OpenACS: robust web development framework

Author: Rocael Hernández, Galileo University, Guatemala / OpenACS Core Team, roc@viaro.net
Author: Andrew Grumet, OpenACS Core Team, aegrumet@alum.mit.edu

Abstract:

OpenACS is a full featured web development framework to create scalable applications oriented to collaboration and online communities. Is in use by many big players such as greenpeace.org or the e-learning platform of the MIT Sloan School of Management. While the system is not trivial, here are explained some of the most interesting and still relatively simple facilities that the framework provides. Everything from templating, separating the code from the presentation, database interactions according to the programming language, auto-documentation features, automated test engine, internationalization, and many more are written in Tcl, which has shown to be extremely powerful for writing the foundation logic and the application pages.

1. Introduction

The Open Architecture Community System (OpenACS) is a Web development framework for building applications that support online communities.

OpenACS provides a robust infrastructure, building on top of the following standard components: the Tcl programming language, a Postgres or Oracle database for storing the application data, AOLserver for HTTP service and *nix or Windows operating systems.

Like other modern Web frameworks, OpenACS supports: templating for separating the logic from the presentation, internationalization to present the user interface in the user's preferred language, a modular package system to create sub-applications, a role and permissioning system, a content repository to store all manner of content and maintain versioning.

2. Basic infrastructure

2.1. AOLserver

OpenACS is built atop the mighty AOLserver, the open source, multithreaded HTTP server that powers **http://www.aol.com**. AOLserver provides a rich Tcl API, server-side processing of custom tags via AOLserver Dynamic Pages (ADP), database connection pooling and a cron-like service for running scheduled code in the background. For more about AOLserver, please see the AOLserver home page (http://**www.aolserver.com**).

2.2. Templating system

OpenACS divides responsibility for serving most requests among two or three file types having distinct purposes. The basic split is between a script file that sets up dynamic variables and a markup file that renders them. The script file is a Tcl script having a .tcl extension. The markup file has an .adp extension and looks much like standard HTML. In fact, HTML is valid ADP, providing a gentle slope for programmers and designers who are just getting started with the framework. Because they resemble HTML, ADP files may be edited in standard tools such as Dreamweaver.

The OpenACS templating system supports the ability to work with page fragments via the <include> tag. The same Tcl/ADP file pair idiom operates at this level. Hence if a page contains a tag

```
<include src="motd">
```

the template system will search the current filesystem directory for motd.tcl to set up the dynamic variables and motd.adp for the markup.

More interestingly, the OpenACS templating system supports a master/slave relationship among page fragments. A page fragment that specifies a <master> will have its contents embedded into the master template at a location designated by the <slave> tag in that

template. Master templates are typically used to standardize headers and footers across a site. A nice property of the master/slave system is that the master template provides a bird's eye view of the entire page in a single file. And of course, the same Tcl/ADP file pair idiom applies to master templates as well.

Before we get too far down the two-file path it's worth pointing out a third type of file that also participates in serving most requests: the query file. Query files have an .xql extension and contain SQL that is specific to the template. The function of these files will be explored further in the next section.

2.3. Database API

The OpenACS database API is designed to maximize Tcl integration with the database, allowing data to pass back and forth as simply and naturally as possible. Let's dive in and consider a code example:

```
set title "Late Night With Conan O'Brien"

db_foreach get_matches {
    select description, tvchannel, when_start, when_stop
    from xmltv_programmes
    where title = :title
} {
    do_something_with $title $description $tvchannel
    do_something_else_with $when_start $when_stop
}
```

This code block loops through a tv listings database looking for airings of Late Night with Conan O'Brien. The `db_foreach` command runs a loop, just like Tcl's `foreach` command, but also fetches column variables for each row result, sets them in the code block environment and finally executes the code block. Note the appearance of `:title` in the SQL query, which is the second argument to `db_foreach`. The colon (:) syntax serves to pass the value of the Tcl variable `title` to the database. This mechanism automatically handles any quoting that may be necessary (in this case the single quote in Conan O'Brien's last name), an added security feature.

The first argument to `db_foreach`, `get_matches`, is a required query name. While the above example specifices a SQL query for compactness, OpenACS scripts typically specify their queries by name, leaving the SQL block empty. Query names are used to look up SQL statements in special query files having an .xql extension. These sit in the same directory next to the .tcl and .adp files, and themselves come in several flavors. For the "motd" page fragment, standard SQL statements are places in motd.xql, Postgres-specific statements are places in motd-postgresql.xql and Oracle-specific statements are placed in motd-oracle.xql. This arrangement provides just enough database abstraction to allow reuse of Tcl scripts with multiple RDBMSes, while stopping short of attempting much larger and more difficult problems such as object-relational mapping.

The database API encompasses a number of commands that work similarly, including `db_string`, `db_1row`, `db_list`, `db_list_of_lists`, `db_multirow`, etc., all

of them naturally mix with Tcl, dramatically simplifying the development of Web-database applications.

The database API functions also accept an optional database name argument, to allow applications to connect to multiple databases from the same Tcl script, if necessary.

More information: http://openacs.org/api-doc/proc-search?query_string=db_&search_type=All+matches&name_weight=5&param_weight=3&doc_weight=2

2.4. Declarative programming

Among other Web development frameworks, OpenACS is still unique, powerful and simple, and that's based on the programming advantages created within the OpenACS, such as *declarative programming,* which is understood as the opposite of writing the logic of the program using the *normal procedural programming*, instead use an *special declarative syntax* to reflect how the program will act, based on a possible entry, but not relied only on that. A positive effect of writing in a *declarative syntax* is that the programs usually are more robust in its behavior and more readable by other developers, since the declarative syntax is standard and always more ordered.

2.4.1. Form processing, *ad_form*

As an example, for *web form management*, OpenACS has `ad_form`. This procedure implements a high-level, declarative syntax for the generation and handling of HTML forms. It includes special syntax for the handling of forms tied to database entries, including the automatic generation and handling of primary keys generated from sequences. You can declare code blocks to be executed when the form is submitted, new data is to be added, or existing data modified.

```
ad_form -name form_name -export {foo {bar none}} -form {

        my_table_key:key(my_table_sequence)

        {value:text(textarea)              {label "Enter text"}
                                            {html {rows 4 cols 50}}}
    } -select_query {
        select value from my_table where my_table_key = :my_table_key
    } -validate {
        {value
         {[string length $value] >= 3}
         "\"value\" must be a string containing three or more
characters"
        }
    } -new_data {
        db_dml do_insert "
            insert into my_table
              (my_table_key, value)
            values
              (:key, :value)"
    } -edit_data {
        db_dml do_update "
            update my_table
            set value = :value
            where my_table_key = :key"
    } -after_submit {
        ad_returnredirect "somewhere"
        ad_script_abort
    }
```

In this example, `ad_form` will first check to see if `my_table_key` was passed to the script. If not, the database will be called to generate a new key value from `my_table_sequence`. If defined, the query defined by `-select_query` will be used to fill the form elements with existing data.

On submission, the validation block checks that the user has entered at least three characters into the textarea. If the validation check fails the `value` element will be tagged with the error message, which will be displayed in the form when it is rendered. If the validation check returns true, one of the `new_data` or `edit_data` code blocks will be executed depending on whether or not `my_table_key` was defined during the initial request. `my_table_key` is passed as a hidden form variable and is signed and verified.

This example illustrates how to declare a form, define validation and define a set of actions to be taken on standard events.

More information about `ad_form` can be found here: http://openacs.org/api-doc/proc-view?proc=ad%5fform, from which part of this sub section has been taken.

2.4.2. *List builder*

The list-builder is used create sophisticated table-like reports with many popular capabilities, here is an example.

```
template::list::create \
    -name packages \
    -multirow packages \
    -elements {
        instance_name {
            label {Service}
        }
        www {
            label "Pages"
            link_url_col url
            link_html { title "Visit service pages" }
            display_template {<if @packages.url@ not nil>Pages</if>}
        }
        admin {
            label "Administration"
            link_url_col admin_url
            link_html { title "Service administration" }
            display_template {<if @packages.admin_url@ not
nil>Administration</if>}
        }
        sitewide_admin {
            label "Site-Wide Admin"
            link_url_col sitewide_admin_url
            link_html { title "Service administration" }
            display_template {<if @packages.sitewide_admin_url@ not
nil>Administration</if>}
            hide_p {[ad_decode $swadmin_p 1 0 1]}
        }
        parameters {
            label "Parameters"
            link_url_col param_url
            link_html { title "Service parameters" }
            display_template {<if @packages.param_url@ not
nil>Parameters</if>}
        }
    }

(taken from packages/acs-admin/lib/service-parameters.tcl)
```

For this example you see first the `-name` of the list-builder. Then the declaration of the `-multirow` name used for populating the report with data, which is usually extracted from the database. Then the `-elements` (columns) to be displayed in the report. Each `element` is defined as a name of the variable that is set at the `multirow` in the case that the variable will be used as data to be displayed in that column, like `instance_name`. Also the `element` name can be an arbitrary name in the case that is used the parameter `display_template`, which is used to include HTML tags or ADP logic when displaying data passed by the `multirow`. For each element you always define `label` which is the *title* of the column, and more special parameters such as:

`link_url_col` that expect a variable name which must be set at the `multirow` that contains a link for automatically display the data of that column as a link.

The list builder has many more important features like order by column, filters, bulk actions, pagination, etc.

And this is the result seen in the browser produce by the list-builder example:

Service Administration

Service	Pages	Administration	Parameters
API Browser	Pages		Parameters
Attachments			Parameters
Authentication			Parameters
Automated Testing	Pages	Administration	Parameters
Bootstrap Installer			Parameters
Bulk Mail Portlet			Parameters
Calendar Portlet			Parameters
Categories	Pages		
Content Repository	Pages		Parameters
Developer Support	Pages	Administration	Parameters
Documentation	Pages		
dotLRN Bulk Mail Applet	Pages	Administration	
dotLRN Calendar Applet			Parameters
dotLRN File Storage Applet	Pages		Parameters
dotLRN Forums Applet	Pages		Parameters
dotLRN Portlet			Parameters
Datos estáticos (HTML)	Pages		Parameters
Events			Parameters
FAQ Portlet			Parameters
File Storage Portlet			Parameters
Forums Portlet			Parameters
Kernel			Parameters
Localization	Pages	Administration	Parameters

2.4.3. *Upgrades*

Since OpenACS is a modular system consisting on independent packages, each package has its own version and can require upgrade scripts when updating the package in a given OpenACS installation. This is a simple example of a declarative upgrade syntax:

```
ad_proc -public mypackage::apm::after_upgrade {
    {-from_version_name:required}
    {-to_version_name:required}
} {
     apm_upgrade_logic \
        -from_version_name $from_version_name \
        -to_version_name $to_version_name \
        -spec {
            2.0.3 2.1.0 {
            ….upgrade code here….
             }
            2.1.0 2.1.1 {
            …. More upgrade code here….
             }
            }
}
```

3. Advanced infrastructure

3.1. Serving files: packages, instances, site-map, request processor

Like other Web environments OpenACS can serve familiar file types such as .html and .gif files from a document root. The OpenACS standard document root is $OACS_HOME/www. Put a file at $OACS_HOME/www/hello.html and it will appear at http://yourserver.example.com/hello.html.

OpenACS can also run scripts that set up variables and display them in HTML-like templates, and also embed templates within other templates via include and master/slave tags. These topics are covered in the **Template system** section above. In the sections below we explore OpenACS more advanced mechanisms for serving files.

3.1.1. Packages

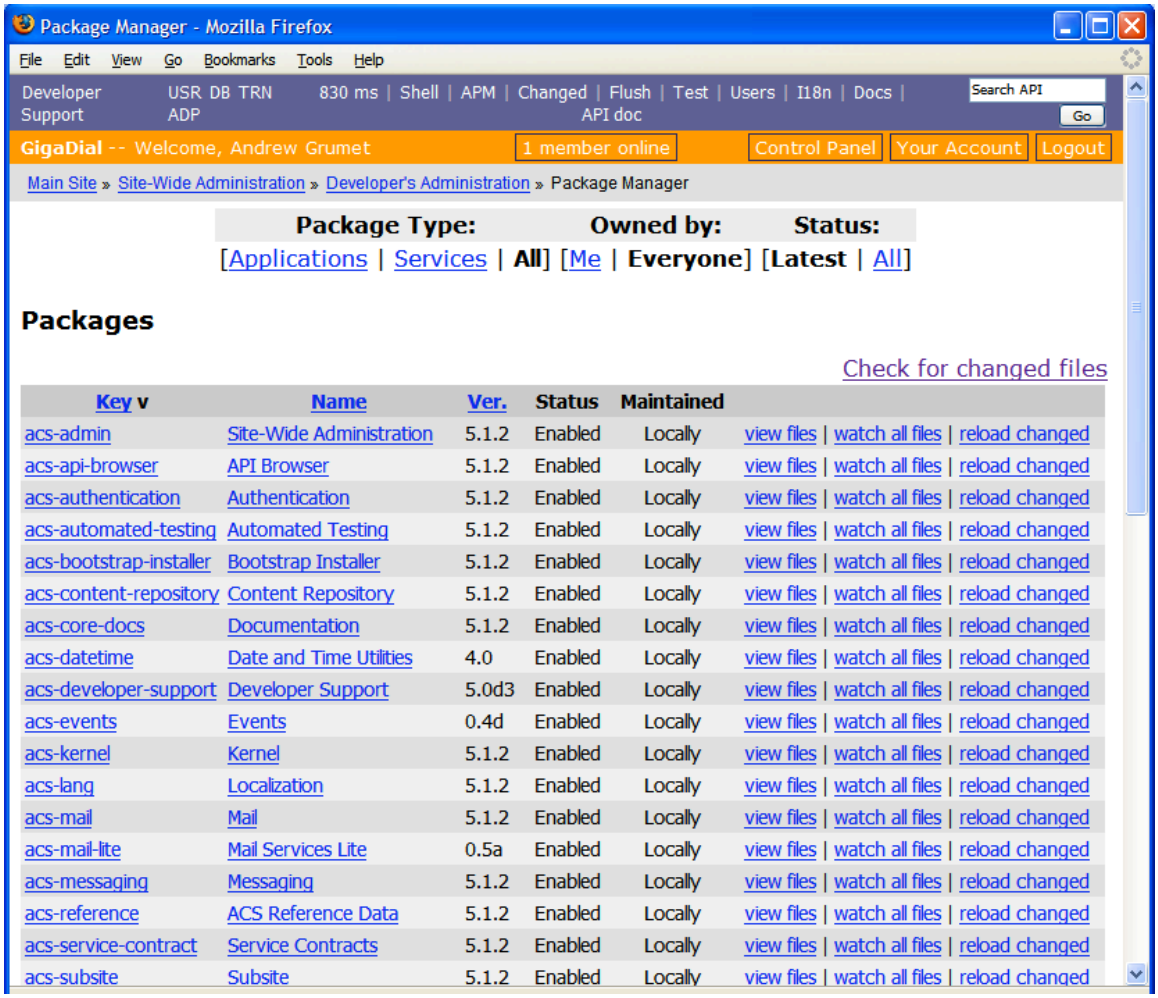
OpenACS is modularized into a set of packages that can be found in the $OACS_HOME/packages subdirectory. Each package may contain SQL scripts, Tcl libraries and visible pages, as illustrated in the abbreviated directory layout below:

```
$OACS_HOME/
    packages/
        acs-admin/            # Core package.
        acs-api-browser/      # Core package.
        ...
        forums/               # Forums package.
            catalog/          # i18n message catalogs.
            forums.info       # Package specification file.
            lib/              # Re-usable tcl/adp templates.
            sql/              # Data model scripts.
            tcl/              # Tcl library.
            www/              # Package document root.
                forum-view.tcl
                forum-view.adp
        ...
    www/                      # Default document root.
```

This example draws attention to the forums package, one of dozens of application packages available for use with OpenACS. Other available packages include a Web-based files storage system (which also is WebDAV-enabled), calendaring, blog authoring, assessment, news aggregation, wikis, photo galleries, RSS support, XML-RPC, SOAP support and many more. A full list of packages can be browsed at http://cvs.openacs.org/cvs/openacs-4/packages/.

Packages are managed with the OpenACS package manager, which handles upgrades and tracks versions, dependencies and files much like Linux package managers do.

A view of the Package Manager:



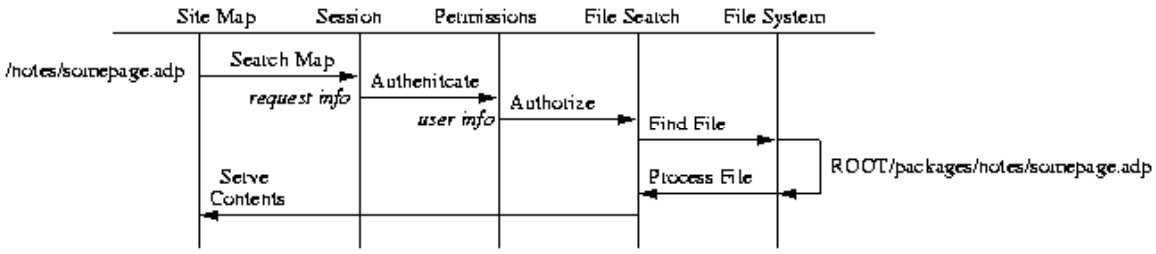
3.1.2. Site map, package instances, request processor

Each package can have its own document root that functions like the default document root at $OACS_HOME/www. The document root for the forums package is located at $OACS_HOME/packages/forums/www, as illustrated in the abbreviated directory layout above.

Package document roots are mapped to visible URLs through a set of database tables, configuration data, libraries and administration Web pages known collectively as the **site map**. Using the site map we can map $OACS_HOME/packages/forums/www to, for example, http://yourserver.example.com/tclers-forums/. But it gets more interesting, because the site map allows for re-use of packages at multiple URLs. Hence we can host a discussion forum for C programmers by adding a new site map entry that maps the forums package to http://yourserver.example.com/cprogrammers-forums/.

These mappings are referred to in OpenACS-speak as "package instances". As the terminology hints, the mapping to /tclers-forums has distinct configuration data from the mapping to /cprogrammers-forums. Hence the /tclers-forums instance might contain a Tcl Forum, a Tk Forum and an AOLserver Forum, while the /cprogrammers-forums instance contains a Small and Fast Forum and a Some Compilation Required Forum. Because package instances are OpenACS objects, they can be have different permission settings, so that some users may be able to read and post to the /tclers-forums but not the /cprogrammers-forums, and vice-versa. The OpenACS object system will be covered in more detail below.

Before doing that, let's take a brief diversion into the mechanics of how files are served, Requests for OpenACS Web pages pass through a Request Processor, which is a global filter and set of Tcl procs that respond to every incoming URL reaching the server. The following diagram summarizes the stages of the request processor assuming a URL request like http://yourserver.example.com/notes/somepage.adp.



The stages are:

1. *Search the Site Map*, to map the URL to the appropriate physical directory in the filesystem.
2. *Authenticate the user*.
3. *Authorize* the possible specific defined permissions that the site node might have.
4. *Process the URL*, search for the appropriate file and server it.

3.2. Object system and services

Deep in OpenACS' design is the notion that one should be able to build common services that are useful across the toolkit. Hence a commenting engine ought work equally well for blog posts as it does for images in photo galleries. Furthermore, any sufficiently interesting piece of data in the system ought to carry basic accounting information with it, such as who created it and timestamps for creation and modification.
The OpenACS object system addresses these requirements by defining a central SQL table called `acs_objects` and giving this table a column for each generic bit of information. Most importantly, `acs_objects` has a primary key column, named `object_id`. This primary key is the anchor on which all other information about an object rests. If the object is a blog post, the post body might live in a separate table `blog_posts` whose primary key, `post_id`, is a reference back to

`acs_objects.object_id`. If the object is an image, it might contain a binary field containing the image bits or alternatively a text field pointing to the physical storage location of the image file, and also an `image_id` primary key that is a reference back to `acs_objects.object_id`. Since each blog post and each image has a row in `acs_objects`, comments on either can be inserted into a table that contains an `on_what_object` column that points back to the `object_id`.

Any data that participates in the OpenACS object system can tell us its title, what kind of object it is, when it was created and by whom. It can also be assigned permissions, commented on, categorized, have files attached to it, and benefit from any other *object-level services* that we can dream up.

The OpenACS object system, site map and instances are the foundation of OpenACS. More information about these can be found at the following URLs:

http://openacs.org/doc/openacs-5-1/request-processor.html
http://openacs.org/doc/openacs-5-1/subsites.html
http://openacs.org/doc/openacs-5-1/packages.html

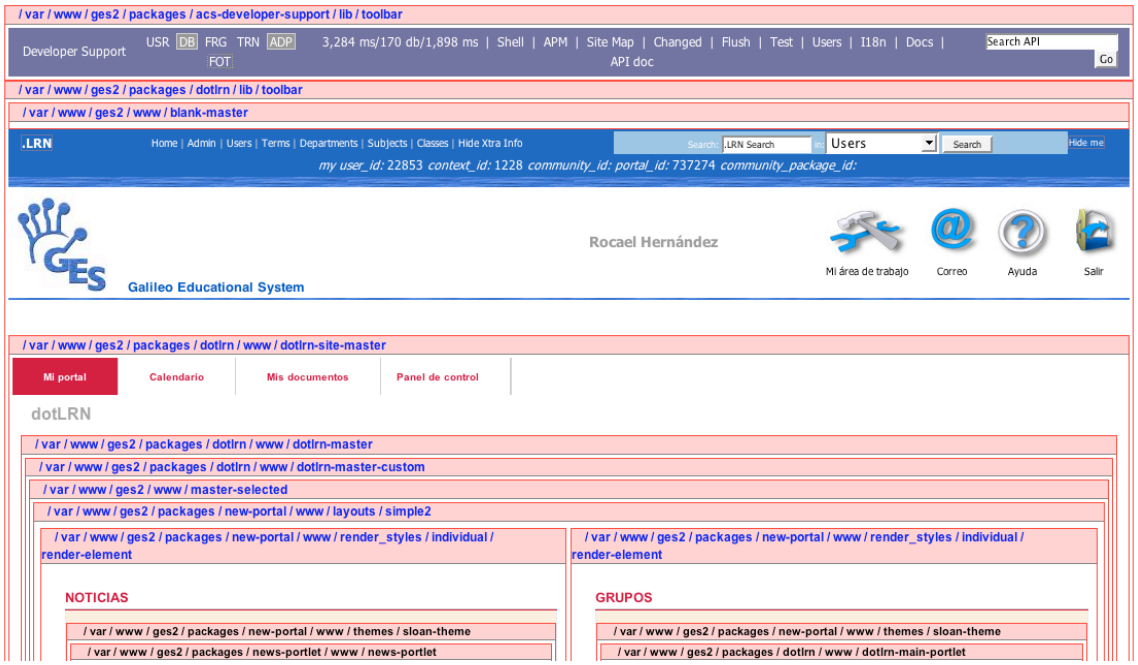
3.3. Developer Support

OpenACS provides a set of developer support tools to improve the development process, debugging, testing, and searching of the API. These tools enhance many of the day to day activities of the developers.

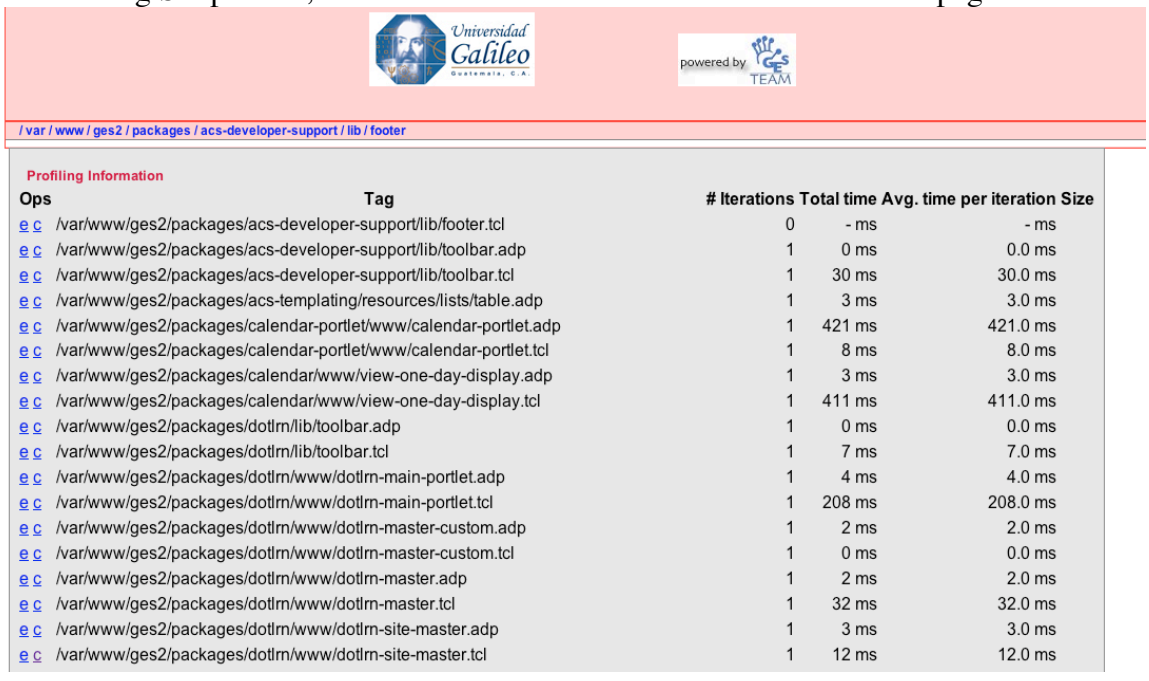
The functionalities that developer support provides are:

1. **Time to serve** a given request. Good for performance problem detection and improvement.
2. **Tracing database calls** involved in a given request, where the involved queries are located, what was the actual query executed and how long did it take to return the data. Useful for improving queries that might be slowing your application performance.
3. **Tracing scripts** involved in serving a given request. Showing the time taken to perform a given script, its location, code, and error that it might bring. Especially important for tuning applications performance.
4. **ADP reveal**, to show in the browser which part of the rendered html belongs to a given script.
5. **User Switching,** as an admin account switch easily to another account to reproduce and test actions, and then simply go back to the administrator account.
6. **OpenACS Shell,** which is a Tcl shell with all the API available within OpenACS, in order to simplify the testing of small pieces of Tcl within your browser.

An screenshot of the ADP Reveal:



The Tracing Script view, information shown at the bottom of each server page:



Ops	Tag	# Iterations	Total time	Avg. time per iteration	Size
e c	/var/www/ges2/packages/acs-developer-support/lib/footer.tcl	0	- ms	- ms	
e c	/var/www/ges2/packages/acs-developer-support/lib/toolbar.adp	1	0 ms	0.0 ms	
e c	/var/www/ges2/packages/acs-developer-support/lib/toolbar.tcl	1	30 ms	30.0 ms	
e c	/var/www/ges2/packages/acs-templating/resources/lists/table.adp	1	3 ms	3.0 ms	
e c	/var/www/ges2/packages/calendar-portlet/www/calendar-portlet.adp	1	421 ms	421.0 ms	
e c	/var/www/ges2/packages/calendar-portlet/www/calendar-portlet.tcl	1	8 ms	8.0 ms	
e c	/var/www/ges2/packages/calendar/www/view-one-day-display.adp	1	3 ms	3.0 ms	
e c	/var/www/ges2/packages/calendar/www/view-one-day-display.tcl	1	411 ms	411.0 ms	
e c	/var/www/ges2/packages/dotlrn/lib/toolbar.adp	1	0 ms	0.0 ms	
e c	/var/www/ges2/packages/dotlrn/lib/toolbar.tcl	1	7 ms	7.0 ms	
e c	/var/www/ges2/packages/dotlrn/www/dotlrn-main-portlet.adp	1	4 ms	4.0 ms	
e c	/var/www/ges2/packages/dotlrn/www/dotlrn-main-portlet.tcl	1	208 ms	208.0 ms	
e c	/var/www/ges2/packages/dotlrn/www/dotlrn-master-custom.adp	1	2 ms	2.0 ms	
e c	/var/www/ges2/packages/dotlrn/www/dotlrn-master-custom.tcl	1	0 ms	0.0 ms	
e c	/var/www/ges2/packages/dotlrn/www/dotlrn-master.adp	1	2 ms	2.0 ms	
e c	/var/www/ges2/packages/dotlrn/www/dotlrn-master.tcl	1	32 ms	32.0 ms	
e c	/var/www/ges2/packages/dotlrn/www/dotlrn-site-master.adp	1	3 ms	3.0 ms	
e c	/var/www/ges2/packages/dotlrn/www/dotlrn-site-master.tcl	1	12 ms	12.0 ms	

OpenACS also provides an implicit mechanism to document any script or procedure that you might create, and then display and search that scripts or procedures, its documented information, expected inputs and outputs, and its code though a Web interface at your own OpenACS installation, like yourserver.com/api-doc, have a look here:
http://openacs.org/api-doc/

Finally, using the package manager, the Web server can be instructed to reload Tcl library files without a restart of the webserver, and keep *watching* them for subsequent changes, which is quite useful for the developers.

3.5. Testing Framework

OpenACS provides a full featured testing framework to create, maintain and run automated test in your applications. The main characteristics of it are:

- Define tests, as smoke, config, database, web, etc. in a per package basis. And with the API provided by the test framework you have a UI to check results, and log events when executing a test case.
- Test your application at the proc level, using specific Test API to define and check if your procs are behaving as expected.
- Test your end-user web scripts using a third party tool, such as tclwebtest or perl::mechanize, to test the end-user pages automatically simulating end user navigation (clicks) through the application and finally check the outpus using the Test API.
- Define procedures that many tests might call, in order to automate similar activities that need to be done before running a test case.
- Rollback the generated data after executing a test case.
- Rollback code section, to perform actions that will rollback the actions done by running a test case, specially designed for those situations where you cannot encapsulate the database calls to rollback, like when you are calling the scripts with an automated tool.
- Run tests by installation, package or by choosing an specific test case.

3.6. Internationalization

OpenACS provide a facility to internationalize its user interface texts (not the data stored) to any desired language, right now OpenACS is translated to more than 20 languages. The end users can change the language user interface by simply selecting it. Each OpenACS package can be internationalized separately, maintaining a message keys catalog, which consist in a specific XML DTD where each possible user interface message is stored as a message key, and then translated to a given language. Although all the data is stored in the xml file, everything is also stored in the database, for performance reasons and to facilitate the edition of the message keys OpenACS provides a simple UI for translation of message key. And exists an official translation server for the community at: translate.openacs.org.

3.7. Callbacks

The callbacks are a simple method to execute procedures stored in each of the possible installed packages that a given OpenACS installation might have. The objective is to give the core applications to invoke in non-core packages that may or may not be installed, but without cluttering core with code that belongs to other packages, making possible to have independence among packages.

The architecture work as

```
-> Core proc for removing user
   -> Invoke callbacks based on what's installed
      -> Package A logic for removing user
      -> Package B logic for removing user
      -> Package C logic for removing user
      ...
   -> Core logic for removing user
```

as opposed to this architecture

```
-> Package A proc for removing user
   -> Package A logic for removing user
   -> Call core proc for removing user

-> Package B proc for removing user
   -> Package B logic for removing user
   -> Call core proc for removing user
```

Callback implementations would be declared like this:

```
ad_proc -callback module::op -implementation implname  { ... }
```

Where `ad_proc` is a wrapper of the normal TCL proc, which mainly gives auto-documentation structure for any procedure.

Core uses tcl introspection to find which callbacks exist and invokes them. eg

```
foreach proc [info procs ::callback::module::op::impl::*] {
  $proc $args
}
```

To invoke a callback you do this

```
callback [ -catch ] [ -impl impl ] callback [ args... ]
```

The callbacks is a great improvement in order to keep simple yet separated, coherent and modular a web framework that constantly evolves and grows. The larger goal is to promote reuse of standard pages and functions by removing the need to create per-package versions of these.

4. Domain level tools

4.1. Content Repository

The Content Repository (CR) is a central service application that can be used by other applications to manage its content. The developer must define and declare the interactions with the content repository. The main features that the content repository are:

- CR is capable to store any kind of information (files, data, text).

- Define application specific data structures that will use the CR features and services within the application context.
- Revision control, which means that every addition and subsequent changes of an application is registered, so the versions are kept and can be roll-backed.
- Ability to handle hierarchies, and folder structures, and inherit its properties.
- Easily interact with the CR and its specific defined structures through a well-defined API, in order to add data or present it.
- Also handles publish states for content that might need it.
- Identify content through a content type definition.
- Associate the content with external applications.
- Store specific templates to use when displaying specific content types.
- Create relations between content repository items or external database objects.
- Embedded search facility to automatically expose the content to the search engine the system might be using.

The process for an application to use the content repository goes as:

1. Define the database structure your application needs that will use some of the CR features.
2. Use the standardized API (to add, edit, delete, present) to create scripts for your CR-enabled-application.

The CR logic is stored in database functions and triggers, but it has a Tcl API that glue all the involved database calls and makes straightforward for the developers to create applications that use the CR.

4.2. Results: Vertical Applications

Within OpenACS a vertical application means a set packages that interact together to provide a specific domain set of functionalities.
The following are good examples of vertical applications completely running in OpenACS and using Tcl as the programming language:

4.2.1. Project Manager

Full featured Project manager, consist in a set of functionalities that can be summarized as: track tasks, estimates and actual progress for a project. It consist in more that 5 packages. Features that are part f it: project creation, customers management, task assignments, logs, comments, create processes, rates, etc.

4.2.2. Assessment Tool

Extremely powerful application to create several types of assessments such as online self-tests, surveys, tests and gathering of information in general. The main features of this application are: IMS-QTI support for import and export of assessments, branching, sections, several types of questions and many more are part of it. More information at: http://cvs.openacs.org/cvs/*checkout*/openacs-4/packages/assessment/www/doc/index.html?rev=1.6

4.2.3. Communities of Practice

Communities of Practice within OpenACS are represented by a set of tools available through all the website to interact with any kind of applications. The main objective is to give the power to the end user describe in many forms any object at the system and make relations among them. Those tools are:
- Rate
- Comment
- Categorize
- Link
- Search

And those tools can be used in all the applications that the end user might have available, such as forums, file storage, etc. For example, you can *categorize* a given *file*, and then *link* it to a *forum thread*, which you are also *rating* and *commenting*. Then a third user will *search* for given topic, and the first result will be the *file*, and when he looks at the *file*, he'll see as well all its related *objects*, such as the *forum thread*.

4.2.4. .LRN and the E-LANE project

.LRN (pronounced dot-learn, www.dotlrn.org) is a full featured LMS with extra community building capabilities, uses packages available for OpenACS and integrate other vertical applications described here. Its focus is to help the creation of learning and research communities, and not just server as an LMS.

The E-LANE Project (European-Latin America New Education, www.e-lane.org) is using .LRN as its foundation technology to promote and demonstrate e-learning within this geographical area, the demonstration is based on three fundamental factors: the learning methodology, the content development, and the technological platform, the three of them brought together to use it in real scenarios for further improvement of .LRN.

As a contribution to the .LRN community, E-LANE has developed among other applications the user-tracking package, which is mainly focused to "Analyze users behavior and syndicate it by object type".
This package is basically using Tcl to create the UI and the administrative actions, and glue them with awstats (http://awstats.sourceforge.net/), which in this case is the log analyzer used to parse the AOLserver request logs, and those logs contains a set of special keys written on each http request logged, which is the base to generate specific request reports for a given user and/or object type.

5. Conclusions

OpenACS is a complete web development framework, which is based in Tcl, and uses tools like tdom for xml parsing, tclwebtest for creating web test, etc. Also provides a set of functionalities to enhance the web development as mentioned in this paper. Tcl has shown to be an extremely flexible and powerful scripting language to create web based applications, and OpenACS becomes an excellent option to create a web community site of any kind or size.

Further information, this paper and the presentation can be found online at
http://openacs.org/tcltk05