Virtual Emergency Operations Center
Individual Development Documentation

Matt Mooney

Table of Contents:

Overview and Environment

This document outlines the development tasks completed by Matt Mooney during a summer community based research program at the University of Notre Dame funded. The first thing I did before any development was read background material and research papers on the vEOC project and explore the technologies used in the project. I was already very familiar with HTML, PHP and MySQL while I had to spend a little more time learning Javascript. I familiarized myself with the interface of the vEOC and learned the call sequences for the pages.

I primarily developed on my personal Windows Vista platform. I used TortiseSVN to manage my Subversion updates and commits and Adobe Dreamweaver CS4 as my PHP/HTML/Javascript development environment. On several occasions I used Putty to SSH into the server to directly manipulate files or the MySQL databases. I used updated versions of both Firefox 3 and Chrome 6 for testing the vEOC.

General Improvements

I spent most of the initial weeks just finishing up parts of the vEOC that were set-up, but not properly working or parts that were in need of small aesthetic changes. This allowed me to become more familiar with the internal workings of the project as well as the various programming (naming, sequence, etc) conventions used by other project members. These initial changes included adding database queries, editing the display information already on hand, and touching up forms, display tables and menus. Most of these improvements were rather self-explanatory and don't need any explanation beyond the HTML, Javascript, PHP code and associated comments.

Contract Resources

I imported information from number of the resource providers that are contracted with Miami-Dade EOC into a table called contract_resources. This table is considered "global" across all scripts and users. It is also not modifiable from the vEOC interface (i.e. must be changed from the back-end). I then built an interface (logistics.php) for users to access and search the contract resources available while checking resource requests from other players in another (linked) window.

User Console

In an attempt to streamline the interface and create a less-confusing environment, I wrote a separate login, pickplayer, and main console page (RegularLogin3.php, pickplayer3.php, userconsole.php). This new console combines the main panel, exercise panel and any windows opened from the main panel onto a single browser page using inline frames, or "iframes". From this point forward, index.php and userconsole.php serve the same purpose, i.e. set up session variables, some database tables and bring the panels to the user. I will use the terms together or interchangeably, but they serve the same essential function.

Database Table Reorganization

While working on several pages, I realized that many of the tables being created at login should in fact be used by all players during an active exercise. I changed the naming convention for several of the tables to reflect that goal. For example, the shelter or road closure updates issued by one player should be seen by all players. Such tables are now named with the

following convention: *ScriptTablename*. Some tables kept the original format where it was appropriate (e.g. After Action Reports or player logs): *UserScriptPlayerDateTablename*

Now that several tables would remain constant throughout an exercise, the creation and clearing of these tables had to be moved from the index.php/userconsole.php page. I built a new page (databasereset.php), accessible only from the Exercise Developer console, that clears/re-builds these Script-wide tables. It is important to note that when an exercise developer creates a new script, the ExDev must also build the supporting tables using this tool.

I also realized that we needed a way to quickly get rid of the *UserScriptPlayerDate* specific tables. I originally tried writing a shell script that we would run form the backend, but I realized that this is not a good solution for the end-user (only useful to those of us with backend access). So I added another function to the Database Control Tools section of the Exercise Developer that the ExDev can clear the user-specific tables from the database.

It is extremely important to note that the table-naming convention must be the same across the databasereset.php and index.php/userconsole.php pages. While the tables are only modified in databasereset.php, the index/userconsole pages set the PHP Session variables that contain the table names so that they can be accessed and modified by the players throughout the exercise. These session variables are crucial for all other pages to work. In fact, storing the table name in Session variables allows us to easily access/change the tables without changing hard-coded table names on every PHP page that accesses a database table.

Target Capabilities and Metrics Developer

In 2007, the Department of Homeland Security issued a standardized list of targeted capabilities and time metrics to meet these capabilities. Each targeted capability and each metric form this document are loaded into the vEOC.

This was a one-time procedure that hopefully will not have to be repeated. I had to copy and paste each field from the original PDF file into an excel spreadsheet. I then converted the spreadsheet to a CSV to import it into the respective MySQL tables (*mastertcl, mastermetrics*).

These tables are accessible from the Exercise Developer Target Capabilities Developer. The master tables for both capabilities and metrics can be searched or browsed by categories (organized by mission and task). They can then be “added” to the current script. This simply copies that specific task or metric into the script-specific table for capabilites or metrics (*Scripttcl, Scriptmetrics*).

I borrowed much of the logic in this section from another project that I contribute to, the EPICS Red Cross Disaster Database. The borrowed code is open-source and documented accordingly.

Disaster Map

I used the Google Maps Javascript API v3 to implement an interactive, collaborative, live disaster plotting map. Extensive documentation and examples are available on the Google Maps API family of sites that are extremely helpful. I used a number of overlay and point examples to help with the flood regions, radioactive regions, fires and medical emergency data points.

The basic premise for the Disaster Map (disastermap.php) is that users can choose a type of disaster to add, and this adds a specific event listener to the map. When a user clicks a point, a marker is added to the map with an appropriate icon (that marker has its own listeners for clicks or drags to control dragging or removing the marker). The “flood” and “radioactive” types require at least three points in order to properly define a plane in which the disaster area is contained.

When the user clicks “save map,” a custom Javascript function collects the Lat/Lng coordinates for each marker and forwards them to the disastermap_sav.php via POST. On this page, the data is sorted by type and inserted into the appropriate *Scriptdisastermap* table. This information is then retrieved and parsed by another custom disastermap.php Javascript function to display the information on the map.

Logging

I also designed a flexible logging system for the vEOC. This logging system largely piggybacks on the existing communication system implemented with cometD from the Dojo toolkit. I didn’t know too much about this technology, but I didn’t need to fully understand it in order to use the basic functions, especially for logging.

Basically, I added the necessary Dojo library to each PHP page that is part of the site and added a dynamic javascript page generated by logjs.php. The functions in this dynamic script publish (but do not subscribe) to the /log channel. I simply need to call the logme(), or loginject() in the case of controller.php, function when any event fires that we wish to log. For example, <button onclick=”logme(‘thispage’, ‘clicked the button’) /> will send a message to the Exercise Developer (smpanel.php).

It is on this page (smpanel.php), that we have cometD subscribe to the /log channel. A function here reads the channel for messages, extracts the variable we want, and inserts the data into the *Scriptlog* table of the database via an AJAX request.

AI Tutor (AIDAC)

In order to develop an artificial intelligence solution for assisting vEOC users, I looked into existing chatbot technologies. I discovered the work of Dr. Richard Wallace, who developed and won multiple AI-related awards with his Artificial Linguistic Internet Computer Entity (ALICE) bot. ALICE is composed of a set of markup files written in Artificial Intelligence Markup Language (AIML, an XML extension). These files make up the "knowledge" of the bot, but the responses are generated by a program known as an AIML interpreter.

In ALICE's case, that interpreter is Program B written in Java. Since the vEOC is already built on PHP/MySQL technology, I found two AIML interpreters that would run on PHP and store the AIML in a MySQL database for fast searching. The first, Program E, is now what is known as "abondonware" because it was an open-source project that now lacks any active development or support. It was riddled with minor bugs, and making any changes to the AIML files required completely rebuilding the Program E database.

The second interpreter, known as Program-O, had a few problems of its own, but has an active lead developer and very active and supportive community (via online forums). I was able to modify the PHP scripts enough to have the program better fit our needs and fix a few of the bugs that I ran into. Specifically, the original Program-O displayed only the last line of the bot response. This works fine in most cases (even multi-sentence cases) but it fails to deliver the entire response when a
 tag is present. The original program/chat.php contained these lines:

```
$res .= "<div class=\"demouser\">You: ".stripslashes(urldecode($response_Array['input'][$i]))."</div>";
$res .= "<div class=\"demobot\">Bot: ".stripslashes(urldecode($response_Array['that'][$i]))."</div>";
```

Therefore, the 'that' portion of the response array is being printed as a response. However, according to ALICE standards, "that" is only the last line of a multi-line (paragraph) response. So, in a sense, the code is correct: The last line should only be displayed under response_Array['that'][]. However, the bot should be printing "answer" back to the user, not "that".

I modified my PHP to display $response_Array['answer'] for the most recent answer. I did try frontOfStack'ing (one of the internal functions) the answer, but it didn't work properly, so the following worked okay for our solution:

```
$res .= "<div class=\"demouser\">You: ".stripslashes(urldecode($response_Array['input'][$i]))."</div>";
$res .= "<div class=\"demobot\">AIDAC: ".stripslashes(urldecode($response_Array['answer']))."</div>";
```

There were some other minor, aesthetic changes made to the Program-O source, but the files and code comments are all present in the SVN repository, so I won't go into more detail on the customization.

Finally, I named our bot "Artificial Intelligence Disaster Assistance Chatbot" (AIDAC) and gave it a simple personality using the ProgramO admin interface. To "educate" AIDAC, I loaded the Annotated ALICE AIML (AAA) into the database. I also wrote several custom AIML categories as per the AIML 1.x standards in order to "teach" AIDAC about the vEOC. Finally, I removed a small subset of the categories from the standard AAA package so that I could ensure AIDAC correctly answered vEOC questions. I reviewed the logs to see what else needed to be added to the database to properly answer user questions.

The updates to AIDACs AIML set were made using the ProgramO admin interface. This interface is one of the main reasons I chose ProgramO along with the fact that adding new AIML files does not require completely rebuilding the *programo* MySQL database. The admin pages is accessed at /veoc/programo/admin. Currently you can use the same username/password combination that is used for "root" access on the MySQL database.

Dashboards

The dashboards are charts and graphs available to the players in the exercise panel for visualizing important data. These graphical displays are of exercise-live data and are updated whenever a user reloads the page containing the graphs. Again, we rely on Google for this nice API; this time it is the Google Visualization API (the interactive, Javascript side of the house, not the image side). Once again, the documentation and examples provided by Google is extensive and very helpful. I simple followed the API examples and layed out the charts we desired. The code comments are also very helpful in this area.

Researcher Console

The researcher console allows the individual studying the results of an exercise easily see some common statistics (on the players) from a given exercise. To generate each of the

statistics, I built a simple set of queries that parses the information from the exercise log. We are able to make these reports player-specific because the log tracks all player actions that we desired by specifying that they should be logged (identifying the player in the process) and tracks injects sent to each player. The current system reports what percentage of injects a player did not respond to (by comparing the number of injects acknowledged to the number of injects sent to that player) and the average inject response time (by comparing the real world time that the inject was acknowledged to the time the inject was sent). The goal is to eventually query the TCL and Metrics tables as well to compare the player actions to expected actions and required time windows, but these advanced features are not yet implemented in this initial development phase.

Handbook/Initial Status/Responsibilities Developers and Displays

While these handbooks are all separate developer/display systems using unique tables, they all function in essentially the same way. Rather than giving each script its own table, I built a master table where each script occupies only one row of the table. Each row, however, contains multiple TEXT and/or BLOB datatypes so that we can still store a lot of data for each script. Looking over the code will clarify how these pages work, since they are relatively simple database add, update and display pages.

The one unique feature of the Initial Status Developer is the ability to upload (and subsequently display) an image file. I have worked with file uploads on a PHP/MySQL system before, so this wasn't anything terribly new to me. It was hard to find good examples on the internet, so I once again relied on borrowed code from the EPICS Red Cross project. It is, of course, open-source and documented in the vEOC source code.

If you will be adding similar pages or modifying the current pages, be sure so look over the existing code so that you understand the notions behind PHP file uploading, MySQL BLOB storage, and PHP header-file displays.

Revamped Exercise Controller and Exercise Panel

I performed a number of performance and maintainability tweaks in the Exercise Controller (controller.php) and Exercise Panel (exercisepanel.php). I consolidated the start, stop, and other exercise control channels into one channel called /script/control. The variable "status" now carries all exercise control commands over this channel. This makes the control system easier to understand and more extensible. I also renamed the inject variable traveling over the

various player-inject channels to “inject”. Prior to this, the renamed variables had names like “test” that didn’t help the programmer understand what was happening.I updated the exercise panel to reflect the channel and variable name changes accordingly.

I also split the function to handle incoming dojo.cometd messages (exercisepanel.php) to handle the unique requirements of each channel. The talktome() function still handles all incoming “calls” (chat requests) from other players. The scriptcontrol() function now handles all exercise control status updates coming over the single /script/control channel. The injecthandler() function handles injects meant only for one player, updates the panel communication links (to red) accordingly, and adds the inject to the received inject list. The globalupdate() function receives injects sent to all players over the /ALL channel, alerts the player, and adds the update to the received inject list (accessible from the exercise panel).

Consolidated Database Connections

One of the more simple improvements I added to the vEOC was the consolidation of the database connection code. Each PHP page that requires access to the database must connect to the database using the proper hostname, username and password combination. The script must then access the correct database (named “auth” in this case). However, in order to maintain portability and flexibility, it is critical to be able to change these variables without having to edit every PHP page with database access. Therefore, it is extremely inconvenient to have this “connect” part of the script hard-coded on every page that accesses the database. It is much more convenient and viable to “include()” a single script on several pages that contains all of the necessary information. This way, when a single factor changes due to a configuration tweak or server change, we can quickly adapt the entire system to the new database setup.

Server Portability Adjustments

When the original development server crashed, I inspected much of the code for hard links to the original server address. Such links cause page errors because the links refer to pages as pages on the old server, not pages relative to the current setup.

I had already ensured that the database connections were managed when I took care of the consolidation as described in the previous section. However, all of our cometD connections were being initiated over a specific hostname path: “veocdev.nd.edu:8080/cometd/cometd”.

Obviously, this did not work once we started up the off-site server. I tried using "localhost" as the web address, but this did not work. After some searching, I found that I could simply link to the path on the machine without any hostname in this manner: "/cometd/cometd". This allows the current machine to operate on any domain name. This is still not an ideal solution, as the cometD files should be located in a directory relative to the web root and maintained in the Subversion code repository.

Exercise Archives

I implemented a simple mechanism to "archive" an entire exercise's tables. The ExDev can access the tools on the Database Tools panel. Behind the scenes, the code simply collects all of the tables associated with a given script and copies them to tables of the same structure and same name with the exception of the key ARCHIVE at the beginning of the name. This, in a sense, creates an intuitive separation between the archive and true exercise tables.

There is, however, one drawback I found that I ended up leaving in the system because it does in the end benefit the archive function. Because I use "script%" ('%' is a MySQL wildcard) to find table names to archive, the script table itself is archived under the new name "*ARCHIVENamescript*" where Name is the selected script. This new table will now show up as a script option everywhere else on the system. I initially considered changing the convention to *"ARCHIVETableARCHIVE"* so that it wouldn't get picked up, but realized that having the archived script show up as an option could be beneficial. For one, it allows us much easier access to those tables for review and analysis using the vEOC Researcher panel. In addition, if for some reason an exercise needs to paused or branched in some way, this can handle that scenario. The archived scripts can also be removed rather easily using the same database control tools page.

Contact

Should there be any questions about this document of the work I've done in support of this project, I should be available at the following address: mmooney3@alumni.nd.edu

References and Additional Documentation

These are the links to the Google APIs that I referenced:
http://code.google.com/apis/maps/documentation/javascript/
http://code.google.com/apis/visualization/documentation/using_overview.html

I used this site as a quick MySQL reference guide:
http://www.pantz.org/software/mysql/mysqlcommands.html

I used w3schools.com as reference for JavaScript and HTML DOM:
http://www.w3schools.com/

These sites were helpful for learning how to use and write AIML files for the chatbot:
http://www.pandorabots.com/pandora/pics/wallaceaimltutorial.html
http://www.alicebot.org/articles/wallace/dont.html

I used these forums for ProgramO and AIML support:
http://programo.smfforfree.com/index.php
http://forum.alicebot.org/

I borrowed the open-source (for non-commercial use) calendar from this site:
http://www.dynarch.com/projects/calendar/

This is the site for the ND EPICS Red Cross project that I referenced:
http://code.google.com/p/ndepics-redcross/