

TclHttpd Web Server

This chapter describes TclHttpd, a web server built entirely in Tcl. The web server can be used as a stand-alone 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 application.

This chapter will appear in the 3rd Edition of *Practical Programming in Tcl and Tk*, by Brent Welch.

Copyright (c) 1999 Brent Welch, all rights reserved. Please contact welch@acm.org for more information.

*T*clHttpd 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 towards the end of the chapter, on page 230. It works on Unix, Windows, and Macintosh. You can have the server up and running quickly.

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. There is none of the complexity of 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, 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

This section describes the software architecture of TclHttpd. This explains how the server dispatches requests for URLs to different modules and what basic functions are available to help respond to URL requests. You need to understand this as you go to extend the server's functionality or integrate the server into your own application. As we go along, there will be references to Tcl files in the server's implementation. These are all found in the `lib` directory of the distribution, and you may find it helpful to read the code to learn more about the implementation. [CD-ROM ref] Figure 18-1 shows the basic components of the server.

At the core is the `Httpd` module, 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 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 Document 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.

A Simple URL Domain

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

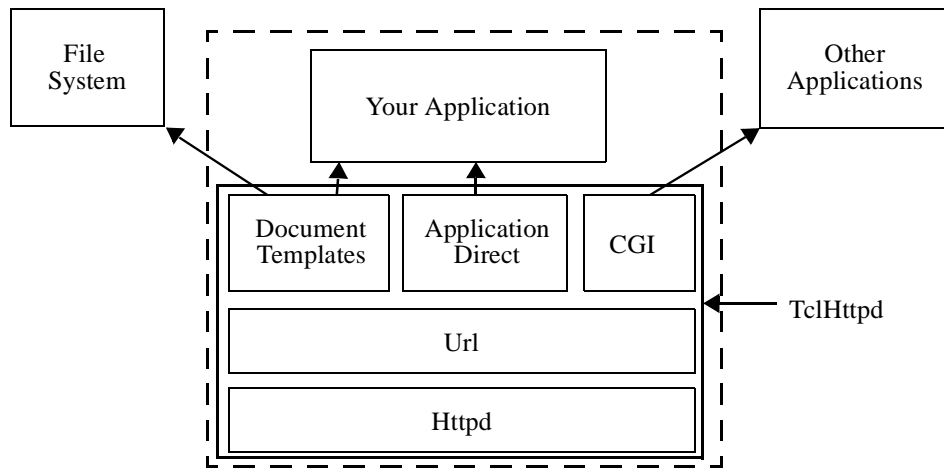


Figure 18-1 The dotted box represents one application that embeds TclHttpd. Document templates and Application Direct URLs provide direct connections from an HTTP request to your application.

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. This example shows the basic structure of a domain handler and introduces some useful support procedures. 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 start with `/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_DecompileQuery $data(query)] {
            append html "<tr><td>$name</td><td>$value</td></"
        }
        append html "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. 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:

```
foreach {name value} [Url_DecodeQuery $data(query)] {
  # Process query data
}
```

[TODO - Clean up `TclHttpd` so there is a cleaner interface to the query data]

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 two related procedures, `Httpd_ReturnFile` and `Httpd_Redirect`. The first returns the contents of a file as the result of the transaction. It is called like this:

```
Httpd_ReturnFile $sock mimetype filename
```

The `Httpd_Redirect` procedure generates a 302 return code that causes the remote client to fetch a different URL. This is a very useful trick employed in complex page flows. It is invoked like this:

```
Httpd_Redirect newurl $sock
```

If you need to return an error in response to a request, use `Httpd_Error`. Its code parameter is a standard numeric code (e.g., 404 for Not Found)

```
Httpd_Error $sock code
```

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. The Tcl procedures compute a string that is the result data, which is usually HTML. That's all there is to it.

[TODO - Use the implementation of the direct interface to highlight Tcl's introspection capabilities in the appropriate chapter.]

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 echos 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 a little funny.

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.

[TODO - try a hard-wired format and figure out the % encoding problem]

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 non-alphanumeric 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, you need to think about the encoding. Example 11-3 on page 126 show the `Url_Decode` procedure that decodes these values. This procedure is used by the application direct domain implementation, so your procedure argument values are already decoded.

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, or personal icons, that is organized by user and domain names. The idea is that the database contains images corresponding to your email 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
    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
}
```

The `Mtype` module that maintains a map from file suffixes to MIME content types. For example, it maps `.gif` to `image/gif`.

Document Types

The Document domain 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.

You can make up new types to support your application. For 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 implement 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 can 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.
- The server creates a `page` global Tcl variable that has context about the page being processed. Table X 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 Y 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 sourced by the server from the URL document root down to the directory containing a template file. These files are designed to contain procedure definitions and variable settings that are shared among pages. 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 the procedures are loaded when needed.

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 byte-code 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 advantage of putting code into `.tml` files is that changes are picked up immediately with no effort on your part. However, that code is only run one time, so the byte-code compiler just adds overhead.

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 18-4 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 18-4 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 18-5 shows a sample template file:

Example 18-5 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 18-6 has a simple implementation of `SitePage`:

Example 18-6 `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 18–7 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 18–7 `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 $$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 both change 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 18-5. The proper label for this is already defined in `$site(pages)`, so we could introduce a `SiteLink` procedure that uses this:

Example 18-8 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 `.tcl` file, the templates will automatically track the changes.

Form Handlers

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 hits 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 users 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 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 (page 29) 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 simply 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 forms 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 18–9 shows a form that uses this handler. Other built in handlers are described starting at page 236

Example 18–9 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 18–10. 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 email, if you format it with Tcl list structure you can process it quite easily.

Example 18–10 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
}
```

Template Form Handlers

The drawback of using the built-in form handlers is that you have to 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

email. 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 in your form handler pages, or set the `page(dynamic)` variable to 1.

Example 18–11 shows a name and address form. This form is on a page that posts the form data to itself. Once all the data has been filled in correctly, the `Form_Check` procedure in Example 18–12 redirects to the next page in the flow.

Example 18–11 A self-checking form.

```
<form action=$page(url) method=post>
<input type=hidden name=form value=NameAddr>
[Form_Check nextpage.html mark {email first last addr1 city
zip}]
<table>
[foreach {label name} {
  "Email"      email
  "First Name" first
  "Last Name"  last
  "Address 1"  addr1
  "Address 2"  addr2
  "City"       city
  "State"      state
  "Zip/Post"   zip
  "Country"    country
} {
  if {[info exist mark($name)]} {
    append _ "<tr><td>$mark($name) $label</td>\n"
  } else {
    append _ "<tr><td>$label</td>\n"
  }
  append _ "<td><input [form::value $name]</td></tr>\n"
}
set _]
</table>
<input type=submit>
</form>
```

This page embeds a `foreach` loop to make generation of the HTML table for the form easier. The loop appends to the `_` variable, which is used by convention for this purpose in web pages. The `form::value` procedure, which comes with `TclHttpd`, is designed for self-posting forms. It returns:

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

The `value` is the value of the form element based on incoming query data, or just the empty string if the query value for `name` is undefined. This way the form can post to itself and retain values from the previous version of the page.

Example 18–12 The Form_Check form handler.

```

proc Form_Check {nextpage markVar required} {
    global page
    upvar $markVar mark
    if {[info exist page(query)]} {
        array set query $page(query)
        if {[info exist mark]} {
            unset mark
        }
        foreach field $required {
            if {![info exist query($field)] ||
                [string length $query($field)] == 0} {
                set mark($field) *
            }
        }
        if {![info exist mark]} {

            # No missing fields, so advance to the next page.
            # In practice, you must save the existing fields
            # before redirecting to the next page.

            Doc_Redirect $nextpage
        }
    }
    return ""
}

```

The `Doc_Redirect` procedure raises a special error that causes `TclHttpd` to redirect the client browser to a different page. Otherwise, the `Form_Check` procedure defines an element of the `mark` array for each missing field. The code in Example 18–11 uses this to flag missing fields. In practice, your form handler should do something with the good data, such as put it into a database.

As a matter of style, there is a bit too much code inside the HTML page in Example 18–11. You could consider combining the logic that displays the form with the logic that checks for required fields. In this case the HTML page has a single call to your procedure that implements "both sides" of the form: the HTML input form and the Tcl code that validates the field and saves the results. This approach is useful if you re-use the same form on different pages.

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.beedub.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 (non-standard) port:

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

If you are running without an internet 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 startup 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 the 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/libhtml`, 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 1–1. 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 startup 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 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 email 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 hard-wire the document root and webmaster email. 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 1–1. The command line values are mapped into the `Config` array by the `httpd.tcl` startup script.

Table 1–1 Basic TclHttpd Parameters

Parameter	Command Option	Config Variable
Port number. The default is 8015.	<code>-port number</code>	<code>Config(port)</code>
Server name. The default is <code>[info hostname]</code> .	<code>-name name</code>	<code>Config(name)</code>
IP address. The default is 0, for "any address".	<code>-ipaddr address</code>	<code>Config(ipaddr)</code>

Table 1-1 Basic TclHttpd Parameters

Parameter	Command Option	Config Variable
Directory of the root of the URL tree. The default is the <code>htdocs</code> directory.	<code>-docRoot directory</code>	<code>Config(docRoot)</code>
User ID of the TclHttpd process. The default is 50. (UNIX only.)	<code>-uid uid</code>	<code>Config(uid)</code>
Group ID of the TclHttpd process. The default is 100. (UNIX only.)	<code>-gid gid</code>	<code>Config(gid)</code>
Webmaster email. The default is <code>webmaster</code> .	<code>-webmaster email</code>	<code>Config(webmaster)</code>
Configuration file. The default is <code>tclhttpd.rc</code> .	<code>-config filename</code>	<code>Config(file)</code>
Additional directory to add to the <code>auto_path</code> .	<code>-library directory</code>	<code>Config(library)</code>

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 only necessary 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 ID 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` Tcl command is implemented by a C extension found under the `src` directory. If you have not compiled this for your platform, 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 email 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 email 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 website 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 if the 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 only count 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 `libtmpl` directory of the document tree:

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

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 17.

```
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/logport_990218.`) The error log, however, is not rotated, and all errors are accumulated in `/tmp/logport_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 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

they should be executed as CGI scripts. The various file types supported by the server are described later.

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

Standard Application-Direct URLs

The server has several modules that provide application-direct URLs. These application-direct URL 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 1–2 Status application-direct URLs.

<code>/status</code>	Main status page showing summary counters and hit count histograms.
<code>/status/doc</code>	Show hit counts for each page. This page lets you sort by name or hit count, and limit files by patterns.
<code>/status/hello</code>	A trivial URL that returns "hello".
<code>/status/notfound</code>	Show miss counts for URLs that users tried to fetch.
<code>/status/size</code>	The displays an estimated size of Tcl code and Tcl data used by the TclHttpd program.
<code>/status/text</code>	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 1–3 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 1-3 Debug application-direct URLs.

<code>/debug/after</code>	List the outstanding <code>after</code> events.
<code>/debug/dbg</code>	Rendez-vous with <i>TclPro Debugger</i> . This takes a <code>host</code> and <code>port</code> parameter. You need to install <code>prodebug.tcl</code> from <i>TclPro</i> into the server's script library directory.
<code>/debug/echo</code>	Echos its query parameters. Accepts a <code>title</code> parameter.
<code>/debug/errorInfo</code>	Displays the <code>errorInfo</code> variable along with the servers version number and webmaster email. Accepts <code>title</code> and <code>errorInfo</code> arguments.
<code>/debug/parray</code>	Displays a global array variable. The name of the variable is specified with the <code>aname</code> parameter.
<code>/debug/pvalue</code>	A more general value display function. The name of the variable is specified with the <code>aname</code> parameter. This can be a scalar, an array, or a pattern that matches several variable names.
<code>/debug/raise</code>	Raise an error (to test error handling). Any parameters become the error string.
<code>/debug/source</code>	Source a file from either the server's main library directory or the <code>Doc_TemplateLibrary</code> directory. The file is specified with the <code>source</code> 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. Example 18-13 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 just contain 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 18-13 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 email form data. Currently it is UNIX only as 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 following application-direct URLs are useful form handlers. You can specify them as the `ACTION` parameter in your `<FORM>` tags.

Table 1-4 Application-direct URLs that email form results.

<code>/mail/bugreport</code>	This sends email with the <code>errorInfo</code> from a server error. It takes an <code>email</code> parameter for the destination address and an <code>errorInfo</code> parameter. Any additional arguments get included into the message.
------------------------------	---

Table 1-4 Application-direct URLs that email form results.

<code>/mail/forminfo</code>	This is designed to send email containing form results. It requires these parameters: <code>sendto</code> for the destination address, <code>subject</code> for the mail subject, <code>href</code> and <code>label</code> for a link to display on the results page. Any additional arguments are formatted with the Tcl <code>list</code> command for easy processing by programs that read the mail.
<code>/mail/formdata</code>	This is an older form of <code>/mail/forminfo</code> that doesn't format the data into Tcl lists. It only requires the <code>email</code> and <code>subject</code> parameters. The rest are formatted into the message body.

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 email 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.

The `Mail_FormInfo` procedure is designed for use in HTML+Tcl template files. It takes no arguments, but insteads 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 18-9 on page 228.

