|  |
| --- |
|  |

|  |  |  |
| --- | --- | --- |
|  |  |  |
|  | |  |
| 12/14/2013 | |  |
| @CMU Mobile Application Design & Development | |  |
| Independent Study Project F13 | |  |
|  | |  |
| Carnegie Mellon University - Australia | |  |
| Authored by: Hsuan-Chih Chuang | |  |
|  | |  |

Table of Contents

1 Executive Summary 3

2 @CMU Web Service Development 3

2.1 The ELGG Social Engine 3

2.1.1 Data Model 4

2.1.2 View 5

2.1.3 Controller 5

2.2 ELGG Directories Structure 5

2.3 ELGG Plugin Development 8

2.3.1 Configuring @CMU for Plugin Development 8

2.3.2 Creating the ELGG Plugin Skeleton 9

2.3.3 Web Service Development 12

3 Mobile Client Application Design 16

3.1 Design Considerations 16

3.2 Application Features 17

3.3 User Interface Design 18

3.4 Web Service Client Development 20

3.4.1 Application Flow 20

3.4.2 Asynchronous Web Service Calls using AJAX 21

3.4.3 Pipelining Web Service Calls 22

4 Limitations and Future Work 23

5 Works Cited 24

# Executive Summary

The Carnegie Mellon University-Australia student social portal is annex to rich and valuable resources to both current and prospective students of the Australian campus. These resources include, but are not limited to, career/internship opportunities as well as seminars and social event announcement. An inherent shortcoming to the current platform, however, is that information on the platform is not readily accessible to students on-the-go, thus making it difficult for student to fully take advantage of the platform resources on a regular basis.

The goal of the independent study project is to evaluate and improve on the usability of the current Carnegie Mellon University-Australia student social portal. A more accessible platform will invariably promote its utilization. Improvements to the platform will take into consideration the philosophies driving the existence of the platform:

* Allowing prospective students to join the CMU-A social community when an offered letter is received.
* Assisting students with pre-departure induction to accommodation and transportation.
* Providing grounds for student IT self-service through detailed computing instructions.
* Connecting prospective and current students with university alumni network.
* Keeping students updated with campus-wide announcements relating to seminars, career opportunities, extracurricular activities, and so on.

This project will be executed in 2 phases:

* Evaluation and improvement of the current platform structure, including how information should be organized and presented to the audience to make the user-experience more intuitive.
* Design and development of a mobile application for the currently platform to render its content more accessible to university students and staff.

# @CMU Web Service Development

This provision aims to guide its audience through the fundamentals of web service development for the @CMU social platform. The chapter begins with an introduction to the ELGG social engine – an application framework that lays the groundwork underpinning @CMU.

Understanding the ELGG platform infrastructure is an essential stepping-stone toward developing web services for @CMU. Basic knowledge with the workings of the ELGG Social Engine is absolutely imperative in developing software to coordinate transactions between data representation and the data model.

An overview of the ELGG project structure also serves as a prerequisite to web service development for @CMU, as extension development for the ELGG framework follows the convention-with-configuration paradigm. Specific guidelines exist to manage the project structure such that efforts required in configuring the extension can be minimized.

Lastly, this provision will conclude with a trivial web service development example to string together the concepts discussed here.

## The ELGG Social Engine

The ELGG social engine is an open source platform that is designed to exploit the conveniences of the LAMP solution stack. A typical environment setup for the ELGG social engine requires an Apache web server running on a Linux operating system, using MySQL as the backend database and the PHP scripting language for wiring up the server-side functional logic.

Under the hoods of the ELGG social engine is essentially a social networking framework. It provides the basic functionalities required to host a social network. Flexibility in the framework also offers developers the option to customize the social network to look and run the way they want it to.

The ELGG social engine design complies with the Model-View-Controller (MVC) design pattern, whereby the Model is realized through the ELGG data model, the Controller through Action and Page Handlers, and the View used in data presentation. Customizations and functional extensions to the ELGG social engine are implemented and installed through plugins.

Figure 1 outlines the application flow in ELGG.

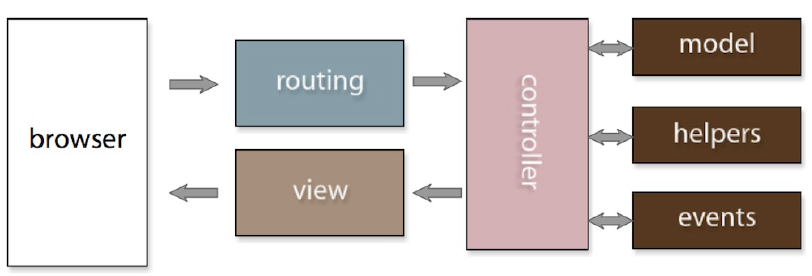


Figure 1: Application Flow in ELGG (Profitt)

### Data Model

As with most other dynamic web application, ELGG also uses a database for its data back-end. ELGG’s data model is backed by most recent versions of the MySQL database. An upside to ELGG’s data model is that a plugin author does not need to concern him or herself with creating database tables to align with custom functionality. ELGG’s data model is flexible to the extent that, in cases when other database functionality is added, or when new data back-end is introduced to support additional infrastructure requirements, updates to existing plugins and custom code is not required.

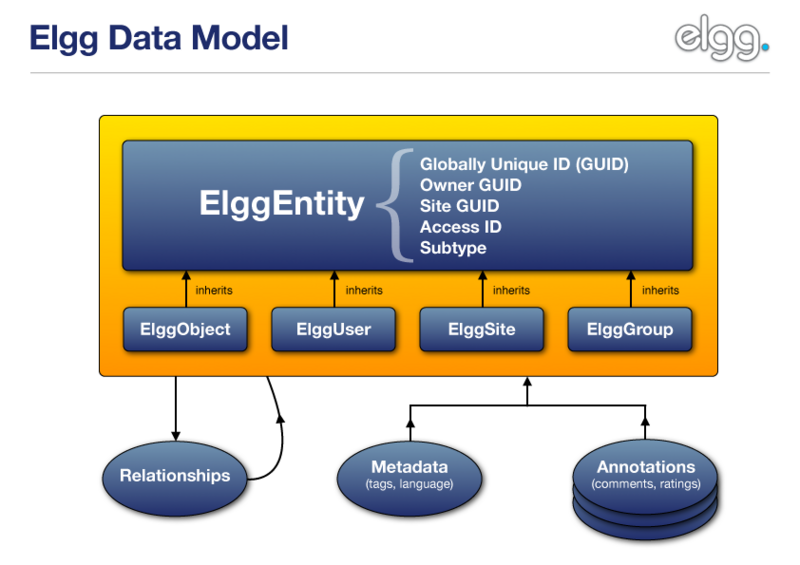


Figure 2: ELGG Data Model (Elgg)

In ELGG, everything runs on a unified data model, based on atomic units of data called [entities](http://docs.elgg.org/wiki/Engine/DataModel/Entities). Every entity in the system inherits the [ElggEntity](http://docs.elgg.org/wiki/Engine/DataModel/Entities) class. This class controls access permissions, ownership and so on.

### View

Views in the MVC pattern are used in the creation and presentation of data output. The Views system of the Elgg platform is responsible for delegations ranging from page layouts to links and form inputs. Additionally, the Views system also facilitates features such as automatic RSS generation and fast mobile interface prototyping.

### Controller

Controller in the MVC pattern is the component that handles transactions between the view/user and the system model. This invariably means that the controller can either dispatch actions to the model to update the model’s state, or conversely, dispatch updates to views so that they can reflect changes in the model.

Elgg controllers are categorized into 2 classes:

* Action Controller – This controller handles all user actions
* Page Controller – This controller handles all page activities

## ELGG Directories Structure

Development under the Elgg framework is motivated by the convention-with-configuration design paradigm, whereby guidelines exist to designate the directory and file structure of the project. This practice aims to minimize the efforts involved in composing settings and configurations directives in order to port new functionalities to the project.

Figure 3 outlines the directories structure of the Elgg framework. The section to follow provides an overview of each directory, and explains in detail the coding guidelines developers should follow.

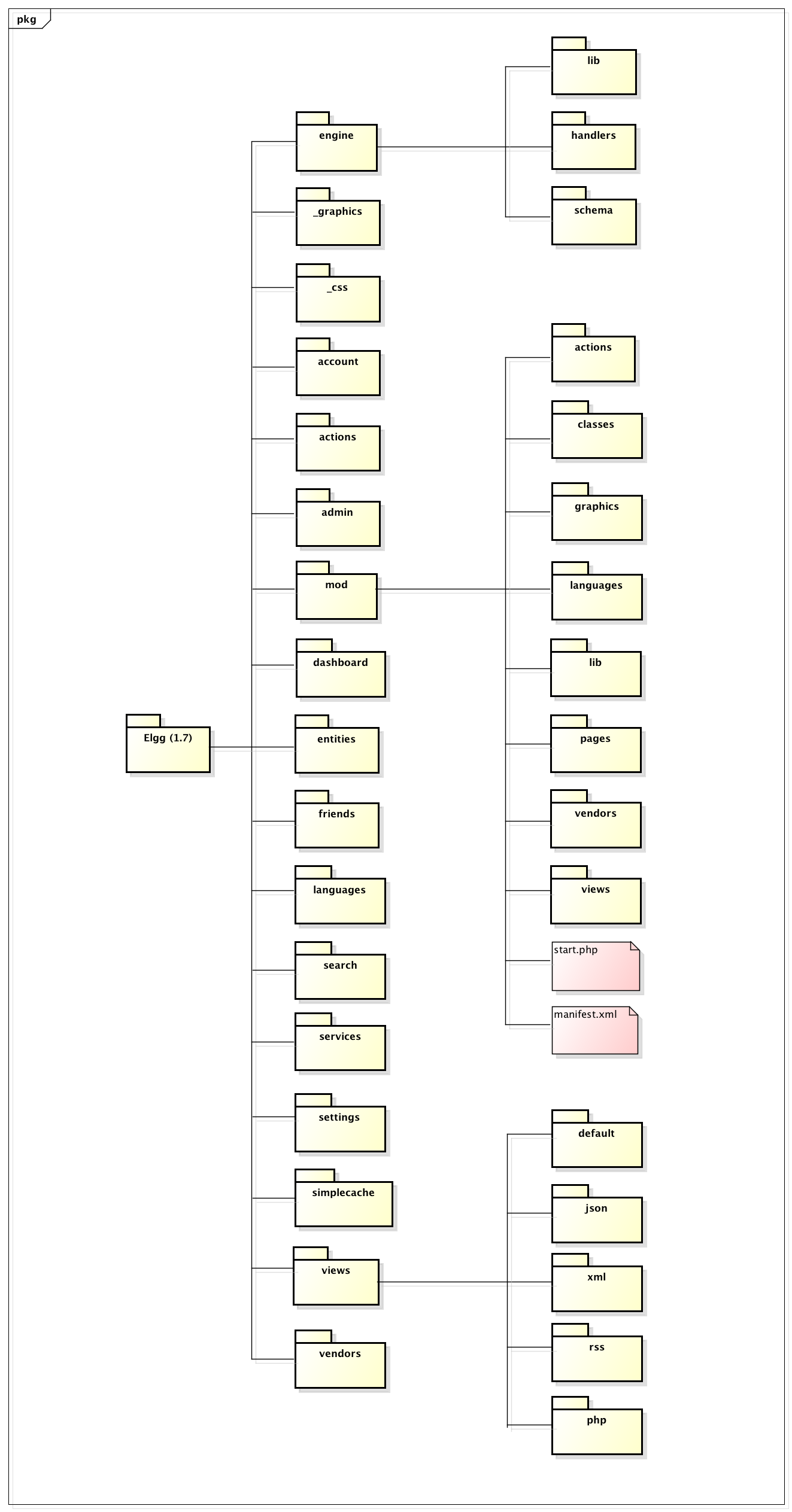


Figure 3: ELGG Project Directories

|  |  |
| --- | --- |
| \_css | This directory contains wrapper files for CSS and JavaScript. |
| \_graphics | This directory contains images used for the ELGG core as well as the default ELGG theme. |
| account | This directory consists of ELGG pages that implement and handle user account management activities such as registration and password management. |
| actions | This directory contains handlers that deal with actions in the system. These actions include form or button events coming from the user. |
| admin | This directory contains ELGG pages that implement and handle system administration related activities. |
| dashboard | This directory contains the page that is responsible for organizing and presenting dashboards in ELGG. |
| engine | This directory contains the source code to ELGG’s core engine. Components of the core engine include controllers that are responsible for database access, session handling, and event dispatch. |
| entities | This directory contains the implementation to the primitive ELGG entities that make up the data model. |
| friends | This directory contains ELGG pages that implement and handle buddy system related activities. |
| javascript | This directory historically contains JavaScript code used in ELGG, but is now deprecated. |
| languages | This directory contains localization files used in supporting language varieties tailored for the targeted audience. |
| mod | This directory contains various ELGG plugins for functional extensions and customizations. |
| search | This directory contains ELGG pages that implement and handle searching in ELGG. |
| services | This directory contains implementation of other ELGG services such as the RESTful APIs. |
| settings | This directory contains ELGG pages that implement and handle user settings. |
| simplecache | This directory contains the primary simplecache page. |
| vendors | This directory contains third-party JavaScript and PHP libraries required by the ELGG framework. |
| views | This directory contains all the ELGG view components in the MVC design pattern. |

Table 1: ELGG Project Directories Explained (Elgg)

## ELGG Plugin Development

Similar to functional extensions and customization development in ELGG, development and installation of web services in ELGG are also accomplished through plugins. The next section will take the reader through an implementation and installation of a simple plugin. Before we dive into such details, however, we need to administer specific configurations in @CMU to facilitate plugin development and debugging. The details are described below.

### Configuring @CMU for Plugin Development

The initial step to configuring the environment for plugin development is to first activate the Developers plugin. This will add to the administrative sidebar menu a Develop section alongside other useful tools. Navigate to the Developer Settings page from the menu and apply the following changes:

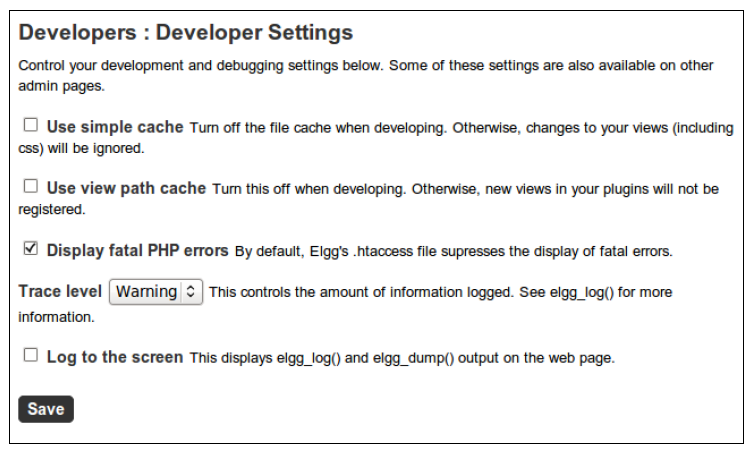


Figure 4: Developer Settings Page (Castello)

* **Turn Off the File Cache**

ELGG typically caches generated content to avoid having to recreate it every time. This generally expedites the loading of web pages, but can potentially cause problems during debugging as changes to a plugin may be ignored. Both “Use Simple Cache” and “User View Path Cache” options should be unchecked.

* **Turn On Display Fatal PHP Error**

Turning on this option allows the user to see fatal PHP errors in the browser. This option is usually disabled to prevent technical error messages from flooding the browser, but should be turned on for our purposes.

* **Set the trace level to Warning**.

Setting the trace level to “Warning” will force the system to log all warnings and errors to the server’s log. Sometimes we need to trace back potential system failures to warnings rather than errors, and this configuration will give us both.

### Creating the ELGG Plugin Skeleton

As discussed earlier, ELGG plugins exist in the **/mod** directory. Every ELGG plugin has a top-level directory named after the plugin’s intended functional purpose. A full-fledged plugin structure in ELGG resembles that shown in Figure 5, but a basic plugin requires only a top-level directory with two files.

actions/

example/

action.php

other\_action.php

classes/

ExampleClass.php

graphics/

example.png

js/

example.js

languages/

en.php

lib/

example.php

pages/

example/

all.php

owner.php

vendors/

example\_3rd\_party\_lib/

views/

default/

example/

css.php

forms/

example/

action.php other\_action.php

js/

example.php

object/

example.php

example/

context1.php

context2.php

plugins/

example/

settings.php usersettings.php

widgets/

example\_widget/

content.php

edit.php

activate.php

deactivate.php

CHANGES.txt

COPYRIGHT.txt

INSTALL.txt

LICENSE.txt

manifest.xml

README.txt

start.php

Figure 5: ELGG Plugin Structure (Elgg)

**Required Files**

The 2 files required in creating a basic ELGG plugin are:

* **manifest.xml**:

manifest.xml is used to document the details about the plugin. Information included in manifest.xml are the plugin name, author, version, and a brief description about the plugin. Information documented in manifest.xml is visible from plugins page in the administration section of the ELGG platform. This file is placed under the plugin root directory. More information on how to write a manifest.xml can be found at <http://docs.elgg.org/Manifests>.

* **start.php**

start.php implements the entry point to the customization or feature that is to be installed onto ELGG. start.php can be considered as plugin bootloader. This file is also placed directly under the plugin root directory.

**Actions**

The plugin developer should create an **actions/** directory to store all script associated with actions. Furthermore, the name of the file associated with a specific action should directly reflect the action it is designated to perform. For instance, the action **this/example/action/** would go in **plugin\_name/actions/this/example/action.php**

**\*.txt Files**

.txt are placed in the plugin folder for the purposes of plugin documentation. These files must be in Markdown syntax as they will generate links on the plugin management sections.

**README.txt**

README.txt is used to provide additional information of other nature about the plugin.

**COPYRIGHT.txt**

This file contains information about the plugin’s copyright.

**LICENSE.txt**

This file describes the license under which the plugin is released.

**INSTALL.txt**

This file provides additional installation instructions for the plugin if the procedure is relatively complex.

**CHANGES.txt**

This file documents the change history of the plugin in terms of version number. The most recent version starts at the top.

**Pages**

The **pages/** directory is used to store scripts that are used for generating pages. This directory is placed directly under the plugin root. Additionally, page-generating scripts should be placed in directories whose names corresponds to the their handler. For example, the script for page **site\_name.com/handler\_name/view/page\_name** needs to be located at **mod/plugin\_name/pages/handler\_name/view.php**.

The rationale behind having this structure is to sustain a direct relationship between urls and scripts. This makes development work much more straightforward for plugin developers.

**Classes**

All classes defined by your plugin need to be included in a **classes/** directory, since classes placed in this directory will be automatically loaded on demand, and need not be included explicitly.

**Vendors**

Any third-party libraries used by the plugin will go into in the **vendors/** folder. The **vendors/** folder typically goes under the plugin root.

**Lib**

The **lib/** folder usually contains the source code defined by your plugin.

**Views**

The **views/** folder is used to store views that intend to override core views. Any views placed in this folder will override views of the same names that are located in the ELGG core. More information about views can be found here: [Views](http://docs.elgg.org/wiki/Views).

**Javascript**

JavaScript used by every page will typically go into the top level **js/** folder, whereas JavaScript not used by all pages should go into specific **views/js/** folder. More information about JavaScript can be found here: [Javascript](http://docs.elgg.org/w/index.php?title=Javascript&action=edit&redlink=1).

**activate.php and deactivate.php**

The **activate.php** and **deactivate.php** implement logic that are executed when the plugin is activated and deactivated, respectively. Logic implemented by these files are intended for execution of one-time events such as registration on activation or garbage collection on deactivation.

### Web Service Development

With the plugin development environment setup and a better understanding of the ELGG plugin structure, we can now begin with implementing a simple web service.

ELGG implements a framework that facilitates the development of web services, allowing for accessibility of ELGG functionalities and data to external websites and applications. These RESTful APIs implement a REpresentational State Transfer (REST) and Remote Procedure Call (RPC) hybrid similar to those provided through Twitter web services.

The key components to constructing an extensive web service API in ELGG are web service implementation/provision, (key-based) API authentication, and (token-based) user authentication. These components are discussed in the following sections.

#### Developing a Simple Web Service

As noted in earlier sections, one of the compulsory files for web service plugin is manifest.xml. The purpose of manifest.xml is to provide information about the plugin, thus the content of manifest.xml will vary depending on the nature of each plugin. Figure 6 provides an example of manifest.xml.

<?xml version="1.0" encoding="UTF-8"?>

<plugin\_manifest xmlns="http://www.elgg.org/plugin\_manifest/1.8">

<name>@CMU Web Services</name>

<author>Hsuan-Chih Chuang</author>

<version>1.0</version>

<description>This is a Web Service Plugin for @CMU</description>

<website>http://www.elgg.org/</website>

<copyright>(C) Hsuan-Chih Chuang 2013</copyright>

<license>GNU General Public License version 2</license>

<requires>

<type>elgg\_release</type>

<version>1.8</version>

</requires>

</plugin\_manifest>

Figure 6: Sample manifest.xml

ELGG supports development of web services through a set of built-in functions and classes such that implementations can be readily exposed for use by external applications. One can start by creating a function which implements the logic for the web service one intends to provide. Figure 7 gives an example of a simple web service that returns a string passed to the ELGG server by the application client.

Figure 7: A Simple Web Service

/\*\*

\* A simple web service

\*/

function my\_echo($string) {

return $string;

}

Before we can put the functionality to test, we need to first expose it as a web service to the application client. To do so, we make use of the built-in function **expose\_function()** as shown in Figure 8.

expose\_function( $ method,

$ function,

array $ parameters = NULL,

$ description = “”,

$ call\_method = “GET”,

$ required\_api\_auth = false,

$ require\_user\_auth = false

)

Figure 8: Function Prototype: **expose\_function()**

Parameters of **expose\_function()** is outline below. The function returns a Boolean value to indicate whether the expose is successful.

|  |  |  |
| --- | --- | --- |
| **PARAMETER** | **DATA TYPE** | **DESCRIPTION** |
| **$method** | String | API name for the web service, for example “test.echo”.  This parameter is **mandatory**. |
| **$function** | String | Function callback of the API, for example “my\_echo”.  This parameter is **mandatory**. |
| **$parameters** | Array | Array of parameters to be passed into the function.  Optional parameters are typically passed in following required parameters.  This parameter is **optional**. |
| **$description** | String | A human readable description of the function.  This parameter is **optional**. |
| **$call\_method** | String | The HTTP method to be used for the service call.  This parameter is **optional** and defaults to “GET”. |
| **$require\_api\_auth** | Boolean | Does the service require API authentication (using the API key)?  This parameter is **optional** and defaults to “false”. |
| **$require\_user\_auth** | Boolean | Does the service require user authentication?  This parameter is **optional** and defaults to “false”. |

Table 2: Function Parameters: **expose\_function()**

Having understood how a function can be exposed, the simple web service example we implemented earlier can now be exposed as follows:

expose\_function("test.echo",

"my\_echo",

array("string" => array('type' => 'string')),

'A testing method which echos back a string',

'GET',

false,

false

);

Figure 9: Exposing a Web Service

The plugin folder should now contain **manifest.xml** and **start.php** (comprising of the simple web service implementation and the expose), and is ready to be added to the **/mod** directory under the ELGG root structure. At this point the plugin should appear in the “Plugins” page on the @CMU website under the “Administration” section. Simply click on the “Activate” button to activate the plugin.

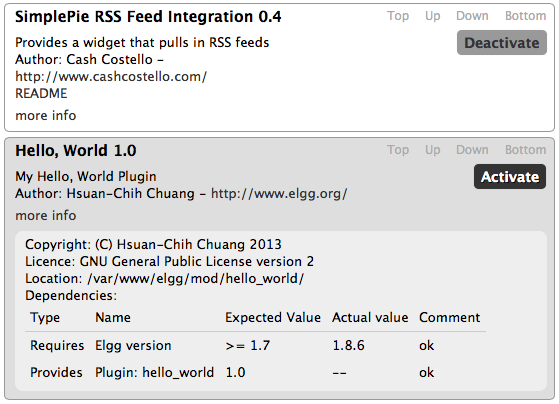


Figure 10: Plugin Activation Menu

Once the plugin has been activated, it is registered to the list of supported web services for @CMU. This list can be viewed by visiting: [http://at.cmu.edu.au/elgg/services/api/rest/xml/?method=system.api.list](http://yoursite.com/services/api/rest/xml/?method=system.api.list).

To test the web service from a web browser, submit an URL with method “test.echo” and a random string that you want to be returned back to the browser:

[http://at.cmu.edu.au/elgg/services/api/rest/xml/?method=test.echo&string=ATCMUTest](http://yoursite.com/services/api/rest/xml/?method=test.echo&string=testing), and the XML response should display in the browser.

In addition to XML responses, ELGG’s web services API framework also supports other formats like JSON and serialized PHP. Different formats can be requested by substituting XML for JSON or PHP in the URL.

#### Key-Based Authentication

Key-based authentication can be used to control access to supported web services. Typically this is to limit the extent to which third party applications can have in accessing the resources provided through the platform. Other utilizations of key-based authentication include using the API key to limit the number of service calls permitted to an application on a given day.

ELGG supports 2 modes of API authentication: key-based authentication and signature-based HMAC authentication, although other modes of authentication may also be added by the developer.

Key-based authentication is among the most commonly accepted approach. Web services provided by Twitter, Flickr, and Google all use this approach. This is also the chosen approach for this particular project. The ELGG “apiadmin” plugin needs to be installed to allow generation of API keys; the plugin can be downloaded from ELGG’s plugin repository at GitHub (<https://github.com/Elgg/apiadmin>).

API keys for @CMU web service access can be generated and provided by the @CMU site administrator upon request. These keys can then be included in web service calls for accessing site resources that require API authentication. Restricted service calls without an accompanied API key will generally be rejected and the “Method call failed the API Authentication” message is returned.

#### Token-Based Web Service Authentication

Most websites require the end-user to be logged on before the user can post data on the site. Identity checks as such prevent users from pretending to be and tampering with accounts as somebody they are not. Token-based authentication is designed for these purposes.

To be able to access and manipulate content on the @CMU website, the user must first obtain an authentication token via providing a valid account username/password information. The **auth\_gettoken** API is implemented and made available to client applications to serve this very purpose. The generated token is assigned to the user for a limited period of time for authenticating service calls. Consequently, this token must be included as a call parameter in most service call URLs indicated by the parameter name **auth\_token**.

The majority of the content on @CMU is available only to logged-on users. Therefore most web services implemented for @CMU also require that the identity of the access client be validated. Since session tokens will remain valid for some period of time, token-based authentication offers security advantages in that the client does not need to attach username/password to web service calls. Another advantage to token-based authentication not necessarily relevant here is that a user can grant access privileges to an application delegate for a limited period of time while protecting the user’s login credentials. Alternatively, an OAuth approach can also be taken as an extension to the token-based approach.

# Mobile Client Application Design

## Design Considerations

There are four approaches to making the @CMU website mobile:

* **Responsive Website**

A responsive website adjusts its layout according to the dimensions of the devices it is being viewed on. This approach is ideal for mobilizing websites that are especially rich in information and content variety. Developing a responsive website involves effort only on the server and no effort on the client end.

* **Web Application**

A web application is similar to responsive websites in that development work involves effort only on the server end. Web applications, like most native applications, are generally more task oriented than they are content-focused. This characteristic distinguishes web application design from responsive website design.

* **Native Application**

A native application is an application designed and developed to run on specific client device(s). This means that a native application will need to be programmed for the operating system and device architecture of the target device. Applications of this nature can be downloaded from application marketplaces such as the APP Store or Google Play.

* **Hybrid Application**

A hybrid application is a native application built using cross-platform technology. Most popular hybrid applications are developed using a combination of HTML, JavaScript and CSS. Hybrid application development is easy to pick up because it leverages web technologies that are widely used. Another advantage to hybrid application development is that it embraces the write-once-run-anywhere philosophy.

The @CMU mobile application is developed using the hybrid application approach not only because it aligns with the objectives of the Mobile Application Development course, but also because it is practical for fast-prototyping purposes (as is in the case of this independent study project). The final product of the @CMU application could potentially evolve into native applications if need be, but a hybrid application would suffice for what this project aims to achieve.

## Application Features

This project intends to implement the following @CMU mobile application features:

* **User Login**

Access to information on @CMU requires user credential verification. Furthermore, access privileges will vary depending on the identity of the logged-in user. The user login page handles access privilege management on behalf of the @CMU portal. User login is implemented as single-sign-on – user credentials are stored encrypted to facilitate subsequent APP launches.

* **Latest Blog Feed**

Latest blog feed delivers the latest blog post updates from @CMU. Students can usually find information about latest campus events, internship opportunities, available student accommodations, and so forth.

* **@CMU Member Contact Book**

The powerful student/alumni network is a CMU legacy the school is particular proud of. The @CMU member contact book provides a means to connect students and alumni. Students and alumni can keep in touch through the @CMU mobile application and keep each other updated with their career plans.

* **Student Information by Category (in Substitution of Map & Navigation Functionalities)**

The Student Information by Category section organizes blog posts and student information by their categories. The categories currently include program handbook, internship, career, campus events, accommodation information, orientation information, and IT support (wireless connection setup, printer setup). These categories are subjected to change depending on how student information will be organized in the future.

## User Interface Design

The figures in section 3.3 outlines the overall user interface design of the @CMU mobile application. At the top level, the end-user will be able to navigate between the main APP sections (latest blog feeds, member contact book, student information) using the navigation bar located at the bottom of the screen.



Figure 11: User interface design - overview

Each APP section showcases a list of feed/member/category entries at the top level. The end-user can then click on each entry to have the application load a page consisting details about the specific entry (as shown in Figure 12: User interface design - detailed view).

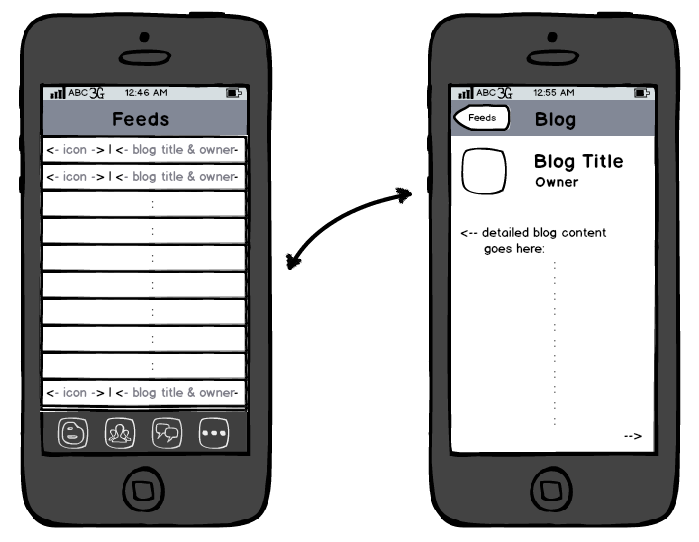


Figure 12: User interface design - detailed view

## Web Service Client Development

### Application Flow

The diagram below depicts the flow of the @CMU mobile application as a web service client. Essentially each view transition will trigger a web service call to fire from the device. Once the response to received from the web service server, the view on the client end is updated with the content of the response.

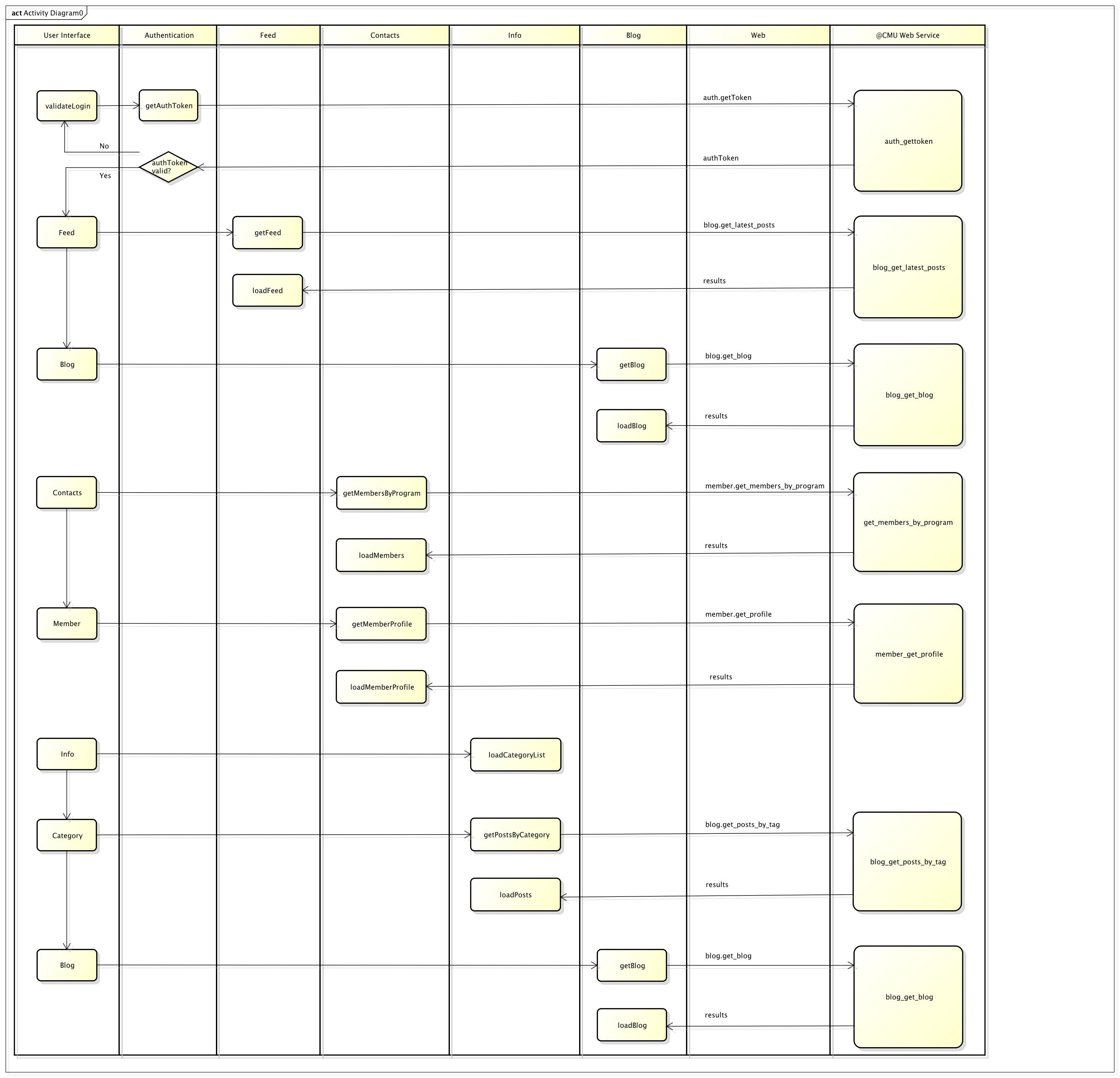


Figure 13: Application Flow

### Asynchronous Web Service Calls using AJAX

A vital part of human-computer interface design is to provide the audience an interactive user experience. In other words, the user interface must be reactive to user intervention at all times – even while the device is awaiting specific web service calls to respond. Consequently, all web service calls must be non-blocking.

AJAX is a resort that helps with developing asynchronous web service calls. An AJAX call consists of service call parameters and a callback function. The callback function implements the procedure to be taken after the response to the service has been received, and executes asynchronously.

When an AJAX web service call fires, the program does not block while waiting for the response from the server. Rather, the application continues with program execution and the block of code handling the web service response is called when the response is received. The code snippet below demonstrates how an AJAX call is typically implemented.

function callAJAX() {

$.ajax({

url : call.URL,

type : call.type,

dataType : "application/json",

data : data,

complete : function(response) {

/\*

\* this anonymous function implements the AJAX callback

\*/

if (response.status == 0) {

//browser could be on offline mode

} else if (response.status == 404) {

//404 error

} else {

doSomethingWithTheResponse(response);

}

}

});

/\*

\* the program continues with execution of

\* doSomethingElseWhileWaitingForResponse()

\* after the AJAX call is fired

\*/

doSomethingElseWhileWaitingForResponse();

}

Figure 14: AJAX Call Example

### Pipelining Web Service Calls

One of the major design considerations worth noting is how to implement application data loading in a way that it does not compromise the user experience. This is especially important when the application needs to pull large amounts of data from a web server when the network connection is slow.

An approach to overcome problems of this nature is lazy-loading. Since the visual real estate on a mobile device is usually quite small, it doesn’t make a lot of sense for the application to pre-fetch data that the user will not be able to see in the current view. Rather, data should be pulled and loaded into the view as the user scrolls down the page.

Lazy-loading is feasible if the application is able to determine when the user has scrolled to the end of the html page in the view. A mathematical derivation to achieve this is depicted below.

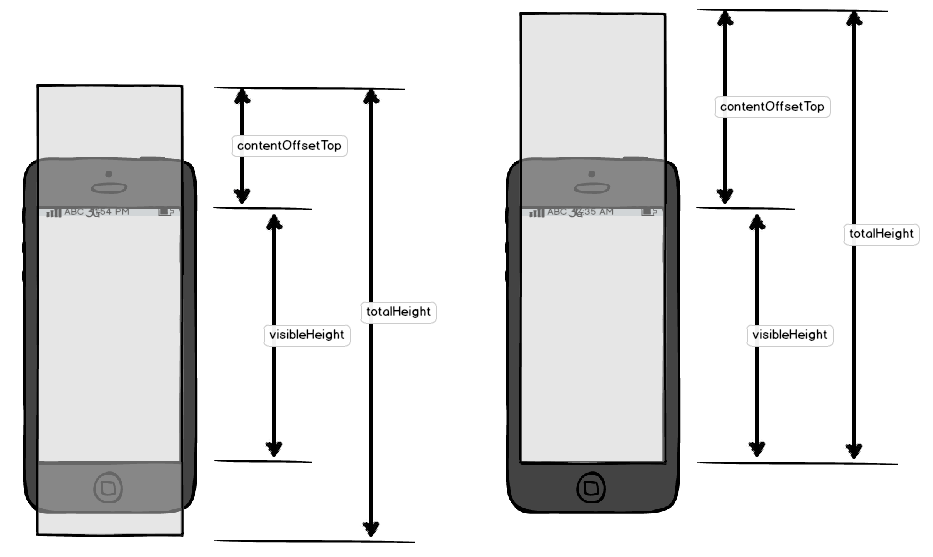


Figure 15: Lazy-Loading Visualized

function isAtBottom() {

var contentOffsetTop, visibleHeight, totalHeight;

if (document.documentElement.scrollTop) {

contentOffsetTop = document.documentElement.scrollTop;

} else {

contentOffsetTop = document.body.scrollTop;

}

visibleHeight = document.documentElement.clientHeight;

totalHeight = document.body.scrollHeight;

return (totalHeight <= contentOffsetTop + visibleHeight);

}

Figure 16: Lazy-Loading Code Snippet

HTML document object properties allow the application to size:

1. the height of the device’s screen,
2. the height of the entire HTML document, as well as
3. the height of the HTML document portion that has been scrolled past the top of the device’s screen.

The application is then capable of determining whether the bottom of the HTML document has reached the bottom of the device’s screen using these 3 parameters. In other words, if the sum of parameters 1 and 3 equates to the second parameter, this is an indication that the user has scrolled to the bottom of the HTML page and that new data needs to be fetched and loaded to the document.

# Limitations and Future Work

The current version of the @CMU mobile application provides a collection of features that serve as foundation for streamlining information communication between site users. The application, however, is not entirely complete. Limitations to the current application lay ground for future development work to further refine the application. These limitations can broadly be classified into 2 categories: technical limitations and feature limitations.

* **Technical limitations**
* The Feed and Info implementation already supports lazy-loading. The Contacts section, on the other hand, currently loads the entire member list all in one go. This is by no means desirable. Consequences to this drawback amplify especially when the network connection is slow. This phenomenon is expected to exacerbate as the member list continue to grow. Implementing lazy-loading Contacts, however, presents its own challenges because the search filter for the member list relies on the entire list to be locally cached to function properly. This problem will be an interesting undertaking for whoever will continue with the application development.
* The application badge icon and the application splash screen have not yet been installed onto the APP. Although this issue is relatively easy to resolve, it is vital in rendering the application complete.
* **Feature limitations**
* This version of the application functions merely as a viewer. The application allows its users to read information that already exists on @CMU, but doesn’t yet allow them to post information to @CMU. Two-way communication might be a nice feature to have in future versions.
* Push notification provide a convenient way of notifying the users with any information update that might interest them. This feature could be valuable to the end-user.
* The Info section is currently of limited value because there is currently not a systematic way of categorizing posts (post tags are not very organized). Mechanisms need to be put in place to ensure that user posts are tagged according to a predefined set of categories. Much of this work will involve the site administrator.

# Works Cited

Elgg. (n.d.). *What events are available in the Elgg core*. Retrieved from Elgg: http://docs.elgg.org/wiki/What\_events\_are\_available\_in\_the\_Elgg\_core%3F

Elgg. (n.d.). *Engine/DataModel - Elgg Documentation*. Retrieved from Elgg: http://docs.elgg.org/wiki/Engine/DataModel

Elgg. (n.d.). *How is the code divided among the different directories?* Retrieved from Elgg: http://docs.elgg.org/wiki/How\_is\_the\_code\_divided\_among\_the\_different\_directories%3F

Profitt, B. (n.d.). ElggCamp Satiago 2011. Santiago, Chile: Elgg.