

Create dynamic Web sites with Perl

PART
THREE



A customer feedback form is a must for any site. Charlie Stross shows you how it's done

Most people pick up Perl with one thing in mind: automating their Web sites. Of course it can do far more, but since it's the commonest Common Gateway Interface (CGI) programming language, that's what most people think of when they hear the word Perl.

If CGI is new to you, check out some of the larger sites around, like **www.yahoo.com**. When you see a URL ending in gibberish like `/query.cgi?cb=0&q=`, the chances are that you're getting served pages via a CGI program. For details on the mechanics of CGI, see O'Reilly's OpenBook, *CGI Programming and the World Wide Web* on this month's SuperDisc.

Back in 1993-94 most Web documents were static, unchanging HTML files. Where it was necessary to provide access to a database or other program, Perl proved just the tool for the job. These days there are other tools for doing the job, including ASP, PHP and JSP (covered in this month's *Java Workshop*), but you're still far more likely to find an ISP supporting Perl than any of the others.

Your first CGI script

Open up the example program on the SuperDisc. It does two things. First, it creates an HTML page containing a form. Clicking the Submit button passes the same script parameters (the bits after the '?' in the URL). If the script is called with the parameter 'live' it tries to send us an e-mail.

We could have written our script as an HTML page and a single-purpose CGI script, but this way the form and the script are indivisible. We can't accidentally break it by loosing one half or misdirecting the HTML form to the wrong URL.

Lines 7 and 8 import two Perl modules: CGI and Mail::Internet. We qualify the 'use CGI' command with some parameters – these specify the methods we want to explicitly import from the CGI namespace in addition to the normal CGI stuff.

CGI is a big module. If you want to learn about it you really need to read the documentation that comes with it (in POD format) – it runs to about 40 pages. Type



SEE OUR **SUPERDISC**

PATH: \handson\files\perl



OOP!

Using objects in Perl

If you've used CGI, you've already used Perl's objects, bless their cotton socks

An object in Perl is simply a piece of data (like a scalar, a hash, or something else) that knows what package it belongs to. We tell an object what it is by using the `bless()` command;

```
bless $foo, MyClass;
```

tells `$foo` that it is of type `MyClass` (and, by extension, subroutines from the package `MyClass` apply to it).

There are a couple of ways of invoking methods – subroutines specific to the blessed data type. We can explicitly call the method, qualifying it with the name of the package it belongs to, and passing the object as a parameter:

```
$foo = CGI::new();
```

Or we can use the indirection operator:

```
$foo = CGI->new();
```

If we've defined a module search path, we can even live dangerously and try it like this:

```
@INC = (qw(CGI));
$foo = new();
```

The module search path, `@INC`, sets up our inheritance tree. It's simply an array, each element of which is a class name (the classes being packages which are first imported into the program or module we're defining by calling `use()`). If we are defining a class that is a sub-class of CGI, we put CGI in `@INC`. Objects belonging to our new class then inherit all of CGI's methods. For example:

```
package CGI::Custom;
use CGI;
@INC = (qw(CGI));
```

First, we start a new package called `CGI::Custom`. This package uses CGI (meaning, it imports everything in the package CGI at compile time). Then we add CGI to `@INC`. `CGI::Custom` objects now inherit all of CGI's methods, unless we override one by defining a new method with the same name.

Methods are just subroutines that know what sort of data they operate on. A common idiom you'll see in modules is:

```
sub my_method {
    my $obj = shift;
    # do something to $obj, and
    return
}
```

The first parameter to `my_method()` is the object it is being applied to. We can call it like this:

```
my_method($obj);
```

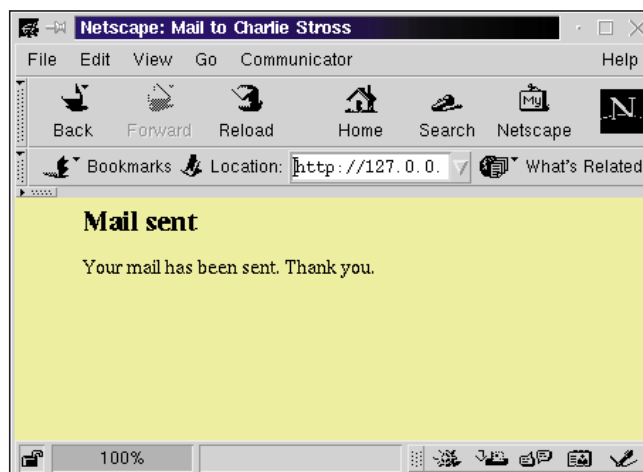
or:

```
$obj->my_method();
```

Finally, if we get confused about the object we're working on, we can use the `ref()` command. `ref()` examines a reference and tells us what it is; either a scalar, or a reference to an array or hash, or the class into which it has been blessed.



That glorious moment – your first CGI script works! It's a good idea to test it on a Web browser on your own computer before uploading to a live site.



The confirmation page could be spruced up with a happy graphic and a link back to the site's index, but we'll leave that to you.

'perldoc CGI' in Linux or see ActivePerl's Module documentation in Windows.

At line 18, we create a CGI object. This implicitly checks our environment to see if we're running as a CGI script, imports bits of data from the Web server we're running under, and creates an object we can do useful things with. We do just one thing, at line 20: we look for an HTML form parameter called 'live'. If it's present, we call a subroutine called `process_form()`, to process the form input. If it isn't present, we call `print_form()`, to produce the HTML form for this script.

Tag, you're it

`print_form()` is the simpler of the two routines, although it doesn't look it at first sight. We pass it a single parameter (`$q`, the CGI object), then call various CGI methods via `$q` that generate

HTML tags and print them. `print_form()` could equally well be written as a simple `print` statement to output HTML. But doing it programmatically is clearer, and if we were feeling daring we could do nifty Perl things like generate extra fields on the fly, or expand macros, and so on.

One point to note is that CGI's `startform()` method implicitly creates a form that points back to the current CGI script. Broken links to scripts are a thing of the past.

`process_form()` does a bit more leg-work. First, we extract the named parameters from the form. Then, in lines 78 to 83 we create and send an e-mail message! This looks deceptively simple; actually, Mail::Internet conceals a vast mass of complexity from us. Just create a new Mail::Internet object (using the `new()` method), use the `subject()` and `to()` methods to say what

it's about and who it's addressed to, then call `open()` on it. This returns a file handle, print the contents of the message into this file handle and close it. When it closes, the mail is automatically sent.

Finally, we print some HTML – the traditional way, just to demonstrate that you can mix and match HTML and CGI's programmatic HTML generation in the same script, or even the same subroutine.

Now you've created the script, it's time to test it out. You'll find details on the *SuperDisc* on getting it running on your own server, or uploading it to a Web site. **PCP**

NEXT MONTH

Creating dynamic pages from an on-line database.

CGI EXPLAINED

The Common Gateway Interface

Learn more about your environment in CGI

The Common Gateway Interface (or CGI for short) dates back to 1992, when the Web was in its infancy. People were just beginning to realise that as well as serving up HTML documents Web servers would be called on to run programs that produced output in HTML. CGI supplied a mechanism to get data into such a program and send its output back to the user.

When Apache (or another CGI-compatible server) receives an HTTP request for an executable object, it sticks a chunk of information about the request in the environment – an area of shared memory accessible to child programs. Then it runs the CGI program, which pulls the information it needs out of the environment.

Anything the CGI program prints on its standard output filehandle is then returned direct to the requesting client – usually your visitor's Web browser. Normally CGI programs need to start their output by stating what type of data they're producing. It's common to see hand-rolled CGI programs start by printing:

```
Content-type: text/html
```

followed by a blank line – the minimal HTTP 0.9 header required for an HTML document. If it's missing, the Web browser simply won't know how to handle the output stream.

Look at the `CurrEnv.cgi` script on the *SuperDisc*. It's a minimal – but useful – CGI program. The script doesn't do anything with its inputs – but it lets you see the environment variables the CGI script can see. To do this, it calls the subroutine `Dumper` (imported from the module `Data::Dumper`), to dump out a neatly-formatted version of the hash `%ENV`. `%ENV` is the current Perl processes view of its environment.

Experiment by writing an HTML form, with the SUBMIT button pointing at this script. You'll be able to see the HTTP request showing up in the environment in the form of a variable called `QUERY_STRING` (if you use the HTTP GET method in your form).

Having to parse this mass of data by hand is a nuisance, which is why we have the excellent `CGI.pm` module, written by Lincoln Stein (and a host of contributors). `CGI.pm` takes a lot of the legwork out of writing CGI scripts by letting you

treat an HTML form as an object. You can query the object for the current values in a named field, or you can tell it to print out various HTML form elements in a programmatic manner, setting various options along the way. For example:

```
print CGI->textfield("fred")
```

prints out an HTML text entry field within a form, named 'fred'. And if you're processing data submitted from this form:

```
my $cgi = new CGI;
my $freds_value =
$cgi->param("fred");
```

The `new()` method in CGI parses the CGI environment variables and extracts data; thereafter, we can query the value of the CGI parameter 'fred' via the `param()` method.

As always, there's a mass of documentation on CGI included with your standard Perl installation. See 'perldoc CGI' under Linux or ActiveState's Module Docs under Windows.