A State-Driven, Service-Oriented Dynamic Web Development Framework

Stephen E. Huntley
stephen.huntley@alum.mit.edu

Abstract

I introduce a Tcl-based framework for development of dynamic web sites. It was written as an attempt to get away from object-oriented web development paradigms which have largely failed to deliver the advantages that dynamic scripting tools ought to provide. It leverages some of Tcl's unique features, such as hierarchical namespaces, module-based package management and extensive call stack introspection; in order to provide a hierarchy of composable dynamic services in response to a URI, inspired by service-oriented programming techniques.

1. Introduction

The subject of this paper is a collection of Tcl packages designed to work together as a simple, modular web application framework; specifically it is designed to bring to the task of authoring and delivering a dynamic web site the relative ease, flexibility, reliability and empowerment associated (by those who know it well) with development of a Tcl software program.

The last decade has seen a tremendous increase in complexity of programming techniques used in the most popular web application frameworks. These include heavy reliance on object-oriented programming, model-view-controller design patterns and object-relational mapping for database access. The Ruby on Rails framework is probably the most high-profile embodiment of these trends, with packages such as Spring and Hibernate in Java and Zend and CodeIgniter in PHP taking similar approaches and reaching similar levels of adoption and mindshare.

The appeal of such products is the promise of rapid prototyping and fast delivery of database-driven web sites achievable by designers with relatively little experience.

The design approaches and associated hype typified by these frameworks have become so widespread and ingrained in programming culture that I have encountered younger programmers who have expressed surprise that other means of designing a dynamic web site are even conceivable.

The drastic downside of these trends in web site engineering is that the promise of quick prototyping and fast early results has been bought with introduction of intractable medium-to-long term issues of code management, stability, security and scalability. Attempts to fix these issues have led to increases in the complexity of the frameworks themselves, often thereby leaving behind the inexperienced programmers the frameworks were originally intended to appeal to. Dynamic web development has come to resemble the dispiriting slog of enterprise architecture programming, rather than the quick and satisfying scripting experience that programmers turned to the lighter-weight, non-compiled languages for in the first place.

As a capper to these struggles, which have been simmering for some time, in the beginning of 2013 a series of catastrophic security flaws were found in the software of the Ruby on Rails framework[1]. These bugs make total takeover of web servers running the software trivially easy for attackers. The revealed flaws were so deeply entwined in the code that they appeared to some expert observers to herald a whole new class of bugs; where, even if the immediately known issues were patched, it might take years to track down the underlying vulnerabilities, if they do not in fact turn out to be inherent in the object-oriented, object-relational architectural approach (and thus ultimately ineradicable).

All this is by way of saying that the past ten years or so of web authoring progress is today looking like something of an evolutionary dead end. A total rethink of how to design small to medium-sized dynamic websites, which might have seemed like an indulgence just a year ago, now importunes more like a matter of urgent importance.

This program is a serendipitous result of attempts to solve a number of pressing web design and delivery problems encountered by me professionally over the past several years. These problems relate to issues of code deployment, identity management, role-based access to resources, workflow and collaboration. Despite the intensifying complexity of the mainstream application frameworks, the state of the art in grappling with these issues has hardly budged in the Twenty-first Century.

The code attempts to leverage Tcl's natural strengths: its hierarchical namespaces, its strong package management features, its string-oriented data processing paradigm, its event loop and its call stack introspection features, all map exceptionally well to the problem spaces presented by the HTTP protocol and its stateless, hierarchically-indexed, string-based orientation. Thus the program is able to pursue the goal of mapping the design of a dynamic web site to the well-understood task of writing a Tcl program as encapsulated in a collection of packages.

As a result of its serendipitous creation, the program is something of a mutant hybrid; I hope that like some other mutant hybrids, it will prove fitter for survival in the wild than other more purebred species.

2. Design Approaches: Service-orientation and Hierarchical Indexing

The code relies on a simple central design axiom: the URI of a request from an external client to a web server can be used internally by the server as a master index to control access to the range of heterogeneous information sources a web framework is today expected to access and coordinate to fulfill the external request.

The simplest example is mapping of a HTTP query's URI to a Tcl procedure, which is programmed to return the desired dynamic response. In this instance comparison may be made to Tclhttpd's “application direct URL” feature[2]. This feature allows the developer to register URIs as keys and procedure names as values in a global array. On receipt of a query with a matching URI, the correspondingly mapped procedure is called. Tclhttpd additionally decodes the query's named values and calls the procedure with the values mapped to the appropriate arguments.

This is a handy feature, but the mappings of URI to code are arbitrary, and no structure is provided for organization or marshaling of the procedure code within the project.

The new code does a similar mapping of URIs to procedures, but does so without the need for maintenance of any key-value arrays; there is in fact no need for any metadata superstructure at all -- it instead simply leverages Tcl's hierarchical namespace feature and maps the nodes of the URI to a namespace path, which is presumed to exist or be loadable from a package module. Like Tclhttpd, it also decodes query strings (and REST-format values as well) and goes on to do a smart best-fit mapping of the lot to procedure arguments.

The program takes a few additional steps in management of the mapping of URI to procedure. In order to prevent inadvertent unintended access to code, at startup it requires specification of a namespace path prefix which defines the space of allowable procedure mappings. It requires exporting of specific procedures (via the [namespace export] command) as a prerequisite for being called as a means of declaring them “public.” It also carries forward the concept of hierarchically-indexed access to services by looking for and loading Tcl package modules in the module filesystem path space if a procedure suitable for mapping is not already loaded in the interpreter.

This behavior illustrates how as an alternative to the dominant but flawed object-oriented design paradigm, this project looks for inspiration to concepts associated with the approach known as “service-oriented programming.” Service-oriented programming is typically used by large enterprises in pursuit of “service-oriented architecture,” which aims to coordinate a large number of disparate heterogeneous networked computer resources by strictly segregating them and letting them interact via a simplified, structured API schema.

To quote Wikipedia[3]: “In SOP [service-oriented programming]... software modules are strictly encapsulated through well-defined service interfaces that can be externalized on-demand as web service operations.” In addition: “a service can be composed of other nested services in a hierarchical manner.”

While making no attempt to conform to any official definitions or protocols, the code attempts to follow the service-oriented approach by casting Tcl package modules in the role of the aforementioned “software modules,” with namespaced and exported procedures stored in the modules as the “well-defined service interfaces.”

If generalized away from its enterprise programming origins, service-oriented programming might be seen as an abstraction of the “Unix way;”[4] that is, a means of fulfilling programming goals by using small code units each of which performs simple tasks well and interacts with the others via simple text-based interfaces. This project tries to iterate that abstraction by recognizing that a web server's function has evolved into that of a coordinator and collator of a variety of resources and information nodes, and thus the service-oriented approach is applicable reflexively both to the server's external role as a fulfiller of information requests on a hierarchized global network, and to the web framework's internal data marshaling challenges.

Now the beginnings of a hierarchy of services within the web framework can be envisioned, all leveraging the tools naturally available and used by the Tcl programmer in assembling a sophisticated software installation. The request URI maps to a procedure well-encapsulated in a hierarchical namespace. The namespaced procedure exists in a package which is stored in a Tcl package module file whose name by convention reflects the namespace path. The package module is stored in a filesystem directory path that mirrors the module name.

The program code transparently manages all these levels in search of the appropriate service mapping that will fulfill the web client's request. Thus the process of designing a dynamic web site boils down to tasks that feel quite natural to the experienced Tcl programmer: creating namespaced packages containing publicly-declared interfaces of exported procedures. The developer need only write such a package and drop the package file into the defined module library path space in order to instantiate a new web service API. The program takes care of all the rest: determining package availability, loading package modules, mapping procedures and arguments to inborne URIs, executing the procedures and returning results.

3. Additional Facilities

The code is formatted as a collection of packages. The project files are laid out so that they may be delivered as a Starkit, as a single package with sub-packages included in an internal library directory and loaded as needed, or as a traditional application with a bin subdirectory containing an rc file and a configuration file.

Included is the pure-Tcl web server Wibble, and may be started as a stand-alone web server. Or, it can be accessed via gateway protocols such as CGI or SCGI from a third-party web server.

3.1 Package: tserver

As stated, the program is a collection of packages, and its operation may be best explained in more detail by closer examination of the individual packages:

The package named tserver coordinates and controls the other packages. In addition, it manages startup and configuration tasks. By design tserver server code listening on a port plays nicely with other event-generating code that may be running in the interpreter. Unlike Tclhttpd, which keeps its configuration values in global variables and has a complex interrupt protocol if the developer wants to make use of event programming independent of the web server's function, tserver keeps its configuration state in the namespace in which it was started; and all event-level state is kept within the event's call stack.

Thus the framework can run multiple listener ports with the same configuration in the same namespace, or it can run listeners with completely different configurations in separate namespaces.

The tserver package manages event-level state by providing “getter” and “setter” procedures which are call stack-aware. The getter procedure can access server-level state from the controlling namespace. The setter uses the [info frame] command to introspect which call stack it occupies in the event-loop structure, and sets arbitrary desired values in a place restricted to the call stack's scope. Once a call stack-level value has been set, the getter will always retrieve that value, even if there is a server-level value with the same name. Thus code executed in response to a query can customize the call stack's state; a variety of independent services can contribute to construction of a response, and a coordinating procedure can finally collate the information contained in the event-level state into a response.

3.2 Package: Wibble

The tserver package makes use of the pure-Tcl web server package Wibble[5], written by Andy Goth. The version used is customized to eliminate use of coroutines in order to allow pre-Tcl version 8.6 use and for performance, and other optimizations.

Thus the program can be used as a stand-alone web server, or can be configured to accept straight redirects from another server or reverse proxy without additional processing. Since the flow of code in Wibble is well-encapsulated via use of dictionaries, it is fairly easy to write new front ends for the Wibble code which accept different input protocols, simply by constructing a custom request state dictionary and injecting it into the Wibble code flow. For example, an SCGI front end has been written.

Wibble uses its own controller/dispatcher for matching query URIs to procedures. It utilizes a table of “zone handlers,” where a zone is a URI prefix and the handler is a procedure, with optional custom state variables added to the request dictionary before dispatch. The framework own dispatcher function is constructed as a Wibble zone handler, so calls to it can be restricted to the domain of one or more specified URI prefixes.

The zone handler table is one of Wibble's most innovative and useful features. It is superficially like Tclhttpd's application direct URL feature (mentioned above); but instead of an array of keys and values, with one match opportunity, Wibble's dispatcher runs down the whole table and calls every procedure whose associated URI prefix matches the incoming query. Thus configuration of the table offers creative opportunities for multiple pre- and post-processing phases, and pipelining of response structures through multiple procedures. How this facility interacts felicitously with the framework service-handling architecture will be elaborated on further below.

3.3 Package: pkgTree

The package named pkgTree provides the controller and dispatcher procedures sketched above in section 2, as well as utilities for managing the package modules which contain the service code.

The package is so named because of the way the Tcl package module feature provides an alternate way of storing code packages: in a tree-structured filesystem directory location. (N.B. The “package module” feature is distinct from the classic package storage and discovery method; i.e., putting packages and their pkgIndex.tcl files in subdirectories of a directory which is listed in the global auto_path variable.)

Thus for example the module file for package textutil::string version 0.7.1, instead of being stored in a package library directory along with a pkgIndex.tcl file to control how it is sourced, is stored in a subdirectory of a module directory determined by the package name; i.e., <tm>/textutil/string-0.7.1.tm

The new(ish) package module feature provides a convenient and visually meaningful way to organize web server API code, since the pkgTree dispatcher assumes that the directory layout of the specified module path location mirrors the valid URIs of the server's API.

Thus, for example, if the dispatcher is initialized with the package prefix "API" and receives a query of the form:

http://example.com/document/statistics/wordcount?doc=tutorial.txt

then it will use the [namespace which] command to see if the procedure:

::API::document::statistics::wordcount

exists. If it does not, the dispatcher looks for the package:

API::document::statistics

and loads it. The dispatcher checks again if the procedure "wordcount" exists in the package, and additionally checks if the procedure is exported. If so, and the procedure has an argument named "doc", then the procedure is called with the argument appropriately set.

The combination of checking for the prefix ("API"), and for exported procedures only, ensures that only procedures the developer intends to be available are called.

Since REST-type calls are supported transparently, if the URL had been of the form:

http://example.com/document/statistics/wordcount/tutorial.txt

the dispatch process would have proceeded in the same way.

These facilities all come together when the developer comes to want to expand the API; for example to support the call:

http://example.com/document/transforms/translate?doc=tutorial.txt&lang=german

then the developer would simple create a package module called:

API::document::transforms

containing the exported procedure "translate" and drop the module file in the location <tm>/API/document/transforms-0.1.tm. Once that module exists the new API call is live, and the dispatcher takes care of loading and executing it, no server restarts or reloads necessary.

In line with the concept of using the URI as an index into different services, the pkgTree package also contains the procedure "resource". The resource directory allows an API procedure to retrieve static files from the filesystem without having to specify an explicit pathname.

The resource procedure is initialized with the location of a directory laid out as a mirror image of the API package module directory; that is, with sub-directories corresponding to

URIs supported by the API. If the resource procedure is called within an API procedure, it uses introspection to discover the namespace and procedure name of its caller, and retrieves the contents of the static file located in the resource directory corresponding to the URI path. If the retrieved file contains template code, the code will be executed and the final result returned.

Thus if a query of the form:

http://example.com/account/statement.html

were received, the procedure:

::API::account::statement.html

would be called. If this procedure were simply to contain a call to the procedure **::pkgTree::resource**, then the file **<resource>/account/statement.html** would be retrieved. This file could contain template code for returning customized information to the caller, which the resource procedure would see to executing.

4. Using Framework Features to Drive a State Machine

An advantage of mapping URIs to namespaces is that the pkgTree package's dispatcher can use the [namespace path] and [namespace which] commands to determine quickly which procedures are visible and executable. This approach contributes significantly to the performance, security and logical clarity of the code.

For the sake of security, the pkgTree dispatcher was designed with the requirement that it be initialized with a call to the [namespace path] command in the dispatcher's namespace to set a namespace value which acts as a prefix. The value set by the [namespace path] command is analogous to the PATH environment variable in a UNIX operating system, it determines the visibility of procedures as the PATH value determines the visibility of executable files. Thus no procedure that doesn't share the [namespace path] prefix will be callable by the dispatcher, so it is impossible for malicious hackers to construct a URL that would result in execution of code that was not intended to be part of the web server API.

Although not a design goal, the potential for a useful hack quickly became evident. There is no reason the namespace prefix set at startup time has to stay at the same value -- the [namespace path] command can be called at any time to set the visibility of code dynamically, and so the range of reachable procedures can be customized on a per-connection basis.

Given this capacity, it's not hard to envision the desirability of sometimes restricting a visitor's access to the API to a limited subset, depending on privileges or role. It's not hard to take the imaginings further to an API design that's segregated into logical groupings, even placed into entirely separate package module paths meant to satisfy different usages.

All that's needed to complete this scenario is an easy way to set the namespace path for an incoming connection, in a way that is clear and straightforward and thus not prone to confusion which would jeopardize security.

I have found, serendipitously, that the Wibble zone handler feature is the perfect tool for setting per-connection server state. It both allows creative configuration options and ensures the transparency necessary for high confidence in correctness of configuration. As stated above, the zone handler table can contain multiple entries that match a single URI, and each matching handler procedure is called in succession unless and until a procedure specifically calls an interrupt routine that terminates processing.

There is of course no necessity for any single handler procedure's code to address the contents of the server response, any desired task can be performed. Thus it is near trivially simple to segregate identity-validating, state-setting, and response-compiling operations in separate zone handlers.

And thereby an order of execution can be set up: an initial zone handler can validate identity by checking cookie values, session ID number or any other standard means. The handler can use the tserver package's per-connection state setting features to set an identity variable. A second zone handler can map an identity to a namespace path or set of paths, thus restricting the visibility of API procedures by the pkgTree package to a desired subset. Thus the different packages of the web framework leverage one another to restrict access to server resources using well-understood Tcl features. Finally, another zone handler can actually call the pkgTree dispatcher, which will only execute the requested API procedure if the dispatcher can detect its presence via the [namespace which] command.

Given the ability to specify such an order of execution, a namespace path can be utilized as more than a token of role-based access, its utility can be taken further to represent a session state that persists across visits. For example, a client interaction that requires sequential filling out of three separate forms can be controlled by using a pre-processing zone handler to restrict the visitor's access to the correct form at each point, and a post-processing handler that monitors if the form was completed successfully and storing in the session state the permitted namespace path of the next visit by the client.   This would make problems like backtracking and double form submitting impossible.

In this way, a server built around this framework can be cast in the role of a state machine, with the API namespace defining the total state space, and zone handlers controlling state transitions, driving visitors through a custom state sub-space on a per-session basis.

A state machine framework promises to make it easy to attain some web programming goals that have been difficult or impossible with other frameworks, such as workflow programming, sophisticated collaboration scenarios, and programmable virtual servers mediated within the framework.

4.1 Additional State Configuration Options

The above section presents one detailed scenario for controlling access to server resources. But access to virtually every other information source can be similarly programmed.

As stated, the tserver package allows configuration of server state on a per-connection basis. This includes for example the value of the “docroot,” the default location for static files. Instead of such complex and error-prone tools as htaccess files, the docroot can be given a default value of a directory containing only publicly accessible files, and only reset to a directory of more sensitive files after identity validation. There is thus no chance of misconfiguration causing leakage of access to restricted files, because the restricted files are simply invisible to the server until the validation step.

The custom version of the Wibble server package also includes a feature allowing the ability to reload the entire zone handler table and restart processing of handlers. Thus whole categories of function or complex workflows can be modularized and segregated in their own handler tables, and loaded as needed.

5. Conclusion

The aim of this paper is two-fold. First, to introduce this new framework and its architecture. Second, to communicate the observation that the experience of designing this program made it clear anew to me that Tcl's feature set makes it particularly well-suited to interfacing with the World Wide Web and serving the demands of the HTTP protocol. The failure of object-oriented design practices to deliver reliable solutions leaves a gaping need on the part of computer professionals who need to operate reliably and securely on the Internet. Tcl deserves to be seen as newly relevant in this space, not just as a good general-purpose scripting tool with potential for application to the WWW, but as the language and environment best suited to delivering on the usage goals of dynamic web development frameworks.

References:

[1] http://www.kalzumeus.com/2013/01/31/what-the-rails-security-issue-means-for-your-startup/

[2] http://tcl.activestate.com/software/tclhttpd/#advanced

[3] http://en.wikipedia.org/wiki/Service-oriented_programming

[4] http://c2.com/cgi/wiki?UnixWay

[5] http://wiki.tcl.tk/23626