Developers Like Hypermedia, But They Don't Like Web Browsers

Leonard Richardson
Canonical USA
leonardr@segfault.org

ABSTRACT
Although desktop developers often have trouble consciously understanding RESTful concepts like "hypermedia as the engine of application state", this does not prevent them from intuitively understanding client-side tools based on these concepts. However, I encountered unexpected developer resistance after implementing a security protocol I and other web developers had thought uncontroversial: the most common mechanism for authorizing OAuth request tokens. This developer resistance has implications for many web services that share their authentication credentials with a corresponding website.

Categories and Subject Descriptors
H.3.5 [Information Storage and Retrieval]: Online Information Services—Web-based services; H.5.4 [Information Interfaces and Presentation]: Hypertext/hypermedia.

General Terms
Human Factors

Keywords
REST, hypermedia, OAuth, OpenID, developer relations

1. INTRODUCTION
I am the lead developer of `lazr.restful`, a Python library for publishing RESTful web services in a Zope environment. The biggest `lazr.restful` site is Launchpad,[1] which hosts collaborative development for the Ubuntu Linux distribution, many of Ubuntu's component packages, and many unrelated open source software projects.

In late 2008 I told three stories[9] recounting my advocacy of RESTful design in the face of my colleagues' skepticism. A year later, I present two stories about everyday usage: what happens when outside developers start using a RESTful web service. The first story is about being proved right by your users; the second about what happens when the users rebel.

2. EVERYBODY LOVES HYPERMEDIA
By general consensus, the most difficult RESTful constraint to grasp is "hypermedia as the engine of application state."[13] People who have trouble understanding HATEOAS in the context of web services understand it perfectly well in their everyday use of computers. Web browsers are based on HATEOAS. Ordinary computer users use an algorithm like this to accomplish something on a website: (I've translated the algorithm into RESTful terms.)

1. Retrieve a hypermedia representation of the home page.
2. Decode the representation to determine the current resource state.
3. Based on the representation's semantic cues, decide which hypermedia link or form is likely to bring you closer to your goal.
4. Click the link or fill out the form. Your browser will make another HTTP request and the result will be another hypermedia representation.
5. Go back to step 2 and repeat until the resource state is to your liking.

Although developers understand how the web works as well as non-developers, I've noticed two points of resistance when translating this algorithm into the world of web services. The first is in step 1, where the client starts at the well-known URI of the home page. Many developers prefer to use predefined rules to construct the URI of the object they "really" want to access, and go directly there. A simple, well-known example is the web service for the social bookmarking website del.icio.us, which describes its web service in a human-readable document, listing a number of URI "endpoints", each with a distinct function.[14] In violation of the HATEOAS principle, these useful URIs are nowhere to be found within the web service itself.

The second point of resistance is in step 3, with the idea that the state of a resource includes meta-information about its capabilities and its relationships to other objects. Some developers of web services prefer to keep this information (especially the information about capabilities) in human-readable form, and regard machine-readable hypermedia depictions as redundant.

Resistance to the HATEOAS principle is implicit in the design of many web services, and when I began work on the Launchpad web service, this resistance took the form of pushback from my colleagues. My perspective was not dismissed—I'd been hired specifically for my web service expertise—but I got a clear message that the Launchpad team's focus was on producing results, not exploring arcane theories.

"Results" in this context meant the kind of user-friendly development tools generally associated with SOAP/WSDL services. On the server side, it meant an easy way for developers to publish their existing data models as a web service. On the client-side, it meant a Python client which makes web service access idiomatically similar to local object access.

The competing vision for a Launchpad Python client was a library hard-coded with information about the Launchpad web service's "endpoints". This is a common design for custom web service clients, but it has one big disadvantage: these clients are brittle. They are written or generated based on a particular set of assumptions about the structure of the web service, and when the service changes, it violates those assumptions and the clients stop working.

Because of this, open source web service clients like `pyamazon` and `pydelicious` (for Amazon's ECS and the del.icio.us web service, respectively) have undergone serial changes of ownership. The web service changes and the library breaks, but the original client developer no longer has any active projects that use the web service. Someone with a more pressing need takes the project over, and the cycle repeats—or else the project is abandoned.

> "Originally written by Mark Pilgrim, I took over maintenance of the project in January 2004. and am now looking for somebody else who would be interested in taking over the maintenance of the project." [4]

> "pydelicious broke on the last del.icio.us API update and I was unable to contact the author so I'm posting the repaired code here going forward." [8]

Rather than pedantically explain the value of HATEOAS to my colleagues, I suggested that we exploit the hypermedia constraint to quickly ship a library that wouldn't have this problem. I proposed a client library whose exact capabilities would be determined by hypermedia served by the server. This client would present a Python interface corresponding to whatever hypermedia it received, analogous to the way a web browser displays a graphical representation of whatever hypermedia it receives.

My proposal became `launchpadlib`, a library that presents Launchpad as a densely interconnected network of Python objects similar to that found in an ORM library.[2] This network of objects corresponds exactly to the densely hyperlinked network of representations available from the web service.

In `launchpadlib`, the simplest way to get from one object to another is to follow a Python object reference. In web service terms, this corresponds to following a link. Save operations become PUT or PATCH requests, just as save operations in an ORM become database commands. Delete operations become DELETE requests. Here's some sample code:

```
>>> from launchpadlib import Launchpad
>>> service_root = Launchpad.login_with(
... "my application",
... "https://api.launchpad.net/beta/")
>>> my_account = service_root.me
>>> print my_account.name
Leonard Richardson
>>> my_account.name = "L. Richardson"
>>> my_account.lp_save()
```

[2] The `launchpadlib` library is actually a thin Launchpad-specific wrapper around a more generic client library, `lazr.restfulclient`.

Once `launchpadlib` was released I made an interesting discovery: a developer may have blind spots about the concept "hypermedia as the engine of application state", but they will use `launchpadlib` as if they did not have those blind spots. When developers ask me for help and send me code snippets, I see them using hypermedia as the engine of application state.

It would be hubristic to claim that `launchpadlib` is as easy to use as a web browser, but it's the same kind of tool as a web browser: a client programmed by hypermedia documents received from the server, presenting a number of possible next steps based on that hypermedia, each next step representing a change to the application state.

Whence this ease of use? Well, every `launchpadlib` session begins by constructing a "client" object. But this object doesn't just handle authentication and network details. It retrieves a hypermedia representation of the service's "home page". This automatically pushes the developer past step one of the HATEOAS algorithm, and past the first blind spot.

What about the second point of resistance, the reluctance to follow a link? The `launchpadlib` 'client' object offers a set of 'next steps' derived from the hypermedia representation. It's easy for a developer to load up a 'client' object in an interactive Python session and explore those 'next steps' by using the `dir()` command and following object references. This is the simplest way to explore the web service: it effectively turns `launchpadlib` into a web browser, allowing for "surfing".

The hypermedia algorithm is recursive, and once the developer follows one link, they might as well keep following links until they find what they're looking for. Once they're done exploring, the developer doesn't have to write any new Python code—they just have to clean up the code they wrote while exploring in the interactive session.

Launchpad's URIs do follow certain patterns: the URI path designating a 'person' resource is always "/~{username}". It's possible to craft a URL and load the representation of that resource directly into Launchpad, bypassing the normal workings of hypermedia. This is the equivalent of hacking the URI in your browser's address bar, and it's something a developer does a lot when they have a blind spot in step one of the HATEOAS algorithm. A web service that does not use hypermedia, like the del.icio.us service, assumes that a developers will write code to craft every URI their client accesses. When using the Launchpad web services, some developers do craft URIs for performance reasons, but I don't see it very often: it's easier to follow links from the 'home page'.

The very complexity of the Launchpad web service makes the hypermedia-based "surfing" style the more attractive option. If the

Launchpad web service only had a few kinds of resources, then a stripped-down, endpoint-based web service like the del.icio.us web service would be comprehensible. But actually the Launchpad web service has over sixty kinds of resources. Hypermedia is the best way to represent that diversity: it hides the parts you're not interested in behind links you didn't click. And it turns out developers love this style—as long as you don't tell them that the secret ingredient is "hypermedia as the engine of application state".

3. THE OAUTH REVOLT

Our developer-users ratified with their behavior our decision to write a hypermedia-based service and client. But an influential minority of our users rebelled against another one of our decisions, simply refusing to use the system as designed. I brokered a compromise which seemed promising enough to make it into an early draft of this paper, but which turned out to be untenable in the long term—untenable, it turns out, because of the way the World Wide Web embodies the HATEOAS constraint.

The Launchpad web service protects its resources with OAuth authentication[2], a request signing mechanism that depends on a 'access token' shared between client and server. An OAuth client like `launchpadlib` can obtain an access token just by asking the server, but an access token is useless until the end-user authorizes it, explicitly delegating some of their human authority to a computer program.

The OAuth standard does not define how the end-user is supposed to authorize an access token. We defined a protocol similar to the one used by other OAuth implementers like Twitter and Google, and similar to proto-OAuth mechanisms defined by providers like Flickr. [7, 1, 15] In our protocol, the `launchpadlib` application hands control over to the end-user's web browser and opens a web page on www.launchpad.net.

The web page explains to the end-user that an application 'foo' (the application that's using `launchpadlib`) wants access to their Launchpad account. The end-user may deny this request for access, may grant the application full access, or may grant limited access (like access to public data only). Once the end-user makes their decision, the access token is authorized (or revoked) and `launchpadlib` can begin using the Launchpad web service on the end-user's behalf (or not).

Here's the rationale for handing control over to the web browser: the end-user is in a tricky security situation. They're about to grant a third-party application access to their Launchpad account. We need to make it easy to distinguish between a legitimate delegation of authority and a phishing attack—an attempt to fraudulently obtain credentials by taking advantage of .people's natural tendency to give computers whatever information they ask for.

In my opinion, the best way to maintain the end-user's trust is to handle the authentication from their web browser. The browser is a trusted client. You already trust it with your passwords, and you trust your address bar not to lie about which server a web page came from. When the OAuth authentication process uses the web browser, the end-user can bring to bear all their experience in detecting web-based phishing attempts.

Launchpad bug #387297[11] summarizes what happened next. Several developers who were using `launchpadlib` in their third-party applications did not like this system, and routed around it. First they performed experiments, sniffing the HTTP interactions between `launchpadlib`, their web browser, and the Launchpad website. Then they wrote their own programs that asked for the end-user's username and password directly, and used screen-scraping and canned HTTP requests to simulate the browser-based authorization protocol we'd designed.

These developers weren't stupid. They knew how the system worked—they had to understand it in order to simulate it. They just didn't see the point. Stephan Hermann, a developer of a Launchpad desktop client called Leonov, wrote this about `launchpadlib`:[3]

> [T]he login and approval of `Launchpadlib` was a bit "strange" at the time when I looked at lplib. So I went and wrote a little wrapper class, which does the authentication and authorization (approval) automatically, without the need of a browser or interactive methods.

From Hermann's point of view, the API provider (my colleagues and me) did something "strange". Going along with our odd design would have inconvenienced his users. So he wrote a wrapper class that isolated the strange behavior. This let him tightly integrate Launchpad authorization into his native UI instead of jumping through bizarre browser-based hoops. The problem is, we designed the system specifically to prevent what Hermann wants to do.

Hermann isn't the only `launchpadlib` developer to rebel against our design. At least one other desktop application, Ground Control, uses a similar hack, and the ubuntu-dev-tools Ubuntu package includes a reusable script called `manage-credentials`.

The `manage-credentials` script takes a Launchpad username, password, and access level as command-line arguments. It logs in as the Launchpad user and grants itself a certain level of access. Here's the relevant code: [5]

```
# use hack
credentials =
    Credentials(options.consumer)
credentials =
    approve_application(credential,
      options.email,
      options.password,
      options.level,
      translate_api_web(options.service),
      None)
```

With this hack, the developer doesn't even have to let the end-user decide how much access they want to grant! Whoever calls `manage-credentials` can hard-code a certain value and get read-write access to the end-user's private data, without even telling the end-user there are other options.

Even this is not the worst of the desktop client depravity. While researching bug #387297, I heard of applications written before

Launchpad provided a web service, applications which crawled the end-user's Firefox profile looking for a saved Launchpad password. [M. Korn, personal communication] These were well-intentioned pieces of software designed to improve the end-user's interactions with Launchpad. But from an architectural standpoint, they were spyware.

Not all developers rebel against the browser-based security model. The F-Spot photo manager features integration with Flickr's web service, and rather than ask for your Flickr username and password, it tells you to click a button. Clicking the button opens up your web browser and begins Flickr's OAuth-like authentication protocol.

Similarly, many `launchpadlib` developers used `launchpadlib`'s browser-based authentication protocol without complaining (or at least without rebelling). But a prominent minority preferred to write hacks, to productize the hacks into scripts, and to include the scripts in official Ubuntu utility libraries.

I must admit that `launchpadlib` did not have the smoothest possible implementation of the credential-obtaining protocol (it's much better now). And our documentation doesn't shout out the security rationale for this protocol. It just says: "This lets your users delegate a portion of their Launchpad permissions to your program, without having to trust it completely."[10]

But even after reading an explanation of my point of view, Stephan Hermann preferred his original design, the one we tried to prohibit. In a comment on Launchpad bug #387297, he wrote:[11]

> Actually, I don't think there is a difference between trusting a webbrowser and an UI client... The approach with username + password is bad, but having no other chance to avoid a browser for ui clients, I think our leonov workaround is the best thing someone can do.

I interviewed Markus Korn, author of the `manage-credentials` script. He understands perfectly well how our OAuth protocol works; he just doesn't buy into the security rationale. When I asked him why he'd written `manage-credentials`, he told me: "The idea was to not bother the user with a web browser window when he is using a GUI." [M. Korn, personal communication]

I asked Korn what he would have said in his own defense if I'd confronted him while he was writing `manage-credentials`, telling him that he was subverting Launchpad's security model. He volunteered: "The user of my applications cares more about smoothness than security, because he trusts me as the developer of this app."[Korn, personal communication]

I also interviewed Martin Owens, the developer of the Ground Control desktop interface to Launchpad. He didn't think browser-based authentication was any more secure than a desktop application that asks the end-user for their Launchpad password: [M. Owens, personal communication]

> The web browser is a large application with arbitrary display and execution of code which comes from unknown and untrusted sources. It's got a very large attack area. It's got a fairly weak trust network... The user, I hope, would trust applications installed on their computer, especially if those applications are installed by default on the operating system CD...

It's not surprising that desktop developers put more trust in desktop applications than does a web developer like myself. If you install an application on your computer, you give that application as much implicit trust as you give your web browser. In a modern Linux environment, applications are typically open source and installed from trusted repositories, reducing the possibility that a given application will contain spyware.

Objectively speaking, Hermann, Korn, and Owens have the final say. The Launchpad web service was designed for toolmakers like them. Although hundreds of people use the Launchpad web service, they use it through applications like Leonov and Ground Control.

Because of this I decided to compromise with the toolmakers. I didn't like the idea of each desktop developer independently sniffing the token authorization protocol we'd designed for a web browser, and writing their own imitation browser to run through that protocol. Eventually one of those developers would make a mistake, leaking a Launchpad user's password or storing it in an insecure location. I also didn't like the way some of the imitation web browsers chose their own level of access to Launchpad, instead of leaving that decision up to the end-user.

My inspiration was `pinentry`, a suite of small desktop applications (part of the GnuPG project) which "read passphrases and PIN numbers in a secure manner." [6] The `pinentry` suite centralizes passphrase-gathering functionality in one simple, easily audited code base. I wrote a `pinentry`-like program, a canonical desktop application for taking the user's Launchpad password and authorizing an OAuth access token. Although this program duplicates the behavior of a web browser, I could at least keep every desktop developer from developing *their own* imitation browser.

I planned to package three different versions of this `pinentry`-like program with `launchpadlib`: one to blend in with GTK+ GUIs, one for Qt-based GUIs, and one for console applications. This would meet Korn's goal of "not bother[ing] the user with a web browser window." Instead of handing control over to the web browser, a developer would be able to hand control to a native desktop application that closely resembled their own desktop application.

I wrote a console-based application, `launchpad-credentials-console`, and was talking with desktop developers interested in writing GUI versions, when all my work was rendered obsolete by a change to the Launchpad website.

Up to this point, Launchpad was like most websites in having a special "login" page, which invited the end-user to type their username and password into an HTML form. My `launchpad-credentials-console` authenticated with Launchpad by constructing an HTML form submission and POSTing it to the appropriate Launchpad URL. Leonov, Ground Control, and manage-credentials also authenticated with Launchpad using constructed form submissions.

In March 2010, the Launchpad login page disappeared. Launchpad users no longer give their username and password directly to Launchpad; they are now redirected to a OpenID provider, the Launchpad Login Service, and they authenticate with that OpenID provider. This is more convenient for the end-user, but it's disastrous for launchpad-credentials-console and all similar applications.

Consider a human being using their web browser to visit Launchpad, on the day after the old login page disappears. The home page still features an HTML link in the upper right-hand corner that says "Log In / Register". Clicking that link takes the end-user through a process in which they fill out HTML forms and submit them. At the end of the process, the end-user finds him or herself logged in to Launchpad.

On this day, the user's browser sends drastically different HTTP requests than the day before, but the differences between the old login procedure and the new one are encapsulated in hypermedia. When the end-user is using a hypermedia-aware client (ie. a web browser), the login system can change drastically and the user will not suffer anything worse than possible confusion. The "hypermedia algorithm" for obtaining a given application state still works.

Now consider a human being trying to run a launchpadlib script the day after the login procedure changes. At the crucial moment, when the end-user needs to authorize an OAuth request token, launchpadlib opens the end-user's web browser. The end-user is sent through the (new) login procedure and then gets a chance to authorize an OAuth request token or refuse authorization. Again, the change to the login procedure is encapsulated in hypermedia, which a web browser can always understand and a human being can always navigate.

Finally, consider someone trying to log in to Launchpad using launchpad-credentials-console. The end-user types their username and password into the console application, which sends a constructed HTML form submission. But Launchpad no longer recognizes that form submission! Launchpad is no longer in charge of handling login attempts; it can only redirect people to an OpenID provider. The launchpad-credentials-console program broke when the login procedure changed, in the same way pyamazon broke when Amazon's ECS service changed.

Leonov, Ground Control, manage-credentials, and launchpad-credentials-console all broke on the same day and for the same reason. These programs hid the workings of hypermedia ("arbitrary display... from unknown and untrusted sources") from the end-user, and now they're paying the price.

Can these applications be made to work again? In the short run, certainly. It's just HTTP. A developer can sniff the way browsers interact with the Launchpad Login Service, find the point at which the username and password are sent to the server, and teach a program to send the same request to the same URL.

In the long run, a hypermedia-oblivious program like launchpad-credentials-console cannot be made to work. The problem is not just that the login procedure might change again. We know the login procedure will change again, and it will change in a way that makes programs like launchpad-credentials-console impossible.

As of March 2010, Launchpad only accepts OpenID identifiers from one source: the Launchpad Login Service. In the future, Launchpad will be a full-fledged "relying party". End-users will be able to log in to Launchpad using an identifier from any OpenID provider: the Launchpad Login Service, the Ubuntu Single Sign On Service, Google, LiveJournal, MySpace, or any other.

When a user tries to log in to Launchpad, they will temporarily be redirected to their OpenID provider and asked to authenticate with their provider. Each provider has its own way of authenticating the end-user. Most OpenID providers use username-password combinations, as Launchpad used to and as the Launchpad Login Service does now, but each provider serves slightly different HTML forms and accepts slightly different HTTP requests. And nothing prevents an OpenID provider from authenticating with an x509 certificate, or defining an authentication procedure that makes the end-user solve a CAPTCHA or digitally sign a challenge string.

A web browser supports nearly any protocol for authorizing an OAuth access token. The differences between protocols are encapsulated in hypermedia and a human can navigate any protocol using the same, general hypermedia algorithm. Programming a hypermedia-oblivious client with all these possibilities is simply impossible. Once Launchpad's OAuth token authorization protocol allows authentication with an arbitrary OpenID provider, that authentication must take place in a web browser.

Desktop developers I've spoken with are understandably unhappy that their applications are broken. [Owens, personal communication.] But launchpadlib's normal browser-based mechanism for authorizing OAuth tokens still works. The Launchpad team is working with Ubuntu desktop developers on a desktop-wide solution that should reduce the number of times an end-user has to use their web browser, but because it's very early in development, this paper is not a good place to discuss it.[12]

4. CONCLUSIONS AND RECOMMENDATIONS

When I designed the Launchpad web service, I expected one of my tasks would be developer education. After all, when designing the service I'd had to convince my colleagues of the benefits of RESTful design. But thanks to a hypermedia-aware client-side library, I found little conceptual resistance from developers. Our experience with web browsers shows that you don't have to understand "hypermedia as the engine of application state" to take advantage of it.

I did experience developer resistance to HATEOAS when it came to hypermedia experienced through the web browser. There, the problem wasn't the hypermedia; it was the browser. Although relatively few web services protect their resources with OAuth, I predict that any web service that does will find its developers writing libraries along the lines of `manage-credentials`.

I propose a natural experiment: as I write, a client for the Twitter web service can authenticate its requests using an OAuth token, or by providing a Twitter username and password with HTTP Basic Auth. Twitter developers plan to deprecate Basic Auth starting in June 2010. [7] I predict that as Basic Auth is deprecated, client-side Twitter hackers will resist Twitter's OAuth token authorization protocol, just as client-side Launchpad hackers resisted Launchpad's similar protocol. How the Twitter developers will react to this resistance is an open question—especially if they ever intend to make Twitter an OpenID relying party.

It was frustrating to see `launchpad-credentials-console` suddenly break, along with all the other hypermedia-oblivious ways of authorizing Launchpad's OAuth tokens, but it also provided an object lesson in the value of hypermedia-aware clients. Regardless of desktop developers' reservations about the web browser, it's the only client that can authenticate with an arbitrary OpenID provider. More seriously, I think this problem illustrates a general obstacle towards OpenID adoption on websites that also provide web services.

Consider an alternate universe in which, by the beginning of 2010, Launchpad's web service somehow became as popular as Twitter's. Instead of four desktop clients that simulate a web browser to authenticate Launchpad's OAuth tokens, there would be dozens. Instead of an audience of a few hundred software developers, our web service would be used by millions of ordinary people.

What happens at this point if we decide to make Launchpad an OpenID relying party? We can't break dozens of clients and confuse millions of people. But the existing, hypermedia-oblivious clients won't allow a user to authenticate using an OpenID identity from (eg.) MySpace. Such a user would have to get a Launchpad account to use a web service client, defeating the purpose of making Launchpad an OpenID relying party.

When we saw that `launchpad-credentials-console` was broken, the Launchpad team had an internal discussion: should we fix the hypermedia-oblivious clients and forget about making Launchpad an OpenID relying party, or should we go ahead with our OpenID plans and abandon `launchpad-credentials-console?` This decision would have affected the entire Launchpad website, not just the web service.

For the sake of being good OpenID citizens, we decided to abandon the hypermedia-oblivious clients. The more popular a web service is, and the more hypermedia-oblivious clients there are in active use, the more difficult it will be to decide to make the corresponding website an OpenID relying party.

If your web service users authenticate using the same credentials they use on some corresponding website, give some thought to that site's future. If you protect your web service's resources with OAuth, you should decide now whether you ever want that site to be an OpenID relying party. If you use HTTP Basic Auth or some other authentication mechanism, make the same decision—and consider how your web service authentication mechanism might need to change.

5. ACKNOWLEDGMENTS

Thanks to Markus Korn and Martin Owens for explaining the desktop developer's perspective. Thanks also to my sister, Rachel Richardson, for formatting this paper.

6. REFERENCES

[1] Eric Bidelman
http://code.google.com/apis/gdata/articles/oauth.html

[2] E. Hammer-Lahav, Ed. "The OAuth 1.0 Protocol".
http://tools.ietf.org/html/draft-hammer-oauth-10

[3] S. Hermann, "Some internals...".
http://www.sourcecode.de/content/some-internals

[4] M. Josephson,
http://www.josephson.org/projects/pyamazon/

[5] M. Korn, "manage-credentials". Source code hosted in bzr repository. lp:ubuntu-dev-tools/manage-credentials.

[6] Werner Koch
http://www.gnupg.org/related_software/pinentry/index.en.html

[7] Raffi Krikorian et al.
http://apiwiki.twitter.com/OAuth-FAQ

[8] G. Pinero et al.
http://code.google.com/p/pydelicious/

[9] L. Richardson, "Justice Will Take Us Millions of Intricate Moves".
http://www.crummy.com/writing/speaking/2008-QCon/

[10] L. Richardson et al,
https://help.launchpad.net/API/launchpadlib

[11] L. Richardson, et al. "manage-credentials should not ask for Launchpad password directly".
https://bugs.edge.launchpad.net/launchpadlib/+bug/387297

[12] L. Richardson et al, "Trusted credential-management apps are broken and doomed",
https://bugs.launchpad.net/launchpadlib/+bug/532055

[13] J. Webber, "HATEOAS - The Confusing Bit from REST".
http://jim.webber.name/downloads/presentations/2009-05-HATEOAS.pdf

[14] Unknown author
http://delicious.com/help/api

[15] Unknown author
http://www.flickr.com/services/api/auth.spec.htm