

;login:

THE MAGAZINE OF USENIX & SAGE

June 2003 • volume 28 • number 3

inside:

PROGRAMMING

Practical Perl

by Adam Turoff

USENIX & SAGE

The Advanced Computing Systems Association &
The System Administrators Guild

practical perl: keeping it simple

Adam is a consultant who specializes in using Perl to manage big data. He is a long-time Perl Monger, a technical editor for *perl.com*, and a frequent presenter at Perl conferences.

CPAN modules are a great way to construct a program by reusing existing components. However, gluing CPAN modules together is certainly not the only way to write a Perl program. In some cases, it is both simpler and easier to write a program using CPAN modules. After all, with or without CPAN, Perl is still a great language for text hacking, automation, and application glue.

Building a program using CPAN modules is a great way to start writing a new program, and a great way to reduce development time. However, using CPAN modules adds dependencies that can complicate deployment. Remember that a program will not run unless all of its required modules are installed. In some extreme cases your program may run, but it will exhibit buggy behavior because it uses an older version of a module dependency. Although these issues are not insurmountable, they are worthy of consideration, especially in situations where a program will be installed both widely and often.

Dave Cross' NMS project (<http://nms-cgi.sourceforge.net>) is a perfect example. Dave wanted to write replacement programs for the ancient, buggy, insecure yet popular scripts found on Matt's Script Archive. Dave's replacement programs are targeted at unsophisticated users who want to add a stock feature on a Web site. Many of these users are not Perl programmers, nor do they wish to learn Perl just to install a silly guest-book script. The NMS programs use standard Perl features and core modules that are distributed with Perl, and have no dependencies on any modules found on CPAN. This makes it easy for users to just drop a file in their cgi-bin directory and get something that "just works."

I came across a similar situation recently. I maintain an online journal (Web log) called "use Perl," a community Web site for Perl programmers (my journal can be found at <http://use.perl.org/~ziggy/journal/>). I also receive email notifications when my friends post entries in their journals. I don't always have time to read these journals when they are posted, and often a few dozen will accumulate while I am busy working on a project. So I

wrote a quick little program to download the email notifications so that I can catch up on my backlog as quickly as possible.

I use procmail at my ISP to store all email notifications of "use Perl" journal postings in a separate mailbox. The easiest way for me to automate viewing a few dozen journal entries at a time is to download the mailbox file, extract the entry URL in each message, and load it in a Web browser. This certainly isn't the most important program I have ever written, but it does exhibit two virtues of being a programmer: laziness and impatience. After all, I do not want to spend hours at a time scanning through old journal entries. I'd rather read these messages as quickly as possible and move on to more interesting tasks.

First Attempt: Use CPAN Modules

I wrote my first view-useperl script about two years ago. I think it took all of 10 minutes to write. The process it automates is quite simple: Download a mailbox from my ISP, get the first URL in the body of each message in the mailbox, and display each URL in turn. I started by looking around CPAN, and I found Mark Overmeer's Mail-Box distribution. This distribution contains the Mail::Box and Mail::Box::Manager modules, which handle the key task of parsing a mailbox into a series of messages. I've used it before, and it suited my needs for this quick hack.

The first version of my view-useperl script looked something like this:

```
#!/usr/bin/perl -w

use strict;
use Mail::Box::Manager;

## (1) Download the mailbox from the ISP.
my $mbox = "/tmp/useperl.$$";
system("scp $ENV{USEPERL_MAILBOX} $mbox")
    and die "Error downloading mailbox.\n";

## (2) Open the temporary mailbox.
my $manager = new Mail::Box::Manager;
my $folder = $manager->open(folder => $mbox);

## (3) Convert the mailbox from a list of messages
## to a list of URLs. (Find the first URL in each message
## body.)
my $i = 0;
my @urls;

while(1) {
    my $msg = $folder->message($i++);
    last unless $msg;
    $msg->body() =~ m/(http:.*?)/sm;
    push(@urls, $1);
}
```

```

## (4) Close and delete the temporary mailbox.
$folder->close();
unlink($mbox);

## (5) Show the URLs, one at a time.
foreach my $url (@urls) {
    print $url;
    system "open '$url'";
    <>;
}

```

The main body of this script starts in part (1) by copying the mailbox from my ISP to a local file. The remote location of the mailbox containing my use.perl.org notifications is stored in an environment variable, USEPERL_MAILBOX. I use this technique to avoid hard-coding sensitive information in my source code. Because the location is stored in an environment variable, I can publish the source code without requiring other users to edit the program text in order to customize it.

The system call may seem counterintuitive at first. That's because system is unlike normal Perl primitives, like open, that return true on success and false on failure. Instead, system returns a nonzero failure code (true), and a zero value (false) to indicate success. This is consistent with the behavior of the system call in the standard C library. Although it made sense for Perl to adopt the C-style behavior many years ago when most Perl programmers had some background in C, the situation is quite the opposite today. Perl6 will reverse this legacy behavior so that the standard "open or die" idiom can be used with system as well, and do away with the current, counterintuitive "system and die" usage.

The code in part (2) creates a Mail::Box object to scan through the messages in the mailbox I just downloaded. The Mail::Box interface requires that I first create a manager object, and use a factory method on that object to create a folder object, \$folder.

The while loop grabs each email message in the folder, one at a time, and terminates when there are no more messages to retrieve. It then locates the first URL in each message body, and appends that to a list of URLs.

Part (4) is some basic housekeeping code to close the folder object and delete the local copy of the mail folder.

The real value of this program is found in part (5). Finally, I have a list of URLs I want to load in my browser. Because MacOS X is my platform of choice, I use the standard open command to open each URL. This command will intelligently open a filename using the appropriate application. In this case, when I pass a URL to the open command, it will load that Web page in my preferred Web browser. On another system, I could write a small program called open that is just smart enough to

take a URL and load it in a Web browser. Alternatively, I could replace open with a mozilla -remote or similar command.

I usually run this program when a few dozen emails have accumulated in my mailbox. Opening a few dozen browser windows all at once is a great way to consume a lot of memory, saturate my network connection, and generally make my computer quite sluggish. Instead, I ask for a line of input after each Web page is loaded. This allows me to load URLs quickly, yet only open a few at a time. The input is irrelevant, so a simple return will allow the program to continue and load the next URL.

Problems with Dependencies

As I mentioned before, I wrote this script about two years ago. Since then, my main computer died, I purchased a replacement, reinstalled an OS on my laptop a few times, and have at least two Perl installations on each machine I regularly use (both version 5.6.x and 5.8.0). From a systems management perspective, it's been an eventful few months.

My view-useperl script is always one of the first things I install in my home directory on a new machine. With all of the shuffling, I've come to regret using Mail::Box for this little script. After all, if it only took 10 minutes to write, why should I need to spend another few minutes on each new machine to install a dependency when I use my program on a new machine? Mail::Box certainly is *not* a bad module, but this is the only program I use that requires it. Eliminating the dependency will make it easier to copy my script around as I switch machines (and Perl installations).

Once I came to this realization, I saw it was time to rewrite my program to not use Mail::Box anymore, and just process the mailbox files directly.

Second Attempt: No CPAN Modules

Mail messages in a mailbox file start with a line that contains the word From, a space, an email address, and a date. (This is why a line in the body of an email message that begins with "From" usually has a ">" character preceding it.) Mail messages are also separated by a blank line preceding the From line.

With this information at hand, I decided to rewrite view-useperl. I can view a mailbox file as a series of email records, and then use standard Perl operators to convert a mailbox file into a sequence of relevant URLs.

The email messages I am processing are consistently formatted. The first URL in each message is the location of the journal entry I want to view. Also, there are no URLs found in the message headers, so I don't even need to bother splitting the message header from the message body. The first URL I encounter

in each message will be the URL I want to view. Subsequent URLs will appear in each message, and they should be ignored.

Here is the updated `view-useperl` script that takes advantage of these observations:

```
#!/usr/bin/perl -w
use strict;

sub read_mailbox {
    my $mbox = shift;

    ## Read a sequence of messages,
    ## delimited by a blank line and 'From'.
    local $/ = "\n\nFrom ";

    open(my $fh, $mbox);
    my @urls = map {m/(http:.*?)/sm; $1} <$fh>;
    close($fh);
    unlink $mbox;

    return @urls;
}

## (1) Download the mailbox from the ISP.
my $mbox = "/tmp/useperl.$$";
system("scp $ENV{USEPERL_MAILBOX} $mbox")
    and die "Error downloading mailbox.\n";

## (2) Read the URLs found in the mailbox.
my @urls = read_mailbox($mbox);

## (3) View each URL in turn.
foreach my $url (@urls) {
    print $url;
    system "open '$url'";
    <>;
}
```

Not only does this program avoid CPAN modules, but it is slightly shorter and a little easier to read. Because it does not have any external dependencies, I can expect it to work wherever I copy it (so long as I define my `USEPERL_MAILBOX` environment variable).

The overall structure of this is unchanged. First, fetch the mailbox. Next, convert a set of email messages to a list of URLs. Finally, display the URLs, one at a time. And this sequence of steps is clearly stated in the main program text. The more complicated process of converting a mailbox into a list of URLs is now handled by an appropriately named sub, `read_mailbox`. This sub eliminates the need to create `Mail::Box` objects, yet it performs the same task: converting a mailbox into a set of email messages, extracting the first URL from each message, and returning a list of URLs.

The first thing `read_mailbox` does is modify the input record separator, stored in the `$/` special variable. This variable con-

tains a new-line character by default, and that is why file input operations generally occur one line at a time. Because a mailbox is nothing more than a concatenation of email messages, I can specify a sequence of characters found between email messages as a record separator. When I read the mailbox file in list context, I receive a meaningful list of email messages, not a meaningless list of lines in a file.

The `$/` variable is used globally within a Perl program. Changing its value is bad style, unless changes are localized to a specific scope. Here, the statement `local $/ = "\n\nFrom "` modifies the value of the input record separator within the `read_mailbox` sub and any subs that it calls. When `read_mailbox` exits, the previous value of `$/` is restored. This is important, because the console input operation at the end of the program expects the record separator to be a new line, not `"\n\nFrom "`.

Now that `$/` has been modified appropriately, reading messages is a breeze. First, I open the local copy of the mailbox. Next, I read the mailbox in as a list of messages (`my @urls = ... && $fh`). But the messages themselves aren't very meaningful in this particular program. In fact, each message is just a stream of meaningless text, followed by a URL, followed by more meaningless text. The `map` operation transforms the list of email messages read in from `&& $fh`; and replaces each chunk of text with the first URL in that chunk. These URLs are then gathered together into the list `@urls`.

In effect, a few lines of module initialization and method calls in the original program are replaced with one line to reset the value of `$/` and one line to read and convert a mailbox into a list of URLs. This is the power of Perl.

Observations

CPAN is certainly the best thing that has ever happened to Perl. However, Perl without CPAN is not too shabby. Sometimes, the easiest or the best solution to a problem is to *avoid* CPAN. In the example of my `view-useperl` program, the before and after versions are equally good. However, the modified version has a very attractive property – it does not require me to install `Mail::Box` on each new computer I use.

The reason why I chose to avoid `Mail::Box` has nothing to do with the quality of that module, but just reflects the desire to remove an external dependency for `view-useperl`. If my little program were doing something more complicated, like deleting some messages in a mailbox file, or selecting messages based on header characteristics, then I certainly would have kept using it. Instead, I chose to rewrite this program as a common needle-in-a-haystack type of solution. The fact that the data file was a mailbox is largely irrelevant here.

There are other circumstances where avoiding CPAN modules is simply unwise. When using a relational database, there's no good reason to avoid the DBI family of modules. Similarly, there's no reason to start with raw socket programming to fetch Web pages when the LWP library has been doing a great job for many years. And the list goes on and on.

Conclusion

Whether you are writing a quick hack or a program of significant size, there are many very good reasons to start with CPAN modules to make your job easier. However, there are also some circumstances where there are benefits to avoiding CPAN modules. Remember that both options are available to you. Choose wisely.

USENIX and SAGE Need You

People often ask how they can contribute. Here is a list of tasks for which we hope to find volunteers.

The SAGEwire and SAGEweb staff are seeking:

- Interview candidates
- Short article contributors (see <http://sagewire.sage.org>)
- White paper contributors for topics like these:

Back-ups	Emerging technology	Privacy
Career development	User education/training	Product round-ups
Certification	Ethics	SAGEwire
Consulting	Great new products	Scaling
Culture	Group tools	Scripting
Databases	Networking	Security implementation
Displays	New challenges	Standards
Email	Performance analysis	Storage
Education	Politics and the sysadmin	Tools, system
- Local user groups: If you have a local user group affiliated (or wishing to affiliate) with SAGE, please email the particulars to kolstad@sage.org so they can be posted on the Web site.

`login`: always needs conference summarizers for USENIX conferences. Contact Alain Hénon, ah@usenix.org, if you'd like to help.