Toward RESTful Desktop Applications

William H. Duquette
Jet Propulsion Laboratory, California Institute of Technology
William.H.Duquette@jpl.nasa.gov

Abstract

The REpresentational State Transfer (REST) architecture includes**:** the use of Uniform Resource Locators (URLs) to place a universe of data into a single namespace; the use of URL links within the data to allow applications and users to navigate the universe of data; HTML/CSS for the presentation of data; a limited set of operations that are available for all URLs; multiple content types; and content negotiation when retrieving data from a URL. REST is primarily used in web applications; however, pure desktop applications can also benefit from RESTful concepts and technologies, and especially from the integration of web-like technologies with classic application software. This paper describes how REST concepts and technology have been used in the Athena simulation to present a vast sea of heterogeneous data to the user.

1. Background

The Athena Stability & Recovery Operations (S&RO) Simulation is a model of political actors and the effects of their actions in a particular region of the world. The region is divided into neighborhoods, in which reside various civilian groups. The actors have a variety of assets, including money, military and police forces, and means of communication, which they use to achieve their political ends. The extent to which they succeed depends on the attitudes of the civilians, which change in response to current events. The model runs for a period of months to years, and produces a vast quantity of data, all of which needs to be presented to the analyst in some form or other.

The Problem

Athena stores most of its data in an SQLite3 *run-time database* (RDB). In Athena V2.0 most data was made available to the user by taking the output of a particular database table or view and throwing it into a `tablelist`-based browser.[1] Such a tabular display is useful; but when the information about a particular entity, an actor, say, is extremely heterogeneous, one tabular display cannot tell the whole story. It is possible to collect together the information about the actor by looking across a number of tabular browsers…but not surprisingly our users thought that the application ought to be doing this for them.

If only there was an easy way of presenting heterogeneous data to the user, while taking advantage of relationships within the data as an aid to navigation….

The Solution

HTML/CSS is a powerful, well-understood means of presenting heterogeneous data to the user. Uniform Resource Indicators (URIs) are a powerful means of identifying specific resources to present to the user from within a vast sea of such resources. Links to URIs embedded in the data are a powerful means of allowing the user (or the application) to navigate the sea of data. The resource pointed at by a URI can exist in multiple content-types; through content negotiation, the client can retrieve the content-type that is most useful for its purposes. These have generally been used in web applications. However, there is no reason why these concepts cannot be fruitfully used in the desktop environment within the context of a single application with no network interfaces, when the application's data model calls for it.

2. The Desktop REST Architecture

HTML, URIs, and the rest of the web technologies described above were created to support an architecture called REpresentational State Transfer (REST) [2]; an application that uses REST is called a *RESTful application*. REST is a web architecture; this section describes how we have modified the basic concept to create a desktop REST architecture within our application.

REST: A Summary

A RESTful application, or client, accesses *resources*: collections of data, or indeed any kind of entity, by means of *Uniform Resource Indicators* (URIs), of which there are two kinds, *Uniform Resource Locators* (URLs), for resources that can be located and retrieved on-line, and *Uniform Resource Names*, which are unique names for entities that exist off-line.

The client accesses these resources by means of a handful of verbs, which in principle apply to all resources. In a traditional REST app, which uses HTTP for its transport, these are usually GET, PUT, POST, and DELETE.

The resources are provided to the client by a *server*, and the server provides the data in a form called the *content type*. Content types are typically expressed as MIME types such as `text/plain` and `text/html`. A single resource might be available in any number of content types, and the precise data returned for the resource might differ from one content-type to the next. (E.g., `text/html` contains structure in a way that `text/plain` does not.)

The client accesses a server using an *agent*. The client gives the agent the URI of a resource, and a verb, and the agent locates the server and accesses it on the client's behalf. In particular, the agent handles *content negotiation*: given the content types the client is prepared to handle, the agent works with the server to provide the resources to the client in the content type it would most prefer.

A resource's content frequently contains URIs linking to related resources. The client can make use of these URIs to navigate the sea of resources.

The most common content type is `text/html`, because it provides a way to display the resource data attractively and allows the user to navigate the data space by clicking on links. These days, HTML documents typically use Cascading Style Sheets (CSS) for formatting and Javascript for interactivity. In a Tcl/Tk application, naturally, Tcl replaces Javascript.

These concepts and technologies provide just the thing to display heterogeneous, highly linked data to the user.

Why Not a Web App?

The advantages of the REST architecture would seem to be an argument for implementing Athena as a web application, yet there are compelling reasons for not doing so.

- Athena already exists as a single-user desktop application; moving to the web would change the architecture considerably.
- Network interfaces come with security headaches. And although Athena is not classified, it is often used in classified environments where network resources are tightly controlled and security is taken *very* seriously.
- Ease of installation is key; we do not want to require the users to install a web server. We could work around the installation issue by embedding something like TclHTTPD in Athena; but that still leaves us with the security headaches.
- We've not been asked to, nor do we have funding to make such significant changes, or to come fully up to speed on robust, secure web applications.

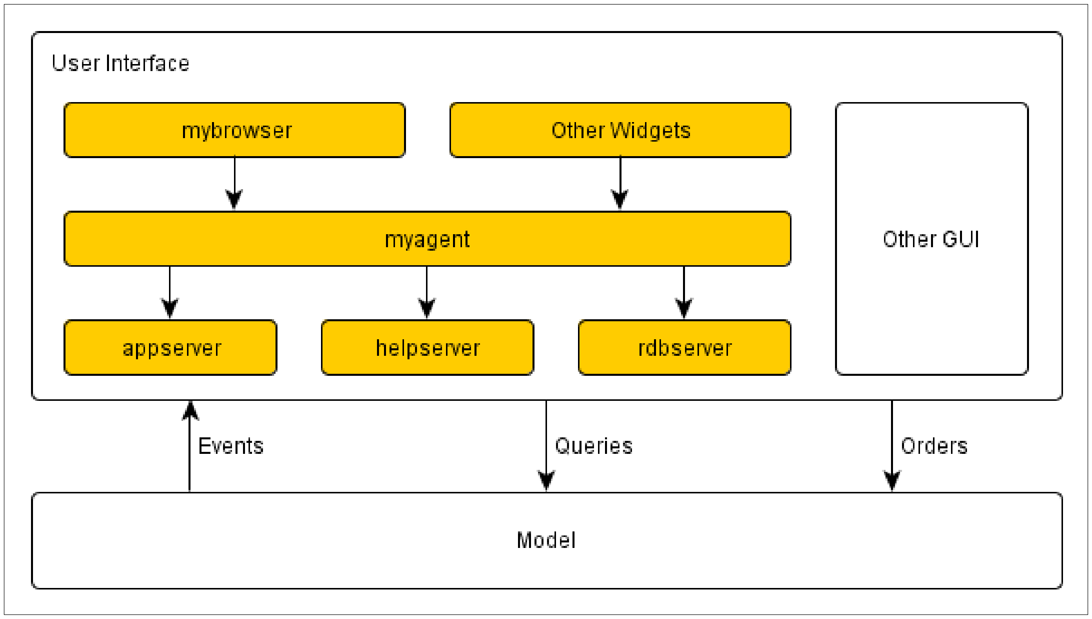
Adapting REST to the Desktop

So the question becomes, how do we use these RESTful concepts in a desktop application? We need to:

- Define a set of URIs that give access to various application resources.
- Determine the relevant content types. We use standard content types like `text/html`, but also types relevant to the desktop environment, such as `tk/image` and `tk/widget`.

- Implement a content server, and an agent with which to access it. Because the server resides within the application itself, access can be synchronous; the protocol reduces to a set of procedure calls.
- Specify tools for parsing URIs. We use the uri package from Tcllib.[3]
- Create tools for generating HTML output. (Yes, I wrote yet another HTML-formatting module. It's just something I do.)
- Choose a widget for displaying HTML/CSS
- Implement a web-browser-like mega-widget on top of TkHTML 3.0.
- Implement other widgets that can take advantage of server content.

With the RESTful components added, Athena's architecture is as shown in the following diagram; the new components are shown with a shaded background.



The Model represents the non-GUI portion of the application, including all management of scenario data and the simulation proper. As described in [4], Athena's User Interface interacts with the Model via three mechanisms. First, the UI can query the Model in any way it likes, provided that the queries do not affect the content of the Model in any way. Second, it can send *orders* to the model; all changes to Model content and operation are triggered by these orders. Third, the Model can send *events* to the UI, to notify it of particular happenings within the model. This portion of the Athena architecture remains unchanged from previous versions.

As a consequence of this existing architecture, we have not implemented the PUT, POST or DELETE verbs of the REST architecture; the existing mechanisms handle these operations perfectly well. Instead, we have focused on the GET operation, which is what we chiefly need to present information to the user.

At present, Athena includes three servers. The `helpserver` serves up on-line help pages from a pre-compiled help database. The `rdbserver` provides access to the schema and content of the application's run-time database as an aid to development and debugging. The `appserver` is the most important of the three, as it provides access to the Model's resources. These servers are all instances of the `myserver` type.

Each of these servers is registered with the `myagent` module; instances of `myagent` provide GET access to the servers, and also do content negotiation.

Instances of `mybrowser` can be used to browse the content of these servers in the usual way; and there are other widgets that access the servers as well.

3. Displaying HTML/CSS

Desktop REST stands or falls on the application's ability to display HTML content. And in order to display HTML content, or at least HTML-like content, in a Tcl/Tk application, you need to have an HTML widget. There is no perfect choice; this is a place where Tcl/Tk is sadly lacking. The available options are these:

- Solutions based on the Tk `text` widget
- TkHTML 2.0
- TkHTML 3.0 [5]
- A wrapper around Gecko or some similar engine HTML engine.

It is possible to do a mostly adequate job of displaying an early version of HTML in a Tk `text` widget; it handles links and interaction perfectly well, and it can even display images and embedded widgets. HTML-style tables are a problem, however, and tools to position images and embedded widgets precisely relative to the text (e.g., wrapping paragraphs around an image) are lacking. In short, the Tk `text` widget is a solution, but only a mediocre one for this purpose. (Were we to use it, we'd probably abandon HTML in favor of a Tcl-based presentation language, to avoid parsing.)

Athena 1.0 and 2.0 had a help browser based upon TkHTML 2.0. It is stable, having been abandoned long ago, but it is highly quirky and its HTML support is archaic. Font support is problematic; for example, you can have monospace type or bold type, but not both at the same time. It claims to support embedded widgets but in our experience all attempts to do so end in a crash. In our experience TkHTML 2.0 edges out the Tk `text` widget for display of rich content, primarily due to its support for tables, but it is not very satisfactory.

Another option is TkGecko [6], a Tk wrapper for Mozilla's Gecko HTML engine. It is clear from the TkGecko paper that Gecko is very much a moving target, and that wrapping it in a robust way is by no means easy. It would be an interesting choice if we wished to display live web content from over the network, but we do not; and stability is crucial.

TkHTML 3.0 is an HTML/CSS renderer implemented as the basis for a Tcl/Tk web browser. Abandoned some years ago, it has not kept up with the latest web standards. It has more than enough horsepower for displaying application data, however, including tables, embedded images, embedded widgets, and complex formatting. The bare widget lacks event bindings and other features that were provided by the web browser within which it was to be embedded, but once these are provided it becomes quite satisfying to use. It is fast, versatile, and sufficiently stable for our use, and is what we have opted to use.

4. The URI Scheme

Athena uses two distinct URI schemes, neither one of which is found in the wild: the `my://` scheme and the `gui://` scheme.

The `my://` Scheme

The most usual URI scheme used in Athena is the "`my://`" scheme, which is a simplification of the familiar `http://` scheme. `my://` URLs have the same syntax as `http://` URLs, with the unnecessary parts (port numbers, passwords, etc.) omitted:

```
my://server/path…?query#anchor
```

Here the server is the name of a `myserver` registered with `myagent`, and the path, query, and anchor are defined as usual.

We chose the name "`my:`" for this scheme because the named resource belongs to the application itself, rather than to some other entity out in the network. We considered abusing the `http://` scheme but rejected this for two reasons. First, we wanted to make it absolutely clear that Athena has no network interface; it is not pulling resources down from the web. Second, it allows us to modify the standards for `http://` URLs without causing confusion to future developers.

The `gui://` Scheme

The `gui://` scheme is a set of Uniform Resource Names (URNs) for entities in the Athena GUI. Links using this scheme are not handled directly by the `myagent`/`myserver` infrastructure; instead, the `mybrowser` widget hands them to its parent object via a callback, which hands to the application for handling. The upshot is that the user can click on a link in a browser, and the application will take them to some other tab in the GUI, or pop up an order dialog. For ease of parsing, the `gui://` scheme also uses a subset of the usual `http://` syntax.

5. Content Types

The `myserver` component allows each instance of the server to serve up content of any imaginable type. The standard MIME types `text/html` and `text/plain` are used for HTML and plain text context respectively; for consistency, application-specific content types are named in the same style, with “`tk/`*`type`*” used for Tk-specific content and “`tcl/`*`type`*” used for other kinds of data. The application-specific content types currently in use described in the following subsections.

The `tk/image` Content Type

The content consists of the name of a Tk image. An instance of `mybrowser` can display `tk/image` content directly and as the `src` of an HTML `<img>` tag.

The `tk/widget` Content Type

The content consists of a Tcl script to create the widget so that it can be displayed in an HTML page. The HTML `<object>` tag is used to embed widgets in pages; for example, the following HTML embeds a time plot in the page:

```
<object data="my://app/plot/time?start=2+vars=basecoop"
width="100%" height="3in"></object><p>
```

The query portion of the URL specifies the variables to plot, and the start time of the interval for which they should be plotted. The server uses these to customize the widget options, and then returns the script to create the widget. (The TkHTML 3.0 widget handles the width and height itself.) For example:

```
timechart %W –vars basecoop –start 2
```

The server doesn't know the window name to use, so it inserts a "%W" in place of the window name. The mybrowser substitutes in the window name and creates the widget, which then appears in the web page.

The Netscape Tcl plugin was never so easy.

The `tcl/enumlist` Content Type

This content type is simply a Tcl list of enumerated values; it is usually used to populate pulldowns in HTML forms, but can also be used by non-browser widgets.

The `tcl/enumdict` Content Type

This content type is similar to `tcl/enumlist`, but the value is a dictionary of enumerated values and their human-readable equivalents. It is also used to populate pulldowns in HTML forms.

The `tcl/linkdict` Content Type

This content type is used to represent trees of links. A `tcl/linkdict` is a nested dictionary mapping URLs (relative to the current server) to link metadata, primarily a human readable `label` and a `listIcon`, a Tk image to display next to the label. As such it represents one node in the tree, and its immediate children. By recursively retrieving `tcl/linkdicts` for the URLs, a component like the `linktree` widget can build up a tree of model entities or help pages.

6. Software Components

The Athena infrastructure includes the following software components.

The `myagent` Component

The `myagent` component is responsible for managing all interaction between clients and the various `myserver` instances. Servers register themselves with the `myagent` module, and instances of `myagent` retrieve data from the servers, doing all necessary URI resolution and content negotiation.

When creating an instance of `myagent`, the client specifies the content types it is prepared to handle, and the default server to contact:

```
myagent $agent \
    -defaultserver app \
    -contenttypes {text/html text/plain}
```

The client can then retrieve a URI's content as follows:

```
set cdict [$agent get $url]
```

The agent will throw a NOTFOUND error if the data cannot be retrieved; otherwise, it returns a dictionary with three keys: `url`, `contentType`, and `content`, which the client can do with as it pleases. If desired, the client can specify the desired content type or types explicitly:

```
set cdict [$agent get $url tk/widget]
```

Instances of `mybrowser` will normally accept `text/html`, `text/plain`, and `tk/image`, but will explicitly ask for `tk/widget` when handling an `<object>` element.

The `myserver` Component

Instances of the `myserver` component are registered with `myagent`, and thus become accessible to the application. Each instance of `myserver` defines the set of URLs that it can handle, and the content types for each:

```
myserver ::appserver
myagent register app ::appserver

appserver register / {/?}   \
    text/html [list /:html] \
    {Athena Welcome Page}

appserver register /actor/{a} {actor/(\w+)/?} \
        text/html [list /actor:html]          \
        "Detail page for actor {a}."
```

Each of these calls technically registers a pattern, rather than a specific URL; the handler handles all URLs that match the pattern. The first pattern registered above is simply "/", the top-level page for the server; the second registers a URL with a place holder for an actor's symbolic name.

For each pattern, we specify a unique name, e.g., `/actor/{a}`, and a documentation string; these are used in the server's `/urlhelp` page, which every instance of `myserver` provides automatically. Next, we provide a regular expression, which matches URLs of the correct pattern. (Note that “^” and “$” are added to the expression automatically.) The regular expression may include parentheses to indicate match parameters; these will be provided to the handler. Finally, for each URL we specify a set of content types and handler commands.

Thus, when the server is given a URI it matches it against the registered resources; if a match is found, and the URI has a compatible content type, the handler for that content type is called. For example:

```
proc /actor:html {udict matchArray} {
    upvar 1 $matchArray ""

    set actor [string toupper $(1)]

    if {![actor exists $actor]} {
        return –code error –errorcode NOTFOUND \
            "Unknown entity: [dict get $udict url]"
    }
    .
    .
    .
    return $content
}
```

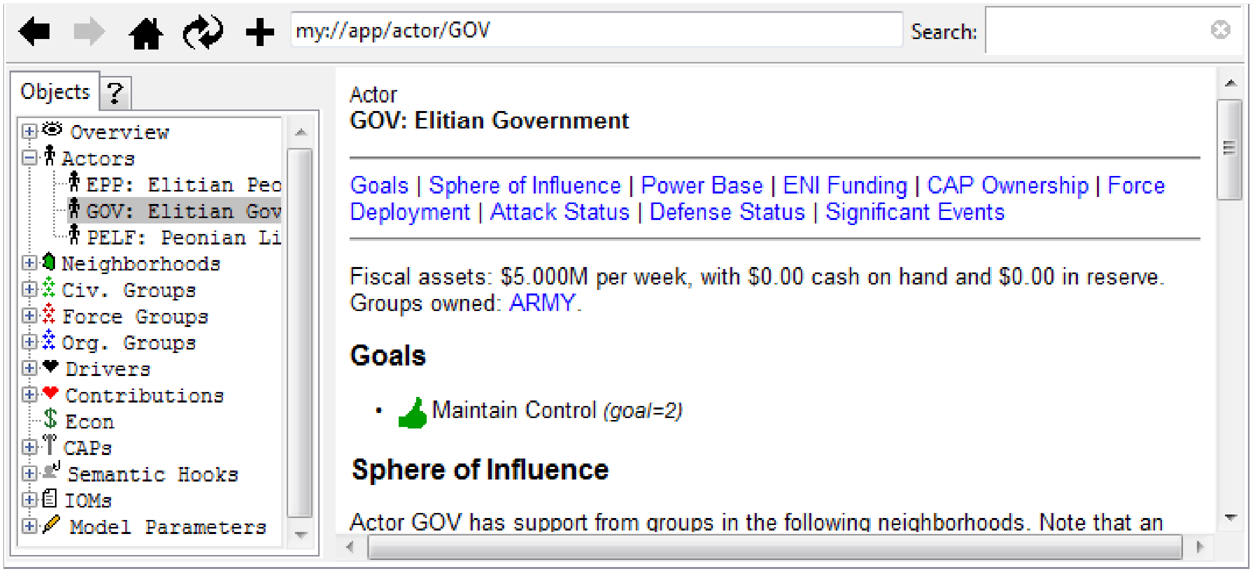
The *udict* parameter is a dictionary of the components of the URI: the path, the query, and so forth, as returned by `uri::split`. The *matchArray* parameter is the name of an array variable containing the matches from the regular expression; in this case, the actor’s symbolic name. The handler may make use of both the *udict* and the *matchArray* or neither.

The `mybrowser` Component

The `mybrowser` component is a web-browser-like widget built on top of TkHTML 3.0. It has its own instance of `myagent`, and thus can retrieve resources from servers. In addition to the normal browser navigation tools, it has the following capabilities:

- Display `text/html`, `text/plain`, and `tk/image` resources.
- Embed `tk/widget` content in `text/html` pages, when specified using the `<object>` tag.
- Support HTML forms.

The following figure shows an instance of `mybrowser`. The toolbar, scroll bars, html pane, and the paned window widget that allows the side bar to be resized, are all provided by `mybrowser`; the sidebar itself is an instance of `linktree` (see Section The linktree Component).



Actor
GOV: Elitian Government

Goals | Sphere of Influence | Power Base | ENI Funding | CAP Ownership | Force Deployment | Attack Status | Defense Status | Significant Events

Fiscal assets: $5.000M per week, with $0.00 cash on hand and $0.00 in reserve.
Groups owned: ARMY.

Goals

- Maintain Control *(goal=2)*

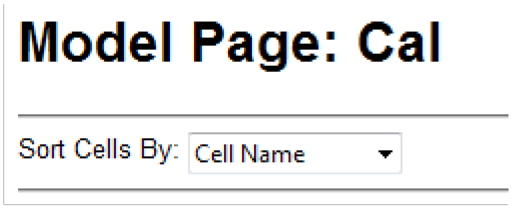
Sphere of Influence

Actor GOV has support from groups in the following neighborhoods. Note that an

The browser's support for HTML forms is robust but idiosyncratic. Athena has its own set of data entry field widgets which do not entirely match up to the standard HTML form fields; consequently, it provides its own mapping of `<input>` types and attributes to data entry fields, ignoring the standard HTML input types completely. For example, this HTML creates a form consisting of a single "enum" field, essentially a pulldown containing items from an enumerated list. The list of values comes from URL `my://app/enum/sortby`, which must provide content type `tcl/enumdict`. The default value for the pulldown is "`name`".

```
<form action="my://app/page/Cal" autosubmit="yes">
<label for="sortby">Sort Cells By:</label>
<input name="sortby" type="enum" content="tcl/enumdict"
       src="my://app/enum/sortby" value="name">
</form>
```

The form looks like this in use:



Model Page: Cal

Sort Cells By: Cell Name

When the form is submitted, which will happen automatically when the user selects a new value from the pulldown, the form’s values will be appended to the `action` URL as a query, and the URL will be retrieved:

```
my://app/page/Cal?sortby=name
```

At present, `mybrowser` supports `enum`, `text`, and `submit` input types.

The `myhtmlpane` Component

The `myhtmlpane` component is essentially a `mybrowser` without the navigation controls. It is intended to display a single page, retrieved from a `myserver`, as an alternative to a window defined using normal Tk widgets. If the user clicks on a link on the page, the URI is passed along to the application for display in the application’s main browser.

The linktree Component

The `linktree` component is a Tk treectrl widget configured to display a tree of resource links retrieved from a given URL. The widget retrieves its top-level items from the URL, and then works its way recursively down the tree, retrieving `tcl/linkdict` content at each node. The descent ends when a leaf no longer has any `tcl/linkdict` content associated with it. Optionally, the `linktree` can retrieve content for non-leaf nodes when they are first expanded.

The sidebar in the browser screenshot in Section The mybrowser Component shows a linktree of simulation entities.

The htmlframe Component

Although not actually part of the RESTful infrastructure, the `htmlframe` widget has proven to be a useful addition to the toolkit. It is simply a TkHTML 3.0 widget configured to layout its children according to an HTML layout string. For example:

```
htmlframe .f
ttk::entry .f.first
ttk::entry .f.last

.f layout {
    First Name: <input name=”first”><p>
    Last Name: <input name=”last”><p>
}
```

It includes a `set` method to set attributes of HTML elements by `id`; thus, the application can customize the appearance by setting CSS classes or styles on particular elements dynamically, or simply by providing a new layout. And since the TkHTML 3.0 widget supports scrolling, it is easier to create a scrolling window than it is using a standard `frame` widget.

This can be a much simpler way to create a complicated GUI layout than using the normal Tk geometry managers.

7. Status and Future Work

The infrastructure described in this paper is currently in use in two applications: the Athena simulation proper and in a separate development tool used to debug certain kinds of models. It has proven to be powerful, effective, and easy to use. The Athena application defines three servers and over sixty distinct URL patterns, many of them with placeholders. Many pages use forms and embedded objects, and that number is expected to increase over time.

It is possible that future applications may opt to extend the `myagent/myserver` pair with PUT, POST, and DELETE operations, and make use of these instead of Athena's existing “order” mechanism for editing and creating application data. Such an application would be truly RESTful, rather than merely “accidentally RESTful”, as now.

8. A Bit of Advocacy

Tk needs a robust, solid, well-documented HTML widget for uses like those shown here, and the existing TkHTML 3.0 widget makes a good starting point. The secret is to stop chasing the big browsers; we will never have enough development horsepower to keep up with Mozilla, Microsoft, and Google, and even if we could produce a widget that was completely up to date and could display any HTML page on the web, it would be out-of-date in months, if not weeks.

But this is OK. An HTML widget need not be capable of doing everything Firefox does to be useful to the application.

9. References

[1] Nemethi, Csaba, Tablelist Widget, http://www.nemethi.de/.

[2] Sletten, Brian, "Resource-Oriented Architectures: Being 'In the Web'," in *Beautiful Architectures*, pp 89-109, 2009, O'Reilly & Associates, ISBN: 978-0-596-51798-4.

[3] Kupries, Andreas, and Ball, Steve, `uri` URI Utilities package, found in Tcllib, http://tcllib.sourceforge.net/doc/uri.html.

[4] Duquette, William H., "The State Controller Pattern: An Alternative to Actions", 17th Tcl/Tk Conference, http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2010/WillDuquette/Statecontroller.pdf.

[5] Kennedy, Dan, TkHTML 3.0 Widget, http://tkhtml.tcl.tk/tkhtml.html.

[6] Petasis, Georgios, "TkGecko: Another Attempt for an HTML Renderer for Tk", 17th Tcl/Tk Conference, http://www.tclcommunityassociation.org/wub/proceedings/Proceedings-2010/GeorgePetasis/TkGecko.pdf

10. Acknowledgements

This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration, during the development of the Athena Stability & Recovery Operations Simulation (Athena) for the TRADOC G2 Intelligence Support Activity (TRISA) at Fort Leavenworth, Kansas.