ADVANCED CGI

Author: Lincoln Stein <lstein@cshl.org> Date: 7/26/99

This tutorial begins a collection of CGI scripts that illustrate the four basic types of CGI scripting: dynamic documents, document filtering, and URL redirection. It also shows a few tricks that you might not have run into -- or even thought were possible with CGI.

It continues with the next step beyond CGI scripting: the creation of high performance Apache modules with the mod_perl API.

Part I -- TRICKS WITH CGI.PM

Dynamic Documents

The most familiar use of CGI is to create documents on the fly. They can be simple documents, or get incredibly baroque. We won't venture much past the early baroque.

Making HTML look beautiful

<I> <hate> <HTML> <because> <it's> <ugly> <and> <has> <too> <many> <$#@*&> <angle> <brackets>. With CGI.pm it's almost good to look at. Script I.1.1 shows what a nested list looks like with CGI.pm.

Script I.1.1: vegetables1.pl

```
#!/usr/bin/perl
# Script: vegetables1.pl
use CGI::Pretty ':standard';
print header,
   start_html('Vegetables'),
   h1('Eat Your Vegetables'),
   ol(
      li('peas'),
      li('broccoli'),
      li('cabbage'),
      li('peppers',
         ul(
            li('red'),
            li('yellow'),
            li('green')
            )
```

```
        ),
      li('kolrabi'),
      li('radishes')
      ),
   hr,
   end_html;
```

Making HTML concise

But we can do even better than that because CGI.pm lets you collapse repeating tags by passing array references to its functions. Script 1.2 saves some typing, and in so doing, puts off the onset of RSI by months or years!

Script I.1.2: vegetables2.pl

```
#!/usr/bin/perl
# Script: vegetables2.pl
use CGI ':standard';
print header,
   start_html('Vegetables'),
   h1('Eat Your Vegetables'),
   ol(
      li(['peas',
          'broccoli',
          'cabbage',
          'peppers' .
          ul(['red','yellow','green']),
          'kolrabi',
          'radishes'
      ),
   hr,
   end_html;
```

Or how about this one?

Script I.1.3: vegetables3.pl

```
#!/usr/bin/perl

# Script: vegetables3.pl
use CGI::Pretty qw/:standard :html3/;

print header,
   start_html('Vegetables'),
   h1('Vegetables are for the Strong'),
   table({-border=>undef},
         caption(strong('When Should You Eat Your Vegetables?')),
         Tr({-align=>CENTER,-valign=>TOP},
            [
             th(['','Breakfast','Lunch','Dinner']),
             th('Tomatoes').td(['no','yes','yes']),
             th('Broccoli').td(['no','no','yes']),
             th('Onions').td(['yes','yes','yes'])
             ]
            )
```

```
        ),
    end_html;
```

Making Interactive Forms

Of course you mostly want to use CGI to create interactive forms. No problem! CGI.pm has a full set of functions for both generating the form and reading its contents once submitted. Script I.1.4 creates a row of radio buttons labeled with various colors. When the user selects a button and submits the form, the page redraws itself with the selected background color. Psychedelic!

Script I.1.4: customizable.pl

```
#!/usr/bin/perl
# script: customizable.pl

use CGI::Pretty qw/:standard/;

$color = param('color') || 'white';

print header,
    start_html({-bgcolor=>$color},'Customizable Page'),
    h1('Customizable Page'),
    "Set this page's background color to:",br,
    start_form,
    radio_group(-name=>'color',
                -value=>['white','red','green','black',
                         'blue','silver','cyan'],
                -cols=>2),
    submit(-name=>'Set Background'),
    end_form,
    p,
    hr,
    end_html;

=head1 Making Stateful Forms
```

Many real Web applications are more than a single page. Some may span multiple pages and fill-out forms. When the user goes from one page to the next, you've got to save the state of the previous page somewhere. A convenient and cheap place to put state information is in hidden fields in the form itself. Script I.2.1 is an example of a loan application with a total of five separate pages. Forward and back buttons allows the user to navigate between pages. The script remembers all the pages and summarizes them up at the end.

Script I.2.1: loan.pl

```
#!/usr/local/bin/perl

# script: loan.pl
use CGI qw/:standard :html3/;

# this defines the contents of the fill out forms
# on each page.
@PAGES = ('Personal Information','References','Assets','Review','Confirmation');
```

```
%FIELDS = ('Personal Information' => ['Name','Address','Telephone','Fax'],
          'References'           => ['Personal Reference 1','Personal Reference 2']
          'Assets'               => ['Savings Account','Home','Car']
          );
# accumulate the field names into %ALL_FIELDS;
foreach (values %FIELDS) {
    grep($ALL_FIELDS{$_}++,@$_);
}


# figure out what page we're on and where we're heading.
$current_page = calculate_page(param('page'),param('go'));
$page_name = $PAGES[$current_page];

print_header();
print_form($current_page)         if $FIELDS{$page_name};
print_review($current_page)       if $page_name eq 'Review';
print_confirmation($current_page) if $page_name eq 'Confirmation';
print end_html;

# CALCULATE THE CURRENT PAGE
sub calculate_page {
    my ($prev,$dir) = @_;
    return 0 if $prev eq '';   # start with first page
    return $prev + 1 if $dir eq 'Submit Application';
    return $prev + 1 if $dir eq 'Next Page';
    return $prev - 1 if $dir eq 'Previous Page';
}

# PRINT HTTP AND HTML HEADERS
sub print_header {
    print header,
    start_html("Your Friendly Family Loan Center"),
    h1("Your Friendly Family Loan Center"),
    h2($page_name);
}

# PRINT ONE OF THE QUESTIONNAIRE PAGES
sub print_form {
    my $current_page = shift;
    print "Please fill out the form completely and accurately.",
       start_form,
       hr;
    draw_form(@{$FIELDS{$page_name}});
    print hr;
    print submit(-name=>'go',-value=>'Previous Page')
       if $current_page > 0;
    print submit(-name=>'go',-value=>'Next Page'),
       hidden(-name=>'page',-value=>$current_page,-override=>1),
       end_form;
}

# PRINT THE REVIEW PAGE
sub print_review {
    my $current_page = shift;
    print "Please review this information carefully before submitting it. ",
       start_form;
    my (@rows);
    foreach $page ('Personal Information','References','Assets') {
       push(@rows,th({-align=>LEFT},em($page)));
       foreach $field (@{$FIELDS{$page}}) {
```

```
            push(@rows,
                 TR(th({-align=>LEFT},$field),
                    td(param($field)))
                 );
            print hidden(-name=>$field);
        }
    }
    print table({-border=>1},caption($page),@rows),
       hidden(-name=>'page',-value=>$current_page,-override=>1),
       submit(-name=>'go',-value=>'Previous Page'),
       submit(-name=>'go',-value=>'Submit Application'),
       end_form;
}

# PRINT THE CONFIRMATION PAGE
sub print_confirmation {
    print "Thank you. A loan officer will be contacting you shortly.",
       p,
       a({-href=>'../source.html'},'Code examples');
}


# CREATE A GENERIC QUESTIONNAIRE
sub draw_form {
    my (@fields) = @_;
    my (%fields);
    grep ($fields{$_}++,@fields);
    my (@hidden_fields) = grep(!$fields{$_},keys %ALL_FIELDS);
    my (@rows);
    foreach (@fields) {
       push(@rows,
            TR(th({-align=>LEFT},$_),
               td(textfield(-name=>$_,-size=>50))
               )
            );
    }
    print table(@rows);

    foreach (@hidden_fields) {
        print hidden(-name=>$_);
    }
}
```

Keeping State with Cookies

If you want to maintain state even if the user quits the browser and comes back again, you can use cookies. Script I.2.2 records the user's name and color scheme preferences and recreates the page the way the user likes up to 30 days from the time the user last used the script.

Script I.2.2: preferences.pl

```
#!/usr/local/bin/perl

# file: preferences.pl

use CGI qw(:standard :html3);
```

```
# Some constants to use in our form.
@colors=qw/aqua black blue fuschia gray green lime maroon navy olive
    purple red silver teal white yellow/;
@sizes=("<default>",1..7);

# recover the "preferences" cookie.
%preferences = cookie('preferences');

# If the user wants to change the background color or her
# name, they will appear among our CGI parameters.
foreach ('text','background','name','size') {
    $preferences{$_} = param($_) || $preferences{$_};
}

# Set some defaults
$preferences{'background'} = $preferences{'background'} || 'silver';
$preferences{'text'} = $preferences{'text'} || 'black';

# Refresh the cookie so that it doesn't expire.
$the_cookie = cookie(-name=>'preferences',
                     -value=>\%preferences,
                     -path=>'/',
                     -expires=>'+30d');
print header(-cookie=>$the_cookie);

# Adjust the title to incorporate the user's name, if provided.
$title = $preferences{'name'} ?
    "Welcome back, $preferences{name}!" : "Customizable Page";

# Create the HTML page.  We use several of the HTML 3.2
# extended tags to control the background color and the
# font size.  It's safe to use these features because
# cookies don't work anywhere else anyway.
print start_html(-title=>$title,
                 -bgcolor=>$preferences{'background'},
                 -text=>$preferences{'text'}
                );

print basefont({-size=>$preferences{size}}) if $preferences{'size'} > 0;

print h1($title);

# Create the form
print hr,
    start_form,

    "Your first name: ",
    textfield(-name=>'name',
              -default=>$preferences{'name'},
              -size=>30),br,

    table(
         TR(
            td("Preferred"),
            td("Page color:"),
            td(popup_menu(-name=>'background',
                          -values=>\@colors,
                          -default=>$preferences{'background'})
               ),
            ),
```

```
        TR(
           td(''),
           td("Text color:"),
           td(popup_menu(-name=>'text',
                         -values=>\@colors,
                         -default=>$preferences{'text'})
              )
           ),
        TR(
           td(''),
           td("Font size:"),
           td(popup_menu(-name=>'size',
                         -values=>\@sizes,
                         -default=>$preferences{'size'})
              )
           )
        ),

    submit(-label=>'Set preferences'),
    end_form,
    hr,
    end_html;
```

Creating Non-HTML Types

CGI can do more than just produce HTML documents. It can produce any type of document that you can output with Perl. This includes GIFs, Postscript files, sounds or whatever.

Script I.3.1 creates a clickable image map of a colored circle inside a square. The script is responsible both for generating the map and making the image (using the GD.pm library). It also creates a fill-out form that lets the user change the size and color of the image!

Script I.3.1: circle.pl

```
#!/usr/local/bin/perl

# script: circle.pl
use GD;
use CGI qw/:standard :imagemap/;

use constant RECTSIZE     => 100;
use constant CIRCLE_RADIUS  => 40;
%COLORS = (
          'white' => [255,255,255],
          'red'   => [255,0,0],
          'green' => [0,255,0],
          'blue'  => [0,0,255],
          'black' => [0,0,0],
          'bisque'=> [255,228,196],
          'papaya whip' => [255,239,213],
          'sienna' => [160,82,45]
          );

my $draw          = param('draw');
```

```
my $circle_color  = param('color') || 'bisque';
my $mag           = param('magnification') || 1;

if ($draw) {
    draw_image();
} else {
    make_page()
}

sub draw_image {
    # create a new image
    my $im = new GD::Image(RECTSIZE*$mag,RECTSIZE*$mag);

    # allocate some colors
    my $white = $im->colorAllocate(@{$COLORS{'white'}});
    my $black = $im->colorAllocate(@{$COLORS{'black'}});
    my $circlecolor = $im->colorAllocate(@{$COLORS{$circle_color}});

    # make the background transparent and interlaced
    $im->transparent($white);
    $im->interlaced('true');

    # Put a black frame around the picture
    $im->rectangle(0,0,RECTSIZE*$mag-1,RECTSIZE*$mag-1,$black);

    # Draw the circle
    $im->arc(RECTSIZE*$mag/2,RECTSIZE*$mag/2,CIRCLE_RADIUS*$mag*2,CIRCLE_RADIUS*$ma

    # And fill it with circlecolor
    $im->fill(RECTSIZE*$mag/2,RECTSIZE*$mag/2,$circlecolor);

    # Convert the image to GIF and print it
    print header('image/gif'),$im->gif;
}

sub make_page {
   print header(),
   start_html(-title=>'Feeling Circular',-bgcolor=>'white'),
   h1('A Circle is as a Circle Does'),
   start_form,
   "Magnification: ",radio_group(-name=>'magnification',-values=>[1..4]),br,
   "Color: ",popup_menu(-name=>'color',-values=>[sort keys %COLORS]),
   submit(-value=>'Change'),
   end_form;
   print em(param('message') || 'click in the drawing' );

   my $url = url(-relative=>1,-query_string=>1);
   $url .= '?' unless param();
   $url .= '&draw=1';

   print p(
       img({-src=>$url,
            -align=>'LEFT',
            -usemap=>'#map',
            -border=>0}));

   print Map({-name=>'map'},
             Area({-shape=>'CIRCLE',
                   -href=>param(-name=>'message',-value=>"You clicked in the circle
                       && url(-relative=>1,-query_string=>1),
                   -coords=>join(',',RECTSIZE*$mag/2,RECTSIZE*$mag/2,CIRCLE_RADIUS*
```

```
                  -alt=>'Circle'}),
              Area({-shape=>'RECT',
                  -href=>param(-name=>'message',-value=>"You clicked in the square
                      && url(-relative=>1,-query_string=>1),
                  -coords=>join(',',0,0,RECTSIZE*$mag,RECTSIZE*$mag),
                  -alt=>'Square'}));
   print end_html;
}

Script I.3.2 creates a GIF89a animation.  First it creates a set of
simple GIFs, then uses the I<combine> program (part of the ImageMagick
package) to combine them together into an animation.
```

I'm not a good animator, so I can't do anything fancy. But you can!

Script I.3.2: animate.pl

```
#!/usr/local/bin/perl

# script: animated.pl
use GD;
use File::Path;

use constant START      => 80;
use constant END        => 200;
use constant STEP       => 10;
use constant COMBINE    => '/usr/local/bin/convert';
@COMBINE_OPTIONS = (-delay => 5,
                    -loop  => 10000);

@COLORS = ([240,240,240],
           [220,220,220],
           [200,200,200],
           [180,180,180],
           [160,160,160],
           [140,140,140],
           [150,120,120],
           [160,100,100],
           [170,80,80],
           [180,60,60],
           [190,40,40],
           [200,20,20],
           [210,0,0]);
@COLORS = (@COLORS,reverse(@COLORS));

my @FILES = ();
my $dir = create_temporary_directory();
my $index = 0;
for (my $r = START; $r <= END; $r+=STEP) {
    draw($r,$index,$dir);
    $index++;
}
for (my $r = END; $r > START; $r-=STEP) {
    draw($r,$index,$dir);
    $index++;
}

# emit the GIF89a
$| = 1;
print "Content-type: image/gif\n\n";
```

```
system COMBINE,@COMBINE_OPTIONS,@FILES,"gif:-";

rmtree([$dir],0,1);

sub draw{
    my ($r,$color_index,$dir) = @_;
    my $im = new GD::Image(END,END);
    my $white = $im->colorAllocate(255,255,255);
    my $black = $im->colorAllocate(0,0,0);
    my $color = $im->colorAllocate(@{$COLORS[$color_index % @COLORS]});
    $im->rectangle(0,0,END,END,$white);
    $im->arc(END/2,END/2,$r,$r,0,360,$black);
    $im->fill(END/2,END/2,$color);
    my $file = sprintf("%s/picture.%02d.gif",$dir,$color_index);
    open (OUT,">$file") || die "couldn't create $file: $!";
    print OUT $im->gif;
    close OUT;
    push(@FILES,$file);

}

sub create_temporary_directory {
    my $basename = "/usr/tmp/animate$$";
    my $counter=0;
    while ($counter < 100) {
       my $try = sprintf("$basename.%04d",$counter);
       next if -e $try;
       return $try if mkdir $try,0700;
    } continue { $counter++; }
    die "Couldn't make a temporary directory";
}

=head1 Document Translation
```

Did you know that you can use a CGI script to translate other documents on the fly? No s**t! Script I.4.1 is a script that intercepts all four-letter words in text documents and stars out the naughty bits. The document itself is specified using additional path information. We're a bit over-literal about what a four-letter word is, but what's the fun if you can't be extravagant?

Script I.4.1: naughty.pl

```
#!/usr/local/bin/perl
# Script: naughty.pl

use CGI ':standard';
$file = path_translated() ||
       die "must be called with additional path info";
open (FILE,$file) || die "Can't open $file: $!\n";
print header('text/plain');
while (<FILE>) {
   s/\b(\w)\w{2}(\w)\b/$1**$2/g;
   print;
}
close FILE;
```

4.1 won't work on HTML files because the HTML tags will get starred out too. If you find it a little limiting to work only on plain-text files, script I.4.2 uses LWP's HTML parsing functions to modify just the text part of an HTML document without touching the tags. The script's a little awkward because we

have to guess the type of file from the extension, and *redirect* when we're dealing with a non-HTML file. We can do better with *mod_perl*.

Script I.4.2: naughty2.pl

```
#!/usr/local/bin/perl

# Script: naughty2.pl
package HTML::Parser::FixNaughty;

require HTML::Parser;
@ISA = 'HTML::Parser';

sub start {
    my ($self,$tag,$attr,$attrseq,$origtext) = @_;
    print $origtext;
}
sub end {
    my ($self,$tag) = @_;
    print "</$tag>";
}
sub text {
    my ($self,$text) = @_;
    $text =~ s/\b(\w)\w{2}(\w)\b/$1**$2/g;
    print $text;
}


package main;
use CGI qw/header path_info redirect path_translated/;

$file = path_translated() ||
    die "must be called with additional path info";
$file .= "index.html" if $file=~m!/$!;

unless ($file=~/\.html?$/) {
    print redirect(path_info());
    exit 0;
}

$parser = new HTML::Parser::FixNaughty;
print header();
$parser->parse_file($file);
```

A cleaner way to do this is to make naughty2.pl an Apache HANDLER. We can make it handle all the HTML documents on our site by adding something like this to access.conf:

```
  <Location />
    ... blah blah blah other stuff
    Action text/html /cgi-bin/naughty2.pl
  </Location>
```

Now, whenever an HTML document is requested, it gets passed through the CGI script. With this in place, there's no need to check the file type and redirect. Cool!

Smart Redirection

There's no need even to create a document with CGI. You can simply *redirect* to the URL you want. Script I.4.3 chooses a random picture from a directory somewhere and displays it. The directory to pick from is specified as additional path information, as in:

```
/cgi-bin/random_pict/banners/egregious_advertising
```

Script I.4.3 random_pict.pl

```
#!/usr/local/bin/perl
# script: random_pict.pl

use CGI qw/:standard/;
$PICTURE_PATH = path_translated();
$PICTURE_URL = path_info();
chdir $PICTURE_PATH
       or die "Couldn't chdir to pictures directory: $!";
@pictures = <*.{jpg,gif}>;
$lucky_one = $pictures[rand(@pictures)];
die "Failed to pick a picture" unless $lucky_one;

print redirect("$PICTURE_URL/$lucky_one");
```

File Uploads

Everyone wants to do it. I don't know why. Script I.5.1 shows a basic script that accepts a file to upload, reads it, and prints out its length and MIME type. Windows users should read about `binmode()` before they try this at home!

Script I.5.1 upload.pl

```
#!/usr/local/bin/perl
#script: upload.pl

use CGI qw/:standard/;

print header,
    start_html('file upload'),
    h1('file upload');
print_form()    unless param;
print_results() if param;
print end_html;

sub print_form {
    print start_multipart_form(),
       filefield(-name=>'upload',-size=>60),br,
       submit(-label=>'Upload File'),
       end_form;
}

sub print_results {
```

```
    my $length;
    my $file = param('upload');
    if (!$file) {
       print "No file uploaded.";
       return;
    }
    print h2('File name'),$file;
    print h2('File MIME type'),
    uploadInfo($file)->{'Content-Type'};
    while (<$file>) {
       $length += length($_);
    }
    print h2('File length'),$length;
}
```

Part II: MOD_PERL -- FASTER THAN A SPEEDING BULLET

mod_perl is Doug MacEachern's embedded Perl for Apache. With a mod_perl-enabled server, there's no tedious waiting around while the Perl interpreter fires up, reads and compiles your script. It's right there, ready and waiting. What's more, once compiled your script remains in memory, all charged and raring to go. Suddenly those sluggish Perl CGI scripts race along at compiled C speeds...or so it seems.

Most CGI scripts will run unmodified under mod_perl using the Apache::Registry CGI compatability layer. But that's not the whole story. The exciting part is that mod_perl gives you access to the Apache API, letting you get at the innards of the Apache server and change its behavior in powerful and interesting ways. This section will give you a feel for the many things that you can do with mod_perl.

Creating Dynamic Pages

This is a ho-hum because you can do it with CGI and with Apache::Registry. Still, it's worth seeing a simple script written using the strict mod_perl API so you see what it looks like. Script II.1.1 prints out a little hello world message.

Install it by adding a section like this one to one of the configuration files:

```
  <Location /hello/world>
    SetHandler  perl-script
    PerlHandler Apache::Hello
  </Location>
```

----------Script II.1.1 Apache::Hello ----------

```
 package Apache::Hello;
 # file: Apache::Hello.pm

 use strict vars;
```

```
use Apache::Constants ':common';

sub handler {
    my $r = shift;
    $r->content_type('text/html');
    $r->send_http_header;
    my $host = $r->get_remote_host;
    $r->print(<<END);
<HTML>
<HEADER>
<TITLE>Hello There</TITLE>
</HEADER>
<BODY>
<H1>Hello $host</H1>
Hello to all the nice people at the Perl conference.  Lincoln is
trying really hard.  Be kind.
</BODY>
</HTML>
END
    return OK;
}
1;
```

You can do all the standard CGI stuff, such as reading the query string, creating fill-out forms, and so on. In fact, CGI.pm works with mod_perl, giving you the benefit of sticky forms, cookie handling, and elegant HTML generation.

File Filters

This is where the going gets fun. With mod_perl, you can install a *content handler* that works a lot like a four-letter word starrer-outer, but a lot faster.

Adding a Canned Footer to Every Page

Script II.2.1 adds a canned footer to every HTML file. The footer contains a copyright statement, plus the modification date of the file. You could easily extend this to add other information, such as a page hit counter, or the username of the page's owner.

This can be installed as the default handler for all files in a particular subdirectory like this:

```
<Location /footer>
  SetHandler perl-script
  PerlHandler Apache::Footer
</Location>
```

Or you can declare a new ''.footer'' extension and arrange for all files with this extension to be passed through the footer module:

```
 AddType text/html .footer
 <Files ~ "\.footer$">
    SetHandler  perl-script
    PerlHandler Apache::Footer
 </Files>
```

----------Script II.2.1 Apache::Footer ----------

```
package Apache::Footer;
# file Apache::Footer.pm

use strict vars;
use Apache::Constants ':common';
use IO::File;

sub handler {
    my $r = shift;
    return DECLINED unless $r->content_type() eq 'text/html';
    my $file = $r->filename;
    return DECLINED unless $fh=IO::File->new($file);
    my $modtime = localtime((stat($file))[9]);
    my $footer=<<END;
<hr>
&copy; 1998 <a href="http://www.ora.com/";>O\'Reilly &amp; Associates</a><br>
<em>Last Modified: $modtime</em>
END
;
    $r->send_http_header;

    while (<$fh>) {
        s!(</BODY>)!$footer$1!oi;
    } continue {
        $r->print($_);
    }

    return OK;
}

1;
```

Dynamic Navigation Bar

Sick of hand-coding navigation bars in every HTML page? Less than enthused by the Java & JavaScript hacks? Here's a dynamic navigation bar implemented as a server side include.

First create a global configuration file for your site. The first column is the top of each major section. The second column is the label to print in the navigation bar

```
# Configuration file for the navigation bar
/index.html             Home
/new/                   What's New
/tech/                  Tech Support
/download/              Download
```

```
/dev/zero               Customer support
/dev/null               Complaints
```

Then, at the top (or bottom) of each HTML page that you want the navigation bar to appear on, add this comment:

```
<!--#NAVBAR-->
```

Now add Apache::NavBar to your system (Script II.2.2). This module parses the configuration file to create a ‘‘navigation bar object’’. We then call the navigation bar object’s `to_html()` method in order to generate the HTML for the navigation bar to display on the current page (it will be different for each page, depending on what major section the page is in).

The next section does some checking to avoid transmitting the page again if it is already cached on the browser. The effective last modified time for the page is either the modification time of its HTML source code, or the navbar’s configuration file modification date, whichever is more recent.

The remainder is just looping through the file a section at a time, searching for the <!--NAVBAR--> comment, and substituting the navigation bar HTML.

----------Script II.2.2 Apache::NavBar ----------

```
package Apache::NavBar;
# file Apache/NavBar.pm

use strict;
use Apache::Constants qw(:common);
use Apache::File ();

my %BARS = ();
my $TABLEATTS   = 'WIDTH="100%" BORDER=1';
my $TABLECOLOR  = '#C8FFFF';
my $ACTIVECOLOR = '#FF0000';

sub handler {
    my $r = shift;

    my $bar = read_configuration($r)         || return DECLINED;
    $r->content_type eq 'text/html'          || return DECLINED;
    my $fh = Apache::File->new($r->filename) || return DECLINED;
    my $navbar = $bar->to_html($r->uri);

    $r->update_mtime($bar->modified);
    $r->set_last_modified;
    my $rc = $r->meets_conditions;
    return $rc unless $rc == OK;

    $r->send_http_header;
    return OK if $r->header_only;

    local $/ = "";
    while (<$fh>) {
       s:<!--NAVBAR-->:$navbar:oi;
    } continue {
       $r->print($_);
    }
```

```
    return OK;
}

# read the navigation bar configuration file and return it as a
# hash.
sub read_configuration {
    my $r = shift;
    my $conf_file;
    return unless $conf_file = $r->dir_config('NavConf');
    return unless -e ($conf_file = $r->server_root_relative($conf_file));
    my $mod_time = (stat _)[9];
    return $BARS{$conf_file} if $BARS{$conf_file}
      && $BARS{$conf_file}->modified >= $mod_time;
    return $BARS{$conf_file} = NavBar->new($conf_file);
}

package NavBar;

# create a new NavBar object
sub new {
    my ($class,$conf_file) = @_;
    my (@c,%c);
    my $fh = Apache::File->new($conf_file) || return;
    while (<$fh>) {
       chomp;
       s/^\s+//; s/\s+$//;   #fold leading and trailing whitespace
       next if /^#/ || /^$/; # skip comments and empty lines
       next unless my($url, $label) = /^(\S+)\s+(.+)/;
       push @c, $url;     # keep the url in an ordered array
       $c{$url} = $label; # keep its label in a hash
    }
    return bless {'urls' => \@c,
                  'labels' => \%c,
                  'modified' => (stat $conf_file)[9]}, $class;
}

# return ordered list of all the URIs in the navigation bar
sub urls  { return @{shift->{'urls'}}; }

# return the label for a particular URI in the navigation bar
sub label { return $_[0]->{'labels'}->{$_[1]} || $_[1]; }

# return the modification date of the configuration file
sub modified { return $_[0]->{'modified'}; }

sub to_html {
    my $self = shift;
    my $current_url = shift;
    my @cells;
    for my $url ($self->urls) {
       my $label = $self->label($url);
       my $is_current = $current_url =~ /^$url/;
       my $cell = $is_current ?
           qq(<FONT COLOR="$ACTIVECOLOR">$label</FONT>)
               : qq(<A HREF="$url">$label</A>);
       push @cells,
       qq(<TD CLASS="navbar" ALIGN=CENTER BGCOLOR="$TABLECOLOR">$cell</TD>\n);
    }
    return qq(<TABLE $TABLEATTS><TR>@cells</TR></TABLE>\n);
}
```

```
1;
__END__

<Location />
  SetHandler  perl-script
  PerlHandler Apache::NavBar
  PerlSetVar  NavConf etc/navigation.conf
</Location>
```

On-the-Fly Compression

WU-FTP has a great feature that automatically gzips a file if you fetch it by name with a .gz extension added. Why can't Web servers do that trick? With Apache and mod_perl, you can.

Script II.2.4 is a content filter that automatically gzips everything retrieved from a particular directory and adds the ''gzip'' Content-Encoding header to it. Unix versions of Netscape Navigator will automatically recognize this encoding type and decompress the file on the fly. Windows and Mac versions don't. You'll have to save to disk and decompress, or install the WinZip plug-in. Bummer.

The code uses Compress::Zlib module, and has to do a little fancy footwork (but not too much) to create the correct gzip header. You can extend this idea to do on-the-fly encryption, or whatever you like.

Here's the configuration entry you'll need. Everything in the /compressed directory will be compressed automagically.

```
<Location /compressed>
   SetHandler  perl-script
   PerlHandler Apache::GZip
</Location>
```

---------- Script II.2.3: Apache::GZip ----------

```
package Apache::GZip;
#File: Apache::GZip.pm

use strict vars;
use Apache::Constants ':common';
use Compress::Zlib;
use IO::File;
use constant GZIP_MAGIC => 0x1f8b;
use constant OS_MAGIC => 0x03;

sub handler {
    my $r = shift;
    my ($fh,$gz);
    my $file = $r->filename;
    return DECLINED unless $fh=IO::File->new($file);
    $r->header_out('Content-Encoding'=>'gzip');
    $r->send_http_header;
    return OK if $r->header_only;
```

```
    tie *STDOUT,'Apache::GZip',$r;
    print($_) while <$fh>;
    untie *STDOUT;
    return OK;
}

sub TIEHANDLE {
    my($class,$r) = @_;
    # initialize a deflation stream
    my $d = deflateInit(-WindowBits=>-MAX_WBITS()) || return undef;

    # gzip header -- don't ask how I found out
    $r->print(pack("nccVcc",GZIP_MAGIC,Z_DEFLATED,0,time(),0,OS_MAGIC));

    return bless { r   => $r,
                   crc =>  crc32(undef),
                   d   => $d,
                   l   =>  0
                   },$class;
}

sub PRINT {
    my $self = shift;
    foreach (@_) {
       # deflate the data
       my $data = $self->{d}->deflate($_);
       $self->{r}->print($data);
       # keep track of its length and crc
       $self->{l} += length($_);
       $self->{crc} = crc32($_,$self->{crc});
    }
}

sub DESTROY {
   my $self = shift;

   # flush the output buffers
   my $data = $self->{d}->flush;
   $self->{r}->print($data);

   # print the CRC and the total length (uncompressed)
   $self->{r}->print(pack("LL",@{$self}{qw/crc l/}));
}

1;
```

By adding a URI translation handler, you can set things up so that a remote user can append a .gz to the end of any URL and the file we be delivered in compressed form. Script II.2.4 shows the translation handler you need. It is called during the initial phases of the request to make any modifications to the URL that it wishes. In this case, it removes the .gz ending from the filename and arranges for Apache:GZip to be called as the content handler. The `lookup_uri()` call is used to exclude anything that has a special handler already defined (such as CGI scripts), and actual gzip files. The module replaces the information in the request object with information about the real file (without the .gz), and arranges for Apache::GZip to be the content handler for this file.

You just need this one directive to activate handling for all URLs at your site:

```
 PerlTransHandler Apache::AutoGZip
```

----------Script II.2.4: Apache::AutoGZip----------

```
 package Apache::AutoGZip;

 use strict vars;
 use Apache::Constants qw/:common/;

 sub handler {
     my $r = shift;

     # don't allow ourselves to be called recursively
     return DECLINED unless $r->is_initial_req;

     # don't do anything for files not ending with .gz
     my $uri = $r->uri;
     return DECLINED unless $uri=~/\.gz$/;
     my $basename = $';

     # don't do anything special if the file actually exists
     return DECLINED if -e $r->lookup_uri($uri)->filename;

     # look up information about the file
     my $subr = $r->lookup_uri($basename);
     $r->uri($basename);
     $r->path_info($subr->path_info);
     $r->filename($subr->filename);

     # fix the handler to point to Apache::GZip;
     my $handler = $subr->handler;
     unless ($handler) {
        $r->handler('perl-script');
        $r->push_handlers('PerlHandler','Apache::GZip');
     } else {
        $r->handler($handler);
     }
     return OK;
 }

 1;
```

Access Control

Access control, as opposed to authentication and authorization, is based on something the user ‘‘is’’ rather than something he ‘‘knows’’. The ‘‘is’’ is usually something about his browser, such as its IP address, hostname, or user agent. Script II.3.1 blocks access to the Web server for certain User Agents (you might use this to block impolite robots).

Apache::BlockAgent reads its blocking information from a ‘‘bad agents’’ file, which contains a series of pattern matches. Most of the complexity of the code comes from watching this file and recompiling it

when it changes. If the file doesn't change, it's is only read once and its patterns compiled in memory, making this module fast.

Here's an example bad agents file:

```
^teleport pro\/1\.28
^nicerspro
^mozilla\/3\.0 \(http engine\)
^netattache
^crescent internet toolpak http ole control v\.1\.0
^go-ahead-got-it
^wget
^devsoft's http component v1\.0
^www\.pl
^digout4uagent
```

A configuration entry to activate this blocker looks like this. In this case we're blocking access to the entire site. You could also block access to a portion of the site, or have different bad agents files associated with different portions of the document tree.

```
<Location />
  PerlAccessHandler Apache::BlockAgent
  PerlSetVar BlockAgentFile /home/www/conf/bad_agents.txt
</Location>
```

----------Script II.3.1: Apache::BlockAgent----------

```
package Apache::BlockAgent;
# block browsers that we don't like

use strict 'vars';
use Apache::Constants ':common';
use IO::File;
my %MATCH_CACHE;
my $DEBUG = 0;

sub handler {
    my $r = shift;

    return DECLINED unless my $patfile = $r->dir_config('BlockAgentFile');
    return FORBIDDEN unless my $agent = $r->header_in('User-Agent');
    return SERVER_ERROR unless my $sub = get_match_sub($r,$patfile);
    return OK if $sub->($agent);
    $r->log_reason("Access forbidden to agent $agent",$r->filename);
    return FORBIDDEN;
}

# This routine creates a pattern matching subroutine from a
# list of pattern matches stored in a file.
sub get_match_sub {
    my ($r,$filename) = @_;
    my $mtime = -M $filename;

    # try to return the sub from cache
    return $MATCH_CACHE{$filename}->{'sub'} if
        $MATCH_CACHE{$filename} &&
            $MATCH_CACHE{$filename}->{'mod'} <= $mtime;
```

```
    # if we get here, then we need to create the sub
    return undef unless my $fh = new IO::File($filename);
    chomp(my @pats = <$fh>); # get the patterns into an array
    my $code = "sub { \$_ = shift;\n";
    foreach (@pats) {
        next if /^#/
        $code .= "return undef if /$_/i;\n";
    }
    $code .= "1; }\n";
    warn $code if $DEBUG;

    # create the sub, cache and return it
    my $sub = eval $code;
    unless ($sub) {
        $r->log_error($r->uri,": ",$@);
        return undef;
    }
    @{$MATCH_CACHE{$filename}}{'sub','mod'}=($sub,$modtime);
    return $MATCH_CACHE{$filename}->{'sub'};
}

1;
```

Authentication and Authorization

Thought you were stick with authentication using text, DBI and DBM files? mod_perl opens the authentication/authorization API wide. The two phases are authentication, in which the user has to prove who he or she is (usually by providing a username and password), and authorization, in which the system decides whether this user has sufficient privileges to view the requested URL. A scheme can incorporate authentication and authorization either together or singly.

Authentication with NIS

If you keep Unix system passwords in /etc/passwd or distribute them by NIS (not NIS+) you can authenticate Web users against the system password database. (It's not a good idea to do this if the system is connected to the Internet because passwords travel in the clear, but it's OK for trusted intranets.)

Script II.4.1 shows how the Apache::AuthSystem module fetches the user's name and password, compares it to the system password, and takes appropriate action. The `getpwnam()` function operates either on local files or on the NIS database, depending on how the server host is configured. WARNING: the module will fail if you use a shadow password system, since the Web server doesn't have root privileges.

In order to activate this system, put a configuration directive like this one in access.conf:

```
<Location /protected>
```

```
  AuthName Test
  AuthType Basic
  PerlAuthenHandler Apache::AuthSystem;
  require valid-user
</Location>
```

----------Script II.4.1: Apache::AuthSystem----------

```
package Apache::AuthSystem;
# authenticate users on system password database

use strict;
use Apache::Constants ':common';

sub handler {
   my $r = shift;

   my($res, $sent_pwd) = $r->get_basic_auth_pw;
   return $res if $res != OK;

   my $user = $r->connection->user;
   my $reason = "";

   my($name,$passwd) = getpwnam($user);
   if (!$name) {
       $reason = "user does not have an account on this system";
   } else {
       $reason = "user did not provide correct password"
           unless $passwd eq crypt($sent_pwd,$passwd);
   }

   if($reason) {
       $r->note_basic_auth_failure;
       $r->log_reason($reason,$r->filename);
       return AUTH_REQUIRED;
   }

   return OK;
}

1;
```

Anonymous Authentication

Here's a system that authenticates users the way anonymous FTP does. They have to enter a name like ''Anonymous'' (configurable) and a password that looks like a valid e-mail address. The system rejects the username and password unless they are formatted correctly.

In a real application, you'd probably want to log the password somewhere for posterity. Script II.4.2 shows the code for Apache::AuthAnon. To activate it, create a access.conf section like this one:

```
<Location /protected>
AuthName Anonymous
```

```
 AuthType Basic
 PerlAuthenHandler Apache::AuthAnon
 require valid-user

 PerlSetVar Anonymous anonymous|anybody
 </Location>
```

----------Script II.4.2: Anonymous Authentication----------

```
 package Apache::AuthAnon;

 use strict;
 use Apache::Constants ':common';

 my $email_pat = '\w+\@\w+\.\w+';
 my $anon_id  = "anonymous";

 sub handler {
     my $r = shift;

     my($res, $sent_pwd) = $r->get_basic_auth_pw;
     return $res if $res != OK;

     my $user = lc $r->connection->user;
     my $reason = "";

     my $check_id = $r->dir_config("Anonymous") || $anon_id;

     unless($user =~ /^$check_id$/i) {
         $reason = "user did not enter a valid anonymous username";
     }

     unless($sent_pwd =~ /$email_pat/o) {
         $reason = "user did not enter an email address password";
     }

     if($reason) {
         $r->note_basic_auth_failure;
         $r->log_reason($reason,$r->filename);
         return AUTH_REQUIRED;
     }

     $r->notes(AuthAnonPassword => $sent_pwd);

     return OK;
 }

 1;
```

Gender-Based Authorization

After authenticating, you can authorize. The most familiar type of authorization checks a group database to see if the user belongs to one or more privileged groups. But authorization can be anything you dream up.

Script II.4.3 shows how you can authorize users by their gender (or at least their *apparent* gender, by checking their names with Jon Orwant's Text::GenderFromName module. This must be used in conjunction with an authentication module, such as one of the standard Apache modules or a custom one.

This configuration restricts access to users with feminine names, except for the users ''Webmaster'' and ''Jeff'', who are allowed access.

```
<Location /ladies_only>
  AuthName "Ladies Only"
  AuthType Basic
  AuthUserFile /home/www/conf/users.passwd
  PerlAuthzHandler  Apache::AuthzGender
  require gender F            # allow females
  require user Webmaster Jeff # allow Webmaster or Jeff
</Location>
```

The script uses a custom error response to explain why the user was denied admittance. This is better than the standard ''Authorization Failed'' message.

----------Script II.4.3: Apache::AuthzGender----------

```
package Apache::AuthzGender;

use strict;
use Text::GenderFromName;
use Apache::Constants ":common";

my %G=('M'=>"male",'F'=>"female");

sub handler {
    my $r = shift;

    return DECLINED unless my $requires = $r->requires;
    my $user = lc($r->connection->user);
    substr($user,0,1)=~tr/a-z/A-Z/;
    my $guessed_gender = uc(gender($user)) || 'M';

    my $explanation = <<END;
<HTML><HEAD><TITLE>Unauthorized</TITLE></HEAD><BODY>
<H1>You Are Not Authorized to Access This Page</H1>
Access to this page is limited to:
<OL>
END

    foreach (@$requires) {
        my ($requirement,@rest ) = split(/\s+/,$_->{requirement});
        if (lc $requirement eq 'user') {
            foreach (@rest) { return OK if $user eq $_; }
            $explanation .= "<LI>Users @rest.\n";
        } elsif (lc $requirement eq 'gender') {
            foreach (@rest) { return OK if $guessed_gender eq uc $_; }
            $explanation .= "<LI>People of the @G{@rest} persuasion.\n";
        } elsif (lc $requirement eq 'valid-user') {
            return OK;
        }
    }
```

```
    $explanation .= "</OL></BODY></HTML>";

    $r->custom_response(AUTH_REQUIRED,$explanation);
    $r->note_basic_auth_failure;
    $r->log_reason("user $user: not authorized",$r->filename);
    return AUTH_REQUIRED;
}

1;
```

Proxy Services

mod_perl gives you access to Apache's ability to act as a Web proxy. You can intervene at any step in the proxy transaction to modify the outgoing request (for example, stripping off headers in order to create an anonymizing proxy) or to modify the returned page.

A Banner Ad Blocker

Script II.5.1 shows the code for a banner-ad blocker written by Doug MacEachern. It intercepts all proxy requests, substituting its own content handler for the default. The content handler uses the LWP library to fetch the requested document. If the retrieved document is an image, and its URL matches the pattern (ads?|advertisement|banner), then the content of the image is replaced with a dynamically-generated GIF that reads ''Blocked Ad''. The generated image is exactly the same size as the original, preserving the page layout. Notice how the outgoing headers from the Apache request object are copied to the LWP request, and how the incoming LWP response headers are copied back to Apache. This makes the transaction nearly transparent to Apache and to the remote server.

In addition to LWP you'll need GD.pm and Image::Size to run this module. To activate it, add the following line to the configuration file:

```
PerlTransHandler Apache::AdBlocker
```

Then configure your browser to use the server to proxy all its HTTP requests. Works like a charm! With a little more work, and some help from the ImageMagick module, you could adapt this module to quiet-down animated GIFs by stripping them of all but the very first frame.

----------Script II.5.1: Apache::AdBlocker----------

```
package Apache::AdBlocker;

use strict;
use vars qw(@ISA $VERSION);
use Apache::Constants qw(:common);
use GD ();
use Image::Size qw(imgsize);
```

```
use LWP::UserAgent ();

@ISA = qw(LWP::UserAgent);
$VERSION = '1.00';

my $UA = __PACKAGE__->new;
$UA->agent(join "/", __PACKAGE__, $VERSION);

my $Ad = join "|", qw{ads? advertisement banner};

sub handler {
    my($r) = @_;
    return DECLINED unless $r->proxyreq;
    $r->handler("perl-script"); #ok, let's do it
    $r->push_handlers(PerlHandler => \&proxy_handler);
    return OK;
}

sub proxy_handler {
    my($r) = @_;

    my $request = HTTP::Request->new($r->method, $r->uri);

    $r->headers_in->do(sub {
       $request->header(@_);
    });

    # copy POST data, if any
    if($r->method eq 'POST') {
       my $len = $r->header_in('Content-length');
       my $buf;
       $r->read($buf, $len);
       $request->content($buf);
       $request->content_type($r->content_type);
    }

    my $response = $UA->request($request);
    $r->content_type($response->header('Content-type'));

    #feed response back into our request_rec*
    $r->status($response->code);
    $r->status_line(join " ", $response->code, $response->message);
    $response->scan(sub {
       $r->header_out(@_);
    });

    if ($r->header_only) {
       $r->send_http_header();
       return OK;
    }

    my $content = \$response->content;
    if($r->content_type =~ /^image/ and $r->uri =~ /\b($Ad)\b/i) {
       block_ad($content);
       $r->content_type("image/gif");
    }

    $r->content_type('text/html') unless $$content;
    $r->send_http_header;
    $r->print($$content || $response->error_as_HTML);
```

```
    return OK;
}

sub block_ad {
    my $data = shift;
    my($x, $y) = imgsize($data);

    my $im = GD::Image->new($x,$y);

    my $white = $im->colorAllocate(255,255,255);
    my $black = $im->colorAllocate(0,0,0);
    my $red = $im->colorAllocate(255,0,0);

    $im->transparent($white);
    $im->string(GD::gdLargeFont(),5,5,"Blocked Ad",$red);
    $im->rectangle(0,0,$x-1,$y-1,$black);

    $$data = $im->gif;
}

1;
```

Another way of doing this module would be to scan all proxied HTML files for <IMG> tags containing one of the verboten URLs, then replacing the SRC attribute with a transparent GIF of our own. However, unless the <IMG> tag contained WIDTH and HEIGHT attributes, we wouldn't be able to return a GIF of the correct size -- unless we were to go hunting for the GIF with LWP, in which case we might as well do it this way.

Customized Logging

After Apache handles a transaction, it passes all the information about the transaction to the log handler. The default log handler writes out lines to the log file. With mod_perl, you can install your own log handler to do customized logging.

Send E-Mail When a Particular Page Gets Hit

Script II.6.1 installs a log handler which watches over a page or set of pages. When someone fetches a watched page, the log handler sends off an e-mail to notify someone (probably the owner of the page) that the page has been read.

To activate the module, just attach a PerlLogHandler to the <Location> or <File> you wish to watch. For example:

```
<Location /~lstein>
   PerlLogHandler Apache::LogMail
   PerlSetVar mailto lstein@cshl.org
</Location>
```

The ‘‘mailto’’ directive specifies the name of the `recipient(s)` to notify.

----------Script II.6.1: Apache::LogMail----------

```
package Apache::LogMail;
use Apache::Constants ':common';

sub handler {
    my $r = shift;
    my $mailto = $r->dir_config('mailto');
    return DECLINED unless $mailto
    my $request = $r->the_request;
    my $uri = $r->uri;
    my $agent = $r->header_in("User-agent");
    my $bytes = $r->bytes_sent;
    my $remote = $r->get_remote_host;
    my $status = $r->status_line;
    my $date = localtime;
    unless (open (MAIL,"|/usr/lib/sendmail -oi -t")) {
       $r->log_error("Couldn't open mail: $!");
       return DECLINED;
    }
    print MAIL <<END;
To: $mailto
From: Mod Perl <webmaster>
Subject: Somebody looked at $uri

At $date, a user at $remote looked at
$uri using the $agent browser.

The request was $request,
which resulted returned a code of $status.

$bytes bytes were transferred.
END
    close MAIL;
    return OK;
}
1;
```

Writing Log Information Into a Relational Database

Coming full circle, Script II.6.2 shows a module that writes log information into a DBI database. The idea is similar to Script I.1.9, but there's now no need to open a pipe to an external process. It's also a little more efficient, because the log data fields can be recovered directly from the Apache request object, rather than parsed out of a line of text. Another improvement is that we can set up the Apache configuration files so that only accesses to certain directories are logged in this way.

To activate, add something like this to your configuration file: PerlLogHandler Apache::LogDBI

Or, to restrict special logging to accesses of files in below the URL ‘‘/lincoln_logs’’ add this:

```
<Location /lincoln_logs>
  PerlLogHandler Apache::LogDBI
</Location>
```

----------Script II.6.2: Apache::LogDBI----------

```
package Apache::LogDBI;
use Apache::Constants ':common';

use strict 'vars';
use vars qw($DB $STH);
use DBI;
use POSIX 'strftime';

use constant DSN       => 'dbi:mysql:www';
use constant DB_TABLE  => 'access_log';
use constant DB_USER   => 'nobody';
use constant DB_PASSWD => '';

$DB = DBI->connect(DSN,DB_USER,DB_PASSWD) || die DBI->errstr;
$STH = $DB->prepare("INSERT INTO ${\DB_TABLE} VALUES(?,?,?,?,?,?,?,?,?)")
     || die $DB->errstr;

sub handler {
    my $r = shift;
    my $date    = strftime('%Y-%m-%d %H:%M:%S',localtime);
    my $host    = $r->get_remote_host;
    my $method  = $r->method;
    my $url     = $r->uri;
    my $user    = $r->connection->user;
    my $referer = $r->header_in('Referer');
    my $browser = $r->header_in("User-agent");
    my $status  = $r->status;
    my $bytes   = $r->bytes_sent;
    $STH->execute($date,$host,$method,$url,$user,
                  $browser,$referer,$status,$bytes);
    return OK;
}

1;
```

Conclusion

You'll find more tricks in my books, articles and Web site. Here's where you can find them:

"How to Set Up and Maintain a Web Site"

General introduction to Web site care and feeding, with an emphasis on Apache. Addison-Wesley 1997.

Companion Web site at http://www.genome.wi.mit.edu/WWW/

"Web Security, a Step-by-Step Reference Guide"

How to keep your Web site free from thieves, vandals, hooligans and other yahoos. Addison-Wesley 1998.

Companion Web site at http://www.w3.org/Security/Faq/

"The Official Guide to Programming with CGI.pm"

Everything I know about CGI.pm (and some things I don't!). John Wiley & Sons, 1998.

Companion Web site at http://www.wiley.com/compbooks/stein/

"Writing Apache Modules in Perl and C"

Co-authored with Doug MacEachern. O'Reilly & Associates.

Companion Web site at http://www.modperl.com/

WebTechniques Columns

I write a monthly column for WebTechniques magazine. You can find back-issues and reprints at http://www.web-techniques.com/

The Perl Journal Columns

I write a quarterly column for TPJ. Source code listings are available at http://www.tpj.com/

Good reference works written by others:

Advanced Perl Programming

By Sriram Srinivasan, published by O'Reilly & Associates. Very good introduction to DBI, TK and the XS language.

Essential Perl Programming

By Joseph Hall with Randal Schwartz. An elegant guide to the dos and don'ts of Perl Programming. Published by Addison Wesley.

Perl Cookbook

By Tom Christiansen and Nathan Torkington. All the tricks and idioms in one convenient tome. Also published by O'Reilly & Associates. Get this book!