Creating Open Office Documents with Perl

Just Mail It with Perl

It seems hardly worth launching Open Office just to print a standard letter with a few lines of text. But before you resort to using a pen and paper, why not fire up your Perl interpreter? You can use Perl to create a handy tool that will help you format letters.

BY MICHAEL SCHILLI

Open Office has become a genuine alternative to Windows solutions such as MS Office. Just complete a few simple installation steps, and you can use a variety of programs to manipulate Open Office documents without even launching the Open Office package.

Of course, Perl has a module that gives you access to Open Office data. A small program called *Mailit* (see Listing 1) helps you create letters quickly and easily. To do so, *Mailit* uses templates like the one shown in Figure 1, adding the recipient, the subject, the body text, and the current date. The template uses the following placeholders:

```
[% date %]      => current date
[% recipient %] => recip. addr.
[% subject %]   => subject line
[% text %]        => body text
```

Mailit uses a plain text file (see Figure 2) and an Open Office document as a template to create a letter with a neat layout (see Figure 3). The simple text file is structured in paragraphs. The first paragraph contains the subject line, and after it are paragraphs containing the body text of the letter. *Mailit* automatically generates the date (top right), which it formats to reflect the current locale settings.

On the Shoulders of Giants

No less than five CPAN modules were used to implement *Mailit*. Needless to say, the modules can do a lot more than the simple scripting we will be looking into in this article.

The first of the bunch, *OpenOffice::OODoc*, provides an object-oriented interface to the content and structure of Open Office documents. *Mailit* only needs text replacement and just uses the *OpenOffice::OODoc::Text* subclass.

The *new* constructor in line 36 of *Mailit* first opens the Open Office file, a template document with the correct formatting and placeholders, as shown in Figure 1. The *getTextElementList()* method then extracts a list of all the text elements in the document. The return values are pointers to *XML::XPath::Node::Element* objects, as *OpenOffice::OODoc* relies heavily on *XML::XPath* for the internal representation of XML-based Open Office files under the hood.

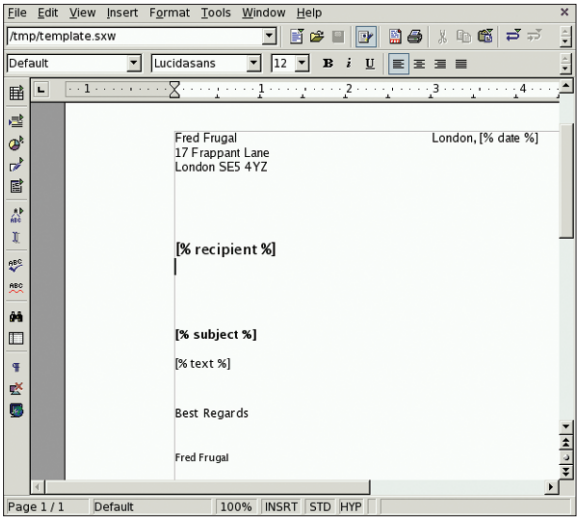

Figure 1: The Open Office document template *letter.sxw* uses placeholders, which are replaced dynamically by text strings. A Perl script does the actual replacement work.

To extract the text from a paragraph element *$e* returned by the list function, you need to call the *OpenOffice::OODoc::Text* object *getText()* method, passing it a pointer to the element: *$doc->getText($e)*.

Mailit then checks the text for placeholders with the *[% xxx %]* format and

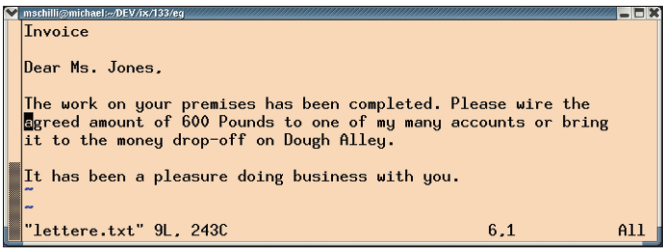

Figure 2: The text version of the letter in the vi editor. The first paragraph provides the subject, and the remaining paragraphs contain the body text.

replaces these values as previously defined. The program then uses the *setText()* method (line 95) to write the results back to the document, again using a pointer to the element: *$doc->setText($e, $text)*.

Text Replacement Made Easy

The second module used by *Mailit*, the powerful Template Toolkit, does the actual work of replacing the text. Template Toolkit is the new "in" module for Web applications; it fills designer-styled HTML templates with dynamic data. In line 71, *Mailit* creates a *Template* object. The *%vars* hash defined in line 73 assigns dynamic values to the placeholders in the document. In *Mailit*, the *Template* module *process()* method, which is called next, expects three parameters:

- A string with the template text
- A reference to the *%vars* hash
- A reference to a function, which *process()* calls after completing text replacement, passing it the result string.

The last of these three is optional, but it makes sense in our example, as *Mailit* can use it to call *setText()* and pass the modified text to the Open Office document. *save* in line 109 stores the document in a new temp file, which was created by the *File::Temp* module in line 102.

File::Temp is a Cadillac among tempfile modules. This module's major strength is its ability to create temporary files without colliding with existing files. Programmers simply select the directory where they want to store the files (*DIR => '/tmp'*), the file suffix (*SUFFIX => '.sxw'*), and a template for naming the files. *TEMPLATE => 'ooXXXXX'* starts the file name with *oo* (for Open Office), and then adds five random characters. The full name of a temp file would look like the following: */tmp/oo2hkss.sxw*

The *UNLINK* parameter tells the module to delete the file when the corresponding object is removed. The module then returns a reference that can be used as a file handle, and which converts to the tempfile name if interpolated within a string.

The tried and trusted *Date::Calc* module, the fourth on our list, helps *Mailit* to ascertain the current date and convert it to the local format *Month XX, Year*. To do so, the module first sets the locale *Language(Decode_Language("English"));* and then calls the *Date_to_Text()* function to convert the number returned by the *Today()* function to the name of the month.

Human-Readable Address Database

Mailit uses an address database to retrieve the multiple-line recipient address, which replaces the *[% recipient %]* placeholder in the document.

The choice of a format for a simple address database was one of the most important issues while implementing *Mailit*. There are so many options, and XML is probably one of the better ones. This said, the excessively triangular structures that XML uses impact its readability and can cause headaches for human readers.

In contrast to XML, Brian Ingerson's YAML (YAML Ain't Markup Language) is not only easy to parse but easy on the eye. In YAML, an address database that uses mnemonics to index its records, and assigns values to name, street, and city, would look like this:

```
fred:
  - Fred Davis
  - 123 Any Ave
  - Lawrence KS 66044
julie:
  - Julie Jones
  - 7 Lincoln
  - Barnard KS 67418
```

When the filename is passed to the YAML Perl module's *LoadFile()* function, the function returns a pointer to a hash, which contains the mnemonic as a key, and the entries as pointers to arrays:

```
{
'julie'=> [
  'Julie Jones',
  '7 Lincoln',
  'Barnard KS 67418'
],
'fred' => [
  'Fred Davis',
  '123 Any Ave',
  'Lawrence KS 66044'
], ...
}
```

YAML can do a lot more. As the name suggests, YAML is not a markup language but rather a flexible data serializer capable of converting Perl's deeply nested core data structures into easily readable ASCII texts and then importing them back to Perl after manipulation.

In the Mail

Mailit expects the text version of the letter, either as a filename or as input via stdin. *mailit letter.txt* and *cat letter.txt | mailit* have exactly the same effect, as

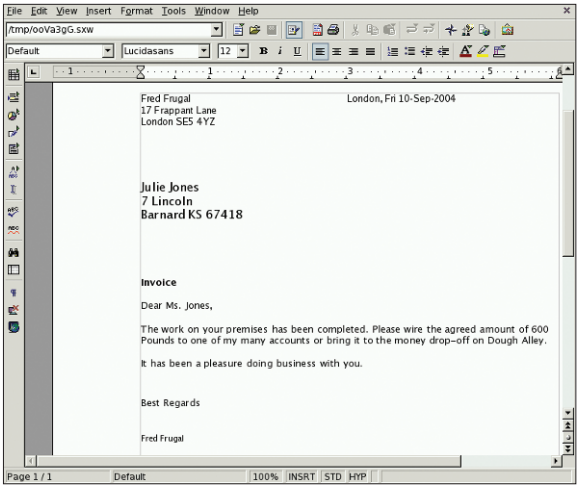

Figure 3: The document to be printed, after *Mailit* has completed its work on the template (Figure 1): subject line and body text from the plain text in Figure 2 have been inserted, as well as the addressee and the date.

line 40 uses Perl's magic input diamond. The regular expression in line 45 separates the first paragraph from the rest of the letter and stores the corresponding segments separately in *$subject* and *$body*.

The *pick()* function defined in line 118 expects a list of mnemonics for recipients and presents them to the user as a numbered list, prompting the user to pick a number to select an address. A typical *Mailit* session looks like this:

```
mailit letter.txt
[1] julie
[2] fred
[3] zephy
Recipient [1]> 1
Preparing letter for Julie Jones
Printing /tmp/ooGd8H3.sxw
```

To print the temporary document, the program simply calls Open Office in line 114. The *-p* option tells Open Office not to launch the GUI, but instead send the **.sxw* file to the standard printer. When the file is printed, you will find a perfectly formatted letter ready for the mail. But this isn't like the Internet, so you'll need to buy a stamp. ■

INFO

[1] Listings: *http://www.linux-magazine.com/Magazine/Downloads/48/Perl*

[2] OpenOffice project homepage: *http://openoffice.org*

Listing 1: mailit

```
001 #!/usr/bin/perl
002 ##################################
003 # mailit -- Print letters
004 #           with OpenOffice
005 # Mike Schilli, 2004
006 # (m@perlmeister.com)
007 ##################################
008 use warnings;
009 use strict;
010
011 my $CFG_DIR =
012   "$ENV{HOME}/.mailit";
013 my $OO_TEMPLATE =
014   "$CFG_DIR/letter.sxw";
015 my $ADDR_YML_FILE =
016   "$CFG_DIR/addr.yml";
017 my $OO_EXE =
018 "$ENV{HOME}/ooffice/soffice";
019
020 use OpenOffice::OODoc;
021 use Template;
022 use YAML qw(LoadFile);
023 use File::Temp;
024 use Date::Calc qw(Language
025   Date_to_Text
    Decode_Language
026   Today Date_to_Text);
027
028 Language(
029   Decode_Language("English")
030 );
031 my ( $year, $month, $day ) =
032   Today();
033
034 my $doc =
035   OpenOffice::OODoc::Text
036   ->new(
037   file => $OO_TEMPLATE, );
038
039 # Read from STDIN or file
040 my $data = join '', <>;
041
042 # Split subject and body
043 my ( $subject, $body ) =
044   ( $data =~
045     /(.*?)\n\n(.*)/s );
046
047 # Remove superfluous blanks
048 my $text;
049 for my $paragraph (
050   split /\n\n/, $body )
051 {
052   $paragraph =~ s/\n/ /g;
053   $text .= "$paragraph\n\n";
054 }
055
056 my $yml =
057   LoadFile($ADDR_YML_FILE);
058 my $nick = pick(
059   "Recipient",
060   [ keys %$yml ]
061 );
062
063 my $recipient =
064   $yml->{$nick};
065
066 print
067   "Preparing letter for ",
068   $recipient->[0], "\n";
069
070 my $template =
071   Template->new();
072
073 my %vars = (
074   recipient => join(
075     "\n", @$recipient
076   ),
077   subject => $subject,
078   text    => $text,
079   date    =>
080     Date_to_Text(
081       $year, $month, $day),
082 );
083
084 for my $e (
085   $doc->getTextElementList()
086   ) {
087
088   my $text_element =
089     $doc->getText($e);
090
091   $template->process(
092     \$text_element,
093     \%vars,
094     sub {
095       $doc->setText( $e,
096         $_[0] );
097     }
098   );
099 }
100
101 my $oo_output =
102   File::Temp->new(
103   TEMPLATE => 'ooXXXXX',
104   DIR      => '/tmp',
105   SUFFIX   => '.sxw',
106   UNLINK   => 1,
107   );
108
109 $doc->save(
110   $oo_output->filename );
111
112 print
113   "Printing $oo_output\n";
114 system(
115   "$OO_EXE -p $oo_output");
116
117 ##################################
118 sub pick {
119 ##################################
120   my($prompt, $options) = @_;
121
122   my $count = 0;
123   my %files = ();
124
125   foreach (@$options) {
126     print STDERR "[",
127       ++$count, "] $_\n";
128     $files{$count} = $_;
129   }
130
131   print STDERR
132     "$prompt [1]> ";
133   my $input = <STDIN>;
134   chomp($input);
135
136   $input = 1
137     unless length($input);
138   return "$files{$input}";
139 }
```