Tcl-On-Track
Website tools for TclHttpd

Clif Flynt
Noumena Corporation,
8888 Black Pine Ln,
Whitmore Lake, MI 48189,
http://www.noucorp.com
clif at noucorp dot com

October 20, 2014

Abstract

Tclhttpd is a powerful and versatile web server. But while Tclhttpd has many tools for making web pages, it has few tools for making web sites.

Tcl-On-Track is designed to make it easy to create websites. It includes drop-in packages for blogs, user management, announcing events, and shopping carts as well as a tool for customizing CSS files and collections of CSS recipes for common page layouts.

1 Introduction

In the early days of the web, just making a set of pages was considered pretty cool. Anyone familiar with vi or emacs could hack together a page, put in some href links between pages and presto, a website was born.

This changed quickly as people realized that a set of pages should have a consistent look and feel with similar heads, tails and display conventions. The initial solution of pasting identical HTML into each page to create a consistent look across pages got old quickly and developers looked for programmatic ways to generate the boilerplate sections of their pages.

CGI scripts and dynamic content addressed the issue of common heads, tails and generated content, but CGI scripts are slow and dynamic content tools weren't always easy to use.

Tclhttpd provides support for active content in files and programmatic generation of content with template (`.tml` files), Application Direct urls that map a distinct URL directly to a Tcl procedure and with the `Url_PrefixInstall` command that maps entire hierarchies to Tcl procedures.

The template file support allows developers to embed Tcl scripts within a web page by placing the script between square braces. This allows the developer include processing within a web page. The Application Direct and Domain Mapping tools allowed programmatic generation of html pages.

One problem with CGI scripts, template files and url-to-procedure mapping is that these techniques encourage the intermingling of HTML and code. This has led to many muddy website designs in which data processing and display formatting are hopelessly intertwined.

While the template and domain mapping constructs make the developer's life simpler by making it easy to construct web pages with common head and tail procedures, they don't provide the website developer with prebuilt layout tools or higher level functions like blogs or content management.

The Ars Digita project addressed the problem of higher-level tools under AOLServer by including many packages like storefronts, blogs, and photo archives with the Ars Digita distribution.

Unfortunately, the learning curves for AOLServer, Oracle and Ars Digita are each steep. Setting up an Ars Digita site is not for the faint of heart.

The follow-on system, OpenACS, reduces the entry cost by simplifying the packages and using postgres instead of Oracle, but it's still a mammoth project that's hard for a newcomer to pick up.

Ruby On Rails gave website designers a powerful set of tools for creating websites, it was designed for simplicity and low entry cost. It supports building a cookie-cutter web site easily and also supports expansion via custom Ruby code.

Tcl On Track merges the Rails and Ars Digita models by providing sample index.tml files for common layouts, a set of (mostly) independent modules that can be copied into the `custom` folder to provide features like announcements, blogs and user management, programmatic wrappers for common CSS patterns, and a tool for customizing CSS definition files.

The initial release of Tcl On Tracks uses a filesystem based content management system with an sqlite database to track registered users.

The rationale for using files instead of placing content in a database was two-fold:

- To make the package quicker to develop (I was under a time-crunch to develop websites).
- To make the system simple for an unskilled user (anyone can upload a text file to a folder).

Like Rails, Track was designed to implement a Model-View-Controller pattern for websites. The design calls for a set of models that accept data in some system-specific format and convert it into a standard format for other applications to further process. Thus, there are modules that can read PDF files, modules that recognize pure Tcl scripts, ones that process pure text and ones that accept pure HTML.

Each of these modules generates a list of plain-text items with no formatting information. The lists are designed to be passed to other scripts to have html formatting information added.

As with most real-world applications, the implementation isn't quite as dogmatic as it might be.

2 Components

Tcl on Tracks consists of these components:

- TclOO Model controllers that return data as lists.
- Tcl namespace commands to reformat lists of data as html snippets.
- Sample `index.tml` files to stitch together the digested snippets into common page layouts.
- Sample css definition files to implement common page layouts.
- Tclsh and Wish applications to generate and customize CSS definitions.

2.1 Models with TclOO

The Model level components of the MVC pattern are built using TclOO. Each class's methods return data in a consistent format. Since the data to be digested can come in many formats, the classes are implemented as a base class that provides methods to process semi-digested data and mixins that interact with different representations of data and return a semi-digested format that the base class can rework.

For example, the `announce` class, which displays short teasers with links to a full announcement looks somewhat like the following example. (This example is simplified from actual code to make the pattern more obvious.)

```
oo::class create announce {
  variable State
  variable teasers

#################################################################
#   constructor {args}--
#    Create Announce object
# Arguments
# args
#  dictionary with
#   def       Type of mixin to add
#   folder    Full path to toplevel folder of announcements
#   lines     Number of lines in a teaser
#   depth     How far down a file tree to search for announcements.
```

```
#   order     Display announcements oldest or newest first
#             May be -decr or incr for lsort order
#
# Results
#   New object is created.
#   Data repository may be accessed
#
  constructor {args} {
    variable State

      # lines - number of lines to show in teaser.

    array set State {
      lines 8
      order -decr
    }

    # Update state from args
    # ...

    # mix in specific data handler
    oo::objdefine [self] mixin Ann_$State(type)

    # Call mixin method to fill teaser array
    my updateTeasers
  }
}

# Mixin class for announcements that exist in PDF format

oo::class create Ann_pdf {
  method updateTeasers { } {
    variable State

    foreach fl [glob -nocomplain $State(folder)/*.pdf] {

        set txt [exec /usr/local/bin/pdftotext -l 1 $fl -]

        set count 0
        foreach l [split $txt \n] {
            lappend newText $l
    if {[incr count] > $State(lines)} {
        break
    }
        }
```

```
        set txt [join $newText \n]
        set teasers($fl) $txt
    }
  }

  method showPage {} {
    # upload PDF file
  }
}
```

As shown above, an object's creation call must include a `-def` option to mix in the proper raw data handlers.

In a website, the tclhttpd template files interact with the base class methods to get lists of values. The lists are either formatted within the .tml file or are passed to *view* procedures to be formatted into html.

The initial design called for *model* methods to return simple lists. As more complex pages were developed, it's became necessary to expand the return to lists of lists. With more real-world experience, it's becoming obvious that this technique is also limited. The next major revision of Tracks will have the models return digested data as a Tcl `dict`.

The Models included in the initial Tracks release are:

blogOO
: Blog pages with user comments, teaser overviews.

announceOO
: Show announcements with no user comments, teaser overviews.

itemOO
: More complex announcements with more items.

navOO
: Returns title/URL pairs for navigation bars.

2.2 Views created with procedures

A common goal of software engineers is to find repeated functionality and extract it into procedures. A web pages offer a rich field for finding repeated patterns to be extracted.

The `view` functions accept data in a known format and return html. These procedures don't require a class hierarchy, so namespaces are used to organize the procedures.

These procedures range in complexity from simple procedures that generate html snippets (similar to the `html` package in `tcllib`), to those that return complex html strings including generating customized HTML headers, page tops, and user registration forms with back-end validation and database interactions.

The pattern for the simple functions resembles

```
proc doStuff {data} {
  set rtn <SOMETAG>
  foreach item $data {
    append rtn "\n<OTHERTAG>[SomeProcess $item]"
  }
  append rtn "\n<CLOSETAG>"
  return $rtn
}
```

Another repeated pattern is the HTML header and page top. Every web page needs to construct a header. While the headers will be similar, you may not want them to be identical.

The `tracks::head` procedure is an expansion of the standard `html::head` procedure. It invokes `html::head` to create the base header using values provided with `html::headTag`, `html::meta` and others. It then merges values from the command line dictionary for CSS and title values, adds a DOCTYPE directive, and customized meta tags, and per-page CSS directives as defined in a configuration file.

The extra header meta tags are defined in a per-folder array that can be placed in the `.tml` file as shown in this code snippet:

```
namespace eval ::track {
  # Define a site name
  set trackState(siteName) "C. Flynt"

  # Customized meta tags by folder
  #
  array set meta {
    / {
      keywords {novels, stories, science fiction, historical, fantasy}
      description {c.flynt fiction author}
      author {C. Flynt}
      }
    Novels {
      keywords {novels, fiction, science fiction, historical, fantasy}
      description {Free fiction samples}
      author {C. Flynt}
      }
    Blog {
      keywords {blog, fiction, science fiction, historical, fantasy, writing, re
      description {Blog personal, how to,}
      author {C. Flynt}
    }
  }

  # Define extra head tags for folders
```

```
  array set headTags {
    / {
        {link rel="stylesheet" href="/noucorp.css"}
        {link rel="stylesheet" href="/cflynt.css"}
      }
  }
}
```

A non-repeated pattern that didn't need an Object Oriented approach is user management. The Tracks package includes registration, login and change password procedures that return a form which can be embedded within the developer's web-page framework.

The procedures implemented in the view methods include:

- Page Layout

  layout.tcl
  : Procedures to do basic page layout

  css.tcl
  : CSS recipes for common page widgets (box with header, two column, etc)

  formCSS.tcl
  : A template driven form processor that doesn't need cookies. Adapted from the form processor in `Practical Programming in Tcl/Tk`

  track.tcl
  : Support tools for the track package.

  redirectTo.tcl
  : a Direct_Url front end for showing text files

- User Management

  newuser.tcl
  : Uses formCSS.tcl to generate and validate a new user.

  changePwd.tcl
  : Uses formCSS.tcl to allow a user to change a password.

  login.tcl
  : Provides validate and process procedures for a login form.

- Data converters

  def.tcl
  : Reads content definition files (tcl scripts)

  file.tcl
  : Reads pure text content files

  folders.tcl
  : Navigates folder based hierarchy.

html8.tcl
: Converts text to html, adding paragraph markers for blank lines, etc.

tagTweak.tcl
: Balances tags when extracting subsets of pages as teasers/blurbs.

html.tcl
: Extended html support for Style and generating tables and lists.

debugPg.tcl
: Tools to assist in debugging pages.

2.3 Tools

Chrome and Firefox provide tools to help a web designer tweak CSS. The ability to enter some CSS and immediately see the effect makes understanding the tricky interactions faster.

Even with these tools, hand edited CSS files get cumbersome very quickly.

The tracks package includes two tools for generating and tweaking CSS files. These applications are:

makeCSSTemplate.tcl
: Scans .tml files and generates a css template with Tcl variables linked to the CSS classes.

genCSS.tcl
: GUI to set Tcl variable values and generate a CSS file.

The `genCSS.tcl` application is shown below. It maps Tcl variables to various `div.`*something* and `span.`*something* constructs and view what the settings will be.

The `Patterns` window lets the user set patterns of variable names to select and a value to assign to elements that match that pattern. When assigning values to variables the patterns are read from top to bottom, and each match is assigned the associated value. Thus the more generic patterns (`*Background`) come first and more specific patterns (`NavbarBackground`) come later.

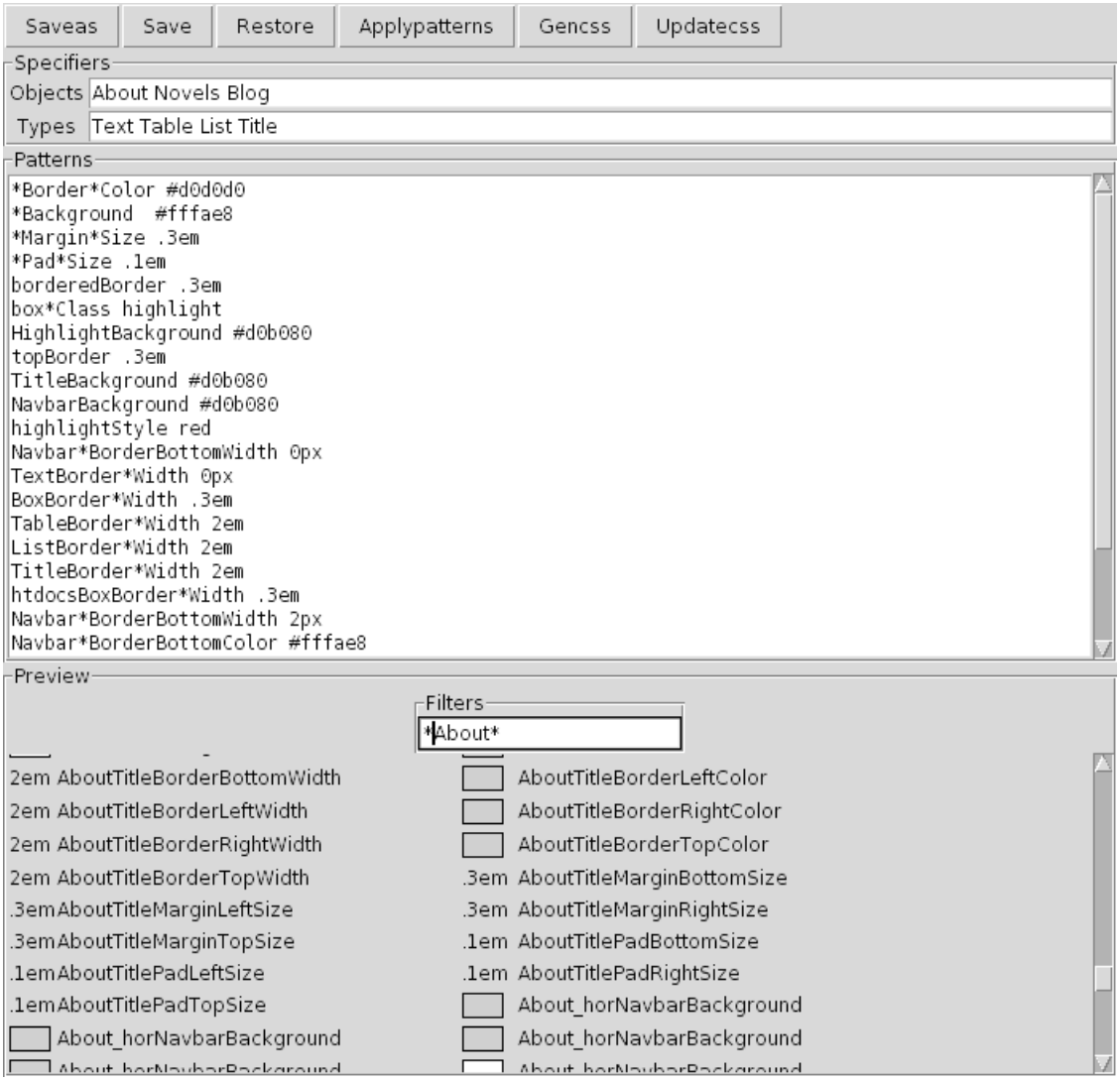

Figure 1: http://www.cflynt.com/

These two tools were used for some sets of web pages and proved easier to use than hand-editing the CSS files, but the underlying concepts are too simplistic and don't make good use of CSS hierarchies.

I don't expect to distribute these tools with the initial Tracks release.

3 Use

Using Tracks is simple:

- Decide on the layout you want.
- Copy the appropriate index.tml file to your folder.
- Cut/Paste appropriate code into the `.tml` file.
- Customize index.tml and or .tml for your site (set name, title, etc)
- Add content files to website
- Tweak CSS

The `index.html` page for the `www.cflynt.com` site is shown below.

The folder name is assigned to the variable `id`, which is also used to the name the object that will process the content files in this folder and return digested data to the web pages.

This technique for generating the object name allows the `index.tml` file to be used in any folder without modifying that section of the page code. The downside is that it allows a potential name conflict if two folders in the website hierarchy have the same name. If there are folder name collisions, the user will need to edit the file and set `id` to a unique value.

```
[folders::top -title "$track::trackState(siteName)"]
[Doc_Dynamic]
<div class="scroll">

[
set lst {}
set id [string trim [file tail [file dirname $page(filename)]] /]

if {[info command $id] eq ""} {
  announce create $id -folder [Doc_Root] -type text -depth 1
}
foreach {in trio} [$id getTeasers] {
  lappend lst [lindex $trio 1] [lindex $trio 2] $id,$in
}
css::showBoxes $id /MkPg SPid $lst

]
</div>
</body>
</html>
```

which generates this page:

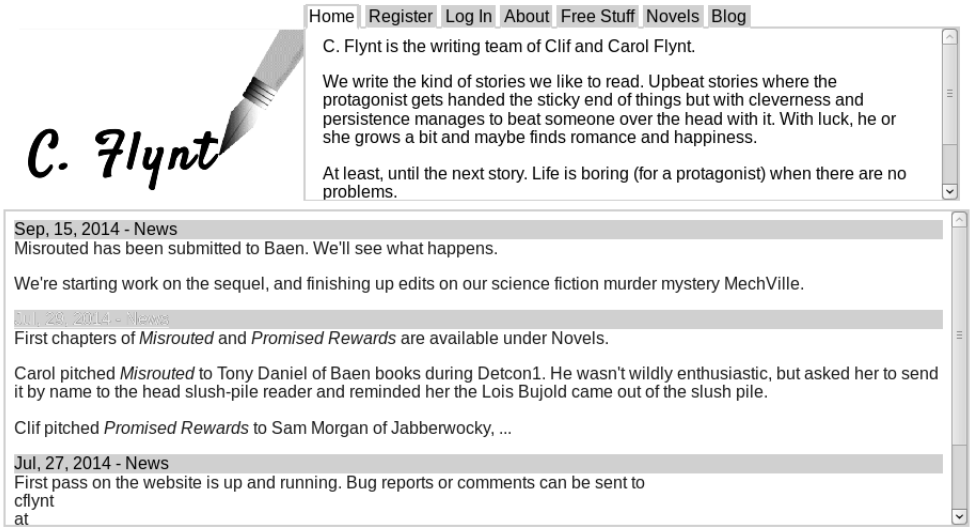

Figure 2: http://www.cflynt.com/

The home folder for this site contains:

- Folders for Navbar

  About
  : Contains bio information, about the company, contact info, etc.

  Blog
  : A set of folders with blog entries and comments

  Free_Stuff
  : A set of folders with free short stories

  Novels
  : A set of folders with novel teasers

  imgs
  : Common images referenced by pages (logo, etc)

- MainPage Contents

  20140727-News.txt, 20140729-News.txt ...
  : Announcements for the main page body. File name is massaged for box title.

  abstract.htm
  : An abstract for the top header.

  teaser.tcl
  : Short blurbs for news items. Automatically generated.

- Pages immediately available

  index.tml
  : Home page.

  changePassword.tml
  : Change password form.

  login.tml
  : Login form.

  newuser.tml
  : Registration form.

  welcome.tml
  : Welcome after registration page.

- Website administrivia

  cflynt.css
  : The CSS file.

  favicon.ico
  : The icon file.

  robots.txt
  : Instructions for Robots.

  notfound.tml
  : Error page for 404 errors.

The `folders::top` command creates the top of the web page and the page header. It scans the existing folders to see if there is information that should be included in the top from subordinate folders (in this case, there is none).

4 High Level HTML and CSS Constructs

A web page of average complexity has a great deal of text devoted to page layout.

One of the goals of the Tcl On Tracks project is to reduce the amount of typing that the web developer needs to do. There are many common form, table and css constructs that can be hidden in a procedure to generate the more complex html.

A simple example is to take two sets of html text and create two columns. The procedure to handle this is shown here:

```
proc 2ColElements {col1Html col2Html } {
    return [subst {
<div class="grid2col">
  <div class="col2Left">
    $col1Html
```

```
  </div>
  <div class="col2Right">
    $col2Html
  </div>
</div>
    }]
  }
```

The web page snippet would resemble this:

```
# Initialize the lists to avoid duplicating the page
# when it is reloaded.
set lst1 {}; set lst2 {}

# $id is an object created for the current folder
# getTeasers returns an id and the target URL, title and text
# to display as a list triplet of data.
#
foreach {in1 trio1 in2 trio2} [$id getTeasers] {
  lappend lst1 [lindex $trio1 1] [lindex $trio1 2] $id,$in1
  lappend lst2 [lindex $trio2 1] [lindex $trio2 2] $id,$in2
}

# Convert the lists to a set of html displaying headers
# as an HREF and then the text.  The sections of each box
# are in a CSS div.
set html1 [css::showBoxes $id showAnnounce.htm BLid $lst1]
set html2 [css::showBoxes $id showAnnounce.htm BLid $lst2]

# Make a two column display
css::2ColElements $html1 $html2
```

The `formCSS.tcl` package provides a more complex reformatting of data. This package has been used by the `www.tcl.tk/community/tcl20xx` registration pages for over a decade. It is an expansion on the forms program described by Brent Welch in *Practical Programming in Tcl/Tk*.

The form is generated and generic processing is done with the `Form_MultiPostProcess` procedure.

```
#############################################################
# proc formCSS::Form_MultiPostProcess {id fields nextPage}--
#    Process a form
# Arguments
# id            The identifier for this form
# fields        A list of elements to display in the form
#   fields format
#    {type   req key {text/selections} default}
```

```
#     input  1/0 key Label-beside     {}
#     select 1/0 key Label-beside     {choice list}
#     submit {}  key {text}           defaultP
#     break  {}  {}  {}               {}
# nextPage      The next page for this mess, relative or full URL
#
# Results
#   If first pass, new html page is defined and returned.
#   If second pass, $id.validate is invoked
#    if valid form data,  $id.calculate or $id.process
#    may be called to process data.
#
```

The somewhat cryptic description of the fields list is just a long list with the type of data to receive, followed by a boolean for whether or not it's required, the HTML name to use for the form value, the prompt and a possible default value.

Selection elements are handled slightly differently with the last value being pairs of label/values.

```
set formDef {
  textInput 1 email "Email:" {}
  select    1 mailinglist "Join Mailing List?" { Yes 1 No 0}
}
```

An added feature of the `formCSS` package is support for user-defined validation and form processing procedures. These are registered before the form is created with commands like the command shown below.

In the example, the `id` declared for the form would be `newuser`. This command registers the `newuser::validate` procedure with the form. When a browser submits a form, the form is checked to confirm that all required fields have data, and then the validation procedure is invoked. If that returns true, then the the `process` procedure is invoked.

```
  formCSS::setProc newuser.validate ::newuser::validate 1
  formCSS::setProc newuser.process ::newuser::addUser 1
```

5 Design Constraints

The `tclhttpd` engine provides several very powerful tools for building web pages.

It also provides ways to muddy the separation between data processing and data presentation.

The ability to put Tcl scripts into a `index.tml` file is a powerful tool for allowing the user to tweak data while generating the html code. It also provides

a hair-trigger weapon for shooting yourself in the foot by doing processing in the web page.

The Direct URL facility is much cleaner than CGI scripting, but as with putting data processing code in a `index.tml` file, there is a strong temptation to put html directives in a Direct URL or new Document Direct procedure, muddying the separation between data and presentation.

The primary design goal of Tcl On Track has been to keep the Model-View-Controller separation truly separate, with all Model code in procedures, and all presentation code in the .tml files.

The mud in this has been the desire to simplify verbose HTML and CSS constructs like forms and page layout.

The Tracks library has procedures that accept data in pre-digested formats and then returns a large amount of formatted HTML code. These procedures don't have any understanding of the page or data contents, they just take a list and return HTML or CSS.

6 Future

Every implementation of a design demonstrates the flaws of the design and suggests features for the next, better design.

The creator of Ruby on Rails mentioned in one writeup that the Rails project came after he wrote several other web frameworks.

The current version of Tcl On Track is the second major revision and rework. As such it has fewer (or at least different) bad choices from the first functional design which was adequate for creating slightly complex websites.

The next rework will focus on codifying rules for what belongs in a `index.tml` file vs what functionality belongs in the model and view procedures.

While the current CSS tools are functional, the CSS definition and editing application generates very verbose CSS scripts. They don't even use the simple and obvious techniques for simplifying a CSS definition.

The sample pages and view functions need to be reworked to make better use of the cascading nature of CSS. The current pages are naive in their CSS usage.

Tcl-On-Track has not actually been released, but when a codeset escapes it will be made available at `http://www.noucorp.com`.