CHAPTER 18

TclHttpd Web Server

This chapter describes TclHttpd, a Web server built entirely in Tcl. The Web server can be used as a standalone server, or it can be embedded into applications to Web-enable them. TclHttpd provides a Tcl+HTML template facility that is useful for maintaining site-wide look and feel, and an application-direct URL that invokes a Tcl procedure in an application.

This chapter is from *Practical Programming in Tcl and Tk*, 3rd Ed.

http://www.beedub.com/book/

TclHttpd started out as about 175 lines of Tcl that could serve up HTML pages and images. The Tcl `socket` and I/O commands make this easy. Of course, there are lots of features in Web servers like Apache or Netscape that were not present in the first prototype. Steve Uhler took my prototype, refined the HTTP handling, and aimed to keep the basic server under 250 lines. I went the other direction, setting up a modular architecture, adding in features found in other Web servers, and adding some interesting ways to connect TclHttpd to Tcl applications.

Today TclHttpd is used both as a general-purpose Web server, and as a framework for building server applications. It implements `www.scriptics.com`, including the Tcl Resource Center and Scriptics' electronic commerce facilities. It is also built into several commercial applications such as license servers and mail spam filters. Instructions for setting up the TclHttpd on your platform are given toward the end of the chapter, on page 266. It works on Unix, Windows, and Macintosh. Using TclHttpd, you can have your own Web server up and running quickly.

This chapter provides an overview of the server and several examples of how you can use it. The chapter is not an exhaustive reference to every feature. Instead, it concentrates on a very useful subset of server features that I use the most. There are references to Tcl files in the server's implementation, which are all found in the `lib` directory of the distribution. You may find it helpful to read the code to learn more about the implementation. You can find the source on the CD-ROM.

Integrating TclHttpd with your Application

The bulk of this chapter describes the various ways you can extend the server and integrate it into your application. TclHttpd is interesting because, as a Tcl script, it is easy to add to your application. Suddenly your application has an interface that is accessible to Web browsers in your company's intranet or the global Internet. The Web server provides several ways you can connect it to your application:

- *Static pages* — As a "normal" Web server, you can serve static documents that describe your application.
- *Domain handlers* — You can arrange for all URL requests in a section of your Web site to be handled by your application. This is a very general interface where you interpret what the URL means and what sort of pages to return to each request. For example, `http://www.scriptics.com/resource` is implemented this way. The URL past `/resource` selects an index in a simple database, and the server returns a page describing the pages under that index.
- *Application-Direct URLs* — This is a domain handler that maps URLs onto Tcl procedures. The form query data that is part of the HTTP GET or POST request is automatically mapped onto the parameters of the application-direct procedure. The procedure simply computes the page as its return value. This is an elegant and efficient alternative to the CGI interface. For example, in TclHttpd the URLs under `/status` report various statistics about the Web server's operation.
- *Document handlers* — You can define a Tcl procedure that handles all files of a particular type. For example, the server has a handler for CGI scripts, HTML files, image maps, and HTML+Tcl template files.
- *HTML+Tcl Templates* — These are Web pages that mix Tcl and HTML markup. The server replaces the Tcl using the `subst` command and returns the result. The server can cache the result in a regular HTML file to avoid the overhead of template processing on future requests. Templates are a great way to maintain common look and feel to a family of Web pages, as well as to implement more advanced dynamic HTML features like self-checking forms.

TclHttpd Architecture

Figure 18–1 shows the basic components of the server. At the core is the `Httpd` module (`httpd.tcl`), which implements the server side of the `HTTP` protocol. The "d" in Httpd stands for *daemon*, which is the name given to system servers on `UNIX`. This module manages network requests, dispatches them to the `Url` module, and provides routines used to return the results to requests.

The `Url` module (`url.tcl`) divides the Web site into *domains*, which are subtrees of the URL hierarchy provided by the server. The idea is that different domains may have completely different implementations. For example, the Docu-

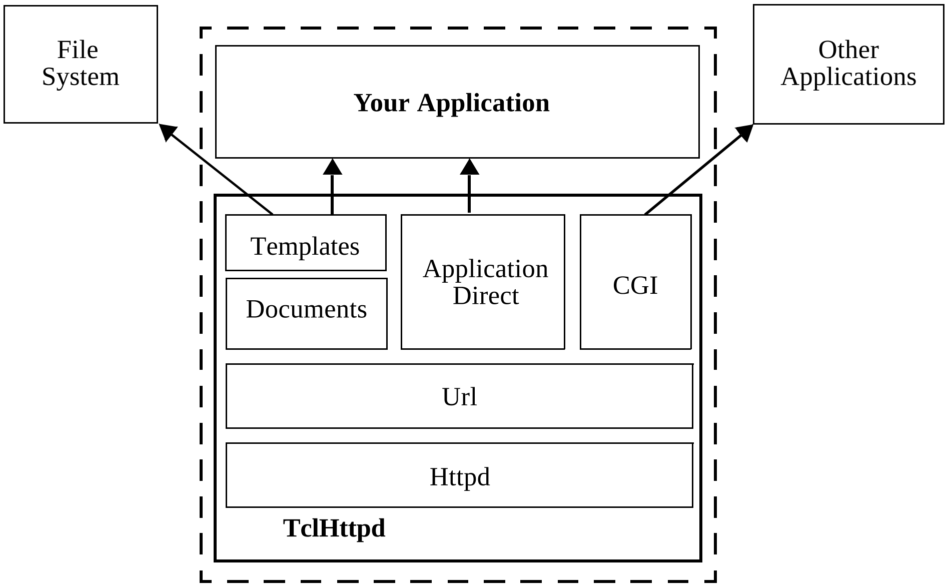

Fig. 18–1 The dotted box represents one application that embeds TclHttpd. Document templates and Application Direct URLs provide direct connections from an HTTP request to your application.

ment domain (`doc.tcl`) maps its URLs into files and directories on your hard disk, while the Application-Direct domain (`direct.tcl`) maps URLs into Tcl procedure calls within your application. The CGI domain (cgi.tcl) maps URLs onto other programs that compute Web pages.

Domain Handlers

You can implement new kinds of domains that provide your own interpretation of a URL. This is the most flexible interface available to extend the Web server. You provide a callback that is invoked to handle every request in a domain, or subtree, of the URL hierarchy. The callback interprets the URL, computes the page content, and returns the data using routines from the `Httpd` module.

Example 18–1 defines a simple domain that always returns the same page to every request. The domain is registered with the `Url_PrefixInstall` command. The arguments to `Url_PrefixInstall` are the URL prefix and a callback that is called to handle all URLs that match that prefix. In the example, all URLs that have the prefix `/simple` are dispatched to the `SimpleDomain` procedure.

Example 18–1 A simple URL domain.

```
Url_PrefixInstall /simple [list SimpleDomain /simple]

proc SimpleDomain {prefix sock suffix} {
   upvar #0 Httpd$sock data

   # Generate page header

   set html "<title>A simple page</title>\n"
   append html "<h1>$prefix$suffix</h1>\n"
   append html "<h1>Date and Time</h1>\n"
   append html [clock format [clock seconds]]
   # Display query data

   if {[info exist data(query)]} {
      append html "<h1>Query Data</h1>\n"
      append html "<table>\n"
      foreach {name value} [Url_DecodeQuery $data(query)] {
         append html "<tr><td>$name</td>\n"
         append html "<td>$value</td></tr>\n"
      }
      append html "</table>\n"
   }
   Httpd_ReturnData $sock text/html $html
}
```

The `SimpleDomain` handler illustrates several properties of domain handlers. The `sock` and `suffix` arguments to `SimpleDomain` are appended by `Url_Dispatch` when it invokes the domain handler. The `suffix` parameter is the part of the URL after the prefix. The `prefix` is passed in as part of the callback definition so the domain handler can recreate the complete URL. For example, if the server receives a request for the URL `/simple/page`, then the prefix is `/simple`, the suffix is `/request`.

Connection State and Query Data

The `sock` parameter is a handle on the socket connection to the remote client. This variable is also used to name a state variable that the `Httpd` module maintains about the connection. The name of the state array is `Httpd$sock`, and `SimpleDomain` uses `upvar` to get a more convenient name for this array (i.e., `data`):

```
upvar #0 Httpd$sock data
```

An important element of the state array is the query data, `data(query)`. This is the information that comes from HTML forms. The query data arrives in an encoded format, and the `Url_DecodeQuery` procedure is used to decode the data into a list of names and values. `Url_DecodeQuery` is similar to `Cgi_List` from Example 11–5 on page 154 and is a standard function provided by `url.tcl`.

Returning Results

Finally, once the page has been computed, the `Httpd_ReturnData` procedure is used to return the page to the client. This takes care of the HTTP protocol as well as returning the data. There are three related procedures, `Httpd_ReturnFile`, `Httpd_Error`, and `Httpd_Redirect`. These are summarized in Table 18–1 on page 259.

Application Direct URLs

The Application Direct domain implementation provides the simplest way to extend the Web server. It hides the details associated with query data, decoding URL paths, and returning results. All you do is define Tcl procedures that correspond to URLs. Their arguments are automatically matched up to the query data as shown in Example 13–3 on page 179. The Tcl procedures compute a string that is the result data, which is usually HTML. That’s all there is to it.

The `Direct_Url` procedure defines a URL prefix and a corresponding Tcl command prefix. Any URL that begins with the URL prefix will be handled by a corresponding Tcl procedure that starts with the Tcl command prefix. This is shown in Example 18–2:

Example 18–2 Application Direct URLs.

```
Direct_Url /demo Demo

proc Demo {} {
   return "<html><head><title>Demo page</title></head>\n\
      <body><h1>Demo page</h1>\n\
      <a href=/demo/time>What time is it?</a>\n\
      <form action=/demo/echo>\n\
      Data: <input type=text name=data>\n\
      <br>\n\
      <input type=submit name=echo value=’Echo Data’>\n\
      </form>\n\
      </body></html>"
}
proc Demo/time {{format "%H:%M:%S"}} {
   return [clock format [clock seconds] -format $format]
}
proc Demo/echo {args} {
   # Compute a page that echoes the query data

   set html "<head><title>Echo</title></head>\n"
   append html "<body><table>\n"
   foreach {name value} $args {
      append html "<tr><td>$name</td><td>$value</td></tr>\n"
   }
   append html "</tr></table>\n"
   return $html
}
```

Example 18–2 defines /demo as an Application Direct URL domain that is implemented by procedures that begin with Demo. There are just three URLs defined:

```
/demo
/demo/time
/demo/echo
```

The /demo page displays a hypertext link to the /demo/time page and a simple form that will be handled by the /demo/echo page. This page is static, and so there is just one return command in the procedure body. Each line of the string ends with:

```
\n\
```

This is just a formatting trick to let me indent each line in the procedure, but not have the line indented in the resulting string. Actually, the \-newline will be replaced by one space, so each line will be indented one space. You can leave those off and the page will display the same in the browser, but when you view the page source you'll see the indenting. Or you could not indent the lines in the string, but then your code looks somewhat odd.

The /demo/time procedure just returns the result of clock format. It doesn't even bother adding <html>, <head>, or <body> tags, which you can get away with in today's browsers. A simple result like this is also useful if you are using programs to fetch information via HTTP requests.

Using Query Data

The /demo/time procedure is defined with an optional format argument. If a format value is present in the query data, then it overrides the default value given in the procedure definition.

The /demo/echo procedure creates a table that shows its query data. Its args parameter gets filled in with a name-value list of all query data. You can have named parameters, named parameters with default values, and the args parameter in your application-direct URL procedures. The server automatically matches up incoming form values with the procedure declaration. For example, suppose you have an application direct procedure declared like this:

```
proc Demo/param { a b {c cdef} args} { body }
```

You could create an HTML form that had elements named a, b, and c, and specified /demo/param for the ACTION parameter of the FORM tag. Or you could type the following into your browser to embed the query data right into the URL:

```
/demo/param?a=5&b=7&c=red&d=%7ewelch&e=two+words
```

In this case, when your procedure is called, a is 5, b is 7, c is red, and the args parameter becomes a list of:

```
d ~welch e {two words}
```

The %7e and the + are special codes for nonalphanumeric characters in the query data. Normally, this encoding is taken care of automatically by the Web browser when it gets data from a form and passes it to the Web server. However, if you type query data directly or format URLs with complex query data in them, then you need to think about the encoding. Use the Url_Encode procedure to encode URLs that you put into Web pages.

If parameters are missing from the query data, they either get the default values from the procedure definition or the empty string. Consider this example:

```
/demo/param?b=5
```

In this case a is "", b is 5, c is cdef, and args is an empty list.

Returning Other Content Types

The default content type for application direct URLs is text/html. You can specify other content types by using a global variable with the same name as your procedure. (Yes, this is a crude way to craft an interface.) Example 18–3 shows part of the faces.tcl file that implements an interface to a database of picons — personal icons — that is organized by user and domain names. The idea is that the database contains images corresponding to your e-mail correspondents. The Faces_ByEmail procedure, which is not shown, looks up an appropriate image file. The application direct procedure is Faces/byemail, and it sets the global variable Faces/byemail to the correct value based on the filename extension. This value is used for the Content-Type header in the result part of the HTTP protocol.

Example 18–3 Alternate types for Application Direct URLs.

```
Direct_Url /faces Faces
proc Faces/byemail {email} {
   global Faces/byemail
   set filename [Faces_ByEmail $email]
   set Faces/byemail [Mtype $filename]
   set in [open $filename]
   fconfigure $in -translation binary
   set X [read $in]
   close $in
   return $X
}
```

Document Types

The Document domain (doc.tcl) maps URLs onto files and directories. It provides more ways to extend the server by registering different document type handlers. This occurs in a two-step process. First, the type of a file is determined by its suffix. The mime.types file contains a map from suffixes to MIME types such as text/html or image/gif. This map is controlled by the Mtype module in mtype.tcl. Second, the server checks for a Tcl procedure with the appropriate name:

```
Doc_mimetype
```

The matching procedure, if any, is called to handle the URL request. The procedure should use routines in the Httpd module to return data for the request. If there is no matching Doc_*mimetype* procedure, then the default document handler uses Httpd_ReturnFile and specifies the Content Type based on the file extension:

```
Httpd_ReturnFile $sock [Mtype $path] $path
```

You can make up new types to support your application. Example 18–4 shows the pieces needed to create a handler for a fictitious document type application/myjunk that is invoked to handle files with the .junk suffix. You need to edit the mime.types file and add a document handler procedure to the server:

Example 18–4 A sample document type handler.

```
# Add this line to mime.types
application/myjunk     .junk

# Define the document handler procedure
#  path is the name of the file on disk
#  suffix is part of the URL after the domain prefix
#  sock is the handle on the client connection

proc Doc_application/myjunk {path suffix sock} {
   upvar #0 Httpd$sock data
   # data(url) is more useful than the suffix parameter.

   # Use the contents of file $path to compute a page
   set contents [somefunc $path]

   # Determine your content type
   set type text/html

   # Return the page
   Httpd_ReturnData $sock $type $data
}
```

As another example, the HTML+Tcl templates use the .tml suffix that is mapped to the application/x-tcl-template type. The TclHttpd distribution also includes support for files with a .snmp extension that implements a template-based Web interface to the Scotty SNMP Tcl extension.

HTML + Tcl Templates

The template system uses HTML pages that embed Tcl commands and Tcl variable references. The server replaces these using the subst command and returns the results. The server comes with a general template system, but using subst is so easy you could create your own template system. The general template framework has these components:

- Each .html file has a corresponding .tml template file. This feature is enabled with the Doc_CheckTemplates command in the server's configuration file. Normally, the server returns the .html file unless the corresponding .tml file has been modified more recently. In this case, the server processes the template, caches the result in the .html file, and returns the result.
- A dynamic template (e.g., a form handler) must be processed each time it is requested. If you put the Doc_Dynamic command into your page, it turns off the caching of the result in the .html page. The server responds to a request for a .html page by processing the .tml page. Or you can just reference the .tml file directly. If you do this, the server always processes the template.
- The server creates a page global Tcl variable that has context about the page being processed. Table 18–7 lists the elements of the page array.
- The server initializes the env global Tcl variable with similar information, but in the standard way for CGI scripts. Table 18–8 lists the elements of the env array that are set by Cgi_SetEnv in cgi.tcl.
- The server supports per-directory ".tml" files that contain Tcl source code. These files are designed to contain procedure definitions and variable settings that are shared among pages. The name of the file is simply ".tml", with nothing before the period. This is a standard way to hide files in UNIX, but it can be confusing to talk about the per-directory ".tml" files and the *file*.tml templates that correspond to *file*.html pages. The server will source the ".tml" files in all directories leading down to the directory containing the template file. The server compares the modify time of these files against the template file and will process the template if these ".tml" files are newer than the cached .html file. So, by modifying the ".tml" file in the root of your URL hierarchy, you invalidate all the cached .html files.
- The server supports a script library for the procedures called from templates. The Doc_TemplateLibrary procedure registers this directory. The server adds the directory to its auto_path, which assumes you have a tclIndex or pkgIndex.tcl file in the directory so that the procedures are loaded when needed.

Where to put your Tcl Code

There are three places you can put the code of your application: directly in your template pages, in the per-directory ".tml" files, or in the library directory.

The advantage of putting procedure definitions in the library is that they are defined one time but executed many times. This works well with the Tcl bytecode compiler. The disadvantage is that if you modify procedures in these files, you have to explicitly source them into the server for these changes to take effect. The /debug/source URL described on page 264 is handy for this chore.

The advantage of putting code into the per-directory ".tml" files is that changes are picked up immediately with no effort on your part. The server automatically checks if these files are modified and sources them each time it processes your templates. However, that code is run only one time, so the byte-code compiler just adds overhead.

I try to put as little code as possible in my *file*.tml template files. It is awkward to put lots of code there, and you cannot share procedures and variable definitions easily with other pages. Instead, my goal is to have just procedure calls in the template files, and put the procedure definitions elsewhere. I also avoid putting if and foreach commands directly into the page.

Templates for Site Structure

The next few examples show a simple template system used to maintain a common look at feel across the pages of a site. Example 18–5 shows a simple one-level site definition that is kept in the root .tml file. This structure lists the title and URL of each page in the site:

Example 18–5 A one-level site structure.

```
set site(pages) {
   Home                /index.html
   "Ordering Computers"/ordering.html
   "New Machine Setup" /setup.html
   "Adding a New User" /newuser.html
   "Network Addresses" /network.html
}
```

Each page includes two commands, SitePage and SiteFooter, that generate HTML for the navigational part of the page. Between these commands is regular HTML for the page content. Example 18–6 shows a sample template file:

Example 18–6 A HTML + Tcl template file.

```
[SitePage "New Machine Setup"]
This page describes the steps to take when setting up a new
computer in our environment. See
<a href=/ordering.html>Ordering Computers</a>
for instructions on ordering machines.
<ol>
<li>Unpack and setup the machine.
<li>Use the Network control panel to set the IP address
and hostname.
<!-- Several steps omitted -->
<li>Reboot for the last time.
</ol>
[SiteFooter]
```

The SitePage procedure takes the page title as an argument. It generates HTML to implement a standard navigational structure. Example 18–7 has a simple implementation of SitePage:

Example 18–7 `SitePage` template procedure.

```
proc SitePage {title} {
   global site
   set html "<html><head><title>$title</title></head>\n"
   append html "<body bgcolor=white text=black>\n"
   append html "<h1>$title</h1>\n"
   set sep ""
   foreach {label url} $site(pages) {
      append html $sep
      if {[string compare $label $title] == 0} {
         append html "$label"
      } else {
         append html "<a href='$url'>$label</a>"
      }
      set sep " | "
   }
   return $html
}
```

The `foreach` loop that computes the simple menu of links turns out to be useful in many places. Example 18–8 splits out the loop and uses it in the `SitePage` and `SiteFooter` procedures. This version of the templates creates a left column for the navigation and a right column for the page content:

Example 18–8 `SiteMenu` and `SiteFooter` template procedures.

```
proc SitePage {title} {
   global site
   set html "<html><head><title>$title</title></head>\n\
      <body bgcolor=$site(bg) text=$site(fg)>\n\
      <!-- Two Column Layout -->\n\
      <table cellpadding=0>\n\
      <tr><td>\n\
      <!-- Left Column -->\n\
      <img src='$site(mainlogo)'>\n\
      <font size=+1>\n\
      [SiteMenu <br> $site(pages)]\n\
      </font>\n\
      </td><td>\n\
      <!-- Right Column -->\n\
      <h1>$title</h1>\n\
      <p>\n"
   return $html
}
proc SiteFooter {} {
   global site
   set html "<p><hr>\n\
      <font size=-1>[SiteMenu | $site(pages)]</font>\n\
      </td></tr></table>\n"
   return $html
}
proc SiteMenu {sep list} {
    global page
    set s ""
    set html ""
    foreach {label url} $list {
       if {[string compare $page(url) $url] == 0} {
          append html $s$label
       } else {
          append html "$s<a href='$url'>$label</a>"
       }
       set s $sep
    }
    return $html
}
```

Of course, a real site will have more elaborate graphics and probably a two-level, three-level, or more complex tree structure that describes its structure.You can also define a family of templates so that each page doesn't have to fit the same mold. Once you start using templates, it is fairly easy to change both the template implementation and to move pages around among different sections of your Web site.

There are many other applications for "macros" that make repetitive HTML coding chores easy. Take, for example, the link to `/ordering.html` in Example 18–6. The proper label for this is already defined in `$site(pages)`, so we could introduce a `SiteLink` procedure that uses this:

Example 18–9 The `SiteLink` procedure.

```
proc SiteLink {label} {
    global site
    array set map $site(pages)
    if {[info exist map($label)]} {
       return "<a href='$map($label)'>$label</a>"
    } else {
       return $label
    }
}
```

If your pages embed calls to `SiteLink`, then you can change the URL associated with the page name by changing the value of `site(pages)`. If this is stored in the top-level "`.tml`" file, the templates will automatically track the changes.

Form Handlers

HTML forms and form-handling programs go together. The form is presented to the user on the client machine. The form handler runs on the server after the user fills out the form and presses the submit button. The form presents input widgets like radiobuttons, checkbuttons, selection lists, and text entry fields. Each of these widgets is assigned a name, and each widget gets a value based on the user's input. The form handler is a program that looks at the names and values from the form and computes the next page for the user to read.

CGI is a standard way to hook external programs to Web servers for the purpose of processing form data. CGI has a special encoding for values so that they can be transported safely. The encoded data is either read from standard input or taken from the command line. The CGI program decodes the data, processes it, and writes a new HTML page on its standard output. Chapter 3 describes writing CGI scripts in Tcl.

TclHttpd provides alternatives to CGI that are more efficient because they are built right into the server. This eliminates the overhead that comes from running an external program to compute the page. Another advantage is that the Web server can maintain state between client requests in Tcl variables. If you use CGI, you must use some sort of database or file storage to maintain information between requests.

Application Direct Handlers

The server comes with several built-in form handlers that you can use with little effort. The `/mail/forminfo` URL will package up the query data and mail it to you. You use form fields to set various mail headers, and the rest of the data is packaged up into a Tcl-readable mail message. Example 18–10 shows a form that uses this handler. Other built-in handlers are described starting at page 263.

Example 18–10 Mail form results with `/mail/forminfo`.

```
<form action=/mail/forminfo method=post>
   <input type=hidden name=sendto value=mailreader@my.com>
   <input type=hidden name=subject value="Name and Address">
   <table>
      <tr><td>Name</td><td><input name=name></td></tr>
      <tr><td>Address</td><td><input name=addr1></td></tr>
      <tr><td> </td><td><input name=addr2></td></tr>
      <tr><td>City</td><td><input name=city></td></tr>
      <tr><td>State</td><td><input name=state></td></tr>
      <tr><td>Zip/Postal</td><td><input name=zip></td></tr>
      <tr><td>Country</td><td><input name=country></td></tr>
   </table>
</form>
```

The mail message sent by `/mail/forminfo` is shown in Example 18–11.

Example 18–11 Mail message sent by `/mail/forminfo`.

```
To: mailreader@my.com
Subject: Name and Address

data {
   name   {Joe Visitor}
```

```
    addr1  {Acme Company}
    addr2  {100 Main Street}
    city   {Mountain View}
    state  California
    zip    12345
    country   USA
}
```

It is easy to write a script that strips the headers, defines a `data` procedure, and uses `eval` to process the message body. Whenever you send data via e-mail, if you format it with Tcl list structure, you can process it quite easily. The basic structure of such a mail reader procedure is shown in Example 18–12:

Example 18–12 Processing mail sent by `/mail/forminfo`.

```
# Assume the mail message is on standard input

set X [read stdin]

# Strip off the mail headers, when end with a blank line
if {[regsub {.*?\n\ndata} $X {data} X] != 1} {
    error "Malformed mail message"
}
proc data {fields} {
    foreach {name value} $fields {
        # Do something
    }
}
# Process the message. For added security, you may want
# do this part in a safe interpreter.
eval $X
```

Template Form Handlers

The drawback of using application-direct URL form handlers is that you must modify their Tcl implementation to change the resulting page. Another approach is to use templates for the result page that embed a command that handles the form data. The `Mail_FormInfo` procedure, for example, mails form data. It takes no arguments. Instead, it looks in the query data for `sendto` and `subject` values, and if they are present, it sends the rest of the data in an e-mail. It returns an HTML comment that flags that mail was sent.

When you use templates to process form data, you need to turn off result caching because the server must process the template each time the form is submitted. To turn off caching, embed the `Doc_Dynamic` command into your form handler pages, or set the `page(dynamic)` variable to 1. Alternatively, you can simply post directly to the *`file`*`.tml` page instead of to the *`file`*`.html` page.

Self Posting Forms

This section illustrates a self-posting form. This is a form on a page that posts the form data to back to the same page. The page embeds a Tcl command to check its own form data. Once the data is correct, the page triggers a redirect to the next page in the flow. This is a powerful technique that I use to create complex page flows using templates. Of course, you need to save the form data at each step. You can put the data in Tcl variables, use the data to control your application, or store it into a database. TclHttpd comes with a Session module, which is one way to manage this information. For details you should scan the session.tcl file in the distribution.

Example 18–13 shows the Form_Simple procedure that generates a simple self-checking form. Its arguments are a unique ID for the form, a description of the form fields, and the URL of the next page in the flow. The field description is a list with three elements for each field: a required flag, a form element name, and a label to display with the form element. You can see this structure in the template shown in Example 18–14 on page 258. The procedure does two things at once. It computes the HTML form, and it also checks if the required fields are present. It uses some procedures from the form module to generate form elements that retain values from the previous page. If all the required fields are present, it discards the HTML, saves the data, and triggers a redirect by calling Doc_Redirect.

Example 18–13 A self-checking form procedure.

```
proc Form_Simple {id fields nextpage} {
   global page
   if {![form::empty formid]} {
      # Incoming form values, check them
      set check 1
   } else {
      # First time through the page
      set check 0
   }
   set html "<!-- Self-posting. Next page is $nextpage -->\n"
   append html "<form action=\"$page(url)\" method=post>\n"
   append html "<input type=hidden name=formid value=$id>\n"
   append html "<table border=1>\n"
   foreach {required key label} $fields {
      append html "<tr><td>"
      if {$check && $required && [form::empty $key]} {
         lappend missing $label
         append html "<font color=red>*</font>"
      }
      append html "</td><td>$label</td>\n"
      append html "<td><input [form::value $key]></td>\n"
      append html "</tr>\n"
   }
   append html "</table>\n"
   if {$check} {
      if {![info exist missing]} {
         # No missing fields, so advance to the next page.
         # In practice, you must save the existing fields
         # at this point before redirecting to the next page.

         Doc_Redirect $nextpage
      } else {
         set msg "<font color=red>Please fill in "
         append msg [join $missing ", "]
         append msg "</font>"
         set html <p>$msg\n$html
      }
   }
   append html "<input type=submit>\n</form>\n"
   return $html
}
```

Example 18–14 shows a page template that calls `Form_Simple` with the required field description.

Example 18–14 A page with a self-checking form.

```
<html><head>
   <title>Name and Address Form</title>
</head>
<body bgcolor=white text=black>
   <h1>Name and Address</h1>
   Please enter your name and address.
   [myform::simple nameaddr {
      1 name    "Name"
      1 addr1   "Address"
      0 addr2"  "Address"
      1 city    "City"
      0 state   "State"
      1 zip     "Zip Code"
      0 country "Country"
   } nameok.html]
</body></html>
```

The `form` package

TclHttpd comes with a `form` package (`form.tcl`) that is designed to support self-posting forms. The `Form_Simple` procedure uses `form::empty` to test if particular form values are present in the query data. For example, it tests to see whether the `formid` field is present so that the procedure knows whether or not to check for the rest of the fields. The `form::value` procedure is useful for constructing form elements on self-posting form pages. It returns:

`name="`*`name`*`" value="`*`value`*`"`

The *`value`* is the value of form element *name* based on incoming query data, or just the empty string if the query value for *name* is undefined. As a result, the form can post to itself and retain values from the previous version of the page. It is used like this:

```
<input type=text [form::value name]>
```

The form::checkvalue and form::radiovalue procedures are similar to form::value but designed for checkbuttons and radio buttons. The form::select procedure formats a selection list and highlights the selected values. The form::data procedure simply returns the value of a given form element. These are summarized in Table 18–6 on page 261.

Programming Reference

This section summarizes many of the more useful functions defined by the server. These tables are not complete, however. You are encouraged to read through the code to learn more about the features offered by the server.

Table 18–1 summarizes the Httpd functions used when returning pages to the client.

Table 18–1 Httpd support procedures.

Httpd_Error *sock code*	Returns a simple error page to the client. The *code* is a numeric error code like 404 or 500.
Httpd_ReturnData *sock type data*	Returns a page with Content-Type *type* and content *data*.
Httpd_ReturnFile *sock type file*	Returns a *file* with Content-Type *type*.
Httpd_Redirect *newurl sock*	Generates a 302 error return with a Location of *newurl*.
Httpd_SelfUrl *url*	Expands *url* to include the proper http://*server:port* prefix to reference the current server.

Table 18–2 summarizes a few useful procedures provided by the Url module (url.tcl). The Url_DecodeQuery is used to decode query data into a Tcl-friendly list. The Url_Encode procedure is useful when encoding values directly into URLs. URL encoding is discussed in more detail on page 247.

Table 18–2 Url support procedures.

Url_DecodeQuery *query*	Decodes a www-url-encoded query string and return a name, value list.
Url_Encode *value*	Returns *value* encoded according to the www-url-encoded standard.
Url_PrefxInstall *prefix callback*	Registers *callback* as the handler for all URLs that begin with *prefix*. The callback is invoked with two additional arguments: *sock*, the handle to the client, and *suffix*, the part of the URL after *prefix*.

The `Doc` module provides procedures for configuration and generating responses, which are summarized in Tables 18–3 and 18–4, respectively.

Table 18–3 `Doc` procedures for configuration.

`Doc_Root` ?*directory*?	Sets or queries the *directory* that corresponds to the root of the URL hierarchy.
`Doc_AddRoot` *virtual directory*	Maps the file system *directory* into the URL subtree starting at *virtual*.
`Doc_ErrorPage` *file*	Specifies a *file* relative to the document root used as a simple template for error messages. This is processed by `DocSubstSystem` file in `doc.tcl`.
`Doc_CheckTemplates` *how*	If *how* is 1, then `.html` files are compared against corresponding `.tml` files and regenerated if necessary.
`Doc_IndexFile` *pattern*	Registers a file name *pattern* that will be searched for the default index file in directories.
`Doc_NotFoundPage` *file*	Specifies a *file* relative to the document root used as a simple template for page not found messages. This is processed by `DocSubstSystem` file in `doc.tcl`.
`Doc_PublicHtml` *dirname*	Defines the directory used for each users home directory. When a URL like ~*user* is specified, the *dirname* under their home directory is accessed.
`Doc_TemplateLibrary` *directory*	Adds *directory* to the `auto_path` so the source files in it are available to the server.
`Doc_TemplateInterp` *interp*	Specifies an alternate interpreter in which to process document templates (i.e., `.tml` files.)
`Doc_Webmaster` ?*email*?	Sets or queries the *email* for the Webmaster.

Table 18–4 `Doc` procedures for generating responses.

`Doc_Error` *sock errorInfo*	Generates a 500 response on *sock* based on the template registered with `Doc_ErrorPage`. *errorInfo* is a copy of the Tcl error trace after the error.
`Doc_NotFound` *sock*	Generates a 404 response on *sock* by using the template registered with `Doc_NotFoundPage`.
`Doc_Subst` *sock file* ?*interp*?	Performs a `subst` on the file and return the resulting page on *sock*. *interp* specifies an alternate Tcl interpreter.

The `Doc` module also provides procedures for cookies and redirects that are useful in document templates. These are described in Table 18–5.

Table 18–5 Doc procedures that support template processing.

`Doc_Coookie` *name*	Returns the cookie *name* passed to the server for this request, or the empty string if it is not present.
`Doc_Dynamic`	Turns off caching of the HTML result. Meant to be called from inside a page template.
`Doc_IsLinkToSelf` *url*	Returns 1 if the *url* is a link to the current page.
`Doc_Redirect` *newurl*	Raises a special error that aborts template processing and triggers a page redirect to *newurl*.
`Doc_SetCookie -name` *name* `-value` *value* `-path` *path* `-domain` *domain* `-expires` *date*	Sets cookie *name* with the given *value* that will be returned to the client as part of the response. The *path* and *domain* restrict the scope of the cooke. The *date* sets an expiration date.

Table 18–6 describes the form module that is useful for self-posting forms, which are discussed on page 257.

Table 18–6 The form package.

`form::data` *name*	Returns the value of the form value *name*, or the empty string.
`form::empty` *name*	Returns 1 if the form value *name* is missing or zero length.
`form::value` *name*	Returns `name="`*name*`" value="`*value*`"`, where *value* comes from the query data, if any.
`form::checkvalue` *name value*	Returns `name="`*name*`" value="`*value*`" CHECKED`, if *value* is present in the query data for *name*. Otherwise, it just returns `name="`*name*`" value="`*value*`"`.
`form::radiovalue` *name value*	Returns `name="`*name*`" value="`*value*`" CHECKED`, if the query data for *name* is equal to *value.* Otherwise, it just returns `name="`*name*`" value="`*value*`"`.
`form::select` *name valuelist args*	Generates a `select` form element with name *name*. The *valuelist* determines the `option` tags and values, and *args* are optional parameters to the main `select` tag.

Table 18–7 shows the initial elements of the page array that is defined during the processing of a template.

Table 18–7 Elements of the page array.

`query`	The decoded query data in a name, value list.
`dynamic`	If 1, the results of processing the template are not cached in the corresponding `.html` file.
`filename`	The file system pathname of the requested file (e.g., `/usr/local/htdocs/tclhttpd/index.html`).

Table 18–7 Elements of the `page` array. (Continued)

`template`	The file system pathname of the template file (e.g., `/usr/local/htdocs/tclhttpd/index.tml`).
`url`	The part of the url after the server name (e.g., `/tclhttpd/index.html`).
`root`	A relative path from the template file back to the root of the URL tree. This is useful for creating relative links between pages in different directories.

Table 18–8 shows the elements of the `env` array. These are defined during CGI requests, application-direct URL handlers, and page template processing:

Table 18–8 Elements of the `env` array.

`AUTH_TYPE`	Authentication protocol (e.g., `Basic`).
`CONTENT_LENGTH`	The size of the query data.
`CONTENT_TYPE`	The type of the query data.
`DOCUMENT_ROOT`	File system pathname of the document root.
`GATEWAY_INTERFACE`	Protocol version, which is `CGI/1.1`.
`HTTP_ACCEPT`	The Accept headers from the request.
`HTTP_AUTHORIZATION`	The Authorization challenge from the request.
`HTTP_COOKIE`	The cookie from the request.
`HTTP_FROM`	The From: header of the request.
`HTTP_REFERER`	The Referer indicates the previous page.
`HTTP_USER_AGENT`	An ID string for the Web browser.
`PATH_INFO`	Extra path information after the template file.
`PATH_TRANSLATED`	The extra path information appended to the document root.
`QUERY_STRING`	The form query data.
`REMOTE_ADDR`	The client’s IP address.
`REMOTE_USER`	The remote user name specified by Basic authentication.
`REQUEST_METHOD`	GET, POST, or HEAD.
`REQUEST_URI`	The complete URL that was requested.
`SCRIPT_NAME`	The name of the current file relative to the document root.
`SERVER_NAME`	The server name, e.g., `www.beedub.com`.
`SERVER_PORT`	The server’s port, e.g., 80.
`SERVER_PROTOCOL`	The protocol (e.g., `http` or `https`).
`SERVER_SOFTWARE`	A software version string for the server.

Standard Application-Direct URLs

The server has several modules that provide application-direct URLs. These application-direct URLs lets you control the server or examine its state from any Web browser. You can look at the implementation of these modules as examples for your own application.

Status

The /status URL is implemented in the status.tcl file. The status module implements the display of hit counts, document hits, and document misses (i.e., documents not found). The Status_Url command enables the application-direct URLs and assigns the top-level URL for the status module. The default configuration file contains this command:

```
Status_Url /status
```

Assuming this configuration, the following URLs are implemented:

Table 18–9 Status application-direct URLs.

/status	Main status page showing summary counters and hit count histograms.
/status/doc	Shows hit counts for each page. This page lets you sort by name or hit count, and limit files by patterns.
/status/hello	A trivial URL that returns "hello".
/status/notfound	Shows miss counts for URLs that users tried to fetch.
/status/size	Displays an estimated size of Tcl code and Tcl data used by the TclHttpd program.
/status/text	This is a version of the main status page that doesn't use the graphical histograms of hit counts.

Debugging

The /debug URL is implemented in the debug.tcl file. The debug module has several useful URLs that let you examine variable values and other internal state. It is turned on with this command in the default configuration file:

```
Debug_Url /debug
```

Table 18–10 lists the /debug URLs. These URLs often require parameters that you can specify directly in the URL. For example, the /debug/echo URL echoes its query parameters:

```
http://yourserver:port/debug/echo?name=value&name2=val2
```

Table 18–10 Debug application-direct URLs.

`/debug/after`	Lists the outstanding `after` events.
`/debug/dbg`	Connects to *TclPro Debugger*. This takes a `host` and `port` parameter. You need to install `prodebug.tcl` from *TclPro* into the server's script library directory.
`/debug/echo`	Echoes its query parameters. Accepts a `title` parameter.
`/debug/errorInfo`	Displays the `errorInfo` variable along with the server's version number and Webmaster e-mail. Accepts `title` and `errorInfo` arguments.
`/debug/parray`	Displays a global array variable. The name of the variable is specified with the `aname` parameter.
`/debug/pvalue`	A more general value display function. The name of the variable is specified with the `aname` parameter. This can be a variable name, an array name, or a pattern that matches several variable names.
`/debug/raise`	Raises an error (to test error handling). Any parameters become the error string.
`/debug/source`	Sources a file from either the server's main library directory or the `Doc_TemplateLibrary` directory. The file is specified with the `source` parameter.

The sample URL tree that is included in the distribution includes the file `htdocs/hacks.html`. This file has several small forms that use the `/debug` URLs to examine variables and source files. Example18–15 shows the implementation of `/debug/source`. You can see that it limits the files to the main script library and to the script library associated with document templates. It may seem dangerous to have these facilities, but I reason that because my source directories are under my control, it cannot hurt to reload any source files. In general, the library scripts contain only procedure definitions and no global code that might reset state inappropriately. In practice, the ability to tune (i.e., fix bugs) in the running server has proven useful to me on many occasions. It lets you evolve your application without restarting it!

Example 18–15 The `/debug/source` application-direct URL implementation.

```
proc Debug/source {source} {
   global Httpd Doc
   set source [file tail $source]
   set dirlist $Httpd(library)
   if {[info exists Doc(templateLibrary)]} {
      lappend dirlist $Doc(templateLibrary)
   }
   foreach dir $dirlist {
      set file [file join $dir $source]
      if [file exists $file] {
         break
```

```
        }
    }
    set error [catch {uplevel #0 [list source $file]} result]
    set html "<title>Source $source</title>\n"
    if {$error} {
        global errorInfo
        append html "<H1>Error in $source</H1>\n"
        append html "<pre>$result<p>$errorInfo</pre>"
    } else {
        append html "<H1>Reloaded $source</H1>\n"
        append html "<pre>$result</pre>"
    }
    return $html
}
```

Administration

The /admin URL is implemented in the admin.tcl file. The admin module lets you load URL redirect tables, and it provides URLs that reset some of the counters maintained by the server. It is turned on with this command in the default configuration file:

```
Admin_Url /admin
```

Currently, there is only one useful admin URL. The /admin/redirect/reload URL sources the file named redirect in the document root. This file is expected to contain a number of Url_Redirect commands that establish URL redirects. These are useful if you change the names of pages and want the old names to still work.

The administration module has a limited set of application-direct URLs because the simple application-direct mechanism doesn't provide the right hooks to check authentication credentials. The HTML+Tcl templates work better with the authentication schemes.

Sending Email

The /mail URL is implemented in the mail.tcl file. The mail module implements various form handlers that e-mail form data. Currently, it is UNIX-specific because it uses /usr/lib/sendmail to send the mail. It is turned on with this command in the default configuration file:

```
Mail_Url /mail
```

The application-direct URLs shown in Table 18–11 are useful form handlers. You can specify them as the ACTION parameter in your <FORM> tags. The mail module provides two Tcl procedures that are generally useful. The MailInner procedure is the one that sends mail. It is called like this:

```
MailInner sendto subject from type body
```

The sendto and from arguments are e-mail addresses. The type is the Mime type (e.g., text/plain or text/html) and appears in a Content-Type header. The body contains the mail message without any headers.

Table 18–11 Application-direct URLS that e-mail form results.

`/mail/bugreport`	Sends e-mail with the `errorInfo` from a server error. It takes an `email` parameter for the destination address and an `errorInfo` parameter. Any additional arguments get included into the message.
`/mail/forminfo`	Sends e-mail containing form results. It requires these parameters: `sendto` for the destination address, `subject` for the mail subject, `href` and `label` for a link to display on the results page. Any additional arguments are formatted with the Tcl `list` command for easy processing by programs that read the mail.
`/mail/formdata`	This is an older form of `/mail/forminfo` that doesn't format the data into Tcl lists. It requires only the `email` and `subject` parameters. The rest are formatted into the message body.

The `Mail_FormInfo` procedure is designed for use in HTML+Tcl template files. It takes no arguments but instead looks in current query data for its parameters. It expects to find the same arguments as the `/mail/forminfo` direct URL. Using a template with `Mail_FormInfo` gives you more control over the result page than posting directly to `/mail/forminfo`, and is illustrated in Example 18–10 on page 255.

The TclHttpd Distribution

Get the TclHttpd distribution from the CD-ROM, or find it on the Internet at:

```
ftp://ftp.scriptics.com/pub/tcl/httpd/
http://www.scriptics.com/tclhttpd/
```

Quick Start

Unpack the tar file or the zip file, and you can run the server from the `httpd.tcl` script in the `bin` directory. On UNIX:

```
tclsh httpd.tcl -port 80
```

This command will start the Web server on the standard port (80). By default it uses port 8015 instead. If you run it with the `-help` flag, it will tell you what command line options are available. If you use *wish* instead of *tclsh,* then a simple Tk user interface is displayed that shows how many hits the server is getting.

On Windows you can double-click the `httpd.tcl` script to start the server. It will use *wish* and display the user interface. Again it will start on port 8015. You will need to create a shortcut that passes the `-port` argument, or edit the associated configuration file to change this. Configuring the server is described later.

Once you have the server running, you can connect to it from your Web browser. Use this URL if you are running on the default (nonstandard) port:

```
http://hostname:8015/
```

If you are running without a network connection, you may need to specify `127.0.0.1` for the hostname. This is the "localhost" address and will bypass the network subsystem.

```
http://127.0.0.1:8015/
```

Inside the Distribution

The TclHttpd distribution is organized into the following directories:

- `bin` — This has sample start-up scripts and configuration files. The `httpd.tcl` script runs the server. The `tclhttpd.rc` file is the standard configuration file. The `minihttpd.tcl` file is the 250-line version. The `torture.tcl` file has some scripts that you can use to fetch many URLs at once from a server.
- `lib` — This has all the Tcl sources. In general, each file provides a package. You will see the `package require` commands partly in `bin/httpd.tcl` and partly in `bin/tclhttpd.rc`. The idea is that only the core packages are required by `httpd.tcl`, and different applications can tune what packages are needed by adjusting the contents of `tclhttpd.rc`.
- `htdocs` — This is a sample URL tree that demonstrates the features of the Web server. There is also some documentation there. One directory to note is `htdocs/libtml`, which is the standard place to put site-specific Tcl scripts used with the Tcl+HTML template facility.
- `src` — There are a few C source files for a some optional packages. These have been precompiled for some platforms, and you can find the compiled libraries back under `lib/Binaries` in platform-specific subdirectories.

Server Configuration

TclHttpd configures itself with three main steps. The first step is to process the command line arguments described in Table 18–12. The arguments are copied into the `Config` Tcl array. Anything not specified on the command line gets a default value. The next configuration step is to source the configuration file. The default configuration file is named `tclhttpd.rc` in the same directory as the start-up script (i.e., `bin/tclhttpd.rc`). This file can override command line arguments by setting the `Config` array itself. This file also has application-specific `package require` commands and other Tcl commands to initialize the application. Most of the Tcl commands used during initialization are described in the rest of this section. The final step is to actually start up the server. This is done back in the main `httpd.tcl` script. For example, to start the server for the document tree under `/usr/local/htdocs` and your own e-mail address as Webmaster, you can execute this command to start the server:

```
tclsh httpd.tcl -docRoot /usr/local/htdocs -webmaster welch
```

Alternatively, you can put these settings into a configuration file, and start the server with that configuration file:

```
tclsh httpd.tcl -config mytclhttpd.rc
```

In this case, the mytclhttpd.rc file could contain these commands to hardwire the document root and Webmaster e-mail. In this case, the command line arguments *cannot* override these settings:

```
set Config(docRoot) /usr/local/htdocs
set Config(webmaster) welch
```

Command Line Arguments

There are several parameters you may need to set for a standard Web server. These are shown below in Table 18–12. The command line values are mapped into the Config array by the httpd.tcl start-up script.

Table 18–12 Basic TclHttpd Parameters.

Parameter	Command Option	Config Variable
Port number. The default is `8015`.	`-port` *`number`*	`Config(port)`
Server name. The default is `[info hostname]`.	`-name` *`name`*	`Config(name)`
IP address. The default is `0`, for "any address".	`-ipaddr` *`address`*	`Config(ipaddr)`
Directory of the root of the URL tree. The default is the `htdocs` directory.	`-docRoot` *`directory`*	`Config(docRoot)`
User ID of the TclHttpd process. The default is `50`. (UNIX only.)	`-uid` *`uid`*	`Config(uid)`
Group ID of the TclHttpd process. The default is `100`. (UNIX only.)	`-gid` *`gid`*	`Config(gid)`
Webmaster e-mail. The default is `webmaster`.	`-webmaster` *`email`*	`Config(webmaster)`
Configuration file. The default is `tclhttpd.rc`.	`-config` *`filename`*	`Config(file)`
Additional directory to add to the `auto_path`.	`-library` *`directory`*	`Config(library)`

Server Name and Port

The name and port parameters define how your server is known to Web browsers. The URLs that access your server begin with:

```
http://name:port/
```

If the port number is 80, you can leave out the port specification. The call that starts the server using these parameters is found in `httpd.tcl` as:

```
Httpd_Server $Config(name) $Config(port) $Config(ipaddr)
```

Specifying the IP address is necessary only if you have several network interfaces (or several IP addresses assigned to one network interface) and want the server to listen to requests on a particular network address. Otherwise, by default, server accepts requests from any network interface.

User and Group ID

The user and group IDs are used on UNIX systems with the `setuid` and `setgid` system calls. This lets you start the server as root, which is necessary to listen on port 80, and then switch to a less privileged user account. If you use Tcl+HTML templates that cache the results in HTML files, then you need to pick an account that can write those files. Otherwise, you may want to pick a very unprivileged account.

The `setuid` function is available through the TclX (Extended Tcl) `id` command, or through a `setuid` extension distributed with TclHttpd under the src directory. If do not have either of these facilities available, then the attempt to change user ID gracefully fails. See the `README` file in the `src` directory for instructions on compiling and installing the extensions found there.

Webmaster Email

The Webmaster e-mail address is used for automatic error reporting in the case of server errors. This is defined in the configuration file with the following command:

```
Doc_Webmaster $Config(webmaster)
```

If you call `Doc_Webmaster` with no arguments, it returns the e-mail address you previously defined. This is useful when generating pages that contain `mailto:` URLs with the Webmaster address.

Document Root

The document root is the directory that contains the static files, templates, CGI scripts, and so on that make up your Web site. By default the httpd.tcl script uses the *htdocs* directory next to the directory containing *httpd.tcl*. It is worth noting the trick used to locate this directory:

```
file join [file dirname [info script]] ../htdocs
```

The `info script` command returns the full name of the `http.tcl` script, `file dirname` computes its directory, and `file join` finds the adjacent directory. The path `../htdocs` works with `file join` on any platform. The default location of the configuration file is found in a similar way:

```
file join [file dirname [info script]] tclhttpd.rc
```

The configuration file initializes the document root with this call:

```
Doc_Root $Config(docRoot)
```

If you need to find out what the document root is, you can call Doc_Root with no arguments and it returns the directory of the document root. If you want to add additional document trees into your Web site, you can do that with a call like this in your configuration file:

```
Doc_AddRoot directory urlprefix
```

Other Document Settings

The `Doc_IndexFile` command sets a pattern used to find the index file in a directory. The command used in the default configuration file is:

```
Doc_IndexFile index.{htm,html,tml,subst}
```

If you invent other file types with different file suffixes, you can alter this pattern to include them. This pattern will be used by the Tcl `glob` command.

The `Doc_PublicHtml` command is used to define "home directories" on your HTML site. If the URL begins with ~username, then the Web server will look under the home directory of username for a particular directory. The command in the default configuration file is:

```
Doc_PublicHtml public_html
```

For example, if my home directory is `/home/welch`, then the URL `~welch` maps to the directory `/home/welch/public_html`. If there is no `Doc_PublicHtml` command, then this mapping does not occur.

You can register two special pages that are used when the server encounters an error and when a user specifies an unknown URL. The default configuration file has these commands:

```
Doc_ErrorPage error.html
Doc_NotFoundPage notfound.html
```

These files are treated like templates in that they are passed through `subst` in order to include the error information or the URL of the missing page. These are pretty crude templates compared to the templates described earlier. You can count only on the `Doc` and `Httpd` arrays being defined. Look at the `Doc_SubstSystemFile` in `doc.tcl` for the truth about how these files are processed.

Document Templates

The template mechanism has two main configuration options. The first specifies an additional library directory that contains your application-specific scripts. This lets you keep your application-specific files separate from the TclHttpd implementation. The command in the default configuration file specifies the libtml directory of the document tree:

```
Doc_TemplateLibrary [file join $Config(docRoot) libtml]
```

You can also specify an alternate Tcl interpreter in which to process the templates. The default is to use the main interpreter, which is named {} according to the conventions described in Chapter 19.

```
Doc_TemplateInterp {}
```

Log Files

The server keeps standard format log files. The `Log_SetFile` command defines the base name of the log file. The default configuration file uses this command:

```
Log_SetFile /tmp/log$Config(port)_
```

By default the server rotates the log file each night at midnight. Each day's log file is suffixed with the current date (e.g., `/tmp/log`*`port`*`_990218`.) The error log, however, is not rotated, and all errors are accumulated in `/tmp/log`*`port`*`_error`.

The log records are normally flushed every few minutes to eliminate an extra I/O operation on each HTTP transaction. You can set this period with `Log_FlushMinutes`. If minutes is 0, the log is flushed on every HTTP transaction. The default configuration file contains:

```
Log_FlushMinutes 1
```

CGI Directories

You can register a directory that contains CGI programs with the `Cgi_Directory` command. This command has the interesting effect of forcing all files in the directory to be executed as CGI scripts, so you cannot put normal HTML files there. The default configuration file contains:

```
Cgi_Directory /cgi-bin
```

This means that the `cgi-bin` directory under the document root is a CGI directory. If you supply another argument to `Cgi_Directory`, then this is a file system directory that gets mapped into the URL defined by the first argument. You can also put CGI scripts into other directories and use the `.cgi` suffix to indicate that they should be executed as CGI scripts.

The `cgi.tcl` file has some additional parameters that you can tune only by setting some elements of the `Cgi` Tcl array. See the comments in the beginning of that file for details.