The Httpd Module and Toadhttpd

Presented at

The 25th Annual Annual Tcl Developer's Conference
(Tcl'2018)
Houston, TX
October 15-19, 2018

Abstract

This paper explores the internals of the new Httpd module in tcllib. It will explain the development path from the original Tclhttpd, and why objects, coroutines, and the demands of ever more complex web applications drove this new approach. The paper will also explore the development of the httpd module into a full fledged web server, Toadhttpd. The paper will also explore embedding the httpd module into an existing application (the Integrated Recoverability model) as a documentation viewer. The paper will also outline how the httpd module will be used in upcoming projects as a supercomputing job dispatch hub.

Sean Deely Woods
Senior Developer
Test and Evaluation Solutions, LLC
400 Holiday Court
Suite 204
Warrenton, VA 20185
Email: yoda@etoyoc.com
Website: http://www.etoyoc.com

Introduction

For clarity, this paper discusses two different projects:

Httpd 4.0	A module for Tcllib which implements an HTTP listener API. Intended for embedding in existing applications as well as running tests that require an HTTP or SCGI listener.
Toadhttpd	A fully developed web server which uses Httpd as its core. Intended as a replacement for Tclhttpd

At it's simplest, the Httpd 4.0 module can be used internally by any program which has access to Tcllib:

```
# Simple Server with Httpd
package require httpd 4.0
::httpd::server create HTTPD \
   port 8015 doc_root ~/htdocs
HTTPD start
```

Httpd 4.0 and Toadhttpd started life as version 4.0 or the venerable Tclhttpd. Influenced by the likes of HereTcl and Wibble, I wanted to bring coroutines and TclOO into Tclhttpd. I succeeded… in making a third coroutine/TclOO based web server.

Trying to reconcile the new coroutine and TclOO architecture with the old namespace and virtual host interpreter approach was impossible. And because the first thing most application writers had to do was patch the Tclhttpd internals, trying to devise a migration path from existing Tclhttpd apps was going to be fruitless.

What helpful concepts I could steal from the old Tclhttpd, I did steal. Document Templates (.tml files) work exactly the same way in Httpd 4.0 as they work in Tclhttpd. And actually, if templates don't require too much in the way access to the old Tclhttpd internals, most content will port straight from the old server to the new.

Application direct Urls in Httpd 4.0 are slightly more complicated, but in my defense, they really should have been this complicated all along.

Tclhttpd's attempts to hide complexities led to even more complexities. There were magic variables that had to be set if you were not returning a content type other than HTML. There was a magic command to hint to the server that your page wanted to do a 3xx redirect. A magic command told the client not to cache the page. None of them got to the heart of the matter: most web applications need access to the incoming headers from the request and control of the outgoing headers of the reply. At the same time, not all incoming forms map neatly to a key/value list suitable for the arguments of a Tcl proc.

To contrast the two approaches, let me pull examples directly from Chapter 18 of Brent Welch's *Practical Programming in Tcl/Tk:*

```
# Application Direct URL - Httpd 3.5.x
Direct_Url /demo Demo
Direct_Url /faces Faces
proc Demo {} {
  return “
<html><head><title>Demo page</title></head>
<body><h1>Demo page</h1>
<a href=/demo/time>What time is it?</a>
<form action=/demo/echo>
Data: <input type=text name=data>
<br>
<input type=submit name=echo value=’Echo Data’>
</form>
</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
}
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
}
```

Now I present the the equivilent code in Httpd 4.0:

```
# Application Direct URL - Httpd 4.0

# 1) Note that the serve is an object
::httpd::server create SERVER port 8015
# Stock server doesn’t have a dispatch
# implementation, this plugin provides a simple
# one
SERVER plugin \
  dispatch ::httpd::plugin.dict_dispatch
SERVER start

SERVER uri direct * demo {} {
  my puts “
<html><head><title>Demo page</title></head>
<body><h1>Demo page</h1>
<a href=/demo/time>What time is it?</a>
<form action=/demo/echo>
Data: <input type=text name=data>
<br>
<input type=submit name=echo value=’Echo Data’>
</form>
</body></html>"
}

SERVER uri direct * demo/time {} {
  set form [my FormData]
  if {[dict exists $form format]} {
    set fmt [dict get $form format]
  } else {
    set fmt "%H:%M:%S"
  }
  my reply Content-Type Text/Plain
  my variable reply_body
  set reply_body \
    [clock format [clock seconds] \
     -format $fmt]
}

SERVER uri direct * demo/echo {} {
  # Compute a page that echos the query data
  my puts “<head><title>Echo</title></head>"
  my puts "<body><table>"
  foreach {name value} [my FormData] {
    my puts \
      ”<tr><td>$name</td><td>$value</td></tr>”
  }
  my puts “</tr></table>”
}


SERVER uri direct * faces/byemail {
  superclass ::httpd::content.file
} {
  my variable reply_file
  set email [dict getnull [my FormData] email]
  set reply_file [Faces_ByEmail $email]
  my reply set Content-Type \
      [::fileutil::magic::filetype $reply_file]
}
```

The first major change is that in Httpd 4.0, the server is an object instead of an interpreter. This allows a process to support more than one server listening on more than one port at a time. And it allows each of those ports to be bound to a different set of rules.

The default server ships with an empty dispatch method. It is assumed that a user of Httpd will be providing one of their own. Tcllib provides a plug-in called `httpd::plugin.dict_dispatch` which implements a dict based dispatch system. That plugin provides a method ensemble `uri` which includes a method `uri direct` to attach a method body to a URI pattern.

The first argument to `uri direct` is the pattern for the host name, or * for any host name. The second is the pattern for the REQUEST_PATH. Note that the leading slashes on paths will be removed. The third argument is a dict that will be passed to the server’s dispatching system in order to tell it how to reply to this request. The final argument is the body of a method that will actually generate the reply.

Behind the scenes, the `uri direct` command creates a class with a unique name for every uri pattern registered with the command. That body ends up as the `content` method of that custom class.

If the field **superclass** is given, any classes listed will be fed to a superclass statement in the newly formed class. You will note that in the Httpd 4.0 example, the *faces/byemail* url inherited from the `httpd::content.file` class. This is because the `httpd::content.file` class has a dispatcher that is more appropriate for returning streams of binary data. If the **reply_file** variable is set, it’s dispatch method knows to transmit the designated file. We’ll get into writing your own dispatch method a little later in this paper.

Socket programming with Coroutines

To understand what drove the implementation of Httpd 4.0, you have to understand implementing socket programming using coroutines. Coroutines are made possible by the Non-Recursive Engine (NRE) that came out with Tcl 8.6. Quoting Tip 328:

> *…a coroutine allows a command to suspend its current execution and return (or yield) a value to its caller. The caller may later resume the coroutine at the point where it previously yielded, allowing it to perform further work and potentially yield further values.*

If you have written socket programming in versions of Tcl prior to 8.6, you have probably been using a state machine. You bind a command to intercept new bits of data coming in from a stream. But either you change that binding as the data exchange evolves, or you have some internal state tracked by socket that explains to your proc where you actually are in the exchange.

I have prepared a simple example of this style. The socket command invokes `connectProc` with each new connection. `connectProc` sets up the parameters for our stream, and binds incoming packets to our `streamProc` command.

The first line of the protocol is a user's name. The second line is some sort of authentication. All lines after that are an echo of the incoming line with "USER Said:" prepended to the output.

Our state is implemented as a dict value stored within a global array. For brevity we are doing without the normal error handling and other embellishments that typically adorn well written socket code.

```
# State Machine Socket Example
proc connectProc {
  chan clientip client port
} {
  chan configure $sock \
    -blocking 0 \
    -translation {auto crlf} \
    -buffering line
  dict set ::state($sock) \
    [dict create state open]
  chan event $sock readable \
   [list streamProc $sock]
}
proc streamProc sock {
  uplevel #0 ::state($sock) state
  set line [chan gets $sock]
  switch [dict get $state state] {
    open {
      dict set state state pass
      dict set state user $line
    }
    pass {
      if {$line != "password"} {
        puts $sock "Go away"
        close $sock
      }
      dict set state state auth
    }
    auth {
      set user [dict get $state user]
      puts $sock "$user Said: $line"
    }
  }
}

socket -server connectProc 666
```

In the next example I have implemented the same protocol using a coroutine. It starts off the same way with `socket` command invoking `connectProc` with each new connection. After configuring our stream's parameters we use the `coroutine` command to produce a coroutine for us. We tell that coroutine to call the `streamProc` command.

Next we bind readable events to the coroutine we have just created.

Inside the body of `streamProc` you will see multiple calls to the `yield` command. Those are the points where the coroutine gives up control, and waits for an event to happen.

Note that we don't have any kind of global state in this implementation. We create the variable `$user` and can utilize it further along in our script, as if this were a standard proc and we weren't being constantly interrupted by the need for more input.

Also note that we can terminate the coroutine at any point by invoking `return`.

```
# Coroutine Socket Example
proc connectProc {
  chan clientip client port
} {
  chan configure $sock \
    -blocking 0 \
    -translation {auto crlf} \
    -buffering line
  coroutine ::coro#$sock \
    [list streamProc $sock]
  chan event $sock readable \
    ::coro#$sock
}
proc streamProc sock {
  yield [info coroutine]
  set user [chan gets $sock]
  yield
  set pass [chan gets $sock]
  if {$line != "password"} {
    puts $sock "Go away"
    catch {close $sock}
    return
  }
  while {[chan gets $sock line]>=0} {
    puts $sock "$user Said: $line"
    yield
  }
  catch {close $sock}
}

socket -server connectProc 666
```

httpd::server

The server class, `httpd::server`, brokers connections, creates coroutines, and then hands off the transaction to `httpd::reply` instances to finish processing. But along the way there are plenty of opportunities to customize that process. But before we get to that, I have to take a few minutes out to discuss safety.

Safely Implementing HTTP

Web servers implement the Hypertext Transfer Protocol (HTTP), as outlined by a variety of RFCs, but the one most people can agree on is

RFC 2068. Or at least that what I'm agreeing on, and to quote Andrew Tanenbaum:

"the nice thing about standards is there are so many to choose from."

HTTP requests start as a single line, followed by MIME headers, and then possibly followed by a stream of data.

```
GET index.html HTTP/1.1
Host: www.example.com
```

HTTP replies start as a single line, followed by MIME headers, and then possibly followed by a stream of data.

```
HTTP/1.1 200 OK
Date: Mon, 23 May 2005 22:38:34 GMT
Content-Type: text/html; charset=UTF-8
Content-Length: 138
Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT
Server: Apache/1.3.3.7 (Unix)
ETag: "3f80f-1b6-3e1cb03b"
Accept-Ranges: bytes
Connection: close

<html>
<head>
  <title>An Example Page</title>
</head>
<body>
  Hello World, this is a very simple HTML
document.
</body>
</html>
```

The HTTP protocol itself is fairly straightforward. The problem is that attack bots exploit pedantic interpretations of the standard. Attacks range from the simple to the devious. The simplest is issuing a GET, with no mime headers nor CR/LF terminator. Your more advanced bots make earnest attempts at a stack smash by packing a request with a stream of random data but no newline in hopes that your parser eventually segfaults. The truly devious tactic is to take so long to finish sending the request that the server is either denying services to other requests or gets confused and allows the attacker to peer inside of other requests.

To shield the server against attacks of these sorts, the Httpd module employs a new feature added to the *coroutine* module in tcllib called `coroutine::util::gets_safety`. This is designed to be called by a coroutine and replace a naive `gets` call with one which terminates if the incoming line is getting unreasonably large or of the client seems to be taking entirely too long to get to the point.

HTTP replies start off life as a call from the binding on `socket -server` to the server's public `connect` method:

```
method connect {sock ip port} {
  ###
  # Consult our list of blocked addresses
  ###
  if {[my Validate_Connection $sock $ip]} {
    catch {close $sock}
    return
  }
  set uuid [my Uuid_Generate]
  # Create a coroutine
  set coro [coroutine ::httpd::coro::$uuid \
    {*}[namespace code \
       [list my Connect $uuid $sock $ip]]]
  # Bind that coroutine to the next readable event
  chan event $sock readable $coro
}
```

When the socket is readable, the event system calls the coroutine, and the coroutine picks up where it left off, inside of the `Connect` private method of the server.

```
method Connect {uuid sock ip} {
  # Yield immediately on creation
  yield [info coroutine]
  try {
    # Unbind the readable event that triggered me
    chan event $sock readable {}
    chan configure $sock -blocking 0 \
    -translation {auto crlf} -buffering line
    # Pull the request line or die trying
    set readCount [::coroutine::util::gets_safety\
      $sock 4096 http_request]
    # Pull the mime headers or die trying
    set mimetxt [my HttpHeaders $sock]
    dict set query UUID $uuid
    dict set query mimetxt $mimetxt
    # Build the server fields of the request
    dict set query http \
     [my ServerHeaders $ip $http_request $mimetxt]
    # Shim for plugins to manipulate the request
    my Headers_Process query
    # Generate a reply data structure
    set reply [my dispatch $query]
  } on error {err errdat} {
    # Log and generate a 400 Bad request
    return
  }
  if {[dict size $reply]==0} {
    # An empty data structure generates a
    # 404 Not Found error. But we will generate
    # that page as a normal request
    set reply $query
    my log BadLocation $uuid $query
    dict set reply http HTTP_STATUS \
      {404 Not Found}
    dict set reply template notfound
    dict set reply mixin \
      reply ::httpd::content.template
  }
  # Create an object to process the rest
  # of the reply
  set pageobj [::httpd::reply create \
     ::httpd::object::$uuid [self]]
  # Pass control of the rest of this reply
  # to that object
  tailcall $pageobj dispatch $sock $reply
}
```

The HttpHeaders method is implemented in a meta class for the module and looks like this on the inside:

```
method HttpHeaders {sock} {
  set result {}
  set LIMIT 8192
  chan configure $sock -blocking 0 \
    -translation {auto crlf} -buffering line
  while 1 {
    set readCount \
      [::coroutine::util::gets_safety \
       $sock $LIMIT line]
    if {$readCount<=0} break
    append result $line \n
    if {[string length $result] > $LIMIT} {
      error {Headers too large}
    }
  }
  return $result
}
```

Next we invoke the server's *dispatch* method after the MIME headers have been read and encoded. If a match is found, this method will return a dict. If none is found, it will return an empty value, which will allow the object to consult registered plugins for data. (We show how to implement the *dispatch* method later in the *Toadhttpd/ Httpd Plugins* section.)

httpd::reply

Any further explanation of the Httpd 4.0's inner workings will require stripping away the syntactic sugar of `uri direct`.

In the example in the next column, I have created an additional URI. Instead of using `uri direct`, I do things the hard way by creating my own class and registering it via `uri add`.

The *mixin* directive tells the server object which behaviors to mix into the object prior to invoking the object's *dispatch* method.

Mixins for `httpd::reply` not need to inherit any other classes. You can feel free to create a hierarchy of ancestry with your content generators and know that you won't get tangled with the hierarchy of the Httpd module's classes. Also note that mixins are assigned slots, which allows multiple classes to be mixed in orthogonally. If you have one class that implements site styles, that class can use a separate slot from the class designated to generate the content.

```
# Example - Manually performing the
# steps in uri direct
oo::class create mydemo.clay {
  method content {} {
    # Compute a page that echos the
    # query data
    set title [my clay get title]
    my puts “<head><title>$title</title></head>”
    my puts "<body><table>"
    foreach {name value} [my FormData] {
      my puts \
       ”<tr><td>$name</td><td>$value</td></tr>”
    }
    # Also expose data from the clay
    # data structure
    foreach {name value} [my clay get content/] {
      set tvalue [subst $value]
      my puts \
        ”<tr><td>$name</td><td>$tvalue</td></tr>”
      }
    }
    my puts “</tr></table>”
  }
}
# Use uri add to associate the class with the uri
SERVER uri add * demo/clay {
  mixin {content mydemo.echo}
  title {Clay is so cool}
  content/ {
    name {Example}
    uuid {[uuid::uuid generate]}
  }
}
# Use uri add to associate the same class
# with a different uri with different
# settings
SERVER uri add * demo/claymore {
  mixin {content mydemo.echo}
  title {Clay is so much cooler}
  content/ {
    name {A different example}
    uuid {[uuid::uuid generate]}
  }
}
```

method content

The `http::reply` class expects the developer to provide their own `content` method, implemented in another class that will be mixed into the reply object at runtime.

Unlike a proc registered with Tclhttpd's `Direct_Url` system, the `content` method does not return a value. Instead it populates an internal variable *reply_body*. For convenience, `httpd::reply` provides a `puts` method which appends the arguments provided to the *reply_body*.

You will also notice the `content` does not take any arguments. Instead of mapping incoming form data to arguments, Httpd 4.0 exposes them as a dict that is available on demand. Also available are the parameters fed into the `uri add` or `uri direct` method, as well as data discovered during the dispatch process. That information and the raw MIME headers are exposed by the `clay` method. I won't explain `clay` here, but there is a companion paper to this one where you can read more it. (*Clay: A Minimalist Toolkit for Sculpting TclOO.)* For now just think of it as an access function to a private dict.

method reply

The `reply` method ensemble allows the application to modify the headers of the outgoing reply. A reserved reply field *Status* can be used to connote that the page is returning a reply code that is not the standard 200 OK. For the default implementation of `httpd::reply` the Content-Size field is automatically computed, and the Content-Type is assumed to be utf-8 encoded HTML. If you replace the `dispatch` method via a mixin, you will have to manage the headers yourself.

When you are ready to output your content the `result reply` method while collect the information stored in the *reply* dict and output properly formatted MIME headers.

method dispatch

Writing a single page at a time of dynamic content is sufficient for simple projects. However, Httpd recognizes that the world is a complex place and that many web applications do not fit into that simplistic model. As such, the `dispatch` method represents the means to take complete control of the request and response. This level of control is needed for:

- Proxies
- Websockets
- Media streaming
- Chunked Encoding
- HTTP/2

`dispatch` takes two arguments: *socket* and *datastate. socket* is an standard Tcl channel, as created by the `socket` command. *datastate* is a dictionary which feeds the reply object configuration information. The data structure itself is populated by the server's own `dispatch` method, which may be influenced by plugins loaded into the server via the `plugin` method. The only reserved keys for Httpd module internals are:

delegate	Key/Value list of slots and objects or commands to delegate this slots to
http	Key/Value list of MIME headers from the request mapped to SCGI rules
mimetxt	The raw MIME headers for the request
mixin	Key/Value list of slots and mixed in classes
UUID	A GUUID unique to this web reply

```
# Method dispatch from httpd::reply
method dispatch {newsock datastate} {
  my variable chan request
  try {
    set chan $newsock
    chan event $chan readable {}
    chan configure $chan \
      -translation {auto crlf} -buffering line
    my clay mixinmap \
      {*}[dict getnull $datastate mixin]
    my clay delegate \
      {*}[dict getnull $datastate delegate]
    my reset
    set request [my clay get dict/ request]
    foreach {f v} $datastate {
      if {[string index $f end] eq "/"} {
        my clay merge $f $v
      } else {
        my clay set $f $v
      }
      if {$f eq "http"} {
        foreach {ff vf} $v {
          dict set request $ff $vf
        }
      }
    }
    my Session_Load
    my Log_Dispatched
    my Dispatch
  } on error {err errdat} {
    my error 500 $err \
      [dict get $errdat -errorinfo]
    my DoOutput
  }
}
```

method Dispatch

In practice I have found that most of the proforma work that the public dispatch method does is sufficient for all needs, and that I can confine all of the application specific code to a private method Dispatch which mixins can overwrite.

Generating a block of HTML content on the fly has a simple Dispatch method:

```
# Stock Dispatch from httpd::reply
method Dispatch {} {
  # Invoke the URL implementation.
  my content
  my DoOutput
}
```

Serving a file base URL is slightly more complicated, because I could be delivering dynamically generated HTML (for directory listings, Markdown, or Template files), or it could be transmitting binary files:

```
method Dispatch {} {
  my variable reply_body reply_file reply_chan
  my variable chan
  try {
    my reset
    # Invoke the URL implementation.
    my content
  } on error {err errdat} {
    my error 500 $err \
      [dict get $errdat -errorinfo]
    tailcall my DoOutput
  }
  if {$chan eq {}} return
  my wait writable $chan
  if {![info exists reply_file]} {
    tailcall my DoOutput
  }
  try {
    chan configure $chan \
      -translation {binary binary}
    ###
    # Return a stream of data from a file
    ###
    set size [file size $reply_file]
    my reply set Content-Length $size
    append result [my reply output] \n
    chan puts -nonewline $chan $result
    set reply_chan [open $reply_file r]
    my log SendReply [list length $size]
    ###
    # Output the file contents. With no -size
    # flag, channel will copy until EOF
    ###
    chan configure $reply_chan \
      -translation {binary binary} \
      -buffersize 4096 -buffering full -blocking 0
    my ChannelCopy $reply_chan $chan -chunk 4096
  } finally {
    my TransferComplete $reply_chan $chan
  }
```

And then you get the really complicated cases where we are proxying data:

```
# Dispatch method from httpd::content.proxy
method Dispatch {} {
  my variable sock chan
  if {[catch {my proxy_channel} sock errdat]} {
    my error 504 \
      {Service Temporarily Unavailable} \
      [dict get $errdat -errorinfo]
    tailcall my DoOutput
  }
  if {$sock eq {}} {
    my error 404 {Not Found}
    tailcall my DoOutput
  }
  chan event $sock writable [info coroutine]
  yield
  try {
    my ProxyRequest $chan $sock
    my ProxyReply   $sock $chan
  } finally {
    my TransferComplete $chan $sock
  }
}
```

And know that internally the ProxyRequest and ProxyReply are doing further actions. Because we are in a coroutine, we can just invoke them as a subroutine. Those methods can yield every bit as well as the Dispatch method. When they are finished yielding, and either return or reach the end of the body, control will return here.

I like coroutines because I can actually read the code in execution order. If you've ever had to debug state based socket code, it can be a real head scratcher sometimes.

Toadhttpd

Up until now everything discussed has been about the Httpd module. While it's nice and pretty usable on it's own, it lacks many of the finishes that would make it suitable to stand up as a public webserver.

Enter Toadhttpd. Toadhttpd builds on the Httpd module and adds all of the implementation features that a public facing web server actually needs. Logging. Session control. Caching. Plugins to implement your own micro social network. (Ok that one is still in development, but it's coming along.)

Getting Toadhttpd

Toadhttpd hasn't grown large enough to merit a binary distribution as of yet. It is distributed in source form in a fossil repository. So step one is cloning and unpacking the code.

```
# Clone and upack the fossil sources
# Feel free to adjust the paths to your liking
mkdir -p ~/tcl/fossil/
fossil clone https://chiselapp.com/user/hypnotoad/
repository/toadhttpd ~/tcl/fossil/toadhttpd.fos
mkdir -p ~/tcl/sandbox/toadhttpd
cd ~/tcl/sandbox/toadhttpd
fossil open ~/tcl/fossil/toadhttpd.fos
```

The fossil repo contains a tcl based installer, and it is intended that you work from an installed version of the code rather than operate directly from the sources. Several if the modules do not version control their finished form, so the installer actually assembles them for you.

It's also intended that every server running Toadhttpd has an independent copy of all of the source code modules. In this way, you can evaluated newer versions before switching them over to production. And you can also keep an old reliable site up until the heat death of the Universe without ever having to update it.

```
# Make a directory to host your content from
> tclsh ~/tcl/sandbox/toadhttpd/make.tcl \
   install ~/www/mysite
```

~/www/mysite now contains a complete installation of Toadhttpd.

```
cd ~/www/mysite
ls
htdocs                      httpd.tcl        log
modules                      plugin                 var
```

To run the website, simply run the **httpd.tcl** file inside of your friendly neighborhood Tcl interpreter.

```
tclsh httpd.tcl
```

The default behavior is to host the htdocs/ directory adjacent to the **httpd.tcl** file as static content.

For those if you pining for the heady days of Tclhttpd, the stock Toadhttpd understands Tclhttpd style substitution files:

```
cat htdocs/hello.tml
[my html_header {Hello World!}]
Your Server is running.
<p>
The time is now [clock format [clock seconds]]
[my html_footer]
```

You'll note that inside some of the angle bracket is `[my]`. Yes, the substitution is being performed inside of the `httpd::reply` object. Your template has access to all of the reply's methods. If you need to refer to the server, it is delegated as the `<server>` method.Toadhttpd also maintains an sqlite based caching, dispatch, logging, and security system. That database handle is delegated as the `<db>` method. And just because we are running inside of an httpd::reply object doesn't mean it's too late to bolt on new behaviors!

```
[# Make this page regenerate every time
my reply set Cache-Control no-cache
# Change the style engine
my clay mixinmap style ::etoyoc::style
my html_header {Hello World!}]
Your Server is running.
<p>
The time is now [clock format [clock seconds]].
<p>
This page has been accessed [
#Calculate the page hits
set URI [my request get REQUEST_URI]
set count [my <db> onecolumn {
select count(rowid) from log.log where
REQUEST_URI=:URI}]
[my html_footer]
```

Another neat feature is being able to do your own conditional redirects:

```
[my html_header {My Happy File}]
[set URL hello
if {[my request get REQUEST_URI] eq $URL} return
my reply set LOCATION /$URL
my reply set Status 301
return "
The file your were looking for
[my request get REQUEST_URI]
has moved. You will be redirected to:
$URL momentarily"]
<p>
The time is now [clock format [clock seconds]]
[my html_footer]
```

And when you want to access your logs, you can do so via your favorite sqlite implementation:

```
> ls log/
cache.sqlite        log.sqlite
> sqlite3 log/log.sqlite
SQLite version 3.24.0 2018-06-04 19:24:41
Enter ".help" for usage hints.
sqlite> .tables
blackhole          debug              log
session
blackhole_journal  journal            log_info
session_info
sqlite> .schema log
CREATE TABLE log (
time  'UNIXTIME' DEFAULT (now()),
REMOTE_ADDR  'IPADDR',
REMOTE_HOST  'HOSTNAME',
REFERER  'URI',
USER_AGENT  'STRING',
HTTP_HOST  'HOSTNAME',
REQUEST_URI  'URI',
SESSION  'UUID' REFERENCES session
DEFAULT(guuid()),
COOKIE  'STRING',
rowid INTEGER PRIMARY KEY AUTOINCREMENT,
uuid STRING UNIQUE
);
sqlite> select * from log;
1537370493|127.0.0.1|127.0.0.1|http://localhost:
8015/|Mozilla/5.0 (Macintosh; Intel Mac OS X
10_13_6) AppleWebKit/605.1.15 (KHTML, like Gecko)
Version/12.0 Safari/605.1.15|localhost:8015|/hel-
lo.tml|||5|d71a0fb6-0208-43f5-b468-58c797b3c901
sqlite>
```

Configuring Toadhttpd

The server looks for a file name **config.tcl** adjacent to **httpd.tcl** as a place for the developer to add configuration and customization. The script is run inside of the `httpd::server` object's start method. The script can exercise the server's methods. Here is a snippet from my configuration for www.etoyoc.com:

```
# Manage configurable options
my clay set doc_ttl 900
my clay set server/ style ::etoyoc::style
set srvhere [file dirname \
   [file normalize [info script]]]
# Load plugins
package require toadhttpd::clique
package require toadhttpd::fossil
package require toadhttpd::trivia
package require toadhttpd::bootstrap
package require toadhttpd::facade

# Activate plugins
my plugin clique
my plugin bootstrap
my plugin facade

# I have elected to break my config file up
# into site based chunks
foreach directory [glob $srvhere/sites/*] {
  if {[file exists $directory/config.tcl]} {
    source $directory/config.tcl
  }
}
```

```
# If someone asks for /login, we know they
# probably meant my folk fest website's login
my uri add % /login {
  reply {content httpd::content.redirect}
  LOCATION /pff/login
}
```

Security Enhancements

Toadhttpd also includes a means to block nefarious IP addresses based on behavior. There is a class `toadhttpd::content.honeypot` which can be used for marking an IP address as a bad actor.

A little later in my file, I have rules to place IPs in the black list based on URL:

```
# Mark certain URIs to be caught by the security
# model's honeypot
my uri add % {
  /ccvv /admin% /test/wp-admin% /wp-login.php%
  /CGI/Execute /PhpMyAdmin% %.php /manager/html
  /wls-wsat%  /.DS_Store% /.git%
  /.hg%  /.idea%  /.ssh% /.well-known%
   %phpunit%  /sftp_config.%
} {
  mixin {reply toadhttpd::content.honeypot}
}
```

You'll note I'm using SQL style glob characters, because in Toadhttpd, the dispatcher uses the Sqlite like() function. If any of those URIs are encountered they will be dealt with by the following object:

```
::clay::define ::toadhttpd::content.honeypot {
  method content {} {
    my Blackhole {Security Honeypot}
    my puts {
<html><body><h1>You have been blocked</h1></
body></html>
    }
  }
}
```

The `Blackhole` method understands that the requesting IP address has done something nasty, and records a reason as to why. This event is logged in the **blackhole_journal** table.

Toadhttpd/Httpd Plugins

Both Toadhttpd and Httpd are extendable with the same plugin architecture. A plugin is expected to be a mixin class. The method to activate a plugin takes two arguments, the name of the slot and the name of the class (or classes) to load into that slot. If the class argument is left off, the server guesses a name of the pattern: `httpd::plugin.`*`slot`*`.`

On activation, the server object consults the class (via clay) to see how the plugin expects to interact with the server. The following slots in clay have meaning:

plugin/	dispatch	A script to be inserted into the server's *dispatch* method
plugin/	load	A script to be executed inside if the *plugin* method immediately during activation.
plugin/	headers	A script to insert into the server's *Headers_Process* method
plugin/	thread	A script to insert into the server's *Thread_start* method. Which is run during the *start* method. Intended to allow the plugin to kick off one or more worker threads.

Several example plugins are distributed with Tcllib. The simplest is a plugin to implement a dict based dispatcher:

```
# A rudimentary plugin that dispatches URLs from
# a dict data structure
::clay::define ::httpd::plugin.dict_dispatch {
  clay set plugin/ load {
    my variable url_patterns
    set url_patterns {}
  }
  clay set plugin/ dispatch {
    set reply [my Dispatch_Dict $data]
    if {[dict size $reply]} {
      return $reply
    }
  }
  # Implementation of the dispatcher
  method Dispatch_Dict {data} {
    my variable url_patterns
    set vhost [lindex [split \
      [dict get $data http HTTP_HOST] :] 0]
    set uri [dict get $data http REQUEST_PATH]
    foreach {host hostpat} $url_patterns {
     if {![string match $host $vhost]} continue
     foreach {pattern info} $hostpat {
      if {![string match $pattern $uri]} continue
      set buffer $data
      foreach {f v} $info {
        dict set buffer $f $v
      }
      return $buffer
     }
    }
    return {}
  }
}
```

```
::clay::define ::httpd::plugin.dict_dispatch {
  ###
  # Add the URI ensemble to allow outside
  # process to add URI's
  ###
  Ensemble uri::add {vhosts uris info} {
    my variable url_patterns
    foreach vhost $vhosts {
      foreach pattern $uris {
        set data $info
        if {![dict exists $data prefix]} {
           dict set data prefix \
             [my PrefixNormalize $pattern]
        }
        dict set url_patterns $vhost \
          [string trimleft $pattern /] $data
      }
    }
  }
  ###
  # Accept a body of a method as a source of
  # dynamic content, wrap that body in a new
  # class, and attach that class to the
  # given vhosts and uris
  ###
  Ensemble uri::direct {vhosts uris info body} {
    my variable url_patterns
    set cbody {}
    if {[dict exists $info superclass]} {
      append cbody \n \
       "superclass {*}[dict get $info superclass]"
      dict unset info superclass
    }
    append cbody \n [list method content {} $body]
    set class \
      [namespace current]::${vhosts}/${patterns}
    set class [string map {* %} $class]
    ::clay::define $class $cbody
    dict set info mixin content $class
    my uri add $vhosts $uris $info
  }
}
```

Using Httpd in your own program.

When I wrote the abstract I had grand visions of rewriting a major chunk of T&E's task dispatch system. And while that project is still on the books, it wasn't quite ready for prime time. I can however share how Httpd was used to serve up help files from within our software.

Essentially, we have a lot of documentation that is cooked up on the fly from data structures and other introspection tools. And being the lazy programmer I am, the simplest way to present that information is to have the machine crank out HTML on the fly. The process starts when a user clicks a menu item that triggers a command called `::docview::dvopen`. That command hunts around in the local OS for how to open a browser. And finally it creates an instance of httpd, and throws open it's port at a localhost URL that it can then ask the OS to display:

```
proc ::docview::dvopen {{page {}}} {
  variable _server_url
  ::docview::start_server
  if {[string range $page 0 6] eq "help://"} {
    set page [string range $page 7 end]
  }
  set url ${_server_url}/$page
  global tcl_platform
  switch $tcl_platform(os) {
    Darwin {
      set command [list open $url]
    }
    {Windows 95} -
    {Windows NT} {
      set command "[auto_execok start] {} [list
$url]"
    }
    default {
     # A lot of hunting around for name brand
     # browsers
    }
  }
  if [info exists command] {
    if [catch {eval exec $command} err] {
      irmMessageBox -icon error \
-message "error '$err' with '$command'"
    }
  }
  puts "Open a browser to $url if one hasn't
popped up already"
  ::docview::ruleIndex
}
```

The code to start the server is:

```
proc ::docview::start_server {{page {}}} {
  variable _server_port
  variable _server_url

  if {![info exists _server_port]} {
    set _server_port [::nettool::allocate_port
50050]
  }
  if {[info commands ::docview::listener] eq {}} {
   ::docview::server create ::docview::listener \
      port $_server_port \
      doc_root $::docview::docroot
      set _server_url \
"http://localhost:$_server_port"
  }
  puts "Listening on $_server_port"
  return $_server_port
}
```

And ::docview::server looks like:

```
::clay::define ::docview::server {
  superclass ::httpd::server

  method dispatch data {
    set reply $data
    dict set reply class ::docview::reply
    dict set reply docroot [my cget doc_root]
    return $reply
  }
}
```

And docview::reply looks like:

```
::clay::define ::docview::reply {
  superclass ::httpd::reply

  method content {} {
   set path     [my request get REQUEST_URI]
```

```
    if {$path in {{} index index.html index.htm}} {
      set path home
    }
    my puts [::docview::_direct_page $path {}]
  }
}
```

And `::docview::_direct_path` is a ~~nasty~~ interesting set of hacks that digs into the IRM data structures and bangs out raw HTML code. The same function also powers an embedded TkHtml based viewer we use. When we don't have to do things like copy and paste. Or actually read the text.

Just like in all of the examples, we have a server. That server has a dispatch method. That dispatch method pairs a request with a class that will respond to it. That response class generates HTML anyway it knows how.

Conclusions

My goal in this paper was to introduce you to the power of one of the newer modules in tcllib. I hope you find this paper useful, but more importantly I hope you find the module useful.

Cited Works

Cover and clip-art:
Celtic Stencil Designs CD-ROM and Book
Co Spinhoven
http://store.doverpublications.com/0486996786.html

Tip 328: Coroutines
Miguel Sofer & Neil Madden
https://core.tcl.tk/tips/doc/trunk/tip/328.md

RFC 2068: Hypertext Transfer Protocol -- HTTP/1.1
R. Fielding, J. Gettys, J. Mogul, H. Frysytk, T. Berners-Lee
https://tools.ietf.org/html/rfc2068

RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2)
M. Belshe, R. Peon, M. Thomson, Ed.
https://tools.ietf.org/html/rfc7540

TclHttpd Web Server,
Excerpted from Chapter 18 or Practical Programming in Tcl/Tk,
Brent Welch,
https://tcl.tk/software/tclhttpd/tclhttpd.pdf

Clay: A Minimalist Toolkit for Sculpting TclOO
Sean Woods
http://www.etoyoc.com/yoda/papers/tcl2018.Clay_Paper.pdf

Fossil Repositories

Clay:	http://fossil.etoyoc.com/fossil/clay https://chiselapp.com/user/hypnotoad/repository/clay
Taolib:	http://fossil.etoyoc.com/fossil/taolib https://chiselapp.com/user/hypnotoad/repository/taolib
Tcllib:	https://core.tcl-lang.org/tcllib
Toadhttpd:	http://fossil.etoyoc.com/fossil/toadhttpd https://chiselapp.com/user/hypnotoad/repository/toadhttpd