DAVID N. BLANK-EDELMAN

practical Perl tools: perhaps size really does matter

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), 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. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.

dnb@ccs.neu.edu

I REALIZE THIS IS A VERY SENSITIVE topic for some people, so I am going attempt to broach this subject with as much care and finesse as possible. Today we are going to be talking about being small—how to accept and embrace being small, how to use what you have, that sort of thing. In fact, we're going to make you feel good about using things that are downright tiny. Or perhaps I should say "::Tiny", because the subject of this column will be the modules whose names end in ::Tiny.Took

Many of the modules in this family were inspired by work done initially by Adam Kennedy and a series of modules he created. These modules were designed to take some of the more useful but heavyweight modules in the Perl ecosystem and partially reimplement them as lean and mean as possible. There's a somewhat tongue-in-cheek module called Acme::Tiny, not written by Kennedy, whose doc lists a set of "commandments" that represents the gestalt of the ::Tiny family:

1. The module should be implemented in "as little code as possible."
2. The module should implement a useful subset of functionality.
3. The module should use at least 1/10th the amount of memory overhead, ideally less than 100k.
4. The module MUST have no non-core dependencies.
5. The module MUST be only one single .pm file.
6. The module should be back-compatible to at least 5.004.
7. The module should omit functionality rather than implement it incorrectly.
8. If applicable, the module should be compatible with the larger module.

::Tiny Markup

So let's actually look at some of the modules from this family. The first set I'd like to explore is in the general area of markup language processing. Diving into the deep end with the most markup of markup languages, XML::Tiny is a module that could be used if you had some basic XML parsing you needed done but didn't want the memory or module size overhead of the more complete parsers like XML::LibXML. For example, if we used the

Yahoo! Geolocation service we discussed in the June 2006 column, it would hand us back a result that looked like this without the indentation:

```
<?xml version="1.0"?>
<ResultSet xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns="urn:yahoo:maps" xsi:schemaLocation="urn:yahoo:maps
           http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.
xsd">
  <Result precision="address">
    <Latitude>37.859576</Latitude>
    <Longitude>-122.291659</Longitude>
    <Address>2560 9th St</Address>
    <City>Berkeley</City>
    <State>CA</State>
    <Zip>94710-2516</Zip>
    <Country>US</Country>
 </Result>
</ResultSet>
<!-- ws05.ydn.ac4.yahoo.com uncompressed/chunked Sun Jul 25 17:34:42
PDT 2010 -->
```

XML::Tiny will give you back a data structure that looks like this:

```
0 ARRAY(0x100b5baf8)
  0  HASH(0x100d0be48)
     'attrib' => HASH(0x100d0bd40)
        'xmlns' => 'urn:yahoo:maps'
        'xmlns:xsi' => 'http://www.w3.org/2001/XMLSchema-instance'
        'xsi:schemaLocation' => 'urn:yahoo:maps
        http://api.local.yahoo.com/MapsService/V1/GeocodeResponse.xsd'
     'content' => ARRAY(0x100d0be18)
        0 HASH(0x100d0bf38)
          'attrib' => HASH(0x100d0bff8)
            'precision' => 'address'
          'content' => ARRAY(0x100d0bf50)
            0 HASH(0x100d0c040)
              'attrib' => HASH(0x100d0c0a0)
                empty hash
              'content' => ARRAY(0x100d0c028)
                0 HASH(0x100d0be30)
                  'content' => 37.859576
                  'type' => 't'
              'name' => 'Latitude'
              'type' => 'e'
            1 HASH(0x100d0c148)
              'attrib' => HASH(0x100d0c1c0)
                empty hash
              'content' => ARRAY(0x100d0c118)
                0 HASH(0x100d0c0d0)
                  'content' => '-122.291659'
                  'type' => 't'
              'name' => 'Longitude'
              'type' => 'e'
            ...
```

It is the same data structure you would expect from XML::Parser::EasyTree. For this case, I find all of the dereferencing you need to do (i.e., $document

->[0]->{content}[0]->{content} . . . and so on) to be a bit tedious, so here's a quick way we can cheat. XML::Tiny's author also released a separate module called XML::Tiny::DOM that builds on XML::Tiny in a way that makes it *much* easier to get to the data we need. We'll lose some of the ::Tiny purity if we use it, but it means our code can look like this (pay particular attention to the clarity of the last two lines):

```
use LWP::Simple;
use URI::Escape;
use XML::Tiny::DOM;
use IO::Scalar;

# usage: scriptname <location to geocode>

my $appid   =  "{your Yahoo! API key here}";
my $requrl  =  "http://api.local.yahoo.com/MapsService/V1/geocode";

my $request = $requrl .
   "?appid=$appid&output=xml&location=" . uri_escape( $ARGV[0] );
my $RESPONSE = new IO::Scalar \get($request);

my $document = XML::Tiny::DOM->new($RESPONSE);

print "Latitude: " . $document->Result->Latitude . "\n";
print "Longitude: " . $document->Result->Longitude . "\n";
```

I feel compelled before we move on to the next module to let you know that some people are pretty negative about XML::Tiny's lack of XML-parsing rigor and handling of edge-cases. It certainly is the wrong module to use if pedantic parsing of XML is important to the task or the input data is at all complex. But in a case like the one above, it seems to do peachy.

The mention of complex data offers a segue to the next module: YAML::Tiny. YAML::Tiny's documentation explains its purpose like so:

> The YAML specification is huge. Really, *really* huge. It contains all the functionality of XML, except with flexibility and choice, which makes it easier to read, but with a formal specification that is more complex than XML.
>
> The original pure-Perl implementation YAML costs just over 4 megabytes of memory to load. Just like with Windows .ini files (3 meg to load) and CSS (3.5 meg to load) the situation is just asking for a YAML::Tiny module, an incomplete but correct and usable subset of the functionality, in as little code as possible.

(Note: we'll discuss the other large module examples mentioned above later in this column.)

Using it is as simple as using the standard YAML modules (again, from the doc):

```
use YAML::Tiny;
my $yaml = YAML::Tiny->read( 'file.yml' );
my $root = $yaml->[0]->{rootproperty};
my $one  = $yaml->[0]->{section}->{one};
my $Foo  = $yaml->[0]->{section}->{Foo};
...
```

Lest you think ::Tiny is somehow best for markup parsing only, let's look at two modules that handle markup generation. HTML::Tiny lets you write straightforward code to produce HTML documents, so code like this:

```
use HTML::Tiny;

    my $h = HTML::Tiny->new;

    # Generate a simple page
    print $h->html(
        [   $h->head( $h->title(';login Column') ),
            $h->body(
                [  $h->h1( { class => 'headline' }, 'First Section' ),
                   $h->p(
                      'This is a test.',
                      'Have you tried turning it off and on again?'
                   )
                ]
            )
        ]
    );
```

generates HTML like this:

```
    <html><head><title>;login Column</title></head>
    <body><h1 class="headline">First Section</h1><p>This is a test.</p>
    <p>Have you tried turning it off and on again?</p>
    </body>
    </html>
```

In a similar vein, if you parse or generate Cascading Style Sheets (CSS code), you might find Kennedy's CSS::Tiny module (mentioned above) useful for simple, quick work. It can parse a .css file, let you make changes, and write the file back out using much less overhead than CSS.pm.

The last markup-related module I'll mention is worth your attention not least for the gumption the author showed in even attempting to write it. Kennedy's Template::Tiny tries to give the Template Toolkit 2 distribution the ::Tiny treatment. Template Toolkit 2 (known as TT2 for short) is probably the single most popular templating engine in use in the Perl world today. I think you would be hard pressed to find a Perl Web framework that doesn't offer some way to use TT2 to generate content. Like other fully featured templating engines, it offers a panoply of ways to produce output based on a combination of static text and dynamic data. An example TT2 template (from the tutorial) looks like this:

```
    [% INCLUDE header %]

    People of [% planet %], your attention please.

    This is [% captain %] of the
    Galactic Hyperspace Planning Council.

    As you will no doubt be aware, the plans
    for development of the outlying regions
    of the Galaxy require the building of a
    hyperspatial express route through your
    star system, and regrettably your planet
    is one of those scheduled for destruction.

    The process will take slightly less than
    [% time %].

    Thank you.

    [% INCLUDE footer %]
```

Here you can see static text with placeholders that will be filled in with the appropriate content (in some cases the content of a variable, in others the contents of other files). You can also do fun stuff like:

```
<ul>
    [%  FOREACH page IN pages %]
        <li><a href="[% page.url %]">[% page.title %]</a>
    [%  END %]
</ul>
```

TT2 is so feature-full, there is an entire 592-page O'Reilly book on the subject. This means that creating a TT2 ::Tiny module is the equivalent of building a to-scale model of the QE2 in a bottle with the expectation that it should also be able to still transport a small number of passengers.

Given the author, I suspect the public Template::Tiny code is solid, but it is still in enough flux that I hesitate to recommend you use it. I would, however, recommend you follow Kennedy's blog (http://use.perl.org/~Alias/journal/) as he posts about the module's development and the engineering decisions he has made as he progresses. The first set of posts are at:

```
http://use.perl.org/~Alias/journal/39983
http://use.perl.org/~Alias/journal/39991
http://use.perl.org/~Alias/journal/40013
http://use.perl.org/~Alias/journal/40015
```

and for added fun, you can read about the JavaScript port of Template::Tiny at http://use.perl.org/~Alias/journal/40126.

::Tiny Programming

Lest you think the ::Tiny philosophy only applies to markup-related stuff, let's careen off in an entirely different direction and look at two programming-related ::Tiny modules. The first is the ::Tiny entry into an already pretty crowded field: class builders. If you are a Perl programmer with a heavy OOP bent (and yes, there are some of them out there), you know that Perl doesn't by default provide any shortcuts for the sometimes tedious process of creating accessors for data in the objects you define. The modules many people gravitate toward to handle this are something from the Class::Accessor distribution. Kennedy is very clear in his documentation about why Object::Tiny module is an improvement over those modules:

- Object::Tiny is 93% smaller than Class::Accessor::Fast.
- Class::Accessor::Fast requires about 125k of memory to load.
- Object::Tiny requires about 8k of memory to load.
- Object::Tiny is 75% more terse to use than Class::Accessor::Fast.
- Object::Tiny is used with the least possible number of keystrokes (short of making the actual name Object::Tiny smaller).
- And it requires no ugly constructor methods.

See the Object::Tiny documentation for the rest of the comparisons.

Using the module is as simple as:

```
package Foo::Bar;
use Object::Tiny qw{ foo bar baz };
```

and, lo and behold, when you create your Foo::Bar objects you get $object->foo, $object->bar, and $object->baz like magic. Super simple, super easy.

The other programming-related module worth mentioning given our limited space is Try::Tiny, which gives you the fairly standard try/catch exceptionhandling idiom other languages scoff at Perl for lacking. That's the one that looks like this:

```
try      { ... }
catch    { ... }
finally  { ... };
```

The idea is you can run a block of code as a "try"; if it fails the error can be handled by the "catch" code and "finally" is run if either of those two blocks returns success. You can do this sort of error handling using eval() in Perl (and in fact, that's how "try" is run), but it turns out there are a number of fiddly details mentioned in the Try::Tiny documentation that make a bare eval() not as useful as you'd like for this idiom. Try::Tiny handles all of those details for you beautifully.

::Tiny To-Go

There are far too many interesting ::Tiny modules worthy of your attention to fit into one column. I can only show you two more, but I encourage you to go searching on CPAN for "::Tiny". The first of the two harks back to a February 2006 *;login:* column on configuration files. Config::Tiny is a minimalistic .ini-format config file reader and writer. Here's the sample example from the documentation (slightly abridged):

```
use Config::Tiny;
my $Config = Config::Tiny->new();
$Config = Config::Tiny->read( 'file.conf' );
my $one = $Config->{section}->{one};
$Config->{newsection} = { this => 'that' };    # Add a section
$Config->{section}->{Foo} = 'Not Bar!';        # Change a value
delete $Config->{_};                           # Delete a value or section
$Config->write( 'file.conf' );
```

If you need something to read and write config files only machines will touch, Config::Tiny works great. It is a less good option if you care about preserving comments, order, whitespace, etc., in your config file (because it doesn't). For that you'll want to seek out some of the other modules I mentioned in the February 2006 issue.

OK, last module. Along with Try::Tiny it is perhaps one of the most useful in this column. If you have ever had to capture the output of an external program for use in a Perl program, Capture::Tiny is the module for you. Before you do anything else, I'd highly recommend you read the slides to David Golden's talk "How (NOT) to capture output in Perl" at http://www.dagolden.com/wp-content/uploads/2009/04/how-not-to-capture-output-in-perl.pdf.

Read that talk even if you are not planning to use anything ::Tiny. In it he discusses the various ways people try to capture output and the drawbacks of most of the ways you thought were the right ones. By the end of the talk, you'll be begging for a module that just Does The Right Thing. Golden provides a happy ending to his talk by making Capture::Tiny available. It provides capture, capture_merged (STDOUT and STDERR merged), tee and tee_merged functions that do just what you would hope they would do and do it well. If there is one true way to capture output from another command, this is probably it. And it is ::Tiny to boot.

With that last nugget, I think it is time to take my leave. I hope you've gained a new appreciation for the smaller things in life. Take care, and I'll see you next time.