***A T*cl *W*eb *F*ramework**

(*A*rnulf's *T*cl *W*eb *F*ramework)

ATWF

Abstract

ATWF a [Tcl] web framework is a work in progress, which started about 3 years ago with a brain storming wiki page [ToW Tcl on WebFreeWay]. The ideas collected at that page with a lot of feedback from the community were the base for further investigation. Inspired from a payed work project using [PHP]'s [Zend framework] I decided to convert as a starting point the basic parts of that framework nearly 1:1 into [Tcl]. The very first pre-alpha version is now handling the first requests. It has some adaption to [Tcl] constructs, but the rest is nearly identical to the original version.

Contact Information

Arnulf P. Wiedemann

Email: arnulf@wiedemann-pri.de

Tcl Wiki: http://wiki.tcl.tk/apw

Germany Prittriching

1 History (Collecting Ideas)

In May 2007 I started a wiki page named [ToW Tcl on WebFreeWay] inspired through the (comp.lang.tcl) discussion on "Tcl on Rails". I had some thoughts on how to make that (or at least a part of it) happen.

As I was thinking about some Tcl tools to help me create some personal web pages, I thought that could be the starting point of going deeper in that area. Knowing nothing about "[Ruby] on Rails" [RoR] at that time, I had some contact with it through downloading it and looking for some of the features.

My idea was to build something using [Tcl]/[Itcl]/[TclXML]/[TclXslt] and [starkit]s for use with [Apache] 1.3 and [Rivet] or [tclhttpd] or [wub].

It was not my intention to just clone [RoR], but to build some [Tcl]-ish framework and get a little bit inspired from [RoR].

I was asking for features which are useful. My direction at that time was to build some generators to make it easy to build at first simple web pages and later on web pages with more comfort.

My Idea was to have a [XML] structure (maybe similar to a [XML]-Schema specification) and to use that to generate [XSLT] templates, which then can be filled. But that is an additional feature besides "normal" [HTML] files. I was also thinking about a little tool which helps you defining [CSS] files. Another tool would be a simple generator, which uses a [XML], [XSLT] and [CSS] file and generates the [HTML] file (that's very easy using the tools already there, but for beginners that might be hard). I also thought to have as one of the databases a [starkit]/meta DB and I thought some of the tools or all should be in a [starkit] to make it easy to install and start. There should be also included one of the [Tcl] [http] servers to be able to generate a small application to begin with minimal effort.

During the discussion I decided to use [MVC] (model/view/controller) concepts and to have additional layers below model, view and controller and to allow template [HTML] forms which are "preprocessed" by [Tcl] (on the server side) using for example the following syntax:

<% some code %>

or:

<%TCL some code %>

and to let [Tcl] build the final [HTML] output on the fly when called via some cgi-bin script interpreting the above parts within <% … %>.

I was also planning to have a TK GUI which represents the file system structure of the views, templates etc. as a tree so you can navigate and open the files in a text widget or with your favorite text editor. But in case you only wanted to maintain the structure using the GUI, it should be possible to directly edit the files (containing for example the templates) within the file system.

I wanted to put the commands/application for maintaining the structure (the scaffolding) into a [starkit], to make it '''very easy''' to start with the tool, just download the kit file, open a new app and start building the structure and filling in the templates.

After having had a more detailed look at [symfony] [http://www.symfony-project.com/], [propel] [http://propel.phpdb.org/trac] and phing [http://phing.info/trac/] (the [PHP] analog on to [RoR]), I did like some of the stuff they provided for example having [XML] files for storing all the information and for automatically having the possibility to validate all input.

I didn't like that the application(s) was/(were) too big and on some places too complicated, but I thought, if one would strip down that and also have a look at [RoR], that would be the right direction to go.

After having a short look at [AOLserver] ADP, I found that the package had too many hard wired features, which could make it hard for programmers to be flexible.

[Wtcl] used to many calls in my opinion (but that is a question of taste :-))). I wanted to get a generator, which makes the "low level stuff" and produces files which can be modified, if one is not satisfied with the generated stuff.

By the way, the ideas for templating for [RoR] and [symfony] are after all very similar, I think both have been influenced by the [Apache] Torque project (for [symfony] I know it).

I had a closer look at [YAML]. Nevertheless I was thinking to use a GUI to generate the [XML] files with minimal input from a user and only if you wanted to make special things you would modify and not create the [XML] configuration files.

At that time I thought, one advantage is, that there exist a lot of [Tcl] tools to work with [XML] files and using [XSLT] makes it easy to convert it and using [XSD]-SCHEMA allows you to verify input easily. Not that I was fixed to [XML] but at least there was no [Tcl] Parser for [YAML] and the existing C-parser is not easily integrated to [Tcl]. I have looked at it and it is not easy (not difficult but a lot of work) to write a [Tcl] native parser for [YAML].

2 Template Formats

At that point there was some discussion about the format of the templates to build the final [HTML] or [XML] pages.

One of the discussed formats looked like:

```
<!-- Begin of document --->
<h1><%= @page.title %></h1>
<p class="pageDescription"><%= @page.description %></p>
<hr>
<div id="pageText"><%= @page.text %></div>
<p class="pageAuthor"><%= @author.name %> (<a href="mailtto:<%= @author.email %>"><%= @author.email %></a>)</p>
<!-- comments
<table border="0" width="300">
  <tr>
     <th bgcolor="green">Comments (<a href="<%= :SCRIPT_NAME %>?action=add">add a comment</a>)</th>
  </tr>
  <% foreach |entry| @comments %>
     <tr bgcolor="<%= cycle values="#dedede,#eeeeee" advance=false %>"><td>
        posted by <%= escape(@comments.author{entry] => :name) %> on <%= date_format("%e %b, %Y", @comments[entry] => :date) %><br />
        <%= escape(@comments[entry] => text) %>
     </td></tr>
  <% foreachelse %>
     <tr><td>No records</td></tr>
  <% end %>
</table>
<!-- End of document --->
```

After all the discussion my conclusion was, the ideas of [symfony], [RoR], [Smarty] seem to be relativly similar in using the [MVC] concept, having an abstraction from the data source, using a generator to convert an input template in some format be it [HTML], [XML] whatsoever with spread in [Tcl] code to generate (at least for the 3 systems mentioned) [HTML] output.

In [RoR] they have something like "abstract" calls to the data sources with the [CRUD] (Create, Read, Update and Delete) layer, which is differently implemented for each data source for example by having the calls as abstract virtual calls in a base class, which must be implemented by derived classes.

My preference was the spread in code should not only be able to handle one static input of the data source, but to be able to make additional requests to the data source for getting additional input to handle. The data source should be accessed via the abstract [CRUD] interface (shortly described above), the generators should be able to handle different kinds of templates (i.e. [HTML], [XML] ...). There should be controllers, which handle the requests (actions) in calling generators for producing the output. The output is driven by the use of templates (the views) which describe how the output has to be formatted and by use of [CRUD] requests for getting the appropriate data.

As [HTML] and [XML] have both the < ... > tags there is no problem in parsing the stuff, if no validation of the semantic is done, the validation could be done by some different program which could be added to the tool later. For other stuff [WAP], [Postscript], [TeX] etc. one would need a closer look to see, if that is also possible with the same generators or with different ones (at least parsers for the input).

Then I had the idea of doing the templating the other way around and let the template be some [Tcl] code which is called like a preprocessor and produces output (for example a [HTML] page):

An example could then look like:

```
<%= {
 <!-- Begin of document --->
 <h1>} ; <%= {[@page getTitle]} ; <%= {</h1>}
<%= {<p class="pageDescription">} ; <%= {[@page getDescription]} ; <%= {</p>}
<%= {<hr>
 <div id="pageText">} ; <%= {[@page geText]} ; <%= {</div>}
 <%= {<p class="pageAuthor">} ; <%= {[@author getNname]} ; <%= {(<a href="mailtto:}
   <%= {[@author getEmail]} ; <%= {">}
<%= {[@author getEmail]} ; <%= {</a>)</p>}
<%= {<!-- comments -->
 <table border="0" width="300">
  <tr>
      <th bgcolor="green">Comments (<a href="} ; <%= {[set SCRIPT_NAME]}
       <%= {?action=add">add a comment</a>)</th>
  </tr>}
set rows [@comments getRows]
foreach entry $rows {
 <%= {<tr bgcolor="}
   <%= {[cycle values "#dedede,#eeeeee" advance false]}
   <%= {"><td>
         posted by} ; <%= {[escape [[[@comments getEntry] getAuthor] getNname]]}
   <%= { on } ; <%= {[clock format [clock scan [[@comments getEntry] getDate] -format "%e %b, %Y"]}
   <%= {<br />}
   <%= {[escape [[@comments getEntry] getText]]
   <%= {</td>
  </tr>}
}
if {[llength $rows] == 0} {
  <%= { <tr><td>No records</td></tr>}
}
<%= {</table>
 <!-- End of document --->}
```

“@page”: the @ marks it as a "page" object (from the model of that application).

“getText”, “getAuthor” etc. are methods of that object, which return the field value of the appropriate field of the current row of the object.

“<%=” is a proc which is returning its argument "<%= {hallo}" returns "hallo" or if it starts with "[" is evaluating the argument "<%= {[@page getTitle]}" returns the title field contents of the current row of the page object.

The “escape” proc would do the [HTML] escaping.

The generator knows what to do with the results (for example it just outputs them to the client using the result as a [HTML] page).

During that discussion there had been a very interesting suggestion from Jean-Claude Wippler [jcw] [Templates and subst] called “substify”. I was sure that would be a good approach for a template syntax (and an implementation was already there).

With that a solution for templating was found, so the next point was a database abstraction layer. For that I planned to use [nstcl]. After a closer look at it, it seemed to have all the features needed. Nevertheless an additional layer would be needed to separate object relations from database table relations.

3 Implementation Start

After that discussion, which ended in May 2007, there was no progress until December 2008 because I was working on 3 other Open Source projects [itcl-ng], [ntkwidget] and [ReportingTools]. And only a few small parts of code had been implemented at that time.

Finally in April 2010 I got back to that ideas and started a project called [ATWF] (A Tcl Web Framework).

As implementation language I did choose [itcl-ng], for the Db layer I did choose [tdbc] from Kevin Kenny [kbk] to be able to connect to/working with databases.

Then in payed work we were looking for a web framework and [Tcl] was not an option because of not enough people in the project environment were familiar with [Tcl].

So we choose [PHP]'s [Zend framework], as the tool, as one colleague had already experience with it in other projects and others did at least know [PHP]. Looking at that, a lot of features seemed to be very interesting i.e. [MVC] concepts, possibility to overwrite all the defaults provided by the framework, big flexibility in processing a request etc.

I started building some classes for the DB abstraction layer based on [tdbc] using ideas from [PHP] [Zend framework]

I first had the idea of only using some ideas from [Zend framework] and building my own framework completely from ground up. But very soon I noticed that I was adding very similar code to my version from the [Zend framework] code and on the other side there were already a lot of interesting features for driving the handling of a request ready in [Zend framework].

[PHP] code with the use of [PHP] class system (on which [Zend framework] is based) and objects looked very similar to [Tcl] code when using [Itcl] as object system. So I finally decided to use the original code and to convert it very close to the original into [Tcl] Code. By doing that I would be able to use all the features, [Zend framework] had already tested and decided to be useful during their development phase.

I have reimplemented a big part of Db package of the [PHP] [Zend Framework] in [Tcl] within 2 weeks by replacing the [PDO] layer used in [Zend framework] by [tdbc]. Simple selects/inserts/updates/deletes were possible with that, but there was still a lot of work in debugging and testing that part.

In ATWF the identical syntax for the DB interfaces is used as in [Zend Framework]. This is mostly the '''model''' part of [MVC].

After that I implemented a big part of the controller layer also by reimplementing the equivalent sources from [PHP] [Zend Framework] in [Tcl]. By doing that, a bigger part of '''controller''' of [MVC] concepts was done (needs still a lot of testing) so the missing part of [MVC] was the '''view''' part.

For generating web pages, I decided to use [Templates and subst] the “substify” proc from Jean-Claude Wippler [jcw] (this was a part of the '''view''' part of [MVC]).

The first version of the '''view''' part was done in 2nd half of April 2010. The first request using apache2 and cgi-bin was routed and executed and produced a [HTML] page with data from a mysql database, as expected.

Now there were about 30.000 lines of [Zend Framework] converted to [Tcl]. The total number of lines of [Zend Framework] (of the used version 9.3) is about 140.000 lines and about 27.000 lines thereof are targeted to [PDF] handling.

Implementation continued and at beginning of May 2010 there were about 55.000 lines rewritten and the first web pages using apache 2.2.15 and rivet 2.0.0 were running including context switch between [HTML] and [XML] output depending on a “format=...” parameter.

The [HTML] respective [XML] page context is generated using [Templates and subst], which means there is a .html or .xml template with inserted [Tcl] statements and the “substify” proc is generating the real page to be sent to the client.

4 Differences in the Implementation

There were only 4 places, where I have used a different approach to [Zend framework] for implementation.

1. *Database adapters*: as I just had experimented with Kevin Kenny's [kbk] [tdbc] I found that [tdbc] is a suitable replacement for the [PDO] DB adapters of [Zend framework].
2. *Arrays*: all the [Zend framework] classes are derived from an array class and nearly all collections of variables are based on [PHP] arrays. I found the equivalent in [Tcl] are [dicts], which Donal Fellows [dkf] has provided. And as most of the array uses in [Zend framework] are key/value pairs, [dicts] are optimal for that part,
3. *Views*: Generating [HTML] or [XML] code using [PHP] code parser in the view part, which could also handle [HTML]/[XML] code intermixed with escaped [PHP] code was not possible for the equivalent [Tcl] part. But for that I had already a replacement the „substify“ procedure from Jean-Claude Wippler [jcw]. It was easy to replace that part of the view mechanism of [Zend framework] with the „substify“ proc call and some additonal code in ATWF.
4. *Namespaces*: [Zend framework] uses a 1:1 relationship between directory structure and class name architecture. Class names are built from directory path names by replacing the path delimiter (normally a „/“ character) by an underscore („_“) character. That inhibits the use of underscore characters within class name parts. As [Tcl] has already a [namespace] feature I replaced the generation of class names with underscores for the directory parts by using of the [Tcl] [namespace] „::“ syntax and thereby the normal syntax for [Tcl] [namespace]s. Using that approach, I think the idea of the namespace philosophy is much more visible in the ATWF implementation.

5 General Structure of an Application

ATWF uses - as does the [Zend framework] - the [MVC] (Model/View/Controller) technology. This can also be seen in the directory structure used for an application/ project.

A Project in [Zend framework] has the following general layout in the file system:

```
application
        configs
        controllers
        models
        layouts
        views
        modules
                module1
                        controllers
                        models
                        layouts
                        views
                module2
                        controllers
                        models
                        layouts
                        views
                :
                :
public
```

6 Building of URLs for the Requests

URL's for sending a request are built as follows:

....../<module name>/<controller name>/<action name>?<parameters>

- <module name> the name of the module can be empty
- <controller name> the name of the controller (default name index)
- <action name> the name of the action (default name index)

7 Contents of the Application Directories

A controller has a corresponding [Tcl] script in the *controllers* directory with the same name and in that script for every *action* there must be a method with the name <action name>Action i.e. IndexAction for the index action.

The *models* directory normally has a *DbTable* subdirectory and in that directory a DbTable class for every table in the DB used by the project.

The name for a DB Table person is "::Model::Dbtable::Person", if no module name is used, otherwise the module name is prepended as namespace in front of that name "::Module1::Model::Dbtable::Person"

The *views* directory normally contains a scripts directory, which contains a directory for every controller with the name of that controller and in there a [Tcl] script for every action for that controller. These scripts are either templates for a [HTML] page or for a [XML] page with mixed in [Tcl] code, which is executed and replaced when applying the "substify" procedure on that template.

The *layouts* directory normally contains .css etc. files, which drive the layout of a page, so the contents in the views directory and layout in the layouts directory are separated.

8 Bootstrapping

The bootstrapping starts with an index.ttml file, which bootstraps the application. The following steps are performed:

1. scanning of the application.ini file
2. calling of setOptions (which is:)
3. initializing standard resources:
    - Db
    - FrontController
    - Layout
    - Locale
    - Modules
    - Navigation
    - Resource
    - Router
    - Session
    - Translate
    - View
4. For Application::Resource::Resourcebase setBootstrap is called (always)
5. call FrontController for handling of the request

9 Work Flow for a Request

The general work flow is: the *controller* calls an action, the *action* uses the *models* and eventually gets data from the DB using the “DbTable” class for the needed table(s). It processes the result sets and hands the result over to the *view* part. Then the *view* part is called running the rendering code and producing the wanted page and during that part calling the „substify“ procedure for getting the processed [HTML]/[XML] template and producing the *response*. The *dispatcher*, which is driving the controller/action/view part, is then sending the response back to the requester.

The dispatcher is part of FrontController and does the following steps:

- setRequest
- setBaseUrl
- setResponse
- for every plugin call method setRequest
- for every plugin call method setResponse
- initialize router
- router setParams
- initialize dispatcher
- dispatcher setParams
- dispatcher setResponse
- while not dispatched
    - for every plugin call method routeStartuo
    - router route
    - for every plugin call method routeShutdown
    - for every plugin call method dispatchLoopStartup
    - while to be dispatched
        - setDispatched true
        - for every plugin call method preDispatch
        - if request is not completely dispatched continue
        - dispatcher dispatch request
        - notify plugins of dispatch completion in calling postDispatch
        - if request is dispatched break
    - end of to be dispatched
    - for every plugin call method dispatchLoopShutdown
    - if returnResponse return the response
    - else: response sendResponse

When the dispatcher is dispatching a request, it determines the controller and the action from the request using defaults for module, controller and action, if the request has no values for these.

In the action method within a controller the output to be rendered is passed to view object.

If not inhibited by the controller, the view object is called and executes method render which produces the rendered output by also calling “substify” proc on the way and stores the output in the response object.

When calling sendResponse that output is passed back to the sender of the request.

10 Current Status

Right now the first requests sent from an [apache] server using [rivet] for forwarding a request to [ATWF] are running and giving back the requested results. An earlier version was also handling [wub] from Colin McCormack [CMcC] requests, this has to be adapted for the current version again. And I also had a version handling cgi-bin requests, to be adapted too.

Unfortunately there is nearly no demo available at the moment.

11 Todos

There are the following items on the todo list:

- Test suite
- Complete integration into the 3 server types: Apache, wub, cgi-bin
- Prepare alpha version
- Demos
- Documentation
- Look for parts which can be made more Tcl'ish
- Decide which other parts of [Zend framework] are needed/interesting to take over
- Take care of feedback to the alpha version