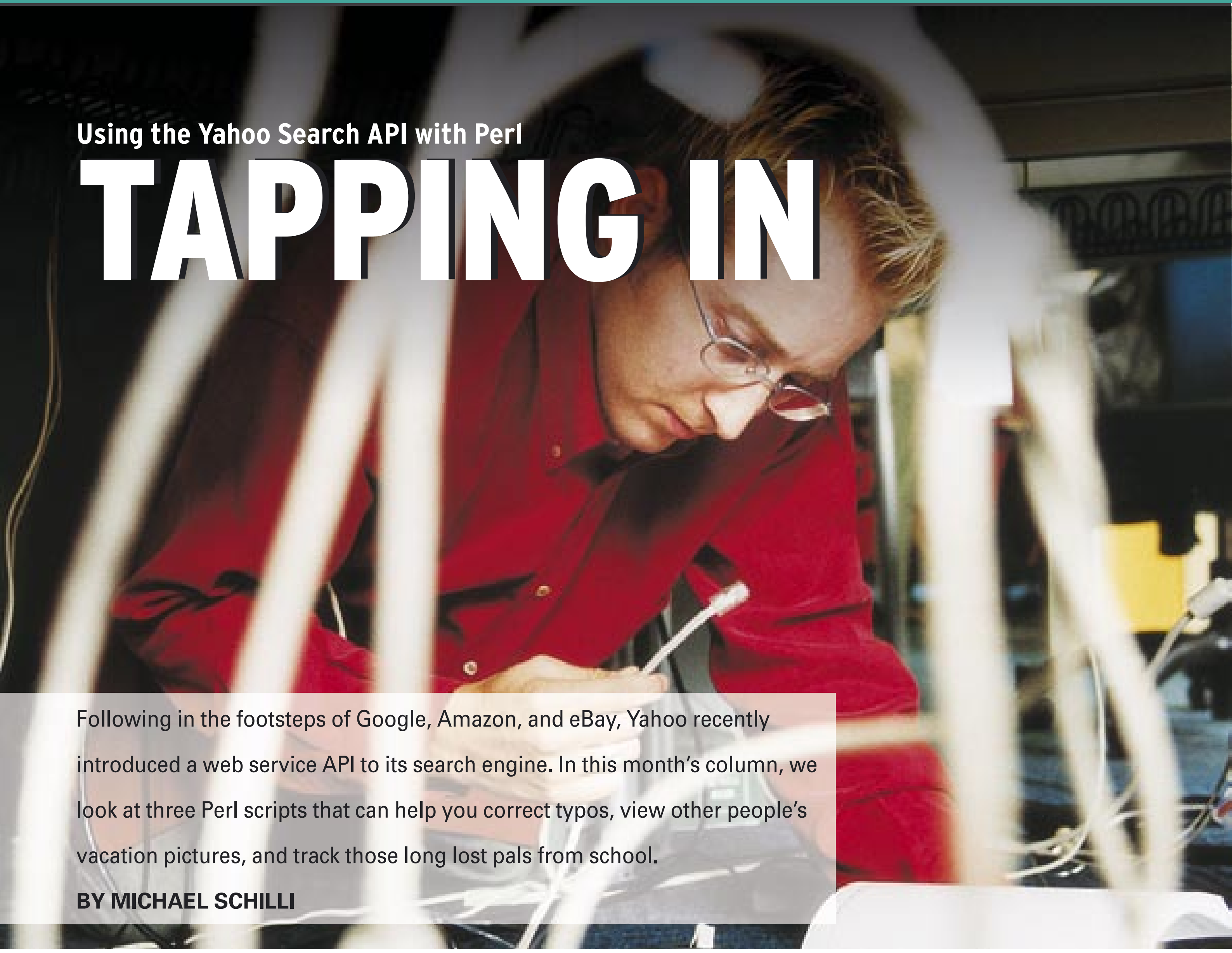


Using the Yahoo Search API with Perl

TAPPING IN

Following in the footsteps of Google, Amazon, and eBay, Yahoo recently introduced a web service API to its search engine. In this month's column, we look at three Perl scripts that can help you correct typos, view other people's vacation pictures, and track those long lost pals from school.

BY MICHAEL SCHILLI

It is quite common for an Internet service to provide a Web API. Developers use the Web API to integrate the Internet service with their own applications. Yahoo just released a REST (Representational State Transfer) interface to its search infrastructure. REST basically lets an application send a URL and get XML back. Of course, there is also a new Yahoo::Search Perl module to match; the module is available from CPAN and was developed by regex guru Jeffrey Friedl.

Yahoo::Search makes it easy to search for documents, images, videos, and many other things. HTTP access and XML extraction are hidden behind simple calls to methods. If you intend to write an application, you need to register to get your own application ID. You will be allowed to issue 5,000 calls per day to the services mentioned in this article. To register at [2], you will need your own Yahoo ID, which you can get in exchange for a valid email address. You'll also have to agree to Yahoo's Privacy Statement; make sure to read it carefully.

Did you Mean...?

If you are not sure how a word is spelled, you can always grab a copy of a dictionary, although it may be missing some of the latest buzzwords, pop culture terms, or proper nouns. The Internet is a big help in this case. Most search engines offer you a "Did you mean?" function; that is, they suggest a sane alternative if you have typed something that they can't follow.

For example, the *typo* script (see Listing 1) uses the Web API's *Term()* method to call the Yahoo web spellchecker with a word or phrase handed to it:

```
$ typo foo foughters
Corrected: foo fighters
```

The spellchecker does not just correct words found in a dictionary. You can spell check nearly any term people write about on the Web. Even well-known politicians:

```
$ typo tonie blayre
Corrected:tony blair
```

Listing 1 shows the implementation. The *use* command in line 11 loads the Yahoo::Search module, which you installed previously using the CPAN shell, and passes the application ID that you obtained from [2] to it. You need to modify the details of the script to reflect your credentials at this point.

The *Terms()* method's *Spell* parameter expects a word or phrase, sends it to the Yahoo service, investigates the XML returned in the response, and extracts the answer if an answer exists. Line 14 stores the response in the *$suggestion* variable. The if-else construct that follows either outputs the response or *No corrections*, if the search engine failed to come up with the goods. Although the service is making progress with internationalization, you may not get the results you expect for searches in foreign languages, especially with search terms that include accented letters.

Remembrance of Things Past

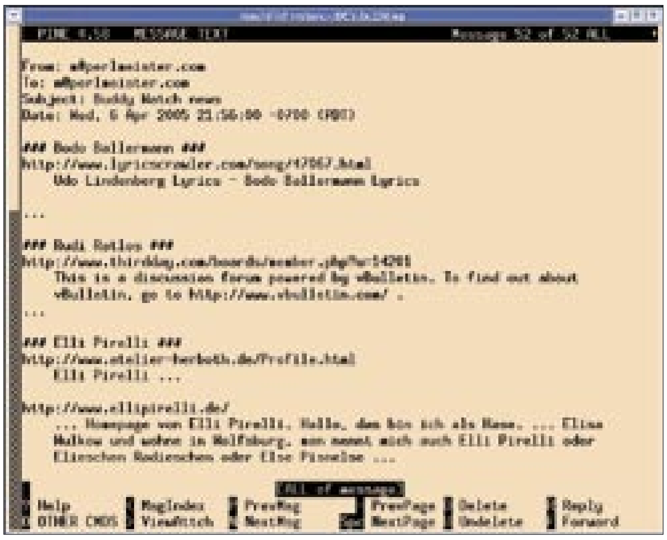
Search engine crawlers relentlessly wander through the known Internet looking for new sources of information. This means that search results can and will change. If you search for an old school pal's name once a day, you will immediately notice the changes if he or she starts setting up a new homepage or becomes famous overnight. Of course, nobody has the time to perform that kind of search manually, and it might be difficult to keep track of the results.

Search Service

To make things easier for you, the *buddy* script (Listing 2) is launched once a day by a cronjob. *buddy* retrieves the first 25 results for a list of names stored in the *~/.buddy* configuration file. *buddy* sends any previously unknown URLs, along with an extract from the website content, to a preconfigured email address. This keeps you up to date by letting you know if one of the buddies on your buddy list turns up as, say, a Nobel prize nominee.

Listing 1: typo

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

my $term = "@ARGV";

die
  "usage: $0 word/phrase ..."
  unless length $term;

use Yahoo::Search AppId =>
  "YOUR_APP_ID";

my ($suggestion) =
  Yahoo::Search->Terms(
    Spell => $term );

if (defined $suggestion) {
  print "Corrected: ",
        "$suggestion\n";
} else {
  print "No suggestions\n";
}
```



```
PINE 4.58   MESSAGE TEXT                          Message 52 of 52 ALL

From: m@perlmeister.com
To: m@perlmeister.com
Subject: Buddy Watch news
Date: Wed, 6 Apr 2005 21:56:00 -0700 (PDT)

### Bodo Ballermann ###
http://www.lyricscrawler.com/song/47067.html
    Udo Lindenberg Lyrics - Bodo Ballermann Lyrics

...

### Rudi Ratlos ###
http://www.thirdday.com/boards/member.php?u=14281
    This is a discussion forum powered by vBulletin. To find out about
    vBulletin, go to http://www.vbulletin.com/ .
...

### Elli Pirelli ###
http://www.atelier-herboth.de/Profile.html
    Elli Pirelli ...

http://www.ellipirelli.de/
    ... Homepage von Elli Pirelli. Hallo, das bin ich als Hase. ... Elina
    Walkow und wohne in Wolfsburg. man nennt mich auch Elli Pirelli oder
    Elinchen Radieschen oder Else Pinselse ...

                               [ALL of message]
? Help       < MsgIndex   P PrevMsg      - PrevPage  D Delete    R Reply
O OTHER CMDS > ViewAttch  N NextMsg    Spc NextPage  U Undelete  F Forward
```

Figure 1: Old school pals have popped up on the Web. The script mails you a summary of the hit list.

Buddy keeps any URLs it finds in a cache for a period of one month. Every time it finds a cached name again, the caching period gets extended. Names that have been missing for a month will be rediscovered as if they were new, thus simulating a bad memory and cheering you up occasionally.

Line 12 of the script expects the email address to which it should send the updates. If you launch *buddy* at the command line and specify the *-v* (for verbose) option, line 24 initializes the Log4perl framework with a log level of *$DEBUG*, making the log more informative. By default, you will only see log messages with *WARN* or higher priority.

Line 26 declares the *mailadd* function (which is defined later on) to tell Perl that this is a new function, allowing any calls that precede the definition to do so without needing parentheses. *mailadd*, which starts at line 93, holds the accumulated mailtext in an *our* variable called *$maildata*. The *mailsend* function in line 100 shares the same variable. Calls to *mailadd* simply append text to *$maildata*. The call to *close()* in line 118 then sends the completed message to the address you supplied (Figure 1) using the Mail::Send module from CPAN.

The *plough* function, which was exported by the Sysadm::Install module, expects a callback function and a filename in line 30. The function parses the buddy configuration file *~/.buddy*, calls the callback function after each line it has read, and passes the content of the line to the $_ variable. Line 31 discards lines commented with *#*, and the *chomp* command bites off the newline. Line 33 pushes any buddies the process has found to the end of the continually expanding *@buddies* array.

Line 53 then uses the *Results()* method to contact the Yahoo service, wrapping your buddy's name from the configuration file in double quotes, and then passing it as a quoted string *qq{"$buddy"}* with the *Doc* parameter, as this is a search for a Web document.

The list of resulting objects returned in the response uses the *Url()* and *Summary()* methods to output the URL and an extract for any hits. The file cache (line 37) is set up behind the scenes by Cache::FileCache in */tmp/FileCache*. The cache keeps any entries for 30 days, as specified by default in the *default_expires_in* parameter.

As the web service strictly requires UTF-8, the names in *~/.buddy* have to be UTF-8 encoded. This is irrelevant if the names are in plain English, but accented characters are a different story. If you have a recent Linux distribution, your editor will store accented characters in UTF-8 by default. If you still use Latin 1, you can run a tool such as *toutf8* with a command line such as *toutf8 buddy.latin1 > ~/.buddy* to quickly convert the file:

```
# toutf8
use Text::Iconv;
my $conv = ↩
Text::Iconv->new("Latin1",↩
"UTF-8");
print $conv->convert
(join '', <>);
```

You only need to modify the email address in line 12 of the script to match your own needs. A cron entry such as *0 5 * * * $HOME/bin/buddy* will call the script every morning, query the search engine, update the cache, and send you an email message with all changes since the last search. It works best with uncommon names; tracking “John Doe” will generate too much noise.

Picture This

The new service also supports searching for images. The search engine will

Listing 2: buddy

```
001 #!/usr/bin/perl -w
002 ##################################
003 # buddy - Track search result
004 #         changes over time.
005 # 2005, m@perlmeister.com
006 ##################################
007 use strict;
008
009 my $BUDDY_FILE =
010   "$ENV{HOME}/.buddy";
011 my $EMAIL_TO =
012   'email@somewhere.com';
013
014 use Sysadm::Install qw(:all);
015 use Yahoo::Search;
016 use Text::Wrap;
017 use Cache::FileCache;
018 use Log::Log4perl qw(:easy);
019 use Getopt::Std;
020 use Mail::Send;
021
022 getopts( "v", \my %o );
023
024 Log::Log4perl->easy_init(
025   $o{v} ? $DEBUG : $WARN );
026 sub mailadd;
027
028 my @buddies = ();
029
030 plough sub {
031   return if /^\s*#/;
032   chomp;
033   push @buddies, $_;
034 }, $BUDDY_FILE;
035
036 my $cache =
037   Cache::FileCache->new({
038     namespace => "Buddy",
039     default_expires_in =>
040       3600 * 24 * 30,
041   });
042
043 my $search =
044   Yahoo::Search->new(
045     AppId => "YOUR_APP_ID",
046     Count => 25,
047   );
048
049 for my $buddy (@buddies) {
050   DEBUG "Search request ",
051         "for '$buddy'";
052   my @results =
053     $search->Results(
054     Doc => qq{"$buddy"} );
055
056   my $buddy_listed = 0;
057
058   DEBUG scalar @results,
059     "  results";
060
061   for my $result (@results) {
062     if($cache->get(
063          $result->Url()
064        )) {
065       DEBUG "Found cached: ",
066         $result->Url();
067
068       # Refresh if found
069       $cache->set(
070         $result->Url(), 1);
071       next;
072     }
073
074     mailadd
075       "\n\n### $buddy ###"
076       unless $buddy_listed++;
077
078     mailadd $result->Url();
079
080     $cache->set(
081       $result->Url(), 1);
082
083     mailadd fill( "    ",
084       "    ",
085       $result->Summary() ),
086       "";
087   }
088 }
089
090 mailsend();
091
092 ##################################
093 sub mailadd {
094 ##################################
095   our $maildata;
096   $maildata .= "$_\n" for @_;
097 }
098
099 ##################################
100 sub mailsend {
101 ##################################
102   our $maildata;
103
104   return
105     unless defined $maildata;
106
107   DEBUG "Sending email: ",
108         "$maildata";
109
110   my $msg =
111     Mail::Send->new();
112
113   $msg->to($EMAIL_TO);
114   $msg->subject(
115     "Buddy Watch News");
116   my $fh = $msg->open;
117   print $fh $maildata;
118   close $fh;
119 }
```

retrieve a number of image URLs that match your search key, passing them to the *slideshow* script (see Listing 3, on Page 74), which will display them at 5 second intervals in your browser window.

The script first displays a simple search form (Figure 3). When a user enters a search key into the form (for example, *San Francisco*) and clicks the *Search* button, the CGI parameter *q* is set, and line 68 calls the *Results()* method of the Yahoo::Search package. The *Image* parameter passes the search key, *Count* limits the results to 50 hits, and a setting of *AllowAdult* with a value of *0* at least tries to prevent adult content from suddenly popping up on your screen.

As the text in the image captions is again UTF-8, the *header()* method in line 26 tells your browser that the dynamically generated web page is UTF-8 encoded.

Cache Memory

The *slideshow* CGI script stores the results of your search. In other words, *slideshow* stores the image URLs and summary texts stored as an array of arrays in a persistent file cache. This technique ensures that the slide show projector does not need to query the search engine each time the projector moves on to the next image.

The Cache::FileCache module stores key-value pairs where the values are simple scalars, but there is no support for nested structures. The *Storable* module can help you work around this lack of support for nested structures. The *Storable* module's *freeze()* function can serialize a data structure before placing it in the cache. If the serialized data needs to be retrieved from the cache, the de-serializer *thaw()* is called to convert the data back to the original nested Perl data structure.

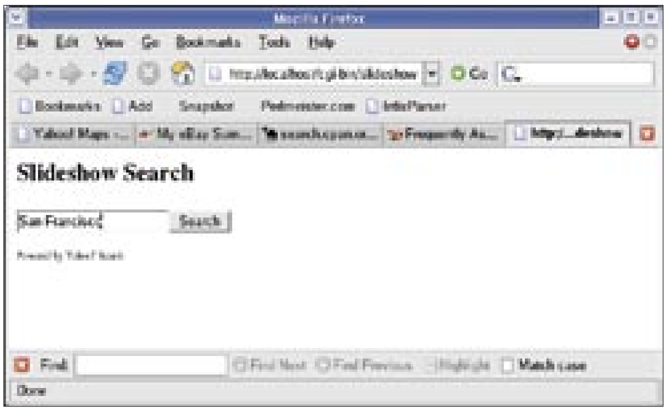


Figure 3: The image search in Listing 3 gives you an exciting slide show, mainly containing private vacation snapshots.

Details

To prevent the CGI script from inadvertently using unchecked and insecure incoming data for system calls (i.e., to avoid the possibility of tearing a gaping security hole in the application), the *-T* – for taint mode – option is enabled right at the start of the script in the Shebang line following the call to the Perl interpreter.

The first *if* block (line 28) is enabled if the script is called both with the query string, and with the serial number of the current image. In this case, the cache will hold a sequence of image URLs with matching captions from a previous call. Line 31 thaws the array of arrays, and the modulo operator in line 35 ensures that the incremented serial number will always point to a position within the array and not to somewhere out of bounds.

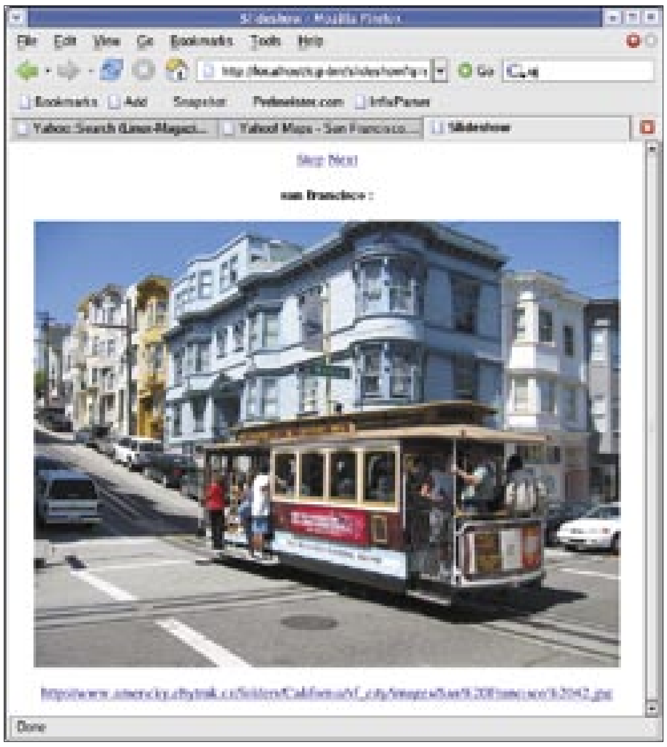


Figure 2: A query for "San Francisco" in the input mask returned a slide show with a view of the city.

The *refresh()* function called in line 36 using the parameter 5 is defined in line 107. This *refresh()* function returns HTML sequences, which pass meta tags to the browser to tell the browser to load the next image after waiting for the number of seconds specified with the interval parameter passed to it.

The optional second parameter for the *refresh()* function specifies whether the next URL to be loaded by the script will display the next image (*next_url* simply increments the numeric parameter *s*), or whether the script should go back to the starting page with the original URL. Line 88 uses this second parameter if no search results are found.

Installation

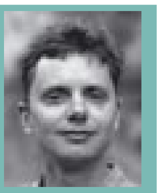
To install the script, simply put it in the *cgi-bin* directory of your web server, type in a search query, sit back, and relax while watching other people's vacation pictures! ■

INFO

[1] Listings for this article: *http://www.linux-magazine.com/Magazine/Downloads/56/Perl*

[2] Yahoo Developer API Homepage: *http://developer.yahoo.com*

THE AUTHOR



Michael Schilli works as a Software Developer at Yahoo!, Sunnyvale, California. He wrote "Perl Power" for Addison-Wesley and can be contacted at *schilli@perlmeister.com*. His homepage is at *http://perlmeister.com*.

Listing 3: slideshow

```
#!/usr/bin/perl -wT
################################
# slideshow - Yahoo image
#     search as slideshow CGI
# 2005, m@perlmeister.com
################################
use strict;

use CGI qw(:all);
use Yahoo::Search AppId =>
  "YOUR_APP_ID";
use Cache::FileCache;
use Storable qw(freeze thaw);

my $cache =
  Cache::FileCache->new({
    namespace => 'slideshow',
    default_expires_in =>
      3600,
    auto_purge_on_set => 1,
  });

my $data;

print header(
  -charset => "utf-8");

if(param('q')
   and defined param('s')) {

  $data = thaw $cache->get(
    param('q'));

  my $seq = param('s');
  $seq %= scalar @$data;
  print refresh(5);
  print center(
    a(
      { href => url() },
      "Stop"
    ),
    a(
      { href => next_url() },
      "Next"
    ),
    p(),
    b( param('q') ),
    ":",
    i( $data->[$seq]->[1] ),
    p(),
    img(
      { src =>
          $data->[$seq]->[0]
      }
    ),
    p(),
    a(
      { href =>
          $data->[$seq]->[0]
      },
      $data->[$seq]->[0]
    ),
  );

} elsif(param('q')) {

  my @results =
    Yahoo::Search->Results(
    Image      => param('q'),
    Count      => 50,
    AllowAdult => 0,
    );

  if (@results) {
    for (@results) {
      push @$data,
        [
        $_->Url(),
        $_->Summary()
        ];
    }
    print refresh(0);
    $cache->set(
      param('q'),
      freeze($data)
    );
  } else {
    print refresh( 0, 1 );
  }
} else {
  print h2(
    "Slideshow Search"),
    start_form(),
    textfield(
      -name => 'q' ),
    submit(
      -value => "Search" ),
    end_form(),
    font(
    { size => 1 },
      "Powered by " .
      "Yahoo Search"
    );
}

################################
sub refresh {
################################
  my ($sleep, $reset) = @_;

  return start_html(
    -title => "Slideshow",
    -head  => meta({
        -http_equiv =>
          "Refresh",
        -content =>
          "$sleep, URL=" . (
          $reset ?
              url() :
              next_url()}));
}

################################
sub next_url {
################################
  my $s = param('s');
  $s ||= 0;

  return
    sprintf "%s?q=%s&s=%d",
    url(), param('q'),
    $s + 1;
}
```