The Meta Google Maps “Hack”

Petros Efstathopoulos, Mike Mammarella, Alex Warth
University of California, Los Angeles
{pefstath,mikem,awarth}@cs.ucla.edu

Motivation

Since the advent of Google Maps, a slew of developers have built systems which use information from other sources to produce composite maps showing the locations of various items. These systems are commonly referred to as “Google Maps Hacks” due to the fact that they were originally created before Google published the API for creating these systems. The example system we will be considering throughout this paper is one which shows the locations of movie theaters on a map, given a ZIP code.

Since the publication of the Google Maps API, even more applications built around this interesting framework have surfaced on the internet. All of these applications have several properties in common: they fetch their data from somewhere on the internet, they process it in some way (perhaps parsing it out of some human-readable format), they combine it with new formatting, and finally they display it on the map. In addition, since the Google Maps API deals only with raw coordinates, all these systems must perform “geocoding” to convert the addresses into (latitude, longitude) pairs.

All this is a lot of work even for seasoned programmers, and a fair amount of it is nearly identical for each such application. Additionally, any time the Google Maps API changes (which it has on several occasions), all of the existing systems break until they are manually updated. Finally, there is no way that a general internet user can accomplish it without first learning a great deal of programming.

Our goal is to create a system that can automate the generation of many simple “Google Maps Hacks.” Not only will this simplify the task of creating useful Google Maps-based applications for programmers, but it can allow ordinary users to do it as well. Fully realizing this second property will require striking a careful balance between the ease of use and the expressiveness of our solution, as we will explain later.

Solution

The architectural goals of our system are the following:

- *P*ractical UI. The user interface must be easy to use yet descriptive enough to allow users to model generic Google Maps Hacks.
- *G*eneric parsing. The system must be able to automatically generate specialized parsers that implement user provided parsing/extraction rules.
- *O*utput formating. We want to provide the user with the ability to customize the way the extracted data will be presented. In effect this means that we want to be able to combine the extracted data with user provided templates and display it in the ”info bubble” of each point on the map.
- *P*ortability. We want our application to generate a set of scripts that are self-contained. This means that the final result of our system is a set of programs and scripts that can be downloaded and and run on almost any web server with no additional modifications. As we will see the “geocoding” requirements of the Google Maps API make it impossible to achieve complete portability.

The first step in any Google Maps Hack is to identify the source of the data to be displayed. In general, to be useful to the end user, the source of the data will support some sort of query. In the movie theater example, this is a ZIP code. Given this query, we construct a URL from which the desired data can be fetched, and the resulting content is passed to a parser. (See Figure 2 for an example.)

Parsing

The parser is a critical piece of the system. Writing a custom parser for each data source would be a lot of work, and it would defeat much of the purpose of this work. On the other hand, the sheer variety of possible data formats seems to almost necessitate a completely general parser, since the input can be virtually anything. Fortunately, most useful data sources on the internet are automatically generated from some backing database, and thus exhibit very regular formatting. This is perhaps especially the case for the sorts of data that most users will want to plot on a map. Therefore, we can simplify the problem dramatically, and we reduce the parsing problem to a small set of "actions":

- `match` matches and skips an exact string in the input.
- `skipSpace` matches and skips any amount of whitespace in the input.
- `skipUntil` advances to and skips some occurrence of an exact string in the input.
- `readUntil` reads the input until some occurrence of an exact string in the input, storing the data read into a named variable.

The user creates a "pattern" consisting of a list of actions, and the pattern is matched to the input document in a loop to read all instances of the described entities. The parser can also perform backtracking: `skipUntil` and `readUntil` can both be thought of as trying to find the soonest match they can, but with the capability to try all possible matches to find the first which also allows later actions to complete successfully.

Additionally, we might want to parse some amount of nested content as well. For instance, a movie theater has some list of movies that it is currently showing. Potentially each of these movies even has a list of showtimes. To support this, at the end of a pattern, the parser will try all appropriate patterns as the next pattern to match in the input. Whichever pattern matches sooner in the input will be chosen as the next pattern. For example, after reading the name of a theater from a list of theaters and movies being shown at each, the parser will attempt to parse a movie and another theater. The assumption is that all the movies for the theater just parsed will occur before the next theater, and so we will parse them before skipping them to parse the next theater name.

In the future, we imagine that instead of writing extraction rules manually, the user may be able to interactively mark sections of a page as the desired content, and the system could generate the parsing rules automatically by iteratively refining a set of rules to match the marked sections, similar to the ideas presented in [2];

After the data has been parsed and the desired information has been extracted and assigned to variables, the next step is to reformat the data into a format appropriate for the pop-up bubble on a Google Map.

Reformatting

The reformatting system we implemented is very similar in nature to a "mail merge," which is a common feature in office software suites. Basically, the user enters a template involving references to fields from the extracted data, and the references are replaced with the data from each extracted item. However, there is a twist: in a mail merge, each record is flat, and thus the template can list each of the fields exactly. Our extraction mechanism can generate nested records, and thus the templates need to be able to refer to the nested elements as well.

To handle this, we essentially allow for a nested mail merge. Each template is split into three parts: a header, a nested list template, and a footer. All three parts can contain references to the record being reformatted; however, the nested list template is processed once for each nested item and can contain references to not only the parent item but to the child items as well. In the movie theater example, we might write a template that looks like this:

```
Tab title:     Theater
Header:        <h1>$THEATER</h1> $ADDRESS<br> <br> Now showing:<br>
List template: <b>$MOVIE</b><br>
Footer:        <br> End of movie listings.

Tab title:     Movies
Header:
List template: <b>$MOVIE:</b> $SHOWTIMES<br>
Footer:
```

Which would, on the first tab, list the theater name and a list of movies playing, and on the second tab, list each movie and its showtimes. The appearance of these elements in an actual Google Maps bubble is depicted in Figure 1.

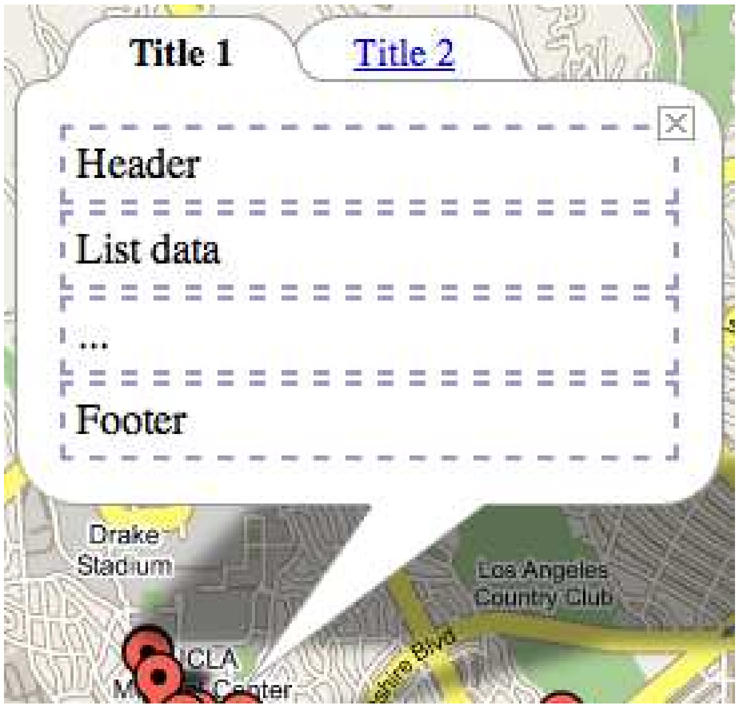
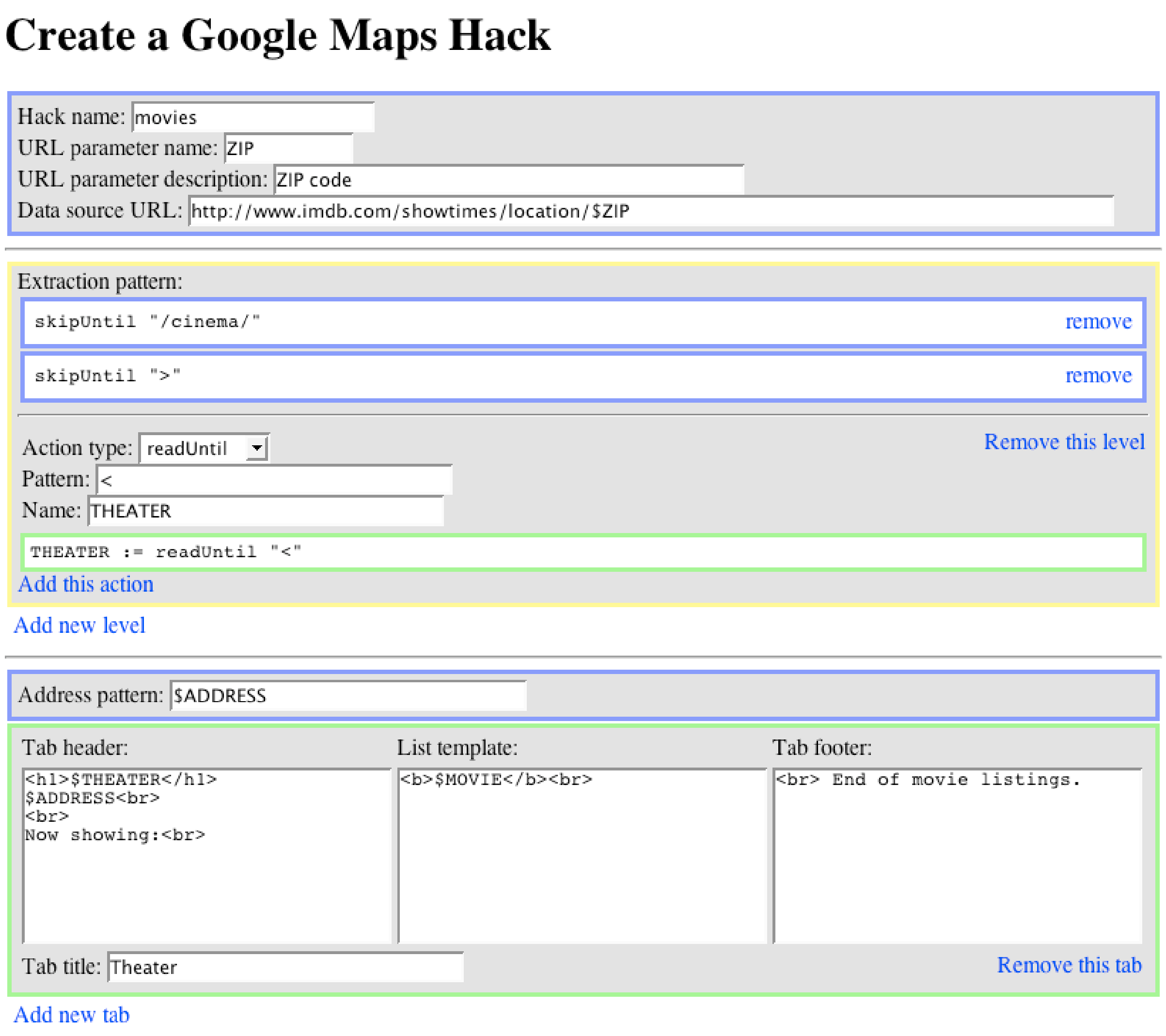


Figure 1: The anatomy of a "bubble"

User interface

To enter this information into the system, we have a web-based user interface implemented largely in Javascript. The extraction rules are selected using a wizard-like element, and can be reordered by dragging them. Extraction nesting levels can be dynamically added and removed, along with all associated rules. Similarly, tab templates can be added and removed dynamically.

The rules and tab appearance can easily be tested during development of a new "hack" by generating the hack and trying it out, since the hacks include debugging modes which allow the user to see what the extracted and reformatted data looks like before being put on a Google map.


Create a Google Maps Hack

Hack name: movies
URL parameter name: ZIP
URL parameter description: ZIP code
Data source URL: http://www.imdb.com/showtimes/location/$ZIP

Extraction pattern:

```
skipUntil "/cinema/"
skipUntil ">"
```

Action type: readUntil
Pattern: <
Name: THEATER

```
THEATER := readUntil "<"
```

Address pattern: $ADDRESS

Tab header:

```
<h1>$THEATER</h1>
$ADDRESS<br>
<br>
Now showing:<br>
```

List template:

```
<b>$MOVIE</b><br>
```

Tab footer:

```
<br> End of movie listings.
```

Tab title: Theater

Figure 2: The user interface for creating a Google Maps Hack

The expressiveness of the extraction language is actually limited by this user interface: actions can only be combined sequentially. In reality, they can be combined with boolean operators as well as the sequential operator, as described in a later section. However, we have found that most realistic data can be extracted with only the sequential combination capability.

The appearance of the user interface is shown in Figure 2. Here the user has started entering in some extraction rules for theaters, with IMDb as the data source. Additionally some of the tab templates above have been entered.

Result: the Meta Google Maps Hack

Generated Code

Once the description of a Google Maps Hack is entered into our system's generation user iterface, several files are generated:

- `go.php`: This is the main script for a Google Maps Hack: it takes as an argument the value of the URL parameter (as described in the previous section), and carries out the process shown in Figure 3 by executing the programs `wget` (a program that downloads webpages), `parser`, `convert.sh`, and `add2coord.pl`, in sequence. A detailed description for each of these programs is included below.

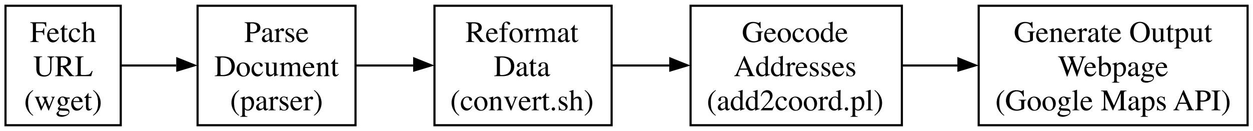

Fetch URL (wget)
Parse Document (parser)
Reformat Data (convert.sh)
Geocode Addresses (add2coord.pl)
Generate Output Webpage (Google Maps API)

Figure 3: Overview of a Google Maps Hack generated by our system

- `parser`: This is the parser which will be used to extract the useful information from the document. This program is generated in two stages: (*i*) source code for the parser is generated using a (very clever!) technique called *monadic parsing* [5], and then (*ii*) this program is compiled using `ghc`, a high-performance Haskell compiler.

- *format files for* `convert.sh`: `A`*n*, `B`*n*, and `C`*n*: These files store the tab reformatting information described in the previous section. `convert.sh` is described in detail below.

The following scripts are used by all Google Maps Hacks generated by our system (and thus are not generated for each new hack):

- `convert.sh` This script implements the reformatting feature. It reads the `A`*n*, `B`*n*, and `C`*n* files mentioned above, reads takes the parser's output, and generates the HTML content to insert in the Google Maps bubble tabs by replacing instances of `$VARIABLE` with the value that the parser rule assigned to the variable named `VARIABLE`. Additionally, as described earlier, it processes nested parser output by using the nested list template once for each nested item.

- `add2coord.pl` This is the part of the system that converts addresses into coordinates (i.e. latitudes and longitudes). This conversion is necessary because (rather strangely) the Google Maps API has absolutely no support for mapping or placing push-pins on addresses—all it “understands” are geographical coordinates.

Putting pins on the map

Geocoding

One of the unfortunate “features” of the Google Maps API is that it can only “understand” geographical coordinated in the form of (latitude, longitude) pairs. This implies that all Google Maps Hacks will have to convert the addresses of the locations to be displayed on the map to coordinates. It turns out that this is not as easy as one might expect, though there are numerous online services that will perform such a conversion for a reasonable price.

The most efficient way to do it is use information available from the U.S. Census Bureau TIGER system [1], in conjuction with the Geo-Coder-US perl library from CPAN [3]. The Geo-Coder-US issues queries to a geocoding database for a given address and returns the coordinates of that address. We used the data downloaded from the U.S. Census Bureau (data version ”2004 - second edition”) to build such a database. We downloaded approximately 4GB of (per state) topological data and used it to build a 800MB database to be used by Geo-Coder-US. The above mentioned `add2coord.pl` script is essentially a wrapper to the Geo-Coder-US library.

Google Maps API

The Google Maps API [4] is mostly implemented in Javascript. Although its usage for simple applications is straightforward, it is not clear how easy it is to use for more complex implementations. A lot of the underlying Javascript code is not visible to the user (which in some cases is a bad thing) and interface is very poorly docmented. Of course the Google Maps API is still in beta phase, but we were somewhat disappointed by its current status. The lack of geocoding is very unfortunate and the support for Ajax did not prove to be useful in this project. Aesthetically the API is excelant and that makes up for a lot of the missing functionality. It is interesting to note that the competing API by Yahoo! is richer in features and provides its own geocoding.

An example hack

The example we have been dealing with so far, movie theaters, is not merely a toy example. It is a real hack we have generated with our system, and the information required to generate it has already been largely discussed. Figure 2 shows some of the actions and templates needed to generate it. The complete definition is as follows:

```
URL parameter: ZIP
Data source URL: http://www.imdb.com/showtimes/location/$ZIP

Extraction rules:
  Level 1:
    skipUntil "/cinema/"
    skipUntil ">"
    THEATER := readUntil "<"
    skipUntil "<td class=\"address\" colspan=\"2\">"
    skipUntil ")</a>"
    skipSpace
    ADDR := readUntil "<"
    skipUntil ">"
    ZIP := readUntil "<"
  Level 2:
    skipUntil "<td class=\"item\" width=\"50%\">"
    skipUntil "<b>"
    MOVIE := readUntil "</b>"
    skipUntil "<small>\n<b>"
    skipUntil "</b>"
    TIMES := readUntil "<br>"

Tab definitions:
  Tab 1:
    Title:         Theater
    Header:        <font size=+2 color=blue>$THEATER</font><br>
                   $ADDR $ZIP<br><br>
                   Now playing:<br>
    List template: <nobr><b>$MOVIE</b></nobr><br>
    Footer:        <br><form action="http://maps.google.com/maps" method="get" target="_blank">
                   <i>Your address</i>:<br>
                   <input type="text" name="saddr" size="20" value="">
                   <input type="hidden" name="daddr" value="$ADDR $ZIP">
                   <input type="submit" value="Directions">
```

```
                </form>
  Tab 2:
    Title:        Movies
    Header:
    List template: <b>$MOVIE:</b> $TIMES<br>
    Footer:
```

Note that the footer of tab 1 here has an HTML form in it, which, when submitted, will open a new page with Google Maps directions in it from the user's entered address to the address of the theater!

In a similarly easy manner, we could add links to movie reviews on other sites, by using the "search for reviews" feature on those sites. Alternately, we could adjust the extraction rules to extract not only the title of the movie from IMDb but also the link surrounding it, which itself is a link to a review.

Conclusion

We believe that we have created a genuinely useful system. Probably the best validation we have comes from the fact that several people have already asked us for the URL of our sample application (the movie theater example), because they are interested in using it! Similar applications take very little time and effort to implement using the Meta Google Maps Hack, and we believe that as the number of applications created by our system increases, so will the number of users, because our project will have more and more visibility.

Shortly after our presentation in class, a friend of ours started thinking about building a Google Maps Hack that displays the locations of registered sex offenders in Ohio using the Ohio Public Sex Offenders data which is publicly available.

Another friend has commissioned us to implement a Google Maps Hack which displays the locations of *Hooters* restaurants in a particular area. What can we say? Apparently some people just love chicken wings.

Future Work

There are several aspects of our implementation we would like to "polish up" before making the Meta Google Maps Hack publically available. Some of these are small issues, such as making sure the system works with most of the widely-used browsers without any hiccups (certainly Firefox, Safari, and Internet Explorer). Still, these "small" issues are certainly important (this one would have prevented our demo in class from turning into a horrible fiasco!).

Another perhaps more significant improvement would be to require a Google Maps API key as a parameter to the creation interface. Currently, all hacks created with our system use our own key, which would certainly become problematic once the general public starts using the Meta Google Maps Hack!

Also, it would be useful to improve the geocoder we use: we have noticed that Google's geocoder is much better (it is able to find many addresses ours cannot). Another limitation of our geocoder is that it only works for U.S. addresses, which makes our system of very limited use to international users. Another solution to this problem might be to package our system as an application that can be run locally by users. This would enable us to make the geocoder a plug-in, instead of forcing users to use ours. Meanwhile, the U.S. Census Bureau released the "2005 - first edition" version of TIGER which we are currently downloading to build a new version of our address database.

Another possible area for improvement is the parser. We believe that we have only scratched the surface of the problem. Although our abstraction (`match`, `skipSpace`, `skipUtil`, and `readUntil`) has worked farily well on the examples hack we have implemented, we realize that looking for a better, perhaps much higher-level abstraction could add immense value to our system. As mentioned previously in this paper, the back-end system we have built for our parser is a general-purpose one, and allows for much more sophisticated parsers to be implemented than we currently allow through the creation interface. The right abstraction would allow users to take advantage of as much of the expressiveness of our framework as possible, while requiring very little or no prior programming knowledge. It would be interesting to explore a direction in which the user interface for creating the parser would consist of selecting and naming regions of pages, and automatically synthesizing the appropriate parser rules to extract that data.

References

[1] U.S. Census Bureau. Topologically Integrated Geographic Encoding and Referencing, 2004, second edition. http://www2.census.gov/geo/tiger/tiger2004se/.

[2] Valter Crescenzi, Giansalvatore Mecca, and Paolo Merialdo. RoadRunner: Towards Automatic Data Extraction From Large Web Sites. VLDB, 2001.

[3] Schuyler Erle and Jo Walsh. Geo-Coder-US CPAN module. http://search.cpan.org/ sderle/Geo-Coder-US/.

[4] Google. Google Maps API. http://www.google.com/apis/maps/.

[5] Graham Hutton and Erik Mejier. Monadic Parsing in Haskell. *Journal of Functional Programming*, 8(4):437–444, July 1998.