Programmable Web - Aaron Swartz

<

Building Programmable Web Sites
Aaron Swartz, 2009

To Dan Connolly, who not only created the Web but found time to teach it to me

Introduction: A Programmable Web

If you are like most people I know (and, since you're reading this book, you probably are -- at least in this respect), you use the Web. A lot. In fact, in my own personal case, the vast majority of my days are spent reading or scanning web pages -- a scroll through my webmail client to talk with friends and colleagues, a weblog or two to catch up on the news of the day, a dozen short articles, a flotilla of Google queries, and the constant turn to Wikipedia for a stray fact to answer a nagging question.

All fine and good, of course; indeed, nigh indispensable. And yet, it is sobering to think that little over a decade ago none of this existed. Email had its own specialized applications, weblogs had yet to be invented, articles were found on paper, Google was yet unborn, and Wikipedia not even a distant twinkle in Larry Sanger's eye.

And so, it is striking to consider -- almost shocking, in fact -- what the world might be like when our software turns to the Web just as frequently and casually as we do. Today, of course, we can see the faint, future glimmers of such a world. There is software that phones home to find out if there's an update. There is software where part of its content -- the help pages, perhaps, or some kind of catalog -- is streamed over the Web. There is software that sends a copy of all your work to be stored on the Web. There is software specially designed to help you navigate a certain kind of web page. There is software that consists of _nothing but_ a certain kind of web page. There is software -- the so-called "mashups" -- that consists of a web page combining information from two other web pages. And there is software that, using "APIs", treats other web sites as just another part of the software infrastructure, another function it can call to get things done.

Our computers are so small and the Web so great and vast that this last scenario seems like part of an inescapable trend. Why _wouldn't_ you depend on other

web sites whenever you could, making their endless information and bountiful abilities a seamless part of yours? And so, I suspect, such uses will become increasingly common until, one day, your computer is as tethered to the Web as you yourself are now.

It is sometimes suggested that such a future is impossible, that making a Web that other computers could use is the fantasy of some (rather unimaginative, I would think) sci-fi novelist. That it would only happen in a world of lumbering robots and artificial intelligence and machines that follow you around, barking orders while intermittently unsuccessfully attempting to persuade you to purchase a new pair of shoes.

So it is perhaps unsurprising that one of the critics who has expressed something like this view, Cory Doctorow, is in fact a rather imaginative sci-fi novelist (amongst much else). Doctorow's complaint is expressed in his essay "Metacrap: Putting the torch to seven straw-men of the meta-utopia"[^mc].

[^mc]: Available online at . It is also reprinted in his book of essays _Content: Selected Essays on Technology, Creativity, Copyright, and the Future of the Future_ (2008, Tachyon Publications) which is likewise available online at .

Doctorow argues that any system collect accurate "metadata" -- the kind of machine-processable data that will be needed to make this dream of computers-using-the-Web come true -- will run into seven inescapable problems: people lie, people are lazy, people are stupid, people don't know themselves, schemas aren't neutral, metrics influence results, and there's more than one way to describe something. Instead, Doctorow proposes that instead of trying to get people to provide data, we should instead look at the data they produce incidentally while doing other things (like how Google looks at the links people make when they write web pages) and use that instead.

Doctorow is, of course, attacking a strawman. Utopian fantasies of honest, complete, unbiased data about everything are obviously impossible. But who was trying for that anyway? The Web is rarely perfectly honest, complete, and unbiased -- but it's still pretty damn useful. There's no reason making a Web for computers to use can't be the same way.

I have to say, however, the idea's proponents do not escape culpability for these utopian perceptions. Many of them have gone around talking about the "Semantic Web" in which our computers would finally be capable of "machine understanding". Such a framing (among other factors) has attracted refugees from the struggling world of artificial intelligence, who have taken it as another opportunity to promote their life's work.

Instead of the "let's just build something that works" attitude that made the Web (and the Internet) such a roaring success, they brought the formalizing mindset of mathematicians and the institutional structures of academics and defense contractors. They formed committees to form working groups to write drafts of ontologies that carefully listed (in 100-page Word documents) all possible things

in the universe and the various properties they could have, and they spent ours in Talmudic debates over whether a washing machine was a kitchen appliance or a household cleaning device.

With them has come academic research and government grants and corporate R&D and the whole apparatus of people and institutions that scream "pipedream". And instead of spending time building things, they've convinced people interested in these ideas that the first thing we need to do is write _standards_. (To engineers, this is absurd from the start -- standards are things you write _after_ you've got something working, not before!)

And so the "Semantic Web Activity" at the Worldwide Web Consortium (W3C) has spent its time writing standard upon standard: the Extensible Markup Language (XML), the Resource Description Framework (RDF), the Web Ontology Language (OWL), tools for Gleaning Resource Descriptions from Dialects of Languages (GRDDL), the Simple Protocol And RDF Query Language (SPARQL) (as created by the RDF Data Access Working Group (DAWG)).

Few have received any widespread use and those that have (XML) are uniformly scourges on the planet, offenses against hardworking programmers that have pushed out sensible formats (like JSON) in favor of overly-complicated hairballs with no basis in reality (I'm not done yet! -- more on this in chapter 5).

Instead of getting existing systems to talk to each other and writing up the best practices, these self-appointed guarantors of the Semantic Web have spent their time creating their own little universe, complete with Semantic Web databases and programming languages. But databases and programming languages, while far from perfect, are largely solved problems. People already have their favorites, which have been tested and hacked to work in all sorts of unusual environments, and folks are not particularly inclined to learn a new one, especially for no good reason. It's hard enough getting people to share data as it is, harder to get them to share it in a particular format, and completely impossible to get them to store it and manage it in a completely new system.

And yet this is what Semantic Webheads are spending their time on. It's as if to get people to use the Web, they started writing a new operating system that had the Web built-in right at the core. Sure, we might end up there someday, but insisting that people do that from the start would have doomed the Web to obscurity from the beginning.[^dt]

All of which has led "web engineers" (as this series' title so cutely calls them) to tune out and go back to doing real work, not wanting to waste their time with things that don't exist and, in all likelihood, never will. And it's led many who have been working on the Semantic Web, in the vain hope of actually building a world where software can communicate, to burnout and tune out and find more productive avenues for their attentions.

For an example, look at Sean B. Palmer. In his influential piece, "Ditching the Semantic Web?"[^dt], he proclaims "It's not prudent, perhaps even not moral (if

that doesn't sound too melodramatic), to work on RDF, OWL, SPARQL, RIF, the broken ideas of distributed trust, CWM, Tabulator, Dublin Core, FOAF, SIOC, and any of these kinds of things" and says not only will he "stop working on the Semantic Web" but "I will, moreover, actively dissuade anyone from working on the Semantic Web where it distracts them from working on" more practical projects.)

[^dt]: Available online at .

It would be only fair here to point out that I am not exactly an unbiased observer. For one thing, Sean, like just about everyone else I cite in the book, is a friend. We met through working on these things together but since have kept in touch and share emails about what we're working on and are just generally nice to each. And the same goes for almost all the other people I cite and criticize.

Moreover, the reason we were working together is that I too did my time in the Semantic Web salt mines. My first web application was a collaboratively-written encyclopedia, but my second aggregated news headlines from sites around the Web, leading me into a downward spiral that ended with many years spent on RDF Core Working Groups and an ultimate decision to get out of the world of computers altogether.

Obviously, that didn't work out quite as planned. Jim Hendler, another friend and one of the AI transplants I've just spend so much time taking a swing at, asked me if I'd write a bit on the subject to kick off a new series of electronic books he's putting together. I'll do just about anything for a little cash (just kidding; I just wanted to get published (just kidding; I've been published plenty of times times (just kidding; not that many times (just kidding; I've never been published (just kidding; I have, but I just wanted more practice (just kidding; I practice plenty (just kidding; I never practice (just kidding; I just wanted to publish a book (just kidding; I just wanted to _write_ a book (just kidding; it's easy to write a book (just kidding; it's a death march (just kidding; it's not so bad (just kidding; my girlfriend left me (just kidding; I left her (just kidding, just kidding, just kidding))))))))))))))) and so here I am again, rehashing all the old ground and finally getting my chance to complain about what a mistake all the Semantic Web folks have made.

Yet, as my little thought experiment above has hopefully made clear, the programmable web is anything but a pipe dream -- it is today's reality and tomorrow's banality. No software developer will remain content to limit themselves only to things on the user's own computer. And no web site developer will be content to limit their site only to users who act with it directly.

Just as the interlinking power of the World Wide Web sucked all available documents into its maw -- encouraging people to digitize them, convert them into HTML, give them a URL, and put them on the Internet (hell, as we speak Google is even doing this to _entire libraries_) -- the programmable Web will pull all applications within its grasp. The benefits that come from being connected are just too powerful to ultimately resist.

They will, of course, be grant challenges to business models -- as new technologies always our -- especially for those who make their money off of gating up and charging access to data. But such practices simply aren't tenable in the long term, legally or practically (let alone morally). Under US law, facts aren't copyrightable (thanks to the landmark Supreme Court decision in _Feist v. Rural Telephone Service_) and databases are just collections of facts. (Some European countries have special database rights, but such extensions have been fervently opposed in the US.)

But even if the law didn't get in the way, there's so much value in sharing data that most data providers will eventually come around. Sure, providing a website where people can look things up can be plenty valuable, but it's nothing compared to what you can do when you combine that information with others.

To take an example from my own career, look at the website OpenSecrets.org. It collects information about who's contributing money to US political candidates and displays nice charts and tables about the industries that have funded the campaigns of presidential candidates and members of Congress.

Similarly, the website Taxpayer.net provides a wealth of information about Congressional earmarks -- the funding requests that members of Congress slip into bills, requiring a couple million dollars be given to someone for a particular pet project. (The $398 million "Bridge to Nowhere" being the most famous example.)

Both are fantastic sites and are frequently used by observers of American politics, to good effect. But imagine how much better they would be if you put them together -- you could search for major campaign contributors who had received large earmarks.

Note that this isn't the kind of "mashup" that can be achieved with today's APIs. APIs only let you look at the data in a particular way, typically the way that the hosting site looks at it. So with OpenSecrets' API you can get a list of the top contributors to a candidate. But this isn't enough for the kind of question we're interested in -- you'd need to compare each earmark against each donor to see if they match. It requires real access to the data.

Note also that the end result is ultimately in everyone's best interest. OpenSecrets.org wants people to find out about the problematic influence of money in politics. Taxpayer.net wants to draw attention to this wasteful spending. The public wants to know how money in politics causes wasteful spending and a site that helps them do so would further each organization's goals. But they can only get there if they're willing to share their data.

Fortunately for us, the Web was designed with this future in mind. The protocols that underpin it are not designed simply to provide pages for human consumption, but also to easily accommodate the menagerie of spiders, bots, and scripts that explore its fertile soil. And the original developers of the Web, the men and women who invented the tools that made it the life-consuming pastime that it is

today, have long since turned their sights towards making the Web safe, even inviting, for applications.

Unfortunately, far too few are aware of this fact, leading many to reinvent -- sloppily -- the work that they have already done. (It hasn't helped that the few who are aware have spent their time working on the Semantic Web nonsense that I criticized above.) So we will begin by trying to understand the architecture of the Web -- what it got right and, occasionally, what it got wrong, but most importantly _why_ it is the way it is. We will learn how it allows both users and search engines to co-exist peacefully while supporting everything from photo-sharing to financial transactions.

We will continue by considering what it means to build a program on top of the Web -- how to write software that both fairly serves its immediate users as well as the developers who want to build on top of it. Too often, an API is bolted on top of an existing application, as an afterthought or a completely separate piece. But, as we'll see, when a web application is designed properly, APIs naturally grow out of it and require little effort to maintain.

Then we'll look into what it means for your application to be not just another tool for people and software to use, but part of the ecology -- a section of the programmable web. This means exposing your data to be queried and copied and integrated, even without explicit permission, into the larger software ecosystem, while protecting users' freedom.

Finally, we'll close with a discussion of that much-maligned phrase, "the Semantic Web", and try to understand what it would really mean.

Let's begin.

Building for users: designing URLs

From billboards, buses, and boxes they peer out at us like alien symbols (except hopefully less threatening): URLs are everywhere. Most obviously, they appear at the top of the browser window while people are using your website, but they also appear in a myriad of other contexts: in the status bar when someone has the mouse over a link, in search results, in emails, in blogs, read over the phone, written down on napkins, listed in bibliographies, printed on business cards and t-shirts and mousepads and bumper stickers. They're versatile little symbols.

Furthermore, URLs have to last.[^1] Those t-shirts and links and blogs will not disappear simply because you decided to reorganize your server, or move to a different operating system, or got promoted and replaced by a subordinate (or voted out of office). They will last for years and years to come, so your URLs must last with them.

Moreover, URLs do not just exist as isolated entities (like ‘http://example.org/lunch/bacon.html‘). They combine to form patterns (‘bacon.html‘, ‘lettuce.html‘, ‘tomato.html‘). And each of these patterns finds its place in a larger path of interaction (‘/‘, ‘/lunch/‘, ‘/lunch/bacon.html‘).

Because of all this, URLs cannot be some side-effect or afterthought, as many seem to wish. Designing URLs is the most important part of building a web application and has to be done first. Encoded in their design are a whole series of implicit assumptions about what your site is about, how it is structured, and how it should be used; all important and largely-unavoidable questions.

Unfortunately, many tools for building web applications try to hide such questions from you, preventing you from designing URLs at all. Instead, they present their own interface to the programmer, from which they generate URLs by making up random numbers or storing cookies or worse. (Nowadays, with Ajax and Flash, some don't provide URLs at all, making a site hell for anyone who wants to send something cool they found to their friends.)

And when people who use such software find themselves having to write a URL on a t-shirt or link to something from an email, they create a redirect -- a special URL who's only purpose is to introduce people to the nightmarish random-number system used by their actual website. This solves their immediate problem of figuring out what to write on the t-shirt, but it doesn't solve any of the more fundamental problems, and it doesn't make it possible for everyone else to make their own t-shirts or send out their own emails.

If your tools don't let you design your URLs, you need to get better tools. Nobody would dare do graphic design with software that didn't let them change the font or paint with a brush that could only make squares. Yet some people think it's perfectly fine to sacrifice control over their URLs, the most fundamentally important part of your website. It's not. Get better tools.[^to]

[^to]: If you need a place to start, there's of course my own toolkit, web.py (http://webpy.org/) as well as the Python web framework Django (http://djangoproject.com/).

Once you have decent tools, it's time to start designing. Let's start with the biggest constraints first. URLs shouldn't change (and if they do change, the old ones should redirect to the new ones) so they should only contain information about the page that never changes. This leads to some obvious requirements.

These were most famously pointed out by the Web's inventor, Sir Timothy John Berners-Lee OM KBE FRS FREng FRSA (b. 8 June 1955, London, England). During a miraculous Christmas break in 1990 that reminds one of Einstein's _annus mirabilis_, Tim not only invented the URL, the HTML format, and the HTTP protocol, but also wrote the first web browser, WYSIWYG web editor, and web server. (Makes you want to give the guy more Christmas breaks.) Although, in fact, this is slightly redundant, since the first web browser (named WorldWideWeb), not only let you read web pages, but let you write them as well. The idea was that the Web should be an interactive medium, with everybody keeping their own notebooks of interesting things they found and collaborating on documents with people and posting stuff they'd done or wanted to share.

Editing a web page was as easy as clicking on it -- you could just switch into

editing mode and select and correct typos right on the page, just like in a word processor. (You'd hit save and it would automatically upload them to the server.) You could create new pages just by opening a new window and instead of bookmarks, you were expected to build web pages keeping track of the sites you found interesting. (The original browser didn't have a URL bar, in part to force you to keep track of pages this way.)

It was a brilliant idea, but unfortunately it was written for the obscure NeXT operating system (which later became Mac OS X) and as a result few have ever gotten to use it. Instead, they used the clone created by a team at the University of Illinois Urbana-Champaign (UIUC), which never supported editing because programmer Marc Andreesen was too dumb to figure out how to do page editing with inline pictures, something Tim Berners-Lee's version had no problem with. Marc Andreesen made half a billion dollars as UIUC's browser became Netscape while Berners-Lee continued doing technical support for a team of physicists in Switzerland. (He later became a Research Scientist at MIT.)

Image: http://www.w3.org/History/1994/WWW/Journals/CACM/screensnap2_24c.gif

The result is that we're only reacquiring these marvelous features a couple decades later, through things like weblogs and Wikipedia. And even then, they're far more limited than the wide-reaching interactivity that Berners-Lee imagined.

But let's turn away from the past and back to the future. Sir Tim argued that to protect your URLs into the future, you needed to follow some basic principles. In his 1998 statement "Cool URIs don't change"[ˆco], described as "an attempt to redirect the energy behind the quest for coolness ... toward usefulness [and] longevity", he laid them out:

[ˆco]: Available at . However, I go on to disagree with Tim's proposed solution for generating Cool URIs. He recommends thoroughly date-based schemes, like ‘http://www.w3.org/1998/12/01/chairs‘. As far as I've noticed, only the W3C has really thoroughly adopted this strategy and when I've tried it, it's only led to ugliness and confusion.

(You may notice that Tim says URI, while I say URL. URL, the original term, stands for Uniform Resource Locator. It was developed, along with the Web, to provide a consistent way for referring to web pages and other Internet resources. Since then, however, it has been expanded to provide a way for referring to all sorts of things, many of which are not web pages, and some of which cannot even be "located" in any automated sense (e.g. abstract concepts like "Time magazine"). Thus the term was changed to URI, Uniform Resource Identifier, to encompass this wider set. I stick with the term URL here since it's more familiar, but we'll end up discussing abstract concepts in later chapters.)

First, URLs shouldn't include technical details of the software you used to build your website, since that could change at any moment. Thus things like ‘.php‘ and ‘.cgi‘ are straight out. For similar reasons, you can drop ‘.html‘, ‘PHP_SESS_ID‘ and their ilk. You'll also want to make sure that the names of your servers (e.g.

‘www7.example.org‘ or ‘plato.example.net‘) aren't seen in your URLs. Moving from one programming language, one format, or one server to another is fairly common; there's no reason your URLs should depend upon your current decision.

Second, you'll want to leave out any facts about the page that might change. This is just about everything (its author, its category, who can read it, whether it's official or a draft, etc.), so your URLs are really limited to just the essential concept of a page, the page's _essence_. What's the one thing that can't be changed, that makes this page this page?

Third, you'll want to be really careful about classification. Many people like to divide their websites up by topic, putting their very favorite recipes into the ‘/food/‘ directory and their stories about the trips they take into ‘/travel/‘ and the stuff they're reading into ‘/books/‘. But inevitably they end up having a recipe that requires a trip or a book that's about food and they think it belongs in both categories. Or they decide that drink should really be broken out and get its own section. Or they decide to just reorganize the whole thing altogether.

Whatever the reason, they end up rearranging their files and changing their directory structure, breaking all their URLs. Even if you're just rearranging the site's look, it takes a lot of discipline not to move the actual files around, probably more than you're going to have. And setting up redirects for everything is so difficult that you're just not going to bother.

Much better to plan ahead so that the problem never comes up in the first place, by leaving categories out of the URL altogether.

So that's a lot of don'ts, what about some do's?

Well one easy way to have safe URLs is to just pick numbers. So, for example, your blogging system might just assign each post with a sequential ID and give them URLs like:

http://posterous.com/p/234 http://posterous.com/p/235 http://posterous.com/p/236

Nothing wrong with that. However, if your site is a little more popular, the IDs can get quite long and confusing:

http://books.example.org/b/30283833

In a situation like this, you might want to encode numbers using base 36 instead of base 10. Base 36 means you get to use all the letters in addition to just numbers, but only one case, so there's no confusion about how to capitalize numbers. (Imagine someone reading the URL over the phone. It's a lot easier to say "gee, five, enn, four" than "lower-case gee, the number five, upper case enn, the number four".)

To be super-careful, you might want to go a step further and skip any numbers that end up having zero, O, one, L, or I in them, since those letters can often be confused.

The result is that you have URLs that look like:

http://books.example.org/b/3j7is

and end up being a lot shorter. While four base 10 digits can only go up to 9999, in base 36 zzzz is actually 1,679,615. Not bad.

One problem with numerical identifiers, however, is that they're not "optimized" for search engines. Search engines don't just look at the content of a page to decide whether it's a good result for someone's search, they also look at the URL which, because it's so limited, is given special weight. But if your URLs are just numbers, they're unlikely to have anything that matches people's search engine queries, making them less likely to be found in search results. To fix this, people are appending some text after the number, as in:

http://www.hulu.com/watch/17003/saturday-night-live-weekend-update-judy-grimes

The text at the end is part of the URL, but it's not used to identify the right page. Instead, the system looks only at the number. Once it gets there, it looks at the current title it has for the number, sees if it matches the URL, and if it doesn't, redirects users to the correct one. That way they can type in:

http://www.hulu.com/watch/17003/

or even:

http://www.hulu.com/watch/17003/this-is-where-i-got-the-joke-above [^sub]

[^sub]: Isn't interesting how even though we typically read books as series of pieces of paper stacked from left to right, we still refer to things that come earlier as "above" the others (or _supra_ if you want to be all Latin about it), as if we were all reading the raw scroll of paper that Kerouac emitted from his typewriter.[^kero] Of course, unlike my typewriter- or notebook-bound predecessors, I'm writing this in a wordprocessor whose simulated form of up-down perfectly mimics Kerouac's physical scroll. Coming full circle, I suppose.

[^kero]: See, e.g., http://www.npr.org/templates/story/story.php?storyId=11709924 for details.

and still get the right page. This isn't perfect, since many users will still think they have to type in the long text "saturday-night-live-weekend-update-judy-grimes", but it's probably outweighed by the number of additional users who will find you more easily on search engines. (Ideally, there would be some way in the URL to indicate to humans that the remaining text is optional, but I haven't seen any conventions here yet. I guess the hope is that they'll notice the number and just get the idea.)

(You'll note that all these URLs are within directories, not at the top-level. This just feels cleaner to me -- I don't like imagining the entire site's files are sprawled across the root directory randomly; it's much nicer to think of them stacked up inside ‘/watch/‘ or ‘/b/‘. But if your main nouns are subdirectories themselves,

as with the user pages on Twitter and Delicious, it might make sense to break this rule. (More on this in a bit.))

Numbers work well in cases where pages get created automatically (maybe you're importing a lot of stuff, or you generate pages in response to emails or incidentally for other actions) or their titles tend to change, but in other cases you might prefer what's called a _slug_. A slug is just a little bit of text that looks good in a URL, like ‘wrt_dfw‘ or ‘beyond-flash‘. When a user creates a page, you have them create the slug at the same time (perhaps including an auto-generated one from the title by default), and then you force them to stick with it (or else make sure to redirect all the old ones whenever it changes).

On sites like Wikipedia, slugs are basically generated incidentally. When you include text like ‘Jackson was hardly a fan of the late [[Robert Davidson]]‘ the site automatically links you to a new page with the slug ‘Robert_Davidson‘. Especially with the numerous conventions about titles Wikipedia has built over the years (along with the endless back-up redirects), the result is surprisingly convenient.

You'll note that all this discussion has basically been about nouns -- the main things that make up your site, whatever those are (videos, blog posts, books). There are typically three other types of pages: subpages (which drill down into some aspect of the nouns), site pages (like about and help and so on), and verbs (which let you do things with the nouns).

Subpages are some of the easiest, and some of the most difficult. In the easy cases, you just indicate the subpage by adding a slash and a slug for the subpage. So, if your page for Nancy Pelosi is at:

http://watchdog.net/p/nancy_pelosi

it seems pretty obvious that your page on her finances should be at:

http://watchdog.net/p/nancy_pelosi/finances

Sometimes the majority of your site is subpages. So with Twitter, a user's page is at:

http://twitter.com/aaronsw

while their status messages get URLs like:

http://twitter.com/aaronsw/statuses/918239758

(Notes: The ‘statuses‘ bit is redundant and the number way too long.)

But things get more complicated when your nouns have more complex relationships. Take Delicious, where _users_ post _links_ under various _tags_. How should things be structured? user/link/tag? tag/user/link?

Delicious, which for a long time used its URL scheme as a primary navigation interface, is so brilliant at its URL choices that it should be carefully studied. They decided that users were the primary object and gave them the whole space

in ‘/‘ (like Twitter). And underneath each user, you could filter by tags, so you have:

http://delicious.com/aaronsw (links from me) http://delicious.com/aaronsw/video (links from me tagged "video") http://delicious.com/aaronsw/video+tech (links from me tagged "video" and "tech")

And then they created a special psuedo-user called tag that lets you see all links with a tag:

http://delicious.com/tag/tech (all links tagged "tech")

(The URLs for links aren't as smart, but let's not dwell on that.) It's hard to give general rules for how to solve such inter-linking problems; you basically have to do what "feels right" for your app. for social sites, like Delicious and Twitter, this means putting the focus on the users, since that's primarily what users care about. But for other apps that might make less sense.

It's tempting to just not decide and support all of them. So, in place of Delicious, you'd have:

http://del.example.org/u/aaronsw (links from me) http://del.example.org/t/tech (links about tech) http://del.example.org/u/aaronsw/t:tech (links from about tech) http://del.example.org/t/tech/u:aaronsw (links about tech from me)

The problem here is that the last two are duplicates. You really want to pick one form and stay with it, otherwise you end up confusing search engines and browser histories and all the other tools that try to keep track of whether they've already visited a page or not. If you do have multiple ways of getting to the same page, you should pick one as the official one and make sure all the others redirect. (In an extreme case, you'd take the ‘video+tech‘ example above and redirect it to ‘tech+video‘, making the official URL be the one where the tags are in alphabetical order.)

Next up: site pages. Looking at Twitter and Delicious basically give away the store above (you mean I can have ‘twitter.com/contact‘ if my username is ‘contact‘?!), you might wonder where they can possibly put their help and login pages. One trick might be to reserve a subdirectory like ‘/meta/‘ and put everything in there. But Delicious and Twitter seem to get by just by reserving all the important potential-page-names and putting stuff there. So, as you'd expect, Twitter's login page is at:

http://twitter.com/login

And, if you're not expecting to have a lot of site pages, this will get you thru. (Be sure to reserve ‘help‘ and ‘about‘, though.)

And, of course, if you're not giving away the store, you don't have any of these problems. So just pick the sensible URLs for the pages that users come to expect. And, of course, be sure to follow all the noun-principles above.

That was easy, so we're left with just verbs. There are two ways you might imagine verbs working:

pass the noun to the verb: /share?v=1234 pass the verb to the noun: /v/1234?m=share

After spending a lot of time experimenting with this, I'm convinced the latter is the right way. It takes up less of the "URL-space", it sorts nicer in people's address bars, and it makes it visually clear that you're doing something to an object.

It's tempting to just use subpages, like:

/v/1234/share

but I prefer the '?m=share' formulation for two reasons: first, it works even when your nouns already have subpages, and second, it makes it clear that the page is meant to do something, not just convey more information. But the converse is true as well. Don't do:

/p/nancy_pelosi?m=finances

making it look like the page is supposed to do something when it really just conveys more information.

Alright, that's enough about picking URLs. Let's move on to actually doing something with them!

Building for search engines: following REST

Let's talk about vacuum cleaners. It's an all-too-common story. You've got a nice shiny new apartment, but it doesn't stay that way for long. Dust falls on the floor, crumbs roll off your plate, flotsam, jetsam, and the little pieces from Jetsons' toys begin to clutter your path. It's time to clean.

Sweeping is fun at first -- it gives you a little time to get lost in thought about your web application while you're doing an ostensibly-useful repetitive-motion activity -- but soon you grow tired of it. But liberal guilt and those Barbara Ehrenreich articles you read make you resistant to hiring a maid. So instead of importing a hard-up girl from a foreign country to do your housework, you hire a robot.

Now here's the thing about robots (and some maids, for that matter): it's not at all clear to them what is trash and what is valuable. They (the robots) wander around your house trying to suck things up, but on their way they might leave tire-treads on your manuscript, knock over your priceless vase, or slurp up your collection of antique coins. And sometimes it gets caught on the pull-cord for the blinds, causing the robot to go in circles while pulling the shutters open.

So you take precautions -- before you run the robot, you pick the cords off the floor and move your manuscript to your desk and take care not to leave your

pile of rare coins in the corner. You make sure the place is set up so that the robot can do its job without doing any real damage.

It's exactly the same on the Web. (Except without the dust, crumbs, Jetsons, maids, tire treads, vases, coins, or blinds.) Robots (largely from search engines, but others come from spammers, offline readers, and who knows what else) are always crawling your site, leaving no nook or cranny unexplored, vacuuming up anything they can find. And unlike the household variety, you cannot simply unplug them -- you really have to be sure to keep things clean.[^ft]

[^ft]: Although see http://ftrain.com/robot_exclusion_protocol.html

Some people think they can just box robots out. "Oh, you need a login to get in; that'll keep out the robots." That's what David Heinemeier-Hansson, creator of Rails, said. He was wrong. Google software that ran on users computers ended up exploring even pages behind the log-in requirement, meaning the robots clicked on all the "Delete" links, meaning robots deleted all the content. (Hansson, for his part, responded by whining about the injustice of it all.) Don't let this happen to you.

Luckily, that genius Tim Berners-Lee (see previous chapter) anticipated all this and set precautions. You see, when you visit a website, you don't just ask the server for the URL, you also tell it what kind of request you're making. Here's a typical HTTP/1.0 request:

GET /about/ HTTP/1.0

The first part (‘GET‘) is called the method, the second (‘/about/‘) is the path, and the third (‘HTTP/1.0‘) is obviously the version. GET is the method we're probably all familiar with -- it's the normal method used whenever you want to _get_ (GET it?) a page. But there's another method as well: POST.

If you think of URLs as little programs sitting inside a server somewhere, GET can be thought of as just running the program and getting a copy of its output, whereas POST is more like sending it a message. Indeed, POST requests, unlike GET requests, come with a payload. A message is attached at the bottom, for the URL to do with as it wishes.

It's intended for requests that actually _do_ something that messes the order of the universe (or, in the jargon, "changes state"), instead of just trying to figure out what's what. So, for example, reading an old news story is a GET, since you're just trying to figure stuff out, but adding to your blog is a POST, since you're actually changing the state of your blog.

(Now, if you want to be a real jerk about it, you can say that all requests mess with the state of the universe. Every time you request an old news story, it uses up electricity, and moves the heads around on disk drives, and adds a line to the server's log, and puts a note in your NSA file, and so on. Which is all true, but pretty obviously not the sort of thing we had in mind, so let's not mention it again. (Please, NSA?))

The end result is pretty clear. It's fine if Google goes and reads old news stories, but it's not OK if it goes around posting to your blog. (Or worse, deleting things from it.) Which means that reading the news story has to be a GET and blog deleting has to be a POST.

Actually, that's not quite true. There are other verbs besides GET and POST (although those are by far the most common). There's GET, HEAD, POST, PUT, DELETE, CONNECT, OPTIONS, PATCH, PROPFIND, PROPPATCH, MKCOL, COPY, MOVE, LOCK, UNLOCK, TRACE (and probably others). GET and POST we've already seen. HEAD is like GET but only requests the headers or a page and not the actual content. PUT is there if you want to replace the contents of the page with something entirely new -- TimBL's original web browser used PUT whenever you tried to save a change you made to a page. PATCH is like PUT but only changes part of a page. DELETE, MOVE, COPY, LOCK, and UNLOCK should be pretty self-explanatory. CONNECT is used for proxying and tunneling other stuff. OPTIONS lets you find out what the server supports. PROPFIND and PROPPATCH are used for setting properties in the WebDAV protocol. MKCOL is for making a WebDAV collection. (These probably shouldn't have all gotten their own methods...) TRACE asks the server to just repeat back the request it got (it's useful for debugging).

But, frankly, GET and POST are the most frequently used, in no small part because they're the ones supported by all Web browsers. GET, of course, is used every time you enter a URL or click on a link, while POST can be used in some forms. (Other forms are still GET, since they don't change anything.)

Following these rules is called following REST, after the 2000 Ph.D. dissertation of Roy Fielding, coauthor (with Tim Berners-Lee and some others) of the official HTTP specification (RFC 2616, if you're interested). Roy, a big bear of a man with a penchant for sports, set out to describe theoretically the various styles ("architectures") of network-based applications. Then he describes the interesting hybrid that the Web adopted, which he terms "Representational State Transfer" or REST.

While REST is often used to mean something akin to "use GET and POST correctly", it's actually much more complicated, and more interesting, and we'll spend a little time on it just so you can see the different kind of architectural tradeoffs that those Masters of the Universe who have to design a system like the Web have to think about.

The first choice made was that the Web would be a client-server system. Honestly, the Web is probably this way because Tim did things this way and Tim did thinks this way because that's how everything else on the Internet was back then. But it's not impossible to imagine that the Web could have been more peer-to-peer, like some of the file-sharing services we see today. (After all, the Web is in no small part just file-sharing.)

The more likely option is, of course, to break away from the Web altogether, and force people to download special software to use your application. After all,

this is how most applications worked before the Web (and how many still work today) -- new software, new protocols, new architectures for every app. There are certainly some good reasons to do this, but doing so breaks you off from the rest of the Web community -- you can't be linked to, you can't be crawled by Google, you can't be translated by Babelfish, and so on. If that's a choice you want to make, you probably shouldn't be reading this book.

The second major choice was that the Web would be "stateless". Imagine a network connection as your computer phoning up HQ and starting a conversation. In a stateful protocol, these are long conversations -- "Hello?" "Hello, welcome to Amazon. This is Shirley." "Hi Shirley, how are you doing?" "Oh, fine, how are you?" "Oh, great. Just great." "Glad to hear it. What can I do for you?" "Well, I was wondering what you had in the Books department." "Hmm, let me see. Well, it looks like we have over 15 million books. Could you be a bit more specific?" "Well, do you have any by Dostoevsky?" (etc.). But the Web is stateless -- each connection begins completely anew, with no prior history.

This has its upsides. For one thing, if you're in the middle of looking for a book on Amazon but right as you're about to find it you notice the clock and geebus! it's late, you're about to miss your flight! So you slam your laptop shut and toss it in your bag and dash to your gate and board the plane and eventually get to your hotel entire _days_ later, there's nothing stopping you from reopening your laptop in this completely different country and picking up your search right where you left off. All the links will still work, after all. A stateful conversation, on the other hand, would never survive a day-long pause or a change of country. (Similarly, you can send a link to your search to a friend across the globe and you both can use it without a hitch.)

It has benefits for servers too. Instead of having each client tie up part of a particular server for as long as their conversation lasts, stateless conversations get wrapped up very quickly and can be handled by any old server, since they don't need to know any history.

Some bad web apps try to avoid the Web's stateless nature. The most common way is thru session cookies. Now cookies certainly have their uses. Just like when you call your bank on the phone and they ask you for your account number so they can pull up your file, cookies can allow servers to build pages customized just for you. There's nothing wrong with that.

(Although you have to wonder whether users might not be better served by the more secure Digest authentication features built into HTTP, but since just about every application on the Web uses cookies at this point, that's probably a lost cause. There's some hope for improvement in HTML5 (the next version of HTML) since they're-- oh, wait, they're not fixing this. Hmm, well, I'll try suggesting it.[^w])

[^w]: http://lists.whatwg.org/pipermail/whatwg-whatwg.org/2008-October/016742.html

The real problem comes when you use cookies to create sessions. For example,

imagine if Amazon.com just had one URL: . The first time you visited it'd give you the front page and a session number (let's say 349382). Then, you'd send call back and say "I'm session number 349382 and I want to look at books" and it'd send you back the books page. Then you'd say call back and say "I'm session number 349382 and I want to search for Dostoevsky". And so on.

Crazy as it sounds, a lot of sites work this way (and many more used to). For many years, the worst offender was probably a toolkit called WebObjects, which most famously runs Apple's Web store. But, after years and years, it seems WebObjects might have been fixed. Still, new frameworks like Arc and Seaside are springing up to take its place. All do it for the same basic reason: they're software for building Web apps that want to hide the Web from you. They want to make it so that you just write some software normally and it magically becomes a web app, without you having to do any of the work of thinking up URLs or following REST. Well, you may get an application you can use through a browser out of it, but you won't get a web app.

The next major piece of Web architecture is caching. Since we have this long series of stateless requests, it sure would be nice if we could cache them. That is, wouldn't it be great if every time you hit the back button, your browser didn't have to go back to the server and redownload the whole page? It sure would. That's why all browsers _cache_ pages -- they keep a copy of them locally and just present that back to you if you request it again.

But there's no reason things need to be limited to just browser caches. ISPs also sometimes run caches. That way, if one person downloads the hot new movie trailer, the ISP can keep a copy of it and just serve the same file to all of their customers. This makes things much faster for the customers (who aren't competing with the whole world for the same files) and much easier on the server operator (who no longer has to serve quite so many copies). The one problem is that it does tend to mess up your download statistics a bit, but server operators can decide if they want to pay that price.

Similarly, servers can run caches. Instead of browsers visiting the server directly, they hit a server cache (technically known as a reverse proxy) that checks to see if it already has a copy of the page and, if so, serves it, but otherwise asks the real server for it. If you build your web app to follow REST, you can often make your site much, much faster just by sticking a nice server cache (like Polipo) in front of it. But, of course, if you do bad things like use session cookies and ignore the rules about GET and POST, the server cache will just screw everything up. (Notice that only GETs can be cached; you wouldn't want to cache the result of something like adding a new blog post or the next blog post would never get added!)

GET and POST are, of course, part of the next piece of architecture, which Fielding calls "Uniform Interfaces". Every web app works the same basic way: there are a series of URLs which you perform methods on. The methods sometimes change the state of the object and the server always returns the

resulting "representation" of the object.

Thus the name: Representational State Transfer (REST).

Building for choice: allowing import and export

Robots and browsers and protocols are fun, sure, but if you want your site to succeed it ultimate has to appeal to humans -- the real people who build and use all that other stuff. And even if information doesn't, humans generally want to be free. If you don't believe me, ask a friend to lock you in their trunk, and then reevaluate your position.

Greedy folks (i.e. businesspeople) tend to be kind of short-sighted about this. "If I put big metal spikes in front of the exit," they think, "my customers will never want to leave! My customer retention rates will go thru the roof." They decide to give a try and they have some big metal spikes installed in front of the exit. And, being the sober-minded realist businesspeople like to pretend they are, they measure customer retention rates before and after the metal spikes. And, sure enough, it worked -- people aren't leaving. Just look at those numbers! But what they didn't measure is that people also aren't coming back. After all, nobody wants to go someplace with spikes on the exits. Think about this next time you find that pop-up ads increase sell-thru rates.

This is why a site like Amazon is such a cluttered mess of sell boxes. Amazon's managers insist that they're rigorously hard-headed engineers. The boxes are there because they sell things and their job is to make money. Clean, clear, uncluttered pages may appeal to kids in art school or Apple interns, but here in the real world cash is king. And like Mark Penn advising Hillary Clinton, if you don't believe them they'll pull out the numbers to "prove" it. Every box, they say, was carefully tested: half the users were given a page with the new box, half without. And the users who got the page with the box bought more.

Well, no duh. Obviously more people are going to buy something if you ask them to, just like more McDonald's customers will SuperSize their order when the disaffected teenager asks them to. But Amazon's gone way beyond that -- now we're into the realm of the girl-with-the-headset pitching us on every third item off the menu. Sure, you may buy more this time, but after the stomachache hits you'll make sure your next outing is to Burger King.

Companies rarely try to measure such effects and even if they did, it's not easy. It's a piece of cake to serve someone an additional link and seeing if they click on it; keeping track of whether they come back to the store in the weeks and months to come is much harder. Worse, the difference made by any one additional box is subtle, and thus hard to measure. The real test isn't whether removing one sell box gets Amazon more customers, it's whether switching to a kindler, gentler layout does. But that would be a radical change for Amazon -- and thus pretty hard to test without raising hackles and freaking people out.

"Well, if you can't measure people," the MBAs say, "you can at least ask them." And thus the dreaded "focus group", whose flaws can dwarf even the most bogus

statistical study. At least with the click-through games you're measuring what people actually do; with focus groups you find out what people want you to think they say they do, which is a very different thing.

For one thing, people are notoriously bad observers of themselves. For the most part, we don't know why or how we do things, so when we're asked we make up rationalizations on the spot. This isn't just carelessness -- it's how the brain works. To accomplish tasks of any complexity, we need to make their component parts automatic -- you'd never get to the store if you had to think about which thigh muscles to move to get your leg in the right position -- and automatic behavior is exactly behavior we don't think about (this is why athletes' memoirs are so boring[^d]).

[^d]: See D. F. Wallace, "How Tracy Austin Broke My Heart" in _Consider the Lobster_ (2005).

So not only are you asking people a question they don't -- can't -- know the answer to, you're also asking them in a nice conference room, filled with other people, after giving them some cash. It doesn't take much reading in social psychology to realize this isn't exactly an ideal situation for honesty. People are, of course, going to say what they think you want to hear, and even if you have the most neutral of moderators asking the questions, they're going to be able to make some educated guesses as to what that is.

Which is why watching focus groups is such an infuriating experience: like a girl pretending to play dumb in a bar, you're watching people act the way they think people expect people like them to behave.

But if you can't measure people and you can't ask them, what does that leave? Well, good old-fashioned experience. As is usually true in life, there's no shortcut around incompetence; at some point, you just need genuine ability. When it comes to pleasing users, this generally has two parts: First, you need the basic skill of empathy, the ability to put yourself in a user's shoes and see things through their eyes. But for that to work, you also need to know what it's like inside a user's head, and as far as I can tell the best way to do this is just to spend lots of time with them.

The best usability expert in the world, Matthew Paul Thomas, spent the first few years of his life doing tech support in a New Zealand cybercafe. This is the kind of job you imagine Stalin exiling programmers to, but Thomas made the best of it. Instead of getting angry at dumb users for not understanding what a "Taskbar" was, he got pissed off at the idiots who designed a system that required such arcane knowledge. And now that he's in a position to fix such things, he understands at a deep visceral level what their flaws are.

I wouldn't wish to force such an exile on anyone (well, I suppose there are a couple anonymous UI designers who might be candidates), but there's certainly no shortage of people who already possess a user's intuition to various degrees. The problem is that no one listens to them. It's always so easy to dismiss them

as naive or dumb or out-of-touch. After all, the additional menu bar option you want to add makes perfect sense to _you_ -- how could it really make things worse?

When a company does focus on their users, it's a real shock. Take Zappos, an online shoe store. Zappos is fanatical about customer service. They quietly upgrade first-time purchasers to overnight delivery, they write cards and send flowers when the situation warrants it, and they not only give complete refunds but pay for shipping the shoes back as well. It's the kind of company people rave about. But, most interestingly for our purposes, if they don't have the shoe you want in stock, they try to find you a competitor that does!

From the short-term perspective, this seems insane: why would you actually do work to help your customers buy shoes from someone else? But, in the long term, it's genius. Sure, you may make one purchase somewhere else, but not only will you go back to Zappos for every other shoe purchase for the rest of your life, you'll also write long, glowing blog posts about how awesome Zappos is.

This is one of the secrets of success on the Web: the more you send people away, the more they come back. The Web is full of "leaf nodes" -- pages that say something interesting, but really don't link you anywhere further. And leaf nodes are great -- they're the core of the Web in fact -- but they're the end of a journey, not the beginning. When people start their day, or their web browser, they want a page that will take them to a whole bunch of different sites and perspectives, not just try to keep them cooped up in one place. What's the by-far most popular site on the Internet? Google Search, a site whose goal is to get you someplace else as quickly and unobtrusively as possible.

The reason all this stuff about metal spikes and New Zealand exiles and shoe stores and leaf nodes is relevant to a book on web apps is because I'm now going to ask you to do something that seems insane, something that sounds like it will kill your site. I'm going to ask you to open up your data. Give it away.

I'll give you a second to catch your breath.

It's not as crazy it sounds. Wikipedia, a successful site by any measure, gives away the store -- you can download full database dumps, including not just every page on Wikipedia, but every change made to every page, along with full permission to republish it as you see fit. It doesn't seem to have hurt their popularity any.

Obviously, I'm not saying you publish users' personal details for everyone to see. It would be crazy for Gmail to put up a site where you could download every one of their users' email. Instead, I'm suggesting you let users get their own data out of your site. People who put their events in your calendar should be able to export their calendar; people who got their email thru Gmail should be able to get it back out again.

Good export isn't just the right thing to do, it can also be a strong way to

attract users. Folks are uncomfortable about pouring their whole life into a hosted web application -- they've been burned too many times by companies that took all their data and went bust. Going out of your way to make sure they can get their stuff out of your site can do a lot to regain their trust.

While I have a lot to say about formats in this book, the actual format you use is kind of irrelevant here. The important thing is that you do it at all. XML, RDF, CSV -- the popular blogging system Movable Type actually just dumped posts as long text files, and while it was a dreadful format to work with, it was better than nothing. As long as you pick something halfway sensible, people will find a way to make it work.

The exception is if there's already a standard (de facto or otherwise) in your field. For example, OPML is pretty widely accepted as the way to export the list of blogs you read. If there is, you just have to support the standard. Sorry. If other software provides a way to important a certain format, you're just going to have to bite the bullet and output in that format. Anything else looks like churlishness and users aren't going to care about the technical details.

And, of course, it goes both ways: a great way to attract users is to provide import functionality yourself. By supporting import from other products, whether they have official export features or not, you make it easy for users to slide into your own version. Even if your competitors don't have an official export function, you can still help users out (and offend your competitors) by scraping data out of their system -- writing custom tools to pull stuff out of their user interface and into your database.

The end result is the kind of frictionless world savvy users can only dream of -- smoothly gliding from one app to another, taking advantage of new features without having to give up your old data. And if the company making it gets bought and the developers who wrote all the new features quit and start a competitor, you can pull your data right back out again and zip over to the new app.

Which means more choice -- and isn't that ultimately best for everyone?

Building a platform: providing APIs

The other week I made one of my rare excursions from my plushly-appointed bed and attended a local party. There I met a man who made a website or entering and visualizing data. I asked him whether he had an API, since it seemed so useful for such a data-intensive site. He didn't, he said; it would be too much work to maintain both a normal application and an API.

I tell you this story because the fellow at the party was wrong, but probably in the same way that you are wrong, and I don't want you to feel bad. If even well-dressed young startup founders at exclusive Williamsburg salons make this mistake, it's no grave sin.

See, the mistake is, that if you design your website following the principles in this

book, the API isn't a separate thing from your normal website, but a natural extension of it. All the principles we've talked about -- smart URLs, GET and POST, etc. -- apply equally well to web sites or APIs. The only difference is that instead of returning HTML, you'll want to return JSON instead.

JSON (pronounced like "Jason"), for the uninitiated, is a simple format for exchanging basic pieces of data between software. Originally based on JavaScript but quickly adopted by nearly every major language, it makes it easy to share data over the Web.

Wait!, you may cry, I thought XML was for sharing data on the Web. Sadly, you have been misled by a sinister and harmful public relations campaign. XML is probably just about the worst format for sharing data. Here's why:

Modern programming languages have largely standardized on the same basic components of internal data structures: integers, strings, lists, hashes, etc. JSON recognizes this and makes it easy to share these data structures. Want to share the number 5? Just write ‘5‘. The string "foo" is just ‘"foo"‘. A list of the two of them is simply ‘[5, "foo"]‘ -- and so on.

This is easy for humans to write and read, but even more importantly, it's _automatic_ for computers to write and read. In most languages you don't even need to think about the fact that you're using JSON: you just ask your JSON library to serialize a list and it does it. Read in a JSON file and you it's just like your program's getting a normal data structure.

XML, on the other hand, supports none of this. Instead, it thinks in terms of elements with character data and programming instructions and attributes, all of which are strings. Publishing data as XML requires figuring out how to shoehorn your internal data into a particular format, then making sure you do all of your quoting properly. Parsing XML is even worse.

The main reason XML is so bad at sharing data is because it was never designed to do that in the first place. It was a format for marking up textual documents; annotating writing with formatting instructions and metadata of various sorts, ala HTML. This is why it does things like distinguish between character data and attribute data -- attribute data is stuff that isn't part of the actual text, ala:

> I'm looking forward to a successful demonstration.

The word "green" is an annotation, not part of the text, so it goes in an attribute. All o this goes out the window when you start talking about data:

> Robert Booker >

Why is ‘age‘ an attribute while ‘name‘ is an element? It's completely arbitrary, because the distinction makes no sense.

Alright, so XML has a few more features that nobody needs. What's the harm in that? Well, it's also missing a whole bunch of features that you do need -- by default, XML has no support for even the most basic concepts like "integer";

it's all strings. And adding it requires XML Schema, a specification so mind-numbingly complex that it actually locks up my browser when I try to open it.

But the costs of such complexity aren't simply more work for developers -- they really come in the form of bugs, especially security holes. As security expert Dan Bernstein observes, two of the biggest sources of security holes are complexity ("Security holes can't show up in features that don't exist") and parsing ("The parser often has bugs... The quoter often has bugs... Only on rare joyous occasions does it happen that the parser and the quoter both misinterpret the interface in the same way.").[^djb]

[^djb]: http://cr.yp.to/qmail/guarantee.html

XML combines the worst of both worlds: it is an incredibly complex system of parsing. Not surprisingly, XML has been responsible for hundreds of security holes.[^cve]

[^cve]: http://cve.mitre.org/

So aside from being simpler, easier, more featureful, safer, and faster than XML, what does JSON have to offer? Well, it has one killer feature that's guaranteed its place atop the format wars: because it's based on JavaScript, it has a deep compatibility with web browsers.

You've probably heard about AJAX, a technique that uses the XmlHttpRequest function in modern web browsers to allow web pages to initiate their own HTTP requests to get more data. But, for security reasons, XmlHttpRequest is only permitted to request pages on the same domain as the web page it initiates from. That is, if your page is at http://www.example.net/foo.html it can request things like http://www.example.net/info.xml but not http://whitehouse.gov/data/dump.xml

For APIs, this is kind of a disaster -- the whole point of opening up your data on the web is so that _other_ sites can use it. If you're the only people who can access it, why go to the trouble?

Luckily, there's one exception: JavaScript. A webpage can embed an tag that points to any random site on the Internet. Even better, JavaScript code can arbitrarily add these script tags to the page. The browser then goes and fetches the page and tries to process it.

Now with regular JSON that wouldn't be too useful -- the browser would download a list or an object or something and wouldn't know what to do with it. So instead of just returning the JSON, it returns the JSON wrapped in a function call:

> myCallback([5, "foo"]);

Then you just have the function ‘myCallback‘ do whatever it was you wanted to do with the data.

Of course, if you're doing lots of requests you'll want to keep them all separate -- they can't all call myCallback. So you support a callback parameter that lets you pick the function name. So a URL like http://www.example.net/info.json?callback=foo would return:

> foo([5, "foo"]);

The whole technique is known as JSONP and, naturally, it's automated by all the major JSON libraries so you don't have to worry about any of these details.

Alright, so now that we have a pile of JSON, where do we put it. The answer, of course, is the same place as your HTML. Going back to an older example, let's say we have some information about a book at:

http://books.example.org/b/3j7is

Where does the JSON go? At the very same place!

You see, HTTP has a nifty feature called Content Negotiation that allows for the same URL to return different formats depending on who's requesting things. The classical example of content negotiation is the transition from GIF images to the newer PNG image format. Some older versions of Internet Explorer didn't support PNG; servers could use Content Negotiation to send them older GIF images instead.

The way it works is that every time you make an HTTP request (like a GET), the client sends along a series of Accept headers saying what formats it likes. Here's a typical example:

> Accept: text/html; q=1.0, text/*; q=0.8, image/gif; q=0.6, image/jpeg; q=0.6, image/*; q=0.5, */*; q=0.1

This says the browser prefers HTML, then takes text, then GIFs and JPEGs, then any other image, then anything else.

But for APIs we don't need to do anything so complicated. We can just have our API clients send:

> Accept: application/json

and have the server keep an eye out for that and return the JSON version if it sees it. Otherwise, it serves the HTML as usual.

Of course, you'll probably want to provide an option for people who can't easily do Content Negotiation. So it's traditional to let:

http://books.example.org/b/3j7is.html

force the server to return HTML while

http://books.example.org/b/3j7is.json

always returns JSON. (And then you could have:

http://books.example.org/b/3j7is.json?callback=myCallback

to support JSONP.)

Alright, let's get concrete. What might one of these JSON pages look like? Let's stick with our book example for a moment. You could imagine a book page looking something like:

> { > 'id': '3j7is', > 'title': 'The ABC book', > 'by_statement': 'designed and cut on wood, by C. B. Falls.', > 'pagination': '[30] p. incl. col. illus.', > 'description': "An all-time favorite and a classic in its field, this big and beautiful ABC book by distinguished artist C. B. Falls has been making new friends with delighted children for over forty years.\nMr. Falls designed the book for his little three-year-old daughter who likes a big book with lots of pictures. The drawings are cut on wood blocks and printed from fourcolor plates, and the artist has personally superintended the reproduction of them. The imagination of a child or grown-up is left free to capture by its own thrill of recognition the familiar in a new-old medium where color has not obscured the outline nor played too many tricks with nature.", > 'publisher': 'Doubleday, Page & company', > 'authors': [> {'id': 'OL115179A', 'name': 'C. B. Falls'} >], > }

And if your site let people update book pages, you could imagine supporting PUT requests on this URI that allowed people to submit an updated version of the JSON object. You'd parse it and then execute the update.

Or, if you just let people comments on books, you could let them POST simple JSON data to the same URI that comments are normally posted to.

In fact, if your really wanted, you could just let them POST form data and parse it the same way as you would input from web browsers. Then you could let them know success or failure via HTTP error codes -- a 500 error would let them know it failed, while a 303 See Other redirect to the page itself would let them know they succeeded. When they followed the redirect and grabbed the page, it too could content negotiate to JSON.

* * *

Alright, now it's time to talk about a touchy subject. I've been holding off on this, but at some point it becomes unavoidable. Yes, I'm afraid it's time to talk about RDF.

You see, all this JSON stuff is great for writing little scripts on clients that talk to other scripts on servers, but it leaves something to be desired when working at Web scale. It's hard to imagine, for example, building particularly useful tools that work across different JSON APIs, the way web browsers work across all different kinds of HTML pages. Each JSON API has its own internal representations and conventions and protocols, which means you need to write special code to deal with each different one.

That's where RDF comes in. The idea behind it is simple: what if we had a format that did to data what HTML did to documents -- provide a single,

consistent representation for them that supports the hypertextual nature of the Web. That probably makes no sense, so let's look at some examples.

RDF documents are quite simple -- they're made up of "triples", simple sentences with three parts: a subject, a verb (called a predicate), and an object. Let's take a bit of our example from before, namely that the book with ID 3j7is has the title "The ABC book" -- in RDF, the subject would be ‘3j7is‘, the verb ‘title‘ and the object the string ‘"The ABC book"‘.

Only RDF is meant to work at webscale, so instead of fuzzy-wuzzy terms like ‘title‘, everything's a URI. As in:

> "The ABC book" .

(Those ‘#‘ signs are there to distinguish the fact that we're talking about the _concept_ described by a web page, rather than the web page itself.)

Of course, typing all those URLs out each time gets old fast, so we tend to abbreviate them:

> @prefix rdfs: . > > rdfs:label "The ABC book" .

Here's a rough rendering of the above JSON in RDF:

> @prefix : . > > > :title 'The ABC book'; > :by_statement 'designed and cut on wood, by C. B. Falls.'; > :pagination: '[30] p. incl. col. illus.'; > :description "An all-time favorite and a classic in its field, this big and beautiful ABC book by distinguished artist C. B. Falls has been making new friends with delighted children for over forty years.\nMr. Falls designed the book for his little three-year-old daughter who likes a big book with lots of pictures. The drawings are cut on wood blocks and printed from fourcolor plates, and the artist has personally superintended the reproduction of them. The imagination of a child or grown-up is left free to capture by its own thrill of recognition the familiar in a new-old medium where color has not obscured the outline nor played too many tricks with nature."; > :publisher 'Doubleday, Page & company'; > :author . > > :name "C. B. Falls" .

Aside from consistently using URIs, RDF has some pretty nice features. For one thing, something you want to do a lot with data is combine it, and RDF makes that very easy. To combine two RDF documents, you just concatenate them -- it's just a list of facts; two lists of facts together makes one long list of facts. It's not quite as simple with JSON, let alone XML.

Another nice feature of RDF is that it makes it easy to map between formats. Converting between two JSON formats typically requires code, but with RDF you can just publish another RDF document that explains the mapping, like:

> rdfs:label = :title .

That way software that knows about ‘rdfs:label‘ know that they can use ‘:title‘ properties the same way.

RDF does have these many nice features, but it does have one big downside: it's nowhere near as easy to use as JSON. Like XML, it has its own data model, which means writing special code to move between its way of viewing the world and yours. There are some tools and techniques to mitigate this (like my own rdftramp[^rt], which tries to make RDF look more like normal Python objects) but it's still a serious problem.

[^rt]: http://www.aaronsw.com/2002/rdftramp/

The RDF world has tried to address it by writing RDF replacements for all the existing tools of the software world: RDF databases, RDF programming languages, RDF query systems, RDF browsers, RDF reasoning engines, and so on. If you want, there's a whole world of RDF you can dive into.

Ultimately, however, I fear this isn't a very promising strategy -- it's going to be hard to create replacements for all these things which are as good or better than the original, and even if you do, people will still have sentimental attachments to the others.

So at this point, I would still categorize RDF as an aspiration. It would be nice as a universal publishing format -- there's a lot that cold be done with it -- but for day-to-day work, JSON is much better.

That said, RDF is, of course, far, far preferable to XML.

Building a database: queries and dumps

APIs are nice and all, but they're fairly limiting: they only give you the answers to questions you already know how to ask. Want to find out more about book 3j7is? Sure, it'll tell you. But want to know which books published recently share an author with a book published over a hundred years ago? That's a little more complicated.

But, luckily, not impossible. It seems ridiculous to come up with your own API that could answer any sort of question like this. But remember those RDF query languages we were making fun of in the last chapter? This turns out to be just the sort of thing they're perfect at.

The official RDF query language is called SPARQL (SPARQL Protocol And RDF Query Language -- pronounced "sparkle"). If you're familiar with SQL, the standard database query language, SPARQL will look similar, only with RDF stuck in all the right places. Here's how you express our previous question in SPARQL:

> PREFIX : > SELECT ?booknew, ?bookold > WHERE { > ?booknew :author ?author . > ?booknew :publication_year ?yearnew . > FILTER (?yearnew >= 2008) > > ?bookold :author ?author . > ?bookold :publication_year ?yearold . > FILTER (?yearold <= 1908) > }

There's a lot there, so let's go through is slowly. First we just declare the prefixes for our URIs, as usual. This is just to save us some typing. Then we say that

we want the values ‘?booknew‘ and ‘?bookold‘ returned for us. In SPARQL, anything beginning with ‘¿ is a placeholder that the query engine will try to find something to fit into.

The ‘WHERE‘ clause puts constraints on what can fit in those placeholders. ‘?booknew‘ has to have an author and a publication year and that publication year has to be equal to or larger than 2008. ‘?bookold‘ also has to have an author and a publication year -- and furthermore, its author has to be the same as ‘?booknew‘'s author. But its publication year has to be equal or less than 1908.

Now because SPARQL is designed to work at web-scale, you don't have to just keep this query at home. Instead, you can point it at another server's search system, called a SPARQL endpoint. You may not have a lot of information about books, but books.example.org probably does -- you can have it search for things that match your query.

To do so, you just take the query we generated above and stick into a properly formatted URL. And -- boom! -- back comes your list of answers.

Now another neat thing about SPARQL is that, done right, it can spread these queries across multiple SPARQL endpoints. So, for example, we can imagine writing a query for books whose authors were Jewish. The information about books and authors we can get from the bookserver, while Wikipedia (whose RDF version is called DBPedia) can tell us about people's religion. Of course, figuring out how to structure these queries in such a way that they don't take forever is an ongoing research project.

In the meantime, we can at least help those who can help themselves to our data, by providing bulk dumps. The theory here is simple: there are lots of queries and merges and visualizations people will want to do with your data that are going to be impractical to do through any sort of API, even one as fancy as SPARQL. So you might as well just give them a full copy of the data set.

And the practice is even simpler: just take the JSON you generate for each item on your site and put it in one big file.

You may want to compress it, since it'll probably be quite large.

Building for freedom: open data, open source

Our story starts with a paper jam. It was 1980 and the Artificial Intelligence Lab at MIT had received an elegant new printer from Xerox. The printer, however, had an unfortunate tendency to jam, causing print jobs to pile up and nothing to get printed until someone happened to notice and fix the jam.

For Richard Stallman, one of the programmers at the AI Lab, this wasn't such a big deal. With their previous printer, Stallman had simply changed the printer driver to detect whether the printer was jammed and, if it was, to notify anyone who had sent it a print job.

"If you got that message, you couldn't assume somebody else would fix it," Stallman later recalled. "You had to go to the printer. A minute or two after the printer got in trouble, the two or three people who got messages arrive to fix the machine. Of those two or three people, one of them, at least, would usually know how to fix the problem."

But the Xerox printer was different: Xerox hadn't provided the lab with the source code to their printer drivers. There was no way for Stallman to add this new functionality to the driver. When Stallman asked Xerox for the code, they refused to provide it, insisting that it was an important trade secret for their business. And when Stallman found a student at Carnegie Mellon who had been given access to the software, that student also refused to provide a copy, saying he'd signed a contract with Xerox not to share it.

Stallman was outraged. Computer software was supposed to be a tool to serve people; that's why he and his labmates spent their time writing software. And yet, through a combination of greed and legal restrictions, people were forced to suffer because they were prevented from improving these tools.

Stallman wanted to ensure no one else would be forced to suffer in this way; he wanted to build a computer system based around principles of freedom. In 1984 he quit his job and announced the GNU project.

Stallman later clarified that free software was software that guaranteed users four freedoms:

0. The freedom to run the program, for any purpose. 1. The freedom to study how the program works, and adapt it to your needs. (Source code is a requirement for this.) 2. The freedom to redistribute copies so you can help your neighbor. 3. The freedom to improve the program, and release your improvements to the public, so that the whole community benefits. (Again, source code is a requirement for this.)

"I consider that the golden rule requires that if I like a program I must share it with other people who like it. I cannot in good conscience sign a nondisclosure agreement or a software license agreement. So that I can continue to use computers without violating my principles, I have decided to put together a sufficient body of free software so that I will be able to get along without any software that is not free."

Stallman codified these freedoms in the GNU General Public License or GPL. If you modify a piece of software that is licensed under the GPL and redistribute it, the license requires that you also redistribute the source code at no extra charge and allow everyone who receives a copy to do likewise.

Since 1984, the GNU operating system (whose most popular flavor is GNU/Linux) has been built and released under the GPL. A 2007 study found that 13% of servers and 1% of desktops were sold running GNU/Linux. And anyone can download the entire operating system for free off the Internet.

The success of GNU/Linux has led to a larger free software movement as well as the "open source" movement, which releases software and its source code under copyright licenses that provide some of the software freedoms.

The Mozilla Firefox browser, for example, is open source and currently makes up around 15% of market. Large portions of the Mac OS X operating system are also open source, including WebKit, the core of Safari, the Mac OS X web browser.

The open source and free software movements have now built free alternatives for just about every major type of computer application, from word processing to video games. And for a time it seemed like Stallman's dream had come true: one could truly continue to use computers without having laws restrict one's freedom -- it was possible "to get along without any software that is not free."

* * *

Meanwhile, Tim Berners-Lee, an Englishman living in France who worked at a physics lab in Switzerland, was frustrated with how difficult it was for physicists to share documents. And so, in 1989, he came up with the World Wide Web, developed the standards that made it work, and built the first web browser and web server.

The power of the browser was its flexibility (or, in law professor Jonathan Zittrain's phrase, its "generative nature"). Just as a general-purpose computer allowed you to run any program, from a music player to a graphing calculator, the web browser let you view any kind of document. A book, a physics paper, or photos of cats with funny captions -- the web browser doesn't care; it displays whatever the server provides it with.

This seems like a trivial point now, but it was a vast change from other networked software at the time. Email programs, for example, are designed simply to display email -- they have an enormously specialized interface for composing emails, finding emails, seeing who an email is from and to, and placing emails in different folders. The same was true for discussion software, chat software, and other pieces of software that communicated over the network.

The Web was different: it did not specialize in any particular type of content, but let you share whatever you like.

This lack of specialization in the Web browser allowed people to move this specialization to the Web server. The traditional Web server simply served up static documents that someone had previously written. But it was quickly clear that there was no reason the server had to be so constrained.

Instead of simply serving up previously-composed documents, the server could compose new documents "on-the-fly" as they were requested. Thus, instead of simply having a document which listed what restaurants have tables available, a web server could be instructed to query the different restaurants, learn their availability, and construct a page from the results.

And users, instead of passively requesting different prewritten documents, could submit requests to the server and actually begin to interact with it. Thus, they could ask the server to reserve one of the tables and send their name and phone number along with that request.

The result was that the humble web browser quickly began to overtake all the other "specialized" applications. Instead of having a special program just for reading email, people read their email over the Web. Similarly, discussion groups, chat rooms, and other forms of social interaction have moved inside the Web browser.

But software developers quickly discovered that, for social creatures like us humans, everything has a component of social interaction. For example, titling and categorizing the photos you take would seem like an obviously solitary activity. But sites like Flickr demonstrate that people love to discuss and categorize photos of their friends, or even strangers, and that people, all things considered, would prefer to organize their photos in a program that exposes them to other people.

The result is the recent "Web 2.0" phenomenon, in which just about every piece of computing is moved onto the Web and made social in some way. For photos and videos, there is Flickr and YouTube. For news, there are sites like Digg and Reddit where you can submit, edit, and vote on news stories. Calendars, todo lists, even music collections and word processors are all being made into dynamic social web applications.

Pundits now discuss a not-too-distant future of "dumb clients" and "cloud computing" where the other applications on the computer disappear and all that is left is the web browser. And for people who use kiosk computers or Internet cafes, that future is already here.

For some, this is an exciting prospect. But for those, like Stallman, concerned with issues of software freedom, it is frightening. Even in the dark days of the proprietary printer driver, Stallman still had control over the computer which drove the printer, even if he did not have the source code to modify it. But with a Web 2.0 application, you don't have even that. The computer running your software is locked away in some distant server farm. You can only communicate with it through your web browser.

Now this does provide some flexibility. Web browsers can be programmed to block ads or extract content. Plugins like Zotero and Greasemonkey let users add new functionality to existing sites by intercepting and modifying documents as they come back from the Web server.

But this is a rather pale notion of freedom, like saying that moviegoers have control over the films they watch because they can hold pictures up in front of the screen as they watch.

Another option, of course, is providing APIs. Thus, instead of having to manually click the "buy" button on an Amazon page to buy a new set of razors, with an

Amazon API you can have a program automatically purchase the razors for you every month.

This is undoubtedly useful, but again, a rather pale notion of freedom compared to the four freedoms that free software provides. If Amazon was truly free, you wouldn't just be able to write programs to automate your usage of the application, you'd be able to change how the application actually works.

The obvious solution to this challenge is simply to release the software on the Web server under the GPL or some other free software license. Then anyone could download a copy and modify it to their heart's content. And a new version of the GPL has been released, AGPLv3, which requires that people who use its software in web applications make their software available to the application's users under its free terms.

But only a completely asocial web application consists purely of software. The vast majority of them are interesting because they give you access to data contributed by other users as well. For example, the software that lets people edit web pages is just about the least interesting thing about Wikipedia. The reason the site is so popular is because so many people have put their accumulated knowledge into that software.

Wikipedia has addressed this by going one step further -- not only is the source code free, the data is too. Anyone can download a copy of the Wikipedia database (excluding users personal information) and start up their own copy of Wikipedia based on it. And then they can modify their copy of Wikipedia's software to work however they please.

It's beautiful in theory, but in practice, of course, nobody does this. Even if your version of Wikipedia was full of fantastic new features, it would still be nearly impossible to get anyone to use it. People use Wikipedia because that's where all the other people are; it's practically impossible to get everyone to switch.

For Wikipedia, the problem is somewhat ameliorated by having some psuedo-democratic control over the site. So Wikipedia is run by a board elected by (a tiny subset) of its users and the board has nominal control over the software and modifications that get made to it. But this is still a far cry from the freedom GNU/Linux users have in the non-networked world. Running for office, getting elected, then pushing your patches through a change-resistant bureaucracy is a lot more difficult than modifying some source code files on your computer and restarting.

And so, the hard-core partisans of software freedom propose that we will see the pendulum once again swing away from centralized server computing and back to a world where we all run applications on our local machines. Only this time, instead of being applications that don,Äôt use the network or only talk to a distant server, they will be peer-to-peer applications, seeking out other users and interacting with them directly.

Some great strides have been made in building peer-to-peer software, in no small

part because of the vast amount of interest in using the technology to share music without getting caught by enforcers of the law. But, especially compared to Web 2.0 server technology, peer-to-peer is still in its infancy. Writing a social application so that it,Äôs peer-to-peer is about a thousand times harder than writing the same program as a web app.

Still, peer-to-peer software, if we could make it work, would seem to give the best of both worlds: the freedom to modify how a program functions on our local computers as well as the ability to share and collaborate with others across the Internet. And so, for those who care about freedom (as well as those who care about sharing music), this seems like an important avenue for further research.

* * *

In the meantime, even if, like the question of how to query across many large SPARQL databases, the problems of web application freedom are unsolved, you can still do get started. The Open Knowledge Foundation, a group promoting freely shared databases, has proposed an Open Software Service Definition.[^ossd] The definition essentially codifies the principles we discussed above:

1. Make your code available as free or open source software 2. Make your data available as Open Knowledge

[^ossd]: http://opendefinition.org/ossd

For free/open source software, there's the official lists of the Open Source Initiative and Free Sofware Foundation to tell whether your license is sufficiently free and open. Examples include the Expat/BSD license, the GPL, etc. The Open Knowledge Foundation similarly lists a series of licenses including some Creative Commons licenses, the GNU Free Documentation License, and so on.

That's the legal details, but the technical ones are just as simple: provide a source code repository for all your code and SQL dumps for all your data.

Of course, this leaves a lot of open questions. What about private data, for example? My own feeling is that people should at least be allowed to download their own data and any data they can access through the Web interface -- e.g. the data about your friends on Facebook.

And there's lots of room for experimenting in building sites that promote more Democratic control. Maybe you can try some things and tell me about them and they'll make them into the second edition of this book.

Conclusion: A Semantic Web?

Well, we've been through a lot together, you and I. We've built up an application from its humble URLs to its high-falutin' notions of democracy. Along the way, we've made its world safe for robots, query systems, researchers, and Richard Stallman. But how do we get from this kind of website to the grand vision of the Semantic Web we've heard so much about?

Let's start by realizing that just being on the Web is an amazing thing. URLs provide a unified addressing scheme for any document -- a pretty miraculous thing. Imagine trying to explain to folks of yesteryear about these incredible words which you could give to anyone and they could take it home, punch it into a box they have on their desk, and get just the article, picture, or video you meant. To a generation whose idea of document retrieval is driving to the local library, filling out some forms, and waiting a few weeks while they tried to make sense of your subscription and hunt it down, this is a pretty serious change.

But REST took it even further. By making the documents accessible by search engines through a standard protocol, you no longer even need to know the right URL for the thing you want. Just type a few choice words into Google and _boom!_ back comes just the thing you wanted in a quarter of a second.

Of course it's not just Google; REST makes possible an interconnected tapestry that supports everything from web browsers to web editors to intelligent translating intermediary proxies.

A hard act to follow. But then came the ability to get not just documents back from these far-off servers, but data. By importing and exporting raw data, we made it possible to switch software programs, providing a somewhat-free market of competition among products, creating massive consumer surplus. Go us!

But we didn't stop there. Just letting users take their data home with them is weak brew compared to sharing it with the wider world. (Picture a wide world of people at home with their weak data brew.) Which is why the Web invented APIs, letting us share the data with everyone who could think of a use for it.

Now we're not just a simple website, pushing pages to the browser and providing a "See also:" like list of other pages one can visit. We're actually exchanging the data itself from application to application, making possible a new world of mashups and intelligent applications.

This is an entirely new notion of the tapestry -- a tapestry of data instead of a tapestry of documents. Documents can't really be merged and integrated and queried; they serve mostly as isolated instances to be viewed and reviewed. But data protean, able shift into whatever shape best suits your needs.

But as our needs grow more varied, we need better ways to get at the data that will best serve them. Which is where our queries and dumps come in. No longer are we hampered by only being able to ask the questions a site's programmers have expected and accounted for; now we can ask whatever questions we like, or do processing that can't even be put in the form of a question at all. Combining these dumps from different data sources, the possibilities are endless.

But where do we go from here?

Obviously the first step is to take the large dumps we've all made and load them into one big database. And, of course, we've started to see people do that, from research projects to commercial companies like Metaweb's Freebase. Freebase is

an enormous collaborative Web-editable RDF-like database, prepopulated with data extracted from Wikipedia and numerous other sources and supplemented with the contributions of various users. Freebase is still quite small, but their aims are ambitious -- creating a database that combines numerous different sources and providing it as a backend to people who want to build more intelligent applications.

Ideally, of course, intelligent applications won't be dependent on a single commercial site, like Freebase, but will merge and combine knowledge from various sites across the Web, crawling and trawling for more useful information and deciding which bits of it to trust.

Already we're seeing things like this in research projects. One of the most exciting Semantic Web tools is a program called cwm, hacked together between (or during) meetings by Sir Tim Berners-Lee himself. Cwm (pronounced coom) is one of the most amazing programs I've seen; it's a veritable data swiss-army knife, all built on RDF.

Of course it does all the basics -- reading and writing RDF files of various formats, combining multiple files, printing all the results out in a pretty format. Naturally, it can also search through the resulting data to answer your questions in much the same way SPARQL does.

But cwm goes a step further. It doesn't just search through data; it thinks about it. cwm can follow logical _rules_; take this one for example:

{ ?x a :Man } => { ?x :mortality :mortal } .

(If something is a man, then it's mortal.) Feed this into cwm, along with:

:Socrates a :Man .

And it will logically deduce that Socrates is mortal. Such rulesets obviously have all sorts of uses, from logical parlor games to actual programming. But one obvious use is providing conversions between different RDF formats. Imagine two schemas: ‘joe:‘, which has ‘small‘, ‘medium‘, and ‘large‘ and ‘starbucks:‘ which has ‘short‘, ‘tall‘, ‘grande‘, and ‘venti‘. Now we can just write a few rules to convert between them:

{ ?x joe:size joe:small } <=> { ?x starbucks:size starbucks:tall } . { ?x joe:size joe:medium } <=> { ?x starbucks:size starbucks:grande } . { ?x joe:size joe:large } <=> { ?x starbucks:size starbucks:venti } .

Feed these rules into cwm, along with the data in one format, and cwm will "think" about it and spit out data in the other format.

But cwm can do much more than just basic logical inference. It also has a wide variety of built-ins, that can do everything from mathematical processing to advanced cryptography. With them, and some clever rules, you can even build entire programs using cwm.

Incidentally, none of this is new -- nearly all this stuff was written in 2001.

cwm is also smart enough to go onto the Web and find more rules like these, crawling through web pages for more bits of RDF to make it smarter. Following links and URLs and getting more data, it can surf the Web for data in the same way a bored teenager surfs the Web for fun.

If you're interested in giving this process a spin for yourself, you can try Tim's latest project: the Tabulator. It's a little add-on to your web browser that lets it see RDF documents in addition to regular web pages. Suddenly documents aren't just a list of boring tags or text, but a pathway of clickable links you can follow to your heart's content. (And, with later versions, you can even edit some of the fields.)

One can imagine tools like cwm and Tabulator sitting behind the applications we use every day, enhancing them with knowledge drawn from the wider Web.

For that's the real idea behind the Semantic Web: letting software use the vast collective genius embedded in its published pages. Think of all the places software uses APIs or databases: your spellchecker queries a website to find the definition of a word, your addressbook does a search to see if your friends are online, your calendar downloads a page to keep you posted on upcoming events. Now, imagine these programs weren't limited to one particular site, but could draw on the intelligence of the Internet at large.

Your spellchecker can suggest related or alternate words, or just keep up to date with the latest slang. Your address book can tell you where your friends are right now and what they've been up to lately. Your calendar can keep an eye out for events you might be interested in.

It's easy to make fun of these kinds of visions. My father, upon seeing such demos, always used to ask, "But why does your toaster need to know about stock prices?" And perhaps, ultimately, they're not worth all the effort. But the Semantic Web is based on bet, a bet that giving the world tools to easily collaborate and communicate will lead to possibilities so wonderful we can scarcely even imagine them right now.

Sure, it sounds a little bit crazy. But it paid off the last time they made that gamble: we ended up with a little thing called the World Wide Web. Let's see if they can do it again.