# Entice – Embedding Firefox in Tk

**Steve Landers**
**Digital Smarties**
**steve@DigitalSmarties.com**

## Abstract

This paper describes Entice - a Tcl/Tk extension that allows multiple Firefox windows to be embedded into Tk and controlled from a Tcl/Tk application. Entice brings to Tcl/Tk applications on X11 similar functionality to that provided by using Internet Explorer and Optcl on Windows. The paper describes the use of the TkXext extention to embed Firefox, the use of web technologies (XUL, Javascript and AJAX) and a small embedded web server for communication with Firefox. Future developments such as providing a generalised cross-platform/cross-browser API will be discussed.

## Background

On Windows it is possible to embed instances of Internet Explorer (IE) into a Tk application using COM – via the Optcl[1] or Tcom[2] extensions. This makes practical a new genre of hybrid client/web applications such as NewzPoint[3] (which displays multiple web pages in a two-tiered tab layout arrangement using IE as the display engine).

Entice is an attempt to provide the same (or similar) functionality for Unix/Linux platforms - in the first instance, Tk under X11 but potentially under other windowing systems such as MacOS X Aqua (although that is beyond the scope of this paper).

There are three distinct requirements in doing this:

- embedding a browser in Tk
- communicating with the browser (i.e. sending content)
- controlling the browser from Tcl

## A browser in a Tk application?

There are a number of approaches to providing a browser within a Tk application:

- use a Tk extension that “wraps” a browser library
- use a Tk browser extension
- embed an external browser window into a Tk frame

The first approach was used in the TkGecko[4] project, which wraps the Mozilla Gecko rendering engine as a Tk extension. This approach provides a full featured browser that is relatively easy to control from Tcl. but comes at the cost of complexity and size. Building Gecko (and indeed most full-featured browsers) is notoriously difficult, and getting the Mozilla event system to work seamlessly with the Tcl event system is tricky.

But more significantly, the resulting Tk extension is large – a problem when the application has to carry around its own copy of a shared library (so as to be self-contained) and counter-productive to the Starkit[5] approach of small / lean applications.

The second approach of using a Tk-based browser addresses the size and complexity issues. But at its current stage of development, even the most advanced one - Tkhtml[6] - whilst very impressive doesn't yet offer the level of compatibility needed in many web applications.

The third approach is to use a pre-existing browser installation and embed its window(s) into a Tk container frame. The trade-off is that it is more complex to control the browser (some operations can only initiated "from the inside") and an application can't be totally self-contained (i.e. as a Starpack). Having said this, these downsides are far less significant that the upside of smalll size and removing the need to build and deploy the browser.

Accordingly, this was the approach chosen for Entice – and the browser of choice was Firefox, which is fast becoming the dominant Mozilla-based browser on Linux and Unix.

## Embedding Firefox

The Tk frame(n) command supports a "–container" flag that tells Tk the frame will be used as a container in which some other application (in this case Firefox) will be embedded.

George Peter Staplin's TkXext[7] was used to perform the embedding. TkXext is a Tcl/Tk extension designed to facilitate automation of software testing, but is also useful for application embedding. It allows the X11 window ID for an application to be located by searching for its window title. This window ID can then be used to re-parent the application window into the Tk container frame.

For example, the following code creates a Firefox instance, creates a Tk frame to hold it, looks for the Firefox instance using TkXext.find.window and then re-parents it into the Tk frame.

```
# create blank Firefox instance
exec [auto_execok firefox]about:blank &

# create Tk frame to hold Firefox
set f [frame .f -width 200 -height 200 -container 1]
pack $f -fill both -expand 1

# find and re-parent the Firefox window into the Tk frame
if {[set id [TkXext.find.window "Mozilla Firefox"]] != 0} {
    TkXext.reparent.window $id [winfo id .f]
}
```

There is a small problem with this scheme - it assumes the default title of a blank Firefox window is "Mozilla Firefox", and that the user doesn't create another blank Firefox window between when the Firefox instance is created and when it is searched for. Whilst this is unlikely, it is still quite possible – so Entice needs set the Firefox window title to a unique value, and to do that it needs to communicate with Firefox.

## Communicating with Firefox

When Entice is initialised it creates an embedded web server on a  listening (i.e. -server) socket. This web server will only accept connections from the local host, and implements a small set of requests used by Firefox to communicate with Entice.

```
if [catch {set listen [socket -server accept 0]}] {
   error "couldn't open listening socket"
}
…
# accept a connection on the listening port
proc accept {sock host path} {
   if {$host ne "127.0.0.1"} {
      # this should never happen under normal circumstances
      catch {close $sock}
      error "received connection from wrong host"
   } else {
      fconfigure $sock -blocking 0
      fileevent $sock readable [list request $sock]
   }
}
```

Firefox is started with its “--chrome” argument pointing back to the web server.  A chrome[8] defines the user interface outside the browser’s content area,  and is specified in XUL[9] (pronounced *zool*) – the XML User Interface Language. It is usually read from the host filesystem, but it can be read from a URL

```
set port [lindex [fconfigure $listen -sockname] 2]
exec [auto_execok firefox] -chrome http://localhost:$port/init &
```

This will cause the embedded web server to receive a /init request

```
# accept a request from Firefox and act upon HTTP GET requests
proc request {s} {
    variable sock
    set sock $s
    set line [gets $sock]
    if {[fblocked $sock]} return
    fileevent $sock readable ""
    fconfigure $sock -buffering full
    lassign [split [string trim $line]] op arg rest
    if {$op eq "GET"} {
        switch -glob $arg {
            "/init" {
                initbrowser
            }
          …
    }
    catch {flush $sock ; close $sock}
}
```

The Entice “initbrowser” proc reads the entice.xul file (comprising approximately 90 lines of Javascript) which is stored externally to entice.tcl to facilitate maintenance. The XUL file does need the appropriate Entice web server URL substituted but is otherwise passed to Firefox untouched.

```
proc initbrowser {} {
    variable port
    variable sock
    set fd [open entice.xul]
    set xul [read $fd]
    close $fd
    regsub -all {%URL%} $xul http://localhost:$port/ xul
    set code [header
           "Content-type: application/vnd.mozilla.xul+xml" \
          "Content-length: nnn"]
    append code $xul
    puts $sock $code
}

proc header {args} {
   return "HTTP/1.1 200 OK\n[join $args \n]\n\n"
}
```

At this point Firefox is running, and it has been supplied with some Entice specific startup code. But what does that code contain, and how can Entice control Firefox when the usual mode of operation is for the browser to pull content from the web server?

## Controlling Firefox

The key to Entice controlling Firefox is the XMLHttpRequest Javascript object running in the browser.

XMLHttpRequest[10] was originally implemented by Microsoft for IE 5 on Windows, and has since been implemented in Mozilla and Apple’s Safari. It is the command at the heart of AJAX[11] (Asynchronous Javascript And XML) that is being promoted as “next big thing” in web technology. XMLHttpRequest allows the browser to wait asynchronously for a request from the server, whilst remaining responsive to the user – and has made applications like Google Earth possible (arguably taking web applications from “sucky” to “adequate” when compared with their more traditional cousins).

When the Entice chrome is loaded by Firefox it creates an XMLHttpRequest object that waits for commands from Entice, as shown in this extract from entice.xul

```
function request(cmd) {
    var url = "%URL%" + cmd;
    req = new XMLHttpRequest();
    req.onreadystatechange = command;
    req.open("GET", url, true);
    req.send(null);
}
// start the Ajax connection
request("cmd");
```

This will cause the “cmd” request to be sent to Entice and the Javascript “command” function to be called when a response is received by Firefox. Let’s look at each of these in turn.

The “cmd” request is quite simple – it just tells Entice that Firefox has started and requests a command. We have already seen the “request” proc, here it is repeated with the “/cmd” section added

```
proc request {s} {
    …
    if {$op eq "GET"} {
        switch -glob $arg {
            "/init" {
                initbrowser
            }
            “/cmd” {
                variable started 1
                return
            }
        }
    }
    catch {flush $sock ; close $sock}
}
```

Note that all “/cmd” returns without closing the socket – the socket is the means that commands are sent from Entice to Firefox to control it.

But what has Entice been doing whilst waiting for Firefox to initialise? Back when Firefox was started Entice immediately waited for it to send the /cmd request - as indicated by the “started” namespace variable.

```
vwait [namespace which -variable started]
```

Now that Firefox has started,  when an Entice API command is invoked by the Tcl/Tk application, it is encoded as XML and written to the socket by the “response” proc

```
proc response {cmd id {arg -}} {
    variable sock
    while {$sock eq ""} {
       # multiple commands issued - wait for next XMLHttpRequest
       vwait [namespace which -variable sock]
    }
    set resp [header "Content-Type: text/xml"]
    append resp "<command><cmd>$cmd</cmd>"
    append resp "<id>$id</id><url>$arg</url>"
    append resp "</command>"
    puts $sock $resp
    catch {flush $sock ; close $sock}
    set sock “”
}
```

As can be seen, the XML protocol is quite simple – the command is passed, along with two arguments : the applicable window ID and optionally a URL.  Hand coded XML (so to speak) is fine – this protocol doesn’t demand tdom or equivalent.

In the browser the XMLHttpRequest Javascript object will receive and decode the response, and call the “command” Javascript function supplied as part of the Entice XUL. The command function is just a big switch that implements the browser side of the protocol that travels between Entice and Firefox.

```
function command() {
    if (req.readyState == 4) {
        if (req.status == 200) {
            var cmd = req.responseXML.getElementsByTagName("cmd")
                                        .item(0).firstChild.data;
            var id = req.responseXML.getElementsByTagName("id")
                                        .item(0).firstChild.data;
            var url = req.responseXML.getElementsByTagName("url")
                                        .item(0).firstChild.data;
            var win = windows[id];
            switch (cmd) {
               …
            }
        }
    }
```

Note that an XMLHttpRequest,readyState of “4” signifies the successful completion pf the HTTP interaction, and an XMLHttpRequest.status of 200 indicates a request completion code of “OK”.

So what are the commands implemented in the command function? For that we need to look at the Entice API.

## The Entice API

To include Entice in an application, naturally enough you just “package require entice”. To initialise Entice use the entice::init command (all Entice commands live in the “entice” namespace).

```
package require entice
entice::init
```

The entice::init command starts Firefox as described previously, and waits for Firefox to load its chrome and create an XMLHttpRequest object.

To create a new Firefox window and corresponding Entice object, use the entice::new command. Here’s an example that creates a paned window with two panes containing Firefox

```
set p [panedwindow .p -orient vertical]
set f1 [frame .p.f1 -container 1]
set f2 [frame .p.f2 –container 1]
.p add $f1 -height 300 -width 600 -sticky nswe
.p add $f2 -height 300 -width 600 -sticky nswe
set w1 [entice::new http://www.tcl.tk .p.f1]
entice::new http://wiki.tcl.tk .p.f2 w2
```

The entice::new command returns an object handle that is used to control the corresponding Firefox window. The new command also supports an optional third argument that specifies the name of the Entice object. In the above example the object name for the first Entice object is stored in the w1 variable, whereas the object name is set to w2 in the second.

Entice can be told to wait until the Firefox window has been embedded using the wait method. This is typically used to delay packing/gridding the container frames until Firefox has been re-parented into it

```
$w1 wait
w2 wait
pack .p –fill both –expand 1
```

Again, note the different between $w1 and w2 - the first is a variable holding the object name generated by entice::new, and the second is the object name we specified.

Other Entice methods are used to manipulate the corresponding Firefox window
- location – change the website being viewed
- back – go back to the previous location (equivalent to the back button)
- forward – go forward to the next location in the browser history (equivalent to the browser forward button)
- reload – reload the current location
- replace – replace the current location in the history with a new location

For example:

```
# change top pane to ActiveState’s website
$w1 location http://www.activestate.com

# now back to www.tcl.tk
$w1 back

# forward to www.activestate.com again
$w1 forward

# reload www.activestate.com
$w1 reload

# replace with ww.eolas.com
$w1 replace http://www.eolas.com

# back to www.tcl.tk
$w1 back
```

There are also methods for controlling the embedding of Firefox
- unparent – unparents the Firefox window – i.e. moves it back into the root window of the hosts window manager
- reparent – re-embeds the Firefox window in a (possibly different) frame - the inverse of unparent
- close - close the Firefox window (the container frame is untouched)
- frame – returns the name of the container frame

For example:

```
# unparent Firefox from existing frame and re-parent in
# another frame
$w1 unparent
pack forget .p
pack [frame .f -container 1] -fill both -expand 1
$w1 reparent .f

# retrieve the container frame name
set container [w2 frame]

# close the Firefox window
w2 close
```

## Another look inside

Now that we've seen the API we can take another look inside Entice, and in particular at the flow of control when creating and embedding a Firefox window.

When the entice::new command is called, it creates a unique ID to use in the Firefox window manager title, so that it can correctly detect and re-parent the Firefox window using TkXext. This ID needs to be difficult to guess, but need not be cryptographically secure – so we just use Tcl's "clock clicks" and the "expr rand" command. The entice::new command then sends a "new" response to Firefox via the XMLHttpRequest mechanism, and creates an alias to the Entice object command via the Tcl "interp alias" facility.

In the browser the Javascript "command" function is triggered and passed the "new" command, which invokes the Javascript "create" function passing it the window ID and the URL

```
function create(id, url) {
    var win = window.open("", "",
      "width=1,height=1,resizable,scrollbars,dependent,status=no");
    win.document.title = "Entice - " + id;
    win.setContextMenu = function (menu) { return false; }
    win.location = url;
    windows[id] = win;
    request("embed?" + id);
}
```

The create function creates a browser window with the specified window title using the DOM (Document Object Model) [12] function calls, and then sends a request back to Entice requesting it embed the Window into Tk.

When Entice receives the embed request it uses TkXext to find the window containing the unique ID generated by the entice::new command, and then re-parents it into the frame provided as an argument to that command.

This all sounds complicated but is in fact relatively simple – Entice requests Firefox create a window and then returns to the application, Firefox creates the window and then requests Entice embed it. And this all happens asynchronously - Asynchronous Javascript and XML and Tcl.

In fact there are only three commands ever sent from Firefox to Entice (although there could be more, as additional functionality is added)

- init – indicating that entice.xul should be passed to the browser
- cmd – indicating that the chrome has been loaded, an XMLHttpRequest object created and Firefox is ready to accept a command
- embed – requesting that the specified window ID be re-parented into the Tk frame

For each protocol command from Entice to Firefox there is corresponding Javascript code that implements the required functionality. This is usually just a call to the Firefox DOM, as shown in the following switch statement from within the "command" function:

```
switch (cmd) {
    case "back":
        win.history.back();
        break;
    case "close":
        win.close();
        break;
    case "forward":
        win.history.forward();
        break;
    case "location":
        win.location.href = url;
        break;
    case "new":
        create(id, url);
        return;
    case "print":
        win.print(); // not yet in the Entice API
        break;
    case "reload":
        win.location.reload();
        break;
    case "replace":
        win.location.replace(url);
        break;
    case "show":
        break;  // not yet implemented
    case "hide":
        break; // not yet implemented
    case "current":
        break; // not yet implemented
}
request("cmd"); // request next command
```

These commands correspond to the functionality provide by Optcl. New Entice API methods can be added by implementing a procedure in entice.tcl and a corresponding entry in the command function within entice.xul.

## Problems and opportunities

Whilst Entice works (and works well) it is not without problems.

In particular, it is possible to kill the Tcl/Tk application and leave a Firefox instance running. In the absence of Tcl core support for kill and signal this means interrupt handling will need to be done in the application code rather than handled within Entice. If this isn't done right it is possible to have Firefox not respond correctly during the initialisation phase (it appears to get confused as to which process it should be talking to). As at the time of writing this paper, this problem hasn't been diagnosed sufficiently to provide a workaround.

On the other hand, an opportunity exists to make Entice extensible – that is, provide a way for additional methods (and corresponding Javascript commands) to be specified at runtime by the application program.

Likewise, the API could be enhanced to facilitate feeding content to the browser from the Tcl/Tk application – perhaps using a library like Htmlgen (part of the Tclxml package) – which facilitates writing Tcl code that generates structured HTML or XML.

And ultimately Entice could be extended to support Firefox and Safari on MacOS X, Firefox on Windows and to wrap OpTcl on Windows – providing a single cross-platform API for embedding and controlling a browser within Tcl/Tk.

## Deployment and Licensing

Entice is deployed as a Tcl package comprising entice.tcl, entice.xul and a corresponding pkgIndex.tcl – it contains no compiled code although it does require the TkXext binary extension to be present. Both Entice and TkXext are fully compatible with the Starkit deployment technology – just drop the Entice and TkXext directories into the lib directory of an application Starkit and the Entice package is available for use in the application.

The code was developed under contract to Eolas Technologies Inc. Eolas intends to release the code for noncommercial use under a Tcl-friendly BSD-style Open Source license, while retaining certain intellectual property rights in the code for commercial use. Eolas intends to make such a release in the very near future. For further details, please contact Mike Doyle or Jim Stetson at Eolas.

## Conclusion

Entice has achieved the goal of delivering a fully-functional browser within a Tcl/Tk application. It does so without the overheads usually involved with wrapping a browser library, or the limitations of using one of the Tk browser extensions.

In doing this, Entice brings a new genre of hybrid client/web applications to Linux/Unix – and is perhaps the first application of AJAX technology to Tcl/Tk.

## Acknowledgements

Entice is one of the Stargus projects sponsored by Eolas Technologies Inc, who's ongoing support for this and other projects like Cryptkit, Critcl 2 and Vlerq is very much appreciated. Accordingly thanks are due to Mike Doyle of Eolas Technologies for his support and encouragement.

George Peter Staplin not only developed TkXext but provided fixes, enhancements and support when Entice stretched it beyond its initial capabilities.

And thanks to Mark Roseman and Mike Doyle for their reading of the draft of this paper.

## References

[1] OpTcl - http://www2.cmp.uea.ac.uk/~fuzz/optcl/default.html
[2] Tcom - http://www.vex.net/~cthuang/tcom/
[3] NewzPoint - http://wiki.tcl.tk/newzpoint
[4] TkGecko - http://wiki.tcl.tk/TkGecko
[5] Starkits - http://www.equi4.com/starkit.html
[6] Tkhtml - http://tkhtml.tcl.tk/
[7] TkXext - http://wiki.tcl.tk/TkXext
[8] Mozilla Chrome - http://www.mozilla.org/xpfe/ConfigChromeSpec.html
[9] Mozilla XUL - http://www.mozilla.org/projects/xul/
[10] XMLHttpRequest - http://en.wikipedia.org/wiki/XMLHttpRequest
[11] AJAX - http://en.wikipedia.org/wiki/Ajax_%28programming%29
[12] DOM - http://en.wikipedia.org/wiki/Document_Object_Model and http://www.mozilla.org/docs/dom/