In these cases, the `<header>` block reserves space for a 70 x 70 pixel image (and resizes all images into that square) and two lines of fixed-size text. (There is room for a third line, but it starts crowding into the text, so we suggest avoiding it.) There is also some formatting applied to the `<h1>` and `<h2>` blocks inside the header block that force each to a single line:

```

<article class="author">
  <header>
    
    <h1>President Abe Lincoln</h1>
    <h2>Honest Abe</h2>
  </header>
  <section>
    <p>
      Four Score and Seven Years ago
    </p>
  </section>
  <footer>
    <p>
      Gettysburg Address
    </p>
  </footer>
</article>

```

Figure 9-15 is the pleasant UI that this HTML produces.

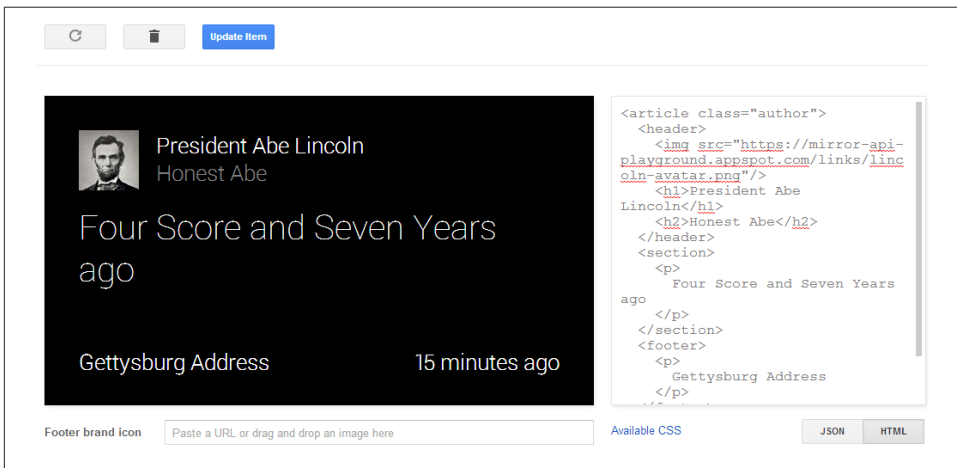


Figure 9-15. Working with medium-sized graphics

The One-Liner

The trick used by the `<h1>` and `<h2>` tags to keep everything on one line is a useful one if we want to make sure formatting doesn't get out of hand. We'll be revisiting it later when we talk about spreading text over multiple cards and bundles.

Background photos are a pretty common way to show a picture if there is very limited text or none at all. This works particularly well with a footer, although it can be used for any text, and there are some additional tricks we can use to make our text stand out even more against the image.

We're going to start with the Hybrid Mosaic template as our basis, only we'll start more basic and build up to it (Figure 9-16):

```
<article class="photo">
  <ul class="mosaic mosaic1">
    <li style="background-image:
      url(https://mirror-api-playground.appspot.com/links/lincoln.png)"></li>
  </ul>
  <footer>
    Honest Abe
  </footer>
</article>
```



Figure 9-16. Playing with mosaics

That doesn't look too bad, but there are a few things to notice. The first is that we now have an unordered list directly under the `<article>` tag. We have several new classes, and we are specifying more than one class for the `` tag (classes are separated by spaces). We'll summarize these new classes in a bit. More confusingly, however, is that we're introducing a new attribute to the `` tag—the `style` attribute, and we're setting it to a somewhat bizarre value. CSS pros glanced at it and moved on, and we're going to suggest the rest of you move on as well. You'll need to put the URL for the image inside the `url()` portion, but make sure you copy the rest verbatim.

Visually, we can see that the image looks cut off at the top and bottom. This is normal—a full bleed image tends to work best on Glass and is what most people will expect.

Trimming the picture instead of providing black bars or pillars makes everything else easier to read. Speaking of easier to read, the footer text gets washed out, too. Before we move on, you may want to experiment with what we get if we change the `<footer>` to a `<section>` and what happens if we remove the `article.photo` selector class. You should probably also play with pictures of different sizes and dimensions to see how they'll map. (We like <http://placekitten.com>, but you can pick any image you want to try out.)

The readability of the text, however, might be a bit of an issue. Fortunately, we have a few classes at our disposal that will darken the background a bit by adding a gradient to the footer. Here is our example from earlier with a bit of an overlay on the image:

```
<article class="photo">
  <ul class="mosaic mosaic1">
    <li style="background-image:
      url(https://mirror-api-playground.appspot.com/links/lincoln.png)"></li>
  </ul>
  <div class="overlay-gradient-short"/>
  <footer>
    Honest Abe
  </footer>
</article>
```

This may be enough for the footer, but if we have more text in the body we'll need more of the background shaded out. We have several classes available to us, ordered here from least obstructing to most obstructing. Unlike some of the previous classes we've seen, these classes must be applied to a `<div>` that is part of the `<article>` block and has no other content. Make sure you find the one that suits your needs best:

- `overlay-gradient-short`
- `overlay-gradient-medium`
- `overlay-gradient-tall`
- `overlay-gradient-tall-dark`
- `overlay-full`

Working with Mosaics

You've probably been wondering about that `mosaic1` class and if there are more `mosaic` classes and what they do. Very clever of you! There are eight in all, cleverly named `mosaic1` through `mosaic8`, giving you the ability to include up to eight recipients/participants/addresses/players/whatever in two rows of four. As **Figure 9-17** shows, the Hangouts Glassware does this to quickly indicate the members of a group chat, with the

last space in the lower right showing the other “rollover” members beyond what can be shown in the mosaic.

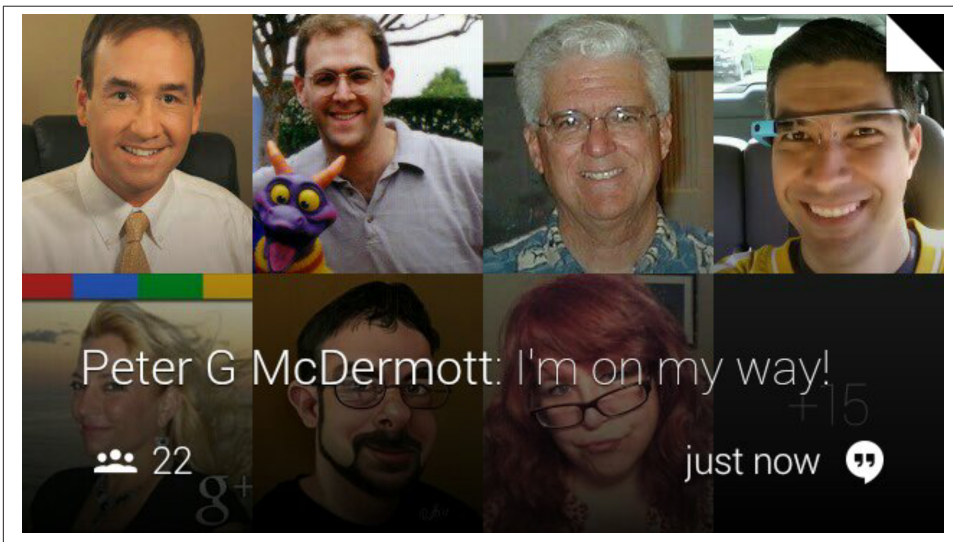


Figure 9-17. Mosaics as applied in Hangouts

They let you add additional images to the background and lay them out in a fairly logical way. Give this a try, and then see what happens if you replace the `mosaic2` with other values (including `mosaic1`) and what happens if there isn't a matching `` tag with the background set (Figure 9-18):

```
<article class="photo">
  <ul class="mosaic mosaic2">
    <li style="background-image:
      url(https://mirror-api-playground.appspot.com/links/washington.jpg)">
    </li>
    <li style="background-image:
      url(https://mirror-api-playground.appspot.com/links/lincoln.png)"></li>
  </ul>
  <div class="overlay-gradient-short"/>
  <footer>
    George and Abe
  </footer>
</article>
```

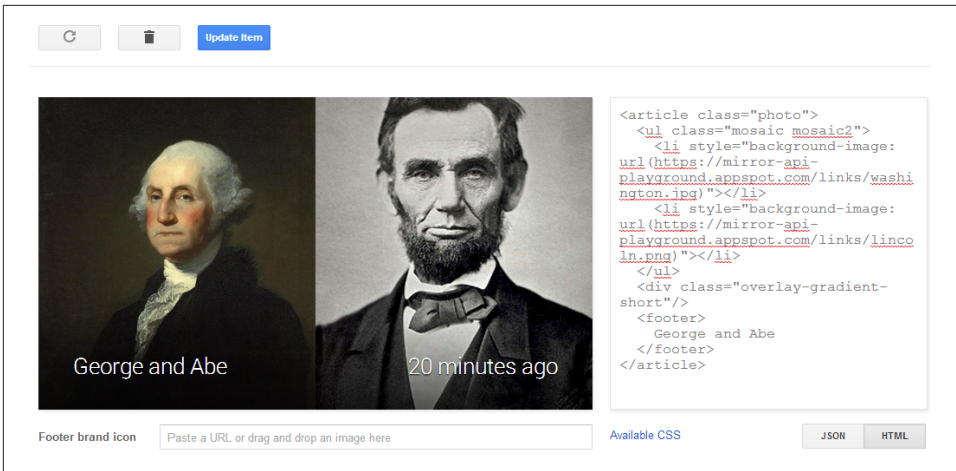


Figure 9-18. A more simple mosaic

Images on the left side of the card, done as a figure, are fairly similar. Consider this minor rework of the last example where we move the mosaic list into a `<figure>` tag and change the footer text into a normal section:

```
<article>
  <figure>
    <ul class="mosaic mosaic2">
      <li style="background-image:
url(https://mirror-api-playground.appspot.com/links/washington.jpg)">
    </li>
      <li style="background-image:
url(https://mirror-api-playground.appspot.com/links/lincoln.png)">
    </li>
    </ul>
  </figure>
  <section>
    George and Abe
  </section>
</article>
```




Glanceable Layouts and the Structure They Imply

A hallmark of good wearable design is giving users instant recognition about the composition of the information being presented to them—and again, in the microinteraction universe this means glanceability within fractions of a second. The layout, order, and size of text, icons, and imagery can instantly convey to the users what type of data they're dealing with.

In messaging applications like Hangouts and Gmail, a photo of a message's sender sits at the top of the mosaic of participants and is larger than images of other recipients, taking up half of the region. This visually relays a sense of structure and hierarchy. This is a good pattern to emulate with your own designs, and one you get for free in both the Mirror API (as a list) and the GDK (when repeatedly calling the `CardBuilder.addImage()` method).

Figure 9-19 is a UI that stacks images vertically on the left side of the card with room for text in a `<section>` element.

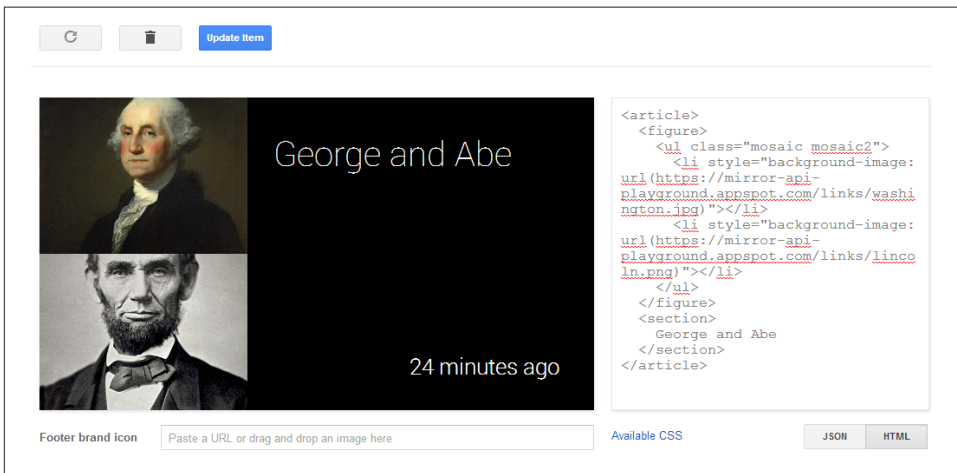


Figure 9-19. Stacking images

Who's on First?

When you're using either of the list-based image layouts, here's a good tip to keep in mind. The person the message is from is always listed first. Others are listed later, usually in order of participation. You can get the list of people addressed in a message from the `Timeline.recipients` property.

For some applications, it may not always be obvious who the “from” may be, but don’t be arbitrary about this convention. For instance, Gmail’s Glassware uses a generic avatar to denote contacts whose avatars aren’t set and also a “+15” structure not unlike the Hangouts example to show how many users in all a message was sent to. It’s great quick visual reference.

When is it best to use one over the other? As always, it depends on your exact needs, but a good rule of thumb is to use the figure layout when you’re representing participants in what is represented on the card, while the background image is best when the text is about that image itself. As we’ve seen, figures are typically used for things like Gmail while the background image is used for something like Field Trip.

Rendering an In-Card Map

Glass provides us with a special image URI, *glass://map*, which renders maps, markers, and paths using styles that work best for the heads-up display. You can set various parameters on the query string for this URI, assigning it as the `src` attribute for an `` tag, which will generate and update maps:

`w`

The map’s width in pixels (required).

`h`

The map’s height in pixels (required).

`center`

The comma-separated latitude/longitude coordinates the map uses as its base.

`zoom`

The magnification level for the map, between 0 and 21, matching the zoom levels you’ll find on Google Maps.

`marker`

The marker type—a “0” indicating a pin or a “1” indicating a start or current position—then a semicolon, followed by the comma-separated latitude and longitude. You can specify multiple marker parameters.

`polyline`

Parameters used to create a path overlay, consisting of a comma-separated width in pixels with a color, followed by a semicolon, and then a comma-delimited list of vertices of the latitude and longitude. You can specify multiple polyline parameters.

If you leave out the width and color details for a polyline, default values will be used. Also, if the parameters for a polyline are set but the center and zoom parameters are not, the map will automatically center and zoom itself to accommodate the drawn path.

As of the current release of the Mirror API, maps are rendered in the 2D map view, and cannot be set to satellite view or Earth view.

It won't show up correctly in the Google Mirror API Playground, but entering the following HTML in the Playground's editor—remember to place the map within `<article>` tags—produces some overly simplistic navigation from New York to Mountain View via Houston:

```
<article>
  <figure>
    
  </figure>
  <section>
    <p class="text-auto-size">New York to Mountain View</p>
    <p class="text-x-small muted">With a stopover in Houston!</p>
  </section>
</article>
```

Figures 9-20 through 9-22 show various ways to lay out data while displaying maps.

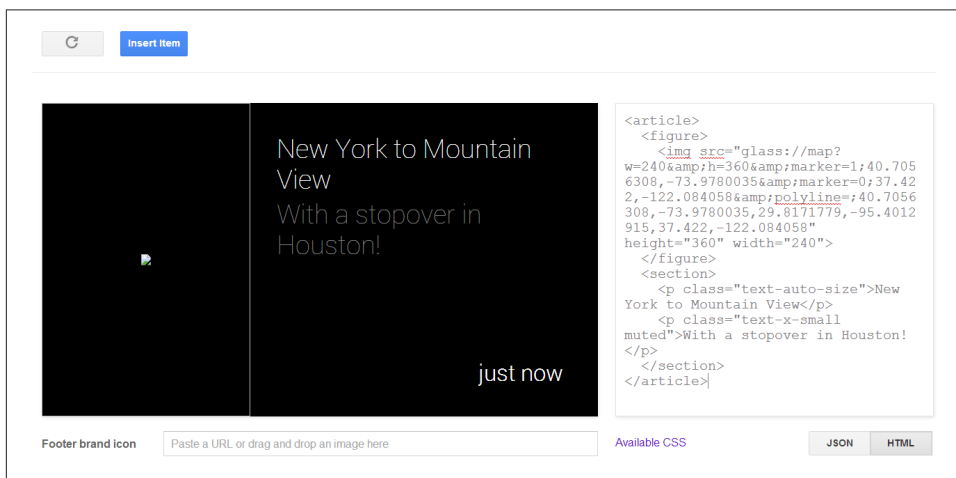


Figure 9-20. Maps don't render in the Playground

So let's see it live. Click the Insert Item button to send the card to Glass and render a nice cross-country trek.

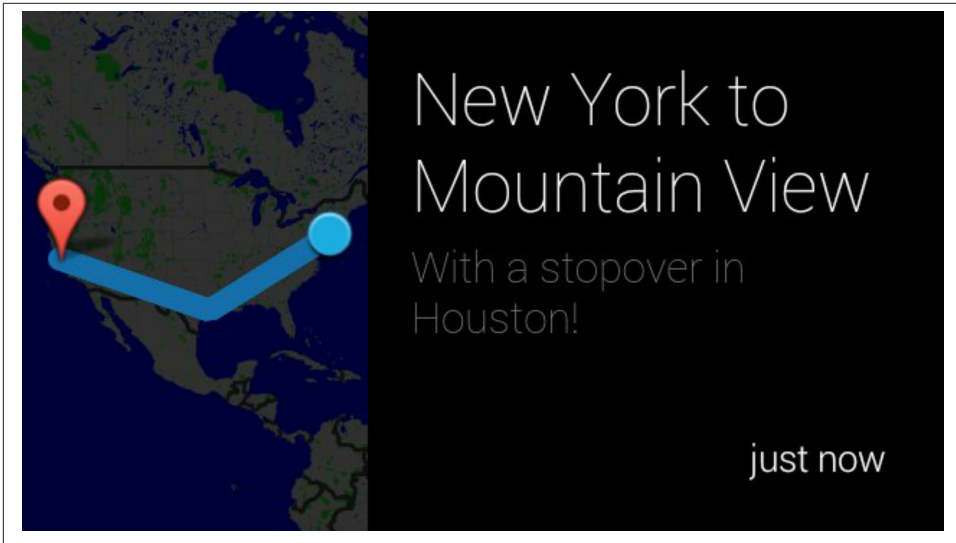


Figure 9-21. Our example when sent to Glass

We expect that your maps will want to be a bit more local. Possibly to use one as the background as the visual card when presenting things about tropical island paradises where one of your authors never has to worry about things like buying antifreeze or snow treads for his tires:

```
<article>
  
  <footer>
    Guam
  </footer>
</article>
```

Either way, you see that maps can be used the same was as in other images, but need to be used with caution. They still must be glanceable to convey the basic information. Don't try to pack too much into an image and expect your user to get a lot out of it.

The map image is a bit tricky to use and get right, especially since you can only see the finished product live on Glass and not in the Playground, but we think there are some pretty good applications just crying out to use maps. You'll find them even more useful when you learn how to trigger navigation in [Chapter 10](#) and can tap into the user's location in [Chapter 12](#).



Figure 9-22. High-level view of a map

Simple Audio

If you've examined the JSON for all of the preceding examples, you would have noticed a common block in all of them:

```
"notification": {  
  "level": "DEFAULT"  
}
```

and probably some of you are wondering what this means. Setting the notification level to `DEFAULT` means, in short, that Glass should emit a short audible tone to alert the wearer that a new card, or new data, has arrived. The `DEFAULT` sound is the only one currently defined, but what if you don't want to make any noise at all? You can omit the `notification` property completely or set the `level` to `"null"`. We suggest the latter, since it works better when doing updates and makes it clearer what you're trying to do.

A better question is why would you do such a thing? It will be rare to do it for new cards, but quite common when updating timeline items that are already in the timeline (which we'll be covering later in this chapter and in [Chapter 11](#)). Certainly for things that update frequently, you don't want to annoy users with constant chirping in their skulls. It can even come in handy in some limited cases even when inserting new items, however. Consider, for example, news applications that allow their users to set "quiet periods" (we saw some examples of these in [Chapter 4](#)) but still wish to send out very important updates. They can send out the update without any notification; the card will be inserted

for the user to discover later if appropriate, but won't send out an alert that may cause problems during that period.

The notification sound is a pretty basic use of audio in your Glassware, but what about having the contents of your cards read aloud? This becomes a little more complex, but is still fairly straightforward to set up. We'll be dealing with the JSON representation of an item, so let's start with our original text object and expand upon it:

```
{
  "text": "Hello Glass",
  "notification": {
    "level": "DEFAULT"
  },
  "speakableText": "hello there Glass",
  "menuItems": [
    {"action": "READ_ALOUD"}
  ]
}
```

We've added two new object properties here. We're going to be covering menu items more extensively in the next couple of chapters, so consider this a bit of a tease.

The `menuItems` property contains an array of, you guessed it, menu items. Each menu item has an action associated with it, and we're going to be using one of the predefined actions to specify the card can be read aloud.

When selected, Glass will do a text-to-speech reading of the contents of the `speakableText` property. If that property isn't set, Glass will try to do a text-to-speech reading of the contents of the `text` property or, if that isn't available, the `html` property after stripping out any markup tags.

In our example, we could have omitted `speakableText` to demonstrate, but we suggest that you always include it if you want your user to listen to your card, and we strongly suggest you always include the option for your users. So why would you have the two of them different? Consider reading an email out loud. If just the text of the email were vocalized, you would miss a lot of additional metadata: who the email was from, what the subject was, and even that this was email you were listening to. All of this can, and should, be included in the speakable text.

Give it a try—if you don't specify `text` and `speakableText`, the menu item is ignored. Nevertheless, *you should make sure you always set `text` and `speakableText` when you are writing things with HTML*. And since the primary method of formatting Glass cards is HTML, that will pretty much be “always.” We'll be returning to how these three fields work together in [Chapter 11](#) when we talk about sharing timeline resources with other Glassware. Be careful when you're converting your HTML to text, however—you want to convert HTML entities to the actual characters they represent and not a jumble of what the markup contains. Make sure your text actually looks like text!



If you take a look at the timeline item object description (and if you haven't—you should), you'll also notice a field called `speakeableType` and wonder what role it plays with the Read aloud menu item. We've seen it used inconsistently, so we generally suggest that you set this to the name of your Glassware, but otherwise not worry about it.

See “[Don't Neglect Audio](#)” on [page 81](#) for ideas on using audible feedback.

Bundles of Fun

So far, we have focused on squeezing all our information onto a single card, but this is pretty unrealistic for many things. Text will sometimes span multiple pages or it may make sense to show conversation threads or other related items as a single bundle. While the user may treat them all as “multiple cards,” we have several different tools at our disposal to implement them, and we should be careful to pick the one that best represents the data we'll be using.

In [Figure 9-23](#), a timeline is displayed, which at any given point likely contains several bundles in addition to singular cards.

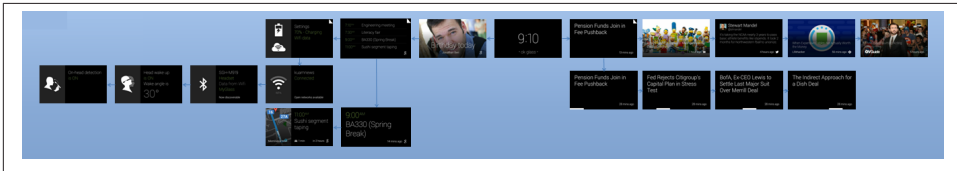


Figure 9-23. Bundles let you better organize related content

The easiest way to show multiple cards at once is to create multiple cards at once and send them at the same time. This requires nothing special on each card—all we need to do is specify more than one article, and each one has its own contents. Consider the following, all as one entry in the Playground as in [Figure 9-24](#):

```
<article>
  <section>
    <p>Page One</p>
  </section>
  <footer>
    Hover/Tap to read more
  </footer>
</article>
<article>
  <section>
    <p>Page Two</p>
  </section>
  <footer>
    Hover/Tap to read more
  </footer>
</article>
```

```
</section>
</article>
```

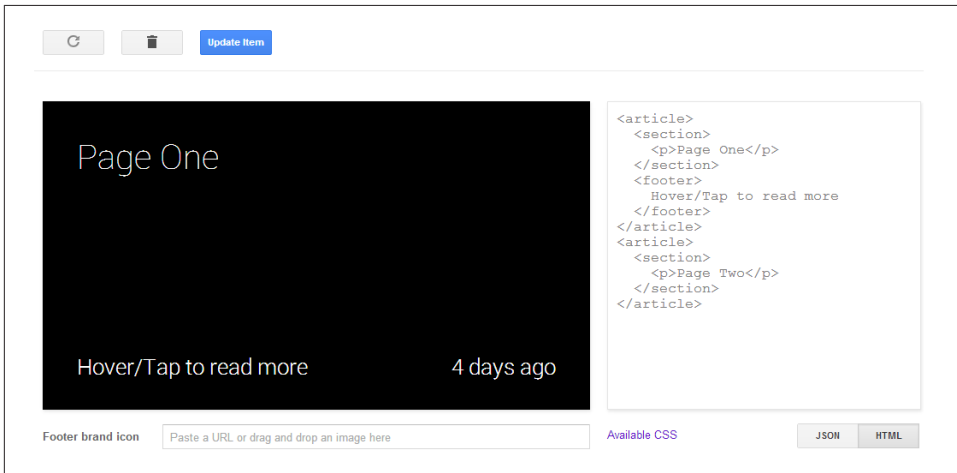


Figure 9-24. A sample card for a bundle

If we hover over the preview area, we should see an arrow on the right, indicating that there are more cards that we can scroll to see. Scrolling to that one, we now find an arrow on the left. If we add more cards than these two, we can keep scrolling through them in the order they're listed. If we send this to Glass and view it there, we can tap on the first card and the system has automatically created a Read more menu item we can tap; select it, and we can swipe between the various articles (Figure 9-25).

This works well if we know how we want to break up our contents, or if they logically divide into components we can compute ahead of time. It doesn't work as well for long text that the wearer may want to scroll through. For this, we can take advantage of an article that the system will automatically split between multiple pages. To do this, we will assign the `auto-paginate` class to the article block, and we will remove any section, figure, header, or footer sections:

```
<article class="auto-paginate">
  <p>
    Bacon ipsum dolor sit amet shank kevin pork chop dolore shankle sirloin, boudin
    veniam corned beef cillum ball tip. Tri-tip et ball tip occaecat sed veniam
    sirloin biltong. Ex biltong pork, sunt occaecat landjaeger excepteur brisket
    consectetur sausage pork belly aute leberkas ribeye. Pig beef ribs ground round
    nulla, est adipisicing drumstick. Et bacon beef magna pork chop.
  </p>
</article>
```

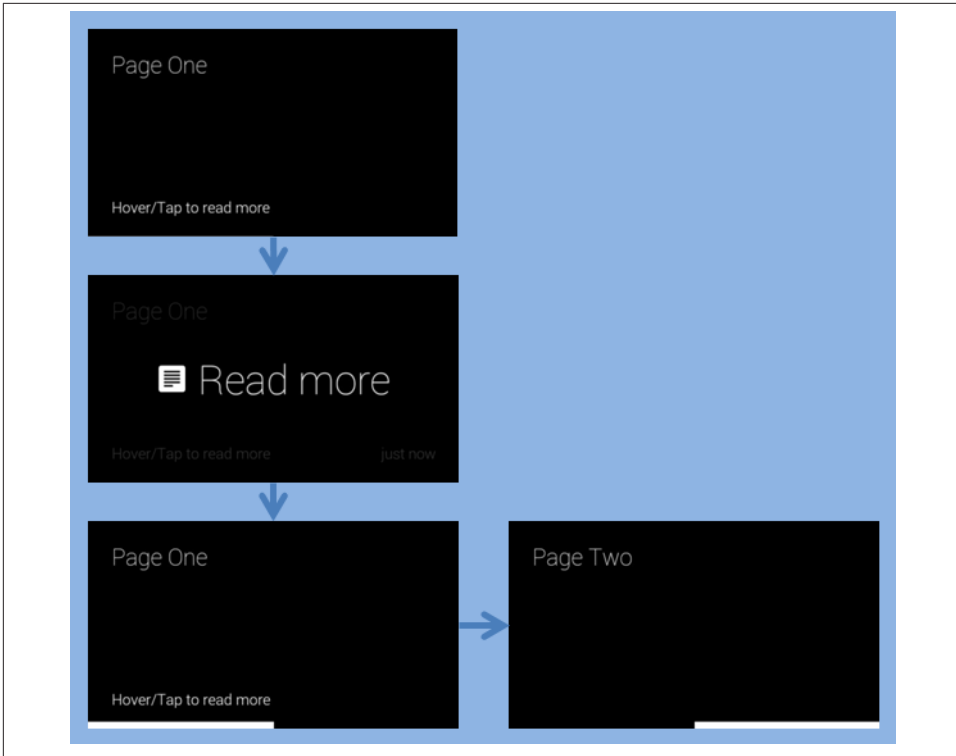



Figure 9-25. A typical bundle flow

You'll see the behavior, as laid out in [Figure 9-26](#), is much the same as with the manual pagination.

Both of these methods, however, have some drawbacks. The biggest is that there is no indication that there is actually more to read. We've hinted at it in the footer of the first card in the manual pagination, but if you're using the footer for other things, this may be impractical. Fortunately, we can combine these two methods, and some additional formatting classes, to make it more obvious.

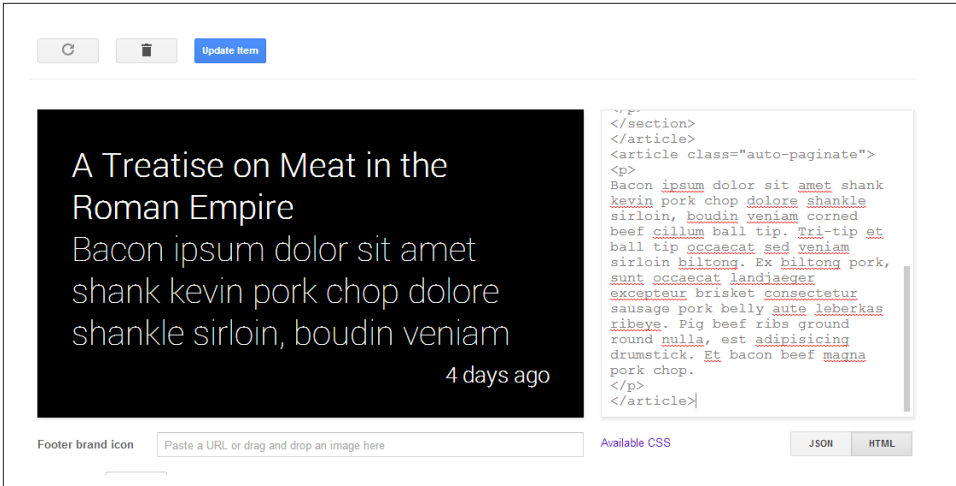


Figure 9-26. Applying the `auto-paginate` class

We'll create one timeline item with two articles on it. The first one will contain the content, and we'll let Glass truncate the contents. For good measure, and for some reasons you'll see in a moment, we'll also include a title here, but you could add a header or footer if you wished instead. The second article will contain the content again, but we will let Glass auto-paginate it for us (Figure 9-27).

```
<article>
<section>
<p><b>A Treatise on Meat in the Roman Empire</b></p>
<p>
Bacon ipsum dolor sit amet shank kevin pork chop dolore shankle sirloin, boudin
veniam corned beef cillum ball tip. Tri-tip et ball tip occaecat sed veniam
sirloin biltong. Ex biltong pork, sunt occaecat landjaeger excepteur brisket
consectetur sausage pork belly aute leberkas ribeye. Pig beef ribs ground
nulla, est adipisicing drumstick. Et bacon beef magna pork chop.
</p>
</section>
</article>
<article class="auto-paginate">
<p>
Bacon ipsum dolor sit amet shank kevin pork chop dolore shankle sirloin, boudin
veniam corned beef cillum ball tip. Tri-tip et ball tip occaecat sed veniam
sirloin biltong. Ex biltong pork, sunt occaecat landjaeger excepteur brisket
consectetur sausage pork belly aute leberkas ribeye. Pig beef ribs ground
round nulla, est adipisicing drumstick. Et bacon beef magna pork chop.
</p>
</article>
```

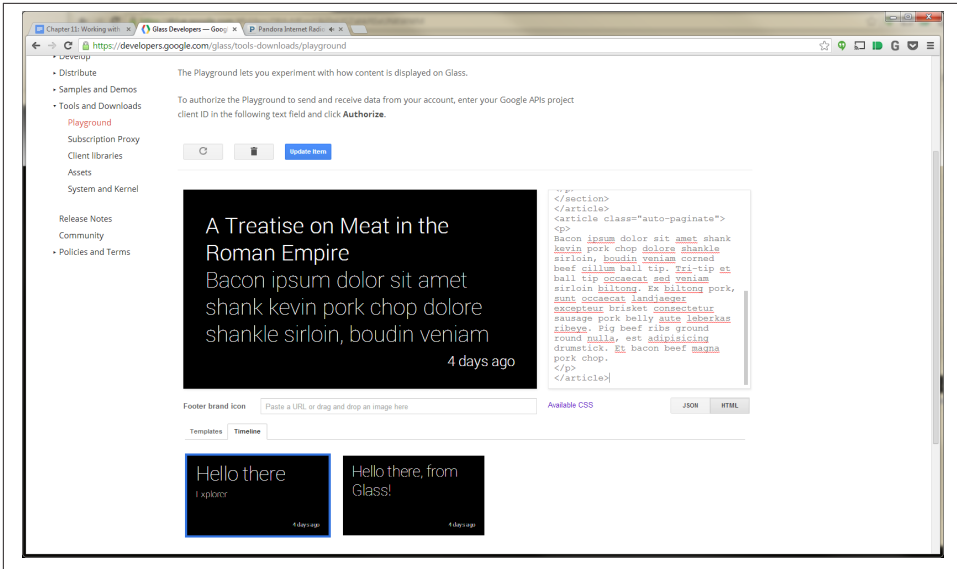


Figure 9-27. The Playground lets you preview bundles, too

Figure 9-28 shows how the content appears on Glass laid out in cards on the timeline, when you click the Update Item button in the Playground.

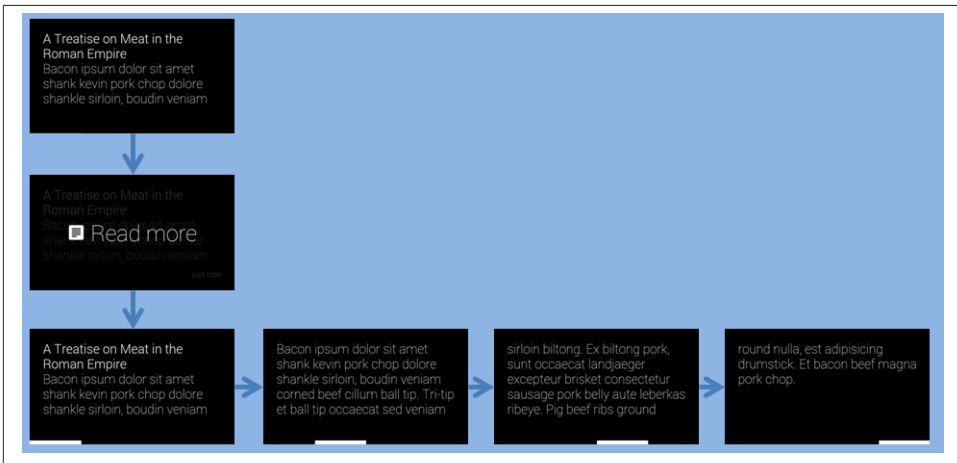


Figure 9-28. The revised bundle flow

This is an improvement, but it still doesn't really give us any cue that we need to tap it to read the rest. We also get to read that cover card a few times, and that is pretty redundant. We'll fix all of these with a few additional classes.

For starters, we'll add the `cover-only` class to the first article, to indicate that when we go to read more, it doesn't need to show it again. We'll also add the `single-line` class to the title to make sure it remains a known size and adds an ellipsis to the end of it. We want it to remain on one line to make sure we know how many lines the rest of the text will be, because the `auto-overflow` class we'll be adding to it also needs some additional attributes to indicate how many lines to show before it adds an ellipsis as well. This ellipsis will be the cue to our users that they can tap on the card to read more.

The other notable change with this HTML block is that the content is now distributed across three cards, instead of four in the previous example. So this technique may save a bit of space, giving your users one less item to swipe through. After having dissected a few examples so far for building, styling, and organizing card content, let's combine these ideas with the `cover-only` class, which is the pattern you should always use. The rendered output is an appealing layout of cards, and supports Read aloud in an intelligent way:

```
<article class="cover-only">
  <section>
    <p class="single-line"><b>A Treatise on Meat in the Roman Empire</b></p>
    <p class="auto-overflow" style="-webkit-line-clamp: 4">
      Bacon ipsum dolor sit amet shank kevin pork chop dolore shankle sirloin, boudin
      veniam corned beef cillum ball tip. Tri-tip et ball tip occaecat sed veniam
      sirloin biltong. Ex biltong pork, sunt occaecat landjaeger excepteur brisket
      consectetur sausage pork belly aute leberkas ribeye. Pig beef ribs ground round
      nulla, est adipisicing drumstick. Et bacon beef magna pork chop.
    </p>
  </section>
</article>
<article class="auto-paginate">
  <p>
    Bacon ipsum dolor sit amet shank kevin pork chop dolore shankle sirloin, boudin
    veniam corned beef cillum ball tip. Tri-tip et ball tip occaecat sed veniam
    sirloin biltong. Ex biltong pork, sunt occaecat landjaeger excepteur brisket
    consectetur sausage pork belly aute leberkas ribeye. Pig beef ribs ground round
    nulla, est adipisicing drumstick. Et bacon beef magna pork chop.
  </p>
</article>
```

...which when updated in the Playground looks like [Figure 9-29](#).

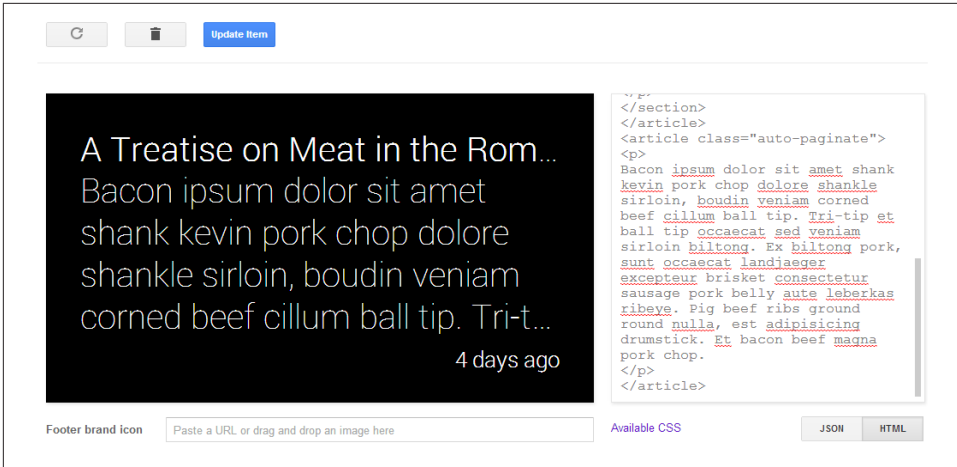


Figure 9-29. Updating the code to use a cover card

...and when inserted into a live timeline on Glass, the final flow of the items, as in Figure 9-30.

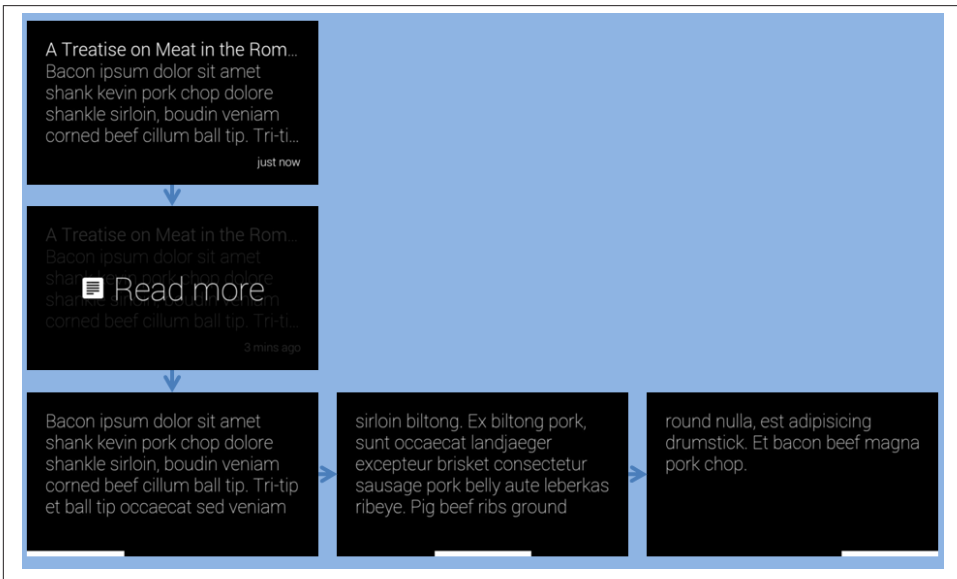


Figure 9-30. Flow with a card designated as the cover

There are some drawbacks to creating a single event that represents multiple cards, although most of them won't be obvious to you until Chapter 10 and Chapter 11. If we examine the JSON that is returned to us for all of the preceding examples, we'll see that

there is a single ID created for the whole collection of cards. This makes sense, but it also means that if (and when) we want to make changes to a card, we end up updating all the cards. It also means that all of the cards have to have the same menu items, even if it makes sense for each to have some slightly different controls. Finally, we may just want to keep track of each card individually so it matches our own internal organization of the data. All of these problems are solved by organizing cards in a bundle.

To bundle cards together, we need to set the `bundleId` timeline item property on each to the same value. What value should we pick for each? It doesn't really matter, as long as they're the same. If you're bundling things together, you probably have some internal identifier that you're already using to identify this group, and you should use it here.

Create the first card in a bundle as you'd create any card, but make sure the JSON in the Playground includes a `bundleId`. We're going to use text in our example in [Figure 9-31](#) to make it easier to read, but you can use HTML:

```
{
  "text": "Joe Montana",
  "bundleId": "mistaken-identity",
  "notification": {
    "level": "DEFAULT"
  }
}
```

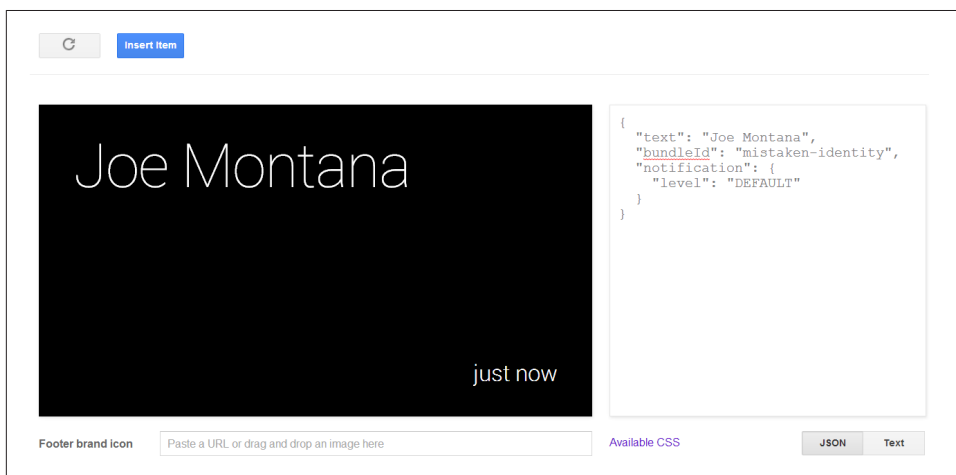


Figure 9-31. Our first “Joe” card

So far so good. There's nothing new here. Now create another card, like [Figure 9-32](#), making sure you specify the same `bundleId`. Make sure you also specify it as a new card and you're not updating the previous card:

```
{
  "text": "Joe Mantegna",
  "bundleId": "mistaken-identity",
  "notification": {
    "level": "DEFAULT"
  }
}
```

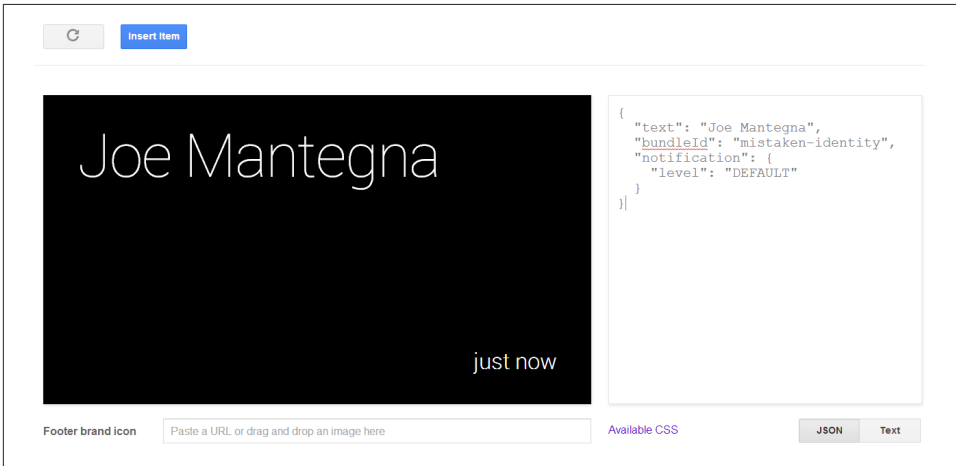


Figure 9-32. Our second “Joe” card

Click the Insert Item button and after receiving the cards on Glass, swipe through them, noticing how the items created at completely different times are now linked by their `bundleId`. And as we’ve mentioned before, Glass arranges the cards with last in, first out (LIFO) ordering, so Joe Mantegna’s item is the de facto cover card for the bundle and the first item in it, as displayed in [Figure 9-33](#).

Replicating this example in the Playground and on your own Glass hardware effectively differentiates between the quarterback Joe Montana and the actor Joe Mantegna, a common misconception (never let it be said that we didn’t give you a thorough education in this book). Here we begin to see some of the limits of the Playground. Although it shows it as two different timeline items, we need to go to Glass to see how they get bundled together. We’ll see the Page 2 card in our timeline, with the dog-eared corner indicating there is more in this bundle. Tapping on it, we’re directly taken into a sub-timeline where we can scroll between the cards. There is no “Read more” prompt—it isn’t necessary in this case.



Figure 9-33. The default ordering

This subtimeline works exactly the same as the main timeline does. Cards are arranged in chronological order, and we can see that they will change if we reinsert Joe Montana. We'll also see that Joe Montana would become the new cover card. This actually poses a bit of a problem sometimes, just as it did when we were managing the cards as a single item earlier. Is there any way to designate one of them as a cover card, or assign a completely different card as the cover?

It turns out there is. There is the `isBundleCover` timeline item property, which expects a Boolean value, which we can set to `true`. Do this for a new page, keeping the same `bundleId`, and you can see what we mean:

```
{
  "text": "Know your Joes",
  "bundleId": "mistaken-identity",
  "isBundleCover": true,
  "notification": {
    "level": "DEFAULT"
  }
}
```

This organization in [Figure 9-34](#) we find to be a bit cleaner for this example, not giving away the interior content on the cover card. Other uses of this pattern would be recipes, with the name of the dish as the cover card, then subsequent cooking steps as contained cards within a bundle.

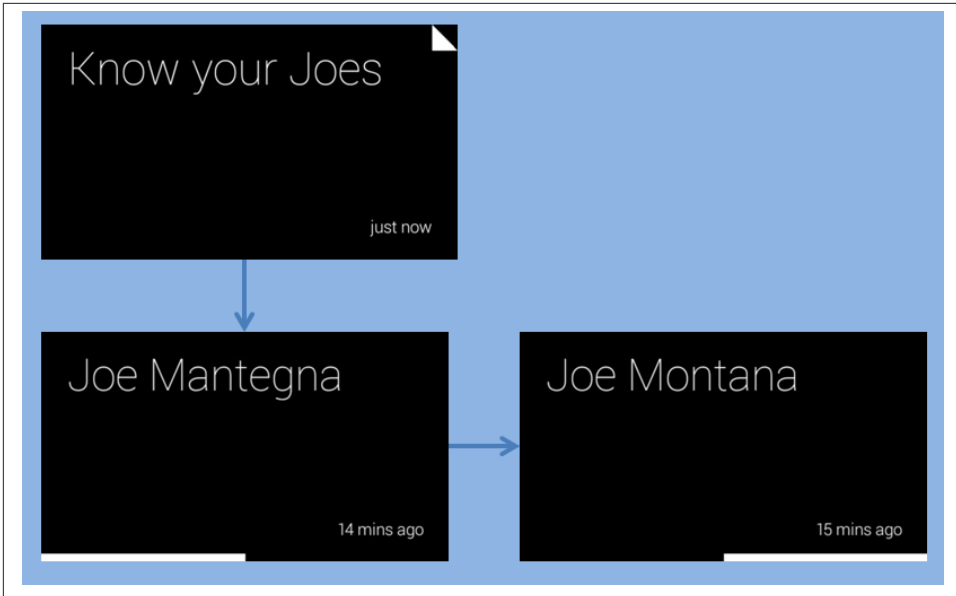


Figure 9-34. Cleaning things up with a dedicated cover card

Things can start to get really hairy when you mix bundles, explicit pagination, and auto-pagination...but we'll let you experiment with that on your own. We will also point out that if it starts looking like a mess to you, your users will think it is even worse. This may be one of those signs that your design is getting a bit too complicated.

Going Beyond the Playground

We are nearing the limits of what we can do with the Playground, but we have come pretty far using it! As you develop your Glassware, don't underestimate the power of the Playground to let you experiment with formatting your cards.

But now it is time to start working with the Mirror API more directly yourself. Create a new set of OAuth credentials (you don't want to use the same credentials that you used for the Playground) and get ready to dig in.

As we promised we'd remind you—you're not going to see a lot of code samples in the language of your choice. Instead, we're going to make sure we can show you the concept and leave the syntax to the expert: you. Google provides some code samples along with the documentation and full API documentation for each library, and those, plus your knowledge and experience with the language, will guide you on those details.

API Explorer

Google does provide some tools to help you experiment with the API without having to write code yourself. Known as the API Explorer, you can access it through the **Developers Console**. Under the “APIs and auth” section, you can click the Mirror API listing and you’ll see a list of all the API methods that are available. Clicking each one gives you a list of the mandatory and option parameters for that method and a brief description of what the parameter means, and lets you authenticate and try out some values to see what happens.

The API Explorer certainly has its uses, and if you’re going to do a lot of work with Google APIs, we suggest you learn how to use it. But it still has some limitations for what we’re going to be working on shortly, so we won’t be...exploring...it in any further detail.

Figure 9-35 shows you how can examine the methods used by the Mirror API and its associated collections. **Figure 9-36** lets you filter cards on a timeline by setting property values.

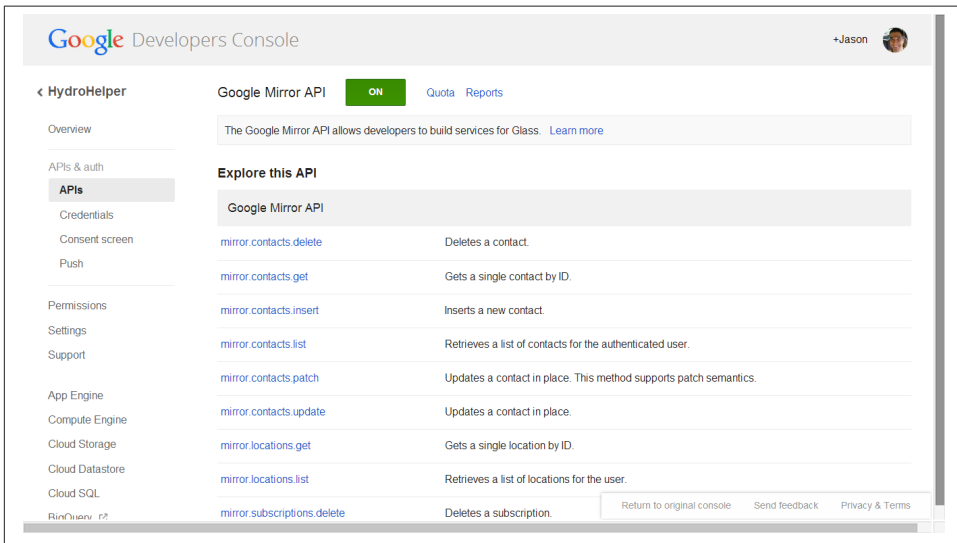


Figure 9-35. Examining methods in the API Explorer

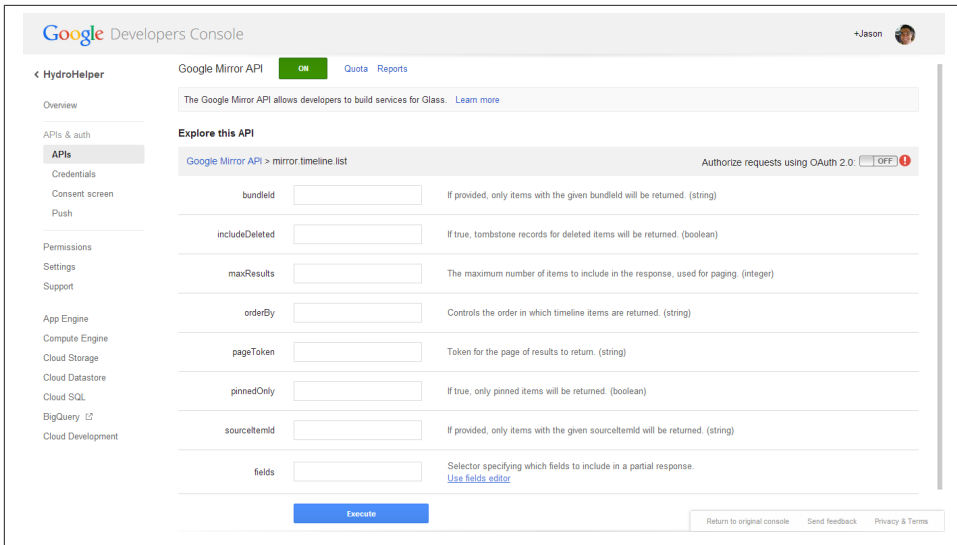


Figure 9-36. Create custom timeline views

Where are you going to find all this wonderful documentation from Google that we keep hinting about? We'll demonstrate by taking a look at the API method we've been playing with so far without you actually knowing: `Timeline.insert`.

We'll find reference documentation for all the API methods at <https://developers.google.com/glass/v1/reference/>. The body of the page contains a brief summary of all the resource types and methods we have at our disposal (you may notice that they roughly correspond to the next few chapters of this book) while the left navigation lets us jump quickly to one of them. Selecting an item on the left navigation takes us to a more detailed page about the resource, which helps us understand the properties that resource has. When using your own API, you can be assured that each of these properties are represented in the language's native property formats—getters and setters for Java, property accessors in .NET, and so forth.

If we go to the Timeline resource, for example, we see the JSON representation of a timeline item, which we've begun to get familiar with. Underneath the JSON view and a list of properties we find a summary of the available methods. We also see these methods on the left side navigation.

Let's visit the insert method. We'll see some URL information, which we can generally ignore. We'll also see the list of parameters that are accepted or required for this method and the properties that can be part of the body of the request. What is the difference between a parameter and a body property? For your purposes, not much, although you'll need to know which values you're working with are parameters and which are properties. For the Mirror API, the body properties are the object that we're working with—

in this case, the timeline item object. Parameters are often used as reference or query fields—we'll see examples of these with `Timeline.get` and `Timeline.list` although there are none that we're interested in for `Timeline.insert`.

If we continue down the page (or find the link right at the top), we get to the Examples section. This is your guide to using this method for your own language—select the tab with your language. Looking at the examples for `Timeline.insert`, we see that they all illustrate inserting a timeline item by taking a few parameters, turning them into the language-specific representation of a timeline resource, setting the text and notification properties on it, adding an attachment, and sending it off.

Don't Believe Everything You Read

Don't just copy and paste the examples into your own code, make sure you understand them in light of what we're teaching you in this book. To illustrate—some of the sample code has (or hopefully had, by the time you read this) documentation saying that the notification parameter can have the value "AUDIO_ONLY". If you read the resource description (not to mention our section earlier), however, you'll see that this value should be "DEFAULT". You'll also notice that all the examples use text as the content of the card—you should be able to extrapolate and figure out how to send HTML instead. We know you can do it!

On the other hand, you also need to make sure you keep up with the latest version (and documentation) for your library. There are occasional changes to the API, and if you started work with a previous version of the library, you may miss out on the newer features. A good rule of thumb is that if you're trying to do something that is shown in the example documentation, and your system balks, download the latest version of the library and try again.

Waitaminnit. Adding an attachment? You can do that? You certainly can—we'll go into a few details next.

Media Matters

Timeline item attachments are conceptually somewhat simple—just like with our email messages, we may want to attach some media to a timeline item to display or make available as part of that time. The implementation, however, gets complicated rather quickly. All the client libraries support one attachment of either an image or video no larger than 10MB when the card is created or updated.

The API allows for more than the single attachment—go ahead and look at the documentation for the Timeline resource and you'll see the `attachments` property, but it isn't marked as writeable, so it isn't obvious how to add them. You'll need to use the `Time`

`line.attachments.insert` method after a card is created to add additional attachments, which may create a race condition between when you upload the attachment and when you reference it as part of the item.

Speaking of which—how do you access the item anyway? Glass provides a special URL you can use to reference each attachment. The first attachment is referred to as `attachment:0`, with later attachments incrementing the counter. You can also use the URL `cid:attachment-identifier` to refer to the attachment by the attachment ID. In general, you'll probably find the former easier to use—and you'll typically only be using a single attachment.

You may be confused by this suggestion to stick to a single attachment. In our earlier examples, we showed how we might put up to eight profile photos in the background or figure section. Shouldn't these be attachments? Not necessarily, and probably not. If we make each of these profile images an attachment, it means that the image will need to be sent along with each card. It also means that Glass can't take advantage of caching to fetch the profile image once and use it with multiple cards. Where possible, you should use a public profile image (such as the one provided by Google+) and reference it by public URL instead of a private image.

Oh, CRUD...

We've talked a lot about how we insert items, but very little about anything else we can do with them. To some extent, that will be the subject of the next three chapters (what, you thought this was it?), but if you think about what we were doing with the Playground, you'll see there are a few more things we can do.

How many of you noticed the trash icon when you were looking at timeline items you had inserted? A few hands? Good. Unsurprisingly, you'll learn that this maps to the `Timeline.delete` method. Pretty straightforward—you need to know the ID of the item to delete, but since you inserted it, you should be able to keep track of it.

Similarly, if you edited a card in the Playground and sent it to Glass, you'd have seen that it replaces the original card with the updated version. There are two methods that let us do this: `Timeline.update` and `Timeline.patch`. The two seem similar, although the documentation states that `Timeline.patch` “supports patch semantics.” What does this mean?

In short, if you want to change the contents of the card, you can either rewrite the entire card by sending all the properties over again with `Timeline.update`. Any properties you omit will be as if you never sent them the first time around. The `Timeline.patch` method, however, assumes that if you didn't send a property, it should have the same value as it did the last time. If you want to remove an item with `Timeline.patch`, you

need to explicitly send it a “null” or “nil” value. Again, in each case you’ll need to know the item ID.

If you’ve done any database programming, `Timeline.update` is akin to executing a SQL statement where every one of the fields in a row in a table are populated, even ones that haven’t changed. `Timeline.patch` is the same as applying new data to an existing row but only specifying those fields where data has been modified.

Which one is better? It really depends on your needs. Sometimes it is difficult for you to keep track of what has changed, and it is just easier for you to resend the entire card. In this case, feel free to use `Timeline.update`. In other cases, you may have a huge `html` property and not want to re-create and resend it because that would be inefficient. This would be an ideal time to use `Timeline.patch`. Keep both in mind as we go forward—we’ll be using these techniques over the next couple of chapters.

For those of you who are used to thinking about things in terms of CRUD operations (Create, Read, Update, Delete), first of all, our condolences, but more importantly, you may have realized that we only talked about three of those when it comes to timeline operations: creating, updating (and patching), and deleting. You’ve seen the documentation, so you’re probably wondering why we haven’t talked about reading items or searching for them. There’s no conspiracy theory—but we couldn’t come up with a good explanation of why you would want to read them until we explained how you’d be notified about changes to them.

And that doesn’t come until the next chapter.

Card Actions and Subscriptions

A number of people have asked recently what the big deal about wearables is. After all, they reason, they're just new notification delivery platforms, right? If that's all Glass was, we might agree. Let's face it—the programs you build admittedly wouldn't be very interesting if they couldn't receive input from users and properly respond to it. Interactivity is where it's at, even with wearable computing, check that—*especially* with wearable computing—and this isn't a facet the Glass team neglected to build into the platform. While some services will be just fine existing as static read-only mediums, you've got the choice of being able to easily integrate a stout feedback loop into your projects if they demand such functionality.

This chapter primes you on what you'll need to know about using the default actions included with cards, how to create your own menu items unique to your Glassware, and how to programmatically manage subscriptions to tie it all together. We discuss how your service needs to define what actions are available within cards, how to accept those actions, and how to verify the authenticity of responses.

Simple Event Actions

We've already learned about a couple of menu items, and learned how to add one to an event. Let's recap what that looked like:

```
{
  "text": "Hello Glass",
  "notification": {
    "level": "DEFAULT"
  },
  "speakableText": "hello there Glass",
  "menuItems": [
    { "action": "READ_ALOUD" }
  ]
}
```

As you may have noticed, `menuItems` is an array, and you could have more than one menu item available to your users. The `READ_ALOUD` menu item is a simple event—Glass can take care of it completely on its own and your Glassware doesn't get a notification when the user triggers it. (Some of the other menu items do let you know what is going on—we'll be covering those in a bit.)

Two other menu actions that behave this way are `OPEN_URI` and `PLAY_VIDEO`. Each of these uses an additional payload attribute on the menu item. The payload is a string indicating what the action applies to—in the case of `OPEN_URI`, it is a URI that Glass should load, while `PLAY_VIDEO` uses the payload as the URL of a video clip to stream.

Here is a (slightly contrived) example of all three built-in menu items on a single card:

```
{
  "text": "Hello Glass",
  "notification": {
    "level": "DEFAULT"
  },
  "speakableText": "hello there Glass",
  "menuItems": [
    {
      "action": "READ_ALOUD"
    }, {
      "action": "PLAY_VIDEO",
      "payload": "https://db.tt/JVepRwr6"
    }, {
      "action": "OPEN_URI",
      "payload": "http://thinkforglass.com"
    }
  ]
}
```

You'll see that we left the `READ_ALOUD` as the first card, since it was the most likely one that people would want to select. As we go through some of the other menu items, you should keep in mind which actions will be the most used and prioritize those to be ordered first, and keep these toward the front, but also note which ones are more dangerous, such as deleting, and place these later in the menu list. This is a design tactic we mentioned earlier as a means of reducing the interaction with the system by minimizing swipes, which promotes microinteractions.

A slightly different menu item is found with the `NAVIGATE` action. Instead of using the payload attached to this menu item, this action will start navigating to the value set as part of the location property of the timeline item and not of the menu item. The location property is a Location object, and is described in more detail on [its own resource page](#). At the most basic, however, we need to specify the latitude (positive values indicating degrees north of the equator, negative indicating south) and longitude (positive

values indicating degrees east of the Prime Meridian, negative values indicating degrees west) of the point we wish to navigate to:

```
{
  "text": "Hello Glass",
  "notification": {
    "level": "DEFAULT"
  },
  "speakableText": "hello there Glass",
  "menuItems": [
    { "action": "READ_ALOUD" },
    { "action": "NAVIGATE" }
  ],
  "location": {
    "latitude": 37.422,
    "longitude": -122.084
  }
}
```

You may have noticed that the menu items are pretty generic. The menu item to launch video playback just says “Play,” with no indication what video it may be showing. Navigation doesn’t tell you where you’re going. It makes sense to assume that the card is providing this information (either in the text or footer or something similar), but this may not always be practical. And if you include multiple menu items of the same action, but with different payloads (for example, two possible movies), it may get confusing.

Glass lets us set the values that are shown for each menu item, overriding the default menu item display. We can update our earlier example to be more specific about what we’ll be watching or what website we’ll be visiting:

```
{
  "text": "Hello Glass",
  "notification": {
    "level": "DEFAULT"
  },
  "speakableText": "hello there Glass",
  "menuItems": [
    {
      "action": "READ_ALOUD"
    },{
      "action": "PLAY_VIDEO",
      "values": [
        {
          "displayName": "Jason - on-air blooper"
        }
      ],
      "payload": "https://db.tt/JVepRwr6"
    },{
      "action": "PLAY_VIDEO",
      "values": [
        {
```

```

        "displayName": "Allen - Glasstalk"
      }
    ],
    "payload": "https://db.tt/VILcUoqU"
  }, {
    "action": "OPEN_URI",
    "values": [
      {
        "displayName": "Think for Glass"
      }
    ]
  },
  "payload": "http://thinkforglass.com"
}
]
}

```

The particularly observant of you will have noticed that the `values` property takes an array of objects. We'll go into why it is an array and what other properties can be set on this object in [“Custom Menu Items” on page 209](#).

Reading More

You may have noticed that we haven't mentioned the other menu item, *Read More*, here. The reason is fairly simple—we don't control this menu item directly. Glass will add it itself when it is necessary, and not include it where it isn't.

You'll also have noticed that all of these are relatively simple menu items. Glass can handle them without having to get or provide anything further to your service. We'll discuss menu items that generate callbacks in a bit, but first we need to look at how you register the callback.

Listen Up!

The Mirror API refers to these callbacks as *Subscriptions*, and they provide information to Google about how you want to be notified when a user takes menu actions on a card, issues a voice command, shares a timeline item with a sharing contact, or changes location. We'll discuss the first one in the next section, and the other three in the next two chapters.

Subscriptions are managed on a per-user basis—it is normal for you to have at least one subscription for each user of your service. Best practice dictates that *you should insert an entry into the **Subscriptions collection** every time a user visits your site*. Existing subscriptions may have been deleted if they disabled your Glassware, but they'll always be revisiting your site if they turn you back on.

Fortunately, adding a subscription is very easy. The `Subscriptions.insert` method takes several parameters, and while some of them are listed as optional, we recommend that you use all of these:

`collection`

We're working with the timeline, so this should be the string "timeline".

`userToken`

A string that Glass will send to you when this user triggers an event to indicate which user has done so. Glass doesn't care what this value is—it will just send it back to you, so you should make it something useful, such as the user ID or some other key that lets you look the person up in your user database.

`verifyToken`

Another opaque string that Glass will just send back with the event. Think of this as a secondary key you can use to verify this call isn't forged.

`callbackUrl`

A URL that the web service responds to. It must be an HTTPS URL with a trusted (aka, not self-signed) certificate that is resolvable and accessible from the public Internet.



Be Secure

Some people have reported that during testing a self-signed certificate may work, despite Google's recommendations. That might be true (we're not brave enough to try), but it makes your subscription susceptible to breaking at any time without warning as the framework continues to grow. The hard-and-fast rule is to use an SSL certificate from a certificate authority, or host your Glassware server on a platform that provides one for you, such as Google App Engine.

During development of your Glassware you can use the `subscription proxy` Google provides as a tool. This lets you use an unsecure URL on your server and routes subscription pings to a dedicated HTTPS endpoint: `https://mirrornotifications.appspot.com/forward?url=<your_callback_url>`. You will still, however, need to use a publicly accessible machine.

Let's look at each of these in a little more detail. Some of this may seem like we're putting the cart before the horse, since you don't know what you're going to be seeing for the various events, but we have a chicken and egg problem here (to horribly mix both metaphors and cliches), so we need to explain a bit about how to handle the event before you even know what an event looks like. All will be made clear in the next section.

collection

We can subscribe to either timeline events or location events. If we wanted to subscribe to both, we need to set up two callbacks, although both can refer to the same URL. Location events will be covered in [Chapter 12](#), so don't worry about them for now; we're just going to focus on timeline events.

userToken

The user and verify tokens are for your use only. Glass doesn't care what you provide here—it is just going to send them back with each event notification. Remember back in [Chapter 8](#) as we discussed OAuth we mentioned that although it associates all the transactions with a user, it doesn't give you any information about who that user is. That is the problem that the user token addresses as well—Glass doesn't know (or care) who the user is, but you do.

You remember back in [Chapter 8](#) we discussed getting the user's ID? No? Well, go back and read all of [Chapter 8](#) then, because it is important and you probably forgot. But to refresh your memory, we said:

So how do we get a user's identity? It depends on what we need, exactly. In many cases, we just need a unique identifier from them—some way to know what user configuration we should be using when the user returns to set up or change their settings.

This unique identifier also will work well as the user token, so when a callback comes in for this user, we can pull up their record and know who performed the action. Most importantly, we'll need to pull up their access token so we can act on the timeline item being reported on. If you think in database terms, this user token works well as a unique key against your user database.

Waitaminute... we'll get a callback telling us something has happened, but we don't know what? We'll clarify in a minute, but in short, the callback tells us a timeline item that is being acted on, but we need the user's permission to get that item. We have that permission in the form of the access token.

verifyToken

But what then is the purpose of the verify token? The callback URL we're providing needs to be available publicly on the Internet. While we're intending it for Google's use only, we have no good way to restrict who might be calling it. So if someone finds out the URL, they may also be able to guess what you're using as the primary key, and then generate false events for you to process. Depending on what these events are, this could be very bad! The verify token should be a random string you generate for each user, stored as part of their user records, and provided only to Glass when you create the

subscription. When an event comes in, you can confirm the notification comes from Glass because they're the only one that can provide a valid verify token.

The Source (Address) of the Problem

You might wonder why you can't verify that the request is coming from Google by checking it against a known list of IP addresses and blocking any that come from outside those addresses. While this works for other Google services (most notably Google's spider bot), it is too early to be able to do this for Glass. Glass is still scaling up to be able to handle a massive number of callbacks, and it isn't reasonable to know the exact addresses of all the machines when they are frequently changing.

callbackUrl

The final parameter is probably the one that is most significant to you—the URL endpoint itself. As we've noted, it needs to be publicly available from Google's servers with a valid HTTPS certificate. While it may be obvious, it also bears making plain that it also must be a URL that your app server will reply to! So you will need to write the code and configure the framework or environment that you have to call your code when this URL is hit.

Glass will access the URL via a POST operation, and the body of the POST will be a JSON object. You'll need to convert this JSON object into something you can work with natively (see [the Mirror API documentation on subscriptions](#) for code samples in a couple of languages) and then access the parameters to get the timeline item ID, the user token, validation token, and some other information about what triggered the event. We'll start going into details of all these fields in the next section.

Responding to Subscription Pings

As with all calls to your server via HTTPS, you'll need to return a response code. A response code of 200 ("OK") indicates that everything is in working order, while other error codes would indicate there is a problem processing the data. You have about 10 seconds to return an "OK" response code—if you return an error or take longer, Google may try to resend the request at a later time. One common problem is getting multiple callbacks on the same event because you didn't send the response code back fast enough.

A good solution to this problem is to accept the event notification, add it to a job queue to handle in another thread or process, and then immediately tell Google that it has been accepted by returning the 200 status code. You don't need to do any actual processing on this event immediately, so you're handling it very quickly. This reduces the chance of double-notifications, plus it sets you up to handle the load you may get as your service becomes wildly popular. As the load grows, you can handle responses from

Google with one server or process, and then distribute the task processing on one or more other machines that do the actual work involved.

And just what needs to be done when the process pulls the task off the job queue? It depends on exactly what action the user performed, but many things will be similar. Let's start taking a look.

Simple Callbacks and How to Handle Them

We're finally getting to the second half of our chicken-and-egg problem by setting up some other menu items that we can get responses about. The two simplest that we can use at this point are "TOGGLE_PINNED" and "DELETE":

```
{
  "text": "Pin me! Delete me!",
  "notification": {
    "level": "DEFAULT"
  },
  "menuItems": [
    { "action": "TOGGLE_PINNED" },
    { "action": "DELETE" }
  ]
}
```

You may want to try these out to just see what they do. After some experiments, you'll have discovered that deleting a card does just that—removes it from your timeline. Pinning and unpinning a card moves it between the “history” timeline to the right of the home card and the “now” portion of the timeline to the left of the home card.

Pinned cards serve a special purpose with Glass. They are meant to provide quick access to some information the wearer is interested in either seeing frequently or manually checking for updates. But remember, the data needs to stay timely—if your Glassware doesn't keep it updated, it will vanish in seven days as Glass purges old items. As we've also discussed, pinned cards shouldn't be used to launch applications, and part of the way this is discouraged is by preventing you from pinning a card—that must explicitly be a user action.

Both deleting and pinning, however, do much more than meets the eye.

If you have a subscription registered (and if you were actually following along, we hope you do), you'll have gotten a callback on your URL that contains a JSON body something like:

```
{
  "collection": "timeline",
  "itemId": "3219c55c-3d26-4c79-a8b5-5ccc0732da88",
  "operation": "UPDATE",
  "userToken": "12345678901234567890",
  "verifyToken": "shibboleet",
```

```
    "userActions": [  
      { "type": "PIN" }  
    ]  
  }  
}
```

The properties for the callback body are pretty straightforward:

collection

This matches the collection string we set for the subscription. Again, it will be “timeline” for now.

userToken *and* verifyToken

These are also the values we set for the subscription.

itemId

This is the item ID for the card the menu items were attached to. Remember that Glass assigns this ID—we have no control over it. We know what it is for a card after we create the card by examining the results, or we can do a query for the card to refresh our memory. We’ll look at the latter shortly.

operation

This will be either INSERT, UPDATE, or DELETE. As we will see in the next chapter, if a new card is permitted to us, we’ll get an INSERT operation. UPDATE operations are sent to us for most menu items except for Delete, which will generate a... anyone? Anyone? That’s right...a DELETE operation.

userActions.type

Something else for the UPDATE operation, this tells us exactly which menu option was called. (It also does for INSERT or DELETE, but those are a little more straightforward.) Each menu item generates a different user action—we’ll cover them as we go. Pinning a card generates the PIN action, while unpinning the card gets us UNPIN.

Given what we know so far, we might consider the following flow when processing a job off the callback queue:

1. Get the user record from the database based on the userToken.
2. Compare the verifyToken from the database with the verifyToken sent in the request.
3. Do something based on the operation and userAction.

That third step is, of course, the kicker. What sorts of things would you do? First, realize what you don’t need to do—Glass has already taken care of deleting or changing the pin status of the card, so you don’t need to do it on Glass.

You might, however, wish to do it if you have your own copy of the card. Consider, for example, a simple text chat system that you're developing a Glass interface for. If a user pins a message or thread, you might want to mark the message as important in your master copy of the message so he can see it on the desktop as well. Similarly, if he marks the message as deleted on Glass, you should also remove it from your own data store.

Having the master copy of a card in our database works really well if we can also store the ID that Glass uses in our database. This isn't always feasible, however, and even if it is, sometimes it might not be a good idea. Fortunately, Glass is willing to be a little flexible on this point. It still will always maintain its own unique ID for a card, but it also provides a secondary field it indexes that you can use for your own identifier. This property of a timeline item is the `sourceItemId`. Glass will store this value as part of a card, but does not care what the value itself is.

So we can update our previous example with the following:

```
{
  "text": "Pin me! Delete me!",
  "sourceItemId": "message-24601",
  "notification": {
    "level": "DEFAULT"
  },
  "menuItems": [
    { "action": "TOGGLE_PINNED" },
    { "action": "DELETE" }
  ]
}
```

Now when we process the event notification, we see that the callback body contains, well, exactly the same thing it contained before. The `sourceItemId` doesn't show up in the callback. So how does this help us again?

We do have the `itemId`, which we can use to fetch the timeline item using the `Timeline.get` method. This will give us a complete Timeline item object, including the `sourceItemId`. With this `sourceItemId` value now in hand, we can now fetch the object from our own database and take whatever action is appropriate.

The timeline item we fetch has some other attributes that may be of use to us if we're processing the delete commands or one of the pin actions. In particular, we have Boolean object attributes such as `isPinned` and `isDeleted` which report on, you guessed it, if the item is pinned or not, or if the item is deleted. If the item is deleted, we may be in for a bit of a shock—many other attributes will have been removed, including the `text` or `html` attributes. We call these object fragments “tombstone” objects, since all they do is to mark that there used to be an item here.

Given this, it becomes extremely common for our processing flow with a job in our job queue to work more like this:

1. Get the user record from the database based on the `userToken`.
2. Compare the `verifyToken` from the database with the `verifyToken` sent in the request: if they do not match, bail out.
3. Fetch the timeline item from Google's server.
4. Do something based on the operation, `userAction`, and the current state of the item itself.

This concept of Glass mirroring what is available on other platforms is a powerful and important one, and one that we hope you picked up from our earlier chapters as well (though it bears repeating). But the two options we've seen to do that mirroring so far are somewhat limiting. While deleting is a common operation, many times users may want to just dismiss a card from Glass without removing it from every platform. Additionally, pinning does not always translate to commands that may exist elsewhere and has the downside that you can't control it from outside Glass, which limits its use as a multiplatform control.

Besides, chances are good that you've created your own command terminology, and you want your Glassware to reflect the same terminology. We need a way to create our own menu items.

Custom Menu Items

Now that we know how to set up a subscription and respond to pings from Google, let's see how to create some more menu items for when users interact with our Glassware, and how to handle them (remember from [Chapter 7](#) how the architecture of the Mirror API mandates that your server never communicates directly with the user, or vice versa). Although Glass provides some common menu items, there is no way it can have all the possible choices you may want for a card. To accommodate your needs, you can create custom menu items that contain your choice of text and icon, and will report their activity to the server. We first need to see how to create the menu items and everything associated with them, and then what it will look like on the server.

The "CUSTOM" `menuItem` action indicates that we have a, you guessed it, custom menu item! Really quite intuitive. What a wearer sees for this menu item is determined by an array of values, which we touched on earlier to modify how a default menu item was listed, and which should include a 50 x 50 pixel icon in the PNG format. Glass distinguishes multiple custom items through the `menuItem` ID, which we designate and which Glass ignores except to send back to us when the menu is selected. So we might set up a color changing card with JSON like this:

```
{
  "html": "<article><section><p class='white'>A cavalcade of color
  </p></section></article>",
```

```

"menuItems": [
  {
    "id": "white",
    "action": "CUSTOM",
    "values": [
      {
        "state": "DEFAULT",
        "displayName": "Watch White",
        "iconUrl": "https://db.tt/0VgPaGii"
      }
    ]
  },
  {
    "id": "red",
    "action": "CUSTOM",
    "values": [
      {
        "state": "DEFAULT",
        "displayName": "Ahead Red",
        "iconUrl": "https://db.tt/0VgPaGii"
      }
    ]
  },
  {
    "id": "blue",
    "action": "CUSTOM",
    "values": [
      {
        "state": "DEFAULT",
        "displayName": "Go Blue",
        "iconUrl": "https://db.tt/0VgPaGii"
      }
    ]
  }
],
"notification": {
  "level": "DEFAULT"
}
}

```

Entering this JSON into the Google Mirror API Playground produces the snapshot of your card shown in [Figure 10-1](#).

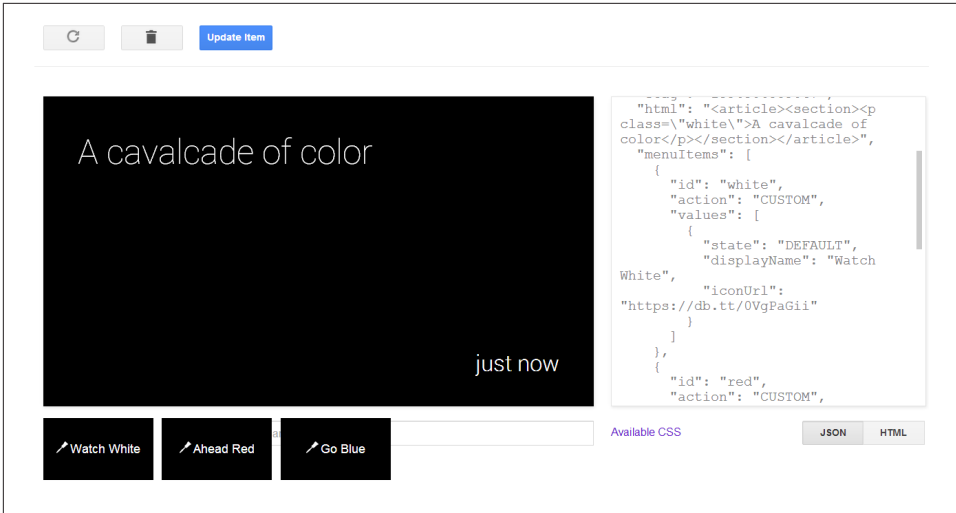


Figure 10-1. Previewing prototypes of menu items

Do note that after you've clicked Insert Item to send the card to Glass, the Playground will auto-generate thumbnails of the custom menu items you've laid out, as **Figure 10-1** indicates. This feature only visualizes custom menu items, not the built-in options.

The JSON may look complicated, but let's break down the `menuItem`s property into some detail. We've seen that this will be an array of objects before, but the objects contain a few more details this round.

Each object contains an ID, in this case containing the color we want to switch the text to, and the `CUSTOM` action. Each object also contains a `values` property, which is an array. Right now, that array contains just one item. That item has the state of `DEFAULT`, an icon representing that we're setting a color, and the text to display—the name of the color to switch to.

Viewed as live items on a timeline, the cards and their flow render like **Figure 10-2**.



Free Tools for the Graphically Challenged

If you're like the both of us, you probably royally suck when it comes to graphics. Thank goodness for the Glass Asset Studio, an amazing tool by our friend Justin Ribeiro that creates white icons out of images in the dimensions Glass expects. It's an indispensable utility to have as you're putting together your Glassware.

Additionally, the icons we note in this chapter were used freely from Openclipart.org. Yay, public domain resources!

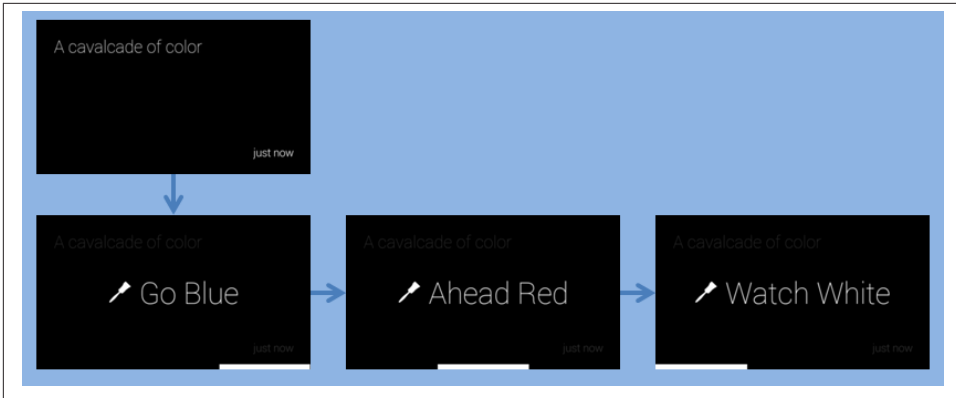


Figure 10-2. Flow for a card with accompanying menu items

If we take a look at this card, we get the text and the three menu items. If we have a subscription set up, selecting one of the menu items will send something like this to the server:

```
{
  "collection": "timeline",
  "itemId": "7e84302a-1da7-46f2-ac72-2c008fce2b4b",
  "operation": "UPDATE",
  "userToken": "12345678901234567890",
  "verifyToken": "shibboleet",
  "userActions": [
    {
      type: "CUSTOM",
      payload: "blue"
    }
  ]
}
```

Most of these should be familiar, but some have new values:

`operation`

Will be set to `UPDATE`.

`userActions.type`

Contains the string `"CUSTOM"` to indicate this is a custom menu item. Big surprise.

`userActions.payload`

This is a new field and will correspond to the ID of the `menuItem` that was selected.

We can use the same process flow we outlined earlier to determine which menu item was selected, but what should we do for each menu item once we get it to actually change the color of the card? In this case, we can call `Timeline.patch` for the `itemId` mentioned in the payload and just specify the `html` parameter with the HTML specifying the new

color class. What if we wanted to be fancier and use a different icon for the color that has been selected? We could send `Timeline.update` with a complete copy of the timeline item configuration and just the relevant menu item changed (or perhaps not even present, so they can't foolishly select the menu item twice). How you alter the card and menu items is completely up to you.

We have mentioned a few times now that you should be deliberate in how you differentiate the menu items for "Delete" and "Dismiss". The former (built in, and covered earlier) will delete the card from Glass and you are expected to delete it from your system as well. The latter is not built in—you're expected to implement it if you want your users to be able to remove cards from the timeline.

We can take care of this easily with a CUSTOM menu item, as illustrated here:

```
{
  "text": "Dismiss or Delete me",
  "menuItems": [
    {
      "id": "dismiss",
      "action": "CUSTOM",
      "values": [
        {
          "state": "DEFAULT",
          "displayName": "Dismiss",
          "iconUrl": "https://db.tt/dRHLFhxG"
        }
      ]
    },
    { "action": "DELETE" }
  ],
  "notification": {
    "level": "DEFAULT"
  }
}
```

On the server, we will get the CUSTOM action with the "dismiss" payload and call the `Timeline.delete` command for the `itemId`. We need to do this explicitly—Glass won't do it for us. At the same time, however, we should not delete our internal copy of the item.

Dismiss Is DIY

You might have taken note about how seemingly backward the ability to dismiss a timeline operation is. Dismiss, you would think, should be a built-in function of the Mirror API, and delete should be custom. But that's not the way it currently works out. This does give you a bit of leeway as to implementation and lets you gather stats about how often people do each operation.

If you add this card and play with it a little, you'll realize that there is a difference in how the "Dismiss" menu item and the "Delete" menu item behave. When we delete, it gives us a brief period where it says it is "Deleting" and we have a moment to cancel things before it tells us that it has been "Deleted". We don't get that at all with "Dismiss". These two other screens are other value states that we have omitted—the DEFAULT value state is required for us to show anything, but the PENDING and CONFIRMED states make for a better experience. We can add these as two additional objects to the "values" attribute thusly:

```
{
  "text": "Dismiss or Delete me",
  "menuItems": [
    {
      "id": "dismiss",
      "action": "CUSTOM",
      "values": [
        {
          "state": "DEFAULT",
          "displayName": "Dismiss",
          "iconUrl": "https://db.tt/dRHLFhxG"
        },
        {
          "state": "PENDING",
          "displayName": "Dismissing",
          "iconUrl": "https://db.tt/dRHLFhxG"
        },
        {
          "state": "CONFIRMED",
          "displayName": "Dismissed!",
          "iconUrl": "https://db.tt/dRHLFhxG"
        }
      ]
    },
    { "action": "DELETE" }
  ],
  "notification": {
    "level": "DEFAULT"
  }
}
```

Let's test this JSON code in the Playground to see the layout of the custom menu item with our icon in [Figure 10-3](#) (again, only Dismiss will be thumbnailed, but not Delete since it's a built-in).

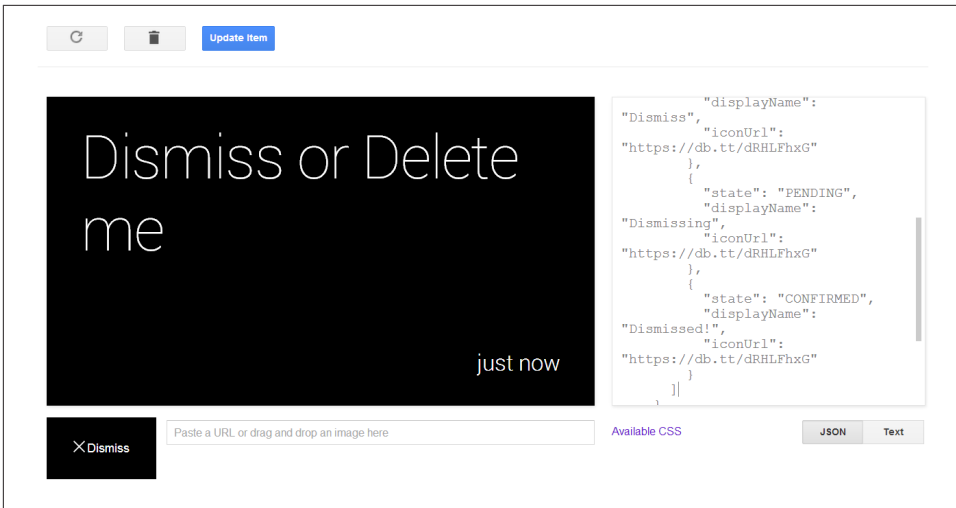


Figure 10-3. Testing a Dismiss action

Looking at the cards live on Glass produces Figure 10-4's expected layout.

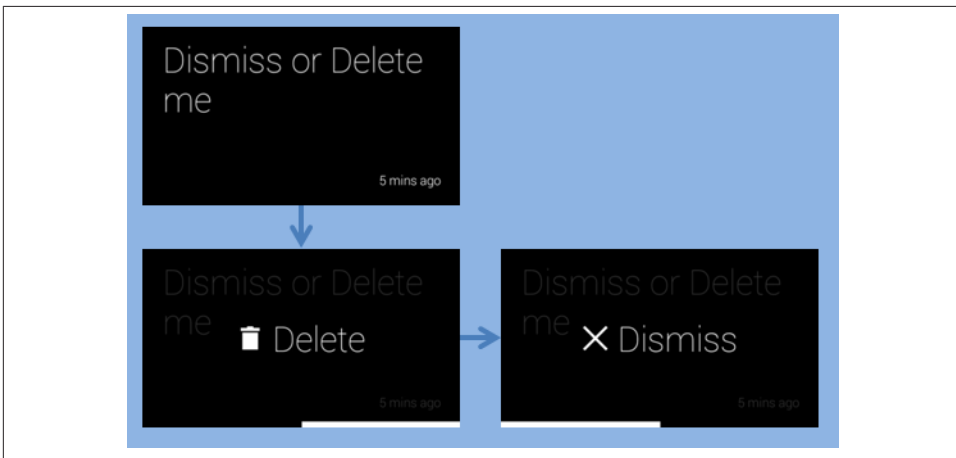


Figure 10-4. Flow for the custom Dismiss action with the default Delete menu item

Glass encourages us to add these additional states, and we strongly suggest that you add them for all your menus. Your users will be expecting them and they are natural parts of the Glass UX.

Keeping in Contact

Earlier we saw the specialized NAVIGATE menu item that would work with a Location object that was part of a timeline item. We have another object that also provides some additional commands when it is part of a card—the Contact object.

The most simple way to use a Contact is as part of the `creator` attribute on a timeline item. If we have assigned a telephone number to this `creator`, we can add a `VOICE_CALL` menu item to allow the wearers to directly call a phone number, in much the same way they could access a URL:

```
{
  "text": "O'Reilly Media",
  "notification": {
    "level": "DEFAULT"
  },
  "creator": {
    "displayName": "O'Reilly Media, Inc.",
    "phoneNumber": "(707) 827-7019"
  },
  "menuItems": [
    { "action": "VOICE_CALL" }
  ]
}
```

This is a pretty simplistic use of the `creator` attribute, however, and it almost seems easier to embed the telephone number as part of a URL and use that method instead. But fear not—there are bigger and better uses for Contact objects.

Returning to our hypothetical example of a simple text messaging application, we can use the **Contacts collection** to indicate the participants in a discussion. Glass will even (sometimes) format the message using information we can provide as part of the `creator` property as well as a list of recipients specified through the `Timeline.recipients` property, but we strongly suggest you use HTML to control the exact layout you want. We're going to ignore our own suggestion for this example, however:

```
{
  "creator": {
    "displayName": "George",
    "id": "user-prez1",
    "imageUrls": [
      "https://mirror-api-playground.appspot.com/links/washington.jpg"
    ]
  },
  "recipients": [
    {
      "displayName": "Abe",
      "id": "user-prez16",
      "imageUrls": [
        "https://mirror-api-playground.appspot.com/links/lincoln.png"
      ]
    }
  ]
}
```



```

    ]
  }
],
"text": "Welcome Abe",
"bundleId": "thread-cicchat-1812",
"sourceItemId": "cicchat-1812-1",
"menuItems": [
  {"action": "REPLY"}
],
"notification": {
  "level": "DEFAULT"
}
}
}

```

Here we have a creator and a (rather small) list of recipients of this message. When we view it in the Playground, we see one of the `imageUrls` belonging to Washington. When we send it to Glass, however, we see a somewhat different format—Washington is on top, in the usual position where we expect the sender to be, and Lincoln is shown on the bottom as (the only) one of the recipients. This serves as a reminder of three things: the Playground isn't always perfect, you need to test things with Glass itself, and you should control the formatting if you want it to look correct.

We've also included a `bundleId` and a `sourceItemId` here. We'll explain why we included them, and why their values are not an arbitrary pick, in a few paragraphs. Let's look at this rendered in the Playground in [Figure 10-5](#).

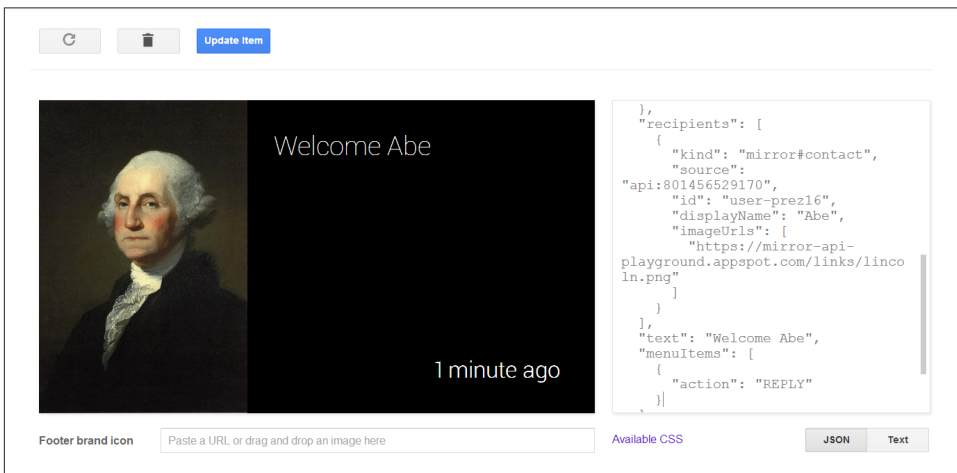


Figure 10-5. Make sure you test your card prototypes on Glass!

And so now for purposes of consistency, let's look at [Figure 10-6](#), which is produced after clicking Update Item and sending the item to Glass.



Figure 10-6. Testing image placement

It's decidedly different, including both presidents!

We also see a new menu item action here, REPLY. Go ahead and try it out—when you select the REPLY menu item, you'll be able to dictate text to Glass that is sent back to the Glassware. This works differently than any other menu item we've seen before since it will generate a response with an INSERT operation. As you may surmise, this suggests that a whole new card has been created to mark this event and we're getting alerted that this new card exists:

```
{
  "collection": "timeline",
  "itemId": "4bf0669e-bdcc-4641-8da6-5725a84fcffa",
  "operation": "INSERT",
  "userToken": "user-prez16",
  "verifyToken": "fourscore",
  "userActions": [
    {
      "type": "REPLY"
    }
  ]
}
```

When we process this callback on our job queue, we'll go through our usual steps. When we get a message with the ID 4bf0669e-bdcc-4641-8da6-5725a84fcffa, we might get something like this:

```
{
  "kind": "mirror#timelineItem",
```

```

    "id": "4bf0669e-bdcc-4641-8da6-5725a84fcffa",
    "inReplyTo": "470e1114-e06f-433b-94ff-67830e7d0107",
    "created": "1861-03-04T12:19:36.713Z",
    "updated": "1861-03-04T12:19:36.713Z",
    "etag": "1396181976713",
    "recipients": [
      {
        "kind": "mirror#contact",
        "source": "api:243969892606",
        "displayName": "George",
        "id": "user-prez1",
        "imageUrls": [
          "https://mirror-api-playground.appspot.com/links/washington.jpg"
        ]
      }
    ],
    "text": "Glad to be here, George",
    "notification": {
      "level": "DEFAULT"
    }
  }
}

```

We see that the creator of the message we replied to is now included as the recipient (if we used the `REPLY_ALL` menu control, all the previous recipients would also be included here). There is no creator set—Glass doesn't know who sent the message (or at least doesn't know the contact info for them). It also hasn't associated it with the bundle or the original source ID. We do, however, have a new field, `inReplyTo`, letting us know the original message that was sent, and we can get both the `bundleId` and the `sourceItemId` by using `Timeline.get` and getting that message as well.

Why do we care about the `bundleId` and `sourceItemId`, however? Well, we want this to maintain consistency with our own database, so we need to make sure this reply gets attached to the right message thread and we know which message we're replying to (since we do nested messaging when we view it on the Web). The `bundleId` corresponds to the thread, and we can now look it up in our database and add it to the thread. It doesn't necessarily need to be the exact value of the thread ID—in this case, we have prefixed it with a string so we know this is a thread on our message board and not a private message, but however the value gets encoded in the string, we need to know how to decode it.

After we insert this new message into the thread, we should now have a new message identifier—what corresponds to the `sourceItemId`. We might want users to reply to their own message, too, so we should do a `Timeline.update` or `Timeline.patch` to update the message *that was just inserted* to give it the new `sourceItemId`, the `bundleId` that corresponds to the thread, and the menu items that all the other posts have. If we're formatting the message differently (perhaps to give it a title or to format a long message as we've previously discussed), this would be a good opportunity as well.

This brings up an interesting point, however. What happens if the original message in our datastore is updated or we need to find out which messages in a thread are still on the timeline? We know how to get things by the ID that Glass knows them by, but how do we get them if we only know our ID? We can use the `Timeline.list` method and specify either the `bundleId` parameter or the `sourceItemId` parameter. This returns a (possibly large) list of matching results. If the results are particularly large, we may need to make multiple calls to page through them all. (You don't need to specify either of these parameters—if you don't, you'll get all the timeline items that your Glassware has inserted that haven't yet expired.)

The ability for `REPLY` and `REPLY_ALL` to take voice-entered text is a very tempting feature. It is easy to think of pinning a card, giving it a `REPLY` menu item, and using this as a way to send voice commands to our Glassware. This is a bad idea for three reasons:

1. Pinning is an ineffective tool to set up a launcher, as we have discussed a few times already.
2. Using the free-form text entry to issue commands can cause all sorts of problems with trying to parse what the user actually said. It may be good in some rare cases, but generally is best avoided.
3. We have a better way to set up `Contacts` as specialized voice commands and sharing endpoints, which we go into detail about in the next chapter.

Sharing Resources with Glassware

We've done some exciting things with timeline items so far by using the Mirror API—creating them, adding menus to them, deleting and updating them, and replying to them, but the real power behind the Mirror API ends up being attached to contacts and being able to share things with them. We've mentioned a lot so far in this book about the ability to share information, and this chapter drills down into what it takes to manipulate this powerful feature in your services. The concept of sharing timeline cards that contain content in a variety of media formats created by the wearer—or even cards distributed by other Glassware—is as important as subscriptions. In fact, the latter couldn't exist without the former!

In this chapter, we'll take a look at the basics behind sharing, but also learn how Google's implementation of sharing takes mobile computing a step further, making for some really interesting opportunities for your projects and the destinations your data can have. This includes both the Share menu item, as well as leveraging voice commands that are present on the main Glass menu.

The Share Menu Item

In our hypothetical chat Glassware from the last chapter, we covered replying to a message. If we think about other operations we tend to do on messages, we can see that we're missing a forward-like command. If you looked at the list of available values for the `Timeline.menuItems.action` property, you'll have noticed that `SHARE` hasn't been discussed yet. You put the two together...and there you have it ([Figure 11-1](#))!

This introduces an interesting problem. It suggests that not everything can be shared—it's up to the Glassware that creates the card to offer the opportunity to share it. You may be tempted to hoard your precious card for yourself—but resist the temptation. Users are going to expect to be able to share cards with other services—this is, after all,

one of the core features of Glass. And this is the fourth Noble Truth from [Chapter 5](#)—*Avoid the Unexpected*.

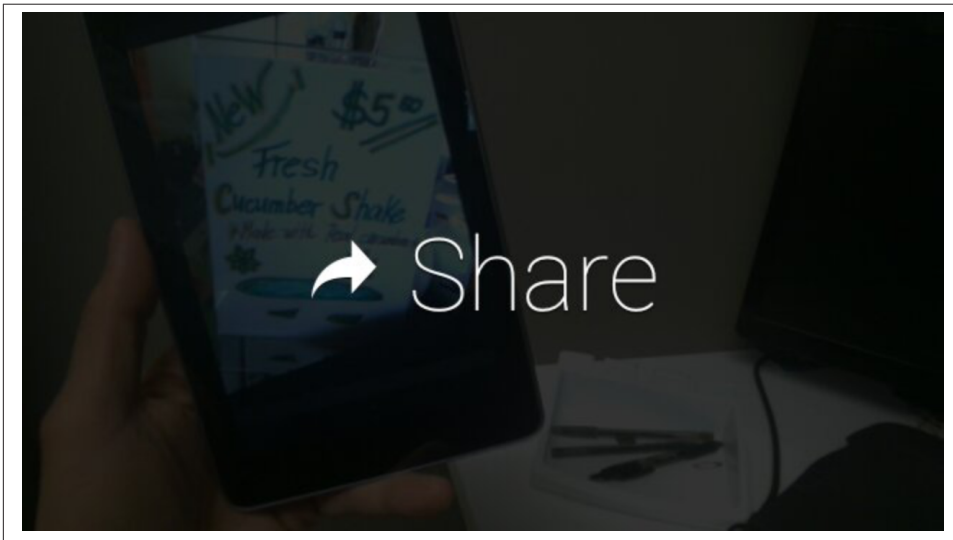


Figure 11-1. The Share action

We’re not going to show you how to add the Share action item—you should be able to figure it out from everything you learned in the last chapter and by referencing the [documentation about the Timeline](#). But you may be wondering how to specify *who* or *what* your users will be able to share their cards with. We’re not going to include a contact as part of the card—that works for replies, but doesn’t make sense for shares.

Instead, contacts will need to be registered in their own dedicated collection.

Share Contacts

Looking at the [Contacts resource representation](#), you should have a sense of familiarity. We have the expected operations: inserting new ones, deleting old ones, getting or listing our contacts, or updating or patching them.

Creating and managing contacts in this way is different than including a contact as part of a timeline item. Contacts attached to a card are useful for just that card; contacts added by calling `Contacts.insert` are available as sharing targets and for voice commands. We sometimes refer to these contacts as “sharing contacts” to highlight their role.

Sharing contacts have a number of attributes you need to set. A few of the properties listed on the resource page aren't necessary for a sharing contact, but these are the basic ones:

`id`

Unlike a timeline ID, you will be setting this value yourself and Glass ignores it except to echo it back to you when you get an event involving this contact. If you have multiple contacts, and we'll explore why you may or may not in a bit, you should set this to something meaningful to you (for example, the ID in your own database of the user represented by this contact).

`displayName`

The name to show for this contact. It should not include the name of your application, since a listing of the contact will show your name before the contact name. This will be shown in a list when triggered from any "OK Glass" audio prompt or on a full card (with the `imageUrls` specified—see next item) when selected from a tap.

Let's look at some examples in Figures 11-2, 11-3, and 11-4 of how a resource in Glass—a photo, a video, even a link—can be shared with an entity in the Glass ecosystem.

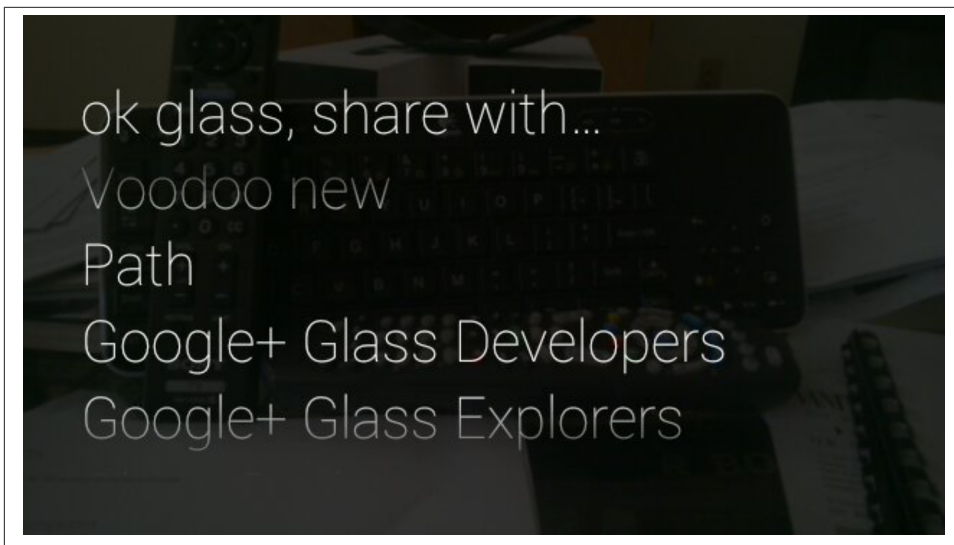


Figure 11-2. The Share voice command



Figure 11-3. Glassware sharing options



Figure 11-4. More sharing options

imageUrIs

This will be a list of URLs to show as the background for this contact. Images should take up the entire 640×360 display and have a nonblack, nontransparent background.

acceptTypes

You should only have your contact listed for content types that you're prepared to handle. It doesn't make any sense for someone to share a video with you if all you know how to handle is a picture, so this is where you would list all of the MIME types that you can handle. By default, if you leave this property blank, Glass assumes support for all MIME types. Wildcards are supported, letting you specify `image/*`, `video/*`, and `audio/*`, in addition to individual formats like `video/mp4`. Glass natively records video in `video/mp4` format and stores images in `image/jpeg` format, but remember that other Glassware may have their contents in other formats.

The Kinda-Sorta-Not Quite Relationship Glass Has with Audio Formats

While you can list supported audio formats for the `acceptTypes` property, Glass at the moment doesn't allow you to attach audio-only files to timeline items. For instance, you can't put an MP3 on a card and push it to a user. The media player Glass uses assumes video, so in cases where you absolutely need an audio track to be distributed, create a video with a static background and then send the URL as a link in a card or upload the clip to YouTube, and let users stream the file from a server, as we covered in [Chapter 10](#).

If we were creating some Glassware that did photo editing, we might create a contact with the following attributes:

```
{
  "id": "invert",
  "displayName": "Invert Colors",
  "imageUrls": [
    "https://dl.dropboxusercontent.com/u/12019700/invertcolors.png",
  ],
  "acceptTypes": [
    "image/jpeg"
  ]
}
```

Most Glassware will want to set up at least one contact when the user first logs in—at the same time you set up the subscription. This makes a great deal of sense since we'll be told about shares through the callback mechanism.

When someone shares an item with the contact for our Glassware, Glass does a few important things that aren't always obvious:

1. It creates a copy of the item being shared and inserts it at the front of the user's timeline to the right of the home card. If you're watching Glass, you'll see this happen.

2. It gives ownership of the copied item to our Glassware. Remember back to **Chapter 10**—each card is owned by one, and exactly one, Glassware. This is how it makes it ours to control.
3. It then notifies us about this new card via the subscription we have set up for this user.

So using a subscription to the Invert Colors contact as we laid out above, we might get a notification body such as:

```
{
  "collection": "timeline",
  "itemId": "f733a4bd-7a9a-405f-a820-624ad3fe6db5",
  "operation": "INSERT",
  "userToken": "12345678901234567890",
  "verifyToken": "shibboleet",
  "userActions": [
    {
      "type": "SHARE"
    }
  ]
}
```

We'll follow our usual procedure to get the info about the user, verify the `verifyToken`, and get a copy of the new timeline item that has been generated on our way to processing it. The item should look familiar, but with some added touches (we've removed some elements for clarity):

```
{
  "kind": "mirror#timelineItem",
  "id": "f733a4bd-7a9a-405f-a820-624ad3fe6db5",
  "recipients": [
    {
      "kind": "mirror#contact",
      "source": "api:1029263551083",
      "id": "invert",
      "displayName": "Invert Colors",
      "imageUrls": [
        "https://dl.dropboxusercontent.com/u/12019700/invertcolors.png",
      ],
      "acceptTypes": [
        "image/jpeg"
      ]
    }
  ],
  "attachments": [
    {
      "id": "ps:5997490649262926546",
      "contentType": "image/jpeg",
      "contentUrl": "https://www.googleapis.com/mirror/v1/timeline/f733a4bd-7a9a-405f-a820-624ad3fe6db5/attachments/ps:5997490649262926546?alt=media"
    }
  ]
}
```

```
    }
  ],
  "menuItems": [
    { action: "DELETE" }
  ]
}
```

We have a lot of information we'll need from these two objects. We know who is doing this, of course, from the user ID. We know that this is a share from the `Timeline.userActions.type` attribute. We can tell that this is a share that wants to invert the colors from the value of `recipients.id` matching that for the invert contact we created. Finally, we have the `Timeline.attachments.contentUrl` attribute, which gives us a URL we can fetch to get the image itself. Notice that there is no HTML or text attribute for this item.

Before we go and fetch it, however, there are two things we need to pay attention to.

First, if we look at the [Timeline.attachments resource](#), we can see that one of the possible attributes is `isProcessingContent`. We don't have that value set here, so we can assume it is set to `false` and the content of the attachment is ready for us to fetch, but especially when we're dealing with video, we might find it set to `true`. So what do we do? Since we're handling this as part of a job queue, we might just defer the job for a few more seconds and run it again later. When that time comes, we can pull the timeline item again and see if the attachment has been processed yet.

Secondly, when it is time for us to get the content, we still need to authenticate to access the `contentUrl`. Fortunately, most of the libraries we're using provide utility methods to do an authenticated URL fetch, but keep in mind that you will either need to use them to get the data, or you will need to include the auth token yourself as part of the HTTP headers.

Once we have the image, we'll do our color inversion on it so we have a new image. What do we do with it now? We have a number of options.

We could update the original item that was shared with us, replacing the existing attachment with the new image that we just generated. If we do this, we might also want to update the menu items so the user can reshare this image with some other Glassware or send it to a friend or something.

We could just create a new timeline item with this image. This preserves the original image shared with us, but supplements it with the new image. We should probably provide both the DELETE and SHARE menu items, while we're at it.

While both of these might be good, depending on your exact situation, one great alternate solution would be to put both of the images into the same bundle. This would involve creating a new item, just as we suggested, but also including a bundle ID. We would also edit the item shared with us to add that bundle ID as well. Depending on

our needs and desires, we may want to add some text to mention which filter was used or explicitly pick which image is the bundle cover.

This works great if we have just one filter to apply, but what if we have more than one to offer? We could do both color inverts and image rotation. If we're rotating the image, we may need to provide rotation of 90, 180, and 270 degrees, not to mention image flip. And then we can start getting fancy by providing all sorts of filters. Would we need to create different Glassware for each one?

Hopefully you've realized that we don't. Each option can be a contact. We can distinguish which filter is being requested by examining the id on the recipients of the new item shared with us and then route it to the correct function to do the processing.

One thing we should keep in mind, however, is that wearers won't want to swipe between dozens of contacts to determine which filter they want to apply. That would create a very poor experience. We have a couple of approaches toward resolving this dilemma.

Firstly, when they sign up (and at any point they return), we can let them configure exactly which filters they want available. Depending on the settings they choose, we can add and remove contacts representing those filters. Remember that we're designing for Glass here—we need to simplify their options on the device as much as possible and leave the more difficult options to mobile or desktop configuration.

Another option, however, which we might want to use in connection with the first, might be to let them select which filter(s) they want to apply for most of their shares—either individually or layered on each other. So they might create a “Frequent filters” sharing option that takes the picture being shared and creates three new pictures: one with faux HDR treatment applied, one that is brighter, and one that applies both a darker filter as well as a desaturation filter. Each of these new pictures would be saved in the same bundle with the original and labeled with a footer indicating which filter(s) were applied.



You Can't Share to Multiple Entities

In case you were curious, you're not able to share to multiple entities at the same time. You'll need to perform and handle multiple transactions if you want to share a resource with a Google+ Communities group, a Google+ circle, and another Glassware service. This is by design, as not all entities will handle a resource the same way. It also deters senders from being spammy and prevents Glassware producers from finding a sneaky way around using their API quota.

While the footer with filter information might be interesting, it might be even more interesting to have the wearers annotate the picture with some text describing what the picture is. We can do this by allowing them to add a caption to the picture through the

sharingFeatures property. This would change our contact to something like **Figure 11-5**:

```
{
  "id": "invert",
  "displayName": "Invert Colors",
  "imageUrls": [
    "https://dl.dropboxusercontent.com/u/12019700/invertcolors.png",
  ],
  "acceptTypes": [
    "image/jpeg"
  ],
  "sharingFeatures": [
    "ADD_CAPTION"
  ]
}
```

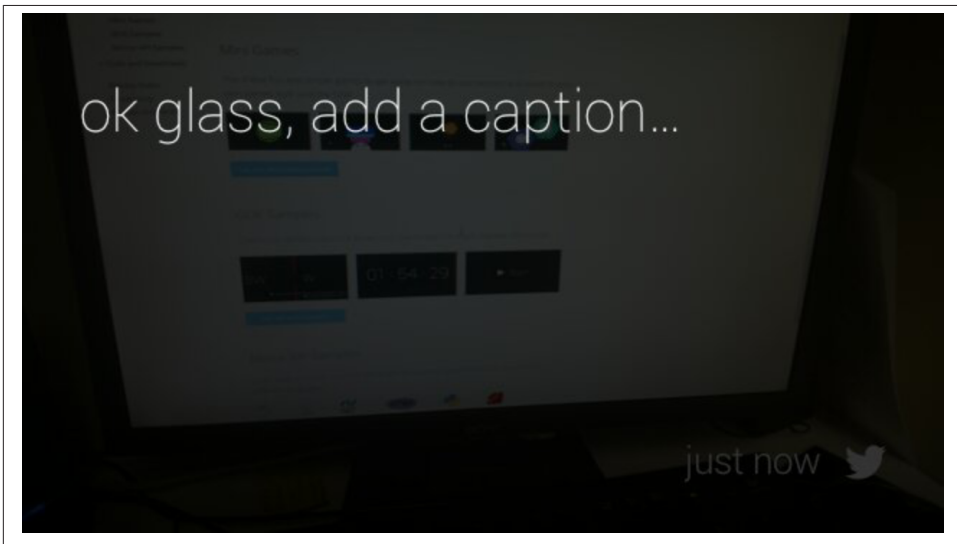


Figure 11-5. Adding a caption to a shared resource

Now, after they share the image, they're prompted to say "OK Glass" and add a caption if they wish. The callback we get is the same, but the timeline item that was inserted now also contains a text attribute with the text that was transcribed:

```
{
  "kind": "mirror#timelineItem",
  "id": "f733a4bd-7a9a-405f-a820-624ad3fe6db5",
  "text": "beach at sunset",
  "recipients": [
    {
      "kind": "mirror#contact",
```

```

    "source": "api:1029263551083",
    "id": "invert",
    "displayName": "Invert Colors",
    "imageUrls": [
      "https://dl.dropboxusercontent.com/u/12019700/invertcolors.png",
    ],
    "acceptTypes": [
      "image/jpeg"
    ]
  }
],
"attachments": [
  {
    "id": "ps:5997490649262926546",
    "contentType": "image/jpeg",
    "contentUrl": "https://www.googleapis.com/mirror/v1/timeline/f733a4bd-7a9a-405f-a820-624ad3fe6db5/attachments/ps:5997490649262926546?alt=media"
  }
],
"menuItems": [
  { action: "DELETE" }
]
}

```

Captioning adds some powerful new options for what you can do with share contacts, but there are a couple of things you should be sure not to do. Don't use this as a way to issue voice commands and don't use this as a way to create a text-only document that is devoid of the media it was shared with. For the former, just create another share contact—the voice system will handle it much more reliably. For the latter, we have voice commands.

Voice Commands

We can take pictures or video with Glass and then share it to our apps, but how do we do the same thing if we want to send a text message? For this, Glass has provided two voice commands off the main menu, and has indicated that additional voice commands are forthcoming.

We can set up contacts so that they are valid targets of the “*Take a note*” command, the “*Post an update*” command, or both. If we think back to our simple text message application, we had *Reply* and *Share* commands...and now we're seeing how to create a new message.

Voice Your Ideas

If you'd like to see other voice commands as part of Glass, Google is taking requests. Make sure your suggestions fill a generic need, and not just a need that your Glassware

alone can fill, and are distinctively pronounced from other commands. Review some examples at the Google's [Voice Command Checklist](#) page, which also has a link to the form to submit your request.

Contacts need to be registered with the voice commands they're willing to accept when we add them; we do this with the `Contacts.acceptCommands` attribute. So if we're setting up our chat client contacts, we might create one such as:

```
{
  "displayName": "Abe",
  "id": "user-prez16",
  "imageUrls": [
    "https://mirror-api-playground.appspot.com/links/lincoln.png"
  ],
  "sharingFeatures": [
    "ADD_CAPTION"
  ],
  "acceptCommands": [
    "POST_AN_UPDATE",
    "ADD_A_NOTE"
  ]
}
```

Did you notice that we didn't explicitly list an `acceptTypes` attribute? In this case, we want any multimedia format to be shared to our pal Abe. A far cry from him sitting at the telegraph station, huh?

If we send him a message by announcing *"OK Glass... Take a note... Abe"* our Glassware will receive a callback with the notification body containing something like:

```
{
  "collection": "timeline",
  "itemId": "09ac1ca0-2de6-40f9-9c20-c1b36f665970",
  "operation": "INSERT",
  "userToken": "12345678901234567890",
  "verifyToken": "shibboleet",
  "userActions": [
    { type: "LAUNCH" }
  ]
}
```

...which is similar to what we've seen before, but with the `userActions.type` of "LAUNCH" indicating this was launched from the home card menu. After we go through our usual verification steps and fetch the item, it might look something like:

```
{
  "kind": "mirror#timelineItem",
  "id": "09ac1ca0-2de6-40f9-9c20-c1b36f665970",
  "text": "got your theater tickets for this evening",
  "created": "1865-04-14T18:06:37.118Z",
}
```

```

"updated": "1865-04-14T18:06:37.118Z",
"displayTime": "1865-04-14T18:06:37.118Z",
"recipients": [
  {
    "displayName": "Abe",
    "id": "user-prez16",
    "imageUrls": [
      "https://mirror-api-playground.appspot.com/links/lincoln.png"
    ],
    "sharingFeatures": [
      "ADD_CAPTION"
    ],
    "acceptCommands": [
      "POST_AN_UPDATE",
      "ADD_A_NOTE"
    ]
  }
]
}

```

This timeline item is pretty basic and pretty much what we've seen before—we have text, but that's about it. So when we process this message we should add it into our database, then update the card so it contains our internal identifier as the `sourceItemid`. We also need to add the creator field, so the REPLY menu command will work correctly...oh, and some menu items such as DELETE while we're at it, too!

We have two notes of caution here, and both should sound somewhat familiar by now. You may be clamoring at this point to submit your application name as a voice command so you can corner the market on its use. Don't bet on this option. Although you have control over the name of the contact used by a voice command, the commands themselves will be set by Google, and the company has indicated that it will be focusing on fairly general trigger phrases using imperative verbs that many apps will be able to use. Remember—*actions, not apps!*

Secondly, you're probably best off treating the text the user has dictated as an opaque message. Don't try to parse it, or prepare to suffer the consequences if you do. In particular, you may notice that when transcribing the spoken word Glass sometimes chooses to use the numeric "2" or other times uses "two," "too," or "to." Android's speech-to-text engine has powerful contextual features that try to discern what form of a homonym is implied by the user, but it's not perfect, so don't bank on it. Being prepared to handle these isn't always fun, and may give you and your user unexpected results, particularly when you are expecting to use the text in a particular context.

On that note, as we've discussed many times—*Glass is all about context*. We'll look a bit further at that context and one spot of context in particular, the user's location, in our next chapter.

Context Is King: Using Location and Other Signals

What are the three most important things about buying a house? Location, location, and location. There may be a lot more relative to building great Glassware, but location is certainly one of the most important cues that we have about what someone wearing Glass will want to know about at any given moment. It is a core principle to the third Noble Truth: *Keep it Relevant*. And the one unifying force that binds not only location data, but also signals relative to time, the user's activity, identity, proximity, scheduled events, and presence of other devices is *context*.

Location isn't the only tool we have at our disposal to help read the mind of someone using our Glassware, but it does serve as a good foundation upon which to base our other contextual inputs. We'll be discussing how Glass directly supports location-aware services, some unexpected ways we can use this information, and how it serves as a model for other context-based data that we may wish to incorporate in our Glassware.

We're going to start with a warning, however, which you'll see echoed a few more times in later chapters. Although all of Glass is evolving, location services are still somewhat immature. You should expect that it will only improve as we move forward, but what we describe here will serve as a cornerstone for what you can expect in the future.

Why Aren't We There Yet?

With location being so important and a huge feature on Android, why does it seem to have so many problems on Glass? We can only speculate, but there are probably a few reasons why. Checking for GPS signals is extremely resource intensive, especially on the battery, so Glass is pretty conservative about how frequently it tries to get your whereabouts when it isn't explicitly navigating. Although there have been some improvements on this front with recent versions of Android, as of the time of this writing Glass was

still using an older Android build—expect some improvements when Glass gets an update.

You might wonder why Glass isn't just using the location services that have been a part of Google Maps for a while now. They may have anticipated some of the changes with Maps that moved location services to Google+ and changed how location tracking was done for mobile. As mobile location tracking continues to shake out from this change, we might expect Glass to adapt as well.

Enabling Location

You may be surprised to learn that you actually need to do something to enable location services with Glass. After all, if our users expect us to read their minds, and where they are is a big factor to this, wouldn't it just be assumed that we should know where they are? Certainly location services on Android tend to work behind the scenes without the user having to be very involved.

As we explored early on, Glass has had to tread lightly when it comes to things that could be seen as invading a person's privacy...and your location is certainly one of the things that a person may most wish to guard. While it is easy to turn your location on and off on your phone, you don't have the same kinds of controls with Glass. So any application that wishes to know where you are must explicitly ask for permission to find out.

Can you guess how it asks for the rights to use geodata? The very same way our Mirror API-based Glassware asks for permission for anything—it has its own OAuth scope that indicates we want to view a user's location. If you've forgotten what an OAuth scope is, take a quick look again at [Chapter 8](#). (Don't worry, we won't notice.)

You should already be requesting the `https://www.googleapis.com/auth/glass.timeline` scope, and probably at least one other scope such as profile so you can get the user's information. To these we will need to add the `https://www.googleapis.com/auth/glass.location` scope, which will instruct Glass to subscribe to location updates, provide location information along with timeline events, and give us the ability to make queries for a user's location at any specific time.

Where Do You Think I Am?

You might remember from [Chapter 10](#) how when we talked about subscribing to timeline events we mentioned that we would talk about subscribing to location events in this chapter? The time has come! And the subscription itself is amazingly simple. For the `Subscriptions.collection` property, we can specify "locations" to get alerted whenever there is new location information for our user.

When the information is available, we'll get a callback similar to ones we've seen for timeline events:

```
{
  "collection": "locations",
  "itemId": "latest",
  "operation": "UPDATE",
  "userToken": "12345678901234567890",
  "verifyToken": "shibboleet"
}
```

We might be tempted to use the same logic to process this callback as we did for timelines, but there are a few things we need to do differently. Most notably, we'll need to call a different method to get the location event than we did to get the timeline item. Instead of calling `Timeline.get` with the ID specified by `itemId`, we'll be calling `Locations.get`.

We know...it's going to be difficult to remember the difference between the two. We think you're up to the task, however.

Calling `Locations.get` returns some JSON not unlike this fragment:

```
{
  "kind": "mirror#location",
  "timestamp": "2014-02-02T17:57:06.770Z",
  "latitude": 37.4038194,
  "longitude": -122.081267,
  "accuracy": 22
}
```

Referencing [the Locations documentation](#), we find the fields are pretty straightforward:

`timestamp`

When the user was recorded at this location. Note that although we're requesting the "latest" position, it doesn't mean it is very recent. We'll discuss this a bit more later on.

`latitude`

A floating-point number indicating degrees north of the equator if the number is positive or south of the equator if the number is negative.

`longitude`

Similarly, this is a floating-point value indicating degrees east of the Prime Meridian (if it is positive) or west of the Prime Meridian (if negative).

`accuracy`

The best estimate of accuracy, in meters, of this reading at the time it was taken.

You Want It WHEN???

How often will we receive a location change event? Currently, no more frequently than every 10 minutes. That seems like a lot, but we can't even assume that we'll get it that frequently all the time. For a variety of reasons that may be out of our control, Glass may not update our location on the schedule we expect.

Fortunately, Google both knows about these issues and understands how important location is. We can expect improvements in frequency as Glass matures. In the meantime, we can start working with what we have with the knowledge that it will only get better.

Getting these periodic updates is a great way to judge when and what to deliver to our users. Are they still at work, are they getting close to quitting time, and does traffic in their area look bad? Perhaps now might be a good time to warn them. Have they not moved very far since the last time we got their location? Sounds like they don't need new driving reminders about where the next gas station is.

That last suggestion raises a good question. We know where a person is now—can we find out where they were recently? Well...yes and no. Yes, there is the `Timeline.list` method that will let you get the most recent timeline events. Unfortunately, at the moment it only ever returns one event—the event with the ID of “latest.” Expect this to change in the future as well.

Location change events are all well and good, but can we tie them to events we're already familiar with—things shared with us or new notes that are taken? This is easier than you might think.

Location as Part of Timeline Events

It turns out that once you request location information via the `glass.location` scope, you'll get this information attached to every timeline item shared with your Glassware.

You may be encouraged by this—if your user is sharing lots of things with your contacts, your Glassware will get lots of location updates, and this will get past the problems of updates only every 10 minutes. Your mind is probably racing with a way to convince people to check in with you specifically so you can get the most accurate location possible.

Sorry to dash your hopes, but location is still only updated every 10 minutes, and this doesn't offer you a way around this. The location attached to a timeline item is just the latest location at the time the item is shared. If your user is sharing an item that was created significantly earlier, the location won't match the place where the timeline event was originally generated.

So why is this included with a timeline item when we could just request it? It saves us a function call (and thus some of our quota for our daily Mirror API calls), and if we're requesting location information at all, it makes sense that we would find it useful when a user shares something.

What Happened to Geotagging Images?

One of the issues that popped up in the early days of Glass was noting how images captured with the camera didn't get EXIF location metadata. Once uploaded to Google+ via Instant Upload the location could be entered, but the manual nature of having to do so irked Explorers.

Geotagging, now a staple of photography in the Social Era, wasn't available for Glass when either tethered to a smartphone via the MyGlass app, or even when on WiFi. This seems tied to the general location issues we've encountered in this chapter. As with everything else having to do with location, it seems safe to assume that this will eventually be resolved.

What sort of uses can we put the location to for a timeline event? Attaching the share to a marker on a map is the most obvious, and there is already some great Glassware that illustrates some of these concepts, but what else is location and a share good for?

It may seem flippant, but it all depends on what your Glassware needs to do. Keep in mind that locations can determine many things beyond the obvious point on a map. Your location may determine what language the local citizens are speaking or what currency they're using, and if your Glassware is involved in translation or currency conversion, you may be able to take advantage of this information. If you know a person's location and the current time, you may be able to determine things such as the current weather, the current tides, the sunset and sunrise times, or how bright it is outside. Timeline events are all marked in UTC time, but the event location may let you convert that into the local time because you'll know the user's current time zone.

Setting Things Straight

This previous concept of using the user's location to determine the current time zone is a good one, but it has a flaw: to get the time zone for a transaction, you'll take the location and probably forward it to a time zone service—for every single message that comes in. That can be a lot of extra server roundtrips and hits against your API quota. Since Glass already has this information, wouldn't it be great if it could provide it for us? It turns out that it can.

The Mirror API provides us with the `Settings.get` method to retrieve this sort of information. For certain key values (Google calls them "IDs"), calling this method will

return the value the user's device is set for. The Settings resource accesses data that the user configures in MyGlass, which lives in the cloud, is accessible through the MyGlass mobile app and on the Web, and is exposed to both the Mirror API and the GDK.

To get the current time zone, we can request it with the `timezone` ID using a URL such as this (and providing the correct auth token):

```
https://www.googleapis.com/mirror/v1/settings/timezone
```

...which might return a JSON object looking something like:

```
{
  "kind": "mirror#setting",
  "value": "America/New_York"
}
```

(Clearly this is the example from Allen. Jason's time zone, given that he lives in a place that's 17 hours ahead of Eastern Standard Time, is literally more futuristic.)

With this, the time zone database that most languages support can convert the UTC time Glass provides us into a time format more useful to the user.

Location Becomes Localization

But just knowing the time zone isn't necessarily enough to format a date and time correctly. Different countries and languages have different styles—not to mention different ways of saying the exact same thing! *Localization*, which displays content such as text, images, and videos in a format that's appropriate for a particular region, and has always been a big part of Android and web development, is an effective solution for this problem. By supporting multiple languages, your Glassware broadens the reach of your products and adds great value. This can be applied not only to content within timeline cards, but also to menu items and voice commands.

MyGlass Speaks Your Language

Did you know Glass displays a different voice command prompt for capturing images depending on what the user has set as her default language in MyGlass? It's true! For users having "English (United States)" as their preference, the familiar "Take a picture" will appear. But for our friends over in the UK, "Take a photo" will be accepted when they have "English (United Kingdom)" as their language (Figure 12-1).

Additionally, time formats and spelling also adjust—for example, the UK setting applies the 24-hour clock to the home card and renders the "Recognize" command as *Recognise*."

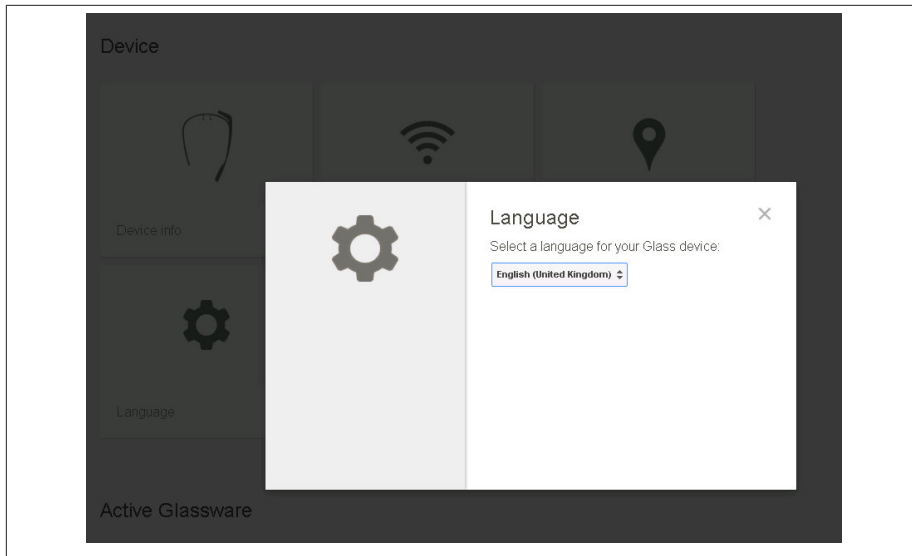


Figure 12-1. The Language setting dialog in MyGlass

Fortunately, the Mirror API provides us with a locale setting to get this information. So we can ask for the following:

```
https://www.googleapis.com/mirror/v1/settings/timezone
```

...and get back a standard locale string as part of the response:

```
{
  "kind": "mirror#setting",
  "value": "en-US"
}
```

Subscribing to Locale Changes Saves API Calls

These fragments of data are useful to get a user's current preference when they set up your Glassware, but they don't really make things easier in the long run. We still might need to issue another API call periodically to get the time zone in case the user is traveling. (It'd be far less likely they'd change their locale, at least.) It won't change that often, but it would be helpful to know when it does.

Fortunately, we can leverage the *Subscription* object that is already part of the Mirror API to do this. Instead of subscribing to the Timeline collection, we can subscribe to the **Settings** collection. This will send us notifications when any of the settings change. If you need a reminder about how subscriptions and notifications work, check out **Chapter 10** on that topic again. (And you may be wondering if you can do this sort of

thing for location—take a look at the [documentation about Subscription](#) and you may find what you’re looking for, but remember some of our caveats earlier.)

Your backend can then use whatever templating framework you want (Rails, Django, ASP.NET, Node.js, etc.) to serve the appropriate resource string, which is then pushed to the Mirror API and delivered to your users.

It’s a very good idea to proactively have your Glassware support multiple languages, even though as we write this the only languages available in MyGlass are the US and UK flavors of English (or is it flavours?). When you submit your Glassware for review, you’ll get helpful pointers from Google on how to best implement localization, if you require it.

You can find more tips from the Glass team on best practices for localizing your Glassware for both [the Mirror API](#) and [the GDK](#).

Other Contextual Signals

In all of these cases, location plays an important contextual role—but it’s not the only type of signal you can use in your applications. This approach emphasizes *the personal network* consisting of the user’s surroundings, the presence of others, what the user is doing at that moment, times and dates, and what’s happening both in the user’s vicinity or based on events they may have through other platforms like calendar or to-do applications. These signals tap a constant stream of inputs swirling around the user at any given moment.

A fantastic practical example of a contextual action is what a buddy of ours, E. John Feig, did with [Talkray](#), a calling and messaging app with Android Wear support. John and his team at TiKL created an auto-reply feature using *activity recognition*, based on the user’s real-life action. If a user receives a message and then chooses the Auto-reply busy button, the device running Talkray checks to see how fast the user is moving; if it detects that the user has a certain velocity, it’ll send “I’ll get back to you later, I’m driving” as a canned response. It’s the perfect contextual microinteraction!

Let this be a springboard for how you might use contextual signals in your own wearable applications.

Android Wear and the Personal Network

The **Android Wear SDK** is built in such a way so it detects all the devices the user is signed into under the same Google account—including Glass. This means it can read data from, push notifications to, and sync information between all connected devices within a user's network seamlessly.

Context and the Future

You're probably looking incredulously at this chapter—that's it? That's all there is to using location with Glass? Well, no. But that's all there is to say about how to use it. The concepts are very simple and build on the things we already know how to do with Glass. The power will be in how you use it.

Location provides an example of how we can use other information in context. We will often need the user's permission to get access to additional sources of data. Some information we will want to act on as it changes and is delivered to us, while other bits of data are only relevant when the user shares something with us. We need to be prepared to handle both to make great contextual Glassware. Location is an incredibly valuable signal that contributes to the overall goal of understanding the users—where they are, what places and things are around them, and who is nearby—in addition to who they are, what their interests are, what social connections they have, and what they're currently doing. But, it's still just one piece of a very large puzzle.

What other sorts of information signals should we be thinking about as Glassware architects? Consider a service delivering information about Major League Baseball—clearly we'll want in-game events delivered so we can pass them on to our users, which is a pretty straightforward scenario, seeing as how America's pastime is ripe with infinite data with insatiable demand. What other contextual signals might we need to craft messages with high utility?

What if we could see that they were having an afternoon meal with a friend, and the profile information for that companion indicated they were a devout fan of the local ballclub? We might send schedules, scores, and stats about their team before (or during) the lunch meeting to provide conversation fodder. Or maybe information from within the division to which the team belongs, which means including regional teams. Or perhaps we might respond to a picture of a sports logo by providing some trivia about that team.

This is Think for Glass in action, as we're leaning hard on relevance to life experience, which the user will surely appreciate.

Continuing our example, what if we know that our user is planning an airplane flight, perhaps because we have access to her calendar or because she told us explicitly. Once we have this information we might use her current location to determine which side of the flight she is on. If she hasn't taken off yet, we might give her gate information and flight status. But if we know she is at the destination airport, we might send her information about where the baggage claim is and let her call her ride to let them know she is on the ground.

Our application needs to be smart enough to know the difference and to have all the information the user needs (departure gate, baggage claim, and ride contact phone—all of which come from different sources) when she needs it. At the same time, if she is still at the source location when the flight lands, we may need to apply some extra logic to figure out if she missed her original flight, or if she is just waiting to make sure a family member has reached their destination safely.

In some ways, this discussion about data sources and context is a fitting conclusion to our chapter on location and also to our section about working with the Mirror API. Glass isn't about what Google gives you—it is about what you do with those tools. You're bringing information to the table that your users are interested in. Really think about how to blend the idea of enriching the user's experience with location data to the Glass paradigm. Make the data and the ways it is delivered to the user part of the experience, not just lumping in a mapping feature.

The Mirror API is all about how you deliver that information to your users and what you're giving them to interact with it. What we'll be talking about going forward are other places with Glass you may wish to explore.

The Google Glass ecosystem includes the ability to build client applications to be installed on Glass that interface directly with the system, doing things that largely aren't available through services dependent on the cloud-resident Mirror API, including off-line capabilities, sensor access, and real-time interactivity. **The Glass Development Kit (GDK)** is a library that extends the larger Android SDK by letting developers write full Android applications in Java and use associated tools for debugging, crash reporting, and analysis.

This chapter gives you an overview of the GDK, its capabilities, the distinct UI elements it provides for Glass, and design patterns for working with each type of UI element—the right way.

Mike DiGiovanni, an insatiably curious coder from New Jersey whose early add-ons for Glass included Winky, which later became the system wink-to-take-a-picture gesture, **enthusiastically proclaimed** about the GDK, “Native Glass development is, by far, the most exciting development that I’ve done in years.” Many throughout the Glass community happily echoed this sentiment.

Installed Apps Running on Glass

It's important to note here that *native development on Glass doesn't change the core goals of the product—you're still catering to microinteractions in a head-mounted display*. This is key to being able to Think for Glass. The fundamentals of the Think for Glass philosophy don't change at all; the GDK just provides a new set of tools at your disposal for building the perfect stage for your idea and enhancing interactivity in your own custom ways.

Here's the billion-dollar secret about native Glassware that many in the media (including many covering the technology beat) got wrong: it didn't just magically appear when the GDK was unveiled at Google's San Francisco office at a hackathon before a group of

anxious coders and reporters. If you use Glass freshly unboxed, without installing anything additional, you're still using a lot of native Glassware all the time, presented in a couple of different ways. Several components of the core Glass firmware are applications providing an experience that completely honors the timeline model, in addition to living outside of it. In a twist of irony, GDK apps enhance the default wearable experience—and at the same time go rogue against it.



Here's some terminology for those of you who are sticklers to detail before you go running us out of town on Amazon. Technically speaking, “native” Android programming refers to the **Android Native Development Kit**, an SDK allowing C/C++ libraries to be used within Java-based projects. The moniker has become muddled somewhat in mainstream use in recent years to the point that “native” now implies “anything mobile” and “not on the Web.”

We'll do our best to not let this get out of hand in this chapter. And since we won't be covering the NDK, just know that in the context of our discussion on the GDK we're using “native” to refer to the writing of Java code for Glass and apps that run locally on the wearable, not Glassware using the Mirror API.

This type of Glassware doesn't necessarily force you to mess with OAuth or go to the cloud to do anything—everything's running on the device as a compiled program. Let this inspire you as a developer! There's lots of room to create here.

Rather than pit the two frameworks against each other in a programmer's holy war, the GDK and Mirror API productively share space, with both using the timeline as a staging environment. They tightly integrate as partners, not rivals, in helping deliver a very convenient user experience. However, *your application code and the timeline run in completely separate processes.*

This is what we fully respect as the sheer brilliance of the Glass ecosystem—there's great blurring between what's running RESTfully and what's a locally executing process. This is a good thing! The trained eye (which by this point in the book you certainly have) can quickly pick out an installed app separate from a cloud-based service, but the timeline is what stitches them together into a coherent, unified interface.

This produces great choice and great opportunity for you as a Glassware designer/developer. You may have an idea germinating in your brain that hinges on being able to read the user's rate of acceleration and bearing at any precise moment in time. Or you might be thinking about presenting data as some cool animation or complex 3D diagram that adapts to changes in real time. Perhaps you've got a can't-miss concept involving an innovative use of the Glass camera, or you demand extremely low-latency communications and system responsiveness. Maybe you just require some custom functionality that the Mirror API just doesn't provide. There's the possibility your de-

velopment team may have a vast legacy library written for an existing app that does cool things you'd like to apply over the wearable paradigm. Or, you might want to make an entire universe unto itself and give people a new way of experiencing the world.

Whereas the Mirror API is an agile platform that abstracts a lot of the gory details of system programming so that you can rapidly iterate and turn your ideas into usable products, the GDK gives you much more control over the exact implementation of what you're trying to do. The trade-off, of course, is that the craft of building applications that run on the device isn't exactly a job most of us mere mortals can do during our lunch hours. It's a very complex and involved process requiring unit testing, debugging, refactoring, and deployment; and having total control means sacrificing speedy development. The benefit to you if you've done Android development previously is that you'll be using the same skills and APIs to build Glass apps.

What Is the GDK?

The Glass Development Kit is an additional library to the Android SDK that lets you code up features specific to Glass. It's an additional Java archive (JAR) you bundle with your Android projects that gives you access to classes to do Glass programming for programmatic control of Glass UX elements like voice recognition, gestures, and location data.

The GDK extends the core Android stack, which lets you work with UI widgets and layouts, along with the core hardware components like sensors and GPS, and managing activities, services, and broadcast receivers. Native programming gives you more granular dominion over user interface elements, as it is immediately responsive to user input.

The basic platform features several components from the Android SDK, which work, more or less, as they do on stock smartphones and tablets (for more on Android development, check out *Learning Android, 2nd Edition*, by Marko Gargenta and Masumi Nakamura:

Location provider

The user's position on Earth.

Camera intent

Control the camera capturing for photos and video.

Recognizer intent

The system's speech-to-text feature.

Options menu

Additional system controls.

The GDK add-on includes several components to let you programmatically work with Glass UI element:

CardBuilder

Manipulate, arrange, and style atomic representations of data in timeline items.

CardScrollView

Manage navigation over a collection of static timeline items.

Live cards

Render and update content rapidly within cards that sit left of the home screen.

Voice triggers

Insert new voice commands into the “OK Glass...” menu to launch apps.

GestureDetector

Capture user movement and trackpad swipes/taps.

The Android SDK and the GDK make a powerful team for an integrated development environment (IDE). The add-on has tight integration with Eclipse as well as the newer **Android Studio**. Work is also ongoing by the community to turn **App Inventor**, the WYSIWYG application creation tool built by Google and now maintained by MIT, into a utility to build Glassware.

While you’re free to build practically whatever your imagination conjures up with the GDK, this also means respecting some limitations. Unlike the Mirror API, you won’t be able to pick any programming language you like—you’re doing Android development, which means you’ll be using Java. (More specifically, you’ll be working with the subset of Java for the Android SDK.)

For the most current instructions on setting up a development environment to build Glassware with your preferred development environment, review the **GDK Quick Start** section of the Glass Developers documentation.

Drawing and Animation

The drawing interface for the GDK is still OpenGL, meaning you can fully draw 2D and 3D graphics directly onto cards using an instance of **the Android canvas** as a drawing surface. The rather antiquated GDI classes are typically only used these days for doing simple shapes and polygons. `RemoteViews` can only render layouts, but rendering directly gives you the full range of Android’s drawing capabilities.

The **Mini Games** pack of Glassware uses several physics and graphics libraries to achieve fluid movement in both 2D and 3D environments.

Because the Mirror API at the time of this writing doesn’t support JavaScript, you’re unable to do animation loops with that framework. So if you’re looking at doing a high-impact game (which may still be low-intrusion, depending on your implementation), you’re going to need to work natively.

The Glass team has a [fantastic code repo on GitHub](#) that shows how to work with OpenGL for things like shaders and textures and other fun topics like that. It's definitely worth perusing.

And what's perhaps most encouraging if you're coming over to Glassware development from old-school Android coding—you don't just have to use the GDK namespaces. The Glass team shared this gem with the community [on Stack Overflow](#):

The API surface of GDK Glassware is not limited to the classes contained in the GDK Add-on. The GDK Add-on merely closes the gaps between the Android SDK and features that are unique to Glass. This means, in general, given a problem that isn't covered by the GDK library directly, just attempt the Android solution.

How the GDK Differs from the Mirror API

The GDK empowers you to make distinct functionality part of your mobile applications. While it shares the timeline metaphor as a user interface with the Mirror API, writing native code for Glass gives you the ability to do so at a more granular level. Since everything happens locally on the device, you won't necessarily need to ping the cloud each and every time you need to do something.

You're able to reuse custom components within your project with the Android NDK, releasing you of the burden of having to port entire libraries over to Java. This can't be easily done with the Mirror API. Additionally, writing locally running programs doesn't let you enjoy Glass sync with its intuitive and self-managed push backend, using Google Cloud Messaging.

The GDK lets you explore three specific areas that aren't available with the Mirror API:

Offline access

Since native apps don't have to rely on constant connectivity, you're fully capable of doing all processing and data storage locally. Your apps could be exclusively offline and not talk to the outside world; or they could provide an option to sync on-demand or periodically (like how Android does for your contacts, calendar, Chrome, Gmail, and other app data), or apply offline support when losing connectivity and persist data to a local store or cache and then sync when a network connection returns (like Google Drive does with its files).

Real-time interactivity

Since a network connection isn't required, you'll be able to capture and handle user events in true real time without latency or the need for server roundtrips. You'll be able to respond to changes in microseconds.

Sensor access

Directly interacting with hardware is at the heart of native mobile programming. The GDK lets you process readings as users move or their environment changes. [Google's documentation](#) cites the following sensors as programmatically available on Glass:

- `Sensor.TYPE_ACCELEROMETER`: Rate of movement, including gravity
- `Sensor.TYPE_GRAVITY`: Influence of gravity on the device
- `Sensor.TYPE_GYROSCOPE`: Rate of rotation
- `Sensor.TYPE_LIGHT`: Amount of ambient light around the device
- `Sensor.TYPE_LINEAR_ACCELERATION`: Three-dimensional vector for acceleration along each axis, excluding gravity
- `Sensor.TYPE_MAGNETIC_FIELD`: Proximity of the device in relation to the user
- `Sensor.TYPE_ROTATION_VECTOR`: Orientation to measure tilt levels

You can find out more about Android sensors and the APIs for interacting with them in the [Android Developers documentation](#). The Android Open Source Project also includes several software sensors, so keep current with the latest information in the docs.

Aside from the exclusive features you can build using the GDK, a few advantages available to you include:

Performance of installed apps on Glass hardware

As of the time of this writing, GDK Glassware is built for Android 4.4.2 as the target version (API Level 19 or higher), so be mindful that while the specs for Glass are comparable to a mid-range smartphone (see [Chapter 3](#) for details), high-end and processing-intensive operations should be done with care. The Glass firmware manages multiple operations incredibly efficiently and does a good job of self-healing from stalls and crashes. Glass is a very capable computer.

Mirroring

Think about how Glassware may leverage modern releases of Android's support for Miracast, which is built on top of WiFi Direct and bypasses the need for devices to share the same router for use in mirroring or second-screen experience apps to other displays. Glass might be used as a remote control much in the same way that Android phones and tablets can fling YouTube content to a monitor running Android TV.

Frontend presentation

In addition to being able to generate cards with text and images and insert them into the timeline like the Mirror API, the GDK provides you with a range of unique

presentation elements that work with the Glass experience to give data new depth and create rich UIs using standard Android widgets as well as drawing custom graphics.



Chromecast, Google's amazing diminutive streaming dongle, wasn't even out for a full day before we—and tons of other people on Google+—began speculating about and ultimately demanding how we might be able to use it in combination with Glass as a live screen-sharing medium. The screencast feature in the MyGlass mobile app is fantastic but limited to the device running it, and debugging tools like [Android Screen Monitor](#) connected to Glass via USB to get the Glass UI on larger displays is a hacky setup. We'd like to see something that works out of the box and send our timelines directly to an HDTV via Chromecast. Or even in huge theater and convention center displays or digital billboards. It's too obvious not to do.

However, GDK development doesn't have the full features of the Android application ecosystem just yet. As we write this, Glass development with the GDK doesn't include Google Play Services. If you try to force the issue and bundle it or try to use the various features it provides, your app will likely break. We hope to see this change soon, but if you manage an app and are translating it for a wearable audience, it'll behoove you to know what components aren't fully supported.

See [“Porting Existing Apps to Glass: DON'T” on page 288](#) for more on this topic.

User Interface Elements of GDK Apps

When dealing with installed applications on Glass, you've got two UI stages on which your apps can perform: *live cards* and *immersions*.

Live Cards

Live cards involve content that's updated frequently. While static cards from the Mirror API can be modified if need be, the content in live cards is expected to be altered more rapidly, even at a rate of several times per second. They live to the left of the home card (with the clock), and represent those events that are currently happening or will happen in the future.

This information may be from the Internet, like a ballot tally during an election; or from your device, like a sensor reading. Because they're ongoing tasks, they run in a process separate from other cards, and in that respect are very much like the Android widgets that run on the home card of tablets and smartphones. The Timer, Stopwatch, and Compass apps are examples of live cards. So is the Settings card telling you how much charge your battery has, updating its content in response to events (in this case, the

change in how much charge you have left). You can even have more than one live card running—as exhibited through all of your Google Now cards, all independently updating their contained information.

Another way live cards differ from static cards is that live cards are still bound to the rules of the timeline, but if the user swipes in either horizontal direction and moves away from a live card it continues to run even if not visible. When the user swipes back to the left of the home card or Glass wakes from standby mode, the live card is still right there, doing its thing. It doesn't have to have focus to continue working. There are lots of neat ways you can use this type of card, since it doesn't have to stay resident and doesn't need the display to be actively on to work.

Live cards exist as Android services rather than statically generated cards or their own activities, and generally are used in situations where there's not much input involved from the user. Examples like the Timer app allow a user to set a timer and then leave it alone as it animates a countdown. A major difference between static cards and live cards is that should Glass reboot, lose power, or run low on resources, live cards will be removed. The [Ongoing Task pattern](#) documentation that Google published shows how to use Android services running in the background with handlers to update live card content.



Don't Skimp on “Stop”

Google's review process requires that all live cards include a *Stop* command to properly dismiss them and kill their underlying service. This isn't an idiom or pattern the Glass team takes lightly—if you submit Glassware for review and it doesn't include *Stop*, your Glassware won't be approved.

However, do make sure to kill processes if they run long or the wearer forgets about them. Being “live,” they should optimally have a shelf life and expire at some point. While modal dialogs and Android toasts don't fit the Glass UX, informing the user that a live card has been running for an exceptionally long time and should be dismissed if it's not needed (fittingly, with a static card and accompanying Glass alert) might be a good way to ensure your app doesn't develop a bad reputation as a battery hog, and possibly receive negative ratings.

Two Flavors of Live Cards

How fast is fast? As a Glassware developer, you normally make the design decision to work with live cards over static cards based on the need to update a timeline item's content within a few seconds. But it goes deeper than that. You've then got to determine how rapidly you'll be replacing what the user sees or gets notified about, and the GDK gives you some options in determining how you control the on-screen content for live

cards. Live card content can be generated either with *low-frequency rendering* or *high-frequency rendering*.

Low-frequency rendering uses a **RemoteViews** object to inflate and set values in `View`-inherited UI elements every few seconds or a couple of times an hour like a change in a sports score, a weather update, or a stock price tracker. This method is much easier to configure and requires only a few lines of code. In contrast, situations where high-frequency rendering comes into play involve direct drawing to the surface on the order of tens if not dozens of times per second, like an animation loop, or displaying real-time accelerometer readings as the user moves her head. This method requires more setup work to handle the rendering logic.

Live cards only support the following layouts and views within the Android SDK:

Layouts

- `FrameLayout`
- `LinearLayout`
- `RelativeLayout`
- `GridLayout`

Views

- `AdapterViewFlipper`
- `AnalogClock`
- `Button`
- `Chronometer`
- `GridView`
- `ImageButton`
- `ImageView`
- `ListView`
- `ProgressBar`
- `StackView`
- `TextView`
- `ViewFlipper`

See Google's documentation for full examples of each type of rendering, and make sure you choose that which is most appropriate for the content you're working with and the experience you're aiming to deliver.

Android services typically drive the lifecycle of live cards, given that they still need to run if the user navigates away from them or Glass falls asleep after nonuses. GDK Glassware can be run as a service that starts when Glass first boots up—if you’re familiar with Android development already, you’ll enjoy the ability to create background processes.

Starting Services at Bootup?

Because Android services that run constantly can be battery killers, the practice is discouraged for third-party Glassware publishers. But developers will no doubt still want to start a service for a live card when Glass is first turned on and have them run as long as the device is on. Consider, for example, the system software that controls the various Google Now cards—each item is a live card and the service driving them spins up when you first boot Glass, running for the duration of your session, and even while the device is suspended when Glass is idle or off your head.

Unfortunately, this level of control isn’t available to external developers at the moment, as the rules for approval of your Glassware for listing in MyGlass require you to have a *Stop* command. Still, the best part of Glassware review (which we detail in this book’s final chapter) is that you get to work with Google directly on implementing your idea in the best way possible.

If you have a genuine use case that demands such an experience, you’ll get help for the best architecture to use to make it come to life—even if that isn’t a perpetual background service.

Good luck!

Two of the most widely used purposes for services are to schedule time-specific or periodical publishing of timeline cards, or to handle long-running operations asynchronously in a dedicated thread like broadcasting system-wide alerts under certain conditions based on sensor input, or downloading assets from the Internet. A funny example of this based on the user’s context might be generating a warning card when the user looks straight up during daylight hours, warning him, “Hey, stupid! Don’t look directly into the sun!” which is the type of contextual computing experience that being able to Think for Glass is all about.

Lastly, it’s important to note that live cards aren’t just for dynamic content—they also work fantastic as dynamic *containers* for data. A perfect example of this is **ViewTube for Glass**, which lets you enjoy content from YouTube without having the timeline stay in a fixed position while the video is in playback. You can search for videos, then stream the clip and swipe away from the ViewTube card (essentially an embedded media player), and do other things or let Glass go to sleep while the content continues to play.

It's a tremendous user experience to not force the user to remain on a certain location on the timeline or be within the Glass browser. If you wanted to adjust the volume, you could swipe to the Settings bundle and make your tweaks there. Pandora's Glassware achieves the same effect while including a menu item within the live card to control the volume level.

In both cases, the live card gave the user freedom while still updating the live card's content—ViewTube with matching search results, Pandora with the next available track.

The docs on managing **ongoing tasks** with live cards give you a great springboard for getting productive quickly, so do read up.

Immersion

This is the type of UX that most people will immediately think of when they first hear the term “Glass apps.” Immersion are dedicated Android components, and as such are the most complex type of element for GDK apps because they give you the most autonomy. You've got carte blanche as far as the experience you want to create, which can be separate from the user's timeline altogether or directly integrated with it for a seamless transition. You also have less constraints on the user input controls than you do with other UI types—if you want to require a Bluetooth keyboard, you can do that, too.

Immersion are also the most challenging type of interface element to build. And true to their namesake they demand the most prolonged attention from their audience, which means the display could stay active for the duration of their use—meaning they can potentially devour CPU cycles and be battery killers if you're not careful. You don't want Glass to run hot. Consider the system application “Show the viewfinder.” It activates the camera on a persistent viewable surface with a framing tool...which if left running for more than 20 seconds starts to notably heat up Glass. It's a really good utility but needs to be used responsibly.

Immersion and their role in the Glass ecosystem are also misunderstood. Developers often think that an immersion *has* to be stared at for long periods of time, like with games or apps using the camera, when there are instances where this isn't the case at all. The navigation system application displays a map contextually accurate to the user by using location provider features, animates the position on the map as the user moves about, and reads turn-by-turn driving directions aloud. But, the app doesn't force the user to stare at it—it auto-dims the display if there's no active use after a few seconds, like the timeline does.

“Oy, My Aching Battery...”

We're tragically projecting that the majority of the crop of native apps built for Glass will unnecessarily be built as immersion, and that this will largely lead to negative

performance based on the processing power, memory, and battery charge they require. Remember, having the Glass projection unit active consumes the charge on your battery more dramatically than static and live cards, so unless you're developing an app where the display needs to stay on with information constantly resident, let Glass naturally go dormant.

An app like **GolfSight** allows the user to sleep Glass without dismissing the app running in memory. This is good program design to emulate.

So as far as our projection, we hope we're wrong. Someone please prove us wrong.

In contrast to live cards, immersions are built to harvest as much input as they can. You'll want to use them as the surface for UIs like games where there is a high degree of interaction with the app. This is the main reason they live separately from the timeline—when interacting with items on the timeline, input you give Glass typically directs the navigation, sending you forward or backward through your items, or selecting menu items or launching new Glassware. Immersions are independent so that they can capture all of those gestures and use them as input for a completely different interface experience.

An immersive app exists as its own activity, and can capture trackpad and gesture events completely differently than Glass does for other services. Still, while applications like games may create custom input control elements, it's best practice to follow the idiomatic uses for swipes and taps (i.e., swipe forward to move left, swipe back to move back, tap to select, etc.) so users don't get confused with their wider use. Since they exist off the timeline, immersions also don't necessarily leave the breadcrumb trail of cards that other app elements might. This can work for or against you—you may want to create an archive of a user's search activity against your recipe database for historical purposes and quick reference, for example.

However, because they live outside of the friendly confines of the timeline, immersions also lack some of the conventions that other UI elements have. Namely, immersions naturally don't dim the display after a certain period of time. They stay on until dismissed or their containing application is exited, hence their reputation for draining the amount of available charge. They also don't have any concept of bundles, so you're on your own when transitioning between screens and launching activities.

And just like live cards have to have a *Stop* action, Google enforces a design rule for immersions: they all have to support the familiar downward swipe gesture on the trackpad to dismiss the activity, which kills it in memory and exits back to the timeline.

One word of caution: while immersions are very good and very powerful, they don't give you license to violate the Noble Truths for Glassware design as described in **Chapter 5**. The Glass "Get directions" application is a good example of this: the display promotes a glanceable, quick reference interface for navigation, but it takes over the system the same way an immersive screen does.

You can learn more about proper **immersion development patterns** from Google’s documentation. **Table 13-1** summarizes the characteristics of each type of native UI element.

Table 13-1. Components of GDK Glassware

Elements	Features	Example
Live cards	Content constantly drawn on cards, low-frequency updates, high-frequency updates	Animation, timers, location
Immersion	Environments separate from the timeline, exist as their own activities, handle touchpad taps/swipes and head gesture events	Games, camera invocation

As a final difference between live cards and immersions, neither allows programmatic control for wake locks in order to control how the device goes to sleep. Live cards are managed by the Glass system to suspend naturally, and immersions stay on. It’s that simple.

More Tools for Rapid Design

When designing your GDK Glassware, a free tool you’ll find indispensable is the **Glassware Flow Designer** as seen in **Figure 13-1**, a web-based flowchart creator that Google built to help you rapidly lay out and visualize how your app will look and behave. Like the Mirror API Playground, it lets you apply Glass-formatted templates to get a snapshot of how content will appear in cards, get a bird’s-eye view of how intuitive your usability process is, and possibly identify areas where things can be streamlined. You’ll be able to quickly spot redundant menu items or rearrange their order based on which ones will be used most frequently.

Glassware Flow Designer, shown in **Figure 13-1**, can be used with any Glassware project, but we find it works especially well for charting out live card projects. Because it runs on Google Drive, you can share designs with other people and collaborate on putting flows together, editing them, and presenting them prior to beginning the fun task of writing code.

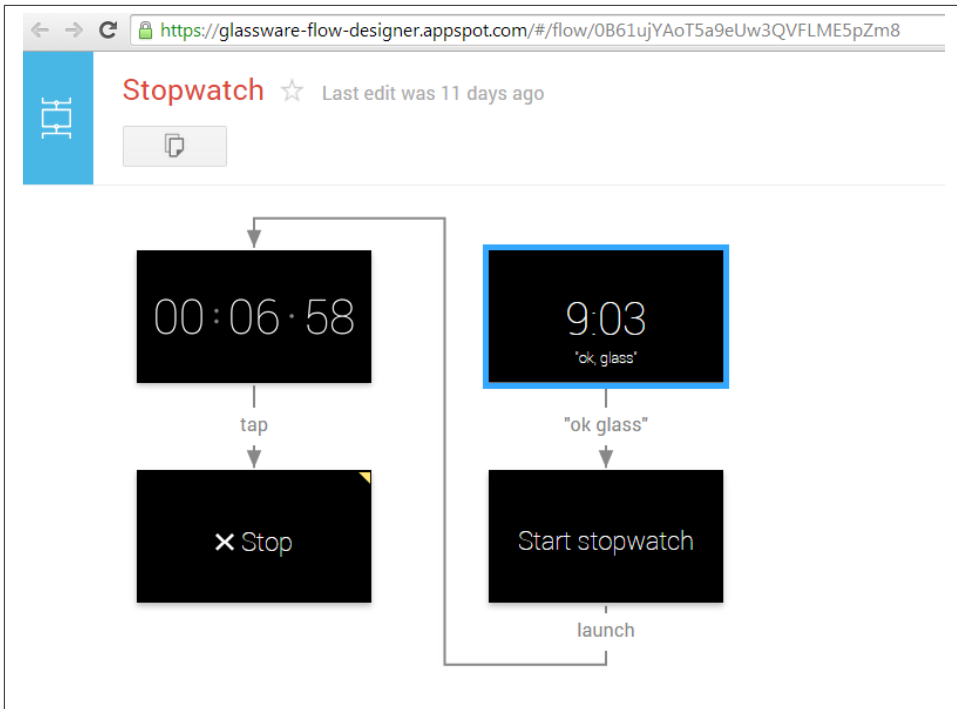


Figure 13-1. Glassware Flow Designer

However, Glassware Flow Designer isn't intended to be an all-in-one solution. Glassware Flow Designer is effective for assembling an overall UX flow, not necessarily denoting things at the component level. It doesn't currently differentiate between cards and other Android components like services or broadcast receivers, databases, or Java objects. **Justinmind Prototyper Pro** is a fantastic tool that lets you visualize gestures, swipes, and taps, and even canvas objects. **GlassWireframe** denotes actions, multimedia, and sensor use.

So if you prefer more detailed sketching when putting specs together for complex applications, these tools are great to have handy.

Whichever you go with, they'll help speed up your production process. When teaching Glassware design sprints, Allen has noticed that when using Glassware Flow Designer teams greatly increase their turnaround time for prototyping and can turn ideas into working products.



What's the Optimal Image Density?

There's a valid concern that astute Android developers will have about the optimal density bucket to use for images in GDK apps. To date, the [Android documentation](#) notes the linear relationship between screen size and pixel density—the larger the display, the larger the density. But Glass, once again, proves different—the resolution for the tiny prism display is 640 x 360, with the effect being viewing a 25-inch high-definition display from 8 feet away. What density bucket should we use in our Glassware?

[Google recommends](#) that drawable resources for Glass be stored in *at least* the `/res/drawable-hdpi` folder in your IDE. That's a good baseline to follow to ensure clarity and performance.

It Was Native All Along!

As a means of illustrating what native apps can do, [Table 13-2](#) presents some recognizable examples of the GDK elements at work using the Glassware that Glass ships with.

Table 13-2. Examples of Glass UI elements

UI element	Glassware examples
Static cards	Nearly everything with the Mirror API, Google Search results
Live cards	Voice Call, "Set wake angle" Settings card, Google Now cards
Immersion	Hangouts, Record video, web browser, navigation, "Bluetooth" card in Settings, camera

Now ask yourself—how many of these surprised you at being native components and not generated by the Mirror API? As a bonus, what element do you think the home card uses? Gold star for you if you guessed live cards! The card updates on its own with the current time, synced to the network and displayed in your local time zone. It also dynamically displays certain system status messages like "In a phone call" or "Glass must cool down to run smoothly," which appear in response to system events. Neat, huh?

Glass Gets a Failsafe Against Overheating

Even though the Glass hardware is designed to direct the heat generated by extended use of the processor and/or display away from the wearer's head (the heat can be felt directly on the touchpad, opposite the user's skull), a defensive mechanism exists that prevents apps from running if Glass was already running hot, as shown in [Figure 13-2](#). While you can (and should) code your app in ways that minimize intense processing or displays that stay active for long durations, Glass will mercifully override any attempt to launch your app and block it until it cools down.

Installed apps using immersions and the onboard sensors are prime candidates that can trigger such a condition, as they impose a certain load on the projector and the processor, respectively. So code smart, but also know that Glass is looking out for you.



Figure 13-2. Glass plays it cool

Also keep in mind that these apps demonstrate an advantage of *hybrid Glassware*, using elements of both native components and static Mirror API cards. (We'll get to this shortly!) By the same token, when using the “Get directions” voice command everything is presented in terms of Mirror API elements, but once you tell Glass to give you turn-by-turn directions, the UI shifts to a live card that updates its visual appearance AND speaks updates aloud, sitting to the left of the home card. Navigation uses hybrid elements as a search-results relationship with static cards and immersive environments. The web browser handles any URLs within static cards. And the “Factory reset” option in the Settings bundle invokes a low-level program that restores the system to its out-of-box default state.

When sending or receiving a voice call, the GDK creates a live card with the contact you're talking to left of the home card with a call counter to keep the elapsed time. Upon termination of the call, a static card is inserted as a call log into the right of the home card with the number you spoke with and the time spent on the call. This is emblematic of the design strategy we've been teaching you that neatly puts what's happening apart from what's already happened. The right tool for the right job.

Some of the native components that control the camera also map their commands not only by menu items and voice commands, but also by a special hardware control—the shutter button. Handle the `onKeyDown()` method in your activity to process shutter button presses.

The GDK Object Model

The classes you'll be working with when using the GDK are organized under the *com.google.android.glass* namespace. You can review the classes and their supported methods, properties, and events from your IDE's autocomplete feature, or by reviewing the class reference documentation online.

The classes cover operations that let you work with cards (static and live), hardware, menus, the touchpad and accompanying user gestures, and timeline dynamics. The touchpad API includes interfaces for you to implement for the various types of tapping and scrolling you're able to capture. It also has an enumeration of values to identifying the type of user input captured, such as directional swiping, two- and three-finger long pressing, and double-tapping.

Packages

com.google.android.glass.app

Model for structuring cards; voice trigger operations to invoke the app from the main menu.

com.google.android.glass.content

Defines the explicit intent actions and extras specific to Glass.

com.google.android.glass.media

Extends the Android Camera API for capturing still images and video.

com.google.android.glass.timeline

Model for live cards and to interact with the timeline.

com.google.android.glass.touchpad

Recognize touch gestures.

com.google.android.glass.view

Extensions for `Menu` and `WindowManager` classes.

com.google.android.glass.widget

Special views that let you implement horizontal scrolling for navigating through collections of cards (the Glass version of a `ListView`).

This list may change with subsequent platform releases, so bookmark [the changelog](#) to see what's new.

System Intents

The GDK also lets you make use of certain system applications provided by Android through implicit intents. If you have the need to let the user go out onto the Web, you

can launch the Glass web browser. If your application involves geolocation and directions, you can call up the Navigation app for Glass and pass it a data URI with a series of parameters that assemble the turn-by-turn sequence as the users move toward their destination. You can also detect when a user is wearing Glass or has taken the headset off, in addition to getting paths for photos and videos the user has captured.

You could combine these ideas for a wearable pizza delivery application—creating a live card for a delivery driver that contained the customer’s name with a timer counting down the time taken to fulfill an order. That card might include custom menu items for providing driving directions to the customer, as well as letting the driver look up specials and promotions on the Web.

In case you’re pondering using system software in your own interfaces, here’s a rule of thumb: a live card can launch an immersion, but an immersion as we’ve noted has to be canceled via downswipe with the user returned to the timeline before other actions can be taken. So if you’re planning out an app and sending the user to the browser or navigation is key, use a live card as your launching pad. Your app can even register an intent that can be launched via a menu item in a static card. So, you’ve got options.

See the [documentation on using system intents](#) for implementation details.

The Magic of Bridged Notifications

One of the ways the Glass ecosystem continues to expand outward and integrates with the larger Android family is how Android Wear uses notifications, and how they’re synced between handhelds and their connected wearable devices. The magic is *bridged notifications*, which takes the default notification APIs for Android handheld programming and magically makes them work on wearables.

Bridged notifications represent an evolution of Android intents and give you great leverage as a programmer—messages can now not only be passed explicitly between components within the same application and implicitly between components within different apps resident on the same device, but also between devices connected through the same Google account, with data synchronization across devices being automatic and nearly instant. This is incredibly powerful.

Users are constantly connected to their personal network of two or more nodes they’re signed into through their Google accounts. We’ve got similar notifications on Glass through [Notification Sync](#), so this is very exciting.

On-Head Detection Halts Running Apps, Too

When enabled, the On-Head Detection feature from the Settings bundle acts as an automatic traffic cop for running apps. If a user removes Glass while a native app is

running, Glass goes to sleep, and certain actions happen depending on the GDK element.

Live cards are paused but continue to run in the background and resume once Glass is returned to the user's head, retaining their state. Things are a little more unforgiving with immersions, though—any executing immersion is killed outright and must be restarted from scratch when Glass is put back on. This could result in a loss of data, so if you are planning an immersive app, you may want to add a warning to users to not remove Glass, or manipulate the Android lifecycle callbacks that are fired when running activities are halted to save any progress (typically in the `onPause()` callback). Or better yet, bake in a periodic autosave feature.

The effects on an application if the user takes Glass off are an oft-overlooked condition that need to be accounted for as part of proper defensive programming. Don't let your users find out the hard way.

Hybrids: The Ultimate Glassware Challenge (and Experience!)

It's important to note that the two frameworks for Glass development aren't mutually exclusive. While the Mirror API and GDK stand on their own in order to create cloud-dependent services and installed applications, respectively, you can also combine the two schools of thought to make a really engaging wearable experience. Hybrid Glass applications (designated in [Table 13-3](#) as “*Mirror API + GDK*”) use the best of both worlds—the quick and lightweight push nature of timeline cards with a full-blown custom native UX ([Figure 13-3](#)).

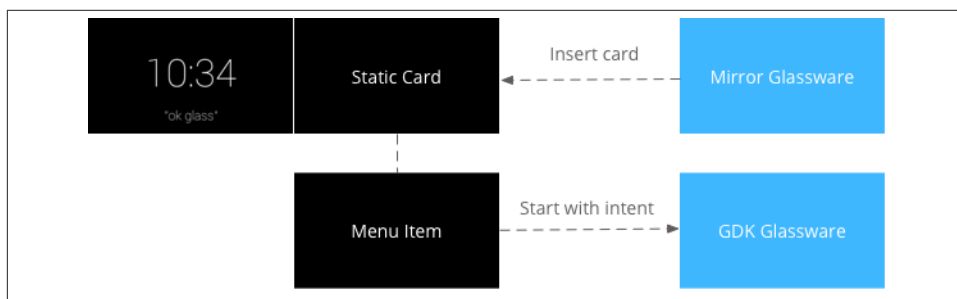


Figure 13-3. Flow for hybrid Glassware (image courtesy of Google)

The communications bridge between the two frameworks is a menu item on the timeline—more specifically, the `OPEN_URI` built-in menu item value from the Mirror API with a corresponding URI as a value for the payload property. You'd normally use this for redirecting the users to a link on the Web after they select the menu item, but it can also

be used to launch a specific activity within a native app. In addition to the system-level intents that can be called, specifying a URI within the scope of your own application like `content://com.book.thinkforglass.totallyawesomesocialapp/status/8675309` jumps right to that screen, which would fire an activity registered to receive that URI type in your application's `AndroidManifest.xml`. But, you could also specify a Java class that makes use of the camera or sensors.

So you could spend the time and build out a very robust installed application with the GDK to handle long-running jobs and local storage, but use the Mirror API as a flexible frontend that doesn't have to be (re)compiled and (re)distributed whenever you make changes to it.

Table 13-3. Classification and requirements for GDK apps, Mirror API services, and hybrid applications

API	Category	Language/framework	Authorization	
Mirror API	Cloud service	Any server-side framework	OAuth 2.0 (required)	Requires connectivity, auto-synced
GDK	Installed application	Java, Android SDK	AccountManager	Offline, real-time interactivity, sensor access
Mirror API + GDK	Hybrid	Combination	Combination	Menu items in static cards launch native activities

Let's look at an example...one that you've probably been using a lot and may not have realized uses both frameworks. Hangouts is a shining example of hybrid Glassware, using Mirror cards and menu items as its UI and a handle that invokes the Call activity natively. In this case, we have an Android application that doesn't have a UI of its own in the traditional sense but exists as a series of components that wait to be launched by Mirror actions.

The actual chatting feature is also a bonus lesson in Glassware design, showing how bundles of static cards, the standard menu items, and the Timeline object can be used for group messaging. Replying is done with the stock Reply menu item, and the mosaic of conversation participants is populated with the `Timeline.recipients` property—which is all Mirror API. What's interesting to note is that the telephony actions for voice calling go through the smartphone to which Glass is connected via Bluetooth, but the chat element runs locally on the device and talks to the cloud directly—so consider how Android is handling the networking for a multifaceted communications application like Hangouts.

The way Hangouts is built should give you some good ideas about how to combine the two frameworks to really do some cool things and not be limited by using just one.

Also, currently if you want to programmatically manipulate static cards you'll have to use the Mirror API *from a GDK app*—but that's OK! You'll need to use the network and authentication techniques, which is a beautiful segue into our next section...

Authentication

GDK apps can run without requiring authentication and can exist just fine without doing the OAuth dance that Mirror API Glassware needs to when communicating with a remote resource. However, there will certainly be cases where native Glass apps need to talk to RESTful APIs, so there is a facility for using standard Android programming techniques, although it's probably a different method than you're accustomed to.

This attacks the single most-frustrating problem of authenticating on Glass versus authenticating on mobile or desktop: the lack of a keyboard. We mentioned earlier how entering passwords could be a daunting task, given that most systems these days require combinations of characters with at least one numeral, possibly one uppercase character, and likely one special character like an ampersand. As rich as the Glass speech-to-text engine is, it'd never be able to decipher complex credentials. So instead of forcing users to authenticate the first time they use an application, they will authenticate when they install the application. And since that installation is done from a mobile or desktop device, we can take advantage of the keyboard those devices naturally provide for us. By this same merit, users are able to review and agree to the permissions an app requires, which wouldn't be so easy on a wearable.

The solution highlights that one of the principles behind effective Glass design is—again—using the right tool for the right job.

In a nutshell, GDK authentication dictates that *when a Glass user installs native Glassware requiring authentication with a third-party service, the user is prompted to sign in, and upon completion the user's account is pushed to Glass*. From that point on, the Glassware talks to the API on behalf of the user. And everything happens through MyGlass via its mobile application or on the Web. It's that simple. This is a much more fluid system than having to build, maintain, and integrate a custom authentication platform of your own. At a macro level, it also ensures consistency in the process wearable users experience across their Glassware.

And here's the trick to how it all works: even though this is a GDK app, it's the Mirror API that pushes a user's account onto his device. Cool, huh?

GDK apps use an Android `AccountManager` object to handle access to their identity with the service. The authentication process uses the special `Accounts resource` to grant applications access. Unlike the OAuth flow used with the Mirror API Glassware, which uses a project registered as a *web application* in the Google Developers Console (as we detailed in [Chapter 8](#)), GDK authentication flows rely on a *service account*. The fundamental difference between the two is that a service account handles server-to-server communications by authenticating a service rather than a user. See [Google's documentation](#) for a step-by-step walkthrough of how to create and configure a Google API service object.

Authenticating GDK Glassware works as follows (Figure 13-4):

1. A user enables your GDK application via MyGlass, which redirects the browser to your login page. This request is made to your server with a `userAuth` parameter.
2. On your login page, the user submits her credentials. (A special OAuth scope, <https://www.googleapis.com/auth/glass.thirdpartyauth>, is required when using this method.) In this step, any Android permissions your app requires are also declared to the user.
3. Upon successful validation of the user's credentials, your backend calls the Mirror API's `mirror.accounts.insert` endpoint with a JSON-formatted request body describing the account and its capabilities.
4. The Mirror API sends the user's account to his Glass device, which is then available via an Android `AccountManager` object. See Google's documentation for required permissions and associated APIs for account retrieval.

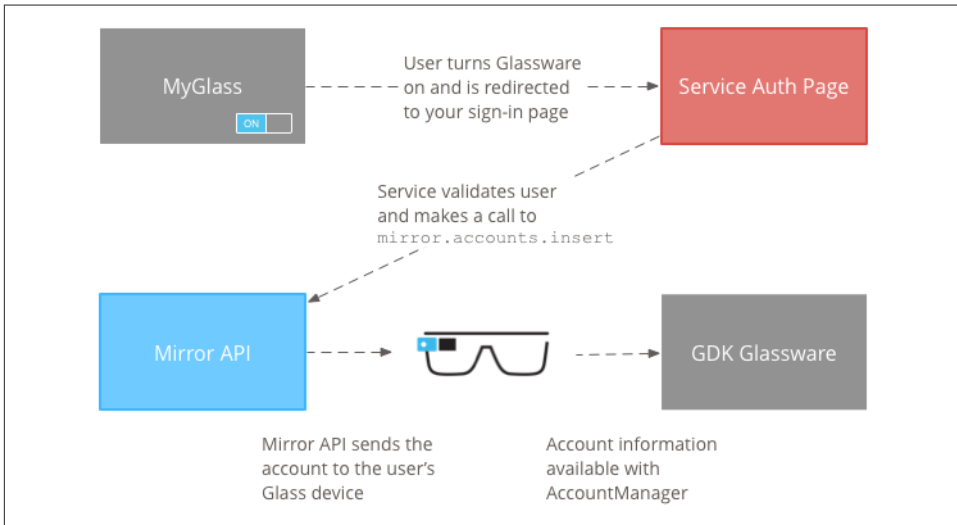


Figure 13-4. The GDK authentication flow (image courtesy of Google)



If You Thought OAuth Was Difficult...

Talking to service accounts server to server is a fairly complicated process, decidedly more than the user-to-server model that OAuth enforces. So it's highly recommended in the interest of preserving your sanity and social life to use the Google API client libraries, which handle the complex plumbing for you.

At the moment GDK authentication is only available to you *after* your APK has been uploaded by Google to MyGlass, so you're unable to work with the API locally on your development machine until then. (Just like when authorizing Mirror API Glassware for testing, you'll see your app appear in MyGlass when signed in with your account, but it won't be available for all Glass users until it's approved and officially goes live.) Google's iterative Glassware review process, which is one of many benefits of having your Glassware approved and cataloged, doesn't mandate that your project be a ready-to-go production app—you can submit a rough prototype and indicate where you'll be implementing web APIs. You'll be able to build and test features while your project is under review.

Authentication continues to evolve, so keep an eye out for new developments in this space as it reaches maturity.

AccountManager for Other Types of Configuration

One of the neat tricks about the way Glass reads AccountManager data is that you can use it to set and change other configuration settings. The AccountManager object model takes a number of key/value pairs for the `userData` property. Give it a spin.

There's a good argument to be made about what might happen in terms of workflow and the installation process if you include signing in to access some sexy new RESTful API down the road. The good news is that the authentication prompt will kick in automatically, forcing the flow.

Just like applying a new permission after-the-fact in a subsequent release of your app, the new enhancements take effect the next time.

Writing Native Code for Glass

Whether you're using Eclipse or Android Studio, you'll need to physically connect Glass to your development machine via the micro-USB cable and then enable Debug Mode in the Settings bundle on Glass, which turns on the ADB and registers your headset as an available device. This lets you run your app live with all of the UX functionality and controls—sensors, taps, swipes, voice commands, and gestures. You'll be running and debugging your code (hopefully more of the former and less of the latter) live on Glass.

When you test builds of your application, your IDE will compile your code into an `.APK` file and then install it on Glass, just like running an app with your smartphone connected. To make sure Glass is being recognized as an AVD, click the Device panel in Dalvik Debug Monitor Server (DDMS; [Figure 13-5](#)). It should be registered and listening for events, with the output being logged in real time to LogCat.

Both IDEs let you specify that you'll be compiling with the GDK, which bundles the necessary JAR library into your project. If you're using Eclipse, creating a new project gives you a boilerplate Android application that will render "Hello world!" in a fairly pedestrian static card that launches immediately when you run the app. You can investigate the project's structure and its code on your own to see how it sets the various values.

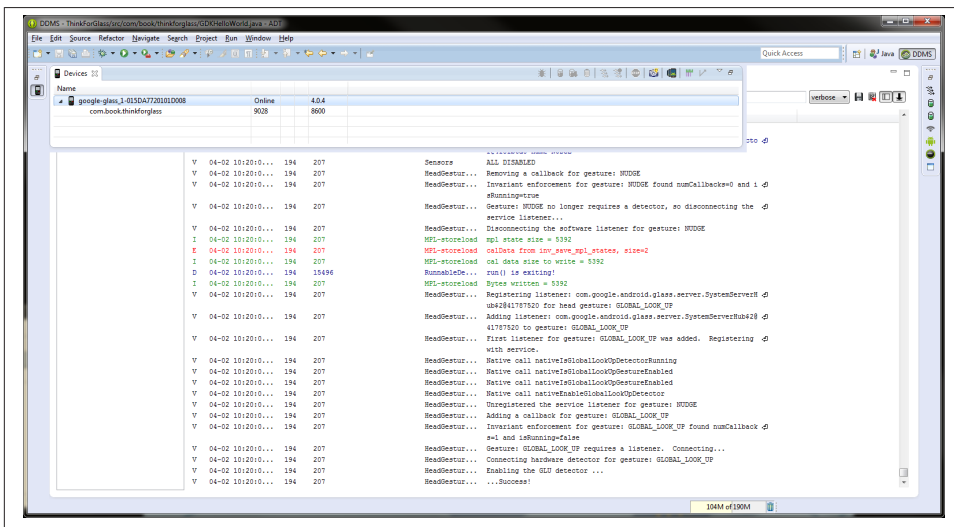


Figure 13-5. Glass recognized in DDMS view in Eclipse



Getting Glass Drivers for Windows

A glitch that many developers using Windows machines noted early on was incompatible USB drivers for Glass, which blocked the use of ADB on that platform. Eclipse and Android Studio have both made the correct driver available through SDK Manager, but if you continue to run into problems, our buddy Andrew Pritykin [produced a helpful tutorial video](#) showing how to update and enable the drivers so that you can install your apps properly and use all the helpful features of ADB.

Of course, Mac OS and Linux users don't need to worry about this minor inconvenience. Carry on.

If you're using Android Studio, selecting the Glass form factor when creating a project generates either a live card application, an immersion, or a blank project that you fill in manually (Figure 13-6). The live card and immersion apps are great learning resources and easily expandable as you become comfortable with each API. They also demonstrate

the recommended coding patterns to optimize performance and usability for both a live card (by using a service) and an immersion (flow, UI, and program control).

Once you've got the hang of setting up simple bare-bones GDK apps, you can tackle some of the more advanced **sample projects** Google's published for complete end-to-end native apps.

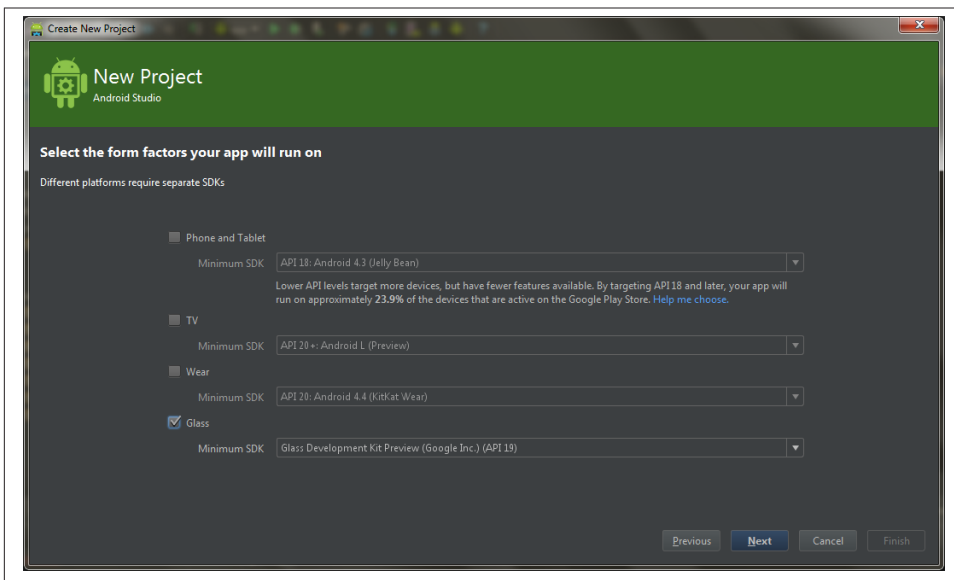


Figure 13-6. Setting up a Glassware project in Android Studio

Producing Native Apps Through Other Frameworks

There's lots of work being done to produce native apps on Glass by using the less-complex and more rapid web stack—HTML, CSS, and JavaScript. This architecture is being used within the Glassware development community by ambitious projects like **WearScript** and **PhoneGap**, which access sensor readings via web interfaces.

Testing Native Glass Applications

We're not going to outline coding up an entire GDK application, due to the fact that (1) the GDK is still evolving and methods, properties, and events are subject to be added, renamed, or outright removed at any time; (2) many of the features are still undocumented as the GDK is still in the Developer Preview stage; and (3) quite frankly, Android apps require a lot of files and documenting them takes up a lot of space. If you know

how to create Android apps, the sample projects and poking around GitHub for cool repositories people are working on will get you started. If you're new to Android programming, check out the [GDK documentation](#), which walks you through a primer on mobile coding.

Visually, the GDK relies on a [Glass theme](#) that sets an application to full-screen without systems elements like the status bar, action bar, clock, or battery life indicator that you're used to seeing in other Android form factors. To achieve the transparent effect of menu items that float above their associated app like you're used to on the timeline, it's helpful to define a custom style in your app's `/res/values/style.xml` resource file and set it as a theme for activities containing menus. This ensures your layouts, fonts, and UI elements match the Glass UX:

```
<resources>
  <style name="MenuTheme" parent="@android:style/Theme.DeviceDefault">
    <item name="android:windowBackground">@android:color/transparent</item>
    <item name="android:colorBackgroundCacheHint">@null</item>
    <item name="android:windowIsTranslucent">true</item>
    <item name="android:windowAnimationStyle">@null</item>
  </style>
</resources>
```

And as noted earlier, you're still working exclusively in landscape mode, with the viewable surface wider than it is taller at 640 x 360.

There are some notable differences between static cards in the GDK and those created with the Mirror API in regards to the amount of formatting control you get. The GDK does include several of the layout templates that the Google Mirror API Playground provides. You can specify full-bleed background images, icons, footers, and timestamps and let Glass handle the formatting. Plus, you've still got the ability to use Android's powerful graphics libraries and drawing classes to create dynamic 2D/3D graphics and animation, which you can't do with the Mirror API.

Also, when testing your app, do make use of the helpful [Glass developer settings](#), available from the Settings bundle. These utilities are like the developer tools Android programmers have relied on for years, letting you preview the layout of elements and visualizing their boundaries in relation to each other, allowing you to control the speed of animation playback, GPU overdraw cycles, and more. Personally, we can't live without the "Keep the screen on while charging" option, which prevents you from losing your back stack state while you test an app.

A View to a Card

For you Android developers who suffered through the Mirror API chapters wondering when you would find out how to display things with Glass, this might be the moment you're waiting for. Hopefully you'll see that many of the best practices we talked about

have some parallels on the GDK side of the world, but that you also have a lot of power to go your own way if you really need to.

If you remember the Playground tool we talked about back in our chapter on the Mirror API, you'll remember how it offers a number of templates, and lets you style them with snazzy HTML. Similarly, the Glassware Flow Designer offers templates and specific fields you can set. Unsurprisingly, Google also offers a Java class to help you build cards with these styles in your code as well.

The `CardBuilder` class, under the `com.google.android.glass.widget` package, lets you create a static card based on these templates. The general pattern is that in the `Create()` method of an activity you'll create a `CardBuilder` object, specifying the template type you're creating. You'll then set various properties on this builder, which we'll discuss later, and conclude by having the builder create a standard `View`.

Each method that sets a property returns the builder, so you can chain them together.

Really, a Card with a View

You'll notice that the `CardBuilder`'s ultimate job is to return a `View` object, similar to every other `View` object you'll encounter in Android. What goes into this object? If you inspect it (and all of its children), you'll see some classes you should be familiar with like `FrameLayout`, as well as some that are clearly made for Glass, such as `MosaicView`.

Unfortunately these are still opaque to us, but hopefully a future version will make them more available.

Let's look at a few simple examples of using `CardBuilder`. The object's constructor expects two arguments—a context and a `CardBuilder.Layout` enumeration that specifies the exact layout being applied.

Basic Text Formatting

For simple cards, similar to the article/section with a `text-auto-size` class from the [base CSS styles for Glass](#), we can use the `CardBuilder.Layout.TEXT` template. This lets us set text that is auto-sized to fit where possible:

```
CardBuilder christine = new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("I felt as conspicuous as a baby whale in a goldfish pond.");

// display the CardBuilder object in the Activity
setContentView(christine.getView());
```

Optionally, you can instantiate a `CardBuilder` object directly as a `View`:

```
View christine = new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("I felt as conspicuous as a baby whale in a goldfish pond.")
    .getView();
setContentView(christine);
```

We'll stick with the former convention for the next few examples. These examples assume that they are being called inside `Activity.onCreate()`, so the "this" variable is the context of the current activity. It also assumes that we'll do something with the View that is built, such as display it or add it to a `CardScrollView`, which we'll talk about later:

```
CardBuilder christine = new CardBuilder(this, CardBuilder.Layout.TEXT)
// a resource located in /res/values/strings.xml
    .setText(R.string.stephen_king_quote);
```

The three previous code blocks produce the static card in [Figure 13-7](#).

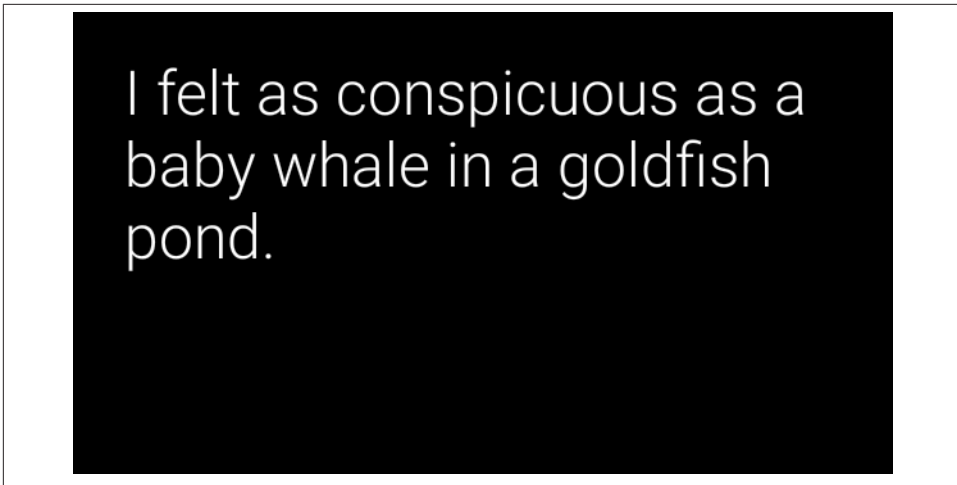
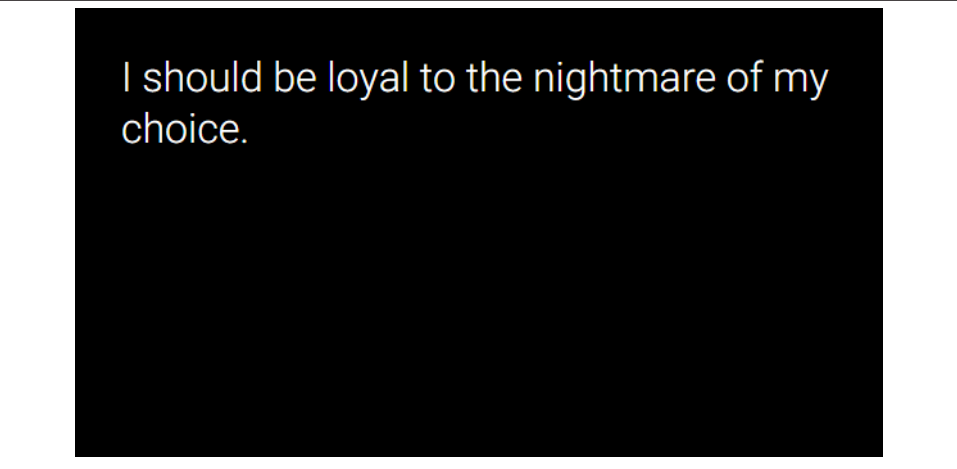


Figure 13-7. The TEXT layout

Another template, `CardBuilder.Layout.TEXT_FIXED`, is similar, except it applies formatting equivalent to the `text-small` class from the base styles CSS ([Figure 13-8](#)):

```
CardBuilder heartOfDarkness = new CardBuilder(this, CardBuilder.Layout.TEXT_FIXED)
    .setText("I should be loyal to the nightmare of my choice.");
```



I should be loyal to the nightmare of my choice.

Figure 13-8. The `TEXT_FIXED` layout

Both of these layouts (as well as most of the layouts we'll be discussing) also let us set the card's footer, just like the HTML we can provide when using the Mirror API. Unlike the Mirror API, however, we can also set the timestamp field that is on the right of the card. Android's `DateUtil` class provides us with some methods that format the timestamp correctly, or we can still put any `CharSequence` we want here (Figure 13-9):

```
long time_then = System.currentTimeMillis()-(24*60*1000); // 24 minutes ago
CardBuilder heartOfDarkness = new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText(R.strings.joseph_conrad_quote)
    .setTimestamp(DateUtils.getRelativeTimeSpanString(time_then))
    .setFootnote("Heart of Darkness");
```

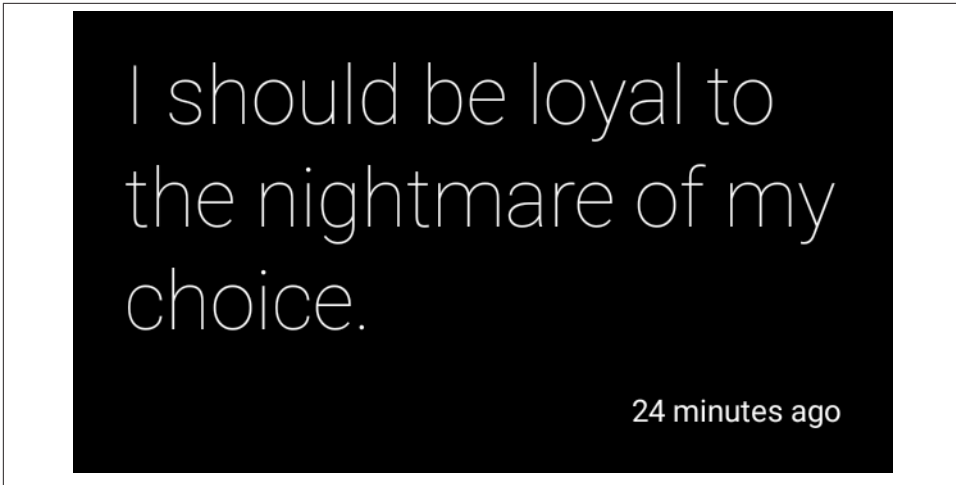


Figure 13-9. A TEXT card with a timestamp

Creating Rich Text

What other formatting can we do? You'll note that there's no `setHtml()` method to go with `setText()`, and we can't add a `View` directly as a child. We might be able to meddle with the `View` that gets created, but that seems like a bad idea. Fortunately, we do have a solution.

The `setText()` method takes a class that inherits from `CharSequence`. Fortunately, the `android.text.SpannableString` class is a `CharSequence`, and it can contain spans of additional formatting markup. While there are a few ways we can generate this marked-up `CharSequence`, the easiest way is to use the `android.text.HTML.fromHTML()` method (Figure 13-10):

```
String html = "<b>bold</b><br><font color='red'>red</font>";  
Spanned htmlSpan = Html.fromHtml(html);  
  
CardBuilder htmlCard = new CardBuilder(this, CardBuilder.Layout.TEXT)  
    .setText(htmlSpan)  
    .setTimestamp("around 2005");
```




Figure 13-10. Rich text formatting

But the `HTML.fromHTML()` method isn't perfect—most notably, the HTML it uses is pretty ancient. If you're used to CSS and HTML class or style attributes, it will feel clunky having to go back to a `` tag. If this bothers you enough, feel free to build your `SpannableString` another way (Figure 13-11):

```
SpannableString htmlSpan =
    new SpannableString( "green\nbold\nrelatively normal" );
htmlSpan.setSpan( new
    ForegroundColorSpan( getResources().getColor( R.color.green ) ),
    0, 5, Spanned.SPAN_EXCLUSIVE_EXCLUSIVE );
htmlSpan.setSpan( new StyleSpan( Typeface.BOLD ), 6, 10,
    Spanned.SPAN_EXCLUSIVE_EXCLUSIVE );

CardBuilder htmlCard = new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText(htmlSpan)
    .setTimestamp("sometime this year");
```

What about other layouts such as tables, left and right justification, and images? Other HTML-based formatting that the Mirror API provides, such as tables and lists, aren't available—at least not yet. Images will be covered in a little bit, as will other template types, including some that aren't available through Mirror.

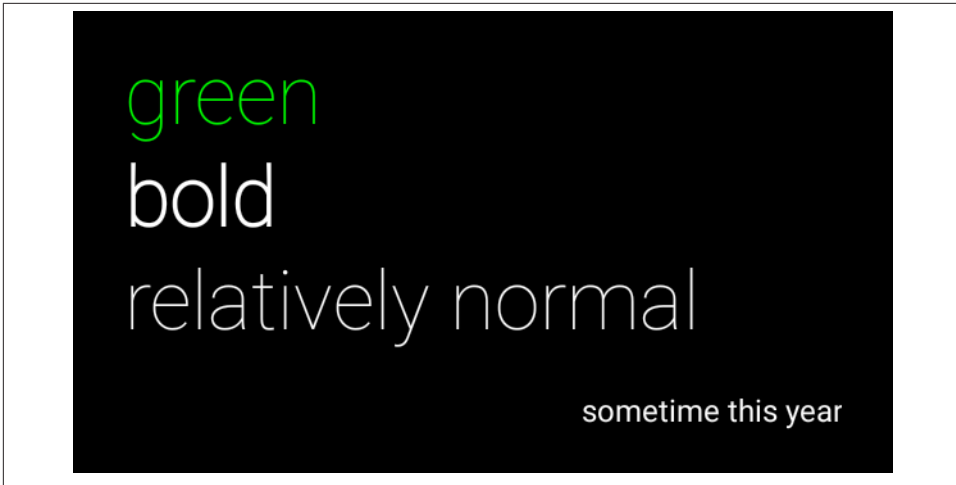


Figure 13-11. Rich text formatting with a `SpannableString`

Ellipses and Excess Content

You may remember that we explored a way to make it clear to our users when there is more information than what we are displaying on the card. Although there is no way to set the dog-ear icon for the card at this time, it seems reasonable a future `CardBuilder` will be able to provide this. Other aspects, such as an ellipsis at the end of the card, are more feasible.

How do we provide this ellipsis? Fairly easily—we just need to provide more text than will fit on the card (Figure 13-12):

```
CardBuilder iOnlySeeSixLines = new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("one\n two\n three\n four\n five\n six\n seven\n eight\n");
```

What about the scenario where we want the header line to show an ellipsis? That is significantly more complicated. But it probably isn't necessary using `CardBuilder`. With the Mirror API, we needed to restrict the title to a single line to make sure we counted lines correctly to apply the ellipsis, but that isn't necessary with the GDK—we can just let it take care of the ellipsis at the end. If you really want a specific header layout, there is also the `CardBuilder.Layout.AUTHOR` template, or you can roll one yourself (see the next section for more on this).



Figure 13-12. You get an ellipsis for free with excess text

Columnar Layouts and Mosaics

One of the very common layouts we've explored is where we'll have the leftmost-third of the screen with an icon, a set of images, or other information that is quickly glanceable. The right two-thirds contain more details, such as a message.

It should come as no surprise that we have two versions of this template for both dynamically sized and for fixed text, `CardBuilder.Layout.COLUMNS` and `CardBuilder.Layout.COLUMNS_FIXED`, respectively. The most common use for these is with a mosaic of images in the left column. We'll use the `addImage()` method to add tiled images—one image for each call, in the order we add them (Figure 13-13):

```
CardBuilder message = new CardBuilder(this, CardBuilder.Layout.COLUMNS)
    .setText("one\n two\n three\n four\n five\n six\n seven\n eight\n")
    .addImage( R.drawable.messageFrom ) // Allen
    .addImage( R.drawable.messageTo1 ) // Jason
    .addImage( R.drawable.messageTo2 ); // Pegman
```



Figure 13-13. Calling `addImage()` repeatedly creates automatic mosaics

As we shared in [Chapter 7](#), the mosaic images in column-based layouts aren't randomly assembled and display a distinct hierarchy. The first image added is displayed as the most dominant in terms of it being positioned at the top, and occupying the most screen real estate. This is often used to denote a sender-recipient(s) relationship in messaging Glassware, but it can be used for other purposes, too. The user instantly gets the gist of what the card represents, a chat conversation or an email, as opposed to a news article, a tweet, a storm alert, or an announcement about a sale for gaudy holiday season sweaters.

Consider the effect you create when calling `addImage()` in a certain order, not just doing so arbitrarily. Conversely, if you *haven't* got a need for such structure in your data, also think about what impact such a mosaic presentation might have on the users viewing it. They might see importance that isn't actually there.

Using Icons

Another excellent use of the `COLUMNS` template is to place a single icon in the center of the leftmost column. Examples of this include the settings cards that Glass uses for its configuration. We can only set a single icon, however, so we would use the `setIcon()` method ([Figure 13-14](#)):

```
CardBuilder message = new CardBuilder(this, CardBuilder.Layout.COLUMNS)
    .setText("one\n two\n three\n four\n five\n six\n seven\n eight\n")
    .setIcon( R.drawable.wifi );
```



Figure 13-14. Icons are simpler than imagery

Mixing It Up

Although the documentation for the column-oriented templates say you can use an icon or the image mosaic, but not both, there doesn't seem to be anything that enforces this right now, and the effect can be interesting and useful. Still, we suggest you avoid trying to combine them in case Google changes something in the future.

Both images and icons can be `Drawables`, `Bitmaps`, or resource references. And if you remember our text-centric layouts from earlier, we'll let you in on a little secret. They're good for more than just the columnar layout—you can use `addImage()` to add one or more mosaic images in the background of a text layout, too. This will darken the mosaic images using something similar to the `overlay-full` base style class to make the text more legible (Figure 13-15):

```
CardBuilder participants = new CardBuilder(this, CardBuilder.Layout.TEXT)
    .setText("one\n two\n three\n four\n five\n six\n seven\n eight\n")
    .addImage( R.drawable.member1 )
    .addImage( R.drawable.member2 )
    .addImage( R.drawable.member3 );
```

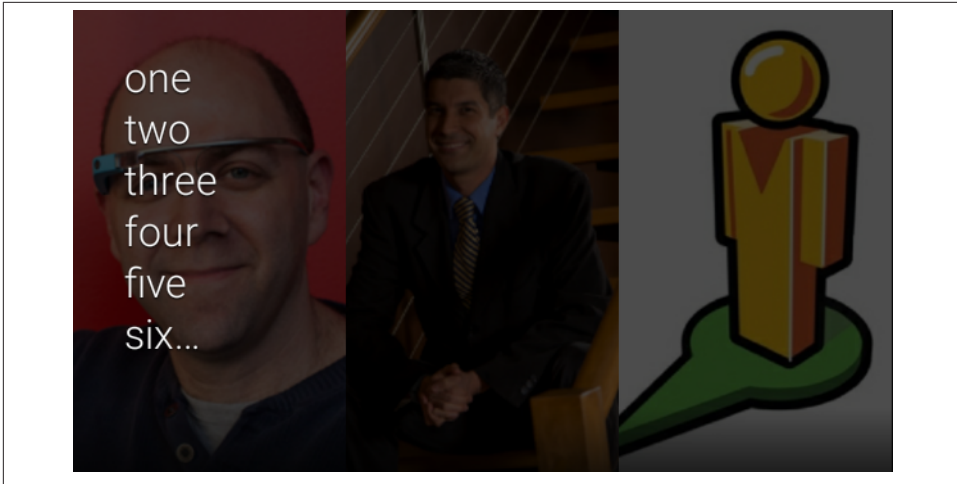


Figure 13-15. Applying images to a TEXT layout

We also have a derivative template that places text at the bottom of the page, with the `overlay-gradient-short` to make it more legible. The layout is more intended for a single line, but it will display up to two lines, and can take an optional icon. This is the `CardBuilder.Layout_CAPTION` template (Figure 13-16):

```
CardBuilder members = new CardBuilder(this, CardBuilder.Layout.CAPTION)
    .setText("one\n two\n")
    .addImage( R.drawable.member1 )
    .addImage( R.drawable.member2 )
    .addImage( R.drawable.member3 );
```



Figure 13-16. Applying images to a CAPTION layout

Other Neat Templates

Although we've seen the most basic types and uses, we need to be aware that there are more templates available. If we are duplicating the contact cards that we get through the Mirror API, for example, we will want to use the `CardBuilder.Layout.TITLE` layout, which takes a background image (or images), a single line of text, and an optional icon next to the contact text:

```
CardBuilder members = new CardBuilder(this, CardBuilder.Layout.TITLE)
    .setText("My Clique")
    .addImage( R.drawable.member1 )
    .addImage( R.drawable.member2 )
    .addImage( R.drawable.member3 )
    .setIcon( R.drawable.logo );
```

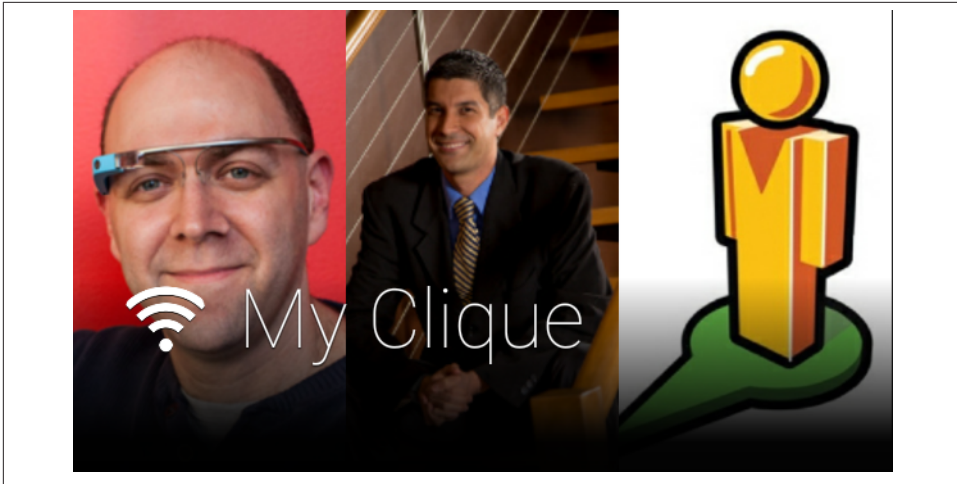


Figure 13-17. Combining an icon, images, and text in a TITLE layout

This kind of card, however, isn't very useful by itself. Most of the time, we will be combining it with other contacts as part of a swipeable list that you'll want to tap on to select. This can be done by using an instance of a `CardScrollAdapter` to manage the list of `Views` (which you've created through `CardBuilder` or elsehow) and a `CardScrollView` to actually control, render, and manage interactions with the cards. To handle the user selecting the card, you'll set up a click handler using `CardScrollView.setOnItemClickListener()`.

To do this, we might create our `CardScrollAdapter` to be flexible and take `CardBuilders`, `Views`, and resources that translate to `Views`. It might look something like this:

```
public class MyAdapter extends CardScrollAdapter {

    private Activity activity;
    private List<Object> items;

    public MyAdapter( Activity activity ) {
        this.activity = activity;
        this.items = new ArrayList<Object>( );
    }

    public MyAdapter( Activity activity, List<Object> items ) {
        this.activity = activity;
        this.items = items;
    }

    @Override
    public int getCount() {
        return items.size();
    }
}
```



```

@Override
public Object getItem( int i ) {
    return items.get( i );
}

@Override
public View getView( int i, View view, ViewGroup viewGroup ) {
    Object item = items.get(i);
    if( item instanceof CardBuilder ) {
        return ( (CardBuilder) item ).getView( view, viewGroup );

    } else if( item instanceof View ) {
        return (View)item;

    } else if( item instanceof Integer ) {
        return activity.getLayoutInflater().inflate( (Integer)item, viewGroup );

    } else {
        throw new ClassCastException( "Unable to create View from "
+item.getClass() );
    }
}

@Override
public int getPosition( Object o ) {
    int index = items.indexOf( o );
    return index < 0 ? AdapterView.INVALID_POSITION : index;
}

public MyAdapter add( Object item ) {
    if( item == null ) {
        throw new NullPointerException( "Unable to add null card" );

    } else if( item instanceof View || item instanceof CardBuilder
|| item instanceof Integer ) {
        items.add( item );

    } else {
        throw new ClassCastException( "Unable to add item of type "
+item.getClass() );
    }
    return this;
}
}
}

```

The Activity that creates the CardScrollView might, at a minimum, look something like this (Figure 13-18):

```

public class MyActivity extends Activity {

    protected MyAdapter adapter;
    protected CardScrollView scrollView;

```

```

@Override
protected void onCreate( Bundle savedInstanceState ) {
    super.onCreate( savedInstanceState );

    // Create an adapter to store the cards and add them
    adapter = new MyAdapter( this );

    // Create some cards we will scroll through
    adapter
        .add( new CardBuilder( this, CardBuilder.Layout.TEXT )
            .setText( "Card 1" ) )
        .add( new CardBuilder( this, CardBuilder.Layout.TEXT )
            .setText( "Card 2" ).getView() )
        .add( R.layout.card_3 );

    // Create the view and set which cards it will use
    scrollView = new CardScrollView( this );
    scrollView.setAdapter( adapter );

    // Setup click listeners
    scrollView.setOnItemClickListener(new AdapterView.OnItemClickListener() {
        @Override
        public void onItemClick(AdapterView<?> parent, View view, int position,
            long id) {
            // TODO: do something when a card is clicked
        }
    });

    // Show this view
    setContentView( scrollView );
}

@Override
protected void onResume() {
    super.onResume();
    scrollView.activate();
}

@Override
protected void onPause() {
    scrollView.deactivate();
    super.onPause();
}
}

```

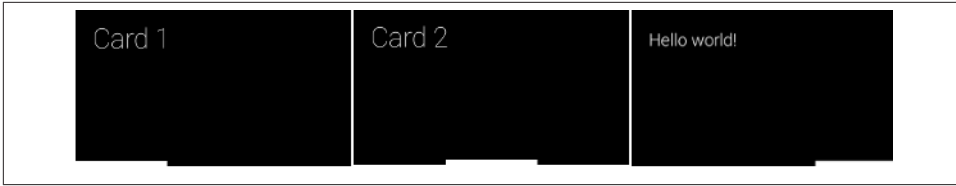


Figure 13-18. Combining cards with a `CardScrollAdapter`

As our sample code illustrated, you're not limited to using a title layout in the `CardScrollAdapter`. You can include any `View` created by `CardBuilder` or even a `View` that you create yourself. With these tools, you can duplicate most of the features of bundled cards or multiple contacts, and even go further by providing rich multilevel and dynamic menus.

We caution you, however, to not get too carried away. Remember that just because you can do all this doesn't mean that it creates a great UX on Glass.

Table 13-4 contains the currently defined list of templates that are available through `CardBuilder` and the attributes that can be set for each. As Google adds new templates, they'll be listed and documented as part of `CardBuilder`. Layout and on the GDK documentation about [building cards](#).

Table 13-4. `CardBuilder`.Layout templates^a

	<code>setText</code>	<code>setFootnote</code>	<code>setTimestamp</code>	<code>addImage</code>	<code>setIcon</code>	<code>setHeading</code>	<code>setSubheading</code>
TEXT/TEXT_FIXED	X	X	X	X	-	-	-
COLUMN/ COLUMN_FIXED	X	X	X	1	1	-	-
CAPTION	X	X	X	X	X	-	-
TITLE	X	X	X	X	X	-	-
MENU	X	X	-	-	X	-	-
AUTHOR	X	X	X	-	X	X	X
ALERT	X	X	-	-	X	-	-

^a x means the attribute is usable for this template, 1 means that only one of these two attributes should be used for this template.

When You Have No Choice—Doing It Yourself

But if none of these templates meet your needs, you always have the ability to create your own layouts using most of the standard `Views` that Android offers. Custom layouts can be created declaratively and then inflated through a `RemoteViews` object. (This is essentially the same pattern used for handling live cards with low-frequency rendering.)

The Glass team provides [a couple of custom layouts](#) using viewgroups and views in XML as guides that correspond to the prefab `Text` and `Column` templates to help you

match the standard Glass UI. However, while this approach gives you the freedom to create your own formatting, it also imposes on you the responsibility to manually implement padding, margins, and various styles that conform to the static card layout. If you need specialized formatting for content that's outside the scope of what Google already provides, that's fine, but don't deviate from the core motif.

Configuring Voice Commands

The range of sanctioned voice commands launch their associated apps, right off the "OK Glass" home screen. These triggers are defined in a special XML file in your project and registered in your application's manifest. To set up a voice trigger, you need to do two things: create an XML file that contains a string value defining your voice command; and register the command in your project's manifest. Then in `AndroidManifest.xml`, specify an `<action>` for an intent filter, either on an `<activity>` element or `<service>` element depending on your implementation; and then create a sibling `<meta-data>` element, using a similar action value for the `android:name` attribute; also, map `android:resource` to your XML resource file with values for a voice trigger:

```
<!-- applied as children of an <activity> or <service>
element in AndroidManifest.xml -->
<intent-filter>
    <action android:name="com.google.android.glass.action.VOICE_TRIGGER" />
</intent-filter>
<meta-data
    android:name="com.google.android.glass.VoiceTrigger"
    android:resource="@xml/voice_trigger" />
```

In this example, the resource value corresponds to `/res/xml/voice_trigger.xml`. Create the folder and file in your IDE and then enter the following for the file:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger command="TUNE_AN_INSTRUMENT" />
```

This sets your trigger phrase to “tune an instrument” (which is cool to say out loud) as the launcher from the "OK Glass" menu for your app. The result allows your app to be launched via voice as seen in [Figure 13-19](#).

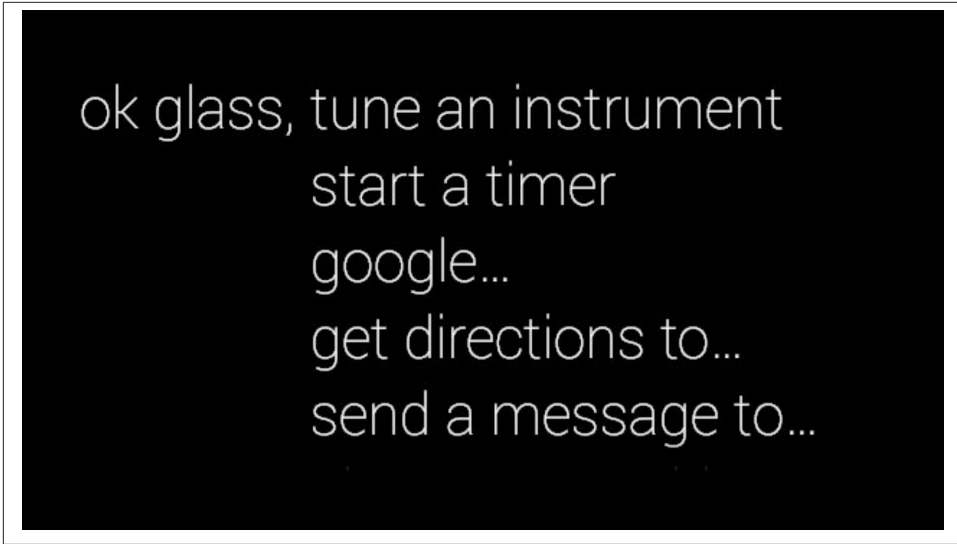


Figure 13-19. Launching an app via hotword

If your app requires additional vocal input, like for gathering a search query statement, a message to be sent, or an address, you can configure that secondary step by adding a child element to `<trigger>`, which evokes the voice input prompt:

```
<?xml version="1.0" encoding="utf-8"?>
<trigger command="TUNE_AN_INSTRUMENT">
  <input prompt="@string/glass_voice_prompt" />
</trigger>
```

You can define and use your own custom commands during development, provided you include an Android permission in your manifest. But again, deployed apps need to use the approved commands:

```
<uses-permission
  android:name="com.google.android.glass.permission.DEVELOPMENT" />
```

If it's imperative that you use a voice command not already on the approved list, you can submit it for review, which may take several weeks (and then new approved voice commands are only released when Google pushes an update of the Glass OS). The selection of your trigger phrase is an important step in your design/development. It should be logical, short, and easy to say, using action verbs with an active voice. The input is open ended, so don't use voice input to gather enumerated values like choosing one of the primary colors. That's what menus are for.

After you've picked a few options, do your homework and see what else is out there and what other approved applications are using to launch themselves, and how your command might fit in when listed beside them.

You might be wondering about possible trigger phrase clashes. How does Glass resolve conflicts when two or more Glassware are based on the same trigger? The system appends “with” at the voice prompt in the event more than one Glassware (enabled with the Mirror API or installed with the GDK) have the same phrase registered. A second-level menu prompt, shown in [Figure 13-20](#), then gives the user the option of which Glassware to use.

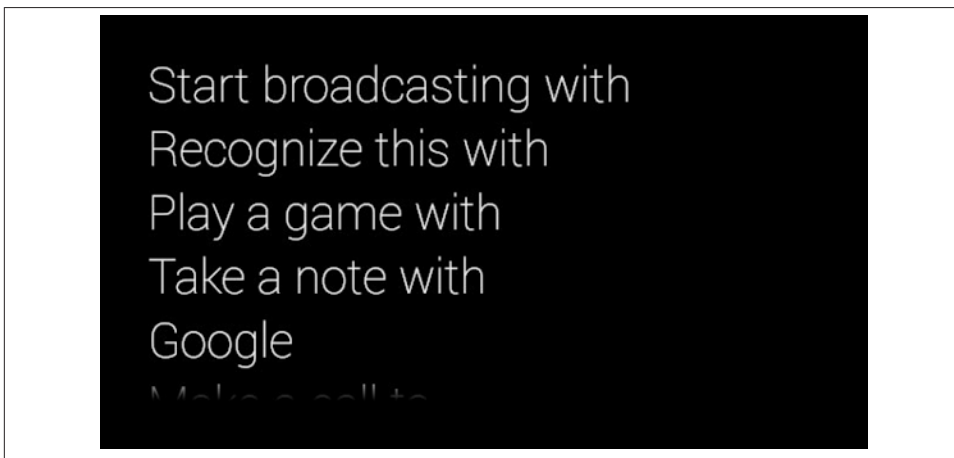


Figure 13-20. The voice prompt screen to launch Glassware

Make sure to review the [VoiceTriggers.Command](#) enumeration for the official list of approved voice commands. If you absolutely must have your own voice trigger phrase, review the [checklist of actions](#) to take and then [submit your own](#) for review.

You can also use *contextual commands*, which allow users to interact with menus within your app. The menus, just as with Android programming for handheld devices, are defined in layouts and inflated into activities, which are attached to their parent window when selected, and displayed immediately on top of it.

You can find out best practices for using contextual commands and other types of vocal input [on Google’s developer site](#).



Match Your Voice Commands Across Other Platforms

If the Glassware you’re building is part of a broader application ecosystem, you probably want to ensure for purposes of consistency that the voice commands available to users are uniform across other wearable platforms. Review the list of [system-provided voice actions](#) for Android Wear, as well as how to include [custom voice actions for Wear devices](#) so that your spoken commands are the same for other types of devices, like an Xbox One Kinect.

With Voice Commands, Google Has the Final Word

We can't underscore enough that the *voice commands* are ultimately under Google's sovereign control. It has final say on what apps are greenlit for placement in MyGlass, and voice commands are a big part of this approval process—don't find out the hard way. We mentioned in the *Design* section of this book how choosing your voice commands for Mirror API Glassware shouldn't ever be treated as an arbitrary exercise, noting the categorical hierarchy of commands as the library of approved Glassware continues to grow.

The same applies for the GDK, as trigger phrases like “*Tune an instrument,*” “*Start a bike ride,*” or “*Prepare a meal*” will be very popular for music, exercise, and cooking apps, respectively. All Glassware will be available under these high-level categories. Note that these categories won't know the difference between Glassware that's native or exclusively cloud-based, so you may wind up sharing space next to a major software publisher as well as a garage operation looking for a break. Remember—actions not apps still rule here, even if you're thinking about how to write an app.

Updating Releases, Versioning, and Crash Reports

A major technical advantage of having your app approved by Google and listed in MyGlass is having any updates you push to those who've installed it be applied silently, just like many apps do through Google Play. Users won't be notified that they have to download a new version, it just gets pushed to their Glass headset automatically through the MyGlass mobile app. But it's kind of a black box at the moment, as we're currently unable to review modifications in Glassware between different versions and there aren't any changelogs we can consult. New features or bug fixes show up magically, but we're unaware of any patches or feature additions until we see/stumble across them.

Some vendors list ad hoc changelogs on their own websites outside MyGlass, but that's not universally implemented. As the GDK platform matures, we hope to see a central facility in MyGlass or Google Play or something else that lets us track changes, if for no other reason than people like us are obsessed with knowing “What's New.”

Additionally, MyGlass doesn't provide automatic crash reporting back to Glassware publishers like Google Play does. However, if Google detects that your app is crashing with some regularity, it will reach out to you and work with you to resolve any issues your app might be having. The bright side is that the review process for apps looking to make it into the MyGlass catalog is very thorough and many crashes should be detected in that step. You also have the benefit of having to build for essentially only one device, as opposed to multiple OEM devices.

You're still able to make use of third-party tool for crash analysis.

Porting Existing Apps to Glass: DON'T

Developers maintaining existing software platforms are foaming at the mouth waiting to start building for Glass. Undoubtedly a ton of cool ideas are being cobbled together as you read this from innovative thinkers wanting to fully use Glass as a platform and notable third-party applications will be rushing to use Glass as a frontend for their services, in addition to every other conceivable digital platform around. That said, *if you're just thinking you can do a straight port of your codebase, you're doing it wrong.*

Glass is a unique device that requires a unique experience. To support this notion, **Google's documentation stresses** that when creating immersions, “Design interactions that make sense on Glass instead of porting over activities from other Android devices.” That's pretty much the motivation for Glassware design and the Think for Glass philosophy in general.

Aside from the partners Google's worked with already to offer apps at launch, it's a safe bet that household brands like Instagram, Flipboard, Mint, Dropbox, and Spotify will build Glass clients, even if the UX doesn't exactly equate to a seamless translation for their stuff. And this is where the big creative opportunity lies. Don't just port your code—extend your platform and really explore the interface that Glass provides. Make something that becomes a part of your experience in a completely unique way. Remember, Glass complements your smartphone, not replaces it, so don't just build yet another client—if you support a game, do a hands-free controller or heads-up display. Make using it cool!

Just like you had to relearn design for mobile apps, porting to Glass doesn't necessarily mean a full and seamless translation of every feature. And this pragmatism is key in learning how to Think for Glass. It makes for a fun challenge.

You might want to take a pause from this chapter for a moment and re-review the material in our earlier chapter in the *Discover* section about what Glass *isn't*, and see how all the knowledge you've gained and ideas you've developed interplay with those constraints. After you've given yourself a refresher, think about how the Glass experience would handle your current UX. The onus is upon developers to repurpose their ideas within a new environment—the right way, not crudely forcing a smartphone app onto a wearable. People by and large still won't want to stare at Glass for extended periods of time, so *keep things glanceable* and continue to think in terms of *actions, not apps*. Many of the widgets and UI paradigms just don't translate to this new venue.

The bottom line is that presence is critical and a proper design plan is the major step in achieving that, not just retrofitting current services. If your translation and/or extension of your platform is truly good, you'll have a real winner on your hands. And happy users.

So Which Framework Is for Me?

We've come at last to the moment of truth. You're well-read on the Mirror API and how to build cloud-dependent services against Google's RESTful backend. And now you know how things work natively, too. When two paths diverge, how do you know which is the right one for your project? The simple \$0.05 answer is *use the right tool for the right job*. Don't let everything you learned in the *Design* part of this book go out the window. It's actually not that hard to do—just figure out what usability components are critical to applying your idea (connectivity, sensors, etc.), what interface elements could best be used to display data and gather input, and how the usability of the application fits into the model of a microinteraction.

Don't simply retrofit an idea with the wrong framework, because that flaw will show and your usability will suffer. Experienced Glass users wise up to what models work best for the platform, and even newbies will quickly figure out that if they can't use your Glassware in their daily lives, they're drop it like a bad cold. Better yet, figure out how to make your idea exist on both as a hybrid application.

You could also frame the argument in terms of the impacts on deployment—the traditional model of rigorously testing to stamp out as many bugs in native programs still holds true, but can be time consuming and creates delays between version releases, which then have to be pushed out through a repository or app store. Traditionally this has meant alerting the user that there's a new version of the app available and then having her download it, or as Chrome has achieved in recent years, applying silent updates. This, after all, is the idea that makes updates to the Glass OS work so fluidly. We'd ultimately like to see this be the case for native apps on Glass, which solves a lot of the disconnect issues between software vendors and users when new versions are ready.

Even within the GDK itself, you might find yourself lost at first in deciding whether to commit to a live card or an immersion. Generally speaking, live cards tend to have a longer shelflife than immersions, but immersions require complete user attention while they're running, and don't permit integration with the timeline. In this light, we see immersions as the exception instead of the rule. Many people defer to using immersions, but they generally aren't the way to go.

Real-world examples of apps that fit the live card category are implementations of media players like Pandora and ViewTube for Glass, time-sensitive actions like AllTheCooks Recipes, scheduling services like to-do applications, telephony applications, and others. Great examples of immersion-based apps are games, search services, applications that make various uses of the camera, location-oriented apps like driving utilities, and video tutorials. There's room for innovation here though, as we're seeing some very clever uses of both UI methods to deliver really effective experiences like **Zombies, Run!** (an immersion) and **Battery Checker** (a live card).

Learn the best practices and patterns, see what others have done, play with some code samples, and then let your own creativity drive your app.

So how do you use the Glassware development model to solve problems? Do you start with the frameworks and then figure out how to use the unique features of each to write an app...or do you begin with an existing domain space and then pick the most appropriate framework? There really isn't a cut and dry answer. The best thing to do would be to write Glassware in the Mirror API and the GDK and get a solid feel for the development process and turnaround cycle.

You may find yourself needing one...or both.

Getting on MyGlass: Glassware Submission, Review, and Distribution

Welcome to the end of the line! (Well, almost.) Now that you know everything about the Glass ecosystem and Google's vision for wearable computing, it's time to get you ready to play the main room. You've committed to memory the proper steps needed to design highly effective wearable applications and have thoroughly mastered the techniques to build rich functionality for the Glass experience, tying it all together with the Think for Glass philosophy. You've reached a critical point following design and development: getting your Glassware submitted for review. This is the all-important final step so your project can be listed in the MyGlass directory so that users can easily discover it and start using your Glassware.

And *getting on MyGlass should be the destination you aim for after building an amazing product, as it yields the greatest rewards*. So in this chapter we're going to complete the cycle and show you how to get there in as little time as possible (as we've done with the rest of this book) by preparing you for the review phase.

Whether you work at a professional software shop using formal lean product management methodologies or you happen to just be a solo hobbyist who hacks and hacks and hacks until things work right and look good, know this: *the more organized you and your project are, the faster you'll be approved and listed in MyGlass*. It's that simple.

So let's get it together, and let's go. There's a place waiting in MyGlass with your Glassware's name on it!

Making Your Awesome Glassware Even More Awesome

We need to stress right off the bat that Google's review process isn't a shakedown, and the Glass team isn't a heavy-handed gatekeeper. The goal is to help you create excellent Glassware and make sure your project follows best practice guidelines. All projects are

welcome, providing they demonstrate the form and function of winning Glassware, abiding by the [Glass Platform Developer Policies](#) and delivering a great wearable experience. The process lays a solid foundation of amazing programs that set the tone for legions more to follow, giving developers inspiration and users uniformity, variety, and choice.

That said, you needn't lose any sleep over compliance: this isn't a federal subpoena to testify in court—more a run-through to make sure that you stuck to the major principles laid out in Google's documentation for design and programming patterns. Google measures various aspects of your Glassware to ensure it performs well, properly conforms with the Glass UX, effectively (but not egregiously) promotes your brand, and achieves the goal of being microinteraction-focused. And by this point in the book, you're well-equipped to pass the test with flying colors.

The Objective of Glassware Review

If you've published an Android application to Google Play or other app stores before, the requirements and process to publish your Glassware should be somewhat familiar. In truth, the phase of evaluating your Glassware and getting it ready for the masses is more reminiscent of the early days of Apple's App Store than the process of submitting work to Google Play or the Amazon Appstore. However, while the sequence of events that is involved in testing, assessing, and making recommendations about your Glassware is more involved and takes longer than an Android app submitted to Google Play, the Glass team isn't taking a massive virtual red pen to your work. They *want* to list you and help you succeed, and the review is all about making your Glassware the very best it can be.

That said, this process is still new and continues to evolve, so you may discover comments or suggested changes that strangely contradict what you've been taught to be the best way to build great wearable applications and services, even teetering on hypocrisy against Google's own documentation. This is normal, so don't freak out. It's part of the fun of living on the bleeding edge of technology.

Remember, this is a learning experience for everyone, and your very participation makes things better.

What MyGlass Gets You

Aside from the obvious prime exposure and promotional plug you'll have by being in MyGlass as an officially approved Glassware project, there are several technical advantages that we've touched on in earlier chapters. If you're looking to build a business and monetize your wearable projects, integrate with other applications you've created, or incorporate other components within Google's ecosystem, being on MyGlass is the way to go:

Expanded Mirror API quota

Google works with you and helps estimate how many more daily API calls you'll need beyond the default 1,000 once you're let past the velvet rope. This assessment is typically based on several factors, including how many calls your Glassware currently makes and the number of average transactions per user your Glassware consumes. You'll also be able to request a specific number of calls per day when you submit your Glassware for review.

Support for GDK authentication

You'll be able to allow users to log in to third-party web services and use remote data in your native app once Google uploads your project's APK onto MyGlass for your private testing.

Whitelisting

If your project isn't meant to be public Glassware, you can request a project to be granted private access, in which case a privileged Google Groups or a Google Apps domain you specify will be whitelisted to access it.

Managed installation

Installing native Glassware is handled entirely by MyGlass. Once your project is listed, users need only enable it from MyGlass and your app will be sent to their device over the air. Glassware vendors distributing outside of MyGlass have forced their users to use ADB to sideload APKs onto Glass or have built separate Windows Installer programs, which is more work and can get messy.

Versioning and distribution

MyGlass also manages pushing any updates you publish to users silently, without any prompts to redownload or reinstall.

Support from Google

The conversation doesn't end once you get on MyGlass. You'll be in the pipeline for any new announcements and platform enhancements, and the Glass team has been known to feature projects as case studies to help other aspiring Glassware creators.

Prereview Activities

So you've prepared yourself for the slings and arrows of constructive criticism and are willing to apply the recommended changes. Let's give you an overview of how you should get ready before you even start the submission process.

The truth of the matter is that it will require a tad more work to get your creation into MyGlass post-development, but the upside is that Google's laid out everything for you. The **Distribution section** of the developer documentation maps out all the prep work you'll need and the action items you'll be taking prior to seeing your Glassware's icon appear right alongside Facebook, Twitter, Evernote, and others. You'll be needing several

graphics to use for the branding of your project. These include art assets used within the Glassware itself as well as screenshots and icons to be featured in MyGlass.

As you should have gathered from this book, *design* is a big part of making sure your Glassware is a hit. Luke Baran, who built the streaming audio Glassware for **Boston NPR station WBUR**, the first Glassware application that delivered live radio content, wrote that **you'd be best suited to submit an app sooner than later**, as the review process doesn't explicitly mean you have to submit an ironclad finished app. It can be a prototype that you're building along the way. *Once you're ready for your handiwork to be beta tested, you're ready to submit.*

During review, you can continue to submit updated releases of your Glassware—making changes to your server backend if you're using the Mirror API or submitting signed APK versions for GDK-based projects. If you've got uber-black ops trade secrets you'd rather not share, don't worry. This isn't a code review, but more of an analysis to see how your Glassware applies the Glass experience.

Since aesthetics are a—and debatably, *the*—major component of the review process, you can help your reviewers along by including wireframe of your project using **Glassware Flow Designer**, which we talked about in **Chapter 13**. Visually laying everything out helps the team understand what you're going for and can identify areas of improvement. They'll still thoroughly test your stuff—but various developers, including some that have worked on more than one project, have indicated that including a flowchart of their Glassware has sped the approval process along considerably.

Things to Think About Before You Submit

There are some additional considerations you'll need to make depending on the type of Glassware you've built relative to *scalability, security, and performance*. If you're working with Mirror API Glassware, you'll want to be confident about your infrastructure's ability to handle potentially large request loads, and that your web server doesn't collapse during high-volume periods due to a sudden surge. This won't be an issue if your server is based on Google App Engine or some other platform-as-a-service provider, but it is something to keep in mind with the Glass-owning public constantly expanding.

If you're using a web host that charges you for additional resources you consume (leaving you with a potentially massive invoice for bandwidth) or you're self-hosting a server, this may make you reevaluate how your Glassware talks to the cloud and uses the APIs. Further, it may force you to think about how to handle downtime. This is the type of advice that you might get from Google once you submit your Glassware and review starts.

Even though Glass communicates with the cloud over HTTPS, you want to consider how bulletproof your security is with authorization/authentication and for any data

stores and handling items like passwords, tokens, and other types of sensitive information. Do you have a contingency plan in place in the event your security is compromised? Google’s policies prohibit the collection, storage, or sharing of confidential user data like Social Security numbers, which the documentation details in full.

In terms of being performant, how might you be able to make your Glassware any more optimized? If you’re working with GDK Glassware, having a high-performance app is a must. Tasks that could be GPU-dependent like animation loops or back buffers, in addition to physics engines, visual transitions between screens, or other jobs with a substantial effect on the processor and battery, which cause Glass to heat up, should be examined. Of likewise importance is how you handle various multimedia formats and processes spawned outside of the main UI thread like networking, downloading large resources, syncing, or I/O.

And although you can submit Glassware without error handling for every type of nuisance that may occur, you do want to demonstrate that you’ve prepared steps to act during cases where Mirror API services seem to unresponsively hang to no avail, or GDK apps outright crash. How do you report these situations to the users, how do you shut down any running services, and how do you recover so they hopefully won’t happen again?



Pandora Adjusts Its Strategy for Its Glassware

As a practical example of publishers changing things up to meet wearable demands, Pandora had to withhold using the auto-inserted advertising feature that’s normally part of its streaming audio service for its freemium web and mobile clients when implementing its Glassware. At the time of its release including or serving ads in Glassware was strictly forbidden by Google, so to be approved, the ad feature wasn’t implemented.

(Pandora also did this years earlier with its app for the original Google TV platform.)

But again, don’t feel as if you’ve got to spend hundreds of manhours slaving away configuring caching headers, tweaking CDN configurations, getting that one pesky Java method to run just a few milliseconds faster, or stamping out bugs prior to submitting your work. The best part about Glassware review is that Google encourages you to turn in a project that’s early in the design phase to get the right start. (Remember, this is the company that lives, breathes, sleeps, and dreams the “*release early, release often*” mantra.) You do want to continuously improve your software, but having a working prototype that’s generally usable is good enough.

It’s perfectly acceptable to stay in beta.

How About Localization?

One thing developers have brought up a lot—notably those working with content-focused Glassware—is how they can make Glassware with international flare; they are looking to support languages other than English. The Glassware Launch Checklist states, “Glassware and its related descriptions must be in English by default. Multiple languages are okay if there is complete feature parity between languages. The review process will support new languages as they become available on Glass.”

When submitting your Glassware, make sure to indicate all of the languages that your project intends to support and review the material from [Chapter 12](#) so Google can check to make sure your resources are serving the appropriate content properly, and get it in the hands of affected users. In this regard, you’re directly influential in expanding Glass to more countries! Let Google know what specific people you’re trying to reach, and it just might open up access there.

Submitting Your Glassware

So now you’re ready to begin the submission process, which has you fill out a web form that describes your Glassware, its format, and how to control it. This last step is quality assurance to make sure you play by Google’s rules. The first thing you need to do is again do a quick once-over of the [Glass Platform Developer Policies](#), just to make sure you’re not erring in the way you populate, distribute, and promote your new product.

Second, go *slowly* through the [Glassware Launch Checklist](#) item by item. The form is highlighted in [Figure 14-1](#). This ensures your Glassware has all of its technical requirements in order. This runs the gamut of everything from the naming of your Glassware, the legal use of Glass art assets and other Google intellectual property, to graphic sizes and formats. These may seem trivial, but they can trip up groups that are more focused on engineering and less on marketing and legal matters.

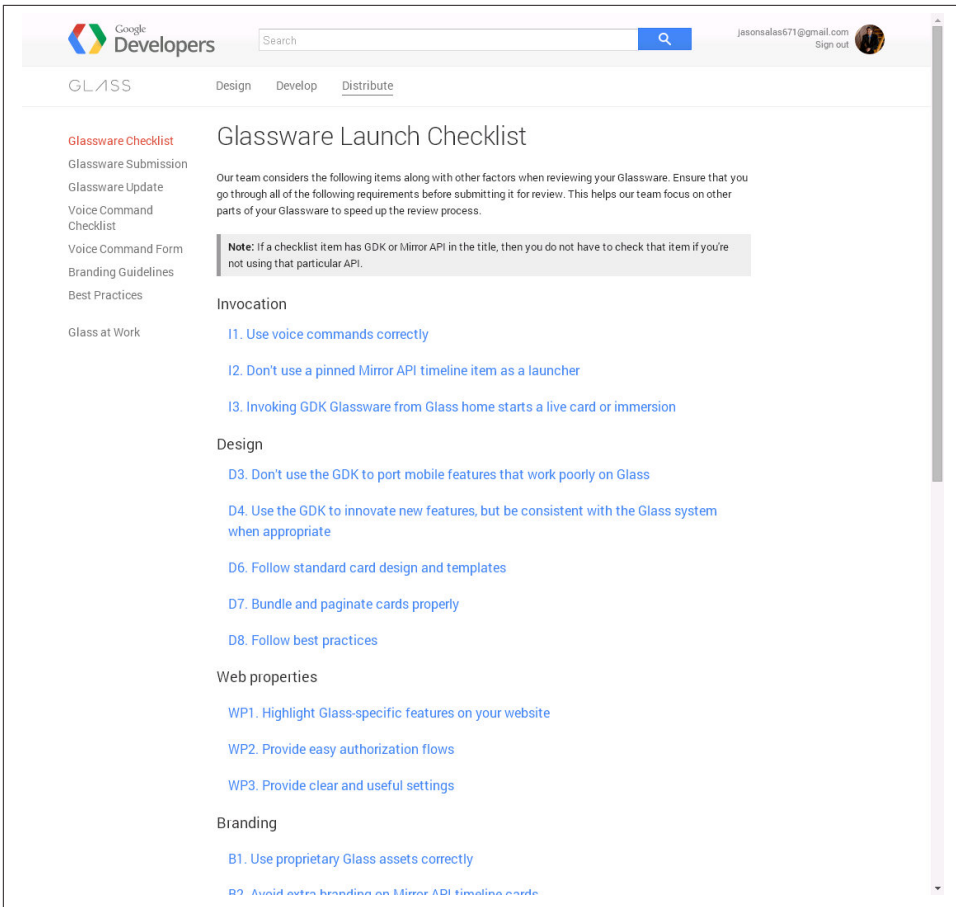


Figure 14-1. The Glassware Launch Checklist



Do I Have to Be Listed Publicly?

You can request to not have your Glassware listed publicly in My-Glass in cases when you're publishing privileged applications or services for organizations like businesses, schools, churches, or families, and don't want them available to the entire Internet. If you wish this to be the case, as [Figure 14-2](#) shows, when submitting you can provide a Google Groups mailing list or a Google Apps domain for private access to your Glassware.

Google Developers

Search

jasonsalas671@gmail.com Sign out

GLASS Design Develop Distribute

Glassware Checklist

Glassware Submission

Glassware Update

Voice Command Checklist

Voice Command Form

Branding Guidelines

Best Practices

Glass at Work

Glassware Review Request

Use this form to request a review of your Glassware. In addition to providing details and assets, you may request an increase in API quota and to have the Glassware listed publicly. You must test your Glassware on Glass before submitting this form. For more information on the review process and this form, please see the Distributing Documentation.

<https://developers.google.com/glass/distribute/>

* Required

Primary contact name *
The primary contact's name. This should be someone who can answer questions about the information provided here.

Primary contact email *
The primary contact's email.

Content shown in Glassware *

I understand that I am responsible for the licensing rights for all rights for all content I source and/or include in my Glassware at all times.

Glassware image assets *

I understand that I am responsible for the licensing rights for all images I submit to this form.

Mailing list *

I agree to be placed on the Google Glass Platform Updates mailing list.

Glassware maintenance *

I understand the Google Mirror API is in developer preview and the GDK is in sneak peek. Both will change frequently. I agree to keep current with platform changes and update this Glassware as appropriate. Failure to do so may result in reduced quota and removal of public listing.

Does this Glassware involve sharing user information with other users or entities? *

Yes

No

Does this Glassware use location information? *

Yes

No

Does this Glassware require or accept voice commands? *

Yes

No

Figure 14-2. Requesting private Glassware

Gotchas

There are a couple of items that catch even the savviest of developers all the time. The Developer Policies state that when users delete their account with your Glassware, with Google, or any links between the two, “You must delete all personal information you obtained from the Google API relating to them,” so handling such situations is a must. This also plays into the tips in “Delete Versus Dismiss” on page 82 that dealt with users deleting data within your Glassware. Also, Google’s grammatical rules for using various forms of “Glass” in context is something you need to intimately familiarize yourself with. You may use “...for Glass” in the title of your Glassware, but not “Glass...”

Submit!

Barring any major issues you have with either the Developer Policies or the Checklist, head on over to the [Glassware Review Request](#) page. You'll need to provide URLs for content areas that will populate your MyGlass profile page, including your terms of use and a privacy policy, contacts, multimedia assets, a support email address, and possibly other resources. You'll also need to describe how your Glassware makes use of the GDK APIs, if applicable (Mirror API Glassware tends to have a bit more public-facing information while installed GDK apps require an additional screen to detail some of their requirements).

Gmail's MyGlass profile page lists links for contacts, support, and more in the bottom-right corner ([Figure 14-3](#)).

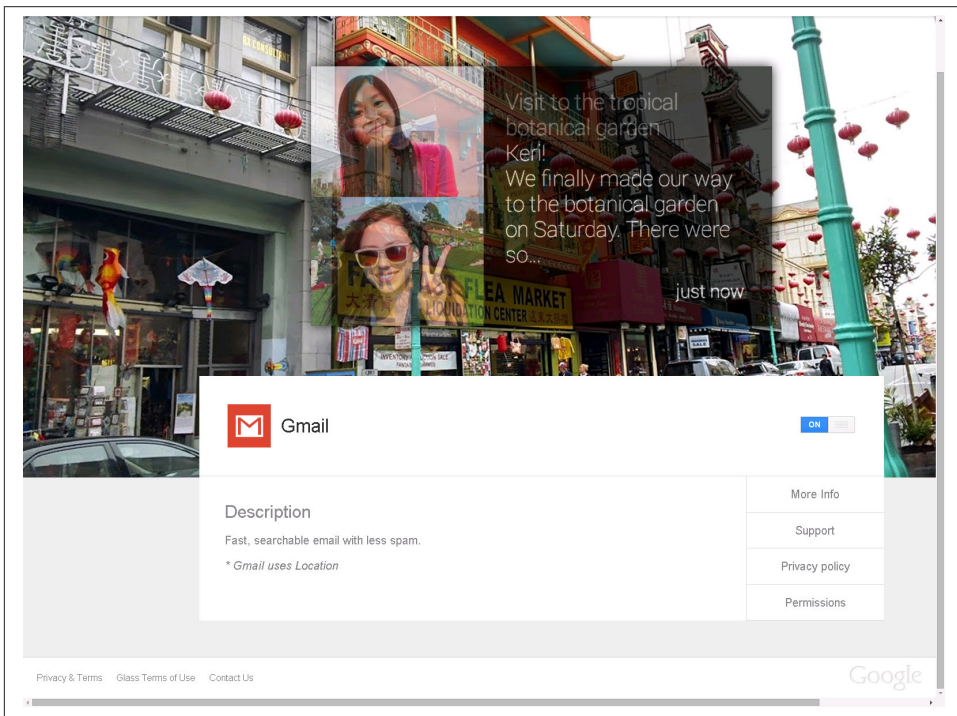


Figure 14-3. A detail page on MyGlass

You can also request a specific volume of Mirror API calls that your Glassware can make daily and whether your Glassware will be listed publicly or privately ([Figure 14-4](#)).

Google Developers [Sign out](#)

GLASS Design Develop Distribute

Glassware Checklist
Glassware Submission * Required
 Glassware Update
 Voice Command Checklist
 Voice Command Form
 Branding Guidelines
 Best Practices
 Glass at Work

Glassware Review Request

Step 3: Display Assets

The asset collection should contain the following files, where (name) should be replaced with your Glassware name (use "_" in place of any spaces).

```
(name)_icon_white_20.png
(name)_icon_white_30.png
(name)_icon_white_40.png
(name)_icon_white_50.png
(name)_icon_white_80.png
(name)_icon_white_150.png
(name)_tile_color_640x360.jpg
```

Please see an example here:
<https://developers.google.com/glass/tools-downloads/compass.zip>

Additional details, including a description of each asset and what it is used for is available here:
https://developers.google.com/glass/distribute/checklist#myglass_assets

Please note, during the approval process the review team will ask for additional assets.

Tile image

Asset collection URL *
 URL to a collection of assets for your Glassware.

Brand color *
 A color representing the Glassware or brand, specified as hexadecimal RGB. For example, "#CC0005" or "#0005CC".

Figure 14-4. Display assets

Lastly, you can indicate what type of feedback would be most helpful to you (Figure 14-5). These include UI, UX, or even writing effective press releases for your Glassware. Don't be shy—this is free advice. Think of it as an early present!

Google Developers

Search

jasonsatae671@gmail.com Sign out

GLASS Design Develop Distribute

Glassware Checklist

Glassware Review Request

Step 6: Help us help you!

Please rate the resources you found the most helpful when developing this Glassware
Do not rate resources that you did not use.

	N/A (didn't use this resource)	1 (not helpful)	2	3 (neutral)	4	5 (super helpful)
developers.google.com/glass	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
StackOverflow	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Local Glass meetup	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Local Google Developer Group	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Glass Design Sprint	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Glass Developer Office Hours	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

What kind of feedback from the Glassware review team will help you the most?

- User interface design
- User experience design
- Voice commands
- Authentication flow
- User settings
- Assets
- Press release strategy

« Back | Submit

Never submit passwords through Google Forms.

100%: You made it.

Figure 14-5. Requesting additional feedback from Google

The Review Process

So now, you're theoretically under review. We know what you're thinking—HOW LONG WILL THIS TAKE UNTIL I'M APPROVED???. The definitive answer about your unique timetable: *it depends*. Don't go investing in blood pressure medication just yet or hair color product to cover all the gray that's suddenly sprouted up—this isn't a stressful period where the seconds feel like days. The process is collaborative, iterative, and fun! Seriously folks, don't go issuing press releases or making public statements... yet. If you need to time it with something else, discuss that with your review team (we'll get to this in a moment). But remember that your review will take time.

After sending in your Glassware's profile, you should get word back from your review team very quickly as a general introduction. But streamlining this approval rides on:

- How clear your project's description was in outlining your experience
- The complexity of your Glassware
- How promptly and correctly you apply the suggestions Google sends you

What Google looks for during the review process isn't a tightly kept secret, and it openly shares the major criteria against which your Glassware is compared:

- Is your Glassware safe?
- Does your Glassware do what it says it does in your description?
- Is it built for Glass?

The best thing about the review process is that it's not subject to subjectivity. The entire Glass team has received the same training, looks for the same areas of concern, and emphasizes the same winning aspects of wearable applications. The review process is a team event, so things tend to be done by committee, and the suggestions you get are consensus from several people. You can rest assured you're getting the most qualified, most well-rounded opinions about how to make your Glassware shine. Plus, at the end of the day the Googlers are just end users like us and you, so they're working to create something they'll, use, too!

Does Mirror Glassware Get Approved Any Faster Than GDK Glassware?

We all know that the big advantage of working with the web stack versus programming natively is a much faster turnaround time, rapid deployment, and next-refresh updating. However, this doesn't mean that native Glassware will take significantly longer to be reviewed and approved than Glassware that's strictly cloud-based. The review timetable is the same no matter which developmental framework you use—Mirror or GDK. If you're looking to get something to market quickly, make sure you follow the steps outlined in this chapter and stick to the Glassware Launch Checklist. That's the key to a quick turnaround—not the framework you chose.

And in case you're wondering, you don't get to cut in line in front of other people whose Glassware is in review if you've already got Glassware listed in MyGlass or Android apps listed in Google Play. Membership in this case doesn't have such privileges. But your prior experience should guide you through review to move quickly to approval.

The feedback you receive is typically a list of items in a spreadsheet that detail out the changes you might want to make to achieve the best possible Glass experience. Google won't dictate what you can and can't implement technically—your idea, your innovation, your vision, and your creativity will be preserved. You'll likely have several items to consider, largely from design and performance perspectives. These may be general suggestions about the arrangement of menu items, the use of input controls like gestures and touch and voice, how the Glassware handles microinteractions, the application of a particular UI element, branding elements, recommendations for the handling of multimedia, technical tidbits like ensuring processes are properly killed after live cards are stopped, or placement of graphics. These tend to be very specific.

Depending on the difficulty of the changes and how fast you can get them applied, you'll receive further feedback as your Glassware moves along the virtual assembly line. This process may span several rounds as your reviewers use your Glassware more and get a better understanding of what you're trying to do. You're more than free to dispute these suggestions and defend your decision to design things in a certain way...the comments you're sent aren't based on rules that are set in stone, and if you feel something you did is right, *don't be afraid to stand your ground*.

Timing Your Release

If you're overseeing the launch of a large application ecosystem that has separate applications for Glass, Android Wear, other various flavors of Android (for handhelds/tablets, vehicles, or TVs), web, Chromecast, Chrome extensions, etc. and you're shooting to roll out all versions of the service at once, it would be wise at this point not to commit to a certain launch date. Synergizing all platforms for a single launch is very tough, with Glass being the slowest to be available to market because of the review phase. Our best advice for trying to pull off a coordinated launch would be to get your Glassware in first and under review, then ask the team for a projected date once they indicate you're nearing approval. You can then submit to Google's other channels and be ready within hours.

You can launch and announce everything in a grand-and-glorious blitz once everything's available.

Categorical Listings

You don't need to concern yourself for the moment about how your Glassware will look once "on the shelf" in MyGlass. Once the volume of approved Glassware hits the level when it can be listed categorically, you'll be able to better directly appeal to your target audience. For now, expect your placement to be alphabetical—you may have built a insanely cool game you titled "Snail Trail," and by virtue of its naming wind up appearing right next to Sky Map, Star Chart, and Shazam. You'd obviously be better suited next to Glassware in the wearable gaming genre, but that's what we have now. It's not perfect, but this will undoubtedly improve with scale.

MyGlass also doesn't have a ratings system yet like other app stores, so your work won't be able to generate the grassroots buzz it might have as people endorse your work (on the flip side of the coin, folks can't unfairly bash the fruits of your labor and have such criticism stand out more). Again, we hope to see this improve with time and volume.

App Analytics

One thing that astute developers will pore over tirelessly is traffic statistics and usage data. We mentioned in [Chapter 3](#) the need for formal, structured insight metrics, so we

hope at some point Google Analytics is available to bundle with Glass products. Google Play Services also isn't available as part of the GDK like it is for Android mobile apps, so that's also on our wish list. For now though, you're able to take advantage of a few tools that gauge how your Glassware is holding up once in the wild.

If your infrastructure runs on Google Cloud Platform, such as relying on Compute Engine or Google App Engine, you'll have access to technical breakdowns to measure responses to requests, to monitor basic usage for things like time-of-day requests, errors, API calls, load, caching, and other metrics. The [App Statistics dashboard](#) for your project in Google Developers Console has great data you'll find helpful, in lieu of more general reports with data you can use for marketing.

Marketing Channels

We highly encourage you to be as shamelessly self-promotional as you can with your marketing in all forms and to network like crazy. Register your own domain name. Create a Facebook brand page or even a designated group so people can talk with/about you. Have a podcast touting your latest features and upcoming developments. Do meet-ups with your fans. Let people congregate in praise of your innovation. It's all good. Numerous blogs and social outlets cover the Glass space and sniff out any new Glassware that come on the scene...minutes after they've been published. Don't let the free press go to waste!

Best of all, the Glassware development community is thriving and passionate and continues to grow everyday. Many of us cross over to Android Wear work and we've mentioned in this book just some of the many friends we've made who are doing amazing integration with other systems. The hallmark of a great Glass developer and a big part of Think for Glass is passing on knowledge to newbies just taking up the practice, so always try to make time to pay it forward and give people who are now your colleagues some timely pointers.

Monetization

Lastly, let's talk about the one thing that's been on your mind probably since the first page of [Chapter 1: How can I make money off my Glassware?](#) You can only build Glassware for so long before the shine of public adoration without compensation begins to wear off (trust us, we both know). The Glass Platform Developer Policies don't allow for traditional monetization techniques to be applied, and anyone caught trying to do so in their Glassware won't be approved and could have their access suspended. We strongly urge you not to test this.

Serving advertisements is a no-no, and you can't charge fees for accessing or using your Glassware. And we're irreconcilably split on how we see some form of AdWords ultimately being available within Glass. Jason likes the idea of having contextually accurate ads and/or commercials appear in a user's periphery and would like to see the oppor-

tunities it spawns; Allen believes that with Glass being ultra-personal, shoving ads in your eyeballs is a poor experience and is contrary to the platform's design goals. "Everyone, including Google," Allen maintains, "is trying to figure out the right way to do this."

The ultimate solution to be revealed might be a form of micropayments, perhaps for in-app purchases, rental charges, or on-demand Glassware. One thing is certain: monetization *will* come in time and in some capacity, and you *will* be able to profit from your work. Just be patient and stay ready for it.

Reflections on the Future

The most intriguing thing about Google Glass is that it's just getting started. The promise of all the things Google's flagship wearable computing platform already is and the exciting prospects of what it can become are nothing short of awe-inspiring. Do a search for "Google Glass use cases" and it'll take you an hour just to go through the results. From general purpose to profession-specific, from the ultra-cool to the downright ludicrous, the sky's the limit for how people are going to put Glass to work.

As we've discussed, a big portion of society speculated, dreamed about, praised, criticized, and condemned the product for more than two years before it even came out. Few products in history have achieved this sustained level of interest that far out from launch.

Throughout this book we've presented concepts, code, design tips, and our philosophy to help ensure you enjoy a long and healthy life with your technology investment and give your Glassware a powerful stage on which to succeed. This final chapter discusses some advanced uses for Glass and what lies ahead for the platform.

Corporate Glass

One of the questions that invariably pops up when considering Glass is how businesses will deal with their employees using it in the workplace. These days the lines between high-performance business machines and off-the-shelf consumer tech are completely blurred, and companies are outfitting their staffers with the very same smartphones, tablets, and laptops that everyday consumers stand in line for and pick up at popular retailers. The BYOD ("bring your own device") concept, thought to be anathema 15 years ago, is gaining popularity these days, and is even encouraged in many other operations across industry. Staffers are free to source their own gear that gets outfitted with corporate access controls and internal applications, distributed and running safely and securely outside of the public Glass ecosystem.

Glass can be a boom for productivity since it's so nonintrusive; or it can serve as the bane of an IT department's existence in being yet another node joining a wireless network, using resources and distracting staffers from actually getting stuff done. It's a legitimate quality control concern. Many organizations—notably government agencies and the military—restrict access to several types of applications they allow on their networks like videoconferencing, instant messaging, various streaming media formats, and many other services.

So a big challenge—and opportunity—is clearly on the horizon for integration and Glassware development. Since all syncing traffic for services based on the Google Mirror API takes place in quasi-real time (after a request/response roundtrip to the service), how can this be filtered to allow some Glassware, like news updates, stock prices, and messaging, to persist throughout the workday, while disallowing notification distractions from other Glassware like games, memes, and porn? Large organizations may prefer the closed model of installed apps written with the GDK so that they control all aspects of the communications chain and won't have to rely on Google's system as middleman for things like company-wide messaging. Or, they may prefer the managed, cloud-oriented environment of the Mirror API with its simple UI, inherent security, and known protocols, and then outright block native Glass apps.

Google itself even created the **Glass at Work** initiative to help spread knowledge about how Glass is being used by workers in different professions, to demonstrate how developers are creating custom solutions within industry, and to stimulate businesses integrating wearable electronics into their operations.

Will a new generation of wearable computing devices spearheaded by Glass assist workers and let them complete more tasks more quickly, therefore being a valuable tool, or will it be yet another time vampire, preventing work from actually getting done? That part's up to you. Google's done all the engineering for the product; the social rules of how it can be used within offices, schools, churches, buildings, and other places need to be developed, put into practice, and enforced. Remember, a key element of how to Think for Glass is getting others to understand what it is, and is not.

Streamlining Operations

Pennsylvania-based **Fiberlink** extended its MaaS360 mobile monitoring and administrative application to support Glass by writing Glassware, empowering IT managers to control wireless devices on their networks hands-free and at any time by seeing all connected staffers' devices across their network in timeline cards. The service includes a slick feature that uses a wearer's headset to locate a lost device and display its whereabouts on a map, giving an administrator the option to lock it down or wipe the device completely.

Glass also has great potential for the laborer not bound to a desk. **Augmate**, a development shop in New York City, is using Glass to connect enterprise backends with digital eyewear to streamline workflows and business processes for field workers, with applications developed using the Glass UX for the automotive, manufacturing, construction, and aviation industries, among others. Imagine your UPS deliveryman scanning your package as he drops it off with Glass. What could Brown do for you then?

Glass in Medicine and Education

Practically every industry and sector is at least considering how Glass might work for them. While opportunity abounds for the startup community and existing brands looking to add another cash cow to their stables by creating can't-miss Glassware, the most emerging areas with inarguably the largest implications on society are healthcare and education. Glass as a communications tool for students and faculty, both for remote learning and for enhancing in-class instruction, is booming at the elementary, secondary, and collegiate levels.

Northeastern University in Boston created a college course on Glass as a driver for healthcare innovation. Similarly, USC has developed coursework in how Glass can transform journalism, and the Glass Creative Collective is a partnership between Google and several design and film schools to advance the craft of visual storytelling. Several daring instructors have used Glass for telepresence and advanced videoconferencing, such as Michigan physics teacher Andrew Vanden Heuvel, who remotely took his students on a once-in-a-lifetime **virtual tour of CERN** via the Hangouts Glassware.

Healthcare practitioners are pushing the ecosystem forward by diagnosing illnesses, treating patients, and collaborating on research, with several surgical procedures now documented from the Glass first-person POV as a teaching tool to extend the audience normally restricted to the surgical gallery and demonstrate live cases with colleagues. Healthcare technologist Dr. Rafael Grossman broke new ground when streaming his work during the insertion of a feeding tube into a patient of his via Hangouts. The Ohio State University **soon after used Glass for telemedicine** when Dr. Christopher Kaeding performed reconstructive knee surgery, sharing the live video with colleagues via Hangouts and chatting with students and colleagues about the technique in real time, while not endangering the patient or requiring the doctor to constantly step away from the operating room. Whereas videoconferencing tools historically involved stationary cameras and monitors at awkward angles, participants literally saw the procedure through his eyes. In addition to Hangouts and the ability of Glass to facilitate video calls, surgeons have been able to review x-rays, radiology reports, and other forms of medical imaging pushed to their HUDs mid-procedure.

It wasn't too long before other physicians followed suit and started sharing their knowledge, too. As of the time of this writing, Glass has been used in a number of anatomical procedures and on three continents.

And in Australia, Glassware is being developed to provide hands-free tutorials for new mothers for proper breastfeeding.

Other physicians and clinics have been asking for Glass to help eliminate the need for clipboard-based patient charts by digitizing data and making it contextual as hospital caregivers make their rounds. Similarly, consultations are easier with video calls. Further still are the healthcare providers that are working on ideas to use Glass to communicate with patients, letting people in trouble make video calls, or being able to relay vital signs to medical responders through timeline cards through sensors. Philips Healthcare is putting a lot of R&D effort into **using Glass for telemetry applications**. Pharmacists are looking to Glass as a way to expedite the filling of prescription orders, which is one of several concepts that Glass Explorers Chris Vukin and Thomas Schwartz are building with evermed, a suite of products to assist practitioners and patients alike. **A touching video** showed how Children's Memorial Hermann Hospital used Glass to let patients take a virtual tour of the Houston Zoo. And we've covered the booming space that is fitness Glassware.

On that note, much work is being done as to using Glass as a part of sports medicine and for the treatment of athletic injuries, as well as for broader use in trauma and triage situations for ER professionals, EMTs, and possibly even field combat medics.

Further, how might this also be used for other medical disciplines like chiropractic care, physical therapy, dentistry, plastic surgery, psychology, or psychiatry? Could a marriage counselor analyzing clients on the couch possibly use a Glassware-based polygraph, essentially a wearable lie detector app, to pick up on patterns in the couple's voices to assess elevated stress levels, which would aide in their treatment? Could a person recovering from an injury going through exercises have their biolevels sent in real time to a telemetry center for analysis to track progress? We certainly hope so.

But of course, the implementation of these ideas in industries isn't trivial. Concerns over privacy and security, HIPPA stipulations, issues involving interacting with young people, endangerment, and skepticism from the medical and educational communities will have to be addressed. Many organizations involved in health have taken the Glass hardware and removed all system software, installing their own proprietary Android forks based on the source Google released (as detailed in the previous chapter), to comply with federal regulations about confidentiality of data between provider and patient.

But the general consensus seems to be that the application of these new tools is a positive step in either line of work. Both the healthcare and education sectors have the resources, brainpower, and motivation to really push the platform far ahead in very rewarding ways. Many believe these to be the most noble of all pursuits when applying Glass in real-world scenarios, with each having ample coverage in the mainstream media and multiple Communities on Google+ actively discussing the pros and cons of Glass and wearable electronics overall. The *Stanford MedicineX* series of Hangouts On Air **prom-**

inently featured [Glass](#) to critique the platform and its potential as a tool. It's a stirring, pragmatic discussion about the future.

If you feel as strongly or just want to discover some of the many ideas being floated, we encourage you to listen in and speak up.

Accessibility

Another area relative to healthcare where the Glass experience can make great strides is for people with limited hearing, use of their limbs, motor skill impairments, or other forms of handicap. Google shared [the inspirational story of Alex Blazczuk](#), a law student who after a car accident doesn't have full use of her hands due to paralysis. She documented going on a camping trip with her friends, negotiating the system controls on Glass by voice commands.

Spoken input and audible output are wonderful ways for people living with disabilities to interact with the system—voice in, voice out. There's also work ongoing to make Glass work for people with autism, where the feedback loop in Glass might not be so natural.

Several companies are also heavily involved working on gesture-based program controls, including eye-tracking. And our friend Mike DiGiovanni, whose work we've rightfully cited before, is [cobbling a GDK app](#) that controls timeline navigation without voice or touch at all, using only head gestures. It's very inspiring work—and very worthwhile.

Home Integration

One of the announcements in 2013 that really shook up Glass devotees was the revelation that Google had submitted and had been approved for patent applications positioning Glass as a hub for home integration, to be used as a remote control for things like controlling settings on your refrigerator and opening your garage door. The patent described Glass employing several communications methods, including RFID, QR codes, Bluetooth, infrared, and the curiously generic “visual identification” over which to wirelessly communicate with connected devices.

This finally gave a viable use case to the promising [Android@Home initiative](#), which admittedly was a space that, after being announced at the Google I/O 2011 developers conference, saw next to no public traction exhibited outside of Google's own Nexus Q project, which coincidentally was scuttled in 2012 mere months after its announcement. This bold revelation really got people starting to think about how Glass could interface and control objects in the real world. If you're a hardware manufacturer looking to add a slick new dimension to your products, this is worth having your R&D team investigate and tinker with.

Revolv is already doing some impressive work with home automation by extending its own platform for Glass, using the HUD as a frontend controller for wireless remote domestic robotics (“domotics”), empowering users with the ability to access their lights, curtain blinds, garage door openers, and locks. Cutting-edge developers at the Nodebot conference were able to control a Parrot AR.Drone quadcopter with Glass using Node.js libraries and Glass gestures—a feat that was repeated by a team led by technologist Dave Martinez at the Breaking Glass hackathon.

And the Glass community cheered loudly when the news broke in early 2014 as Google announced its acquisition of Nest, the company that produces Internet-aware thermostats and smoke detectors. The much-speculated initiative for Google to have relevance with appliances, controllable by smart frontends including Glass, is incredibly cool. Commercials for more than two decades have hinted at consumers being able to monitor and enable/disable security systems and home gadgets with a phone call, touchscreen tap, or voice command. Imagine doing so on Glass in the middle of a jog or while at the store or in the middle of a meeting or while in class—you’d get a timeline card relaying the temperature of the room, and you could speak and/or tap and manipulate the device back home.

You could even set thresholds that would proactively push card updates to you if your living room got too cold or too hot, giving you the ability to raise/lower your IP-controlled device.

Chromecast and Home Entertainment

Not since the inevitable marriage of chocolate and peanut butter have two platforms been so fated to meet and join in union as Glass and Chromecast. Clearly Glass is a ripe platform for casting content through the HDMI dongle either as a possible sender and/or receiver and as a controller or mirroring source. To this end, LynxFit, which we’ve mentioned a few times before, was the first Glassware to incorporate Chromecast, allowing the users to fling their workout guide videos through its GDK app onto a large display. It’s a tremendous cross-platform feature that makes using the app incredibly sticky.

The Mirror API could also possibly be leveraged to generate timeline cards as a relay for a second-screen utility, like “now showing” or “coming up next” screens or something a little more static. Cable providers could let users manage their TV programming in their DVRs and library of media in their home entertainment systems. This might be THE feature that services like Netflix, Hulu, Pandora, and YouTube need to do things like provide Glass wearers access to their queues-at-a-glance, without actually trying to use Glass as a viewing/listening station. Glass could certainly be used to add further value to the leanback experience of enjoying Internet multimedia content on big home displays.

Obviously, we've framed this just in the context of entertainment media, but there are a ton more use cases—and issues—to consider. How could you combine the two?

Android Wear

The escalation in activity for development of applications incorporating Android Wear is also incredible. Glass plays a pivotal role in Google's place in wearable computing, so integrating the Mirror API and the GDK with the Android Wear SDK and sharing notifications across wearable devices is exciting new territory. But just as we laid out in the chapters concentrating on design for Glassware, the litmus test is going to be having wearable devices like Glass, smartwatches, fitness trackers, and others not compete for a user's attention, bandwidth, and notification attention—but to have all of them peacefully coexist.

These platforms shouldn't be seen as rivals, but as partners. So the software shops and developers who master the art of having their applications and services work seamlessly across desktop, web, mobile, and wearables will quickly separate themselves from the pack. We hope to see you break new ground and write about your achievements in a future edition of this book!

Hardware Hacking and the Internet of Things

Just because there's not the inherent ability to do certain things with Glass out of the box doesn't mean someone won't try. It's the magic of having the innovative spirit—you're bound only by your own imagination. Many first-generation ideas like turning Glass into a miner's helmet or a construction worker's scale sound corny...but so did turning an iPhone into a flashlight. And some forward-thinking people have made pretty nice chunks of change from those little endeavors. So because a hacker's work is never finished, let's now shift to moving outside the scope of rooting your device and get into extending Glass to communicate with other platforms entirely.

The community of hackers, builders, and makers using [Arduino](#) and [Raspberry Pi](#) microcontrollers is staggering, and they're doing some really amazing work for very little money. With the rise in hardware startups, the [Accessory Development Kit](#) is a great starting point to create new electronics based on Android. The documentation is worth checking out, as is the [Android Open Accessory protocol](#), on which the ADK is based.

But what if you're not a budding electrical engineer with an entire garage full of spare LEDs, breadboards, soldering irons, and potentiometers? (First, we pity you. You're really missing out—because until you've wired up a circuit that actuates a servo over the Internet based on motion detection, you've not truly lived.) But fear not, there's hope.

Commercial kits let you take advantage of the Internet of Things, the ambitious concept that uses the monstrously expanded address range provided by IPv6 in tandem with smart devices to make them accessible and controllable online. Kits like **Ninja Blocks**, **WeMo sensors**, **FitBit**, **Philips hue lighting rigs**, and **Lego Mindstorms** give builders a prefab-yet-customizable platform on which to build Internet-aware electronics and see what data they're gathering as its gathered. We're really happy to see one of our favorite services, **IFTTT**, trigger actions from APIs from other well-known services in IFTTT's ever-expanding stable of **channels** and connect those events to the Mirror API's event framework. The possibilities are nearly limitless. And this requires developers to really get creative, while sticking to the Think for Glass foundations we've laid out.

A shining example of this type of foresight is the work of Sahas Katta, a young software engineer from California. Over a weekend in June 2013 he essentially rebuilt the Android native app for his relative's Tesla Model S electric car using Glass APIs so he could remotely open/close the sunroof, unlock doors, check the battery's charge, manage climate control settings, and turn on the headlights. He succeeded in making the car hackable. (Mercedes-Benz and Hyundai have since announced plans to integrate Glass into their automobiles as well.)

Even Google cofounder Sergey Brin himself has stated that at some point he'd like to see Glass function as a viewfinder for his DSLR camera, implying really cool cross-device communication. So the true scope for Glass as an extensible platform reaches far outside the box...and prism.

Peripherals/Accessories

This last topic is already seeing some great traction, as tricking out your gear has always been a big thing with techies. With Glass being a modular platform, there's lots of room to make new things to work with it, make it even more personalized, or make it look even more unique. As slick as Glass's design is, some people will want to customize the device and wear it out and about with style and personalization.

GPOP, a design company cofounded by San Francisco-based Glass Explorer David Lee and featuring the designs of artist Virginia Poltrack (creator of the Word of the Day Glassware), wasted no time in putting together snazzy sticker designs giving you a range of alternative frame colors for Glass, using cool visuals like snakeskin patterns, camouflage, a matte carbon fiber texture, iridescent dots, your favorite sports team's logo, crazy stripes, and other ways to make your wearable computing device truly stand out in a crowd.

Custom themes have been a staple of operating systems since Windows 95, and who among us hasn't downloaded a cool skin for our browser or rocked a neat case with our alma mater's logo on our tablet, set a neat ringtone on our phone because we were tired

of everyone using the same polyphonic version of *Canon in D*, or modded the heck out of our Xbox?

The market for Glass peripherals is expected to be huge, giving you yet another avenue to look even more distinct as you wear your computer. Charging platforms, docking stations, holsters, carrying cases, and gear with colors and designs matching your phone and tablet to your Glass headset are expected. We also fully anticipate visual artists to create downloadable templates the community can use for timeline cards to expand the set available in the Google Mirror API Playground.

And of course, there's the merchandise, both officially from Google and knockoff fan products—never discount the importance of good swag.

And the space for accessories for Glass is expected to bloom. No, check that—*explode*. Alternative frames alone should be a big hit. Neoprene or chain lanyards to hang your headset from your neck while not actively using it from companies like Croakies would sell like hotcakes. You can even build upon the ecosystem itself, as the folks over at [Remotte](#) at doing with their handheld remote control device, which uses a variety of sensors, programmable LEDs, and a tactile keypad to control Glass and other Bluetooth-connected smart devices. Neat idea!

In Closing

You now have the tools to do great things with Glass and really push the envelope, so if you're willing and daring, go for it! We'd love to hear, see, and use what you come up with, and we'll be happy to share your ideas in a future edition of this book.

Remember, the first prototype of Google Glass started out as a pair of modified circuit boards taped to either side of an off-the-shelf pair of sunglass frames for counterbalance, with ribbon wires all over the place and a small prism. But with a whole lot of work and thought, the final product didn't turn out too bad (and two guys from opposite sides of the planet even wrote a book about it together).

So we encourage you to use the platform to its fullest. Keep tinkering with your own ideas, keep learning the development frameworks, and commit the design guidelines to memory. There's a lot to be done to make the Glass ecosystem grow and prosper. We can't wait to see what you do with it, and we've both got many more creative ideas that we want to start using with Glass, too.

It's been our pleasure to share our philosophy with you to help you get the most out of the platform. We hope you've enjoyed reading our book as much as we have putting it together. We invite you to interact with us at Glass meetups and events and get in touch with us on Google+. And remember, *always Think for Glass!*

Appendices

A series of helpful appendices with several pro tips are included to give you background information on how to maximize usage of the system, and how users can configure networking, system settings, and manage Glassware registration and installation. We also dive deep into the architecture of several popular Glassware projects and let the teams that produced them tell you in their own words how they put them together and what lessons they learned from doing so.

- *Appendix A, Glassware Done Right: Case Studies from the Field*
- *Appendix B, Hacking Glass*

Glassware Done Right: Case Studies from the Field

We're making sure you get the most bang for your buck, since you were so diligent to read this far. Now that you know how to effectively Think for Glass, it's helpful to see some real-life case studies to see how others are applying the mindset to great Glassware projects, for both the Google Mirror API and the GDK. And to do this, we've enlisted some very reliable friends of ours.

In the short time that Google Glass has been around, we've made countless connections and have logged immeasurable hours in forums, Hangouts, Google+ Communities groups, chat sessions—and, of course, on Glass—discussing with people their passion, ideas, tips, and criticism about the Glass ecosystem. But, more importantly, we've become chummy with several cutting-edge developers who have generously and honestly shared—in their own words—their architecture, triumphs (and horror stories) involved with building great wearable software.

We're in awe of their creativity and we're proud to let them share their insight and expertise with you here. Let their work inspire you and feel free to reach out to them and inquire about their projects. These efforts aren't just great Glassware, they're backed by good people who truly care about our community.

- *Thuuz Sports*: Lets you know what's hot
- *KitchMe*: Glassware that's simply delicious
- *Fancy / ColorSnap*: Leveraging colorful queries
- *LynxFit*: A personal trainer strapped to your head
- *Genie*: The Swiss Army Knife of Glassware
- *Tits & Glass*: Takes Glassware into the bedroom...then pivots to keep it there
- *NameTag*: Launching headfirst into facial recognition

- *CrowdOptic*: Crowdsourced broadcasting
- *Vodo*: Real-time collaboration in Google Drive
- *Preview*: The latest movie trailers are just a glance away
- *GlassFrogger*: Hybrid Glassware using the browser

Thuuz Sports Lets You Know What’s Hot

Winning features: second-screen experience, custom push architecture integration

Long before it debuted as featured Glassware in MyGlass, **Thuuz Sports**, a popular online platform that tracks the pulse of games for several international leagues, had already earned a reputation as a vital component for sports fans on iOS, Android, and Google TV (**Figure A-1**). Thuuz not only keeps track of scores for games in the NFL, NBA, MLB, and NHL, as well as English Premiere League, UEFA, tennis, cricket, and rugby, but the service tracks the velocity of those games by measuring each’s “excitement rating,” an intensity index dictated by several factors, including how close the score is as the game nears its end, fan interest, and other situational signals.

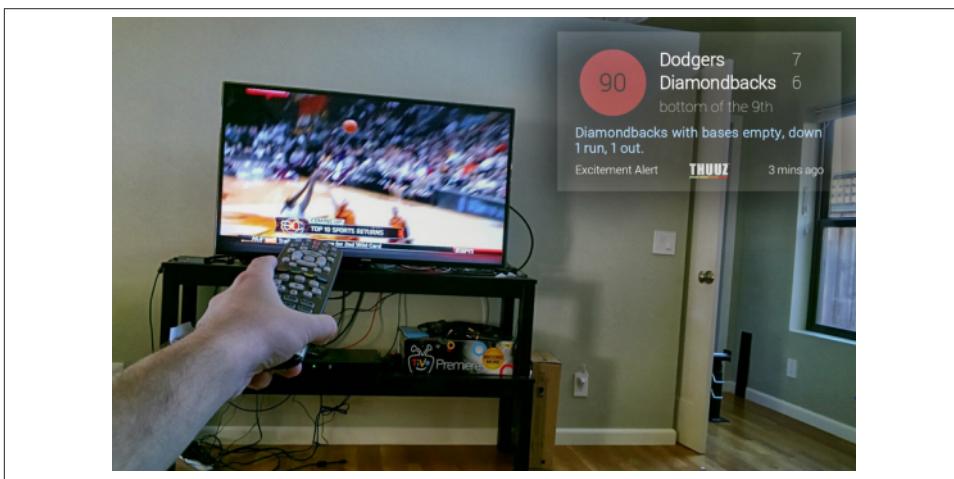


Figure A-1. Thuuz’s Glassware makes the perfect second-screen companion

It makes for a perfect companion to stay abreast of how your favorite teams are doing if you’re unable to watch the game...or if you’re totally hardcore and watching three games at the same time and need alerts from other events as they develop (which is awesome). It’s also an indispensable tool for the fantasy sports crowd, alerting team owners if a player they have is having a monster statistical day. “Thuuz Sports for Glass allows you to get timely updates by alerting you to exciting events as they happen, as

well as reminders for upcoming games you might have forgotten about,” said Jordan Toor, a software engineer on the Thuuz team. “We show you a rating on a scale of 0–100 of how exciting a game in your favorite league is and a teaser of what’s going on.”

The Glassware is written on top of the Mirror API, generating timeline cards with game data and leveraging speakable text so as not to distract users from the actual action while they’re glued to their seats, watching the actual contest (Figure A-2).

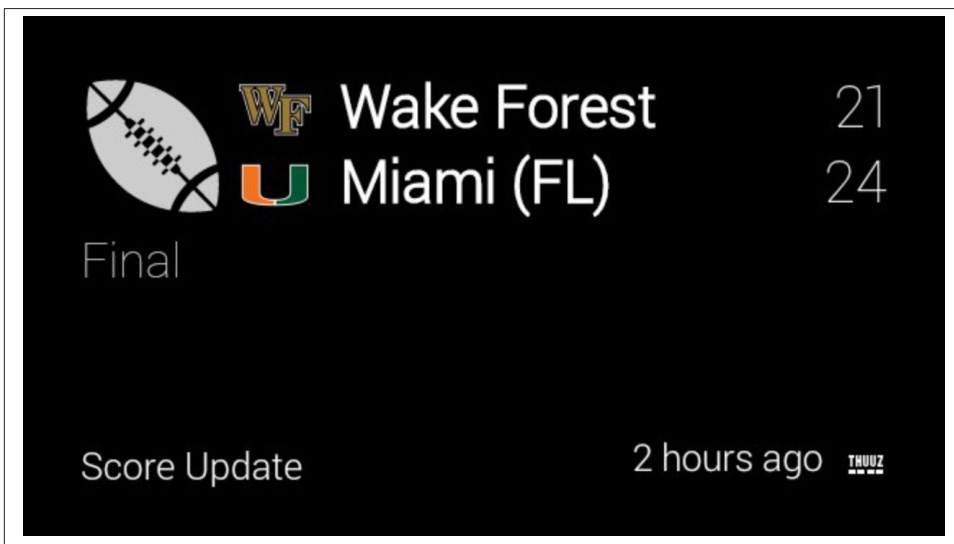


Figure A-2. Web-based configuration is helpful for nearly every sports league

You’re in command of what you track—either specific teams, conferences, or entire leagues via Thuuz’s web-based control panel (Figure A-3).

Thuuz’s Glassware server is written in Python, using Django for database support and template rendering, and talks to Glass through the Google API’s Client Library for Python. Toor explained that integrating the Palo Alto company’s in-place cloud infrastructure for mobile OSes and connected TVs with Glass sync was a breeze. “It was fairly easy to integrate into our push delivery system we wrote in-house,” he noted. “We just added a new device type and then the logic to deliver content via the Mirror API instead of Google Cloud Messaging or Apple Push Notification Service. This is all powered by a workflow system that is distributed between our backend servers.”

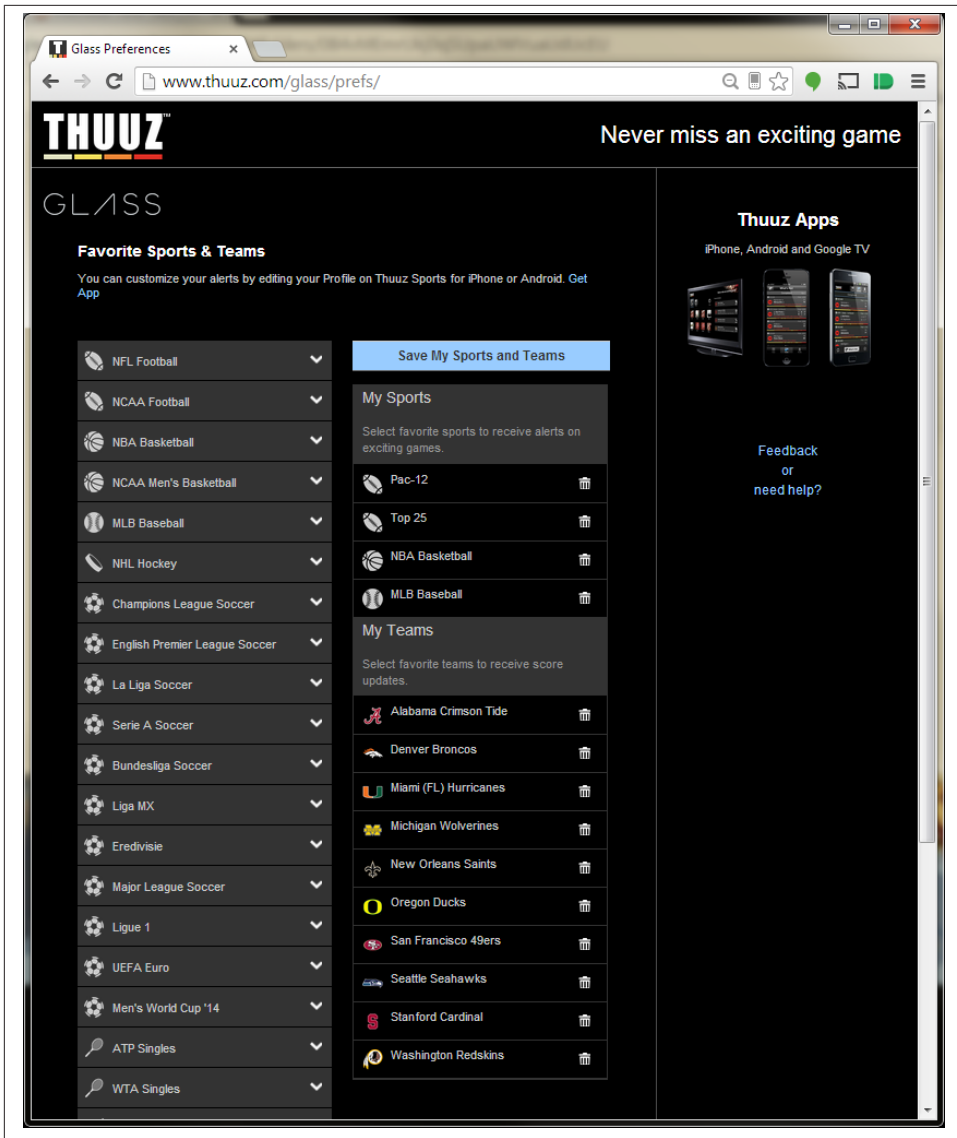


Figure A-3. In-game score updates keep you on top of the action

What became challenging, Toor said, was seamlessly translating Thuuz’s signature UI onto the card paradigm. “Adapting our existing Android and iPhone alerts to look great on Glass, primarily due to differences in formatting,” was a particular hurdle. “And also making sure the speakable text sounds good, as well.”

Even for games you may not be watching or teams you may not be following, Thuuz Sports lets you know if something major is going down. Its Glassware is an extension of the experience that fans have relied on over several seasons, masterfully handling the transition from huge monitors and portable displays to Glass prism—without missing a shot, snap, pitch, or goal.

KitchMe's Glassware Is Simply Delicious

Winning features: web integration, organized bundles

Foodies, rejoice! Whether you're a connoisseur of culinary creations or just someone with the munchies, or if you're an expert chef or more the type that just enjoys the art of cooking, **KitchMe**, by Coupons.com, is Glassware that you can't live without (Figure A-4). The Glassware doubles as a search engine for its sizable listing of recipes, as well as a step-by-step cooking assistant to aide you as you prepare ingredients and meals (Figure A-5). It's a multiplatform extension of KitchMe's browser content with deep social integration that curates recipe listings from all over the Web and aggregates them for easy use in personalized collections you create.

You can also filter recipes by type of diet, course, or style of cuisine, and then send listings to your Glass headset to use while preparing a meal. It's really tight integration with clever use across platforms. Chief software architect Gene Reddick says early tests of voice search capability, which returns bundles of cards for matching recipe results, which can then be shared and pinned for later reference, have proven to be a huge hit and will be available publicly with a future native version of the service. The instructions can then be read to users as they prepare the dishes.

While the service does rely on the trackpad at the moment, as swipes are needed to advance through a card bundle to move from one step to the next, Reddick is hopeful for native voice control down the road, creating truly hands-free cooking. "If this is not provided, we will consider building something ourselves," he speculated. "Either voice control or using the camera to recognize gestures or hand motions."

KitchMe's Glassware server is an ASP.NET MVC application, written in C# and running on IIS, developed with the Google Mirror API. "We also looked at a pure JavaScript solution, but felt that dealing with OAuth was easier handled on the server," recounted Reddick. He said migration of the existing app to Glass wasn't too much of a stretch, noting, "Because most of the KitchMe services were already built and we had working APIs for all the requests we needed for Glass, it didn't require much work to build out the search and recipe services we required. I cropped and scaled all our images to fit the Glass screen resolution."

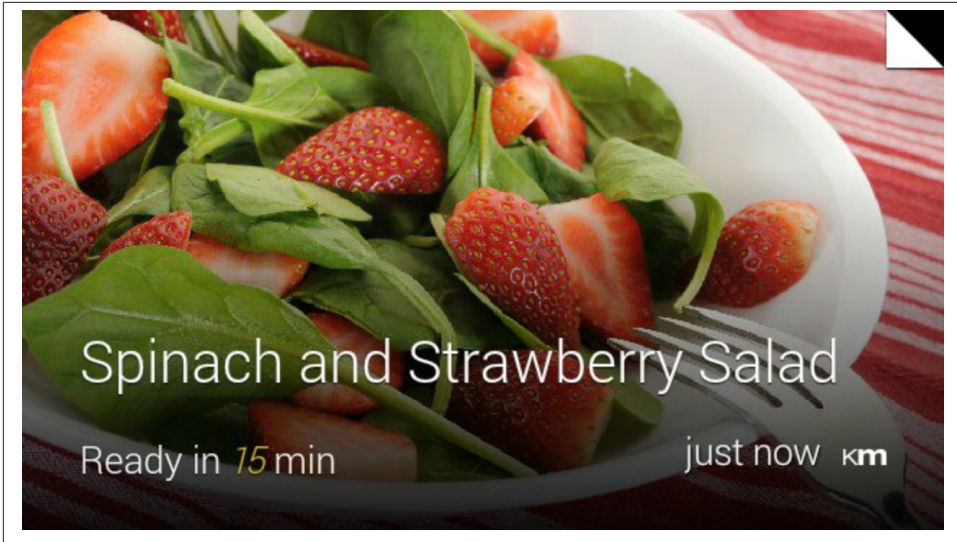


Figure A-4. Recipes are shown beautifully as background images

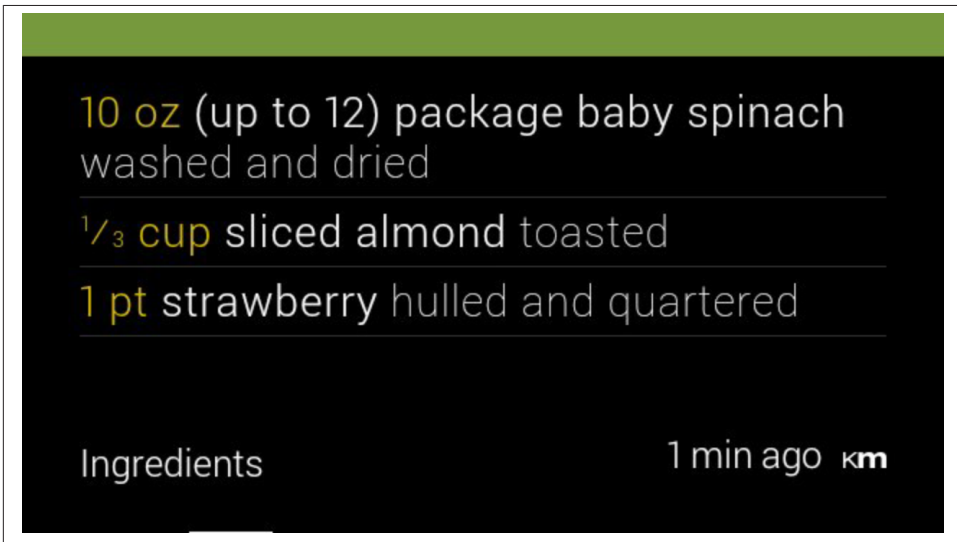


Figure A-5. Recipe steps are represented as bundles

Reddick says the UI was the biggest challenge in bringing KitchMe to life as a wearable computing service. “In getting the UI right, we initially built quite a few more features into the product and displayed a lot more data on each card. With each iteration, we stripped out more information and removed features,” he said.

“In retrospect this seems obvious, but it really took a while to sink in just how far back from our starting point we had to go to get to a clean usable design. Partially this results from the device itself—the screen resolution and the touch-based interface, but more importantly I think from the unique way Glass is used,” Reddick added. “Specifically, interacting with a screen directly in front of your eye feels very different than a screen on your phone held at a greater distance or a computer screen. It feels to me like you have less time to dwell on the screen, less time to notice peripheral detail on a screen that is already at the edge of your peripheral vision.

“Presentation and actions needed to be as simple and clear as we could make them.”

Fancy and ColorSnap Leverage Colorful Queries

Winning features: search-by-color

One of the opportunities that Glass makes possible, like other mobile platforms that preceded it, is the ability to create new ways of achieving proven ideas. Savvy developers can execute complex jobs while maintaining the Glass goal of simplicity and user experiences with minimal use. Two pieces of Glassware that brought to light an emerging space that we find absolutely amazing are **Fancy** and **ColorSnap** by Sherwin-Williams. Both are extensions for Glass of existing platforms for the social shopping service and the paint company, respectively, using color recognition technology.

They introduce the innovative approach of “search-by-color,” wherein Glass wearers can take a picture and share that image with their Glassware. In Fancy’s case, a bundle of cards representing a result set of items matching the colors within the captured photo are returned with the option to purchase for people that like a consistent theme to their clothing, furniture, or household items.

ColorSnap examines the colors of objects within the photo and inserts a single card on the user’s timeline of the same image with a color swatch overlay for a palette of similar shades of paint the company carries (**Figure A-6**).



Figure A-6. ColorSnap matches images with known shades of paint

Fancy even uses a clever trick to let the user know about long-running operations (which are usually on the order of 30 seconds or more)—inserting temporary search cards into a timeline to let the user know the status of things that aren't immediate like product searches based on color. This is an interesting visual technique to denote status for operations that don't return immediately (Figure A-7).

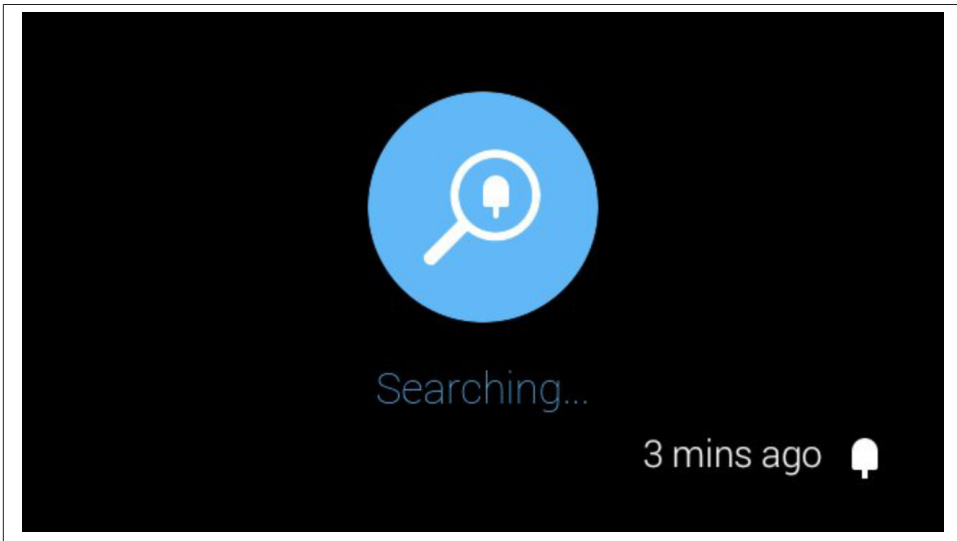


Figure A-7. Fancy's "search in progress" card

Fancy's Glassware inserts temporary cards onto a user's timeline to indicate the status of color searches (Figure A-8).

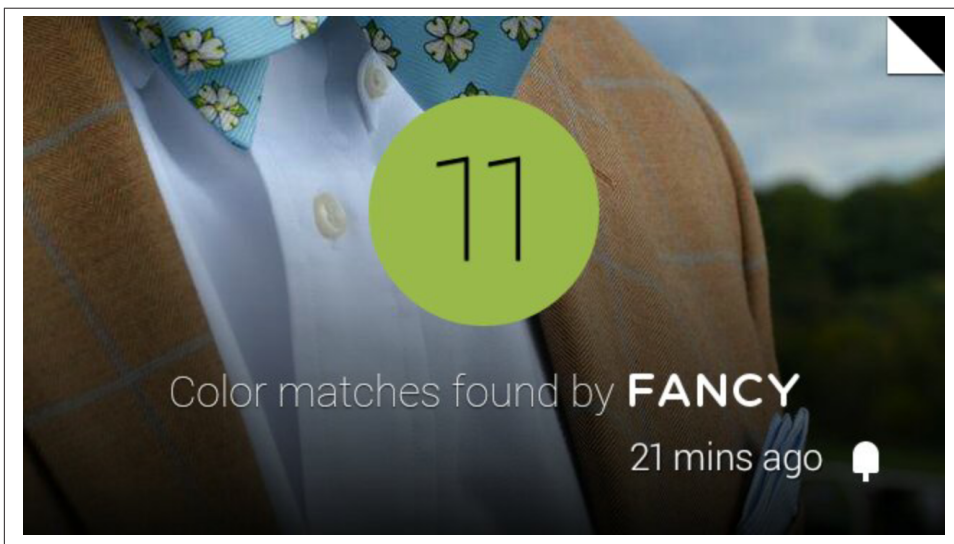


Figure A-8. Matches from Fancy's product database

Both examples are absolutely stellar ways to integrate ecommerce and introduce new ways to think about search. The approach of basing product search on color isn't a snarky retort to the fact that facial recognition technology wasn't allowed early on...it's just a really sharp, quick, and convenient way of interacting with a system and getting results.

Both epitomize what it means to Think for Glass and are endlessly fun to play with.

LynxFit: A Personal Trainer Strapped to Your Head

Winning features: sensor access, Chromecast integration

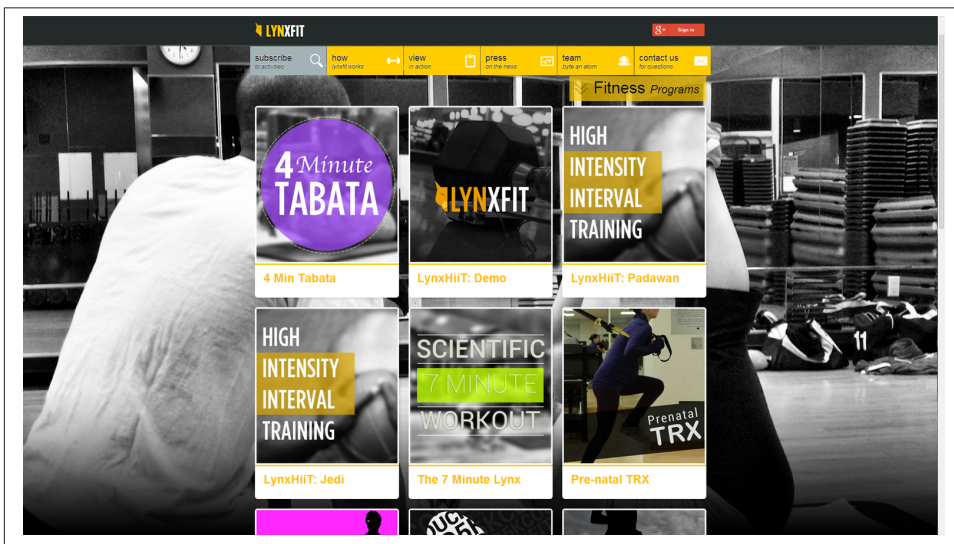


Figure A-9. *LynxFit* is a one-stop wearable workout solution

Your product has gone through several iterations to get where it's currently at. Share how the feature set has changed as the Glass ecosystem has expanded with new frameworks, hardware, gradual rollout to a larger audience, etc.

The Glass SDK and API, combined with the Google Glass cloud platform made exploring various features easy to prototype. The ease of development made rolling out functionality based on feedback very nimble and seamless.

How would you position LynxFit in the grander scope of fitness apps?

LynxFit is a better way to work out, using wearable computing to bring a more fun, immersive experience that motivates users on their path to health and fitness. Working out hands-free is a new level of independence for the user, bringing their fitness fix wherever they are, whenever they need it, without the need of checking a cellphone or tablet, which completely disrupts the flow.

LynxFit was among the first that called for programmatic access to the Glass sensors. Describe the genesis of your concept to have workout software for wearables, and how this matured when you first tried to implement it as Glassware with the Mirror API.

The first implementation of what was then GlassFit, was pushing content to Glass, and even when it had some animated GIFs (in lieu of actually animating graphics in the display) which were pioneering at the time, it was not as interactive as we'd like it to be. The natural evolution for GlassFit was LynxFit, an experience that is voice coached and immersive counting and cheering for the user as they go through their daily fix of fitness, as we like to call it.

What were some of the challenges you experienced translating your idea from essentially a RESTful service to one that was a fully native experience (Figure A-10)? Have you in any way merged the two frameworks for hybrid functionality?

In a way, with LynxFit we found out that we were pushing the limits of Glass and the GDK as we were stressing the battery and processor through extreme use of video, sensors, and processing, which lead to overheat among other issues. We had to carefully craft the app around those limitations playing with the screen wakelife and pulling some other tricks to make it viable.

We are using a hybrid between GDK and Mirror API as there is a server-side process that at the scheduled time sends a card to the timeline to remind the user of their workout time. From these reminder cards the GDK app can be launched. Also, at the end of each workout we send statistics and PIE (Pace, Intensity, and Endurance indexes) through Mirror API as well.

The web admin interface is extremely slick, but the app still features lots of input controls. How might other Glassware developers use this model to provide user administration for their own projects, rather than directly on the device?

We think of it as driving any complexity into the website (and in the future to the mobile app) versus Google Glass. Glass is designed for microinteractions and even when parts of the LynxFit experience are immersive, we try to go with simpler first. The website on the other hand has all the full-blown possibilities for the user to choose their content and schedule it to show up on Glass.

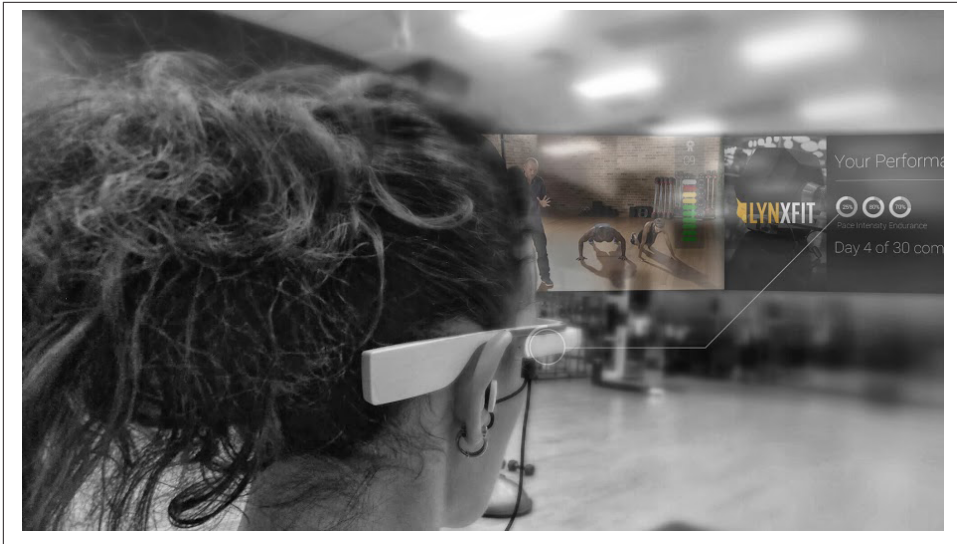


Figure A-10. Set your exercise schedule and progress

You also broke new ground for being the first Glassware to integrate with Chromecast. Talk about the idea behind this and how the architecture and APIs supported it.

The idea is to use any resource available to get the best possible user experience to the user. On that thought trend, Chromecast became a natural extension of the Glass experience when the user has a TV at reach allowing for gamified shared quantified workouts that were before hard to capture. Because we are having requests to implement the Chromecast experience on its own from mobile devices (with the possibility of adding external sensors, Plantronics, Jarvis, Glass) we are reviewing the architecture in order to give mobile a more central role and sharing most of the core code at the same time.

What have some of the challenges been in developing software that's whose very existence is contingent on tracking motion?

There's also a clever gamification angle to the workouts, which makes using the software fun. You've picked up on the hint that this might also have bigger implications for doing extended workouts.

What's on the horizon for LynxFit, as far as applying new ideas, adding new features, maybe premium membership services, integrating other platforms, or even adding new products to the Byte an Atom line?

The play with Byte an Atom Research is to build innovative software (bytes) and kickass hardware (atoms). Right now the primary focus is to enable healthy lifestyles through wearable computers so don't be surprised to see a Lynx-related hardware device on Kickstarter to help that dream along someday soon.

Genie: The Swiss Army Knife of Glassware

Winning features: beautiful UI, multiple uses, location data

Genie is heralded as one of the most well-designed Glassware services available. The UI is simply beautiful—simple and effective. It really works for the medium. Share your approach for good design strategies when working with wearables.

We at 33Labs have always had a strong focus on usability and user experience. Mauro Canzian, our cofounder, has been influenced by his past studies on architecture and has a special talent to bring together graphic design and UX design to get the best experience possible for the user. Mauro actually designed for Google Glass from South America, months before he ever tried it.

On our Google Glass approach the main things we focus on are:

- Following Google's guidelines and standards.
- Keeping it simple, clean, minimalistic in a way, showing only what the user really needs to see on each microinteraction.
- Using a visual color-coded reference for icons, bars, etc. to identify each key section.
- UX using as much as we can on each platform to enrich the experience. In the new GDK development for Genie, we are experimenting with the use of text-to-speech, speech-to-text, two-finger swipes, two-fingers taps, etc.

Early on, Genie made waves by being THE one-stop shopping Glassware service, essentially being the first software suite for Glass. What are you most proud of as far as Genie's feature set, and what aspect of the initial program was the most challenging to build?

Coming out of the Glass Foundry in February 2013, 33Labs spent numerous brainstorming sessions to decide what to build. Such a new platform, so many options. One of the things that we realized early on is that accessing apps would be complicated on Glass as the number of apps grew. That thought pointed us in the direction to make one app that could be highly usable on a daily basis. Together with a longstanding obsession to have computers, and better yet wearable computers, help us expand and augment our faulty human memory the project was born.

You use Mirror API location data by querying for the user's position to mark their parking spot for later reference. How did you enjoy working with the location scope?

It was early on and we found out that some features were not working as we expected. Our initial idea was for the user to tap and instantaneously get location information, but we were not getting the geolocation as a custom menu option was being pressed. We had to trick the Mirror API by using voice recognition in order to get a location out of the system. Now, with all the new possibilities brought up with the GDK this is changing.



Figure A-11. Genie has lots of cards for lots of situations

Genie also applies an interesting spin on the traditional concept of identity, using a calling card metaphor within the service to store public contact information about the user. Talk about how this plays into its broader applications.

This was an idea from our friend David Lorenzini. Given that facial recognition was out of the picture early on, why not use simple voice passphrases to identify a user and be able to exchange their public information, a problem most users want resolved yesterday? It was a lot of fun to experiment with this concept. In the end there were not enough Glass users on the streets for this to work and now it's a feature we're walking off from at this time.

One of the initial challenges was making sure that Glassware had some sort of persistent presence within the timeline. Without using custom voice commands or menu items, Genie achieved this by pinning itself. However, Explorers discovered that when Glass would do housekeeping and purge old cards, it did so for items left of the home screen, too, which would then force users to have to resubscribe to the Glassware to get it back. How did you resolve this problem from a UI standpoint?

That was an interesting surprise to see that the pinned cards were being recycled. At that point we realized we were pushing the limits and using pinned cards in a way they weren't intended (isn't that what developers and users just do?). As a funny anecdote, my friend and Explorer Noble Ackerson told me recently that he was intentionally unpinning and pinning the Genie main menu weekly to refresh it and prevent it from vanishing. In the end the solution came with the "Take a note" voice command addition to the "OK, Glass" main menu, which we are using now.

Genie was created using the Mirror API—any plans to integrate native functionality, either as a rewrite or as a hybrid using the GDK with functionality between the two frameworks?

At this point we're moving into developing with the GDK as the main platform for Genie and using Mirror API in a hybrid style to allow navigation and sharing lists from the timeline.

The main reason we decided to move to the GDK is access to gestures to allow users to handle lists in an intuitive and faster way. Also, we get better response times when notes need to be remembered or recalled. We still love Mirror API for broadcasting and other features, but we feel its just not agile enough for the main purpose of the application.

In the short term we expect to be able to satisfy one of our users' main requests, which is to be able to take notes using offline voice recognition. This should be resolved in the level of Glass Android OS mainly as it upgrades to fresher versions of Android, but also it involves syncing offline/online data, which will be a bit challenging.

MiKandi Takes Glassware into the Bedroom... Then Pivots to Keep it There

Winning features: scalability

Jennifer McEwen, cofounder of Seattle-based MiKandi, a leading innovation shop in the adult industry and producer of the largest third-party adult app market for Android, bore the burden in June 2013 of having to be the first Glassware publisher to have her creation banned—mere hours after it went live. **Tits & Glass**, which lets users browse and vote up sexy imagery, was deemed inappropriate after Google modified its policy governing what type of content would be allowed for its still-in-beta developer platform.

Rather than sit and pout—or worse, just give up—McEwen wasted no time in modifying the service by toning things down considerably and featuring tamer content that didn't violate the clause prohibiting sexually explicit material. Similarly, she gave members the ability to discretely share their own first-person perspective pictures using Glass, which are then viewed on the Web. And like many of its contemporaries, MiKandi has also had to deal with the challenge of managing software with overwhelming demand against a rate-limited API.

As a result, McEwen and her three-person team have had to be **extremely creative and forward-thinking** to ensure MiKandi maintains its presence throughout Google's ecosystem (Figure A-12). Here, McEwen recalls her first-hand experience of putting the system together and keeping it running amid some interesting challenges with user demand and Google terms.

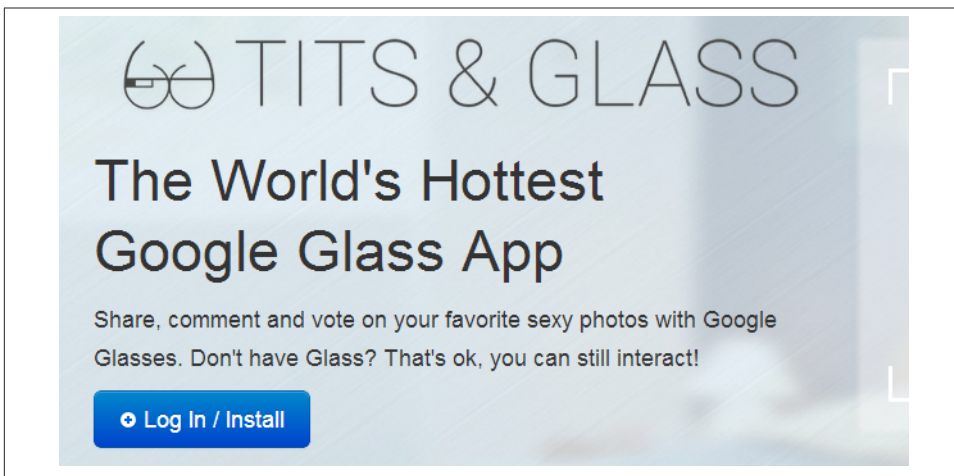


Figure A-12. MiKandi has handled pretty impressive scalability issues gracefully

The MiKandi Experience

First and foremost, Glass is a communications device, so naturally intimate conversations will happen through it. Tits & Glass provides an outlet for adults to share their intimate POV photos with like-minded individuals in a safe, fun, adults-only environment. Originally, the Glassware delivered 18+ content to users but after Google updated their policies to prohibit adult material, we revised it to deliver only SFW content to the device. Users can still upload their adult Glass photos, but they'll only be viewable on the website for now.

Installation/Configuration

Tits & Glass is very simple to install and use—simply link the app with your Google account and we start sending updates, including an introductory greeting with several of our top cards. At the moment, while Mirror API requests are so scarce, we have considered having a welcome card, which requires confirmation to weed out those users who are linking their Google account even though they don't have Glass. We essentially hit our API limits every morning before the majority of our users can use the app.

Home-Rolled Development

As very early Mirror API developers, we didn't have many options available with the Mirror framework, so we had to roll our own. We used our in-house PHP framework, Propellant (similar to CodeIgniter, an open source framework), and added Mirror API modules to our standard library. We backed the Tits & Glass app using Redis. We may well consider open-sourcing a PSR-0 version of our module.

Facilitating Requests: Challenges in Building

One of the biggest challenges is the fact that sharing notifications and other actions do not clear the newly created action card from the timeline. This requires additional API calls to fetch data from the share and clear the extra card.

In addition, the Mirror API effectively requires that developers maintain their own version of app state (which cards are shown, which are hidden, etc). Our first implementation relied on Mirror to keep state, and polled it when updates were required. API limits and expressed best policies on the Mirror API docs pushed us to keep a complete local copy of state of each user's Glass. For the relatively small number of Glass users at the moment, keeping this extra state is a simple problem (and Redis' sorted sets worked really, really well to manage this in a simple and efficient manner). As the number of Glass users grows, however, this will present a nontrivial database/datastore problem to Mirror developers.

On another note, the Mirror API Playground as a card previewer is a long way from being suitable for designers to build and thoroughly test layouts and designs. We ended up building a small card previewer that our designers could use to preview card designs using Chrome but pulling from our live database data.

More broadly, we hope a solid toolkit of wrappers and helpers springs up for the Mirror API. For our experienced devs, working with the API at a wire level was difficult, but doable. I think that many people who just want to tinker will be put off by the difficulty of interacting with the service.

Scalability Issues

The limited number of API requests made Tits & Glass hard to develop. On the upside, it made us build a really frugal app. By keeping local state, we managed to avoid a number of API calls. Image size is obviously a big issue, which is why we compressed the Glass version of images using SendFaster's CRUNCH compression software.

Due to the popularity of Glass, an app can get a huge amount of traffic in a short period of time—a cheap shared hosting account will almost certainly fall over under the load that even a modest Glass app will generate with some modest press. We ended up hosting our Glassware in a load balanced auto-scaling scenario.

Then there's the issue of Google's policy changes. When Glass was announced at Google I/O in 2012, the running joke was that it would be perfect for POV adult content. News broke that we were developing an adult Glass app two weeks before we publicly released it. Google quietly updated their terms over the weekend without giving developers notification, then punished us for violating what they considered "clear" policies. These kind of quiet, last-minute policy changes cause the same frustration as the arbitrary policies changes of Apple's App Store.

Google's trend toward closing their beta technologies and their recent closed policy toward the delivery of Chrome Apps has raised some alarms. By all accounts, Chrome Apps should be more open than Android, but we see the opposite. It's too early to tell whether or not these are signs of an overall shift in attitude from Google, but we're watching them very closely and will continue to push for openness and innovation.

Platform Expansion: Plans on Having an Installed App?

I hope to see Glassware that can facilitate two-way interaction between adults—couples, cam models and fans, or total strangers. In regards to our Glassware in particular, our hands are somewhat tied behind our backs until Google agrees to increase our API calls.

With respect to possibly extending our reach using native code on Glass, we expect that Google will provide a terms of use in the GDK compiler that disallows compilation against the codebase without their permission, as they did with the Chromecast SDK.

Going Forward, Pushing the Envelope

Wearable devices are a brave new world. There's definitely going to be a learning curve for the community. To compare to literature, in *I, Robot* Asimov talked about "early days of robots" where robots were required to have human riders, because people were so scared of the new technology that they hobbled it to feel safe. In the same way, the prohibition on apps like Winky and features like facial recognition cannot and will not stand in the long run.

Such natural and obvious uses of the technology may be delayed for a time, but will not be denied.

NameTag Launches Headfirst into Facial Recognition

Winning features: real time facial recognition

Six months after Google announced they'd be shipping Glass without any sort of facial recognition ability, FacialRecognition.com, which promotes itself as having some of the most accurate facial recognition software in the world, announced a beta of **NameTag**, its native Glass application that lets wearers scan and identify people with the Glass camera by comparing them to known public systems in real time (Figure A-13). It seeks

to provide a deeper way of connecting people in the real world by liberating social media from traditional desktop and handheld devices.

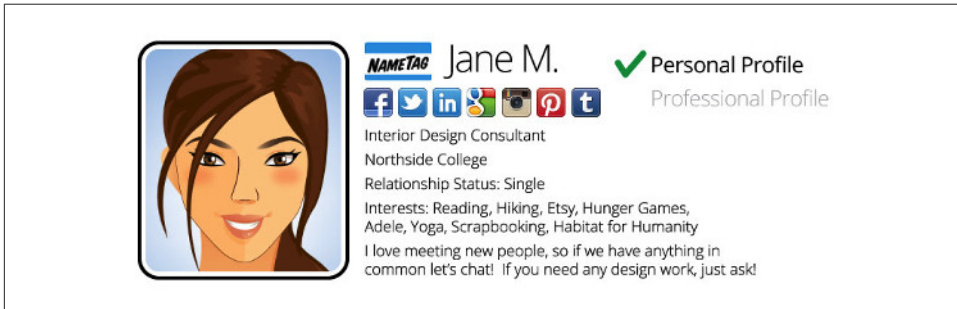


Figure A-13. NameTag compares scanned faces to known, public data sources

While not sanctioned by Google, the app does some very remarkable things and hints at what might be a big market for Glass.

What are the core use cases NameTag was built to address?

Since launching the beta program we have had requests from doctors with lots of patients, professors with lots of students, and sufferers of face blindness, to name just a few. Anywhere it is important to remember a correct name in a timely fashion we can provide a useful service. There has been particular interest from the dating industry to provide a safety background check against a database of serious sexual offenders to warn you of potential dangers. Ultimately we would like to see a situation where you can safely meet people based on mutual interests, so if I capture your face I will instantly see things we are both interested in and this enables a much more effortless conversation.

With respect to usability on Google Glass, some of the initial natural limitations with the device are framing a shot, holding the camera still, and shooting a subject at the right angle to legibly scan them. How might NameTag improve upon this?

NameTag is built with some advanced capture features to frame and crop a face automatically when the user positions the viewfinder reticule over someone's face. The most important thing is to create an intuitive interface that engages the user. Over time the hardware and camera devices in smart glasses will improve and this will allow us to do even more.

What signals does the system look for when not only running tests on 2D digital imagery or printed photos, but actual people in the real world?

People are often moving, standing in poor light, or wearing something on their face, which can frustrate the recognition algorithm. The NameTag app has been extremely

successful at overcoming these challenges. We work hard to optimize for both 2D and 3D recognition situations.

NameTag is a fully immersive experience that doesn't implement the default Glass UX like the timeline or cards. Describe building it as an Android app with the GDK libraries and constructing its user interface for Glass.

It does need some forethought to consider the smaller size of a Google Glass screen. We actually built an initial prototype using a web stack and constrained the proportions to the Google Glass dimensions so we could see how components in the UI would fit together.

Detail your approach for the complex task of not only algorithmically recognizing a user's face, but performing object detection and being able to discern what a face is from digital imagery being received by a camera in real time.

The high-level process happens in a specific order: analyze the video stream for face objects, when a face is detected frame and crop that face, convert the face to biometric data, run a comparison search for similar faces in our database, and finally return data for top matches in our database. We continue to optimize our techniques to achieve better results for the user.

NameTag is a perfect example of network effects—the more systems the app hooks into, the smarter and more valuable it becomes. In addition to the sources it currently taps, what other systems do you see as key to making the service applicable to a wider range of situations, therefore increasing the accuracy of scans?

NameTag operates as a search engine for people. We spider the web accessing publicly accessible data, then compiling it into useful profiles. Our servers will continue to link to new public records to create the best possible results for our users.

From a big data standpoint, describe the process of indexing information from all the disparate systems you use, and keeping such in the cloud.

Future developers need to consider classic spidering and data collection techniques, partnered with Hadoop-style analysis, and running on one of the major cloud infrastructure systems.

What were some of the challenges in building for Glass? Any difficulties working with the Explorer Edition hardware?

We ran into challenges throughout the development of this app because of the limitations of the Explorer Edition hardware. We expect that Google Glass will continue to evolve with better hardware even before being released as a consumer product. This evolution will allow development teams like ours to continue to push the boundaries of what is possible. In the meantime we will continue to make our software more efficient to address issues like overheating.

It's important to note that NameTag is just one application of FacialNetwork.com's innovative approach to working with detection/identification. How might some of your other ideas establish a presence on Glass?

Smart wearable technology will continue to evolve allowing more efficient use of facial recognition. Everything from looking up a business contact at a tradeshow, to a doctor who wants to confirm the identity of the patient they are diagnosing in relation to the chart, or even a university professor who wants to access reports for the student standing in front of them. Many of our beta testers are interested in and are even developing useful applications; we would love to support these apps with the power of our platform and create the maximum value out of this revolutionary technology.

CrowdOptic: Crowdsourced Broadcasting

Winning features: crowdsourced broadcasting model

One of the best things about Glass development is that in a few amazing cases, it's not just an application that arises. Such is the case of **CrowdOptic**, which is using the wearable computing revolution to transform broadcasting, effectively building its own platform to do so. Using an innovative approach, the software is able to detect people using devices like Glass and allow them to share their video feeds with each other, or with other broadcast equipment for an incredible experience and powerful analytical tool.

CrowdOptic works bidirectionally—giving users the ability to ingest live content from other wearers, and also for wearers to distribute their video dramatically onto mammoth digital displays, to television, and to the Web. The merged integrations are controlled via a browser-based dashboard. It's an amazing system that perfectly demonstrates how Glass can be used as a client within other platforms.

Describe your concept of crowdsourcing video content in real time based on proximity and how other Glass users can partake of each other's feeds.

CrowdOptic's patented focal clustering technology (*U.S. Patent No. 8,527,340*) senses where multiple computing devices, including Google Glass and smartphones, are aimed in real time and enables instant, live applications based on where people are looking. CrowdOptic's Broadcast-In feature leverages this technology to allow fans in luxury suites and select seating to aim their Google Glass at any broadcasting device, or in the same direction as any broadcasting device, including another pair of Google Glass, to inherit the video feed from that device.

How difficult or liberating was applying the particularities of Google Glass to your existing software? What aspect of the hardware were you able to exploit to make your platform shine?

Using Google Glass—a computer that sits in your line of sight—with CrowdOptic's focal clustering technology is such a natural combination that developing for the Glass plat-

form was a no-brainer from our perspective. Initially, there were challenges, partially due to the challenges Google itself has faced with balancing form factor against things such as processing power, battery life, heat, etc. These considerations placed limits on the device's capabilities from a hardware perspective and made it more challenging for us initially to do cutting-edge things using Glass. We had to employ various work-arounds, including rooting the device, in order to access the sensor data needed to enable our clustering technology.

Later, updates to the Glass OS allowed for easier access to these sensors.

Detail your backend architecture and how you were able to integrate Glass into it.

The CrowdOptic Glass Broadcast Platform networks multiple Glass units to identify the best views out of all possible video feeds. Our platform requires a CrowdOptic Video Proxy Server (VPS), local WiFi network, Internet connectivity to access the CrowdOptic Cluster Detection Server (CDS), and n units of Google Glass running CrowdOptic software (Figure A-14).

The Glass units (any smart devices) broadcast video streams to the VPS, which connects to the CDS to analyze the quality and perspective of each stream. The best video streams are instantly made available for broadcast-in and broadcast-out integrations that can be managed through a web-based dashboard.

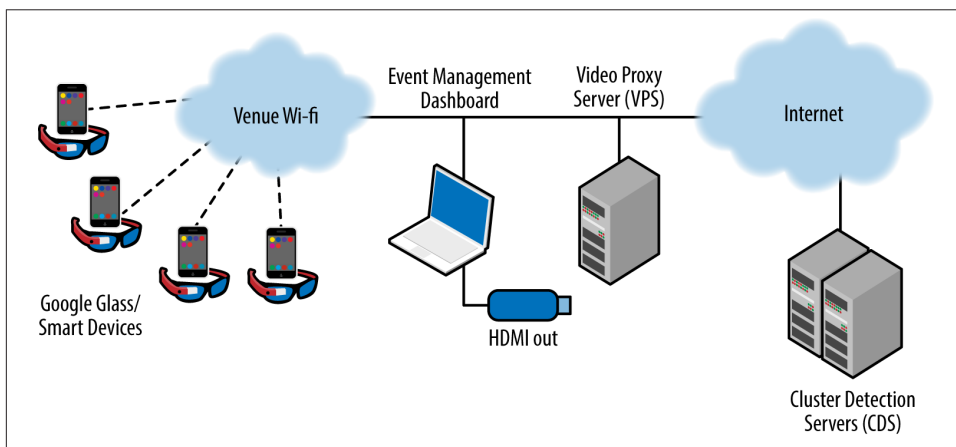


Figure A-14. You can't not be impressed by CrowdOptic's system architecture

You've made big waves with athletic communities. What have the major real-world use cases been so far?

Sports, media, and entertainment are a natural fit for our technology, and adoption of the CrowdOptic platform has rapidly taken off in this vertical. The first professional sports deployment was by the Sacramento Kings, who used our platform to obtain live

crowdsourced video content from Glass units and broadcast the footage in real time to their Jumbotron. The Kings were able to offer a one-of-a-kind fan experience using CrowdOptic as a tool to crowdsource live video content directly from Glass, because our technology allowed them to select which video streams to broadcast in real time from among multiple Glass units recording simultaneously.

How do you see CrowdOptic also being used throughout other industries?

CrowdOptic shines in any environment where there is desire for spectators to see something being filmed and broadcast digitally from multiple angles or points of view. The applications are vast. For example, we have been deployed across industries from consumer packaged goods (L’Oreal) to security (Metlife Stadium). In that latter example, the end users were security agents, who were able to instantly view live video footage of the locations of possible incidents. Importantly, CrowdOptic provided both the location of the camera/witness and the location of the video subject (incident location).

Your software not only has great impacts on personal content sharing, but for professional network-level broadcasting. How could your technology be implemented with TV networks?

We are very involved with TV networks and are working with them to enable electronic news gathering and real-time Glass reporting applications. It is also worth mentioning that any video content that is generated at a televised event can be broadcast to television via the CrowdOptic platform dashboard just as easily as it can be sent to a Jumbotron or other in-stadium digital media or to the Web. This is considered phase two by many of our customers.

Some major deployments are in the works. Stay tuned for announcements.

You’ve done a lot of custom development to achieve the experience you’re after. What about the Mirror API and the GDK did you enjoy using, and where did you find them to come up short?

We have begun using the GDK. The Mirror API isn’t as useful when you’re actively running an application like ours. It mainly allows applications to interact with the user when the application isn’t actively running. In the future, I’m sure we’ll use the Mirror API to alert the user when they’ve wandered into an area where our app could be used. The full GDK hasn’t been released yet so all that we’ve used the GDK so far for is to add a voice launcher for our application. That was a big deal because launching the application before that required some third-party stuff to be installed on Glass.

We’re eagerly awaiting the full GDK release to see what functionality it will offer that we can leverage to enhance the capabilities of our system.

Vodo: Real-Time Collaboration for the Enterprise

Winning features: Google Drive integration, real time collaboration

One of the main ways that Glass could be used is as a tool to help you work better. **Vodo**, which is Allen's first Glassware service, was created for that very reason. His is also the first sanctioned Glassware to incorporate features from the Google Drive API, letting workers collaborate on cloud documents in real time (Figure A-15).



Figure A-15. Vodo ties the collaborative nature of files in Google Drive with the immediacy of Glass

Describe the application hosting environment on which Vodo was built and its components. How long did it take to code?

The latest version of Vodo runs using Node.js on a Unix platform. We use Google's library to communicate with both the Mirror API and the Google Drive API and Mongoose to talk to a Mongo DB data store. The core function for it was written in about two days, and based on work done at the Glass Foundry hackathon, which gave us about 24 hours to develop.

Why was Node.js chosen as Vodo's platform? Any advantages to using it over other server-side frameworks?

Node.js has a reputation of very fast threadless operation, which seemed like it would scale quite well at high load levels. This seemed like an important requirement for Glassware as we got started, and has proven to be very useful in the long-term. Some of the design constraints of Node.js and Mongo DB prompted us to think about good approaches for general Glassware design in any language.

What were most difficult parts of making Google Apps Script talk to Glass?

When the project started at the hackathon, there was no Glass support for Google Apps Script and there were some issues with processing HTTP POST bodies. Since then, both of these issues have been dealt with by the Apps Script team.

The service has always been Mirror API Glassware. At any point did you consider reengineering it as a GDK app, or are there plans to merge its features as a hybrid service between the two frameworks?

It was considered a couple of times, but there are no real advantages to doing so right now. The Mirror API manages all of the network issues that would otherwise occupy much of the code and probably cause most of the problems a user might experience.

Leveraging all the resources that Mirror gives enables us to focus on presenting data in the form most suited for people to use with Glass and not having to worry about features that should “just work.”

How important was it to keep all admin features web-based and not within the Glassware itself?

Using the Web has allowed our team to make the configuration flexible and let users tailor Vodo to their specific needs. Right now, the configuration is fairly simple, mostly allowing users to pick what Google Drive folder they want to work with. But even that would be overwhelming to work with on Glass. It lets us present a very simple Glass interface that is perfectly suited to the concepts of creating or receiving content, while leaving the more complex options to be made through the Web.

What have been the use cases that you’ve seen that have impressed you most?

What has been most impressive are how many different ways people use Vodo as part of their regular workflow. Many people use it to edit task lists on the desktop, where it is easier to do so, but view the contents from Glass. We have a person who produces videos use it to get their recordings off Glass and quickly into their production system.

Most exciting, however, is the building inspector who uses Glass with Vodo to capture pictures and notes about his site surveys; when he returns back to the office, all his field notes and pictures are saved in one folder and he can quickly cut and paste them into his formal report using the documents format in Google Drive. Cases like these prove that Glass is ready for business today, and will only improve as we identify other file types that people use and the best way to represent them on Glass.

Preview: The Latest Movie Trailers Are Just a Glance Away

Winning features: image recognition, real-time video stream processing

Glass goes Hollywood! One of the major areas that Glass and Glassware are going to make a major impact is in media/entertainment. Software engineer Takahiro Horikawa came up with the notion of letting a Glass wearer gaze at a movie poster, recognizing what film the marquee was promoting, and playing back its trailer video. The result is **Preview**, a GDK application that uses real-time processing of the camera’s inbound video feed and cloud-based image recognition algorithms to trigger playback of the official promotional clip (**Figure A-16**). It’s so obvious, logical, and simple, and the implementation is brilliant.

And it’s endlessly fun to use when you’re at the moviehouse, at your desktop, or out on the town and want to see what all the buzz is about.



Figure A-16. Intelligent object recognition is at the heart of Preview

Preview's typical use case is fairly well defined, but talk about the various ways users have told you they're putting the app into action in live settings.

The use case that I and most people are thinking is going to the theater and looking at posters to see the movie trailers to decide which movie to see. While some people have already used it at the real theater, a lot of users tried the app in front of their PC or mobile device and just enjoy how our image recognition works so well. I heard some feature requests: some users said it would be great if Preview notifies them about popular movies in a nearby theater. It will be possibly supported in the future, but I want to focus on image recognition and visual input first, which I believe will be very common sometime soon.

Another interesting feature request is the ability to screencast a trailer or send a link to it from Glass to a nearby smartphone. That way instead of just one person with Glass figuring out which movie to watch, a group could do the same. I think this is a good idea and will consider implementing it...though I still need to figure out the feasibility of it.

Describe Preview's data store hierarchy—where's the content coming from and what's the size of the corpus that you're searching against?

Our backend collects movie data from IMDb, Rotten Tomatoes, and TasteKid using web APIs and integrates them. Preview is currently focusing on the movies that are playing in the theater right now, so the number of movies is several hundred. But the database will grow to accommodate new movies as they come because the database exists on the server.

How does data flow when a Glass wearer begins looking at a movie poster? What's the distribution of cloud versus local computation?

Preview recognizes the movie poster by comparing it to images in the cloud. Preview starts the camera, takes a photo, and sends it to our server to recognize it. This process happens several times until it succeeds. There are some reasons for doing it in the cloud—first, with cloud recognition, we can save a lot of CPU/memory/disk resources on Glass. Second, we can accommodate new movies as they come.

For playing a trailer, our system simply returns the YouTube Video ID associated with the movie's title. After receiving that ID, Preview streams the clip with the built-in Glass media player.

The processing model—analyzing the camera's inbound video stream—pretty much mandates that it needs to be done as a native application. In the interest of having the app consume less battery, do you have any desire to try to implement this as a Mirror API service?

We are just sending a still image for image recognition, so technically it can be replaced with Mirror API. However I don't want to implement Preview as a Mirror API service because of our UX design. Preview is designed for users to access movie trailers easily, quickly, and completely hands-free. If we employ the Mirror API, it may require some touch gestures, which we don't want to do.

Preview also does not consume the battery very much, because Preview is designed for short lifetime—it exits after playing a trailer, so the lifetime is at most two minutes or so; and all recognition is running server side, not on Glass. Hence there would be no strong battery life benefit by using Mirror.

What's the secret sauce to how Preview actually recognizes an object as a movie poster, and then compares it against your database?

Our backend preprocesses all movie poster images in advance, stores “features” for every image, and runs through them against the camera picture at runtime. The feature descriptor that I am using is robust in terms of allowing slight image rotation, scale changes, brightness changes, and blurring, which makes it possible to recognize a movie poster in any situation.

What are some of the challenges of detecting digital images on a computer screen or mobile device as opposed to a real-life poster in a marquee?

There is not a very big difference between them, but if any, the size of movie poster is likely to be different—when people are looking at the movie poster digitally, its size is likely to be small, but in real life, it is probably bigger (and sometimes cropped). Small images are not good for recognition accuracy, so we still need to come up with a way to educate users to zoom into a movie poster when they look at it.

In testing Preview, we scanned the movie poster for Need for Speed, but I hilariously got the trailer for the Need for Speed video game. What other funny outliers have you noted due to matching ambiguities?

Oh, thank you for reporting a bug! :) Yes, this kind of issue could happen. As I said, we collect data from several web resources, and since the data is somehow generated by machine, the association between movie title and YouTube Video ID is sometimes wrong. Another funny case we noticed early after launching was that when a user tried to recognize the Bollywood film *Queen*, they would get the a live performance from the band *Queen*. We fixed that one!

GlassFrogger: Hybrid Glassware Using the Browser

Winning features: use of remote HTTP server to run a game, sensor access via JavaScript

This notion of web applications working with Glass is powerful. How can your site integrate with Glass for sensor access or make use of JavaScript? Hybrid Glassware provides some suggestions at what might be possible.

As a testament to its potential for Glassware development, the team behind **GlassFrogger** (Figure A-17), which won 1st prize in the 2013 Breaking Glass hackathon in San Francisco, used a Dart codebase to create an homage to the classic 8-bit arcade game where an ambitious amphibian attempts to cross a busy highway to reach the safety of his beloved lily pad. The app is hybrid Glassware, using the Mirror API to insert a static card into the user's timeline as a splash screen, with a web app running in the Glass browser as the "native" end.

It avoids the need for time-consuming Android development, using the browser as a stage for a game running on an HTTP server. JavaScript listeners measure the Glass accelerometer to handle a player's real-life movement to dictate how the character advances on the screen. As a Glass wearer jumps up and down, the frog animates fluidly. The game's impressively responsive and really fun to play.

And perhaps most impressively, *this app was built in less than 48 hours*.

The entire game's code is open source and available **on GitHub** so you can study the implementation for yourself.

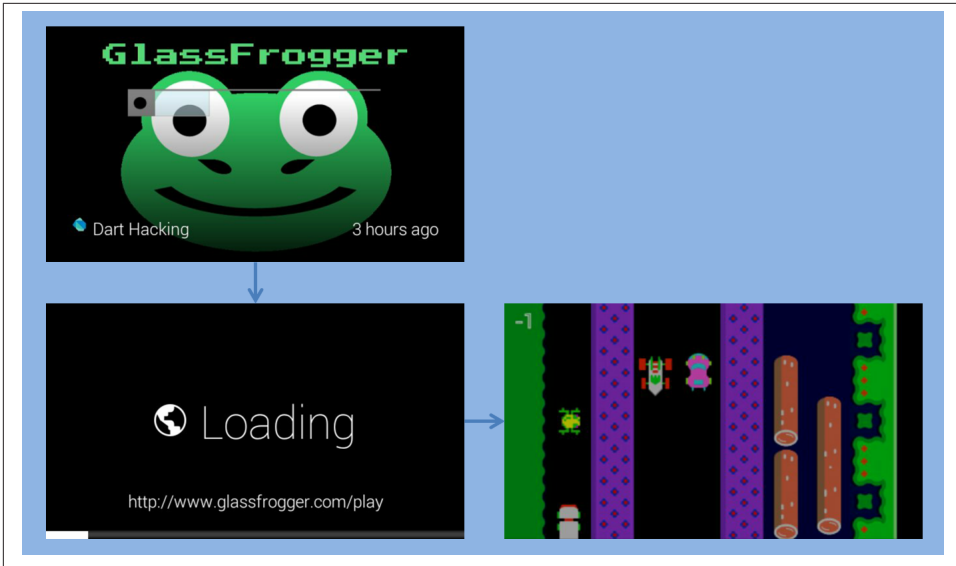


Figure A-17. GlassFrogger's program flow

This is architecture you may want to consider in your own projects, especially if your strong suit is in web development and not coding native apps in Java. You can write a complex web app and run it through the browser—just be mindful of performance issues due to JavaScript execution and graphics, the battery charge consumed by the projector being on for extended periods of time, and how the attention demands deviate from the microinteractions. It's worth looking into.

Hacking Glass

Glass was built with the intention that it would be taken apart and extended. This appendix is admittedly (and boastfully) esoteric—delving into what is viewed as a very much dark but highly respected art. Hackers are commonly misunderstood by the mainstream as being nefarious types who on principle exist to create chaos and unseat everything that organized groups have put together. That's actually a half-true statement.

Hackers are talented technologists who enjoy tinkering endlessly with hardware, components, and software modules to find new and innovative ways to get more utility from out-of-box products, or by inventing from scratch—but for good. They're a unique breed of people whose love of exploration and willingness to take things apart to see how they work is intensely vigilant. They use their considerable skills to investigate aspects of platforms that most people don't care about, and then share information with others in their community as a coordinated effort to make products better.

Hackers tirelessly make technology more valuable by constantly working to expand its capabilities and functionality, as well as to uncover shortcomings and oversights to make a system more stable, more secure, and more performant. They sometimes expose security flaws, performance hindrances, and operational inefficiencies in order to improve the quality of a technology, and typically only ask for acknowledgment for having found the glitch. It's a badge of honor.

Consequently, the Android lineage of Glass makes it a ripe environment for the hacker ethos to thrive.

The ability to hack Glass was made possible by Google fully supporting the curiosity to rebuild the system and expand upon its shipped capabilities or load a completely different Linux distribution. This was met with great zest from hackers worldwide, seeing that Google wasn't afraid to officially let its product be taken apart. Hacking Glass wasn't just possible—it was encouraged.

Those who dare to walk on the wild side can further expand the platform, even if doing so means possibly sacrificing their warranty. For those in whom the passion for hacking burns bright, it's a fair trade in exchange for discovery, knowledge, and community.

So if you share this way of life, want to try something interesting, or want to get involved, this appendix lets you explore the limits of your creativity and have some wacky, hacky fun.

The DIY Movement: When the Status Quo Just Isn't Good Enough

Hacking is fun, challenging, and rewarding. It's a very nonconformist attitude to uphold, even as a white hat. So it was no surprise that when the Glass Explorer Edition units began being distributed, a great many people wanted more out of the platform. Android savants wanted a way to install a launcher and run applications written for that OS on Glass, leading to their quick discovery of Debug Mode and the enabling of Android Debug Bridge (ADB) to sideload apps. In marketing Glass, Google stresses that true to their own hacker heritage the product was deliberately left unlocked so that it could be messed with. *Glass isn't just a platform that can be hacked—it's built to be hacker-friendly.*

What's more, the enthusiasm from Googlers shone through, as Glass team members nudged their fellow coders to go forth and challenge the limits of the system and take it into uncharted territory.

Sideloaded Android Apps on Glass

Debug Mode, which you can enable from the Settings bundle under the Device info card, lets you install apps on your device outside of the normal MyGlass channel by copying Android Packages (APKs) onto the device. To get Android apps to run outside of MyGlass, you'll need to enable Debug Mode and then have ADB running on a desktop or laptop computer while Glass is connected via its micro-USB cable.

- To install native apps manually on Glass:

```
adb install -r <PACKAGE_NAME.apk>
```

- To see what packages are installed on Glass:

```
adb shell pm list packages
```

- To uninstall an app from Glass:

```
adb uninstall <PACKAGE_NAME.apk>
```

Note that installing apps onto Glass this way doesn't at all give the user a heads-up about what permissions are required to run them.

Check out the [ADB section on the Android Developers support site](#) for a comprehensive list of commands and switches it can take. Further, you can use the ultra-cool [ChromeADB](#) extension, which is probably the easiest way to graphically manage the various packages installed on your headset.

Sharing Your Screen

Often, you'll want to share what you see in Glass on other displays, whether for doing demos, working out UI/UX quirks with a Glassware project you're building, helping a friend with tech support, or testing the system itself. In addition to MyGlass's screencast feature that mirrors your screen through your phone, you've got a couple other options once you've connected Glass to a desktop computer via USB and enabled Debug Mode from the Settings bundle):

Live sharing

You can show your screen live on your computer by using the free [Android Screen Monitor](#) tool. Once you've downloaded it on your machine, enter the following at a command prompt:

```
----  
java -jar asm.jar  
----
```

[Lance Nanek has a helpful post](#) about setting up Android Screen Monitor or [Droid@Screen](#), which is equally great.

Screen recording

You can also use the [Android Debug Bridge \(ADB\)](#) to record video of your screen usage for playback later, which is fantastic for social post sharing. Once you've got ADB installed on your computer, enter the following at a command prompt:

```
----  
adb shell screenrecord /sdcard/<FILENAME>.mp4  
----
```

Then, just start using Glass. Hit Ctrl-C when you're done and a video file will be written to Glass in the directory that's immediately accessible when you open Glass as a drive on your computer. You can also use ADB to download the captured video to your desktop by using the following:

```
----  
adb shell pull /sdcard/<FILENAME>.mp4 downloaded_video.mp4  
----
```

Android Studio

If you're working with the GDK to build your Glassware, Android Studio includes the ability to capture screenshots or screencasts of your app in action.

For more about recording your screen, check out the [ADB documentation](#).

Give Me JavaScript, or Give Me...

And all is not lost for those of you who just won't go quietly into the night and accept the fact that JavaScript isn't available within cards—there's *lots* of interest in using the language on Glass. University of Maryland PhD candidate [Brandyn White said](#), “To me, and ideal wearable computer would require almost no user input,” describing his motivation for cocreating [WearScript](#), a JavaScript library that lets applications handle native operations on Glass through JavaScript. In addition to manipulating the timeline, WearScript gives you access to the Glass sensors and displays changes to them in real time. This results in development that's on par with Android development, without all the complexities of native programming.

Similarly, developer [Brenda Jin highlighted](#) the ability to use PhoneGap to obtain real-time sensor data on Glass with JavaScript. In both cases, the libraries communicate with remote servers with impressive responsiveness.

So while we may not be able to do DOM or XMLHttpRequests directly from within cards or other types of client-side coding, there is work being done in our community by your fellow enthusiasts to give JavaScript a place. Check out their projects and get involved!

Officially Unofficial: Rooting Your Headset

This is the ultimate hack for our community. The feeding frenzy momentum of side-loading activity from the community led to one of the most anticipated sessions at Google I/O 2013, “[Voiding Your Warranty: Hacking Glass](#)”, which specifically addressed the hacking of Glass and the caveats that ensue. Not too long after Google publicly released the source code for the Google Glass kernel, which includes the core device drivers for the various hardware and system components. The kernel source and build instructions [can be downloaded](#) alongside a prebuilt GCC compiler and a prerooted bootloader. Google also mercifully provides the latest factory image so that if you brick your device you can always roll it back to the stock build.

However, Google issues [a stern warning](#) about hacking your device: “*Rooting, unlocking, or flashing your Glass voids your warranty and can leave your device in an irrecoverable state. You will no longer receive OTA updates if you unlock or root your Glass. There is no guarantee that you will receive OTA updates even after flashing back to factory specifications. Proceed at your own risk.*”

Rooting Glass is an advanced job that should only be undertaken by people that really know what they're doing or whose projects have very specific requirements, so the letter of the law as defined by our friends in Mountain View is that if you do go this route, you're on your own. That said, you can make Glass take on an entirely new personality by rooting it. Here's how to do it, if you want to go where eagles dare.

You'll need the file *boot.img* to be in the same directory as ADB. Next, perform these actions in sequence by typing the following commands at a command prompt:

- Access the bootloader
 - Ensure your device is recognized:

```
adb devices
```
- Reboot the device into bootloader mode:

```
adb reboot bootloader
```
- Unlock the device and erase personal data
 - Ensure your device is recognized by fastboot:

```
fastboot devices
```
- Execute OEM unlock

```
fastboot oem unlock
```
- Swap out and override the boot partition
 - Flash the boot image:

```
fastboot flash boot boot.img
```
- Reboot into normal state
 - Reboot the device again:

```
fastboot reboot
```
- Access root mode
 - Gain root access:

```
adb root
```

If you need to restore the system software to its shipped state, make sure you have *boot.img*, *recovery.img*, and *system.img* in the same directory as ADB and type the following at the command prompt:

- Ensure your device is recognized:

```
adb devices
```

- Reboot into fastboot mode:

```
adb reboot-bootloader
```

- Ensure your device is recognized by fastboot:

```
fastboot devices
```

- Flash all of the IMG files:

```
fastboot flash boot boot.img  
fastboot flash recovery recovery.img  
fastboot flash system system.img
```

- Erase user data and cache:

```
fastboot erase userdata  
fastboot erase cache
```

- Reboot the device:

```
fastboot oem lock
```

It's also worth noting that even though we outline the steps here to root Glass (and certainly there are other resources in a variety of media formats that detail how to get it done), it's still time well spent to take 36 minutes to **(re)stream the I/O presentation** and thoroughly consult Google's documentation to make sure you do it right.

Enjoy, good luck, be careful...and have fun!

Living on the Bleeding Edge

Whew! See, we told you this was a fairly involved appendix with a lot of insider information. This appendix fully embraced that Glass isn't merely an Android fork, but a product that stays true to its legacy from that OS, and proudly carries on the tradition of Android's openness.

We showed you how to hack Glass, gain root access, and load Android applications on your headset—and how to revert everything back. We also made you aware of the fact that hacking your unit carries with it a certain amount of risk, but the payoff if you know what you're doing can be really great.

Our parting shot would be that if you've got an idea, don't dwell! Write it up, sketch it out, put a prototype together, and start talking about it! Hacking isn't just a thing that

coders do. There are wonderful resources in the maker community, and **Maker Shed** is an excellent resource to share and refine your ideas and see what others are doing.

A

- accelerometer, 15
- acceptCommands attribute (Contacts), 231
- acceptTypesproperty, 225
- access tokens, 143, 204
 - expired, replacing, 144
 - including as part of HTTPS Authorization header, 144
- accessibility, Glass for, 311
- accessories for Glass, 315
- Accessory Development Kit (ADK), 313
- access_type parameter, 140
- AccountManager object, 263
 - using for other types of configuration, 265
- Accounts resource, 263
- actions, 199
 - custom menu items, 209
 - simple event actions, 199
 - subscriptions and, 202
 - thinking in actions, not apps, 14, 107
- Activity object, 281
- activity recognition, 240
- Activity.onCreate method, 270
- ADB (Android Debug Bridge), 350
- adult content on Glass, 44
- aesthetics
 - issues with Glass, 38
 - current state of, 39
 - designing to avoid criticism, 40
 - of the headset, 12
- align-justify class, 165
- alignment for text, 168
- All Notes notebook, 54
- Allthecoooks Recipes, 108
- analytics
 - current state of Glass analytics, 42
 - for your Glassware, 303
 - implications for Glass development, 43
 - metrics on wearables, 135
 - societal issues with Glass analytics, 42
- Android applications, 4
 - sideloading on Glass, 350
 - writing in Java, 243
- Android Debug Bridge (ADB), 350
- Android Developers documentation, 248
- Android Native Development Kit, 244
- Android Open Accessory protocol, 313
- Android OS, Glass and, 105
- Android Screen Monitor, connected to Glass, 249
- Android SDK
 - Glass Development Kit (GDK) components from, 245
 - synchronization tools, 97
- Android services, 252
- Android Studio, 80, 246
 - setting up Glassware project on, 266

We'd like to hear your suggestions for improving our indexes. Send email to index@oreilly.com.

- Android Wear
 - and the personal network, 241
 - design principles, 79
 - Glass and, 106, 313
 - Glass versus, 21
 - Glassware and apps driven by, 73
 - PayPal app for, 105
 - voice actions for, 286
 - android.text.HTML.fromHTML method, 272
 - Android@Home initiative, 311
 - AndroidManifest.xml file, 262, 284
 - animation, 268
 - API Console versus Developers Console, 132
 - API Explorer, 194
 - .APK files, 265
 - App Inventor, 246
 - App Statistics dashboard, 304
 - application model, 12
 - programs built using Glass Development Kit (GDK), 13
 - programs built using Mirror API, 12
 - application preferences, 83
 - appointment reminders from Google Now, 65
 - apps, servers versus, 124
 - AR (see augmented reality)
 - archival storage of Glassware, 45
 - Arduino, 313
 - article tags, 159
 - auto-paginate class, 184
 - cover-only class, 188
 - map within, 179
 - section and footer elements in, 169
 - separating content into multiple cards, 183
 - article.photo selector class, 174
 - attachments
 - Timeline.attachments collection, 130
 - to email, need for Glass improvement in, 58
 - to timeline items, 196
 - sharing, 227
 - audio
 - audio-only files attached to timeline items, not supported, 225
 - card contents read aloud, 182
 - importance in Glass design, 81
 - notifications, 181
 - Augmate, 309
 - augmented reality, 40
 - current state of, Glass and, 40
 - Glass becoming leading contender as AR client, 41
 - unrealistic expectations for Glass, 111
 - authentication, 141
 - authenticated URL fetch, 227
 - GDK apps, 263
 - authorization, 138, 141
 - (see also OAuth)
 - processing authorization code (or error), 142
 - Authorization header (HTTPS), 144
 - Authorized JavaScript origins field, 150
 - Authorized Redirect URI field, 150
 - auto-overflow class, 188
 - auto-paginate class, 184
- ## B
- background images, 173
 - mosaics, 175
 - bandwidth, avoiding excessive use of, 91
 - banking industry, Glass and, 104
 - batteries
 - battery life and Glass usage, 20
 - battery-safe programming with Mirror API, 123
 - immersions' demands on, 253
 - Blippar framework, 41
 - bold text on cards, 162
 - bone conduction transducers, 20
 - bounce rate for Glass content, 42
 - bridged notifications, 260
 - brightness of the display, 15
 - Brivo Labs, OKDoor Glassware, 115
 - bundleId property, 190, 219
 - bundles, 21, 85, 183
 - applying auto-paginate class, 185
 - default ordering in Glass, 191
 - duplicating features of bundled cards in
 - GDK apps, 283
 - isBundleCover timeline item property, 192
 - sample card for, 183
 - typical flow, 184
 - using a cover card, 188
 - business of producing Glassware, 45
 - businesses, use of Glass in the workplace, 307
 - BYOD (bring your own device) concept, 307
- ## C
- callbackUrl (Subscriptions.insert), 205

- camera, 22
 - indicating in use, 34
 - limitations of, 23
 - shutter button control, 258
 - voice actions with, 25
- canceling in-process actions, 82
- Canvas, 246
- CAPTION layout, applying images to, 278
- captions, adding to shared resources, 229
- card-based metaphor, Glass UI, 12
- CardBuilder class, 269
- CardBuilder.addImage method, 177
- CardBuilder.Layout enumeration, 269
- CardBuilder.Layout templates, 283
- CardBuilder.Layout.AUTHOR template, 274
- CardBuilder.Layout.COLUMNS, 275
 - using icons, 276
- CardBuilder.Layout.COLUMNS_FIXED, 275
- CardBuilder.Layout.TEXT template, 269
- CardBuilder.Layout.TEXT_FIXED template, 270
- CardBuilder.Layout.TITLE template, 279
- CardBuilder.Layout.CAPTION template, 278
- cards
 - actions, 199
 - and subscriptions, 202
 - custom menu items, 209
 - simple event actions, 199
 - always testing on Glass, 217
 - automatic deletion of older cards, 82
 - bundling, 85
 - Contact object, 216
 - design, official templates for, 80
 - footer area in, using, 59
 - layout, 79
 - live cards in GDK apps, 249–253
 - live versus static, ordering of, 62
 - master copy in your database, 208
 - pinned, 21
 - security and, 138
 - working with, 153–198
 - audio, 181
 - bundles, 183
 - CRUD operations, 197
 - footer, 169
 - going beyond Mirror API Playground, 193
 - Hello, World!, Glass-style, 153–158
 - HTML, 158–170
 - images, 170–174
 - JSON representation, viewing, 156
 - media attachments to timeline items, 196
 - rendering an in-card map, 178–180
 - updating previously-inserted cards, 155
 - using Playground to prototype and test, 154
- CardScrollAdapter object, 280
- CardScrollView, 270, 280
- CardScrollView.setOnItemClickListener, 280
- changelog, 259
- CharSequence, 271
- Chromecast, 249
 - Glass and, 312
 - Glass integration with, 104
 - LynxFit integration with, 330
- Chrome's Developer Tools, 124
- classes (GDK), 259
- client_id parameter, 140, 143, 149
- client_secret parameter, 143, 149
- cloud computing
 - designing for the cloud, 74
 - not fully utilizing, 113
- cloud-based services, 3
 - Mirror API programs, 12
- code parameter, 143
- collaboration Glassware, Vodo, 342
- collection parameter (Subscriptions.insert), 204
- collection property, 207
 - locations, for Subscriptions, 234
- color options, Glass frame, 39
- colors
 - for text on cards, 162
 - Glass headset, 29
 - mixing for text instead of font sizes, 164
 - searching by, 325
- ColorSnap Glassware, 113, 325
- columns
 - absolute positioning, 166
 - columnar layouts in GDK, 275
 - layout classes in CSS, 167
- com.google.android.glass namespace, 259
- communities (Glass), 100
- Concur Glassware, 105
- configuration, web-based, providing, 83
- CONFIRMED state, 214
- Contact object, 216

- contacts, 131
 - share, 222–230
 - voice commands targeting, 230–232
 - Contacts collection, 216
 - Contacts.acceptCommands, 231
 - Contacts.insert method, 222
 - containers for data, 252
 - content
 - avoiding gotchas in, 90
 - content creation in POV world, 28
 - Glass as content creation utility, 46
 - contentUrl, 227
 - contextual commands, 286
 - contextual information, 65, 88, 233–242
 - environments of users, tailoring Glass design around, 63
 - future uses of, 241
 - in vignettes, 64
 - location and localization, 238
 - location as part of timeline events, 236
 - location services, enabling, 234
 - making user’s real-life activities core part of application experience, 66
 - other than location, 240
 - subscribing to locale changes, 239
 - time zone for user’s location, 237
 - tracking with Google Now, 65
 - wearer’s environment as part of Glass UI, 99
 - cooking assistant, KitchMe Glassware, 323
 - core design principles (see design principles, core)
 - corporate Glass, 307
 - cover-only class, 188
 - crash reports for Glassware, 287
 - creator attribute, timeline items, 216
 - credentials (OAuth), 149
 - CrowdOptic, crowdsourced broadcasting model, 339
 - CSS (Cascading Style Sheets)
 - base CSS styles for Glass, 159, 269
 - built-in layout classes, 167
 - formatting options, CSS classes, CSS selectors, and inline styles, 163
 - using with HTML for cards, 158
 - CUSTOM MenuItem action, 209
- ## D
- Dalvik Debug Monitor Server (DDMS), 265
 - data analysis, using Glass for, 110
 - data stores, 131
 - for Mirror API projects, 136
 - DateUtil class, 271
 - Dawn Data, 110
 - Debug Mode, 265
 - DEFAULT state (menu item actions), 214
 - DELETE menu item, 206, 213
 - PENDING and CONFIRMED states, 214
 - deleting versus dismissing items in Glass, 82
 - design principles, core, 77
 - advocating multitasking, 99
 - Avoid the unexpected, 90
 - error handling, 94
 - in content, 90
 - not hogging bandwidth, 91
 - performance, 91
 - permissions, 93
 - surprises, pleasant, 97
 - synchronization across platforms, 96
 - Build for people, 98
 - design for Glass, 78
 - audio, 81
 - deleting versus dismissing items, 82
 - providing web-based configuration, 83
 - tactical wearable design, 80
 - targeting microinteractions, 78
 - Don’t get in the way, 84
 - exceptions, 87
 - less is more, 86
 - notifications, 84
 - issue commands, don’t start apps, 112
 - Keep it relevant, 87, 233
 - contextual information, 88
 - quickness of delivery, 89
 - targeting microinteractions, 78
 - design, Glass design rivaling Apple products, 16
 - designing for Glass, 78–84
 - avoiding poor design, 103–117
 - choosing wrong development framework for Glassware, 114
 - Glass not necessarily bound to your phone, 110
 - improperly implementing ideas for Glass experience, 104
 - not fully utilizing cloud computing, 113
 - not using prism display for complicated reports, 110
 - overloading the system and wearer, 106

- sticking to prefab templates and styles, 109
- thinking in actions, not apps, 107
- treating Glass like any mobile device, 105
- unrealistic expectations for augmented reality and gaming, 111
 - using default voice commands, 112
- design patterns, official documentation, 80
- designing for the cloud, 74
- glanceable layouts and structure they imply, 177
- Gmail Glassware, 56
 - adding value by omission, 58
 - design takeaways, 58
- Google+ Glassware, 58
 - areas needing improvement, 60
 - design takeaways, 60
- greater goal of wearable design, 71
- outstanding examples, 52
- outstanding Glassware, design takeaways, 61
- Think for Glass mindset, 63
- Twitter Glassware, 53
 - design takeaways, 56
- developer groups, 100
- Developer Policies, 34
 - gotchas for developers, 298
- developer settings, 268
- Developers Console, 132, 194
 - API Console versus, 132
 - App Statistics dashboard, 304
 - metrics on wearables, 135
 - obtaining Client ID information from, 151
- development frameworks, 13
 - (see also Glass Development Kit; Mirror API)
 - choosing wrong framework for Glassware project, 114
- directions
 - getting with voice actions, 25
 - turn-by-turn directions, 81
- Dismiss menu item, 213
 - PENDING and CONFIRMED states, 214
- dismissing versus deleting in Glass, 82
- displayName property (Contacts), 223
- div element, using for absolute positioning of columns, 166
- documentation, reference, for Mirror API
 - methods, 195
- DriveSafe app, 38

- driving, using Glass and, 37, 44
 - current state of concerns, 37

E

- Eclipse, 80
- education, Glass in, 309
- ellipses
 - indicating more content in GDK cards, 274
 - indicating more text in Gmail Glassware card, 58
- email scope, 147
- Emotient, 36
- error handling, 94
 - considerations for Glassware, 295
- events, 125
 - adding menu items to an event, 199
 - location as part of timeline events, 236
 - location, subscribing to, 234
 - related to the timeline, working with, 130
 - security for, 137
- Evernote, Glassware, 54, 100
- eyeFlame built Glassware, 88

F

- Facebook, multiple scope URLs, 146
- facial recognition, 34
 - current state of, 35
 - designing to avoid problems with, 36
 - NameTag app, 336
- Factory reset option (Settings), 258
- Fancy Glassware, 95, 113, 325
- fashion, issues with wearing Glass, 38
- Field Trip Glassware, 41, 88
- figure tags, mosaic list in, 176
- filtered data streams, Gmail Glassware, 56
- financial services industry, Glass and, 104
- first-person perspective, content creation and, 28
- Five Noble Truths (see design principles, core)
- footer element, 169
- footers (timeline cards), 169
 - adding a gradient, 174
- forced perspective, 19
- form factor, 12
- frames, custom, for prescription glasses, 39

G

- GAE (Google App Engine), 131
- gaming
 - GlassFrogger, 346
 - unrealistic expectations for Glass, 111
 - using Glass, 71–74
 - virtual pet for Glass, 73
- GDK (see Glass Development Kit)
- Genie Glassware, 331
- Get directions (voice command), 25, 258
- GitHub
 - GlassFrogger code on, 346
 - OpenGL demo, code repo, 247
- Glanceable UI, 79
 - glanceable layouts and structure they imply, 177
- Glass
 - Android Wear versus, 21
 - for gaming, 71–74
 - future of, 307–315
 - accessibility, 311
 - Android Wear, 313
 - Chromecast and home entertainment, 312
 - corporate Glass, 307
 - hardware hacking and Internet of Things, 313
 - home integration, 311
 - in medicine and education, 309
 - peripherals/accessories, 314
 - streamlining operations, 308
 - hacking, 349–355
 - mirroring what is available on other platforms, 209
 - removing from the head, effects on apps, 260
 - using the system, 20
- Glass Asset Studio, 211
- Glass Creative Collective, 309
- Glass Development Kit (GDK), 13, 243–290
 - advantages of using, 248
 - authentication, 263
 - choosing between Mirror API and, 289
 - configuring voice commands, 284
 - developers mistakenly using instead of Mirror API, 114
 - differences from Mirror API, 247
 - documentation, 268
 - drawing and animation, 246
 - Glassware built against, design takeaways, 62
 - Glassware shipping with Glass, 257
 - Google approval of Glassware built on, 302
 - Google's control over voice commands, 287
 - hybrid Glassware, 261
 - installed apps running on Glass, 243
 - Mirror API and, 244
 - object model, 259
 - On-Head Detection feature, halting running apps, 260
 - overview of, 245
 - add-on, components to work with Glass UI, 245
 - basic components from Android SDK, 245
 - porting existing apps to Glass, not recommended, 288
 - system intents, 259
 - testing native Glass apps, 267
 - tools for rapid design, 255
 - updating releases, versioning, and crash reports, 287
 - user interface elements of apps
 - immersions, 253
 - live cards, 249–253
 - view to a card, 268
 - basic text formatting, 269
 - columnar layouts and mosaics, 275
 - creating rich text, 272
 - creating your own layouts, 283
 - ellipses and excess content, 274
 - other neat templates, 279
 - using icons, 276
 - writing native code for Glass, 265
- Glass ecosystem
 - how it is different, 9
 - platform, not a product, 10
 - technical specifications (as of Explorer Edition), 17
 - timeline and application code running in completely separate processes, 244
 - what it isn't, 46
- Glass Frogger, 72, 346
- Glass Hunt, 72
- Glass Platform Developer Policies, 34, 296
- Glass sync, 3
- glass.location scope, 236
- GlassCopter, 72
- Glassentation, 106

- Glassware, 12
 - authorization, 137
 - built using Glass Development Kit (GDK), 13
 - built using Google Mirror API, 12
 - business of producing, 45
 - case studies from the field, 319–347
 - CrowdOptic, 339
 - Fancy and ColorSnap, colorful queries, 325
 - Genie, 331
 - GlassFrogger, 346
 - KitchMe, 323
 - LynxFit, 328
 - MiKandi, 333–336
 - NameTag and facial recognition, 336
 - Preview, 343
 - Thurz Sports, 320
 - Vodo, 342
 - choosing wrong development framework for your project, 114
 - getting listed on MyGlass, 291–303
 - launching via voice commands, 26
 - making money from, 304
 - native apps shipping with Glass, 257
 - official listing, 31
 - outstanding examples of, 52
 - design takeaways, 61
 - Evernote, 54
 - Gmail, 56
 - Google+, 58
 - Twitter, 53
 - primary entry points for, 109
 - review of, 292
 - submitting for review by Google, 291
 - timing your release of, 303
- Glassware Flow Designer, 80, 255, 269
- Glassware Launch Checklist, 296
- Glassware Review Request, 299
- GlassWireframe, 256
- global distribution of Glass, 43
- Gmail Glassware, 56, 97
 - delivery of messages from Important folder only, 56
 - design takeaways, 58
 - organizing conversation threads into distinct bundles, 58
- GolfSight app, 254
- Google
 - OAuth URL, base, 140
 - review process for Glassware, 291
- Google Analytics
 - integration with Glass, 110
 - integration with Glass cloud-based services, 43
- Google APIs, use by Mirror API projects, 136
- Google App Engine (GAE), 131
- Google Apps Script (GAS), 124
- Google Cast API, 104
- Google Cloud Endpoints, 74
- Google Cloud Messaging for Android, 127
- Google Cloud Platform, 74
 - information about, 131
- Google Developer Groups, 100
- Google Developers Console, 124, 132, 194
 - App Statistics dashboard, 304
- Google Developers site, 77
- Google Drive, collaboration for users working in, 113, 342
- Google Goggles, 34
- Google Mirror API (see Mirror API)
- Google Mirror API Playground (see Mirror API Playground)
- Google Now, 65
- Google Play Services, GDK apps and, 43, 249
- Google Search on Glass, 68–71
- Google+, 34
 - driving force behind this book, 61
 - enabled on Glass, 22
 - Glassware, 58
 - location services moved to, 234
- Google+ API, 148
 - people.get method, 148
- Google+ Glassware, 100
 - design issues, room for improvement, 60
- GPOP, 314
- GPS, 63, 105
 - resource-intensity of checking for signals, 233
- gradient, adding to a timeline card footer, 174
- grant_type, 143
- graphical data (complex), not using Glass for, 110
- graphics
 - free tools for, 211
 - in GDK apps, 268
- gyroscope, 15

H

- h1 and h2 tags, keeping everything on one line, 172
- hacking Glass, 349–355
 - JavaScript, 352
 - rooting your headset, 352
 - sharing your screen, 351
 - sideloading Android apps on Glass, 350
- Hangouts Glassware
 - example of hybrid Glassware, 262
 - messaging via, 25
 - mosaics in, 174
 - synchronization across platforms, 97
 - virtual tour of CERN via, 309
- Happy Aquarium, 73
- hardware hacking, Glass and, 313
- head movement gestures, 14, 311
- head-mounted display (HMD), 4
- headset
 - evolution of, 9
 - modular bundle of technology in, 10
- healthcare, Glass in, 309
- high-frequency rendering, 251
- home automation applications, 89
- home entertainment, using Glass, 312
- home integration, using Glass for, 311
- home screen, 20
 - (see also timeline)
- hosting for Mirror API services, 125
 - Google App Engine (GAE), 131
- hostnames, 150
- HTML
 - formatting for scientific notation, 168
 - Glass convenience classes for text, 168
 - images, 171
 - lists, formatting for Glass, 164
 - tabular data on cards, 165
 - tags accepted for timeline cards, 159
 - using with CSS for cards, 158
- HTML.fromHTML method, 272
- HTTP, 122
 - for events sent to Glass, 126
- HTTP services, Mirror API, 84
- HTTPS, 131
 - Glassware making HTTPS calls to Google API server, 128
- HTTPS Authorization header, 144
- human-computer interaction (HCI)
 - further progress with Glass, 63

- reinvention of, 16
- hybrid Glassware, 258, 261
 - GlassFrogger, 346
- Hybrid Mosaic template, 173

I

- icon-small class, 171
- icons, 171
 - in columnar layouts on GDK cards, 276
 - in GDK app TITLE layout, 279
 - in GDK apps, 277
- id (sharing contacts), 223
- IDEs (integrated development environments), 80, 246
 - working with Mirror API, 124
- images
 - applying to TEXT layout in GDK app, 278
 - geotagging of, 237
 - in GDK apps, 277
 - in TITLE layout, GDK app, 279
 - mosaics on GDK app cards, 275
 - on cards, 170
 - on timeline cards
 - background photos, 173
 - list-based layouts and figure layouts, 177
 - mosaics, 174
 - working with medium-sized images, 172
 - working with small images, 171
 - optimal density in GDK apps, 257
- imageUrls (sharing contacts), 224
- img tags, 171
 - src attribute, 178
- immersions, 14, 87, 253
 - apps based on, examples of, 289
 - development patterns for, 255
 - dismissing, 254
 - GDK immersions used as gaming stages for Glass, 111
 - native apps unnecessarily built as, leading to negative performance, 253
 - On-Head Detection and, 261
 - summary of characteristics, 255
 - UI dissimilar to timeline motif, 109
- Incremental Authorization, 147
- Ingress, Glass and, 111
- input mechanisms, 14
 - criticism of Glass, 71
 - designing for fewer steps, 108
- inReplyTo, 219

- INSERT operation, 218
- Instagram, multiple scope URLs, 146
- installation, errors encountered during, 95
- integrated development environments (see IDEs)
- international distribution of Glass, 43
- internationalization, Google+ Glassware issue with, 60
- Internet of Things, 115, 314
- IP address
 - checking against to verify requests, 205
 - for your server, 131
- isBundleCover property, 192
- isDeleted, 208
- isPinned, 208
- isProcessingContent (Timeline.attachments), 227
- italic text on cards, 162
- itemId property, 207, 208

J

- Java
 - using with Mirror API, 122
 - writing Android applications in, using GDK, 243
- JavaScript, 246
 - in Glass, 159
 - using on Glass, 352
- JavaScript Object Notation (see JSON)
- JavaScript origins, 150
- JSON (JavaScript Object Notation), 128
 - callback body, 206
 - card representation, viewing, 156
 - custom menu item action, 209
 - for bundled cards, 189
 - object model for JSON resource, 129
 - response containing OAuth access token, 143
- Justinmind Prototyper Pro, 256

K

- KitchMe Glassware, 323

L

- Lambda Labs, Face API, 35

- languages
 - supporting languages other than English, 296
 - voice commands adapted for, 238
- latitude (Locations object), 235
- Layar framework, 41
- layout
 - CardBuilder.Layout enumeration, 269
 - CardBuilder.Layout templates, list of, 283
 - custom, creating for GDK apps, 283
 - layouts supported by live cards, 251
 - templates for, 268
- layout classes in CSS, 167
- LinkedIn, multiple scope URLs, 146
- live cards, 249–253
 - apps based on, 289
 - as dynamic containers for data, 252
 - layouts and views supported, 251
 - On-Head Detection and, 261
 - Stop command, 250
 - summary of characteristics, 255
 - two flavors of, 250
- localization, 238
 - considerations for Glassware, 296
 - locale settings, 239
 - tips for localizing your Glassware, 240
- Location object, 200
- locations, 131
 - enabling location services in Glass, 234
 - geotagging images, not available in Glass, 237
 - localization and, 238
 - location as part of timeline events, 236
 - location services in Glass, immaturity of, 233
 - subscribing to locale changes, 239
 - user location and context, 233
 - uses of location in Glassware, 237
- Locations object, important fields, 235
- Locations.get method, 235
- longitude (Locations object), 235
- Loves Data, 110
- low-frequency rendering, 251
- LynxCast, 104
- LynxFit, 104
 - case study, 328
 - using prefab templates and having its own personality, 109

M

- machine-to-machine hardware hacking, 115
- magnetometer, 15
- Make a call to (voice command), 26
- Make Vignette menu item, 64
- Maker Shed, 355
- maps
 - in navigation system application, 253
 - rendering an in-card map, 178–180
- marketing channels for Glassware, 304
- me (user ID), 148
- medicine and education, Glass in, 309
- menuItems property, 182, 200, 211
- menus
 - adding to events, 199
 - custom actions as menu items
 - Gmail Glassware, 56
 - Twitter Glassware, 54
 - custom menu items, 209
 - keeping track of state of menu items, 54
 - setting values for menu items, 201
 - TOGGLE PINNED and DELETE menu items, 206
 - using contextual commands for interaction with, 286
 - VOICE CALL menu item, 216
- Message Queue Telemetry Transport (MQTT), 115
- messages
 - creator and recipients, 217
 - message threads, replies attached to, 219
 - pinned by users, 208
 - sending via voice actions, 25
- microinteractions, 5, 11
 - and required attentiveness of the wearer, 81
 - augmented reality (AR) and, 41
 - emphasizing as core unit of usability for Glass experience, 78
- MiKandi Glassware, 333–336
- MIME types, 225
- Mini Games Glassware package, 72, 111, 246
- Mirror API, 12, 121–136, 242
 - advantages over native development, 122
 - analytics and, 42
 - body properties versus parameters, 195
 - choosing between GDK and, 289
 - components of, 130
 - data flow in, 127
 - differences from GDK, 247
 - disadvantages of, 123
 - events, 125
 - examining methods in API Explorer, 194
 - flow of information between Glassware, Google’s cloud, and Glass, 125
 - GDK and, 244
 - Glassware built with, design takeaways, 61
 - Google approval of Glassware built on, 302
 - how your server talks to Google, 128
 - hybrid Glassware, 261
 - maps rendered in 2D map view, 179
 - perceived as being too limited, 114
 - preparing your project, 131
 - project quota, 134
 - quota for calls against, 54
 - security, 137
 - servers versus apps, 124
 - services running on HTTP servers, 84
 - subscriptions documentation, 205
 - using with cloud computing, 113
- Mirror API Issue Tracker, 112
- Mirror API Playground, 80, 124, 153
 - going beyond, 193
 - maps not rendered in, 179
 - previewing prototypes of menu items, 210
 - testing prototypes, 154
 - working with content, 161
- mirroring, 248
- mobile applications, design landmarks, 63
- mobile devices, treating Glass like any other device, 105
- monetization of Glassware, 45, 304
- mosaics, 173
 - in GDK apps, 275
 - working with, 174
- movies, optical illusion technique in, 19
- MQTT (Message Queue Telemetry Transport), 115
- multitasking in Glass, 99
- MyGlass, 16
 - getting on, 291–305
 - advantages of, 292
 - categorical listings, 303
 - factors to think about before submission, 294
 - objective of Glassware review, 292
 - prereview activities, 293
 - review process, 301
 - submitting your Glassware, 296

- requesting your Glassware not be listed on, 297
- use to disable permissions, 149

N

- NameTag app, FacialRecognition.com, 35, 336
- native Glass development, 244
 - (see also Glass Development Kit)
- NAVIGATE menu item, 200
- navigation system application, 253
- The New York Times, Glassware, 84
- no-border class, 164
- Note from Glass, 54
- Notification Sync, 260
- notifications
 - bridged, 260
 - Google+ Glassware, 59
 - handling responsibly, 84
 - synchronization across platforms, 97
 - Twitter Glassware, 53

O

- OAuth, 138–151
 - credentials, 149
 - geodata, rights to use, 234
 - getting user's identity, 145
 - Google's OAuth provider screen for Glassware, 141
 - Incremental Authorization, 147
 - OAuth2, 138
 - steps in authorization process
 - authorization (and perhaps authentication), 141
 - processing authorization code or error, 142
 - redirection to Google for authorization, 140
 - using (and refreshing) the access token, 144
- OAuth providers, 13
- object model (GDK), 259
 - packages, 259
- object recognition, 34
 - Glassware using, 36
- offline access, 247
- offline, using Glass, 17
- OK Glass, 20
 - activating voice interactions with Glass, 23

- alternative hotwords, not chosen, 27
- voice commands, how they are chosen, 26
- OKDoor Glassware, 115
- On-Head Detection feature, 260
- onCreate method, 269
- Ongoing Task pattern, 250
- onKeyDown method, 258
- OPEN URI menu item, 200, 261
- Openclipart.org, 211
- OpenGL, 246
- operation property, 207
- optics pod, 15
- overheating
 - avoiding in Glass headset, 91
 - Glass failsafe against, 257
- overlay-gradient-short class, 278
- overlays, 174

P

- p (paragraph) tags, 159
 - icon-small class and text-small, 171
 - text-auto-size class attribute, 160
- packages (GDK), 259
- Pandora Glassware, 253, 295
- parameters, method parameters versus body properties, 195
- PENDING state, 214
- performance
 - avoiding overheating of Glass headset, 91
 - immersions and, 253
 - of installed apps on Glass hardware, 248
 - of your Glassware, 295
- peripherals/accessories, 314
- permissions
 - Android permission for custom voice commands, 285
 - detailed information about, 141
 - disabling and requiring, 149
 - scopes and, 146
 - service-level, for working with user's data, 93
- personal area networks, xvii
- personal networks, 240
 - Android Wear and, 241
- personal technology, Glass as, 51
- phone calls, making via voice command, 26
- PhoneGap, 267
- photo editing Glassware, sharing contacts for, 225

- pinned items
 - not using pinned cards to launch applications, 108
 - ordering of, 62
- pinning cards, PIN and UNPIN actions, 207
- pixel density and screen size, 257
- platform-as-a-service providers, 294
- Play a game (voice command), 26
- PLAY VIDEO menu item, 200
- plus.login scope, 147
- plus.me scope, 147
- plus.people.get API, 148
- plus.profile.emails.read scope, 147
- pornographic content, Glass and, 44
- porting existing apps to Glass, not recommended, 288
- Post an update (voice command), 56, 230
- Post an update to (voice command), 26
- POV cinematography and ad-lib narration, 28
- power options, 22
- presentation, 153
 - GDK, unique presentation elements, 248
- Preview, 113, 343
- prism display, 15
 - always using as preview monitor when camera is in use, 34
 - challenge of small dimensions, 104
 - not using for complicated reports, 110
- privacy issues, 32
 - current state of, 33
 - location and, 234
 - questionable places for use of Glass, 32
- profile scope, 147
- programming languages
 - flexibility with Mirror API, 122
 - GDK, using Java, 246
- Project ID, 132
- Project name (for Mirror API), 132
- projection unit, 15
 - great consumer of battery charge, 254
 - science behind, 18
- publish/subscribe model, Mirror API framework, 13

Q

- quota for Mirror API calls, 54, 134

R

- radiation, Glass and, 17
- rapid design tools, 255
- Raspberry Pi, 313
- READ ALOUD menu item, 200
- READ MORE menu item, 202
- real-time interactivity, 247
- recipes archive, KitchMe Glassware, 323
- recipients
 - getting from Timeline.recipients property, 177
 - specified with Timeline.recipients property, 216
- Record a video (voice command), 25
- redirect URIs, 150
- redirect_uri parameter, 140, 143
- reference documentation for API methods, 195
- referrer logs, 42
- regulatory environment, Glass and public policy, 43
 - current state of, 44
 - developing within guidelines, 45
- release early, release often, 295
- relevance, emphasizing, 87
 - location and contextual information, 233
- remote control for Glass, 14, 315
- RemoteViews object, 251, 283
- REPLY ALL menu item, 219
 - voice-entered text and, 220
- REPLY menu item, 218, 262
 - voice-entered text and, 220
- resources, sharing, 221, 223
 - (see also sharing)
- response_type parameter, 140
- REST, 128
- RESTful API versus native SDK debate for Glass, 114
 - unifying the camps, 116
- RESTful services, crafting with decoupled components, 74
- Revolv, home automation work, 312
- Ribeiro, Justin, 115, 211
- rich text, creating in GDK app, 272

S

- scalability, 294
- scope parameter, 140

- scopes, 139, 146
 - getting the bare minimum, 147
 - multiple, for Mirror API Glassware, 146
 - permission to view user's location, 234
 - requesting information from the user, 148
- screen size and pixel density, 257
- searches, 24
 - color as query, 113, 325
 - Google Search on Glass, 68–71
- second-screen companion, Thurz Sports, 320
- section tags, 159
 - div elements in, 166
- security, 137–151
 - disabling and reacquiring permissions, 149
 - for events, 137
 - OAuth, 138
 - of your Glassware, 294
 - subscriptions, 203
- Send a message to (voice command), 25
- sensors
 - access to Glass sensors, 13
 - access to, with GDK, 248
 - Android sensors and APIs for interacting with them, 248
 - LynxFit access to Glass sensors, 328
 - on Glass, 15
- servers
 - for Mirror API projects, 131
 - how your server talks to Google, 128
 - versus apps, 124
- service accounts, 263
- services
 - Android services driving lifecycle of live cards, 252
 - creating and configuring Google API service object, 263
 - starting at bootup for live cards, 252
- settings
 - Debug Mode, 265
 - Factory reset option, 258
 - Glass developer settings, 268
 - subscribing to Settings collection, 239
- Settings.get method, 237
- SHARE menu item, 54, 221
- Share voice command, 223
- sharing, 221–232
 - contacts, 222–230
 - captioning shared resources, 228
 - inability to share multiple entities at once, 228
 - voice commands, 230–232
 - sharingFeatures property, 229
- Sherwin-Williams, Fancy and ColorSnap Glassware, 113, 325
- smartphones, 4
 - Glass and, 16
 - Glass pairing with, 110
- smartwatches, 106
- social norms, software respecting, 6
- social, Glass as, 99
- societal issues with Glass, 31–47
 - aesthetics, 38
 - analytics, 42
 - augmented reality, 40
 - business of producing Glassware, 45
 - facial recognition, 34
 - privacy, 32
 - regulatory environment and public policy, 43
 - using Glass while driving, 37
 - what Glass isn't, 46
- Society of Glass Enthusiasts (SoGE), 100
- sourceItemId, 219
- sourceItemId property, 208
- SpannableString class, 272
- spatial depth for prism display, 19
- speakable text, setting, 81
- speakableText property, 182
- specifications, 17
- Spellista, 111
- Sports template, 170
- SSL certificates, 131
 - subscriptions and, 203
- Stanford MedicineX series of Hangouts On Air, 311
- Start a timer (voice command), 26
- static cards, 249
 - in hybrid applications, 261
 - manipulating programmatically, 262
 - major difference from live cards, 250
- Stickman Ventures, Justin Ribeiro, 115
- Stop command, 250
- Stopwatch app, 108
- Strava's fitness apps, 86
- streamlining workflows and business processes, 308

- styles
 - defining custom style and using as them for menu activities, 268
 - using built-in styles, 109
 - styling
 - for lists, 164
 - options for, 163
 - using style rules in Glass, 159
 - subscription proxy, 203
 - subscriptions, 130, 202
 - adding with Subscriptions.insert method, 203
 - Mirror API documentation on, 205
 - responding to subscription pings, 205
 - security, 203
 - simple callbacks and how to handle them, 206
 - subscribing to locale changes, 239
 - to location events, 234
 - Subscriptions.collection property, 234
 - swiping/tapping on the trackpad, 14
 - synchronization across platforms, 96
 - system intents, 259
 - bridged notifications, 260
 - documentation on, 260
- ## T
- tables
 - HTML tables, data on cards, 165
 - not using for absolute positioning of columns in HTML, 166
 - Take a note (voice command), 230
 - Take a note with (voice command), 26
 - taking Glass off, effects on apps, 260
 - Talkray, 240
 - technical specifications, 17
 - telemedicine, use of Glass in, 309
 - telemetry applications, using Glass, 310
 - telephony device, Glass as, 16, 110
 - templates
 - built-in, using for Glassware, 109
 - Google Mirror API Playground, 154
 - images in, 170
 - layout templates for GDK apps, 268
 - list of templates available through Card-Builder, 283
 - text
 - creating rich text in GDK apps, 272
 - formatting for cards in GDK apps, 269
 - formatting on cards, 162
 - Glass convenience classes for, 168
 - in GDK app TITLE layout, 279
 - mixing colors instead of font sizes on cards, 164
 - size in cards
 - auto-sizing, 160
 - sizes defined by Glass, 161
 - TEXT layout in GDK apps, applying images to, 278
 - text property, 182
 - text-small class, 270
 - TEXT_FIXED layout, 270
 - themes, 268, 314
 - Think for Glass, xv, 63
 - aesthetics, 40
 - augmented reality, 41
 - driving while using Glass, 38
 - facial recognition issues, 36
 - meaning of, 5
 - privacy concerns, 34
 - regulatory environment and public policy, 45
 - what Glass isn't, 46
 - Thurz Sports Glassware, 320
 - time zone, determining from user's location, 237
 - timeline, 20
 - and application code, running in separate processes, 244
 - bundles on, 183
 - CRUD operations on, 197
 - events as building blocks of, 125
 - events related to, working with, 130
 - immersions and, 254
 - information and events in, 21
 - location as part of timeline events, 236
 - UI and cards in, 15
 - timeline cards (see cards)
 - Timeline object
 - documentation, 156
 - html property, 159
 - list of methods, 158
 - property names and descriptions, 157
 - reference documentation for, 195
 - Timeline.attachments, 227
 - Timeline.attachments.contentUrl, 227
 - Timeline.attachments.insert method, 197
 - Timeline.delete method, 213
 - Timeline.get method, 208

- Timeline.insert method, 195
- Timeline.list method, 220
- Timeline.menuItems.action property, 221
- Timeline.patch method, 212, 219
- Timeline.recipients property, 177, 216, 262
- Timeline.speakableText property, 81
- Timeline.text property, 81
- Timeline.update method, 213, 219
- Timeline.userActions.type, 227
- Timer app, 108
- Timer Glassware, 87
- timestamps
 - Locations.timestamp, 235
 - on timeline cards, 169
 - setting timestamp field on cards in GDK apps, 271
- Tinder app, 71
- Titanium Collection of custom-built frames, 39
- TITLE layout, combining icon, images, and text in, 279
- Tits & Glass, 333–336
- TOGGLE PINNED menu item, 206
- tools for Glass design, 80
- touchpad
 - manipulating timeline with, 21
 - on Glass, 18
 - touchpad API in GDK, 259
- Translate this (voice command), 26
- trigger phrases, selecting, 113
- turn-by-turn directions, 258
- Twitter, 146
- Twitter Glassware, 53, 100
 - adding captions to shared resources, 229
 - effective redesign of more complex web UI, 56
 - multiple custom menu items, 54
 - tracking state changes in menu items, 54
 - Web-controlled curation of user’s social stream, 56

U

- UbiTech NYC, 100
- UI
 - Glass, 12
 - minimalism in design and data brevity with high impact, 80
 - prefab templates and styles for, 109
 - wearer’s environment as part of, 99
- Umano built Glassware, 82

- unexpected, avoiding, 90
- UNPIN action, 207
- updates for Glassware, 287
- user configuration settings, 83
- user IDs, 148, 204
- userActions.payload, 212
- userActions.type, 207, 231
- userinfo.email scope, 148
- userinfo.profile scope, 148
- userToken, 204, 207

V

- values property (menu items), 202
- verifyToken, 204, 207
- videos
 - challenges for Glass, 104
 - first-person, 28
 - inability to share via Twitter Glassware, 56
 - recording, 22
 - voice command for, 25
 - streamed, on-demand, 13
- Viewfinder application, 22
- views, 268–284
 - CardBuilder object instantiated directly as, 269
 - custom layouts, 283
 - managing with CardScrollAdapter, 280
 - supported by live cards, 251
 - View object, 269
- ViewTube for Glass, 252
- Vignette Postcards, 64
- vignettes, 64
- virtual pet for Glass, 73
- Vodo Glassware, 113
 - real-time collaboration, 342
- VOICE CALL menu item, 216
- voice commands, 14, 23
 - appropriate use in user’s environment, 63
 - as of Explorer Edition, 24
 - configuring in GDK, 284
 - conflicts in, 286
 - contacts added by calling Contacts.insert, 222
 - default, deviation from, 112
 - for share contacts, 230–232
 - getting directions, 25
 - Google’s control over, 287
 - languages and, 238
 - launching Glassware or services, 26

- matching across other platforms, 286
- official list of, 286
- searches, 68
- sending messages, 25
- taking pictures or videos, 25
- voice controls, 22
- voice print analysis, 36
- voice triggers, supported, 112
- VoiceTriggers.Command enumeration, 286

W

- wearable computing, 4
 - with Glass, 29
- wearable computing devices
 - greater design goal of, 71
 - interactivity, 199
 - tactical wearable design, 80
- WearScript, 267, 352
- weather monitoring applications, 88
- web browser (Glass), 260
- web services, OAuth2 for authorization, 138
- web stack, using to produce native apps on Glass, 267

- web-based configuration, 83
- Windows systems, Glass drivers for, 266
- Wink feature, using to take pictures, 25
- Winkfeed, 85, 146
 - footer information, 170
- winking (input mechanism), 14
- Winky, 243
- workout software for wearables, LynxFit, 328

X

- XML file, voice triggers defined in, 284

Y

- YouTube
 - Glassware, 79
 - ViewTube for Glass, 252

Z

- Zynga video games, 99

Colophon

The animals on the cover of *Designing and Developing for Google Glass* are red-billed blue magpies (*Urocissa erythrorhyncha*). These birds live around the Indian subcontinent and Southeast Asia, including in the lower-altitude western Himalayas. Red-billed blue magpies can dwell in subtropical, moist, or human-modified environments (such as farmland), and their range includes forest, scrubland, and hills.

The red-billed blue magpie has the longest tail of any member of the crow family *Corvidae*. Its body is a little over two feet long; the tail is over half of the entire length (it takes up 17 out of 26 inches, on average). Its head, neck, and chest are black with blue spots, while the lower body is off-white (there are no bonus points for guessing its beak color). The legs, feet, and a ring around the eye also tend to be a vivid orange-red, but some birds have yellow feet and legs. The long, bright blue tail has a broad white tip, and is a striking feature of this bird. The red-billed blue magpie's diet is made up of small animals, especially invertebrates, which are then supplemented by fruits, seeds, and nectar.

This bird's cries are varied and unique, and they are known to be excellent mimickers of other species' calls. Red-billed blue magpies make shallow nests in trees and shrubs and go out to hunt for food in packs. After reproduction, the female magpie incubates three to five eggs on her own while the male provides food. In the wild, red-billed blue magpies have even been observed cleaning the teeth of Malayan tapirs, which provides the birds with an easy meal of bugs and uneaten vegetation.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to animals.oreilly.com.

The cover image is from Lydekker's *Royal Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.