Webshell: A Tcl-based Web Application Framework

Simon Hefti, Ronnie Brunner and Andrej Vckovski
Netcetera AG
{simon.hefti,ronnie.brunner,andrej.vckovski}@netcetera.ch

Abstract

Webshell is a rapid development framework for building powerful and reliable Web applications. It is a standard Tcl extension and is released as Open Source Software. Webshell is versatile and handles everything from HTML generation to data-base driven one-to-one page customization. At Netcetera, we have been using it for years for virtually all our customer projects, which are typically E-commerce shops or electronic banking applications. Webshell is extensible and portable, and its comprehensive set of commands is easily learned. Webshell can currently be used in a CGI environment or as an Apache (1.3 and 2) module. This paper describes the key features of Webshell, details it's architecture, and motivates design decisions.

1 Introduction

In 1999, Netcetera decided to overhaul its main workhorse called Webshell 2 for the development of Web applications, and to make it available under an Open Source license. We have been using Webshell releases 1 and 2 in many projects over several years. With this experience, we felt that we could release the tool as Open Source software, not least also because we did profit from many different Open Source software, including Tcl and many of its extensions. However, the Webshell release 2 was not easy to publish, because:

- it needed private Tcl include files,
- it had our own `Tcl_Main` which made it hard to include other extensions unless they were dynamically loadable. Also, commercial development aids (e.g., the TclPro-suite) were not usable due to the rather special execution semantics of Webshell 2.
- it was written in C++ which did and sometimes still does lead to portability problems and also to a certain degree of redundancy with, e.g., Tcl's base library function such as I/O and memory allocation.
- it did not yet use Tcl's dual-ported objects internally.

In addition, there were other pending requirements from our internal projects, like changeable methods for session management and the ability to work as a server extensions as an alternative to CGI. Finally, we wanted to work through our list of known limitations and problems.

These issues motivated us to completely overhaul Webshell. While keeping the design patterns from Webshell 2, we rewrote most of the actual code. This paper covers Webshell release 3, the result of that major release. It is organized as follows: first, we discuss basic patterns of web application development and compare different approaches. Then we discuss the Webshell concepts and their motivation.

2 History

For many software systems it is very helpful to understand the development history as a motivation for various design decisions. With Webshell 3, the history actually shows an interesting cycle which began with a scripting-only solution, which was followed by a C++-only solution, and gradually developed into a mixed-system, with the scripted part gaining weight.

Initially, a simple Tcl-based layer has been written in 1994 to support simple CGI-based applications for the then young World-Wide Web. This layer did basically handle form input, URL decoding and also some simple ways of session tracking, and it was only used for internal, system-administration related applications. With a first commercial E-business application in mind we then developed a C++-based application framework that solved the same problems but was modeled similar to other application frameworks such as Microsoft's Foundation Classes, ET++, Interviews, etc. This framework provided a class library that allowed a simple and fast development of CGI applications in C++. For configuration purposes we embedded a Tcl interpreter in the class library. That is, the interpreter was basically used as an intelligent configuration file parser.

With the use of this framework we realized that most developers were actually using the interpreter for application logic, too, moving code into the script domain, because this allowed changes without recompilation. As a result, we decided to use the C++ class library to develop a generic application container, having the entire application-logic implemented as Tcl scripts. This generic application container was called Webshell and used Tcl versions 7.4/7.5/7.6. With the release of Tcl 8.0 in 1997 we released Webshell 2 which became the main workhorse for almost all customer projects at Netcetera and also for a few other companies.

The current release, available as Open Source software, has been developed starting in mid 1999. As has been mentioned above, the main reason was to make Webshell easily "distributable" and portable. The outline of Webshell 3 defined the following issues:

- Backport form C++ to C for easier portability, thread-safeness and less redundancy with Tcl-core routines.
- Distribute according to Tcl's Extension Architecture (TEA, see references).
- Be thread-safe to not undermine Tcl's thread-safeness and to be easily included in multi-threaded applications
- Use Tcl's object system wherever possible
- Provide better specialization of the logging module, session management and other features that might need site-specific adaptations.
- Provide both an Apache module and a stand-alone interpreter to use in CGI applications and batch-processing modules.

In early 2000 we released the first two beta releases. We will release Webshell 3 final as soon as Apache 2.0 will be final. Even though there is no strong dependency on Apache 2.0 we still believe that it will be sensible to have a matched release. Tcl-based extensions to Apache 2.0 will have a certain advantage to other modules (e.g., PHP or Perl) since Tcl's core is thread-safe and therefore, easily deployable in a multi-threaded processing module of Apache 2.0.[1]

The following section will now shortly give a general overview on building Web applications and give a few examples of corresponding Tcl-based environments.

3 Web application basics

3.1 Approaches

In the last few years it has become a standard paradigm to develop applications that use a Web browser as user interface, both for the Internet and also for internal applications on an Intranet. Many techniques, commercial and open-source tools have been developed to support the development of such applications that provide dynamic Web content. These tools typically fall into the following categories:

Microscripting or server-side includes

This technique is available with most HTTP servers. It is based on a proprietary extension to HTML, which is parsed by the server whenever the corresponding document is requested. These extensions contain directives for conditional text and text substitution. This technique is only suitable for applications where text substitution is possible, that is, for simple replacements in the document. It also strongly depends on the server software used, as there is no widely accepted standard for the syntax. Such template-based approaches include Microsoft's Active Server Pages (ASP), Java Server Pages (JSP), and PHP. Tcl implementations include AOL's Web Server, NeoSoft's NeoWebScript, or Vignette's StoryServer.

Custom HTTP servers

Most off-the-shelf HTTP-servers serve static contents from a repository, which typically is either a file system or a database management system. It might make sense for some applications to manage

[1]Apache 2.0 does include so-called multi-processing modules which allow on a platform with threads to use them. This provides better performance than the fork/exec-model used in earlier generations.

the HTTP-protocol themselves, i.e., to act as custom HTTP servers, having the full freedom on how the results of HTTP requests are returned. This technique is the most flexible. However, there is much application overhead introduced in properly (and efficiently) handling HTTP-requests, that this method is only sensible for very specific applications. The Tcl-only Web-Server tclhttpd is such an example.

CGI

Often dynamic content is managed using the Common Gateway Interface (CGI). The widespread use of CGI is basically due to its simplicity. The CGI specification relies on POSIX-compliant execution of child processes. Parameter passing happens via command line arguments, environment variables and standard I/O. CGI has become the most popular technique for dynamic content because it is standardized. For Tcl, Don Libes' cgi.tcl (see references) is the most popular Tcl-only package for CGI handling.

Embedded handlers and modules

Most current HTTP server software products support a direct extension through a specific API (e.g., NSAPI for Netscape products, ISAPI with Microsoft Information Server, ”modules” for Apache-based Products). This method is probably the most efficient, since there is a tight binding between server and extension. On the other hand, this solution is dependent on the HTTP server used. Java servlets can be considered to fall into this category.

All approaches mentioned above do have advantages and disadvantages. Let's examine two common trade-offs that are often debated.

3.2 CGI versus embedded execution

CGI has been very popular since the first available Webservers (CERN's httpd and NCSA httpd) did support it. The main drawback of CGI is that the Web-server needs to spawn a new child for every request to handle. This has several impacts:

- Even on platforms that are based on a frequent-subprocess-creation paradigm such as Unix, a process creation is much more expensive than, for example, an in-process function call. That is, CGI is generally "expensive".
- In addition to the process-creation costs, a CGI needs always to reinitialize all application-context unless potentially dangerous techniques such as shared-memory are used.
- Limited or slow resources such as database or mainframe connections with long latencies have to be managed using complex pool managers since a CGI application, being a standalone-process, cannot pass any resources to a follow-up request.

On the other hand, there are also invaluable advantages of CGI:

- CGI is "industry"-standardized. That is, CGI application can be easily installed on any HTTP server supporting CGI (most do).
- CGI is very robust. Since CGI-applications run in a separate process, its isolation of address-space etc. prevents malign applications to bring down the entire HTTP server.
- CGI applications have very short lifetimes. This does not only provide more robustness of the entire system (resource-leaks don't impact general stability), but also allow quick and immediate reconfiguration.
- CGI applications are often easier to debug and test because the can be run without a HTTP server.
- Sometimes the same application might need to work both as CGI and as a standalone application.

Thus, there are situations where a CGI approach is the better way nonwithstanding the negative performance impact. The FastCGI protocol is a way to have multiple requests served by the same instance of an application. However, the robustness is partially lost (even though the application cannot bring down the Web server).

3.3 Template-based approaches versus constructive rendering

At first glance, template-based approaches such as JSP or ASP seem to be a very logic and intuitive approach to provide dynamic content:

- It is a straight-forward idea to replace the static parts of a document with code that generates the corresponding dynamic content when the page is rendered.

- Complex HTML-page-layouts that are (or have been) necessary to overcome HTML's limitations can be developed using standard tools and then "turned" into an application by adding the dynamic parts.
- Application programming merges with content management.

For simple applications and portal-like integration pages these are very powerful and useful approaches. However, with increasing application complexity there are several drawbacks:

- Some applications might require other document types than HTML to be returned to the browser, e.g., images, PDF documents, Java class files, etc.
- The "distributed-transaction" or half-transaction problem (see below) leads to complicated solutions with templates.
- The application logic gets spread over many templates which is problematic from a configuration management perspective.
- Application logic and its visual appearance are often tightly coupled.
- In some cases (e.g., PHP) it is difficult to use common code for both the Web application and other process in a complex system (e.g., daily batch jobs, other user interfaces).

Again, it depends mainly on the problem at hand, which one of these approaches is better suited. In general, the more complex an application is and the more internal state it has, the less a template-based approach should be considered.

The half-transaction problem mentioned above shows a basic drawback of the "a request corresponds to a page" assumption of template-based approaches. Consider a simple order form that has basically two states. In the first state, a form is displayed that lets the user enter, say, delivery information. After submitting the form, the data will be written, say, to a database, and a confirmation page will be displayed. That is, there are basically two HTML pages that are shown in the browser: the form and the confirmation page. Now it is common that the processing of the input data in the form will include various types of validations (e.g., required input fields) that might lead to the necessity to redisplay the form, now with some error messages, preserving the correctly entered values. That is, there are two state transition possible as a result of submitting the form: back to the form or continue to the confirmation page. Based on some application logic it might be either necessary to display the form again or then the confirmation page. The typical template-based solutions are:

- Either place all possible output into a single template and conditionally show only the form part or the confirmation part.
- Or implement some kind of "internal redirect" that forwards the request from the confirmation page to the form page in case of a validation error.

The basic problem herein is that the application logic as seen on a conceptual level (display form, user enters data, collect input, and show confirmation) does not match the HTTP transactions which are: display form, collect input, re-display form, collect input, and show confirmation. The the steps that usually belong together in traditional GUI applications (render dialog and collect user input) are spread over HTPP transactions. In order to minimize the impact on the application's complexity it is therefore often sensible to have a single application that handles all states and transitions in an application rather than to spread it over many templates or scripts.

3.4 Where does Webshell fit into that picture?

Webshell does provide the necessary functionality to allow multiple application states and their transitions to be implemented in a simple way within the same application object. Typically, the various states and their corresponding output are constructed using HTML fragments rather than templates, but Webshell does also support template substitution where it is sensible to use templates.

Webshell-applications can be used both through the CGI interface and also an Apache module. Its CGI interface ensures that Webshell can be employed virtually anywhere, using off-the-shelf web servers. If performance is key, Webshell can be run as an Apache module. In this case, the server does not spend time spawning child processes, and communication between web server and Webshell directly uses the Apache API, which is considerably faster than in the CGI case. In addition, Webshell applications can be split into two parts, the command definition and the command call part, and Apache can keep the command definition part in memory. This eliminates the need to re-read most parts of a Webshell application for every HTTP request, and further increases the responsiveness of the Webshell application. Also, the necessity of connection-pools for long-latency resources can be reduced because connection handles and the like can be kept for multiple requests.

The following section will now describe Webshell's design in more detail.

4 Webshell concepts and design

4.1 Sub-components

Webshell is based on a few sub-components. Each module manages its own data, and locking mechanisms are used where needed. These Webshell modules are extensible through plug-ins. This section gives an overview of the Webshell modules and shortly describes their function and design.

Request and URL management

Handles input from the HTTP protocol, and session tracking

Session context management

Session data handling. On the client-side using Netscape cookies, and on the server-side using various storage managers (file system, data base, main-memory cache)

Output management

Sending data back to the client

Conversion

HTML- and URI-compliant encoding and decoding of data

Encryption

security by strong encryption

Messages

Message-oriented communication over TCP/IP (Berkeley) sockets

Logging

Generation, filtering and distribution of log messages

The Webshell modules are shown in figure 1.

4.1.1 Request and URL management

The Webshell application developer does not need to get involved with the details of the HTTP protocol or the parsing of data. Rather, she concentrates on the application logic and leaves the rest to Webshell. This module parses input from the client, e.g. HTML form data[2], and makes it available to Webshell. In addition, it adds state to the HTTP protocol using the query string or Netscape cookies.

One of the distinctive features of Webshell is its session management capability. HTTP as a protocol is usually stateless (unless the secure variant SSL/TLS is used). By protocol definition, there is no information which lives from access to access. This design has many advantages, like simple client-server application design. If state is needed, however, it makes application development more difficult.

Web-based applications often need to carry information from one HTTP transaction to the next. As an example, a user might have a preferred language. Applications for electronic commerce systems, Internet banking an so on also need mechanisms to identify and group transactions into longer transactions which cover more than one single HTTP request and its response. In other words, state needs to be introduced.

Typically, there are three methods in use to identify state:

- Encoding of state information in the URL
- Encoding of state information in hidden form fields
- Using Netscape cookies.

Webshell is designed to use all of these methods. However, the first is Webshell's preferred method. The reason to rely mainly on the URL is that both hidden form fields and cookies have disadvantages:

[2]Both `x-form-urlencoded` and `multipart/formdata` MIME types are supported

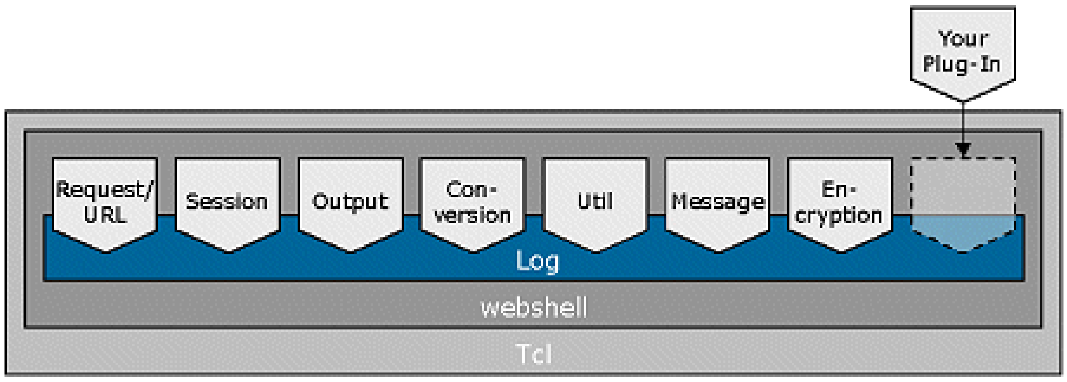

Figure 1: Webshell modules

- Hidden fields are only usable in forms. However, there are frequently cases where state needs to be preserved without explicitly requiring forms. Moreover, the state might be required to be ”bookmarkable”

- Firewall setups can filter cookie information from the HTTP-stream in order to avoid external information from entering an organization. Also, the user can disable cookies, and not all browsers support cookies.

URL encoding, on the other hand, has the disadvantage that it produces somewhat long URLs. Experience has shown that the advantages more than compensate for this drawback.

4.1.2 Session context management

We have described above why web applications need state information. Often, more information is needed than simply preferences. In these cases, Webshell relies on its session management module.

The session management module handles session data, which can be stored on the client side using Netscape cookies, or on the server side using the file system, a data base management system (DBMS), or any other storage manager. The module provides a uniform interface to access the session context regardless of the storage used. It is implemented in Tcl and makes extensive use of namespaces, which makes it easily extensible while ensuring a storage-independent API for the application developer.

4.1.3 Output management

Webshell provides a set of commands to format HTTP-compliant output to be sent back to the client. Webshell manages response objects which direct output to a Tcl channel or a Tcl variable for buffering purposes. The technique of buffering output is typically used to prepend additional information to the output after the data has been processed. For example, one would like a table of contents to a text document.

Again, this module has been designed with the philosophy in mind that the application developer does not need to care about the details of the HTTP protocol. As an example, he does not track whether the MIME headers have been sent or not. Webshell takes care of that because the response objects maintain state.

If control over the HTTP output is needed, the configuration facility of the output management module gives full control over the output. Of course, the output module is fully configurable, as are all Webshell modules.

4.1.4 Conversion

The conversion module makes sure that your data is properly formatted, converting umlauts to their proper HTML entities or their URI encoded equivalent, for example. Like every Webshell module, this module manages its data on its own. In order to ensure thread safety, no Webshell module maintains global or static variables. Such variables would be shared across multiple Tcl interpreters and cause conflicts. Each time an interpreter loads Webshell and thus this module, the module-specific data like the look-up tables for the conversions will be set up, and destroyed again when the interpreter is deleted. The disadvantage of this design, the duplication of data in the memory of one process, is compensated by the long-term stability of the process resulting from clean memory management.

4.1.5 Security by encryption

The proper handling of sensitive data is crucial for banking or E-commerce applications. Three aspects are important: Data transfer, data storage, and session hijacking. The first of these aspects, data transfer, is not Webshell's task. Webshell assumes that data transfer is secured whenever needed, for example by means of SSL, and Webshell deliberately chooses not to provide any additional functionality here.

Handling of sensitive data, on the other hand, is the task of Webshell and the Webshell application. This is why Webshell comes with a strong encryption sub-system to store data securely. By default, Webshell encrypts the state information in every URL it generates. The level of security is chosen by the application developer. She selects both the encryption method and the encryption key. For highly sensitive data, the application developer might not even know the pass-phrase to decrypt data from the productive environment.

Webshell relies on well-known and well-tested encryption methods, which are made available to Webshell via plug-ins. Currently, there are plug-Ins available for the IDEA and the blowfish encryption methods, as well as for a simple encryption method for day-to-day use.

Webshell's standard encryption plug-in protects data with a checksum to prevent cipher tampering. For decryption, Webshells encryption sub-system uses an autodetection system to determine the encryption method.

Note that Webshell provides the framework to handle sensitive data properly. The level of security, however, is determined by the developer of a Webshell application.

4.1.6 Messages on streams

This Webshell module implements a simple platform-independent protocol to facilitate message-based communication over Tcl channels. Particularly, it is used for communication over TCP/IP connections.

4.1.7 Logging facility

Like all service applications, Web-applications need a versatile logging mechanism to report errors and state. In fact, Webshell itself makes heavy use of the logging facility. Logging must be easy to use, fast, and extensible. Typically, Webshell applications handle many requests per second, and the logging facility has been designed with this kind of load in mind.

The Webshell logging module manages a list of filters and a list of destinations. Each log message comes with an `"address"` consisting of a tag and a level, which is compared against two filters before the message is delivered to its destination. First, an overall filter determines whether or not Webshell needs to process the message. Then, a second, per-destination filter determines whether or not the message has to be sent to the log destination.

The Webshell logging module is fully extensible through plug-ins. Out-of-the-box, Webshell comes with plug-ins to log to files, `syslog`, Tcl channels, Tcl commands, and, in the case of mod_websh, to the apache logging facility. Logging to commands gives the application developer a simple mechanism to extend the logging module by writing Tcl commands.

4.2 Hello, world!

The following example demonstrates a very simple Webshell script which produces the simple message `Hello, world`

```
   web::put "Hello, world!\n"
```

This script can be used as is as a CGI application. Webshell takes care of MIME-headers and other details of the HTTP protocol.

4.3 HTML output

Webshell intentionally contains no support for HTML rendering other than built-in translation functions for HTML entities. Therefore, Webshell can be used for non-HTML output as well. Typically, an application contains a set of abstractions on top of HTML that an application developer uses. For example, a generic page of an application might be defined as:

```
proc page {title code} {
   web::put "<html><head><title>"
   web::put $title
   web::put "</title></head>"
   web::put "<body><h1>$title<h1>\n"
   uplevel $code
   web::put "</body></html>"
}
```

And then used as:

```
page "My page" {
   if {$foo==42} {
       web::putxfile helppage.html
   } else {
       showForm
   }
}
```

4.4 CGI and Apache module

Webshell applications can be run both as CGI-applications and within the Webshell-Apache module. Both environments are transparent to the developer to a large degree. That is, all relevant information (e.g., HTTP headers, input and output streams, etc.) are provided to the application through the same interface, independent if CGI or the Apache module is used.

In the case of the Apache module, however, an application can serve more than a single request. For that purpose, the Webshell Apache module maintains a pool of interpreters that can be reused for many requests. Depending on the configuration, the interpreters might be shared among various applications (e.g., if it is used in a micro-scripting fashion) or reserved to individual applications. For every interpreter, an *initializer* and *finalizer* script can be provided which can be used to load necessary code and handle cleanup.

The Apache module has initially be designed for use in Apache 2.0 only. Tcl's thread-safety since version 8.1 makes Tcl an ideal candidate for Apache 2.0 modules. However, the release of Apache 2.0 has been delayed considerably, so that the first two beta releases of mod_websh (Webshell 3.0) were built against Apache 1.3.x.

4.5 Security issues

Web-applications are subject to security issues as are all networked applications. Furthermore, these applications are often deployed in the Internet, making the security issues even more important. In the context of Webshell there are four different relevant security topics discussed here:

- Can Webshell applications be used to gain unauthorized access to a computer?
- What levels of stability do these applications show, i.e., do Webshell applications often fail?
- How does Webshell manage user authentication and user authorization?
- How is sensitive data handled?

Webshell does not offer any direct solution to these problems. Rather, Webshell is designed to avoid problems (intrusion, stability) and allow simple inclusion of various authentication and data encryption models.

The problem of gaining unauthorized access is too important to be solved by the tool used for application development. Therefore, the design of Webshell assumes that the surrounding system architecture, i.e., Web server and Firewall handle these issues. This means, that the Web server is responsible to execute CGI applications with suitable low privileges, in Unix environments typically by setting the effective UID to an unprivileged user and executing the CGI-application and/or Web-server in a chroot-environment.

However, a Webshell-developer can introduce security holes. The flexibility of Webshell clearly allows dangerous things to be programmed. Therefore, security issues have to be checked with every application.

The stability issue has rather different characteristics than the intrusion problems, but it is nonetheless very important. Unstable applications typically ”get touched” very often, leading to generally less secure systems. Also, crashing applications can impair a system's stability if the applications take unexpected execution paths. The design of Webshell is lead by the idea to provide a simple, flexible and easy-to-understand framework. Compared to other products, Webshell deliberately prefers transparency over high integration and automation. This simplicity and flexibility, somehow similar to the Unix model, leads to a high stability of the overall system. At the time of this writing, Webshell has been in production for almost four years, without any major stability problems so far.

User authentication is a third security domain important in many Web-based applications. The design of Webshell deliberately does not include any specific method for user authentication. The reason for this decision is that the specific authentication method needed largely depends on the context of the application. Some applications might authenticate users using a host system, others need specific one-time-password schemes, and some applications might even rely on authentication schemes provided by Web-servers and -browsers. The extensibility of Webshell allows various authentication schemes to be adopted.

Finally, the importance of the secure handling of sensitive data is becoming more and more recognized. Here, Webshell provides a extensible encryption module which allows one to easily encrypt data before storing, and to decrypt the data after reading. For example, a simple encryption is used by default for the query string generation, which might contain sensible information. This plug-in uses checksums to ensure the full transmission of encrypted text and time stamps in order to ensure unique data. The encryption sub-system of Webshell is easily extensible by plug-ins which can be loaded when needed.

5 Conclusions

The Webshell application framework has proved its efficiency and robustness in many real-world applications, ranging from stock quote display systems with many million transactions per day to various E-commerce applications with end-to-end integration. With the release of the software as Open Source, we would like to encourage other developers to use it in their web applications and to help with the further development of the framework. It is also thought as a contribution to thank many authors of Open Source software for their excellent products that are used in many, including our, projects.

6 References

1. *Webshell* `<http://websh.com>`
2. *AOLserver* `<http://www.aolserver.com>`
3. *mod_dtcl Apache plugin* `<http://comanche.com.dtu.dk/dave/>`
4. Welch, Brent. *The TclHttpd Web Server*. 81–95. Proceedings of the 7th Tcl/Tk Conference. Austin, TX, February 14-18, 2000.
5. Welch, Brent, and Thomas, Michael. *The Tcl Extension Architecture*. 151-161. Proceedings of the 7th Tcl/Tk Conference. Austin, TX, February 14–18, 2000.
6. *cgi.tcl* `<http://expect.nist.gov/cgi.tcl>`

A Commands

This appendix lists the main commands of Webshell. For a complete overview, refer to the *Quick Reference* `<http://websh.com/quickref.html>`.

A.1 configuration

web::config

Webshell configuration for encryption, file upload file size limit and the like

A.2 command dispatching and session management

web::command

register code for later execution with web::dispatch

web::getcommand

retrieve registered code

web::cmdurl

create a URL which links back to the application

web::cmdurlcfg

configuration of URL generation

web::dispatch

call registered code depending on request

A.3 request data handling

web::request

access request specific information like HTTP_REFERER

web::param

access parameters from the query string

web::formvar

access variables from the HTML form

A.4 response data handling

web::response

manage Webshell response objects

web::put

send string to response object

web::putx

send string to response object, while executing embedded code

web::putxfile

web::putx content of a file

A.5 logging

web::logdest

manage log destinations

web::logfilter

manage log filters (only messages passing this filter will be written to a destination)

web::log

issue a log message

A.6 context handling

web::context

setup command to manage information (memory-based)

web::filecontext

setup command to manage persistent information (file-based)

web::cookiecontext

setup command to manage persistent information (cookie-based)

web::filecounter

setup command to generate sequence numbers

A.7 file handling and file I/O

web::include

source/load file

web::readfile

read file, store in variable

web::lockfile

file locking using flock()

web::unlockfile

file locking using flock()

A.8 data encryption

web::encrypt

encrypt data

web::decrypt

decrypt data

A.9 uri-/html- en-/decoding

web::htmlify

translate data to HTML

web::dehtmlify

remove HTML tags from data

web::uriencode

convert data to URI format

web::uridecode

convert data from URI format

A.10 inter-process/-system communication

web::send
: send data to socket in network byte-order

web::recv
: recieve data from socket

web::msgflag
: test for flags in web::send/web::recieve protocol

A.11 Apache module specific commands

web::initializer
: register code to be executed when mod_websh loads this application for the first time

web::finalizer
: register code to be executed when mod_websh delets this application

web::maineval
: execute code in the main interpreter of mod_websh

web::interppool
: set/get properties like lifetime and maximal idle-time of mod_websh interpreters

A.12 misc commands

web::match
: check for existence of element in list

web::tempfile
: return name of temporary file

web::copyright
: return version and copyright information

B Requirements

- Tcl 8.2.2 or higher
- For use as a web application, a web server supporting CGI is needed.
- The Webshell module for Apache 2 requires Apache 2.
- The Webshell module for Apache 1.3 requires Apache 1.3.x.