DAVID N. BLANK-EDELMAN

practical Perl tools: en tableau

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), newly available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs.

dnb@ccs.neu.edu

A FRIEND CAME TO ME A WHILE BACK with a problem. He had just purchased an iPhone and needed a way to get his address book from his old phone into the new one. His old phone had software that would let it sync the address book information to a service provided by the carrier. That carrier, let's call them "rhymes-with-horizon" to avoid naming names, hadn't engineered their service to make it easy to take your data with you. There was no "download your address book" (or "export as CSV") feature. At best, they offered a Web interface where you could view and edit the data to a certain extent.

But a Web interface is better than nothing, because if we can see the data in a Web page, we can probably scrape it and return it to its rightful owner. The tricky thing here is the Web page they provided is kind of yucky. The data is embedded in a huge table and there's lots of other markup goop and JavaScript throughout. A simple cut-and-paste won't work for my friend. To get some idea of what I mean, Figure 1 shows a portion of what the table looked like in the browser (with the names and phone numbers changed).

Name	Phone Number	Email
Charlie Parker	**Mobile**2996209109	
Coleman Hawkins	**Work**5834800077	
Hank Jones	**Mobile**2692315826	
Ray Brown	**Home**7372450564	
Lester Young	**Mobile**6633158411	
Bill Harris	**Mobile**6391737453	
Harry Edison	**Mobile**9987145662	
Ella Fitzgerald	**Mobile**2097688862	
Max Roach	**Mobile**5245232003	**Email**maxroach@gmail.com

FIGURE 1: DATA AS RENDERED IN THE BROWSER

To make it a little more legible, Figure 2 (next page) is what it looks like if I outline the table cells using the Firefox Web Developer add-on:

	Name	Phone Number	Email
☐	Charlie Parker	**Mobile**2996209109	
☐	Coleman Hawkins	**Work**5834800077	
☐	Hank Jones	**Mobile**2692315826	
☐	Ray Brown	**Home**7372450564	
☐	Lester Young	**Mobile**6633158411	
☐	Bill Harris	**Mobile**6391737453	
☐	Harry Edison	**Mobile**9987145662	
☐	Ella Fitzgerald	**Mobile**2097688862	
☐	Max Roach	**Mobile**5245232003	**Email**maxroach@gmail.com

FIGURE 2: OUTLINING TABLE CELLS

And that's where we'll pick up the story for this edition's column. In this column we're going to look at an approach for extracting data from even ugly HTML tables. Given how much information is now presented to us in HTML tabular form, it is generally useful to know how to grab the data and work with it on your own terms. In a previous column we looked at the WWW::Mechanize module for navigating Web sites and retrieving certain content. In this column, we're going to assume you've already retrieved the HTML document containing the table of interest (perhaps using WWW::Mechanize) and you now need to process its contents.

There are a number of ways we could approach this problem. We could shred the document using a set of complex regular expressions, but that's no fun at all. It would be a better idea to treat the HTML table like any other HTML and use some of the general-purpose HTML parsing modules like HTML::Parse and HTML::TreeBuilder. Those modules make it much easier to find the <tr> and <td> elements in the document and proceed from there. But probably the best tack we could take would be to use one of the specialized table parsing modules to do the heavy lifting, so that's what we'll do here.

Using HTML::TableExtract for Basic Data Extraction

Regular readers of this column (you know, the ones that have bought all of my albums and have the set of well-worn Practical Perl Tools tour t-shirts) might recall that I'm a big fan of HTML::TableExtract. We'll start with that module and then head into some more advanced territory.

The first step after loading HTML::TableExtract is to specify which table in the document should be considered for extraction. HTML::TableExtract offers several ways to specify the table: the two most commonly used ones are by table headers and by depth/count. With the first method you initialize an HTML::TableExtract object with the names of the column headers you care about from the table in question:

```
use HTML::TableExtract;
my $te = HTML::TableExtract->new(
        headers => [ 'Name', 'Phone Number', 'Email' ] );
```

When we ask the module to parse the data, it will attempt to find all of the tables with those headers and retrieve the data in those columns for every row in those tables.

This usually works quite well, but sometimes you encounter tables that don't play nice with a header specification: for example, tables without any labeled

headers. In those cases HTML::TableExtract lets you specify a depth and count to identify the table in question. "Depth" refers to the level of embedding for a table. If the table is not embedded in any other table, it is at depth level 0. If the table you care about is in another table, that would be depth level 1. Once you establish depth, you then provide an instance number to point at the specific table (both depth and count start at 0). For example, the second table on a page would be depth => 0 and count => 1. The first embedded table in the first table in the document would have depth => 1 and count => 0. These numbers are set in a similar fashion to the headers:

```
my $te = HTML::TableExtract->new( depth => 1, count => 1 );
```

Our sample document has identifiable headers, so our program will start off like the first sample above. We can then perform the actual parse of the HTML file like so:

```
$te->parse_file('contacts.html') or die "Can't parse contacts.html: $!\n";
```

Now our object (if the parse succeeded) will let us query the tables matched and retrieve all of the rows in those tables:

```
foreach my $table ( $te->tables ) {
  foreach my $row ( $table->rows ) {
    print '|' . join( '|', @$row ) . '|' . "\n";
  }
}
```

Usually at this point we're home free, because the information in the table is sufficiently simple that the extraction yields the data we need. But, alas, with our sample document we get stuff that looks like this (I've removed a bunch of whitespace to save magazine trees, but you get the idea):

```
|
        Charlie Parker
          |
            Mobile2996209109
          |<A0>
          |
```

Yucko.

More Advanced Data Extraction with the HTML::Tree Family

Basically, each table cell in our example has a bunch of whitespace and who-knows-what in it, making for a very messy extraction. Here's a snippet of the HTML found in a table row with the whitespace stripped and the elements indented for readability:

```
<tr>
  <th>
    <input type="checkbox" name="contact5"
      value="931dc428-0 11b-1000-86fb-bfd83474aa25"
      onclick="tc(this, '5');">
  </th>
  <td style="padding-top:13px;"></td>
  <td>
    <span class="name less" style="max-width: 182px" id="x5">
      <a href="javascript:toggleContact('5')"
        title="Toggle contact">Max Roach</a>
    </span>
  </td>
```

```
    <td>
      <span class="mobile"><strong>Mobile</strong>5245232003</span>
    </td>
    <td class="end">
      <span class="email"><strong>Email</strong>
        <a href="mailto:maxroach@gmail.com">maxroach@gmail.com</a>
      </span>
    </td>
  </tr>
```

Cleaning up the HTML in this fashion was made much easier by first passing it through the great HTML Tidy program at http://tidy.sourceforge.net/.

There are at least two things we can learn about the data when we peer at it closely:

1. There's a lot of gunk (JavaScript, useless table columns, attributes, markup, etc.) we're going to want to ignore.
2. The information we do care about is found in three places:
   a. An anchor tag (<a>) holds the contact's name.
   b. A holds the phone number. That span has a class attribute (class="mobile") that will let us know the kind of phone it is.
   c. A span with a class of email holds the email address if there is one.

We're not entirely stuck at this point, because HTML::TableExtract has at least one more trick up its sleeve. If you load it like this:

```
use HTML::TableExtract qw(tree);
```

it will bring in a method from the HTML::TreeBuilder module (part of the HTML::Tree package which contains HTML::TreeBuilder, HTML::Element, and HTML::ElementTable). The tree() method from HTML::TreeBuilder can turn an extracted table into an HTML::ElementTable structure (composed of HTML::Element objects):

```
foreach my $table ( $te->tables ) {
        my $tree = $table->tree;

  # ... do stuff with HTML::Element/HTML::ElementTable objects
}
```

This gives us a tree-like data structure composed of the HTML elements in the table. Here's an example dump of the tree created for the previous HTML row snippet to give you an idea of the tree that is created:

```
    DB<1> print $row->dump
  <tr> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26
    <th> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.0
      <input name="contact11" onclick="tc(this, &#39;11&#39;);"
      type="checkbox" value="931dc428-011b-1000-86ff-bfd83474aa25" />
      @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.0.0
    <td style="padding-top:13px;"> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.1
    <td> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.2
      <span class="name less" id="x11" style="max-width: 182px">
      @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.2.0
        <a href="javascript:toggleContact(&#39;11&#39;)" title="Toggle
        contact"> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.2.0.0
          "Max Roach"
    <td> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.3
      <span class="mobile"> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.3.0
```

```
            <strong> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.3.0.0
              "Mobile"
            "5245232003"
        <td class="end"> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.4
          <span class="email"> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.4.0
            <strong> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.4.0.0
              "Email"
            <a href="mailto:maxroach@gmail.com">
            @0.1.7.0.2.0.1.0.2.5.2.0.1.0.26.4.0.1
            "maxroach@gmail.com"
```

This output shows the element (indented to show its level in the tree), a unique identifier, and any textual contents of the element. With that structure we should be able to tease apart the structured (albeit yucky) HTML contents of the table cells in question.

OK, so now it's clobberin' time. Our main tool for taking all of this apart is the HTML::Element method look_down(). We tell it which elements we want in the tree and it will return either the first element that matches that specification (if called in a scalar context) or all of the elements that match (if called in a list context). Our first use of it is to get all of the table rows:

```
my @table_rows = $tree->look_down( '_tag', 'tr',
      sub { $_[0]->look_down( 'class', 'name less' ) } );
```

This line of code requests elements that fit the two-part specification of

1. Find all of the <tr> tags . . .
2. . . . that contain an element with a class attribute of "name less".

look_down() then returns the list of matching HTML::Element objects that fit this bill. To get the actual data, we'll iterate over the objects returned and extract what we need:

```
foreach my $row (@table_rows) {

  my $name   = $row->look_down( 'class', 'name less' );
  my $work   = $row->look_down( 'class', 'work' );
  my $home   = $row->look_down( 'class', 'home' );
  my $mobile = $row->look_down( 'class', 'mobile' );
  my $email  = $row->look_down( 'class', 'email' );

  push @contactlist,
    [
      $name->as_trimmed_text(),
      ($work   ? ( $work->content_list )[1                          : '',
      ($home   ? ( $home->content_list )[1]                         : '',
      ($mobile ? ( $mobile->content_list )[1]                       : '',
      ($email  ? ( $email->content_list )[1]->as_trimmed_text()     : '',
    ];
}
```

The extraction starts with a gaggle of look_down() method calls, each seeking a class attribute with a specific value. Some of the method calls will return an HTML::Element; the rest will not succeed in their search and will return undef instead. Our next step will be to store the information found by the successful searches.

To understand what is going on in the push() statement you may need to flip back to the HTML example code we showed earlier. For the name field we can scoop up any text found in the sub-tree (as_trimmed_text()), because the only piece of text in an element with the class attribute of "name

less" is the actual name. Retrieving the other data is a little bit trickier because it has a pesky label next to the actual number: for example, Mobile.

Our look_down() calls have found elements that look like this:

```
DB<1> print $mobile->dump
<span class="mobile"> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.2.3.0
  <strong> @0.1.7.0.2.0.1.0.2.5.2.0.1.0.2.3.0.0
    "Mobile"
  "2996209109"
```

The element has two things in it: a sub-element and the actual text value we want (the phone number). We really only care about the text value, so we just reference the second element returned by content_list() as in ($mobile->content_list)[1]. The email address needs an extra as_trimmed_text() because the address is stored in an <a> sub-element instead of plain text like the phone numbers.

At the end of this rigmarole, we've got a bunch of lists in @contactlist, each list containing one contact record. We could easily spit it out as a comma-separated value file, like this:

```
foreach my $record (@contactlist) {
  print join( ',', map { '"' . $_ . '"' } @$record ), "\n";
}
```

with the resulting output looking like this:

```
"Charlie Parker","","","2996209109",""
"Coleman Hawkins","5834800077","","",""
"Hank Jones","","","2692315826",""
"Ray Brown","","7372450564","",""
"Lester Young","","","6633158411",""
"Bill Harris","","","6391737453",""
"Harry Edison","","","9987145662",""
"Ella Fitzgerald","","","2097688862",""
"Max Roach","","","5245232003","maxroach@gmail.com"
```

The Mac OS X address book is happy to import a CSV file of this format, so job done.

Eagle-eyed readers (i.e., those not falling asleep on the keyboard) may have noticed that our use of HTML::TableExtract in the last section didn't buy us very much. We still had to grovel around in a parsed tree of HTML elements to get anything done. We could have ditched HTML::TableExtract and gone right to something like HTML::TreeBuilder.

That's a perfectly valid criticism. In most cases, HTML::TableExtract hands you back the data elements you want; in this case, it just helped us find the right table in the document. There is at least one other excellent module for table parsing, called HTML::TableParser, we could have used, but my preliminary experiments with it in this context showed that the ugly HTML in the document gave it a tummy-ache as well. We'll have to save it for another task.

Hopefully, this column has given you an idea of how to extract data from both simple and complex HTML tables. Take care, and I'll see you next time.