ITCC Dashboard Documentation

Authored by Ryan Nixon

University of Alaska Anchorage

4-13-2011

Overview 1

Project Description 2

System Design 3

User Interface Elements 4

Unified Search 4

Module List 5

Subversion 7

Working Copies 7

Deployment 7

Active Objects 11

Provided Information 11

Using Within A Module 11

Logging 13

Module Design 15

Configuration File 15

Module Attributes 16

Description 16

Category 16

Leftnav 16

Page 17

userAttribute/classAttribute 17

userButton/classButton 17

Checks 18

PHP Class 18

Smarty Templates 20

# Overview

The ITCC Dashboard was created as a platform for IT Services Call Center technicians to quickly respond to support requests. It is set up as an easily extensible web application intended to provide access to multiple account directories from many different types of sources such as LDAP directories and other SQL-based software applications.

The typical usage scenario involves looking up an account through the [search interface](#_Unified_Search) which sets an [Active Object](#_Active_Objects) within the application. From there, the technician or other user will take actions upon that Active Object using the provided interface functions or through the application’s modular functionality.

Developers seeking to extend the ITCC Dashboard with new features may use [modules](#_Module_Design) to hook in to the application in specific ways. These hooks include entries in the left navigation bar and account buttons/attributes but additional hooks can easily be created with modifications to the core.

# Project Description

With a development timeline of a single school semester, production of the Dashboard needed organization and a standard software development methodology to be applied and held to over the course of the semester. The project had primary client, Mark Weisman the Call Center Supervisor, and a secondary client, Jim Weller the Web Team Supervisor. Both clients were within easy reach but due to my work hours and scheduling difficulties were not always available. I also had the problem with Mark Weisman not being able to describe what he wanted in the application. Both of these qualities made me avoid the standard waterfall method and look towards prototyping or agile methodologies. In the end after discussing it with Jim Weller we decided upon the Spiral methodology with a 2 week turnaround time.

The application itself was a complete overhaul of a legacy dashboard that was extremely long in the tooth. The legacy system amounted to a single 4900 line if statement in one file that POST’ed back upon itself to maintain the application state. It also made multiple database connections per request and did not manage them properly and required technicians to lookup a user each time they wanted to take action upon them.

The first thing I decided to do when creating the new system was to take all of the database connections and apply the Singleton design pattern so there would only be a single connection per database that was maintained through the lifetime of the request. This can be seen in the base LDAPDatabase.php class of which is subclassed by the various specific directories. In designing the Active Object architecture to allow the application to remember the most recent lookup, I used the Factory design pattern. I also used this with modules, maintaining Loader classes for each in the Libraries directory. Finally, CodeIgniter defines its own pattern for providing access throughout the system to the same functionality. I implemented this same pattern in the ModuleClass and module subsystem, where ModuleClass’s base class defines behavior and the subclasses call that behavior. While a first impression might be that this pollutes the class with potential method naming conflicts, I did this so that in the future the ModuleClass base class could implement public methods to be called directly from a client’s browser through the URL, making module implementations of that functionality not required.

The end result of the project architecture showing the general overview of the class organization is below. Note that due to the exposed functionality throughout the system that CodeIgniter encourages, many of these classes are able to call each other. This causes tight coupling within the core code base, an undesired side effect.



# System Design

The ITCC Dashboard was developed on a hybrid foundation of EllisLab’s [CodeIgniter](http://codeigniter.com/) PHP framework and the [Smarty](http://www.smarty.net/) template system. When working on the internals of the dashboard an understanding of these two systems is not required but will be helpful.

At first glance the file structure may look unwieldy, but there is a defined organizational scheme that will be familiar to any developers who have used the Model-View-Controller design pattern. The root folder of the application contains three subfolders:

* **Application** – Core files containing most of the system logic and CodeIgniter/Smarty frameworks. When making deep changes to the system a developer will want to look in this folder first.
* **Modules** – Provides extensions to the application that are not necessarily part of the core. Most functionality should go here, with the core files supporting it.
* **Public** – The public-facing side of the application where the web server interacts. This folder contains client-side code such as CSS and JavaScript as well as the CodeIgniter initiation file that defines the environment variables for the application.

Inside of the Application folder there are multiple subfolders that serve as the core of the Dashboard:

* **Classes** – Support files for the application libraries. If you are making large-scale changes to Active Objects, Modules, or all LDAP directories you should look here.
* **Config** – Contains multiple CodeIgniter configuration files. Auth.php houses most of the usernames and passwords used site-wide. Config.php has specific CodeIgniter initialization settings. Database.php has the settings for all of the Microsoft SQL server connections (Blackboard, Commonspot, etc.)
* **Controllers** – When browsing through the Dashboard, the files in Controllers represent the URL paths to each page. For example, viewing a user account will call the info() function within user.php. These functions generally drive the application but only serve to connect the data to the views through function calls, not to directly influence the data.
* **Errors** – Contains fallback PHP files for any unhandled errors caught by the application. The 404 page is also located here.
* **Helpers** – Log\_helper.php and render\_helper.php are located here, to assist the application with creating log entries and rendering Smarty templates, respectfully.
* **Libraries** – Core application methods use the libraries located here for data-based actions. This includes getting a list of all modules, connecting to an LDAP directory, initializing the Smarty parser, and working with Active Objects.
* **Models** – Contains the SQL queries used for interacting with Microsoft SQL server databases. If you are adding or changing the way the Dashboard interacts with these databases then you should look here.
* **Views** – All Smarty templates not belonging to a module are in this folder. The most important template is base.tpl which most all of the Smarty templates extend with their own view logic.

# User Interface Elements

The new Dashboard interface uses UAA’s [DenaliView](http://www.uaa.alaska.edu/workshop/) theme as a base template for interaction. This has major advantages as most users will already be familiar with navigating the UAA CMS and will not have a strong learning curve when adopting the Dashboard workflow. The following section covers some of the new elements introduced in the Dashboard that do not exist in DenaliView or the legacy Call Center Dashboard.

## Unified Search

When searching for a specific account it helps to see exactly what directories are being searched. To that end, the Dashboard separates all database searches into their own sections. When you use the Search in the top-right of the interface, you will see something like the following appear:



The default search will attempt to match accounts in all possible ways and can take some time as a result. To minimize this time, the search interface uses AJAX calls so that the page loads as fast as possible and query results show as they come in. If you would like to speed up the queries even further you may choose a more specific search from the dropdown to the left of the search box.

For User accounts you can search by **UAID**, **Username**, & **First and/or Last name**. When drilling down to these levels, the Blackboard search will not be performed to get an additional speed increase. If possible, search by UAID or Username and not First/Last name. Due to the structure of the LDAP directory, First/Last name must run separate searches for each word provided in the search field and combine them into one result set. This causes the First/Last name search to be more general but also to be much slower than the other queries.

For Course accounts you can search by **CRN**, **Class Description** and **Instructor**. If you would like to see classes that a student is a part of, search for that student’s account and view the Blackboard section on their information page. Searching by these items will skip the directory search in order to gain speed increases.

\*\* NOTE: When selecting an account keep in mind that the item you click on is converted to a UAID on the information page. This may cause unexpected results if you choose an account with a malformed UAID or an account only existing in one directory. \*\*

## Module List

In the left navigation there are two items always present at the end of the list of pages, Administration and [Log](#_Logging). Clicking on Administration will take you to the Module List, where you can see an overview of all modules installed in the application. This list includes modules that are disabled (shown by the check mark to the left of the module name) and a description of what functionality the module adds to the system. Below each description is a button allowing a user to view all log entries that that module has created.



Installation

The Dashboard uses multiple frameworks to support its functionality. To install the application onto a system you must make sure you meet all of the dependencies.

1. Download the most recent version of [CodeIgniter](http://codeigniter.com/) 2.x from EllisLabs. After unzipping the downloaded archive, move the system folder into a location on PHP’s include path.
2. In addition to CodeIgniter you will also need [Smarty](http://www.smarty.net/) 3.x installed on your PHP include path. Download the above archive, extract it, and place it in the same folder you have installed CodeIgniter’s system folder.
3. Once the dependencies have been installed you can follow the [Subversion Working Copies](#_Working_Copies) chapter to get a copy of the Dashboard application.
4. CodeIgniter’s default implementation of Microsoft SQL Server libraries is poorly supported. After checking out your working copy there will be a sqlsrv folder present in the base directory. The sqlsrv folder contains a newer version of the database driver from [Kaweb](http://www.kaweb.co.uk/uncategorized/mssql-server-2005-and-codeigniter/) and modified for CodeIgniter 2.x. Take this folder and copy it to CodeIgniter’s system/database/drivers/ folder.

# Subversion

ITCC uses Subversion with tagging for managing revisions. Versions are tracked in the format “[Major].[Minor].[Patch]” (ie. 1.0 for a major release, 1.1 for a new feature, 1.1.1 for a patch to an existing feature).

## Working Copies

The most recent version of the application is stored in the /trunk/ folder of the repository. When checking out a working copy, take the entire folder and all subfolders. If you are checking out to a different location than the normal development folder you will need to update the path to your working copy. The path is located in the /public/index.php file where the $application\_folder variable is set for the “development” environment variable. If you are on a different machine than the normal development location you will also need to update the location to CodeIgniter’s core files. The definition for that is above the $application\_folder variable where $system\_path is set.

## Deployment

If you are ready to push a change out to production, 4 steps need to be taken:

1. Commit the changes that you have made to the development folder. When committing, provide a detailed note of exactly what changes were made (similar to a changelog entry) and ensure that all files are included. If you are working in a shared environment, make sure that you are not committing anyone else’s changes before they are finished with their work.
 
2. If you are hitting a milestone in the program development process, make sure to tag it in Subversion with a new version number. Tags are located in the repository within the /tags folder. Before tagging, go into the /public/index.php file and find the line defining a VERSION constant. Set this to the version you are about to tag and commit the change. The line REVISION below it will be automatically updated when you commit.


3. Go to the production folder for the application. If making a small point upgrade you can safely export your newly committed tag into this folder. Major releases may be served better by emptying the folder and exporting a fresh copy of the application. Use your best judgment when deciding which method to take. Do not checkout into the production folder. Checking out creates a working copy of the application and leaves many Subversion files in the directory structure. Using export results in a clean version of the application.

4. In the /public/index.php file of the newly exported files is a line defining an ENVIRONMENT constant. Change this line from ‘development’ to ‘testing’ or ‘production’ as applicable. This variable determines how error messages are handled within the program as well as managing redirects and imports. If it is not set correctly then all redirects will point to the development copy of the application.

# Active Objects

In order to facilitate clean and efficient code, Active Objects are created to minimize the amount of LDAP/Blackboard lookups for accounts. These objects are provided system-wide and provide basic information about the last-viewed user account or course.

When an active object is set, a box will appear above the Search area for the duration of the current session. It will contain basic identity information for the active account. The icon to the right of the information represents the database used to look up the information. In the above example the UA Unified Directory found the account, and upon every page load the basic information for that account will be reloaded and provided to the currently running code. If the user would like to see more information about the account they can simply click the box and it will return to the account in question.

## Provided Information

An Active Object gathers multiple pieces of information depending on what account type it is.

**For User Accounts:**

* **id/uaid** – The unique identifier for the account
* **url** – Return URL for the account. Changes depending on the lookup method (typically username or UAID)
* **name** – The full name of the account
* **firstName & lastName**
* **uid** – The username of the account
* **dn** – The full distinguished name of the source LDAP account
* **db** – Which LDAP account was used to look up the information
* **IMG\_SRC** = URL to a fallback image for generic user accounts

**For Course Accounts:**

* **id/pk1** – The unique course code
* **url** – Return URL for the account
* **name** – The name of the course
* **crn** – The course reference number

## Using Within A Module

Modules will almost always want to use Active Objects to reference the currently active account. To get the current Active Object you would typically use the following:

$ao = $this->activeObject(ActiveObject::USER);

The $ao variable will now contain an Active Object to be used in the function. You can require a specific type by setting the first parameter to either ActiveObject::USER or ActiveObject::COURSE for users and course accounts, respectively. If this check fails or there is no current Active Object present, an ActiveObjectRequired exception will be thrown. Module developers can choose to catch and handle this exception themselves or allow it to cascade up the stack where the application core will generate a default error message for it.

# Logging

The ITCC dashboard has a logging facility built in. Using it for permanent account actions is highly recommended. Logging errors is also suggested as it allows users and developers of the application to see what it is doing and what is going wrong.

You may view all log entries by clicking on the Log entry in the left navigation. This will display a paginated list of all log entries sorted over time as well as the module and user that made the entry and any associated notes. If the entry was made as an error, it will display in red.



If you would like to view specific log entries for a module, you can filter the log by going to the module’s entry on the [Module List](#_Module_List) and selecting “View Log Entries”.

CodeIgniter helper functions are provided to make log entries. Entry methods take Module, Action, and Entry strings. Module strings define where the log entry is coming from. Leaving it blank or null will make the entry default to the application itself. Action denotes what section of the code is running (or rather what action that code is attempting to do). Entry contains additional helpful data such as what account is being modified. Depending on whether you are making a note or an error you should choose between the following:

**If you are logging from the application core:**

log\_entry($module, $action, $entry)

log\_error($module, $action, $entry)

**If you are logging from within a module’s PHP file:**

$this->log($action, $entry)

$this->error($action, $entry)

Within modules, the Module string is automatically provided.

\*\*Note: If development mode is enabled, additions to the error log will automatically print out an error message to the screen. In production these messages will not appear. \*\*

# Module Design

The ITCC Dashboard was specifically designed for extended functionality through the use of modules that set “hooks” into the user interface. This modular design was created with the intention for ITS employees to easily extend the interface without having to deal with editing the core application files and logic.

Modules are located in their own subfolders in /modules/. They must contain two files, config.xml and a PHP file named the same as the folder name. Config.xml defines the entry-points into the program while the PHP file contains the application logic that corresponds to those hooks.

## Configuration File

A fully-featured module’s config.xml file may look like the following:

<module name="User Account" version="1.0" enabled="true">

 <description>Status and actions specific to a user account including password management.</description>

 <category>User</category>

 <leftnav>false</leftnav>

 <page name='Passwords' function='viewPasswords' />

 <userAttribute name="Active" function='isActive'>

 <check>hasSunOneAccount</check>

 </userAttribute>

 <userAttribute name='Single Mail Delivery Option' function='hasDeliveryOption' description='Checks to see if only one mailDeliveryOption is present in SunOne'>

 <check>hasSunOneAccount</check>

 </userAttribute>

 <userAttribute name='Default Password' function='isDefaultPassword' />

 <userButton name="Activate" function="activate">

 <check>hasSunOneAccount</check>

 <check>!isActive</check>

 </userButton>

 <userButton name="Deactivate" function="deactivate">

 <check>hasSunOneAccount</check>

 <check>isActive</check>

 </userButton>

 <userButton name='Remove Delivery Option' function='removeDeliveryOption'>

 <check>hasSunOneAccount</check>

 <check>hasDeliveryOption</check>

 </userButton>

 <userButton name='Add Delivery Option' function='addDeliveryOption'>

 <check>hasSunOneAccount</check>

 <check>!hasDeliveryOption</check>

 </userButton>

 <userButton name='Reset Password' function='resetPassword'>

 <check>!isDefaultPassword</check>

 </userButton>

</module>

### Module Attributes

**Required**

* **name** - The name of the module as it will be shown in the [Module List](#_Module_List). If ‘leftnav’ is set to true then this will also be the name of the module in the left navigation.
* **version** - The current revision of this module.
* **enabled** - Determines if the module should appear within the application.
	+ \*\* Note - The application’s configuration will still be parsed, so keeping a valid config.xml file is important \*\*

### Description

**Required**

A plain text summary of the functionality provided by the module. This will be displayed with other module descriptions in the [Module List](#_Module_List). A verbose description is extremely helpful since future module authors will use the description as a jumping-off point into the code base.



### Category

**Optional**

A second descriptor for the module to support the Description tag. Modules will be grouped together by their category in both the left navigation and the [Module List](#_Module_List). If a category is not provided, a default value of “Uncategorized” will be provided for the module.

### Leftnav

**Optional**

If true, this places an entry point into your module into the left navigation. It uses the name of the module to represent it in the list and will link to the index() function within your module’s PHP file. The Leftnav option does not need to be set if custom pages are going to be defined.

### Page

**Optional, Multiple**

Defines an entry-point into the program similar to the Leftnav tag. Pages will always be placed within their module’s category. Page tags have two attributes:

* **name** - The string to be displayed in the left navigation.
* **function** - The name of the function in the module’s PHP file to be called when the page entry is clicked.



### userAttribute/classAttribute

**Optional, Multiple, Checks**

User attributes and class attributes appear within the corresponding information pages for their types. They are represented as checkboxes and associate with a boolean-based function within the module’s PHP file. If the function returns true or false then the checkbox will be checked or X’ed out, respectively. If the function returns a string value (perhaps due to an error) then the checkbox will turn into an exclamation mark. When clicked on, the Dashboard will display a dialog with the contents of the returned string. User and class attribute tags have three attributes:

* **name** - The string to be displayed next to the checkbox.
* **function** - The name of the function in the module’s PHP file to be called through AJAX when the information page loads.
* **description** - A long string describing the functionality this attribute provides. This appears when the attribute is hovered over.

### userButton/classButton

**Optional, Multiple, Checks**

User and Class buttons provide one-click actions to be taken on an associated user or course account. Clicking them will link to a function within the module’s PHP file. While workflows using multiple pages may be designed this way, it is a recommended design practice to have the button’s action occur immediately and redirect back to the account page upon completion. User and Class button tags have three attributes:

* **name** - The string to be used for the button text.
* **function** - The name of the function in the module’s PHP file to be called when the button is clicked.
* **description** - A long string describing the functionality this button provides. This appears when the button is hovered over.

### Checks

Some module functionality may only be applicable in certain conditions. To resolve this, Checks may be added to supported tags as children. These Checks are PHP functions within the module that should return boolean values. If any Check returns False, the containing tag will not be processed. You can also add an exclamation mark to the beginning of the check’s definition. Doing this will make the check fail if the called function returns True.

For example, to implement the concept of an account being ‘Active’ you could define the following:

<userAttribute name='Active' function='isActive' description='Account status' />

<userButton name='Activate' function='activate' description='Activate account'>

 <check>!isActive</check>

</userButton>

<userButton name='Deactivate' function='deactivate' description='Deactivate account'>

 <check>isActive</check>

</userButton>

## PHP Class

In addition to the config.xml file, a module must contain a PHP file named the same as the containing folder. This PHP file needs to define a subclass of the ModuleClass class by the name of the PHP file. For example, if you wanted to make a UserAccounts module, you would create a PHP file by the name of UserAccounts.php and place it in the /modules/UserAccounts/ folder. The PHP would look like this:

<?php

class UserAccount extends ModuleClass {

 public function viewAccount() {

 /\*Connect to the current Active Object and render a Smarty template with that information\*/

 }

}

?>

The viewAccount() function could then be called by going to /module/load/UserAccount/viewAccount/ in a web browser.

Note that the PHP class must extend the ModuleClass base class. This class provides many helper methods and defaults for developers and is required for a module to function properly. ModuleClass methods have protected visibility and can be called within a module by using $this->methodName() within the child class. The following methods are provided to assist module creation:

* **session($key, $value = null) -** Allows manipulation of session data to be stored between page loads. If $value is not provided then the method will return the currently stored session value for that key, otherwise it will set the variable to the provided value and return true upon success.
* **module($moduleName) –** Provides an instance of another loaded module. Using this method is discouraged as it couples the application logic between two modules and may cause one to break if the other is modified. Before using this, consider merging the two modules into one.
* **activeObject($type = null) -** Returns the currently set Active Object. A type check can also be performed upon retrieval by passing an Active Object constant into the $type parameter (ie. ActiveObject::USER or ActiveObject::COURSE). If this type check fails or an Active Object is not set, then the method will throw an ActiveObjectRequired exception. See the [Active Object](#_Using_Within_A) section for additional details on using this method.
* **sunone(), federated(), ad(), blackboard() & cms() –** These methods will return instances of the requested directory or database to the module for querying. Each instance has its own behavior specific to the data source.
* **hasSunOneAccount(), hasFederatedAccount(), hasADAccount() –** These methods return boolean values according to if they can find an associated LDAP account for the current Active Object. Results are cached between calls so these methods minimize the number of LDAP queries that are made.
* **log($action, $entry), error($action, $entry)** – Creates an entry in the application log for the current module. See the [Logging](#_Logging) section for additional details.
* **render($templateName = null, array $data = null), renderDefault($data = null), renderError($content = null, $returnURL = null, Exception $e = null)** – Assists modules with rendering stock Smarty templates or their own custom templates. See the [Smarty](#_Smarty_Templates) section for additional details.

# Smarty Templates

To assist with rendering HTML quickly and efficiently, the ITCC Dashboard uses Smarty templates, allowing for the easy reuse of content between pages. The application/views/ folder houses all TPL files that are used by the framework. Three distinct files in this folder are base.tpl, empty.tpl and error.tpl. They contain the following functionality:

* **base.tpl** – Displays the containing HTML for all content on the site. Provides {block} tags to be overridden by sub-templates:
	+ **title** – The title of the page, automatically appended with “IT Services Call Center Dashboard - University of Alaska Anchorage”
	+ **head/scripts/styles/debug** – Additional {block} tags that can be used to add extra content to the <head> tag of the HTML document
	+ **bodyclass** – Allows for customization of the page according to the class attribute of the <body> tag
	+ **splash** – Puts content into the top banner of the main area. Usually an image would be inserted here
	+ **content** – Used by almost all templates to provide content for the page
	+ **analytics** – Adds HTML below all other tags on the page, perfect for Javascript requiring the page to be fully rendered before execution
* **empty.tpl** – Used in many locations, most notably the default render handler. Leaves all defaults in place and only overrides the content block
* **error.tpl** – Used by the error render handler. Will print $content, $e->getMessage(), an anchor leading to $returnURL, and the stack trace of $e if in the development environment, in that order

Using a template within the application is supported by three helper functions and the [ModuleClass](#_PHP_Class) base class. Depending on the context in which you are rendering a template you should choose between the following:

**If you are rendering a template from the application core:**

render($templateName = null, array $data = null)

renderDefault($data = null)

renderError($content = null, $returnURL = null, Exception $e = null)

**If you are rendering a template from within a module:**

$this->render($templateName = null, array $data = null)

$this->renderDefault($data = null)

$this->renderError($content = null, $returnURL = null, Exception $e = null)

The two rendering methods are nearly identical with the exception of the render() method. Calling that method from within a module will render a template within the module’s folder, rather than the /application/views folder.