AN ADDRESS BOOK FOR THE WORLD WIDE WEB

WebBook

SIMSON L. GARFINKEL

Plus your own Perl-based Web server!

WebBook is a free-format, multiuser address book designed to be used with the World Wide Web. WebBook is written in Perl, and includes pattern-matching technology to automatically recognize email addresses and URLs in entries and format them properly. It can also be quite fast. Figure 1 is the interface you'll meet when you access WebBook, while Figure 2 shows a sample run.

To run WebBook, you'll need Perl, a Web browser that can handle forms, and a Web server on which to run Perl scripts. If you don't have a Web server, take heart: At the end of this article, I'll show you how to write your own.

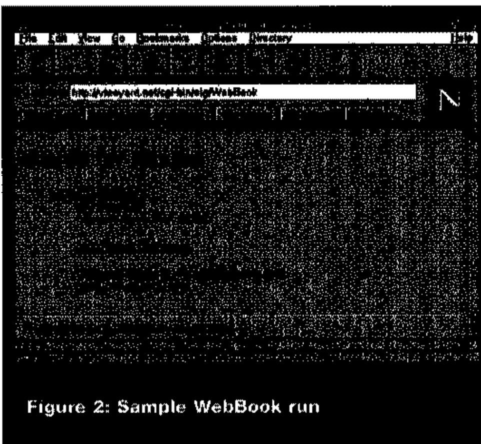

Figure 2: Sample WebBook run

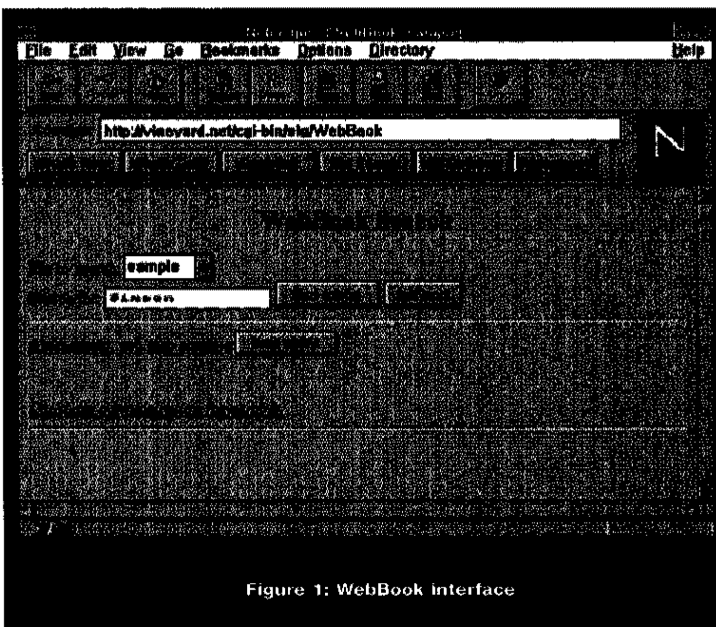

Figure 1: WebBook interface

WebBook Backgrounder

I've been interested in computerized address books since the fall of 1984, when I lost the leather-bound address book I'd had for a number of years. Suddenly, I was without the names, addresses, and phone numbers of all my friends, classmates, and family. And because I'd been keeping all sorts of additional information in my address book, I was also without directions to people's houses, bank-account numbers, and recipes for my favorite desserts.

Starting over, I realized that the logical solution was to computerize my address book to prevent a future catastrophe. But every computerized address book I looked at had a similar failing: They were all designed to store names, addresses and phone numbers, but little more. Although that might work for a sales executive who merely needs to organize potential leads, it doesn't work for me, and it doesn't work for most people.

WebBook

My solution was a free-format database kept in a word processor. To separate each entry, I used a row of equal signs. My text editor's string-search capability located the names and addresses of friends.

Over the years, my address book grew. I wrote a series of emacs macros for quickly navigating through the sea of names. I wrote a program for finding names and printing mailing envelopes. I finally wrote a full blown application using NextStep. Called "SBook" (and now sold by Sarrus Software), many believe it's one of the best address books ever written—except that it only runs on NextStep. The problem with this is that I've moved to the world of Macs, PCs, and Solaris workstations. And these days, I am a user of the World Wide Web, not a particular operating system.

WebBook is the most recent incarnation of my address book. By writing it in Perl to run on a Web server, I avoided SBook's biggest problem—platform dependence. Instead, WebBook uses a Web browser as a generic graphical output device; HTML is the presentation language, and HTTP is the API. This puts a whole new spin on the concept of "client/server," and I'm expecting a lot of programs to adopt this strategy in the future. (I've already written a set of CGI scripts for performing UNIX system administration using many of these same ideas.) WebBook also demonstrates many of the techniques that I've learned over the past year for creating manageable Perl/CGI scripts.

The full WebBook program is over 1000 lines of Perl and contains many features, such as the ability to handle multiple address books and password-protection of entries. Rather than print the full WebBook here, I've created a scaled-down demonstration version; see Listing One (listings begin on page 77). The full WebBook is available at my Web server: *http://vineyard.net/simson/WebBook/faq.html*.

Build for Debugging

The most difficult thing about creating CGI programs is debugging them. I got a head start by using the Perl cgi-lib library developed by Steven E. Brenner at Cambridge University. Brenner's library handles communication between the Web server and your CGI script, automatically unescapes all variables and stuffs them into a Perl associative array, and gives your Perl scripts a uniform way of handling both GET and POST requests. You can get more information about cgi-lib at *http://www.bio.cam.ac.uk/web/form.html*. I've included a subset of cgi-lib in the version of WebBook presented here.

WebBook is written as a single Perl script. Arguments are provided by the standard CGI interface. A special argument, or "action," is used to specify which action the user wishes to perform. If no action is specified, WebBook displays a welcome message, prints some information about the database on the server, and displays a search form. If WebBook is asked to perform an action that it doesn't understand, an error message is returned to the Web browser.

It's easy to get lost within a single Perl script: What gets executed and what doesn't? When Perl starts up, the entire file is parsed; then execution starts at the first line and moves down. Many Perl programmers freely mix

```
# Define globals
$database = "WebBook.data";
$max_entry = 450;


# Get Started:
if (!$standalone) {
    print "Content-type:
            text/html\n\n";
    &main;
    exit(0);
}
```

Example 1: Placing global variable assignments at the top of a Perl program.

```
<form method=post action="WebBook">
Search for: <input type=text size="16" name="name">
<input type=submit name="search-type" value="find name">
<input type=submit name="search-type" value="full-text">
<input type=hidden name="action" value="search">
</form>
<hr>
<form method=post action="WebBook">
Alternatively, you may create a
<input type=submit value="new entry">
<input type=hidden value="new" name="action">
</form>
```

Example 2: HTML code that creates three buttons that can invoke three choices.

global-variable assignments, executable code, and function definitions. But global-variable assignments are performed at run time, rather than compile time, which means that a global assignment deep within the file will not be executed until control passes to that point.

To get around this, I've placed my global-variable assignments at the top of my Perl program; when this block of code is executed, it runs the *main* subroutine called; see Example 1. This strategy makes it easy to incorporate WebBook into other Perl scripts—in particular, those that implement Web servers.

Deciding What to Do

Once the CGI script starts up, it needs to decide what to do. Most CGI scripts just do one or two things—incrementing and displaying a counter, displaying a form and performing a requested action, or whatever. But CGI scripts such as WebBook that can perform many different actions require a means to tell the script which action to perform. In my programming, each link on the form and each Submit button on each HTML form sets the *action* variable to a different value. I frequently display HTML pages that contain multiple forms and submit buttons, all of which invoke the same CGI script but with different values for the *action* parameter.

When an *action* can be executed more than one different way (for example, a search for a name, or a full-text search), I use additional variables, or subactions. For example, WebBook has two basic actions: "search," which causes the program to look for matching entries in the database, and "new," which causes the program to create a new entry. The search action further has two subactions: "find name," which does a search on the name field, and "full-text," which searches the full text of the entries.

Example 2 is HTML code that creates three buttons that can invoke these three choices. Clicking on any of these buttons invokes the WebBook CGI script and runs the subroutine *&main*. Inside the *&main* subroutine, the Perl code in Example 3 receives the action and dispatches it to the appropriate Perl subroutine.

```
sub get_command {
    if(!&ReadParse(*input)) {
        $action = $input{'action'};
        $name   = $input{'name'};
    }

    #
    # Decode the action
    #
    if($action eq "search"){          &do_search($name); return; }
    if($action eq "edit"){            &do_edit($name);   return; }
    if($action eq "save-entry"){      &do_save_entry;    return; }
    if($action eq "delete-entry"){    &do_delete_entry;  return; }

    # Default - display an info message
    &do_info;
}
```

Example 3: Perl code that receives an action and dispatches it to an appropriate Perl subroutine.

```
$escaped_name = $_;
$escaped_name =~ s/ /+/g;

print "<li><big><a
href=\"WebBook?action=edit&name=$escaped_name\">$_</a></big><br>\n";
```

Example 4: Code that displays names.

```
# Now escape the various things in HTML tags

$ent =~ s/&/&#38;/g;
$ent =~ s/</&#60;/g;

# Note: the following four substitutions must be done in order

# 1. Catch the URLs
$ent =~ s#((http|mailto|ftp)://[^ \n\t]*)#<a href="$1">$1</a>#g;

# 2. Catch the email addresses
$ent =~ s#([a-zA-Z0-9_.]+@[a-zA-Z0-9_.]+)#<a href="mailto:$1">$1</a>#gi;

# 3. Now change newlines to <br>'s
$ent =~ s/\n/<br>\n/g;
```

Example 5: Code that implements substitutions using Perl's pattern-matching and substitution capabilities.

Building the Database

WebBook keeps its database in a Perl DBM file. This lets you create an associative array whose contents are automatically stored in a DBM file. (Perl5 includes a more general *tie* function that allows you to bind an associative array with any kind of database back end.) The file is opened with the statement: *dbmopen(%DB, $database,0666);*. An associative array

WebBook

consists of a set of (key, value) pairs. WebBook uses the keys of the associative array to store the name of each person in the database; the person's entry is stored in the value. For example, my WebBook entry might have an element in the associative array with the key of "Simson L. Garfinkel" and the value "PO Box 4188\nVineyard Haven, MA 02568".

Searching and Displaying

The heart of WebBook is searching for names. This is implemented with the function *do_search*, which calls the *display_search_field* function to display the search field, conducts the search with the *find_names* function, then creates the appropriate HTML to display the results.

Names are displayed in an unnumbered list. Each name is displayed as a link; click on the name to edit the entry. Spaces in names must be escaped to "+" characters; other special characters should be escaped as well. As Example 4 illustrates, the code that does this is quite simple.

WebBook automatically escapes the entries' less-than characters (so that they will not be interpreted as HTML tags). It then catches the URLs and displays them as real links, and catches the e-mail addresses and displays them as mailto: URLs. Finally, it turns the newlines into <br> tags.

The code that implements these substitutions use Perl's pattern-matching and substitution capabilities; see Example 5. These particular Perl features are unparalleled in most other computer languages, one reason why Perl is so well suited to building CGI scripts.

Editing

When users click on a name, the WebBook CGI is run with the action variable set to *edit* and the name variable set to be the name of the entry being edited. The *do_edit* subroutine merely displays the entry name as a text field and the entry text inside a text area. The actual editing is done by the user's own browser. When users finish editing, they press the Save button.

Because users might change the name of the entry during the editing, the edit form needs to send the CGI script both the new name (in the *name* field) and the old name (in the *old-name* field). These values are both read by the *do_save_entry* subroutine. *do_save_entry* checks that *$entry* isn't too big. (Perl4 DBM files don't always work if the value is larger than 450 characters; Perl5 overcomes this problem.) The subroutine then deletes the old entry and creates the new one.

No command exists for creating a new entry. That's because the *&do_edit* function is used to create new entries—*do_edit* is simply invoked to edit the entry with the name "". This makes sense if you think of the *action* variable as the selector for a remote-procedure call *systme*, rather than as the name of the particular kind of form being displayed.

Gotcha!

Well, that's enough of WebBook to get you going. But wait—if you type in this script, put it on your server, and try to run it, it probably won't work. Too see why not, run the UNIX tail command (see Example 6) on your Web server's error log.

The problem is that most Web servers run CGI scripts as user "nobody" (UID -2). If your server is configured properly, user "nobody" won't have access to create or modify files stored in the cgi-bin directory. (By the way, when developing CGI programs it's a good idea to keep a window open with the tail -f command running; this lets you immediately catch your mistakes.)

There are a variety of ways around this security problem. One is to make the Perl script SUID to a user specifically created for maintaining the database. Another approach is to specify a *$database* file stored in a directory other than the Perl script. Both of these are good ideas. Implementing them is left as an exercise. For testing,

```
vineyard.net% tail -f /usr/local/etc/httpd/logs/error_log
No write permission to ndbm file at /usr/local/etc/httpd/cgi-bin/wb/WebBook line 165.
```

Example 6: Running the UNIX tail command on your Web server's error log.

```
#!/bin/sh
read a b c
cat $b
```

Example 7: Implementing a simple (but insecure) Web server in just three lines of code.

```
GET /index.html HTTP/1.0
User-Agent: Mozilla/1.12
                (Macintosh; I; 68K)
Accept: */*
Accept: image/gif
Accept: image/x-xbitmap
Accept: image/jpeg
```

Example 8: When a Web browser connects to a Web server, it sends through code similar to this.

```
HTTP/1.0 200 Document follows
Date: Sat, 30 Dec 1995 ? GMT
Server: NCSA/1.4.2
Content-type: text/html
Last-modified: Wed, 27 Dec 1995 ? GMT
Content-length: 6193
```

Example 9: Sample information a Web server might send.

you can set the *$database* file to be "/tmp/database." Correct your permissions problem and try again.

Build Your Own Web Server

After using WebBook for a few weeks, my biggest problem was speed. Sure, Perl is fast enough at searching through the database. However, every time the CGI script was run, the Web server had to start up a copy of Perl, and Perl had to read and compile the WebBook program before WebBook could start respond to requests. On my NeXTstation, the total overhead was nearly two seconds.

I tried playing around with Perl's undump facility, which lets you "compile" Perl by dumping a core file and processing it with the UNIX undump program. Unfortunately, undump is not a standard part of UNIX, and I couldn't get it to work on NextStep. So I took another approach—I wrote my own Web server, and built it into the WebBook program.

It turns out that writing a Web server is rather trivial. A simple Web server just listens on a port, reads a request, and performs the requested action. Usually, the request is a GET for the file named by the second argument. Thus, as Example 7 shows, a simple (but very insecure) Web server can be written in just three lines.

This works because of the simplicity of the HTTP protocol. When a Web browser (in this case, Netscape Navigator 1.12 on a Macintosh) connects to a Web server, it sends through something like Example 8, where */index.html* is the name of the URL being requested—the slash that follows the host name and everything after that. In the case of the Vineyard.NET web site, the Web server might send back Example 9, followed by the document itself.

I then wrote a wrapper for WebBook's *&main* subroutine. This Perl script, "cgi-server" (see Listing Two), takes as arguments the number of a port on which to listen and the name of a script to run. The script binds to the port and awaits HTTP connections. Each time it receives a connection, it forks off a child process that receives the HTML command (and any POST information), sets up the appropriate environment variables, and runs *&main*. You may wish to add more error checking or clean up the Perl code.

Is cgi-server worth it? Absolutely. It causes the slow parts of WebBook—starting up Perl, and then reading and compiling the WebBook program—to happen before the request is received from the Web server. And these parts happen only once, rather than every time a new connection is received. After that, everything is simply kept in RAM.

Conclusion

The World Wide Web is making it possible to write a new generation of client/server programs. These programs use a Web browser as a generic client and HTML as a generic presentation layer for specifying the creation of buttons, text fields, clickable links, and other widgets.

One comment I've heard about WebBook is that all my painstaking work on the server can be done better on the client using Java. Well, that might be true. But not every Web browser supports Java, while even Lynx users can access a WebBook database. Furthermore, writing Java is certainly going to be more complicated than writing a few lines of Perl on a server. On the other hand, I can certainly see many ways in which a Java WebBook client could complement the WebBook server—performing searches on the end user's machine, perhaps, or even giving the user a better text editor than the one built into their Web browser. My bet is that Java programs won't replace server-side CGI scripts, but will actually work with them to minimize the amount of information sent across the network and allow greater flexibility of data presentation.

In the meantime, if you think that WebBook is neat, take a look at the FAQ. You'll find links for downloading your own copy of the full program and adding yourself to a mailing list for WebBook users. Enjoy! ▼

Simson is author of PGP: Pretty Good Privacy *(O'Reilly & Associates, 1994) and other books. He can be contacted at simsong@vineyard.net.*