Building HTML Presentation Slides

> *HTML is a great format for presentation slides. All modern desktop systems can view it with standard software; it's a perfect fit for publishing to the web; and search engines love it. Of course manually editing HTML pages and managing the links between them would be a painful and unpleasant process. That's where Perl can help.*

As Ruby-on-Rails developer David Heinemeier Hansson is fond of saying, "*constraints are liberating*". When I'm preparing a presentation I find that not having to make decisions about fonts, colours, animation effects etc. frees me up to concentrate on the real job of entering the bullet points and thinking about the message I want to convey.

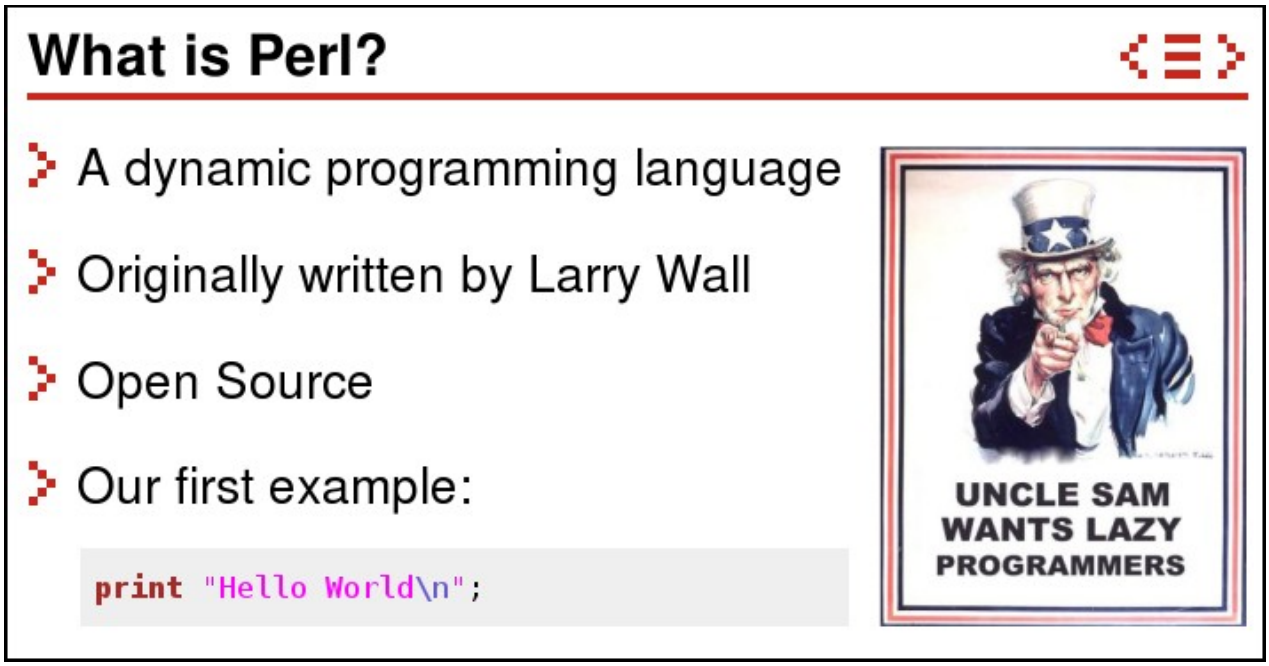
To liberate myself from extraneous decision making, I build my presentation slides using a Perl script which imposes certain constraints. Each slide is made up of:

- a title
- an optional decorative image
- a list of 'content items'
- navigation links

Three types of content items are supported:

- simple bullet text (no nesting)
- code samples (with optional syntax highlighting)
- inline images (typically screenshots)

Here's an example slide that crams in most of these elements:



The system I have developed is in many respects a 'throwaway'. I sometimes find when preparing a new presentation that the subject matter, the audience or even the forum calls for a subtle twist to my standard slide formula. Rather than try to make my code more and more general to cater for all these different possibilities, I simply take a copy of the script and 'hack in' whatever changes are required or perhaps even 'hack out' some standard feature that's getting in the way.

In this article I'll walk through my program architecture and some of the design decisions to give you the background to adapt the script or perhaps develop your own from scratch. This will give you the tools to build a system that meets *your* needs.

What Goes In

When I need to create a new presentation, I `cd` into my *talks* folder; `untar` a skeleton set of files (including the *mkpres.pl* script itself); rename the resulting directory to match the new talk title; and add the new directory to my subversion repository.

To flesh out the presentation, I `cd` into the new directory; edit the *talk.xml* file; and drop some images into the *html/images*. For example, here's the XML input used to generate the sample slide illustrated above.

```
<presentation>

  <title>Title Here</title>
  <author>Presenter's Name</author>
  <email>name@somedomain</email>

  <slide>
    <title>What is Perl?</title>
    <image>uncle_sam.jpg</image>
    <bullet>A dynamic programming language</bullet>
    <bullet>Originally written by Larry Wall</bullet>
    <bullet>Open Source</bullet>
    <bullet>Our first example:</bullet>
    <code syntax="perl"><![CDATA[
      print "Hello World\n";
    ]]></code>
  </slide>

</presentation>
```

Editing raw XML probably sounds like a tedious way to assemble a presentation. Indeed I originally envisaged building a front-end to hide the XML, using either a GTK GUI or a browser-based app. As a stop-gap I created some macros in my text editor (Vim) and discovered that the result was much more streamlined than a GUI could ever be and I've been happy to stick with that ever since. The macros work as follows:

,s

: Add markup for a new slide leaving the cursor positioned to type the title

,b

: Insert a new line with `<bullet>` ... `</bullet>` tags, leaving the cursor positioned to type the text

,c

: Add `<code>` and `CDATA` tags which allow code snippets to be pasted in without worrying about normal XML escaping rules

,r

: Add `<screenshot>` tags, leaving the cursor positioned to type the image filename

What Comes Out

When I run the *mkpres.pl* script, it generates a series of HTML files. In addition to the main content slides, there's a title slide to act as the 'front cover'. A table of contents slide is also generated to allow jumping straight to a specific slide - mainly useful when handling questions at the end.

Each slide file includes navigation links for advancing to the next slide, returning to the previous slide or jumping to the table of contents. A small JavaScript file linked into each slide allows me to advance from one slide to the next by pressing the spacebar - far easier than fumbling round with the mouse to click on navigation links.

A CSS file is used to format the slide pages. The stylesheet applies a large readable font to all text. It replaces the standard HTML bullets with 'themed' images and adjusts alignments and spacing to reduce clutter. The stylesheet also includes a section for assigning colours to syntax highlighted elements in code sections.

How It Works

Even on a relatively small project such as this, it's important to stick with established design principles. One such principle is the 'Separation of Concerns'. Rather than weave together bits of code that pick apart the XML and write out snippets of HTML, the script is organised into routines that each concern themselves with one thing. This separation is apparent right from the start of the script:

```
my $pres = read_presentation_data($file);
generate_presentation_pages($pres);
```

The `read_presentation_data()` routine is concerned with parsing the XML input while the `generate_presentation_pages()` routine generates the HTML output. The link between the two routines is a Perl data structure. The source XML markup is extremely simple - to make data entry easy. The generated HTML is also very simple. The most complex parts of the script are concerned with massaging the input data into a form that can be used to generate the HTML. The data structure will end up looking something like this:

```
{
    'metadata' => {
        'title'    => 'Title Here',
        'author'   => 'Presenter\'s Name',
        'email'    => 'name@somedomain'
    },
    'slides' => [
        {
            'title'    => 'What is Perl?',
            'filename' => 'slide001.html',
            'next'     => 'slide002.html',
            'previous' => 'toc.html',
            'image'    => 'unclesam.jpg',
            'content' => [
                [
                    'bullets',
                    [
                        'A dynamic programming language',
                        'Originally written by Larry Wall',
                        'Open Source',
                        'Our first example:'
                    ]
                ],
                [
                    'code',
                    'print "Hello World\\n";'
```

```
                ]
            ]
        },
        ...
    ],
    'toc' => [
        {
            'filename' => 'slide001.html',
            'title' => 'What is Perl?'
        },
        ...
    ]
};
```

The first step towards producing this data structure is to parse the XML. The XML::LibXML module makes parsing the XML file to a DOM object a snap:

```
my $parser = XML::LibXML->new;
my $doc    = $parser->parse_file($file);
```

Extracting the necessary data is then achieved by querying the DOM object using XPath expressions:

```
my $pres = {
    metadata => {
        title  => $doc->findvalue('/presentation/title')  || '',
        author => $doc->findvalue('/presentation/author') || '',
        email  => $doc->findvalue('/presentation/email')  || '',
    },
};
add_slide_data($pres, $doc);
```

The `findvalue()` method locates the DOM node that matches the supplied XPath expression (e.g. `'/presentation/title'`) and then returns that node's text content as a simple scalar. The `add_slide_data()` routine creates an array of slide data structures and adds the array into the `$pres` hash. It processes the `<slide>` elements using a simple loop:

```
my @slides;
foreach my $slide ( $doc->findnodes('/presentation/slide') ) {
    push @slides, parse_slide($slide);
}
```

The `parse_slide()` routine then uses relative paths (starting with a leading '.') in XPath expressions to extract the required data:

```
my $data = {
    title   => $slide->findvalue('./title') || '',
    ...
};
```

Since the body of each slide may be made up of multiple different types of elements and the order of these elements must be preserved, extracting this data is a little more complex:

```
  my $item_expr = './bullet|./pause|./screenshot|./code|./image';
  foreach my $node ($slide->findnodes($item_expr)) {
      my $type = $node->nodeName;
      if($type eq 'bullet') {
          ...
      }
      elsif($type eq 'pause') {
          ...
      }
```

```
        elsif($type eq 'screenshot') {
            ...
        }
        ...
    }
```

The <bullet> elements are allowed to contain simple HTML markup. This is primarily to allow clickable links in the generated HTML, but adding emphasis to particular words can also be useful. Extracting the literal value of a <bullet> element would effectively strip out the markup and return only the text. Calling $node->toString() would retain the markup but would also retain the surrounding <bullet> and </bullet> tags. Instead, we need to call toString() on each of the children of $node and concatenate the results:

```
push @$blist, join '', map { $_->toString } $node->childNodes;
```

Handling <pause /> elements in between <bullet> elements adds some complexity. The role of the <pause /> is to expose one bullet point each time the presenter presses the spacebar (a useful feature but one that should be used sparingly). One additional HTML page will be generated for each <pause> (one bullet point on the first page, two on the next and so on).

The parse_slide() routine would usually return one hashref containing all the data required to generate one HTML file. However if the slide contains pauses, parse_slide() will return multiple hashrefs. The trick with building up the array of hashrefs is to avoid ending up with multiple references to exactly the same hash. Each time a <pause /> is encountered a deep copy of the current hashref is pushed onto the array:

```
    elsif($type eq 'pause') {
      push @partials, Storable::dclone($data);
    }
    ...
  return @partials, $data;
```

Code snippets also require special handling. For example here's the relevant snippet of code as it might appear in the source XML for a presentation:

```
  <code syntax="perl"><![CDATA[
    elsif($type eq 'code') {
        my $code = outdent($node->to_literal);
        my $language = $node->findvalue('./@syntax') || '';
        $code = syntax_highlight($code, $language) if $language;
        push @$content, [ code => $code ];
    }
  ]]></code>
```

The outdent() routine strips off any leading and trailing blank lines and removes a string of whitespace from the start of each line to leave the outermost block flush with the left margin. If the <code> tag includes a syntax attribute then the Text::VimColor module is used to apply syntax highlighting for the specified language:

```
  return Text::VimColor->new(
      string   => $code,
      filetype => $language,
  )->html;
```

Once all the <slide> elements have been processed, the resulting list of hashrefs is traversed to set up URLs for preceding and following slides, as well as to build up a list of entries for the table of contents.

The process of generating the HTML is a simple matter of feeding the data structure for each page into a template. There are many Perl templating solutions available and which you use really comes

down to personal choice. In this project I used Template::MasonLite which uses the HTML::Mason syntax (or at least a minimal subset). The unique feature of Template::MasonLite is that the template interpreter embeds directly into your script. This means that apart from XML::LibXML, my presentation script uses only 'core' modules included in the Perl distribution.

Writing the main cover page for the presentation involves parsing the template, passing a datastructure to the template object's `apply()` method and saving the result to a file:

```
my $tmpl = Template::MasonLite->new_from_file('./_index.html');

open my $out, '>:utf8', "./html/$file" or die "open(./html/$file): $!";

print $out $tmpl->apply(%$data);
```

Generating all the slide files works exactly the same way except that the template file only needs to be read once and the template object is reused for each slide.

All the strings returned from XML::LibXML will use UTF-8 - Perl's native character encoding. All the output is destined for consumption by a web browser and browsers are quite comfortable with UTF-8, so sticking with UTF-8 for output is the simplest option.

The slide generation program only concerns itself with generating HTML files. The CSS, JavaScript and image files are all managed manually.

Review

This article has examined a common pattern for Perl scripts: parsing input data into a Perl data structure which is then used to generate output. This pattern makes the program more modular by separating the code into sections, each of which is concerned with just one thing.

When working with XML data it's hard to beat XML::LibXML for power. The API provides a bewildering array of methods but XPath expressions allow you to access the data using relatively simple queries and a small number of DOM methods.

Resources

You can download the presentation builder script, HTML templates, CSS, JavaScript, Vim macros and a skeleton *talk.xml* file from: http://wellington.pm.org/articles/slides/

For a tutorial introduction to XPath and an interactive tool for trying out your own XPath queries visit: http://www.zvon.org/

The Template::MasonLite templating system is available at http://www.perlmonks.org/?node_id=393725

ABOUT THE AUTHOR

Grant McLean `<grantm@cpan.org>` lives in Wellington, New Zealand where he works for Open Source specialists Catalyst IT. He's a regular speaker at Wellington Perl Mongers and has presented talks and tutorials at various industry conferences. Grant's CPAN contributions include XML::Simple.