

Aloha Web Server documentation

Hervé Collin
(herve@hawaii.edu)

January 4, 2004

Contents

1	Overview of the Aloha Project	5
1.1	What is Aloha?	5
1.2	Requirements	5
1.3	Software Support	7
1.4	Download the Software	7
1.5	Is this software free?	7
1.6	Do I have to register?	8
1.7	How can I help?	8
1.7.1	Things that needs to be done	8
1.8	Authors and contributors	9
2	Installation	11
2.1	General requirement	11
2.2	Porting objects	11
2.3	Aloha	12
2.4	Hash_utils	13
2.5	Aloha Log	14
3	General Modules	17
3.1	Software content	17
3.2	Overview - how do I choose which modules?	17
4	Aloha Module	19
4.1	Overview	19
4.1.1	POST method	19
4.1.2	GET method	20
4.1.3	Aliases through GET	20
4.1.4	Parsing information	21
4.1.5	Example	23
4.2	Properties Description	24
5	Aloha Log Module	27
5.1	Overview	27
5.1.1	Direct feedback	27

5.1.2	Standard MOO server	28
5.1.3	Through FUP	29
5.1.4	Web Server Statistics	31
6	FUP Module	33
7	Known Bugs	35

Chapter 1

Overview of the Aloha Project

1.1 What is Aloha?

Aloha Web server is a package that enables a MOO to have a graphical interface. Through this GUI, one can have web display which pertains to the MOO environment. Currently, both GET and POST method are supported, and XML support is on its way.

Eventually, a full GUI will be available. Currently, only the bare bone system is available, on which everything else will be built on.

Be aware that this project has started and is being conducted in the direction of EDUCATION. Although, you are of course welcome to use it for whatever means you'd like, many features and modules will be for educational purposes primarily. If time allows or if other developers write new modules that are not necessarily towards educational purposes, they can of course still be added so that everyone can enjoy them.

One more thing: many of the features that will be available through the GUI will be FUP based. Hence, be ready to compile it if you want to use them.

1.2 Requirements

There are several requirements to be met:

- Although extensive MOO programming skill is not required to install Aloha (my case is the best example), a minimum is still required. For instance, one should be familiar with modifying properties of objects, and with the use of the eval() function (usually noted in this document as a semi column: `;$kahuna.property=1`).
- there are lots of room for improvement of current packages and especially for new modules that can add more functionalities. The best example is at

LogMOO where Aldon et Al. made numerous fixes and new features. In this case, a stronger MOO programming background is required.

- A MOO server and database (I was not sure I needed to put that one in, but since I'm at it...)
- HTML!! I know... I don't like HTML either, but it will be assumed that you know how to write your own. What is meant by knowing HTML is not knowing how to use an HTML editor; you will have to know what the basic HTML tags mean to understand what is being done during data transfer and in order to create your own utilities.
- The possibility to use one more port on the actual server. Indeed, the MOO server is usually accepting connections through the default port 7777 (which can be modified in **options.h** or **\$network.port**). Aloha will need another port through which packets coming from browsers, can be accepted to the MOO. Hence, Aloha will listen to a DIFFERENT port. That is, `;listeners()` will display another object (Aloha) and the port that is being used. It is therefore, imperative that the server on which the MOO system is running does not block that extra port (through firewall rules such as iptables or ipchains).
- It will be assumed through this document that a *nix box is used. Although, there is nothing (except for some FUP functionality) that prevents other OS to handle it. Be aware though that no support will be given for problems that arise from other OS than *nix (I know I just said there should not... but I was thinking about modules that use FUP). Some syntaxes in this document will therefore refer to *nix commands: simply find the corresponding for your own OS.
- For the bar bone system of Aloha there is no need for other Web server. However, Aloha only support TEXT based display. The GUI WILL need another web server to handle all the pictures and non TEXT based data. Apache is the one that has been used for this development; however, any should be able to fit the job task. I'd like to point out that you are on your own concerning the **httpd.conf** or any other configuration concerning the web server that you will use for pictures serving.
- If you already have a MOO web server installed, or you are using a database which came with a MOO web server and is running, there is nothing that prevents you from installing and running Aloha as well at the same time. You actually could use several version of Aloha at the same time if you need to. The only requirement is that each listen to its OWN port (otherwise, you will get in trouble).
- If you already have an old version of Aloha prior to 2.1 you can still install the newest without any conflict. This is one of the reasons it has been re-

named \$kahuna and that only this guy has been @corified. You therefore can have several version running at the same time.

1.3 Software Support

Any time you have a question or problem, please send it to me directly at herve@hawaii.edu or to the mailing list (alohamoo-1@hawaii.edu) whose purpose is exactly that. There is no stupid questions and all are valid when new to a system. If you want me to drop me your MOO to troubleshoot, it is no problem either. You most likely however will need to provide me with a temporarily \$wiz perm bit if things get tricky; it's your choice here. Just remember that most likely I do not "care" about your MOO (unless you give me valuable reasons to do so :), what you do with it, and have no intention, time or inclination to install back doors on it to do funky things. At worse, simply write a simple `#0:do_command()` and `#50:do_command()` script to log everything that I will be typing, and you can go back to sleep after you look at it.

In general, when time allows I will always be available to help, and connect to your MOO if needed.

One thing though: please, please, please, please... RTFM! AND make sure that you **Read The last version** of the "Fine" (well... ok I try my best here... give me some slack :) **Manual!** It is always frustrating and deceiving to troubleshoot something and realize that the problem was due to a basic property set up. It is also a waste of time. However, if you do not understand the manual, or find it confusing, please please please, let me know! ;) so I can fix it up. It's the first time I write a manual, so I don't know what I'm doing.

You can also get support by connecting to KCCMO'O at moo.kcc.hawaii.edu port 7777. If you see hErVe, Halwyn, or Cocole and Magnum (he doesn't know about it yet, but just remind him of this new fact ;) - he is way kool and a wonderful and solid programmeur), feel free to stop by.

1.4 Download the Software

Aloha Web Server can be found at: <http://moo.kcc.hawaii.edu/aloha>

All modules can be found there along with the patches, this documentation, the log and the FAQ.

1.5 Is this software free?

Yes, it is. Make sure you read the GNU license published on the web site.

1.6 Do I have to register?

No, you can download as many times as you want, without having to register, download patches and upgrades, etc...

There is a `register()` verb built-in to `$kahuna` which will attempt to use your `$network.maildrop` SMTP server if `$network.active` is set to 1, and will open a connection to `KCCMO'O` and send me a MOOmail from there as well. Several information will be sent out: your IP (from `$network.site`), your MOO name (from `$network.MOO_name`), your port (from `$network.port`) and the port on which Aloha is installed (from `$kahuna.port`). The script is triggered when `@start` is launched.

The purpose of this "nasty" little guy is to have a rough idea of how many people are using the software. Please do not disable it. If those connections might be a source of problems on your own network, then of course. But please send them to me manually. I will not broadcast those information on warez sites, or sell them out (can we do that?? ;) and will not use those information unless I receive specific confirmation from the user.

The reason I am doing this is because the box on which this package resides is from the University system. if one day I am being asked to justify how in hell I am using that box for, then I can show those numbers and say "see? that's why - it's useful, it's being used!".

And you do not need to use Aloha for educational purposes either. Use it for whatever you need. I do not care, and will not check on it to see what you are doing.

1.7 How can I help?

Create an account at <http://sourceforge.net>

and join the project at: <http://sourceforge.net/projects/alohamoo/>

You can also contribute with \$\$, which would allow us to purchase our own domain name (that would be cool ;) - something like: <http://www.alohamoo.com?> ;) See SourceForge site for this purpose.

A great way to contribute though would also be to look at the codes, and make them more efficient, write improvement, new features, and new modules! that's the way to go! :)

1.7.1 Things that needs to be done

You can work on anything you'd like of course, but I just want to list out all the stuff we came out with, or bumped to, and put it aside because of lack of time, or energy. However, those things will eventually be done... (one day). If there is anything that interests you in there, jump right on it!

- `$kahuna:do_login_command()` has three other cases to complete: XML, POST without having to use `multipart/form-data`, and POST using `Content-type: text/plain`.

- Allow Any object name to be used and called through a URL without having to use `$kahuna.name2objnb`
- Non ASCII support for URL (relates to the above one).
- Add user IP in the environment variables

1.8 Authors and contributors

The original author of this free software is Mark Horan who wrote the first version probably in one time between two cigarettes (although I think he quit by now) in 1997. Many revisions were made on it, and in 2001 the first public version was released. Although Mark has moved to other projects and does not maintain it, he remains the guru and creator of this project in our minds.

Judi Kirkpatrick saw some potential in it, and has been until now the main support, both technically and psychologically for further developments. She is the one who has to go through most of the problems and bugs (bad karma :), and she has been the primary user of it for educational purpose to teach her online courses.

Then, there is bibilolo, hidden behind my screen, trying to understand what I am doing and with more difficulties trying to understand the codes I wrote back several months ago when I want to improve the script (brain farts are built-in modules of my brain).

And there is Aldon, the first person who seriously invested some time and effort to improve the software and to make significant input. Most of them, I admit are way above my head, and I do not understand them. He can be found at Lambda and LogMOO where Aloha has been running for several years.

Finally, there are all the volunteers who devoted their free time to play with it, fix bugs, and develop new packages such as (in order of appearance in the project): Laurent Lainé, George Hager, Bradley Dilger, Papa and Jane.

Thank you to all those people!!!

Chapter 2

Installation

2.1 General requirement

There are several things you need to make sure before starting an installation. Some are required, and some are strongly suggested.

- You need a compression software as all modules are compressed with gz. On a *nix platform, all you need to do is to use the command: *gunzip *.gz* with * being the name of the package. On other platform, you will need other decompression tool: pick your favorite one.
- You will need a text editor which does *NOT* wrap. Hence, turn it OFF before porting objects!
- We stringly suggest that you use a MOO editor to port object. Telnet is usually a bad idea and will make your life miserable. Good free MOO clients are TkMOO and tf.
- When porting objects, do not copy and paste the whole thing at once! We strongly suggest that you port all the properties first, then one verb at the time. Make sure you watch your MOO responses after each porting.

2.2 Porting objects

Porting objects refers to the action of transferring objects from one MOO to the other. You usually need to dump the object with some arguments first (with create id=) then copy the whole thing and paste it into the target MOO. We already did half of the porting. So all you have to do to install modules is to copy the content of the modules form your text editor where you opened them, and port them into your MOO.

2.3 Aloha

Download the module called Kahuna_x.x.txt.gz (with x being the version number). Decompress the package using: `gunzip Kahuna_x.x.txt.gz` - Upon decompression you will be left with one file called: `Kahuna_x.x.txt` - Open it up with your favorite text editor.

From your MOO, create an object. This object will be used for Aloha:
`@create $thing named kahuna`

Make sure that you type: `kahuna` EXACTLY (no other name will work for the updates - no UPPERCASE letters neither).
 The MOO will answer something like:
 You now have `kahuna` with object number `#xxxx` and parent generic thing (`#5`).

Where `#xxxx` is the object number used by your MOO (can be `#1234` or `#5678` etc..)

Next, we need to corify the object; type:

`@corify #xxxx as kahuna`

Where you would replace `#xxxx` by your OWN object number.

Your MOO will say something like: *Kahuna has no :init_for_core verb. Strongly consider adding one before doing anything else.* - this is OK.

Let's check that everything is kool by typing:

`@examine $kahuna`

If everything is OK, you will see a short description of your object. So now you can always refer to this object using `$kahuna` instead of remembering its object number.

Highlight all the properties from `Kahuna_x.x.txt.gz` (lines that start with `;;`), copy them and paste them to your MOO. You should see a bunch of "Property added with value ..." and some "`=>0`".

Highlight one verb at the time, copy them and paste them into your MOO.

Do NOT run any verb or start it until you install `Hash_utils` object.

2.4 Hash_utils

Download the module called Hash_utils_x.x.txt.gz (with x being the version number).

Decompress the package using: `gunzip Hash_utils_x.x.txt.gz` - Upon decompression you will be left with one file called: Hash_utils_x.x.txt - Open it up with your favorite text editor.

From your MOO, create an object. This object will be used for Hash_Utils:
`@create $thing named hash_utils`

Make sure that you type: hash_utils EXACTLY (no other name will work for the updates - no UPPERCASE Letters neither).

The MOO will answer something like:

You now have kahuna with object number #yyyy and parent generic thing (#5).

Where #yyyy is the object number used by your MOO (can be #4321 or #8765 etc..)

Next, we need to corify the object; type:

`@corify #yyyy as hash_utils`

Where you would replace #yyyy by your OWN object number.

Your MOO will say something like: *Hash_utils has no :init_for_core verb. Strongly consider adding one before doing anything else.* - this is OK.

Let's check that everything is kool by typing:

`@examine $hash_utils`

If everything is OK, you will see a short description of your object. So now you can always refer to this object using \$hash_utils instead of remembering its object number.

Highlight all the properties from Hash_utils_x.x.txt.gz (lines that start with ;;), copy them and paste them to your MOO. You should see a bunch of "Property added with value ..." and some "=>0".

Highlight one verb at the time, copy them and paste them into your MOO.

2.5 Aloha Log

Download the module called Aloha_log_x.x.txt.gz (with x being the version number).

Decompress the package using: `gunzip Aloha_log_x.x.txt.gz` - Upon decompression you will be left with one file called: Aloha_log_x.x.txt - Open it up with your favorite text editor.

From your MOO, create an object. This object will be used for Aloha Log:
`@create $thing named Aloha Log`

Make sure that you type: Aloha Log EXACTLY (Capital A and Capital L). The MOO will answer something like:
 You now have kahuna with object number #zzzz and parent generic thing (#5).

Where #xxxx is the object number used by your MOO (can be #1123 or #5813 etc..)

This object is NOT corified! Hence, we need to know Aloha where to get it, type:

```
;$kahuna.modules={$kahuna.modules, {"Aloha_Log", #zzzz}}
```

Where #zzzz is the one your MOO gave you when you created the object. Make sure also that you type EXACTLY: Aloha_Log as it is (Capital A and Capital L with an underscore).

Before porting the codes, you need to go back to your text editor where Aloha_Log codes are, and do a find/replace. However, we do not want to replace them all. So you will need to perform the find/search thingy 4 times:

- Find ALL occurrences of the word `@prop Aloha_Log` and replace them with `@prop #zzzz` (where #zzzz is the one your MOO gave you when you created the object).
- Find ALL occurrences of the word `;;Aloha_Log` and replace them with `;;#zzzz` (where #zzzz is the one your MOO gave you when you created the object).
- Find ALL occurrences of the word `@verb Aloha_Log` and replace them with `@verb #zzzz` (where #zzzz is the one your MOO gave you when you created the object).
- Find ALL occurrences of the word `@program Aloha_Log` and replace them with `@program #zzzz` (where #zzzz is the one your MOO gave you when you created the object).

The reason why we did all these is because there are occurrences of the word Aloha_log in the codes which we need to leave untouched.

Now, we are ready to port the object.

Highlight all the properties from Kahuna_x.x.txt.gz (lines that start with ;;), copy them and paste them to your MOO. You should see a bunch of "Property added with value ..." and some "=>0".

Highlight one verb at the time, copy them and paste them into your MOO.

Now, refer to section 5.1 to configure it to your liking.

Chapter 3

General Modules

3.1 Software content

This software comes in Modules. Some are optional, and some are required. The idea is that you will not need to download and install a monster bugger, end up only using some of it, but having to cop with all the crap you don't like. Instead, according to your needs, download and install only the required and needed modules.

You have to make sure of the dependencies of those modules though, which is given on the download page of the Web site.

There are 3 packages available at this time:

- Aloha
- Hash_Utils
- Aloha Log

Aloha (which is called \$kahuna) and Hash_Utils are the two required built-in stone of the software: You have to install those two modules before doing anything.

3.2 Overview - how do I choose which modules?

Aloha and Hash_Utils You do not choose whether you want to install Aloha and Hash_utils, you just have to install both before doing anything else.

Aloha Log This modules allows you to have Log of all traffic coming to the Web server.

It logs all incoming traffics, but not outgoing.

There are different ways of logging your packets: some are basics, and some are sophisticated and neat. The later one REQUIRES FUP to be installed (see section

6).

Chapter 4

Aloha Module

4.1 Overview

Aloha handles both GET and POST methods, which can be used separately or together. For instance, you can pass variables through an HTML FORM and at the same time, more variables through the URL.

Everything will go first through **\$kahuna:do_login_command()**, and from there, will be dispatched towards **\$kahuna:parse_post()** and **\$kahuna:parse_get()** depending on the type of method used.

4.1.1 POST method

There is nothing special about this method except ONE thing. Standard HTML scripting to pass the variables can be used using NAME and VALUE etc...

However, at this point, when using this method, you have to use ENCTYPE=multipart/form-data in the FORM tag. An example would be:

```
<FORM METHOD=POST ENCTYPE=multipart/form-data  
ACTION="http://moo.kcc.hawaii.edu:8888/612/login">
```

The above example implies that Aloha is running on port 8888 and that the content of the FORM values will be sent to the verb named login() on the object #612.

Hence, the general syntax is:

```
http://server:port/object/verb/
```

See **\$kahuna:do_login_command()** for the case where ENCTYPE=multipart/form-data is not included. I am having a problem with this. If you are interested in working on this, feel free coz I gave up (see section: 1.7.1).

There is also a case for Content-type: text/plain, but I haven't developed it and have no idea how it would behave at this stage. Finally, there is an XML case that

Aldon wrote, which I know nothing about. Talk to him about it. Hopefully, he will look into it soon and write something up along with some example scripts.

4.1.2 GET method

The standard way to use this method is as follow:

```
http://server:port/object/verb?var1=value1&var2=value2
```

All variables are passed within the URL, have to be separated by & and have to be composed of their name and values (var1=value1). Hence, = and & are important. Passing variables like: verb?testvalue will not be caught; it needs to be defined by a name: verb?testvar=testvalue.

There is yet, another way to pass variables. This is for authentication purposes only; it goes as follow:

```
http://server:port/object?user=value1&passwd=value2
```

Hence, no verb is required. Now of course, there must be a proxy somewhere along the line to redirect to one if none is provided: it's called **Aloha_default()** and will need to be written on object #1. Eventually, several of them will be provided for specific objects, such as #3, #6, etc...

A singular case is the default page:

```
http://server:port/
```

Here again, a target verb needs to be defined. If **.html_path** is defined, then **default_index** will be used, which implies that the index file will be a pure static HTML file somewhere on your box, handled by apache or any standard web server.

Aloha is a standalone Web server. However, it can deliver only TEXT (at this point). Therefore, if you have the need to server pictures, you will need to rely on other web server running on port 80 (or other). The GUI WILL need apache to display the buttons for instance. However, you can always change that and replace them with text in order to avoid this issue.

4.1.3 Aliases through GET

Aldon made a nice suggestion of being able to replace the object number by their actual name. He also wrote the codes for it, which have been implemented.

There has to be a special mapping for those as many object will have words like: "big blue", or "blue box"; that is, with a space in there. In URLs, there can not be any space present. There were thoughts of filtering all names are replacing any space or weird character with underscore, for instance. This could be done in the future (see section 1.7.1).

The mapping is done through **\$kahuna.name2objnb** where you can enter by hand all object with their names as long as it's a one-word name.

Since Player's name are one-word name, they are automatically considered by

\$kahuna:parse_get() and do not need to be included in **\$kahuna.name2objnb**; yet, make sure there is no , \$ or other weirdos in your name. If you have accent support in your MOO, just forget about it.

4.1.4 Parsing information

In a very simple way... watch ;)

Aloha will deliver any verb with a bunch of information, some asked for, and some not, but which will come along: the environment variables (bunch of stuff coming from the user's browser).

Here is an example:

```
args = {{{}, {"username", "this is a test"}, {"login", "moa"}, {"passwd",
"niark"}}, {"SERVER_SOFTWARE", "Aloha Web Server - Version 2.2"},
{"GATEWAY_INTERFACE", "CGI/1.1"}, {"SERVER_PROTOCOL", "HTTP/1.0"},
{"REQUEST_METHOD", "POST"}, {"SCRIPT_NAME", "submitest"},
{"HTTP_REFERER", "http://127.0.0.1:2222/97/test?login=moa&passwd=niark"},
{"HTTP_USER_AGENT", "Mozilla/4.08 [en] (X11; I; Linux 2.4.20-24.9 i686;
Nav)"}, {"SERVER_NAME", "127.0.0.1"}, {"SERVER_PORT", "2222"},
{"HTTP_ACCEPT", "gzip"}, {"HTTP_LANGUAGE", "en"}, {"HTTP_CHARSET",
"iso-8859-1,*utf-8"}, {"CONTENT_TYPE", "multipart/form-data; boundary=—
—————20363914401191307546892998980"}, {"CONTENT_LENGTH",
"189"}}
```

Ok, maybe I did not choose the best clearest approach, give me another chance. Let's take out all the blabla and look at the structure for a moment.

```
{{{ }, {Data from POST}, {Data from GET}}, {{Environment variables}}
```

So pretty much it gives back a LIST {}, which is composed of two LISTS inside: {{{}, {}}. The first one is for variables that you pass along through GET or POST, and the second one is for environment variables.

The environment variables are generated by **\$kahuna:get_Env()**, which you can modify at your convenience (see section 1.7.1).

Now all variables passed come in pairs: variable name, and variable value. In GET, they come under the cover: var=value while in POST, they could come from the HTML file as NAME="var" VALUE="value". So each variable and its value will come as STRING within a LIST: {"username", "this is a test"}. If there are more than one values for one variable name, they are just appended: the first one will be the name, all the other will be the values: {"username", "this is a first test", "and a second one"}.

If you pass more than one variables in GET, you will have two LISTS: {"login", "moa"}, {"passwd", "niark"} where on my URL, they appeared like:

`http://server:port/97/test?login=moa&passwd=niark`

By now I am sure you start getting the picture.

So to summarize:

```
;args[1] = { {}, { "username", "this is a test" }, { "login", "moa" }, { "passwd",
"niark" } }
;args[2] = "SERVER_SOFTWARE", "Aloha Web Server - Version 2.2",
{ "GATEWAY_INTERFACE", "CGI/1.1" }, { "SERVER_PROTOCOL", "HTTP/1.0" },
{ "REQUEST_METHOD", "POST" }, { "SCRIPT_NAME", "submittest" },
{ "HTTP_REFERER", "http://127.0.0.1:2222/97/test?login=moa&passwd=niark" },
{ "HTTP_USER_AGENT", "Mozilla/4.08 [en] (X11; I; Linux 2.4.20-24.9 i686;
Nav)" }, { "SERVER_NAME", "127.0.0.1" }, { "SERVER_PORT", "2222" },
{ "HTTP_ACCEPT", "gzip" }, { "HTTP_LANGUAGE", "en" }, { "HTTP_CHARSET",
"iso-8859-1,*utf-8" }, { "CONTENT-TYPE", "multipart/form-data; boundary=—
—————20363914401191307546892998980" }, { "CONTENT-LENGTH",
"189" }
```

Similarly,

```
;args[1][2] = { "username", "this is a test" }
;args[2][1] = { "SERVER_SOFTWARE", "Aloha Web Server - Version 2.1" }
```

Ok, so I'd assume you are now expert. Once you see the global picture, it's pretty simple uh? And of course, at this very moment, I can see your arm raising in the air, fusing like a rocket:

what is `;args[1][1]??`

Aaaarrghhh!!!! :(

Sorry, I forgot... I **think** I added up for the variables passed like:

`http://server:port/object?var`

which will be used for authentication. But I never tested it yet. Stay tune; but I'm glad you asked that question, it shows me that you are slick and sharp and that I'm not talking to walls here :)

So anyway, as mentioned above, you do not have to pass BOTH GET and POST. Let's look at another more concret example:

4.1.5 Example

HTML SCRIPT:

Since I'm at it, I will put the syntax to generate an HTML document FROM the MOO itself.

```

page = {};
page = {page, "<HTML>"};
page = {page, "<BODY>"};
page = {page, "<FORM METHOD=POST ENCTYPE=multipart/form-data
ACTION=\`http://moo.kcc.hawaii.edu:8888/123/testverb/\`>"};
page = {page, "Enter the value: <INPUT TYPE=TEXT NAME=\`var\`>"};
page = {page, "</FORM>"};
page = {page, "</BODY>"};
page = {page, "</HTML>"};
$kahuna:ok(page);

```

hopefully, you can see where the line wraps.

So pretty much you create a LIST (in this case called page), you append it with new HTML lines, and when finish, shovel it to **\$kahuna:ok(page)**. Note that if you need to insert quotes, you will need a backslash in front of it (see the MOO programmeur's Manual).

The syntax might therefore be kind of repellent... I know, but do not fear, as they is a script available and called :make_HTML(), which can do all this for you, so all you have to do is use your favorite HTML editor, and shovel your HTML into the script to obtain the above.

Now observe what we can do, say on line 4:

```

page = {page, "<FORM METHOD=POST ENCTYPE=multipart/form-data
ACTION=\`http://", $network.site, ":8888/123/testverb/\`>"};

```

Do you see it? :)

So, now you can create dynamic Web pages, by embedding any property or verb results obtained from your MOO server and display them along.

Ok, let's go back to the MOO codes that will receive the variable var. Following the above method, I usually start my codes by grabbing and sorting the variables, so I can retrieve them easily when I need to. Example:

MOO Codes:

```

hash = args[1][2];
env = args[2];
variable = $hash_utils:fetch(hash, "var");
page = {};
page = {page, "<HTML>"};
page = {page, "<BODY>"};
page = {page, "You are using ",
    $string_utils:explode($hash_utils:fetch(env, "HTTP_USER_AGENT"), " ")[5],
    " and passed the variable: ", variable, "."};
page = {page, "</BODY>"};
page = {page, "</HTML>"};
$kahuna:ok(page);

```

(again, I hope you can see where the line wraps). I know I'm expecting the POST method (hey... we just wrote the HTML), so I'm fetching specifically the POST LIST part, which is `args[1][2]`. `$hash_utils:fetch()` is one that can extract easily what you need. I also need something from the environment variables, so I fetched it the same way, explode it through the space character and used the fifth elements, which was the word "Linux".

Happy Coding!!!

4.2 Properties Description

I will only comment on the properties. The verbs, you can read and understand yourself. I think commenting the properties will help more understanding how the object works, by giving pieces of the puzzle. You will put them together depending on how your brain works.

```
;;$kahuna("port") = 2222
```

*port through which Aloha is listening and towards which you point your browser. For Web server using port 80, the port doesn't need to be explicitly written, but browsers understands it's 80. Hence, to force your browser to make contact to a different port, you have to specify it. Same syntax as **\$network.port**, which means: it's NOT a STRING.*

```
;;$kahuna("operator") = #2
```

This is guy to whom log and feedback that @test and @update will talk to. It's NOT a STRING

```
;;$kahuna("logging") = 0
```

If set to 0, no login occurs. If set to 1, logging occurs (see section: 5.1). Remember that it's like a cascade thingy. This one needs to be set to 1 for all the following properties on logging (**\$kahuna.directlogging**, **\$kahuna.standardlogging**, **\$kahuna.fuplogging**, and **Aloha_log.error_log** to be active.

```
;;$kahuna("directlogging") = 0
```

Two values are possible: 0 and 1. It will allow instanteneous output of logs on the MOO prompt to \$kahuna("operator"), (see section: 5.1.1)

```
;;$kahuna("standardlogging") = 0
```

Two values are possible: 0 and 1. It will allow logs to be saved in the standard MOO server log, (see section: 5.1.2)

```
;;$kahuna("fuplogging") = 0
```

Two values are possible: 0 and 1. It will allow logs to be saved through FUP, which, of course, needs to be installed (See section 6.

```
;;$kahuna("modules") = {"Aloha", $kahuna}
```

This property is extremely important. If it gets corrupted most of the system will get broken. Hence, **@test \$kahuna** is here to fix it up. This verb also has to be executed everytime you install new modules, just to make sure. It essentially replaces the @corifying. It is a LIST, composed of subLISTs, each having two variables: a STRING and an OBJECT. So the minimum that you should have on it is: {{"Aloha", \$kahuna}, {"Hash_Utils", #1234}} where #1234 is the object where you installed Hash_utils object. I would not edit this guy by hand.

```
;;$kahuna("name2objnb") = {}
```

Same structure as **;;\$kahuna("modules")**, which will contain the mapping of object numbers to one-word name. You might want to put there all the commonly used object accessed by the Web, see section 4.1.3

```
;;$kahuna("server_software") = "Aloha Web Server - Version 2.2"
```

Used by **;;\$kahuna:get_env()** to generate one of the environment variables

```
;;$kahuna("html_path") = ""
```

This is for later. When the GUI will be available and there will be the need of apache to support png or jpg file delivery (buttons and stuff); it will also be required to point where the JAVA applet MOOca will be in order to connect from the Web. I will assume here *nix syntax for PATH.

```
;;$kahuna("default_index") = "index.html"
```

Points to the index page when pointing your browser to: `http://server:port/` that resides on any apache public directory, which needs to be on the same box as the MOO server. If **\$kahuna("html_path") = ""** then Aloha will generate it.

```
;;$kahuna.("outgoing_packets") = 0
```

*Technically this guy should not be defined on this object, but I didn't have the heart to include an if statement as in **\$kahuna:do_login_command()**, fearing to slow down delivery processes. It works just as well and is used by Aloha Log to generate the traffic summary page.*

```
;;$kahuna.("ticks_threshold") = 100
```

*this property is used for the following statement: (ticks_left() ; \$aloha.ticks_threshold) && suspend(0)} which appear many many times along with \$command_utils:suspend_if_needed(0). The former syntax was suggested by George Hager for **\$kahuna:ok()** and produces a significantly faster execution of scripts. The problem resides when large content of information is being parsed as an error message complaining about insufficient number of ticks blabla.. which will force you to increase this property. I do not fully understand this issue, but there is a fix for it: see section 7 ;*

```
;;$kahuna.("help_msg") = {"Moo Web Server", "The documentation and sources  
can be found at:", "http://moo.kcc.hawaii.edu/aloha"}
```

When you get stuck, and need help big time!....;)

```
;;$kahuna.("version") = "1073076383"
```

*time at which the bugger as compiled. Uses the standard ;time() btw. All object have one (even FUP, make sure you see section 6!!) and is being used by @getupdate **\$kahuna***

```
;;$kahuna.("aliases") = {"kahuna"}
```

In case you forgot that Aloha's middle name was Kahuna.

Chapter 5

Aloha Log Module

5.1 Overview

The log system is handled by Aloha_Log object. Refer to the installation chapter on how to install it. Once ported, make sure that you run:

```
@test $kahuna
```

to configure it properly. So really, all that follow is for troubleshooting. The only reason you will want to get your hand dirty by changing the properties below is if **@test** has brain fart (which I hope not, but in case, let me know), or if you'd rather do it yourself to get a better understanding of what is going on (which I completely understand... I really had to drag myself into writing **@test**).

There are three ways to log incoming packets; all methods can be configured through the @configure script. For ALL methods to be activated, **\$kahuna.logging** must be set to 1.

- Direct feedback
- Standard MOO server
- Through FUP

5.1.1 Direct feedback

This method of logging is controlled by **\$kahuna.directlogging** which must be set to 1. This is the method that is usually used for trouble shooting. It allows one to monitor all the packets coming in the MOO. To monitor, connect to the MOO and watch. If you see no output, make sure that **\$kahuna.operator** is pointed to the player that you are using to monitor.

if **\$kahuna.directlogging** is set to 1 but **\$kahuna.logging** is set to 0 no logging

will occur.

5.1.2 Standard MOO server

This method of logging is controlled by `$kahuna.standardlogging` which must be set to 1. This method will log everything on the standard MOO server log file. This log file is defined when the server is started; usually with option `-l` (see the README file of the MOO server package). All lines concerning Aloha will have a CAPITAL ALOHA to identify themselves. A typical log from Aloha would look like:

```

Jan 1 19:51:38: ; ALOHA: POST /97/submittest HTTP/1.0
Jan 1 19:51:38: ; ALOHA: Referer: http://127.0.0.1:2222/97/test
Jan 1 19:51:38: ; ALOHA: Connection: Keep-Alive
Jan 1 19:51:38: ; ALOHA: User-Agent: Mozilla/4.08 [en] (X11; I; Linux 2.4.20-
24.9 i686; Nav)
Jan 1 19:51:38: ; ALOHA: Host: 127.0.0.1:2222
Jan 1 19:51:38: ; ALOHA: Accept: image/gif, image/x-xbitmap, image/jpeg, im-
age/pjpeg, image/png, */*
Jan 1 19:51:38: ; ALOHA: Accept-Encoding: gzip
Jan 1 19:51:38: ; ALOHA: Accept-Language: en
Jan 1 19:51:38: ; ALOHA: Accept-Charset: iso-8859-1,*,utf-8
Jan 1 19:51:38: ; ALOHA: Content-type: multipart/form-data; boundary=————
—————1133921644773919629873420913
Jan 1 19:51:38: ; ALOHA: Content-Length: 183
Jan 1 19:51:38: ; ALOHA: —————1133921644773919629873420913
Jan 1 19:51:38: ; ALOHA: Content-Disposition: form-data; name="username"
Jan 1 19:51:38: ; ALOHA:
Jan 1 19:51:38: ; ALOHA: this is a test
Jan 1 19:51:38: ; ALOHA: —————1133921644773919629873420913—

```

However, the logs from Aloha will be mixed up with standard MOO log, obviously... which might make it hard to read, and/or time consuming to scan manually. A cheap way to extract them, could be:

```
less moo.log | grep " > ALOHA:" >> aloha.log
```

where `moo.log` is your MOO server log file, and `aloha.log` will be the file into which all logs from Aloha will be saved into. Now, you can view only the log from the Web server using your favorite text editor, and opening `aloha.log`. Again, even though `$kahuna.standardlogging` might be set to 1, logging will not occur if `$kahuna.logging` is set to 0.

5.1.3 Through FUP

This method of logging is controlled by `$kahuna.fuplogging` which must be set to 1. This is most difficult to set up if one is not familiar with FUP. This method however is the most flexible of all as it allows one to be able to save Aloha logs directly into any designated files. Of course, this file is restraint to the directory `\files` (assuming that you kept the FUP default; refer to `files.c` to be certain).

The version of FUP that needs to be used for the logging system HAS to be higher than version 1.8.1 although we will assume that version 1.9 is installed. Make sure that you read the section about FUP when installing! There is one property to add for `@getupdate()` to work correctly, which is not cited in the FUP README as it concerns Aloha specifically.

There are actually two form of logs. We call them:

- Static method
- Dynamic method

Static method

Controlled by `Aloha_Log.static` which has to be set to 1. If so, there are two properties that handle the file name and its location: `Aloha_Log.log_file` and `Aloha_Log.log_dir`. Those two properties are STRING. This method will therefore save the log into a file that you designate and it will always be the same unless you change them. The drawback of this method is that log files *might* get big, and if you do not write your own *cron* file to archive them regularly, it may take quite a while to search what you are looking for. Of course, *grep* is always here to help us out, but to solve that problem, the dynamic method can be used.

Dynamic method

Controlled by `Aloha_Log.dynamic` which has to be set to 1. In this case, `Aloha_Log.log_dir` STILL has to be set, but `Aloha_log.log_file` is no longer needed. It is up to you now to decide the period of time the same file name should be used.

The choice that you will make obviously (or may be not) depends on the amount of traffic that you expect. To make that choice taken in effect, check `Aloha_Log.dwm` and use: "day", "week", or "month".

There is nothing to prevent you to use more than one method (I always use the three of them). So, several examples of this settings would be:

```
:Aloha_log.dwm={"day"}
:Aloha_log.dwm={"day", "month"}
```

The order of the STR arguments does not matter.

You could decide to use a file name for each day.

For instance, if today is January 1, 2003 then **Aloha_log_01012003.log** is going to be used as the file name and all logs will be saved for that day on that file. The next day and new file will be created: **Aloha_log_01022003.log** and so on. This option will allow you to trouble shoot logs faster if you know when the problem was and will allow you to delete them easily as well. For example, if you decide to erase all logs for January, `rm -rf 01*.log` will do the job cleanly.

You could decide to use a file name for every week.

Again, similar thing will happen and **Aloha_log_startstop_2003.log** is going to be used with **startstop** being the length of seven days. So it will look something like: **Aloha_log_0101_0107_2003.log**. Please note that the above example has log from January first (00:00) until January 7 included (midnight) and I would not see how you would want to modify it (but you do whatever you want :).

Now, really, you ought to give the system a starting date, otherwise it will not know from which day to start counting: this is handled by **Aloha_log.weekinit**. This property is of the format of `time()`. You definitely want to choose a number that correspond to a date that is synchronized to 00:00, otherwise everything will be shifted as the number of seconds to cover a full week will be used. So, if you REALLY want to change that by hand, a cheap by cumbersome way to find the time you need is to use something like: `;ctime(number_here)` where `number_here` is obtained from `;time()` and modified according to your needs.

By default, **Aloha_log.weekinit** is set to 1041415200, which correspond to Wed Jan 1 00:00:00 2003 HST.

You could decide to use a file name for every month.

Same idea here as above, except that now **Aloha_log_Jan2003.log** will be used as a file name and so on.

Side note: Technically, there is nothing that prevents you to use both Static and dynamic methods (although using both might be overdoing it, especially if you already combine methods of logging through **Aloha_log_dwm**).

Last thing: you really have to make sure that the directory EXISTS as the log script will not create it for you (I would rather leave that to you rather than letting the script creating directories on the system...).

Of course, the package FUP has to be installed. So, refer to the section concerning FUP. Once installed, **\$kahuna.modules** should have a new entry: { "FUP", #1234} (assuming that #1234 is the object number used for FUP on your system) if you executed the @test script accordingly.

IT is strongly suggested that while testing if the log works, you enable Direct logging, so that when checking if the FUP method is working, you will be able to see what kind of errors the system is running into IF it bumps into one... (The errors are listed in this document).

5.1.4 Web Server Statistics

A Statistics page is available to check what kind of work is doing the Web Server. It will check the packets coming out (that is the amount of pages being displayed), the ones coming in (through GET and POST) and the amount of error pages generated.

You can access it through your browser:

`http://server:port/xxxx/usage`

Where server is your \$network.site, port is your \$kahuna.port and xxxx is the object number for Aloha Log.

The statistic page is update ONLY is **\$kahuna.logging** is set to 1. In order to keep **\$kahuna:do_login_command()** as minimal as possible, the counting of the GET packets has been forwarded to **\$kahuna:export_log()**, which takes a little more work and use more ticks, but since the task is forked, it does not affect the traffic.

The GET value is controlled by **\$kahuna.incoming_get_packets**.

The POST value is controlled by **\$kahuna.incoming_post_packets**.

The error values are controlled by **\$kahuna.error_log** whose VALUES being the errors type encountered and the object for which they occurred (400, 404, etc..).

Note that the error log are not logged separately.

Chapter 6

FUP Module

FUP is not part of Aloha. It has been originally written by ...

Chapter 7

Known Bugs