Routine Server-based Bulk Notice Generation

Notes on the Gory Details

Peter Corrigan, Head of Library Systems, National University of Ireland, Galway.

Aleph's multi-tiered printing architecture offers many wonderful opportunities for customization and enhancement.

Here we describe elements of a custom Perl program which embeds an XSLT processor and runs either on the Aleph server itself, or on a suitable alternative server-type machine. The custom program utilizes the existing ExLibris supplied stylesheets and is typically called by the system cron to process the routine bulk-courtesy/overdue-notice XML source-files as found in the $data_print directory. Thus, the custom program operates without any user intervention, uses tried and trusted Unix technology, need not rely on any desktop or PC applications, and allows a high degree of control over the message content, format and transport.

With Aleph 15 and later, the Notices-to-Patrons Services `cir-10, cir-50, cir-51` and `cir-52` are typically run from the Job Daemon and generate XML type output for the production of Overdue Notices, Courtesy Reminders and similar bulk correspondence. Typically, this output is transferred to a remote PC running the Task Manager print daemon, where appropriate notices are produced by applying an XSLT stylesheet to the source XML using the Saxon XSLT processor, as bundled with the Aleph GUI. While the transformation of the raw notice-XML into a notice suitable for printing or mailing normally takes place in the Aleph GUI, this capability can be easily implemented on a variety of platforms using a modern scripting language like Perl.

XSLT transformation using Perl and ExLibris' stylesheets

If you have ever taken a peek at the output files created by Aleph's `cir-10, cir-50, cir-51` or `cir-52` services you'll have noticed that the files are not strictly in XML format. Rather, the output is a series of XML documents, all contained within a single file, with each document delimited by a line composed of the string ## - XML_XSL as illustrated in figure 1.

This is not well-formed XML, so we have a problem, because an essential

```
<z36-original-due-date>23/07/2003</z36-original-due-date>
<z36-process-status></z36-process-status>
<z36-loan-type></z36-loan-type>
<z36-proxy-id></z36-proxy-id>
<z36-recall-type></z36-recall-type>
</section-02>

</printout>
## - XML_XSL
<?xml version="1.0"?>
<printout>
<form-name>loan-notice-letter</form-name>
<form-language>ENG</form-language>
<form-format>00</form-format>
```

The record delimiter, the value of which we assign to Perl's input record separator.

Fig 1 Output created by Aleph services is not strictly in XML format.

prerequisite to successful XSLT transformation is that the source be parsable using an XML parser,

To overcome this we can utilize a common Perl idiom for parsing files of multiple text records. We can make Perl's input record separator, (the `$/` variable), equal to our notice data separator, i.e. the string `## - XML_XSL.` Normally, $/ is the newline character. When we alter it to be `## - XML_XSL` Perl's line input operator **<>**, will no longer return a single line from the input file for each iteration. Instead, it returns all the XML for a single *notice* as an ordinary Perl scalar variable, as we iterate over the file within a `while` loop. We'll discuss this in more detail with a practical example later in our code walkthrough, below.
The upshot is that this approach gives us both valid XML, and a convenient way to handle each discreet notice as a simple perl scalar. This technique is scalable and will cope even if the Aleph services output contains many thousands of notices and is hundreds of megabytes in size.

```
01  #!/usr/local/bin/perl
02  use warnings;
03  use strict;
04  use XML::LibXML;
05  use XML::LibXSLT;
06
07  my $parser = XML::LibXML->new();
08  my $xslt   = XML::LibXSLT->new();
09  $parser->base_uri( './stylesheets' );
10  my $style_doc = $parser->parse_file('./stylesheets/loan-notice.xsl');
11  my $stylesheet = $xslt->parse_stylesheet($style_doc);
12  while (defined ($_ = do { local $/ = "## - XML_XSL\n"; <> })) {
13     s/## - XML_XSL\s*//;
14     if (/<\?xml/)    {
15        my $source = $parser->parse_string($_);
16        my $notice = $stylesheet->transform($source);
17        print $stylesheet->output_string($notice), "\f";
18     }
19  }
```

Fig 2. Naïve bare-bones notice transformation – simple.pl

Fig 2 lists a complete, if naïve, script to transform an Aleph-services output file. If we call it simple.pl we could invoke it thus:
`simple.pl aleph-notice-file.dat >file-for-printing.prn`
i.e. taking a notice file on standard input and piping output to a file, perhaps for later printing.

simple.pl Walkthrough

On lines 1-3 we invoke perl, and use our usual `warnings` and `strict` pragmata to restrict unsafe constructs and alert us to possible problems with our code. On lines 4 and 5 we import the `XML::LibXML` and `XML::LibXSLT` modules. On lines 7 and 8 we construct an XML parser instance and stylesheet object. On line 9, we tell our parser the base uri for our stylesheets (i.e. where all the Exlibris XSL include files are to be found), allowing us keep things tidy by having our stylesheets in a separate directory from our program.

We *must* set the base_uri for our parser instance when we use a stylesheet that includes other stylesheets by using `xsl:include or xsl:import` elements with a href attribute that is a relative URI. For example, the following is frequently encountered in the ExLibris supplied stylesheets.

```
<xsl:include href="funcs.xsl"/>
```

Line 10 sees us parse the stylesheet as an XML::LibXML::Document object – in this example we are using a hardcoded filename, but in real life we'd pick it up from the the source document itself, the command line, or an ini file. (We'll look at how we might do that later).

On line 11 we take our style document source tree and create a stylesheet object. The stylesheet object has a convenient *transform* method which does the actual transformation. We'll see that in action in a moment.
Line 12 is where our program starts getting interesting

```
while (defined ($_ = do { local $/ = "## - XML_XSL\n"; <> })) {
```

We are reading a file on standard input, where the normal input-record-separator is a newline character. However, we want to make the string `"## - XML_XSL\n"` our input-record-separator, and have this change confined to as small a part of our program as possible, hence the do loop. If we were to allow the reassiged $/ to extend in scope to the rest of the while loop, we would risk unintended consequences as succeeding lines, or modules utilized in succeeding lines, will expect a \n as input-record-separator. Hence, we localize the new value of $/ to the do block. On line 13 we remove the delimiter using a substitution regex where we replace the pattern `## - XML_XSL\s*` with nothing. On line 14 we just check that we are dealing with actual XML using a simple regex, before we present it to the parser, otherwise our parser will fail. On line 15 we parse our notice printout-XML (the current value of `$_`). On line 16 we perform the actual transform using our stylesheet object's transform method. This returns a reference to another `XML::LibXML::Document` in the variable `$notice`. On line 17 we print our notice to standard output with each notice terminated by a formfeed character using the stylesheets's `output_string` method. We could just as easily have stored the output from `stylesheet->output_string` in a scalar for further processing e.g.

```
my $notice_html = $stylesheet->output_string($notice)
```

with further processing conveniently invoked within our while loop, perhaps calling one or more subroutines.

Extracting what we want with XPath

If you have been using XSLT, you know that an XPath expression consists of a number of path *steps.* These extend a notional path, incrementally closer to a given target node, akin to a filesystem path specifying the location of a file. The XPath steps "walk" across the document tree to arrive at a node or set of nodes. In the same manner as the XSLT stylesheets home-in on a piece of information within the XML source, we can use the same technique within our program using Xpath expressions. For example:

```
my $recipient = $source->findnodes("/printout/section-01/email-address");
```

or

```
my $issns = $source->findnodes("/printout/section-02/z13-isbn-issn");
```

The findnodes method returns a nodelist object in scalar contect or an array in list context. A nodelist is just an ordered list of nodes, that conforms to the DOM specification of a nodelist. It has its own set of methods like push, pop, shift etc.

Sending Mail

When it comes to sending mail from Perl, we have many options available to us. On a Unix system we *could* simply pipe our output to sendmail e.g.

```
open ( MAIL, "| /usr/lib/sendmail -oi -t" );
```

but there are lots of modules to choose from that make things safer, portable, more reliable and facilitiate sending attachments or multipart mail: for example Mime::Lite, Mail::Send, Mail::Sender, Mail::SendMail, and Mail::SendEasy. In NUI, Galway we use Jenda Krynicky's Mail::Sender module to send multi-part alternative mail with a HTML and a text alternative. We could just as easily create a message with attachments and inline images, but in our environment, HTML with a text alternative works well.

We can create a Mail::Sender object thus:

```
…
   use Mail::Sender;
   my  $sender = new Mail::Sender {
         smtp     => '140.203.7.16',
         from     => 'library@nuigalway.ie',
         on_errors=> undef,
        }
 or die "Can't create the Mail::Sender object:$Mail::Sender::Error\n";
…
```

Later we can utilise the Mail::Sender object within a generalized **`mail_send`** subroutine. Line 8 below calls a mail_send routine with content and recipient email address as parameters:

```
01 while (defined ($_ = do { local $/ = "## - XML_XSL\n"; <> })) {
02   s/## - XML_XSL\s*//;
03     if (/<\?xml/)    {
04        my $source = $parser->parse_string($_);
05        my $recip=$source->findnodes("/printout/section-01/email-05 address");
06        my $html_notice =  $html_stylesheet->transform($source);
07        my $plain_notice = $plain_stylesheet->transform($source);
08        mail_send ( $stylesheet->output_string($html_notice),
                      $stylesheet->output_string($plain_notice),
                      $recipient);
09     }
10  }
```

Here we utilize our now familiar while loop as before, but this time we perform two transformations – one to produce an HTML notice (line 6), and the other (line 7), to produce a plain-text notice. On line 8 we call subroutine mail_send with the HTML

and text for a multipart-notice and the email address of the recipient. Subroutine mail_send, contains three Perl statements and looks like this.

```
01 sub mail_send {
02   my ($html_body, $plain_body, $recipient_email_addr ) = @_;
03   eval {$sender->OpenMultipart({
04                       to => $recipient_email_addr,
05                       subject => "Library notice",
06                       encoding => 'quoted-printable',
07                       charset => 'iso-8859-1',
08                       multipart => 'mixed',
09             })
10     ->Part({ctype => 'multipart/alternative'})
11                 ->Part({ctype => 'text/plain',
12          disposition=> 'NONE', msg   => <<"*END*" })
13
14    $plain_body
15
16
17*END*
18         ->Part({ctype => 'text/html; charset= iso-8859-1',
19  disposition => 'NONE', msg => <<"*END*"})
20
21<!-—An optional HTML header  -->
22    $html_header
23
24    $html_body
25
26
27<!— An optional html footer -->
28    $html_footer
29
30*END*
31       ->EndPart("multipart/alternative")
32
33
34   ->Close();
35  } or print "Error sending mail: $Mail::Sender::Error\n";
36}
```

We can place arbitrary content into either version of the message body using Perl variable interpolation

Fig 3 Subroutine mail_send

Despite being 32 lines-long, subroutine `mail_send` contains a mere two perl statements. On line 2 we read the subroutine's parameters into local variables for later processing. The long statement on line 3 makes up the rest of the subroutine. It starts by wrapping everything that follows within an `eval` block. A common Perl idiom, this prevents fatal errors within the block terminating the program prematurely. With the call to **`$sender->OpenMultipart`** on line 3, we invoke the Mail::Sender object, (which we constructed earlier), and commence communication with the SMTP server. On lines 4-8 we set the recipient for this particular message, the subject line, and the character set. We also set the overall MIME subtype for the entire message as multipart mixed. Lines 10-12 set up alternative-message number-one, a text-version

of our multipart-alternative message. The ctype or Content-type is text/plain. This version of the message will display if the mail agent does not understand HTML. The disposition hash-key on line 12 relates to the Content-Disposition header which provides a mechanism to tag each component of a message in order to communicate to the Message User Agent e.g. MS Outlook, an indication of the component's desired presentation semantics and includes metadata describing attachment filename, size etc. See RFC 2183 for more details. We choose 'NONE' for inline display of our message.

```
12 msg   => <<"*END*" })
13
14    $plain_body
15
16
17*END*
```

Lines 12 to 17 above simply implement a HERE document, where anything placed on lines 13-16 will form the text version of our message. We just choose to have our transformed source interpolate via the `$plain_body` variable. The rest of the subroutine simply repeats message creation for ctype text/html. In this case we have the option to interpolate a HTML header and footer. We could additionally have a MOTD (Message of the day), facility to communicate late-breaking information like changes in opening hours, new exhibitions and so forth, by interpolating an appropriate message.

Sending SMS Messages

Net::SMS is a Perl SMS Software Development Kit (SDK) providing high-level control of the Simplewire wireless text-messaging platform, a commercially supported product available from http://simplewire.com. Simplewire provide a comprehensive SMS service including options to receive as well as transmit SMS messages. The SDK was designed to be developer-friendly and it hides the intricacies of the XML format required to communicate with the Simplewire WMP (Wireless Message Protocol) servers.

```
sub make_SMS_notices {
    my $source = shift;
    my $style_doc = $parser->parse_file($opts{'s'});
    my $stylesheet = $xslt->parse_stylesheet($style_doc);
    my $msg = $stylesheet->transform($source);
    my $telephone = $source->findnodes("/printout/section-01/z302-telephone");

    if ($telephone =~ /(^085)|(^086)|(^087)/) {
       $telephone =~ s/[^\d]//;
       if ($telephone =~/^\d{10}$/) {
          $telephone =~ s/^0//;
          $telephone = "+353".$telephone;
          print SMSLOG "Sending to mobile $telephone\n";
          send_SMS($telephone,$stylesheet->output_string($msg));
       }
   }
}



sub send_SMS {
    my ($pin, $message) = @_;
    my $sms = Net::SMS->new();
```

```
    $sms->subscriberID("218-805-XXXY-712118");
    $sms->subscriberPassword("D72XXYZZA");

    # Message Settings
    $sms->msgPin($pin);
    $sms->msgFrom("Library, NUI Galway\n");
    $sms->msgCallback("+1 100 5551212");
    $sms->msgText($message);

    print "Sending message...\n";

    # Send Message
    $sms->msgSend();

    # Check For Errors
    if ($sms->success) {
        print SMSLOG "Message was sent!\n"; }
    else {
        print SMSLOG "Message was not sent!\n";
        print SMSLOG "Error Code: " . $sms->errorCode() . "\n";
        print SMSLOG "Error Description: " . $sms->errorDesc() ."\n";
        print SMSLOG "Error Resolution: " . $sms->errorResolution() ."\n";
        }

}
```

Fig 4 Sending SMS messages

Incorporating book cover-images and links into our notices

A novel enhancement to the existing notices setup is the inclusion of images of the relevant book covers within the notice.

These images might serve to jog a users memory with respect to the items they have on loan. When sent by email, the images provide an opportunity, in appropriate circumstances, to make a hyperlink to a bookseller like Amazon. Alternatively, an OpenURL link may be created, thus providing a route to an appropriate extended service, available via a resolver. OpenURL's are trivial to implement because we have at our disposal the item's metadata as provided by the Z13 Oracle table within the notice XML, and all this conveniently addressable via XPATH expressions.

In the case of an Amazon hyperlink, the user is presented with an opportunity to purchase the item, should they wish to do so. Where appropriate, this provides a potential small revenue stream for the library using the Amazon Associates programme.

To incorporate the images and links into the notices, we perform an *in-memory edit* of the notice-XML parse tree, utilising the DOM (Document Object Model) API as provided by the XML::LibXML module. We do this before XSLT transformation. In Galway we chose to utilise the `Z30-description` node, already a component of our notices, and therefore no changes to the existing XSLT were necessary. We populate this element with an appropriate chunk of HTML, comprising *a* and *img* elements, to make, in our case, a clickable image, with the Amazon details of the item as target, and which incorporates the Library's Amazon Associates ID within the URL . We found that utilising the DOM was not intuitive from the developers perspective, though preferable and more satisfying that using crude regular expressions to alter the

notice. The following code-snippet demonstrates using the DOM API to alter the library notice XML.

```
01  my $XMLdom      = $parser->parse_string($notice_xml);
02  my $root        = $XMLdom->getDocumentElement;
03  my $z30descrips = $root->getElementsByTagName("z30-description");
04  my $z13isbns    = $root->getElementsByTagName("z13-isbn-issn" );
05  my $pos         = 0;
07  foreach my $description ( $descriptions->get_nodelist() ) {
08    $pos++;
09    my $isbn    =($z13isbns->get_node($pos))->firstChild;
10    my $cdata   = make_lnk($description->firstChild,$isbn->data);
11    my $fragment= $parser->parse_xml_chunk( $cdata );
12    $description->appendChild( $fragment );
13  }
```

Fig 5. Using the DOM API to alter our notice XML

We parse the library-notice XML data and create the DOM object on line 1. Next, we return the document element, the top-level element or root element in our XML document on line 2. Using the `getElementsByTagName` DOM API method of our root, we return a list of descendant elements with the `z30-description` and corresponding `z13-isbn-issn` tags respectively in the variables `$z30descripts` and `$z13isbns`. From line 7 to 13 we iterate over each `z30-description` element, acquiring the corresponding ISBN for the current z30-description from the `$z13isbns` nodelist on line 9, the actual ISBN being the childnode of the `z13-isbn-issn` node. On line 10, we call our `make_lnk` subroutine with two parameters – the firstchild node of each `z30-description` and the actual ISBN. See Fig y below for details of the `make_lnk` subroutine.

```
01  sub make_lnk {
02    my ($description, $isbn) = @_;
03    $isbn ||= '9999999999';
04    $description = qq{\n
         <![CDATA[<A HREF =
          "http://myserver.edu/cgi-bin/go-pic?pic=$isbn">
          <IMG SRC="http://myserver.edu/cgi-bin/gen-pic?pic=$isbn"
          BORDER="0" alt="Click for details via Amazon"/></A>]]>\n
      };
05  return $description;
06  }
```

Fig 6 Importing the appropriate image and creating the link.

On line 2 we assign the parameters to the subroutine to two local variables, `$description` and `$isbn`. On line 3 we populate local variable `$isbn` with a dummy ISBN (999999999) if the ISBN has not been defined e.g. if the item in question does not have an ISBN, or it is otherwise undefined.
On line 4 we compose the content for our link as a chunk of HTML within an XML CDATA section and assign it to the local variable `$description`. The HTML chunk comprises hyperlinks to two web services, **`go-pic`** and **`gen-pic`**, both implemented as CGI scripts. The `go-pic` service takes an ISBN as a parameter and does a HTTP redirect to the Amazon page for the item with the corresponding ISBN. The `gen-pic` service is called by virtue of being the value of a SRC attribute within an IMG element. It also takes an ISBN as a parameter. Accordingly, `gen-pic` is executed when the HTML is rendered in the browser. The `go-pic` redirection service is only

executed if the user clicks on the image that is created from the execution of the `gen-pic` service.

Areas for further investigation.

Bounce and Reply Management – bounced notices and user replies could be forwarded to a bot which utililes takes action based on the content of the message. Bounced emails could trigger a print notice for the user. Legitimate replies by users could be forwarded to a human for response.

‘**One Click’ Renewal** – courtesy notices might include a renew button, pointing to a script that perhaps utilizes the X-server API.
In this scenario, instead of exposing the X API to the user, we could have some middleware that requires a password for each renew transaction, perhaps implemented using the user’s PIN as a private key to encrypt the query string received by this middleware. The user will not even be aware of the password. The query string complete with password could be precomputed and sent in the notice to the user to be invoked via the renew button. The renew button would only appear if the item was renewable. We could implement state for this button by making the image tag a link to program e.g.

```
<img src = "http://server/showbutton.cgi?EncryptedQueryHexString">
```

If the user opened the mail, say after the renewal period expired, the button would not appear and be replaced by an invisible one pixel transparent gif or a not renewable button.