acmqueue

The Case Against Data Lock-in

Want to keep your users? Just make it easy for them to leave.

Brian W. Fitzpatrick and JJ Lueck, The Data Liberation Front

Engineers employ many different tactics to focus on the user when writing software: for example, listening to user feedback, fixing bugs, and adding features that their users are clamoring for. Since Web-based services have made it easier for users to move to new applications, it's becoming even more important to focus on building and retaining user trust. We've found that an incredibly effective—although certainly counterintuitive—way to earn and maintain user trust is to make it easy for users to leave your product with their data in tow. This not only prevents lock-in and engenders trust, but also forces your team to innovate and compete on technical merit. We call this *data liberation*.

THE PROBLEM OF LOCK-IN

Until recently, users rarely asked whether they could quickly and easily get their data out before they put reams of personal information into a new Internet service. They were more likely to ask questions such as: "Are my friends using the service?" "How reliable is it?" and "What are the odds that the company providing the service is going to be around in six months or a year?" Users are starting to realize, however, that as they store more and more of their personal data in services that are not physically accessible, they run the risk of losing vast swaths of their online legacy if they don't have a means of removing their data.

It's typically a lot easier for software engineers to pull data out of a service that they use than it is for regular users. If APIs are available, we engineers can cobble together a program to pull our data out. Without APIs, we can even whip up a screen scraper to get a copy of the data. Unfortunately, for most users this is not an option, and they're often left wondering if they can get their data out at all.

Locking your users in, of course, has the advantage of making it harder for them to leave you for a competitor. Likewise, if your competitors lock their users in, it is harder for those users to move to your product. Nonetheless, it is far preferable to spend your engineering effort on innovation than it is to build bigger walls and stronger doors that prevent users from leaving. Making it easier for users to experiment today greatly increases their trust in you, and they are more likely to return to your product line tomorrow.

A SENSE OF URGENCY

Locking users in may suppress a company's need to innovate as rapidly as possible. Instead, your company may decide—for business reasons—to slow down development on your product and move engineering resources to another product. This makes your product vulnerable to other companies that innovate at a faster rate. Lock-in allows your company to have the appearance of continuing success when, without innovation, it may in fact be withering on the vine.

If you don't—or can't—lock your users in, the best way to compete is to innovate at a breakneck pace. Let's use Google Search as an example. It's a product that *cannot* lock users in: users don't have to install software to use it; they don't have to upload data to use it; they don't have to sign two-year contracts; and if they decide to try another search engine, they merely type it into their browser's location bar, and they're off and running.

How has Google managed to get users to keep coming back to its search engine? By focusing obsessively on constantly improving the quality of its results. The fact that it's so easy for users to switch has instilled an incredible sense of urgency in Google's search quality and ranking teams. At Google we think that if we make it easy for users to leave any of our products, failure to improve a product results in immediate feedback to the engineers, who respond by building a better product.

WHAT DATA LIBERATION LOOKS LIKE

At Google, our attitude has always been that users should be able to control the data they store in any of our products, and that means that they should be able to get their data *out* of any product. Period. There should be no additional monetary cost to do so, and perhaps most importantly, the amount of effort required to get the data out should be constant, regardless of the amount of data. Individually downloading a dozen photos is no big inconvenience, but what if a user had to download 5,000 photos, one at a time, to get them out of an application? That could take weeks of their time.

Even if users have a copy of their data, it can still be locked in if it's in a proprietary format. Some word processor documents from 15 years ago cannot be opened with modern software because they're stored in a proprietary format. It's important, therefore, not only to have access to data, but also to have it in a format that has a publicly available specification. Furthermore, the specification must have reasonable license terms: for example, it should be royalty-free to implement. If an open format already exists for the exported data (for example, JPEG or TIFF for photos), then that should be an option for bulk download. If there's no industry standard for the data in a product (e.g., blogs do not have a standard data format), then at the very least the format should be publicly documented—bonus points if your product provides an open source reference implementation of a parser for your format.

The point is that users should be in control of their data, which means they need an easy way of accessing it. Providing an API or the ability to download 5,000 photos one at a time doesn't exactly make it easy for your average user to move data in or out of a product. From the user-interface point of view, users should see data liberation merely as a set of buttons for import and export of all data in a product.

Google is addressing this problem through its Data Liberation Front, an engineering team whose goal is to make it easier to move data in and out of Google products. The data liberation effort focuses specifically on data that could hinder users from switching to another service or competing product—that is, data that users create in or import into Google products. This is all data stored

intentionally via a direct action—such as photos, e-mail, documents, or ad campaigns—that users would most likely need a copy of if they wanted to take their business elsewhere. Data indirectly created as a side effect (e.g., log data) falls outside of this mission, as it isn't particularly relevant to lock-in.

Another "non-goal" of data liberation is to develop new standards: we allow users to export in existing formats where we can, as in Google Docs where users can download word processing files in OpenOffice or Microsoft Office formats. For products where there's no obvious open format that can contain all of the information necessary, we provide something easily machine readable such as XML (e.g., for Blogger feeds, including posts and comments, we use Atom), publicly document the format, and, where possible, provide a reference implementation of a parser for the format (see the Google Blog Converters AppEngine project for an example[1]). We try to give the data to the user in a format that makes it easy to import into another product. Since Google Docs deals with word processing documents and spreadsheets that predate the rise of the open Web, we provide a few different formats for export; in most products, however, we assiduously avoid the rat hole of exporting into every known format under the sun.

THE USER'S VIEW

There are several scenarios where users might want to get a copy of their data from your product: they may have found another product that better suits their needs and they want to bring their data into the new product; you've announced that you're going to stop supporting the product that they're using; or, worse, you may have done something to lose their trust.

Of course, just because your users want a copy of their data doesn't necessarily mean that they're abandoning your product. Many users just feel safer having a local copy of their data as a backup. We saw this happen when we first liberated Blogger: many users started exporting their blogs every week while continuing to host and write in Blogger. This last scenario is more rooted in emotion than logic. Most data that users have on their computers isn't backed up at all, whereas hosted applications almost always store multiple copies of user data in multiple geographic locations, accounting for hardware failure in addition to natural disasters. Whether users' concerns are logical or emotional, they need to feel that their data is safe: it's important that your users trust you.

CASE STUDY: GOOGLE SITES

Google Sites is a Web-site creator that allows WYSIWYG editing through the browser. We use this service inside of Google for our main project page, as it's really convenient for creating or aggregating project documentation. We took on the job of creating the import and export capabilities for Sites in early 2009.

Early in the design, we had to determine what the external format of a Google Site should be. Considering that the utility Sites provides is the ability to create and collaborate on Web sites, we decided that the format best suited for true liberation would be XHTML. HTML being the language of the Web also makes it the most portable format for a Web site: just drop the XHTML pages on your own Web server or upload them to your Web service provider. We wanted to make sure that this form of data portability was as easy as possible with as little loss of data as possible.

Sites uses its internal data format to encapsulate the data stored in a Web site, including all revisions to all pages in the site. The first step to liberating this data was to create a Google Data API. A full export of a site is then provided through an open source Java client tool that uses the Google Sites Data API and transforms the data into a set of XHTML pages.

The Google Sites Data API, like all Google Data APIs, is built upon the AtomPub specification. This allows for RPC (remote procedure call)-style programmatic access to Google Sites data using Atom documents as the wire format for the RPCs. Atom works well for the Google Sites use case, as the data fits fairly easily into an Atom envelope.

Here is a sample of one Atom entry that encapsulates a Web page within Sites. This can be retrieved by using the Content Feed to Google Sites.

```
<entry xmlns:sites="http://schemas.google.com/sites/2008">
  <id>https://sites.google.com/feeds/content/site/...</id>
  <updated>2009-02-09T21:46:14.991Z</updated>
  <category scheme="http://schemas.google.com/g/2005#kind"
        term="http://schemas.google.com/sites/2008#webpage"
        label="webpage"/>
  <title>Maps API Examples</title>
  <sites:revision>2</sites:revision>
  <content type="xhtml">
    <div xmlns="http://www.w3.org/1999/xhtml">
    ... PAGE CONTENT HERE ...
    </div>
  </content>
</entry>
```

We've highlighted (in bold type) the actual data that is being exported, which includes an identifier, a last update time in ISO 8601 format, title, revision number, and the actual Web-page content. Mandatory authorship elements and other optional information included in the entry have been removed to keep the example short.

Once the API was in place, the second step was to implement the transformation from a set of Atom feeds into a collection of portable XHTML Web pages. To protect against losing any data from the original Atom, we chose to embed all of the metadata about each Atom entry right into the transformed XHTML. Not having this metadata in the transformed pages poses a problem during an import—it becomes unclear which elements of XHTML correspond to the pieces of the original Atom entry. Luckily, we didn't have to invent our own metadata embedding technique; we simply used the hAtom microformat.

To demonstrate the utility of microformats, here is the same sample after being converted into XHTML with hAtom microformat embedded:

```
<div class="hentry webpage"
    id="https://sites.google.com/feeds/content/site/...">
  <h3>
    <span class="entry-title">Maps API Examples</span>
  </h3>
  <div>
    <div class="entry-content">
      <div xmlns="http://www.w3.org/1999/xhtml">
      ... PAGE CONTENT HERE ...
      </div>
    </div>
  </div>
  <small>
    Updated on
    <abbr class="updated" title="2009-02-09T21:46:14.991Z">
      Feb 9, 2009
     </abbr>
    (Version <span class="sites:revision">2</span>)
  </small>
</div>
```

The highlighted class attributes map directly to the original Atom elements, making it very explicit how to reconstruct the original Atom when importing this information back into Sites. The microformat approach also has the side benefit that any Web page can be imported into Sites if the author is willing to add a few class attributes to data within the page. This ability to reimport a user's exported data in a lossless manner is key to data liberation—it may take more time to implement, but we think the result is worthwhile.

CASE STUDY: BLOGGER

One of the problems that we often encounter when doing a liberation project is catering to the power user. These are our favorite users. They are the ones who love to use the service, put a lot of data into it, and want the comfort of being able to do very large imports or exports of data at any time. Five years of journalism through blog posts and photos, for example, can easily extend beyond a few gigabytes of information, and attempting to move that data in one fell swoop is a real challenge. In an effort to make import and export as simple as possible for users, we decided to implement a one-click solution that would provide the user with a Blogger export file that contains all of the posts, comments, static pages, and even settings for any Blogger blog. This file is downloaded to the user's hard drive and can be imported back into Blogger later or transformed and moved to another blogging service.

One mistake that we made when creating the import/export experience for Blogger was relying on one HTTP transaction for an import or an export. HTTP connections become fragile when the size of the data that you're transferring becomes large. Any interruption in that connection voids the action and can lead to incomplete exports or missing data upon import. These are extremely frustrating scenarios for users and, unfortunately, much more prevalent for power users with lots of blog data. We neglected to implement any form of partial export as well, which means that power users sometimes need to resort to silly things such as breaking up their export files by hand in order to have better success when importing. We recognize that this is a bad experience for users and are hoping to address it in a future version of Blogger.

A better approach, one taken by rival blogging platforms, is not to rely on the user's hard drive to serve as the intermediary when attempting to migrate lots of data between cloud-based Blogging services. Instead, data *liberation* is best provided through APIs, and data *portability* is best provided by building code using those APIs to perform cloud-to-cloud migration. These types of migrations require multiple RPCs between services to move the data piece by piece, and each of these RPCs can be retried upon failure automatically without user intervention. It's a much better model than the one-transaction import. It increases the likelihood of total success and is an all-around better experience for the user. True cloud-to-cloud portability, however, works only when each cloud provides a liberated API for all of the user's data. We think that cloud-to-cloud portability is really good for users, and it's a tenet of the Data Liberation Front.

CHALLENGES

As you've seen from these case studies, the first step on the road to data liberation is to decide exactly what users need to export. Once you've covered data that users have imported or created by themselves into your product, it starts to get complicated. Take Google Docs, for example: a user clearly owns a document that he or she created, but what about a document that belongs to another user, then is edited by the user currently doing the export? What about documents to which the user has only read access? The set of documents the user has read access to may be considerably larger than the set of documents that the user has actually read or opened if you take into account globally readable documents. Lastly, you have to take into account document metadata such as access control lists. This is just one example, but it applies to any product that lets users share or collaborate on data.

Another important challenge to keep in mind involves security and authentication. When you're making it very easy and fast for users to pull their data out of a product, you drastically reduce the time required for an attacker to make off with a copy of *all* your data. This is why it's a good idea to require users to re-authenticate before exporting sensitive data (such as their search history), as well as over-communicating export activity back to the user (e.g., e-mail notification that an export has occurred). We're exploring these mechanisms and more as we continue liberating products.

Large data sets pose another challenge. An extensive photo collection, for example, which can easily scale into multiple gigabytes, can pose difficulties with delivery given the current transfer speeds of most home Internet connections. In this case, either we have a client for the product that can sync data to and from the service (such as Picasa), or we rely on established protocols and APIs (e.g., POP and IMAP for Gmail) to allow users to sync incrementally or export their data.

CONCLUSION

Allowing users to get a copy of their data is just the first step on the road to data liberation: we have a long way to go to get to the point where users can easily move their data from one product on the Internet to another. We look forward to this future, where we as engineers can focus less on schlepping data around and more on building interesting products that can compete on their technical merits—not by holding users hostage. Giving users control over their data is an important part of establishing user trust, and we hope that more companies will see that if they want to retain their users for the long term, the best way to do that is by setting them free.Q

REFERENCE

1. http://code.google.com/p/google-blog-converters-appengine/wiki/BloggerExportTemplate; and http://code.google.com/apis/blogger/docs/2.0/reference.html#LinkCommentsToPosts.

ACKNOWLEDGMENTS

Thanks to Bryan O'Sullivan, Ben Collins-Sussman, Danny Berlin, Josh Bloch, Stuart Feldman, and Ben Laurie for reading drafts of this article.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

BRIAN FITZPATRICK started Google's Chicago engineering office in 2005 and is the engineering manager for the Data Liberation Front and the Google Affiliate Network. A published author, frequent speaker, and open source contributor for more than 12 years, Fitzpatrick is a member of both the Apache Software Foundation and the Open Web Foundation, as well as a former engineer at Apple and CollabNet. Despite growing up in New Orleans and working for Silicon Valley companies for most of his career, he decided years ago that Chicago was his home and stubbornly refuses to move to California.

JJ LUECK joined the software engineering party at Google in 2007. An MIT graduate and prior engineer at AOL, Bose, and startups Bang Networks and Reactivity, he enjoys thinking about problems such as cloud-to-cloud interoperability and exploring the depths and potentials of virtual machines. He's also way too into Monty Python.