

# Will's Guide to Mashing-up Remote Databases using Page Scraping, May 28 2008 Edition

wschenk@gmail.com

<b>Who is this for</b>	<b>2</b>
<i>tl; dr</i>	2
<b>Case Studies</b>	<b>3</b>
<i>Small: the project.ioni.st podcaster</i>	3
<i>Medium: menumaps</i>	3
<i>Large: Benchcoach.com</i>	3
<b>Architecture</b>	<b>4</b>
<b>Parsing</b>	<b>5</b>
<i>What, incidentally, are we parsing?</i>	5
<i>What are you returning?</i>	6
<i>The Process</i>	6
<b>Loading</b>	<b>10</b>
<i>Storing: meta data and name coding</i>	11
<i>A word on compression</i>	12
<b>Architecture: Medium</b>	<b>12</b>
<b>Normalize</b>	<b>13</b>
<b>Architecture: User Submitted Content</b>	<b>14</b>
<b>Load: The magical bookmarklet</b>	<b>14</b>
<i>Bookmarklet</i>	14

<b>Parse: delegation &amp; type</b>	<b>15</b>
<b>That's it. Simple.</b>	<b>16</b>
<b>Tools we use</b>	<b>17</b>
<b>Can I do this?</b>	<b>18</b>

## Who is this for

One fundamental problem of this new Web 2.0 development is to synchronize a remote website with a local database. Cross site integration. People build mashups because something out there goes 80% of the way; wouldn't it be great to take it that next 20%? Or maybe they are going off in some weirdo direction, and there's 20% of good being suffocated with 80% of crap and goddamn it if they were only smart enough to filter this here and roll that up there and cut that out and bolt this on the side, well THAT would be something. You got an itch? Let's scratch it.

This paper explains scraping, parsing and merging techniques to load data from a remote site with the constraint that we have zero cooperation from said remote site. The audience is people who are either going to implement a mash-up from scratch or who wants to understand some of the technical issues involved with scraping. The bulk of the paper covers scraping and data cleansing; we do not get into advanced merging issues. (If you are looking for a [Theory Of Patches](#), it's somewhere else.)

### tl; dr

This is long and comprehensive. There are other ways which may work for you. If they do; great. The techniques below aren't fundamentally different than others but I've run into a bunch of problems that the following don't address.

Most similar to this paper, <http://www.igvita.com/2007/02/04/ruby-screen-scraper-in-60-seconds> is a write up of how to make a simple scraper easily. There's a lot of good, starter material on that site about a whole bunch of things and I recommend subscribing. It's a great introduction and much shorter than this, but the techniques below supersede this straightforward post in about every way. I think that using array indexing for the XPath queries is brittle in practice, but if you need something quick and fast consider it.

Both yahoo pipes (<http://pipes.yahoo.com/pipes/>) and dapper (<http://www.dapper.net/>) are websites which make it easy to parse things and then republish them machine readable form. The idea is that they handle all the scraping and some of the mangling, and then you have nice clean data to work with. If they worked the way that they claimed it would be great. But frankly my experience has been that they only work on well formed and well marked-up HTML and that's not what's out there. While it's probably not fair to call them toys, when I investigated them they basically didn't work. They may have gotten much better since then.

Visualizing Data by Ben Fry also touches on tangentially related things, but it's very good. Not as good as his Processing Book, but not much really is. I recommend them both, but the second is a classic. Or at least should be.

What follows is what I've found that works in the situations below, which can be quite complex, and the reasoning behind these decisions.

## Case Studies

We're going to talk about three real-world examples, from small to large. These are all my projects; two are play things and the largest one is our company. (By reading this document you are hereby obligated to get all your fantasy baseball friends to sign up for benchcoach. Haha, only serious.)

### Small: the [project.ioni.st](http://project.ioni.st) podcaster

Url: <http://sublimeguile.com/projectionist.xml>

This script scrapes the <http://project.ion.st> website and creates an XML feed for all the songs posted there. It runs once a day. Podcasts require that you specify the length of the mp3 file which is referenced, which is not available on the page. So we need to make some requests to the webserver where the mp3 is being hosted, which in this case is Amazon S3.

This is a standalone ruby script which writes out a simple xml file. This runs as a nightly cron job with zero maintenance.

### Medium: [menumaps](http://menumaps.monkeythumb.net)

url: <http://menumaps.monkeythumb.net/map>

Menumaps pulls restaurant and cuisine information from [menupages.com](http://menupages.com), does geocoding to figure out where all the restaurants are, and provides a really nifty google-map interface to all the data. As someone who lives on the border of a couple neighborhoods, I think neighborhood-base browsing is retarded. Do I care that something is in Carrol Gardens vs Cobble Hill vs Red Hook? Answer: No, I do not. Menumaps lets you drag the map around and pins will pop up where there are restaurants, even if you started your search somewhere else.

Menumaps is rails application with 4 different parsing scripts, stores information in a database and maintains remote database ids for records. It also uses a perl-based geocoding script to do lookups on a separate local database. (Not discussed.) This process works well, but the geocoding takes forever and needs monitoring.

### Large: [Benchcoach.com](http://benchcoach.com)

url: <http://benchcoach.com/>

Benchcoach.com is a fantasy baseball site which offers customized advice and analysis for your fantasy baseball team. We need to get player, rosters, standings, scoring

information from a remote site, all of which is probably hidden behind a login. i.e. much like email, this data is only accessible to the owner of that information and you need to login. (While we're on the subject, I think it's totally ass that I need to enter in my gmail user name and password for these social networking sites which then go and log in as me. That's just creepy.)

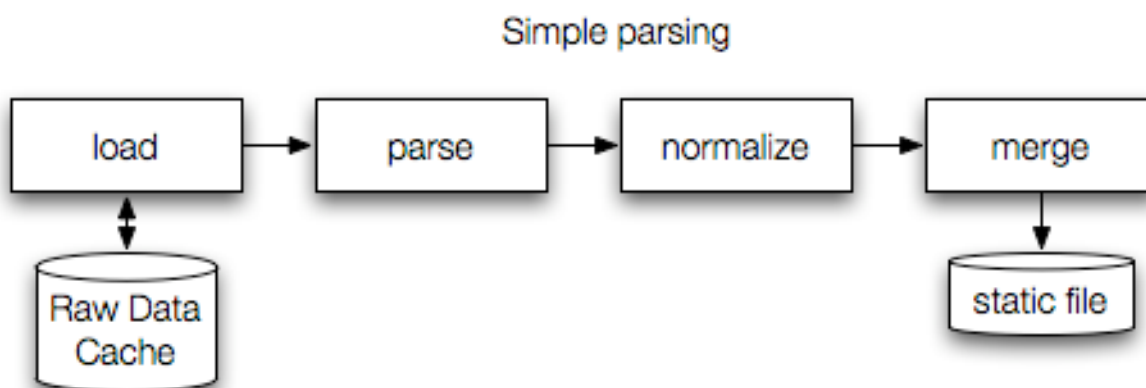
We needed a way for the users to liberate their data from the remote silo without any involvement of that silo, without keeping the users credentials and without us contacting that silo directly. This also needed an extensible architecture to support additional "fantasy league providers". Currently we support Yahoo, ESPN and we're in the process of testing CBS Sportsline. And once we finish that, there are more on the list.

Benchcoach has a huge multi gigabyte database created from multiple sources. There are other non-web feeds, and to give you an idea we have a player id cross reference table with 16 columns. This has a robust, two-step process to normalize incoming data as this process is meant to be run by end users. Needless to say, this is the most complicated.

## Architecture

We use a 4 stage process: **load**, **parse**, **normalize**, and **merge**. While I'm going to write about each stage using **Ruby** tools, there is no reason why you can't do it in **Perl** or **Python** or your favorite language. You will be doing a lot of string manipulation. Many of the tools in Ruby we'll be using are ported or otherwise heavily inspired by their Perl and Python predecessors. If anyone wants to mail me their equivalence I'd be happy to list them below.

Lets first look at the steps to do the simple parsing, and build upon that experience for the more complex cases. The all share a similar overall structure.



Arrows represent how the data is a movin' and a shakin'. Lets jump right into the mess of things, and talk about what you're here for: parsing.

## Parsing

Parsing basically begin and ends with `Hpricot`. The cracks are filled in with regular expressions, so you should study that as well. You need them both, the main structure will be built around CSS/Xpath selectors, which `Hpricot` does, and then the nitty gritty will be cleaned up with `regex`. A common example of this is to pull an idea out of a url embedded in an `<a href>`: you get the element with `Hpricot` but parse the url itself with `regex`.

This stage of the process takes in a stream of characters on the one side and spits out a data structure with the salient information from that stream of characters.

What exactly are you getting parsing? You're going to need to figure out how to pull out the data you are looking for, and that's going to require you to know what it is you are looking at. As a web developer, you probably have an idea of what the web is made of. And, frankly, you'll be wrong; you'll get stuff you don't expect.

### **What, incidentally, are we parsing?**

There is a lot of talk about "structured markup" and "well-formed XHTML" or "the semantic web" and "standard based layout with CSS rather than presentational HTML" and all these things which describe a wonderful world-that-could-be. This is not the world that we live in. These are all great ideas, and indeed the closer the sites are to reifying these ideas the easier your life will be, but that's not how it is.

Your import is constrained by one and one thing only: the page was acceptably rendered in the browser that the person who signed the check was using. Anything after that is bonus. The amount of crap you'll find out there is astonishing. Sites will have multiple K of data in an attribute tag. You are not parsing a semantically marked up document. Sometimes you'll be blessed with some mana from heaven and you'll find some semantic CSS classes or even some microformats, but mostly it's just a series of crufty "good-enough" tweaks done by disinterested people under unrealistic deadlines piled up on top of each other. CSS tags will lie, and represent something different than what you'd expect from the english word that they use because it's often easier to repurpose something than to rename it. No one quotes attributes. Everything is build to work around implementation bugs of IE. Nothing validates. Et fucking cetera. So it's a mess.

As you go deeper into the site, you're trying to reconstruct the templates which were used to render the page. You are trying to figure out how to represent that structure in CSS selectors -- you need to figure out how to reliably identify the slots in the template that get filled in from the remote database. This modeling can get out of hand, or I suppose interesting: after getting into a site you'll develop very clear ideas on the organizational structure of the business entity which build the site. In particular, the relative political strength of the business specifiers and technical implementors, and if the management of the implementors are concerned about "quality" (nurturing) or "results" (assholes). Generally the sites which are the easiest to parse are

technologists-heavy, which is a polite way of saying that they don't really understand the domain nearly as well as they think that they do, and that they include things that no one really gives a shit about for *completeness*. This is a waste of effort on their part, and it is what is going to make your life easier.

Also, if you haven't already, expect to develop some serious contempt for ASP based websites.

Don't worry; if you can figure it out looking at the browser, you can figure out how to parse it mechanically. But you'll have to be clever, and sometimes this cleverness will be on the "how did it end up like this?" variety. Somethings don't lend themselves to clean solutions. However, there is always a solution. But forget about well-formed XML parsers; somethings are so gross that you need to get down and dirty and treat it as one big string which you slice and dice using regex. I'm not going to cover that, because if `tidy + Hpricot` can't deal with your problems, may god have mercy on your soul. (That multi K inside of an attribute problem which makes `Hpricot 0.6` (the latest version at this time) barf. There is a patch but it's not in the distribution yet.)

### **What are you returning?**

You're processing this data and handing it off to the next element on the chain. What format do you put that in? I recommend making a hash of hashes of hashes (of hashes.) I don't think it's a good idea to put it into a model object at this point. First it's a pain in the ass to deal with a model if you have a lot of nested levels. You end up having to add all these convenience methods to add a nested whatever just to make it usable. And if you use hashes, the resulting object is also very fluid as you work on the parser, and I find that its more practical to stay away from the type system as long as possible.

We'll stick it in an object in the normalize stage. In a sense, think of what you're creating right now as more of the params which we will pass to a controller action to validate and store in the database.

### **The Process**

You'll need 4 things open. Only these things open -- seriously. Close mail, twitteriffic, instant messaging, put on the headphones and turn off the the music, and cut down all stimulation. Parsing is not intellectually difficult, but you are going to quickly blow through all of your short-term memory registers. The more pointless little things about the page structure you can hold in your head at a time the easier it'll be.

1. Unit test or RSpec test which loads, parses, and validates the output of the parser.
2. parsing ruby scripts that you're writing
3. irb or script/console to test out CSS selectors and regular expressions.
4. Safari w/ web developer enabled or Firefox w/ Firebug

You're going to be cycling through all of these windows.

1. The first thing you want to do is to get a copy of two example input files, using the process above. We're going to create a corpus to input files and tests for those files. The unit test will run the parser, and make sure that the result is the same. This is useful when you realize a couple weeks later something is broken and you have to muck around with the parser. We're going to write a parser using one file, and then double check it on a second.
2. Start up the console and load up the file. I like to get to the point where I can do `page = Hpricot( data )`
3. Open the file in the browser and select the element that you want to parse and right-click to Inspect Element. You're looking for an enclosing element, or a unique id or css class that will help you navigate the DOM. Generally you'll be looking something which would either let you identify the element you want directly, or something which will give you an array of what you want. For example, here's a shot of Safari on a Yahoo "roster" page, which lists out a bunch of tables representing players on a team:

```

    <div id="datenav" class="roster navlist">
    <div id="startingrosters">
      <div class="startingroster">
      <div class="startingroster">
        <h5>
          <a href="/b1/47073/2">Malpaso </a>
        </h5>
        <a href="/b1/47073/proposetrade?stage=1&mid2=2" class="trade">Propose Trade </a>
        <table cellpadding="0" cellspacing="0" border="0" id="statTable2" class="simpletable">
          <thead>
          <tbody>
          </tbody>
        </table>
      </div>
    </div>
  </div>

```

... > #yspbody > #yspcontent > #yspmain > #startingrosters > div > h5 > a

This is easy to deal with. There's a div with an id of "startingrosters" with a div for each roster with a call "startingroster". Almost too easy. **NOTE:** What you are looking has been parsed and mangled by Safari, **it is not the same html that you are parsing**. What you are seeing (and for Firefox) is the DOM translated back into HTML, not the original source. Very few source files use the `tbody` tag, for example.

4. Type in example code with the console. After we validate that it works, we're going to copy this stuff line for line into the parsing file so make sure that you use sensible variable names **at all times**. None of the one letter crap; it will confuse you very quickly, believe me. So, for example, type `rosters = page/"div.startingroster"`. `rosters.size` is the first thing to check to make sure you have the right stuff.

5. Once you have something that seems to work on the console, copy it into the parser and clean it up. Run the spec. You should print out the returning structure to see how well its doing.
6. Copy the parsing code back into the console and dig deeper. Keeping variable names (what exactly does `col[2]` refer to? Wait, did I shift that array of `trs` to get rid of the header?) sensible and the same is key, and this helps cut down on confusion.
7. Once it works, write a spec for the second file and verify it works. If not, repeat at step 2 and change the parser -- making sure that both tests still run.

Simple! While this example is real, you'll notice that I'm showing you a file from Yahoo. Yahoo is a technology company. Which means good things for you, the parser. You may not be as lucky with a site you need to scrape. Here's an example of the parsing a similar page for CBS. Notice how much sense "SLTables1" doesn't make for a CSS class name. Notice how the list of players is unstructured inside of the table cells, so I need to walk the child element list by hand and look for the BR. I'm not proud, its sort of embarrassing code frankly, but it works.

```
def parse_roster_grid( data )
  page = Hpricot( data )
  ret = {}
  ret[:league_type] = CbsFantasyLeague.name
  ret[:page_type] = :roster
  ret[:remote_league_id] = @url.gsub( /^http:\/\/\/(\d+)\./,
'\1' )
  ret[:teams] = {}

  rows = (page/"div.SLTables1 table")/:tr
  rows.shift

  positions = (rows.shift/:td).collect { |x|
get_position( x.html ) }
  positions.shift

  rows.each do |team_row|
    cols = team_row/:td

    team_cell = cols.shift
    next unless team_cell

    team_info = (team_cell/:a).first
    next unless team_info

    team_id = team_info.attributes['href'].gsub( /.*\//(\d+)\./,
'\1' )
    team_name = team_info.html
```

```

ret[:players_team] = team_id if (team_cell[:img]).first

team = { :name => team_name }
players = []
cols.each_with_index do |box,idx|
  player_info = {}
  box_elems = box.children
  while( elem = box_elems.shift )
    if( elem.class == Hpricot::Text )
      player_info[:position] = 'BN' if( elem.to_s =~ /\(R
\)/ )
    else
      if( elem.name == 'a' )
        player_info[:player_name] = elem.html.gsub( /\n/m,
" ")
        player_info[:remote_player_id] =
elem.attributes["href"].gsub( /.*\//(\d+)\//, '\1')
        player_info[:position] = positions[idx]
      elsif( elem.name == 'br' )
        players << player_info
        player_info = {}
      end
    end
  end
  end
  end
  players << player_info if player_info.size != 0
end

team[:players] = players

ret[:teams][team_id] = team
end

ret
end

```

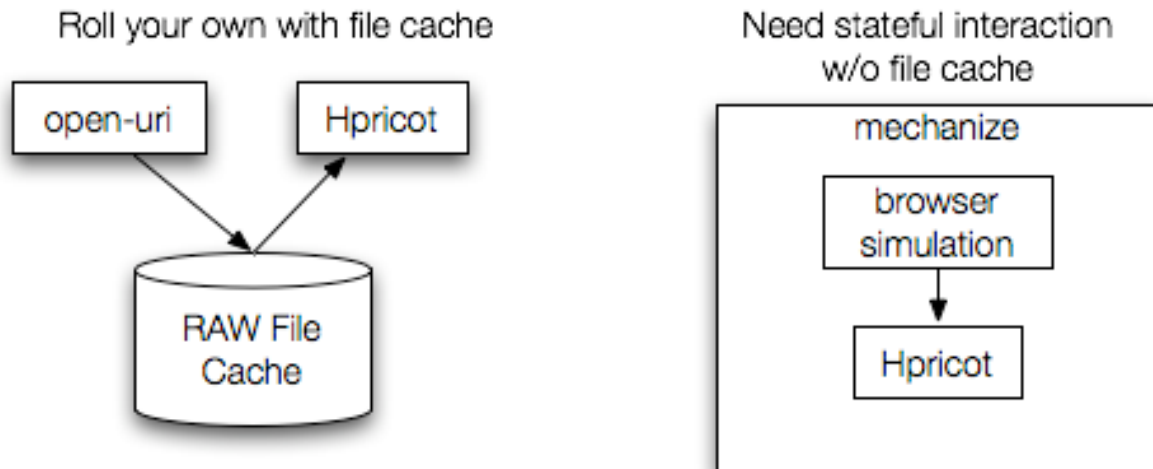
In short, steps 3 and 4 are where the bulk of the brain power is to make this work, and I have no answer for you other than be clever and trying a bunch of things iteratively. In essence, you're trying to figure out the template that was used to generate the html and find sign points.

The `get_position` call helps normalize the data. We're going to talk more about this later, but what you're trying to produce is the sort of parameter map like you would get at the top of a controller action. Don't get wrapped up in "client side validation", we'll do the bulk of that in the normalization process.

## Loading

First things second, you actually need to get the data that we need to parse. I suggest putting all of the actual loading to a specific class, which the other classes call, which will return the data from the cache or load it up again if it's not there. (This also makes it easy to add compression to storing these monster, redundant text files everywhere.) In short, big win in centralizing the caching mechanism.

There are two strategies:



There are a variety of tools you can use for this. I will talk about two. The first is a little more complicated, but it lends itself to caching the files which is incredibly useful. The other lets you maintain session state and cookies when you load up a page, which you need to do when scraping things that require logon. They both use `Hpricot` under the hood.

- `open-uri`. Built in Ruby. Can pass in `{ "If-None-Match" => etag }` or `{ "If-Modified-Since" => date }` to do conditional loading.

```
require 'open-uri'
```

```
mtime = File.mtime( filename ).rfc822
```

```
Timeout::timeout( 30 ) do
```

```
  open( url, { "If-Modified-Since" => mtime } ) do |f|
    data = f.read
  end
```

```
  open( filename, "w" ) { |f| f.puts data }
end
```

The Timeout class does what you'd expect: if the enclosed block don't complete within a set period of time, in this case 30 seconds, it raises an exception. Depending upon the exception you can either retry the file (say a network timeout), move on to the next time (file not found) or abort the whole process (an exceptional exception perhaps.) Regardless, be sure to have it log and mail you when something unexpected happens.

- `mechanize`. Ruby gem. If you need to maintain cookies, for example you need to login to the site before getting access to the data, this is what you want.

```
require 'mechanize'
agent = WWW::Mechanize.new
agent.user_agent = 'Mac Safari'
page = agent.get( PROTECTED_URL )
loginform = page.forms.with.name("login").first
loginform.id = USER
loginform.password = PASSWORD
page = loginform.submit
page = agent.get( PROTECTED_URL )
# Data is now available in page.body
```

You'll want to store the raw version somewhere. This is logically more difficult when using `mechanize` because you're talking about stateful transactions, but the ability to replay the data later, or use it in unit tests is the only way you can hope you build a robust system. For something simple, you might be able to get away with not doing this, but once you find yourself in a situation where you're hitting the remote website everytime you are running a test you'll find that it's faster and more polite to load it from a local disk.

Sometimes the data you get back from the wild is a mess. Often its a mess. When I get to a site that blows up the parsers most of the time the solution is to pass the raw file through `tidy`. (You'll need to get the latest version from CVS.) If you end up going down this route, you'll be happy that you routed all of your loading calls to the raw data store; keep storing the unprocessed data and then pass it though `tidy` on the return call. (Always store unprocessed data. This will let you late-bind the decision to add or remove `tidy`.)

It's useful to be able to call these commands from `irb`. Class methods which make it easier to load up a file remotely, which pass it through `tidy` if need be, make it much easier to test. Contrariwise, when you have to instantiate an object and configure all these attributes before you can get it to load up anything means that you'll be copy-and-pasting a lot when you should be trying to figure out why the page is blowing up. Most of my classes have methods which are only to be run manually.

### **Storing: meta data and name coding**

In addition to the data, you need to store metadata. Question you may need to answer:

- I'm looking at database object 37. This can't be right. What source file did the data come from?
- What's the "modified" date of this file? What is the ETag of this file?
- What url did the file come from?
- What files did a particular user load?

It's easiest if you put this stuff in the name of the file, or insert it on the first line. For one site I name things like

```
#{CACHE_DIR}/#{CLASS_TYPE}/#{REMOTE_ID}
```

and another I put the url in the first line of the file and name the files:

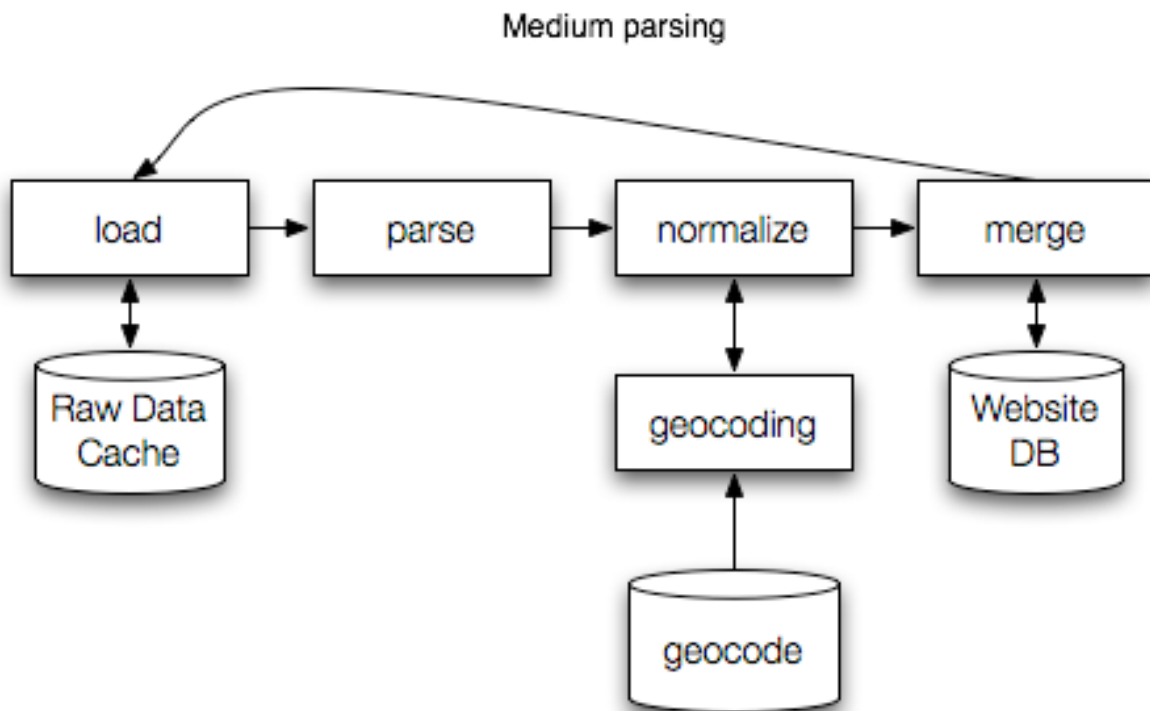
```
#{CACHE_DIR}/#{USER_ID}_#{TIMESTAMP}
```

### A word on compression

```
$ bzip2 -v 8_1210609885
```

```
8_1210609885: 5.744:1, 1.393 bits/byte, 82.59% saved, 66134
in, 11514 out.
```

## Architecture: Medium



Now that we've loaded and parsed the file, we need to clean up the data to make sure that it makes sense. This means that we need to match up remote ids with local ids, fill in any missing slots from other data sources, and additionally load up any further data that was referenced.

## Normalize

We've loaded the data into a structure; now its time to fill in the blanks. There are two goals here: first is to make sure that all lookup values are consistent within our system, and that we pull in any other information from other sources that we need to make this complete. We've written unit tests for the parser, so we can tell that the translation from source file to our object representing the data is correct. Now we need to make sure that the data actually makes sense.

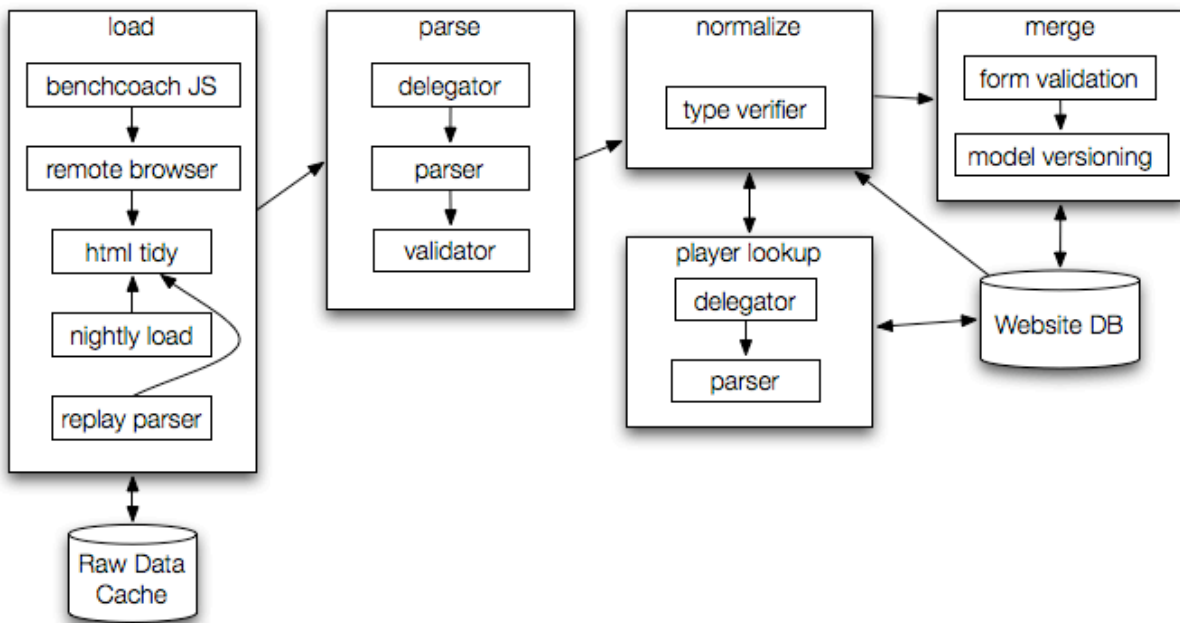
This process is straightforward. You need to maintain either a lookup-hash in the ruby code, a cross reference table in the database, or you need to query a service to do this lookup. Fields that have a one to one mapping are easy to translate; and it's often easier to do this "runtime" during the paring process. For example, if you are pulling from two places, and one thing spells "BENCHED" as "Bn" and the other spells it as "(R)" doing a lookup in a hash is the way to go. This is the only case in which we're not preserving a "remote" value along side of a "local" value; otherwise we need to pass the remote value straight through to the database.

Maintain lookups of this sort don't scale well at all. At a certain point you need to store the remote value in the database and make the lookup a two step process. And indeed, maintaining a list of mappings can be another scraping job itself. Mapping neighborhood ids or player ids is like this. In this case you build a "id cross reference table", which you regularly refresh and populate with the mapping. The `player_xref_id` table for benchcoach.com links together benchcoach, STATS INC, MLB, Baseball Info Solutions, ESPN Sports, Yahoo Sports, CBS Sportsline, Reference Databank, Lahman amongst other IDs together for major and minor-league baseball players. There is a separate process to lookup and link a player for each of the remote databases.

The third and least satisfying situation is when the remote data isn't enumerable before hand. The site of the mp3 file for a podcast needs to be looked up only after because we just don't know what file is there. The diagram above is showing the process for menumaps, which needs to look up a latitude and longitude for each address. Since the data that we get coming in isn't bounded in any useful way (the size of the set of "all addresses" is much larger than the number of addresses we are likely to see) we need to do the lookup as a batch process *after* the load.

Doing a HEAD request to lookup the size of a file of quick; looking up tons of GIS data is not. You'll get much higher throughput if you save in the database with "holes" inside, and then you need to fill those holes in. However, the application which in turn uses this data then has to deal with partial information, like having a listing for a restaurant that you don't know where it is.

## Architecture: User Submitted Content



This diagram shows the way that the synchronization process works between benchcoach.com and the various fantasy team providers. This import framework makes it possible to add additional URL patterns that we accept, so we can continually be adding more and more sites. Currently we parse 18 different urls into 5 different “types” of data from 3 different sites. The remote pages are mapped onto a common abstract type, which is in-turn mapped onto a complex rails model which is stored in the database. It handles version, rollbacks, replay sinks forward, and supports a “time machine” view of the past. We support all browsers (except very old versions of Konqueror), don’t require any user credentials and never contact any remote site directly.

### Load: The magical bookmarklet

We’re going to build on the ideas of the loader that we use before. Instead of using mechanize to simulate the browser and session state, we’re going to use the user’s actual browser, let them go to the remote sites as they normally would, and send what they are looking at on their screen straight back to us.

#### Bookmarklet

There’s a pseudo-form of a URI which starts with “javascript” instead of “http”. It expects there to be javascript code on the right side of the colon which is executed in the context of the current page when the link is clicked. This makes the most sense if you add it to you bookmark bar, so it shows up as a button in the toolbar throughout your browsing experience. The neat thing about this is that since the code is only run as a result of the users action, it runs in a privileged the context of the current page. If we tried to open up a window, or use a frame or something, the javascript security

model would prevent us from access the contents of this file. The user-triggered bookmarklet doesn't have these limitations.

Since there's a limit to the amount of code that you can install in the user's browser, the best technique is to keep it small have it load the bulk of the javascript from your production site. This has the added advantage that you are able to push out new version of the code without having anyone reinstall anything.

The code:

```
<a href="javascript:void((function(){var
%20e=document.createElement('script');e.setAttribute('type','tex
t/
javascript');e.setAttribute('charset','UTF-8');e.setAttribute('s
rc','http://benchcoach.com/importer/
bookmarklet');document.body.appendChild(e)}})());"
onclick="window.alert('To install bookmarklet, you need to
right-click and choose add favorite.');"return false;">Benchcoach
Sync</a>
```

The only caveat is that the javascript in the link must return false -- otherwise the page content will be replaced with the return value.

You can follow along with the code our bookmarklet then runs in the browser by going to <http://benchcoach.com/importer/bookmarklet>

The code first checks to make sure that we are on a supported site by examining the url. If we are, we edit the DOM and insert a form which has two parameters: the current page url, and encoded full data of the page. (We also add a "loading" div on top.) We then submit the form to our page\_parser on the production site.

(I stole this trampoline technique from [ffffound.com/bookmarklet#bookmarklet](http://ffffound.com/bookmarklet#bookmarklet) but I'm sure that there are others.)

We initially were looking into using either a greasemonkey script or an ActiveX control but this has the advantage of very easy installation and that it works on almost all browsers.

## **Parse: delegation & type**

When we receive the data, decode it user `CGI::unescape` to get the raw data and save it to the raw data store. Again, I can't stress how valuable saving the preprocessed data is.

We then submit the url to all of the processors to see if someone can handle it. If they can, we pass it through and then run the result through the type-verifier. At this point we have a hash which should contain what type it is, the remote id, the user, etc -- all of the

data that we need for whatever. We reuse the verifier code that we have in the normalizing phase for each imported file and send mail back to production support of any of the tests fail.

Delegation for the data going into the parser is to figure out which URL you are parsing; these obviously are segmented by site but really they are by site template and in practice multiple URLs use the same or very similar templates. Delegation on the verifier level cares only about types, not the source of data, and its there to make sure that all the data about to be processed is correct.

The action will basically switch on the type which attempts to load what data it can to a form for the user. This is why we think of the hash returning from the parser as the params hash of the action: the parsing stores nothing in the database and only shows a form so that the user can manually verify that the import is correct. We expect that no user actually does this, and pushing the bookmarklet again is configured to auto “confirm” any pages. This way you basically double click to get the page in.

Once that page is submitted, you’ve effectively channeled in all the remote data into your regular forms. We were able to reuse most of our existing code to create things by hand using this technique of pulling in the remote data. We had to add some confirm screens when we were loading in data that used cross-referenced ID to deal with the case when a remote side adds something before we manage to sync.

The verifier is the missing link to keep refine your parsing process. As you add more and more constraints into this as the remote sites add new attribute values you’ll be able to catch them as they occur. And indeed, once you find any problems, you pull the raw file from production, put into your test suite, and then fix the parser.

It takes about two days to tune a site once your importer is live using this process.

## **That’s it. Simple.**

That’s basically all you need to know. Any comments, questions or suggests to [wschenk@gmail.com](mailto:wschenk@gmail.com) and I’ll update this.

## Tools we use

	What	Where	Why
<b>ruby/irb</b>	The language	<a href="http://www.ruby-lang.org/en/downloads/">http://www.ruby-lang.org/en/downloads/</a>	The main advantage in this case is that you have an interactive shell, which allows you to interactively test out how well you selectors are working. (Obviously, this is not limited to ruby.)
<b>Hpricot</b>	Html parser from ruby mad-genius _why	<a href="http://code.whytheluckystiff.net/hpricot">http://code.whytheluckystiff.net/hpricot</a>	An amazing html parser which, amongst other things, does CSS selectors and just just retarded XPath.
<b>tidy</b>	Program from W3C which cleans up messy HTML	<a href="http://tidy.sourceforge.net/">http://tidy.sourceforge.net/</a>	Once you look inside enough pages you really start to be amazed at the sort of crap your webbrowser can deal with. This helps the extreme cases that confuse Hpricot. (ESPN, I'm looking at you.)
<b>mechanize</b>	Ruby gem which allows you to interact with a remote site as if you were a browser	<a href="http://mechanize.rubyforge.org/mechanize/">http://mechanize.rubyforge.org/mechanize/</a>	Sometimes you need to login.
<b>firebug</b>	Fantastic web developer tool for firefox	<a href="http://www.getfirebug.com/">http://www.getfirebug.com/</a>	This is useful in many ways, but we will mainly be using "Inspect Element" to explore our CSS selection options.
<b>safari develop menu</b>	Something similar to firebug for Safari	In Terminal, run <code>defaults write com.apple.Safari IncludeDebugMenu 1</code>	After a while, your eyes get tired and you need to mix it up with another browser.

## Can I do this?

In a word: easily.

In a paragraph: Multi-billion dollar business are based off of web-scraping; this is precisely what we are going. Google's stock price is, as of May 10, 2008, is well over \$500. Yahoo and Microsoft are also huge players in the search market. There's no difference between what they are doing and what I've described above. A lot of sites I've seen have a "terms of service" document which disallow automated scraping. Yet I found that site from a Google search to begin with. Go figure. There are a few take aways from this:

1. No one can own a fact. (e.g. the address of Minca is "536 E 5th St, New York, NY 10009") You can own a database, but not the facts in that database.
2. You own your information, not a company which is holding it for you. (e.g. it's your mail on gmail.com)
3. Programmatic instructions in robots.txt supersedes instructions from some random url.
4. **If you drive traffic to their site, no one, especially the company receiving the traffic, pays attention to their legalese. No one sues search engines even though all of them violate the terms of service. Do things to drive traffic to their site and they will love you; do things to shrink traffic to their site and they will fight you.**

In conclusion: I am not a lawyer, and if you are concerned about the legality you should ask one. However, in practical terms there's no way for them to prevent you from doing this and if you think about what you are doing there's likely a way to make it a mutually beneficial arrangement. Which means that everyone wins.