ANNABELLE CONTENT MANAGEMENT SYSTEM

(c) Sabu Francis, India
Aug 1, 2007

Introduction
This is the Annabelle Content Mangement System (ACMS). It is written entirely in Rebol (http://www.rebol.com), with some parts in Javascript using the Jquery library.

ACMS can rely entirely on text files, so it is very simple to configure and understand. The actual data is kept off the server root; so chances of hacking the contents are minimal. It uses an object oriented paradigm and it also places the OOPs into a M-V-C framework. So one can have a tremendous amount of flexibility and control.

The "Annabelle" in ACMS is the name of my eldest daughter. One meaning of Annabelle is "graceful".

Prerequisites
ACMS works on any server which can execute CGI apps of the "first line contains a hash-bang (#!) path to the executable" variety. (Currently, I can only think of Apache. That is the only server that uses hash-bang for Windows too)

ACMS is NOT a complete off the shelf content mangement system. It is a powerful library. It is also an extremely terse (short) library. Learning it mainly requires an understanding of how ACMS invokes OOPs in Rebol. Actually, Rebol does not have inheritence the way C++, Java, Ruby, etc. programmers are used to... it uses prototypes instead. In ACMS, the normal tree shaped folder structure is used to collect the prototypes of classes to achieve inheritence. Using special files stored in a tree of folders, ACMS becomes a single-inheritence system quite a lot like in Ruby, etc.

An ACMS website is written by writing one or more Rebol CGI application/s which takes advantage of the ACMS library and the concepts of OOPs in ACMS. In order to package a complete content management system that uses ACMS, you would need both ACMS, the CGI applications and also some javascript files, including the Jquery library.

Installation is merely copying these files to the server, setting some chmod restrictions and probably setting a few .htaccess files. Nothing else is required ... unless, someone made a specialized ACMS based site that also used other stuff like MySQL, etc.

ACMS can also work with the Magic! Rebol framework written by Olivier Auverlot. All additional logic required for the specific website is coded using Rebol with the Magic! extensions. Instead of Magic!, ACMS could also use other Rebol web-frameworks (maybe RSP?) but I have not tested those. To use Magic! (and I suspect other Rebol web frameworks too), the server MUST support the "AddHandler" syntax of Apache for automatically handling (routing) files with a given extension through a CGI script. Again, I can only think of Apache doing this correctly.

Why?
I got quite tired of the complications of configuring bloated PHP+MySQL based CMS websites. I wanted something that is very basic, yet extremely powerful; which works on ASCII files as its data store, and which uses the DRY (Don't Repeat Yourself) principle. Though SQL is a very powerful concept as a domain specific language, I personally think inserting an active language (SQL) inside another one (say PHP or Perl) is an invitation for trouble. Sure enough, I've experienced hackers take on the challenge of hacking into such sites by sending mal-formed SQL to the site. Websites that I've written in Drupal, etc. were hacked into quite mercilessly. Most MySQL based sites have security warnings, and often the reason for upgrade is some hackers getting in through some chink in the security process.

I looked hard at what plain-jane regular HTML based websites were in the old days: A bunch of HTML files hanging at the ends of various folders in the tree structured folders of the site. There were two advantages: a) The system need not handle the entire data all at once. If the entire data were to be handled together, that is when indexing of the data becomes necessary. But if I squirrel away bits of data all over the site then Just-in-time (i.e. late binding) retrievals become possible. The second advantage is more subtle: b) I can use the tree structure to implement an OOPs system.

I also wanted something that was extensible infinitely. ACMS can be very easily extended by simply inventing a data structure for the objects required for that extension, a folder (or folders) for placing those objects, one or more CGI applications that can process the new objects and some Javascript glue material written in jquery. Installing any extension would be simply copying them into the right places and chmod the cgi program(s).

Most CMS mixes up the content with its presentation -- something that always leads to complications at some point or the other. I wanted a simple Model-View-Controller system. I tried Ruby on Rails, which I do admit is very powerful; but I find the terseness of Rebol even better.

Background
ACMS was in fact started a long time back. It was not written in Rebol. The first version was in an interpreted version of Prolog, called WebProlog. I may release that version separately. It used to be called "Captain Web" and it was used to develop all the websites of a company I had helped found. The concepts of OOPS were verified in Captain Web. This method of "OOPs" can also be seen in my architectural design software, TAD and it is quite unique to design softwares.

Caveat
I am not a programmer in the strict sense of the term. I am an architect by training, profession and passion. I use programming languages to further my objectives in my profession. So I approach computer languages completely from a different perspective. Hopefully from a position of innocence and not that of ignorance. If it looks I have re-invented something, I would appreciate it being pointed out; so I can make corrections.

OOPS in ACMS
The folder structure of a website is used to describe the OOPS tree in ACMS. The root folder of the web is also the root of the OOPS tree. In ACMS, these folders are referred to as "nodes of the OOPs tree" At every folder (and sub-folders ... oops nodes ... thereof) beneath the root, there exists three files.

.inheritable.class
.nonInheritable.class
.objects

These files are "dot" files so they are hidden and naturally protected by Apache. For a greater protection, they should be made inaccessible for everyone. Only the Rebol executable should ideally be able to write to those files. No other daemon need have access to those files.

Watch out for the "case" in case of Unix/Linux installations.

.inheritable.class
All the instance variables and methods that can be passed down the folders are given here. So if you have a folder structure like this:

```
/
|
|
 articles
        |
        |
        |
        2007
```

As told before, each folder on that tree is called a "node" in ACMS.

If you write into the *.inheritable.class* file at the "/" node, the following Rebol expression

doit: func [ ] [print "this or that"]

Then the doit func is available even at /articles and also at /articles/2007

Now this can be overwritten by another definition of doit either in a particular node's "*.nonInheritable.class*" file or even from within the objects you may be creating at any of node in that branch. You will understand that later.

.nonInheritable.class
In this file, write variables and methods that is applicable only at the node in question. They don't get passed down the folder tree. So if at /articles I write into the .nonInheritable.class file the following

doit: func [] [print "hello world"]

then the doit func available at /articles will yield "hello world" and not "this or that". However, the doit func at /articles/2007 will still yield "this or that" (This is because the location /articles/2007 has not picked up what is written inside the *.nonInheritable.class* at /articles)

.objects
Finally, we come to the objects that would be used as data for the ACMS system. Each "object" can be quite large ... depending on what is represented in that object. Loading a complete set of objects available at a node, including the entire contents of each object, can waste memory and time. Hence the actual contents are not stored anywhere on the web path, but are taken off that and stored in a central repository folder outside the web path. This also ensures that the actual contents are safe and non-hackable.

The *.objects* file at a particular node contains the array (block in Rebol terms) of the *shorter representation* of the final objects that ACMS handles at that particular node. So every node will have a small collection of the *shorter representation* of the final objects. Each of the short representation has a unique global ID that points to the *full representation* in the repository*.*

They distinguish between the "contents" and the "attributes" (i.e. meta-data) in XML and other meta data formats. But in ACMS, both the contents and meta-data are indistinguishable from each other, within the object in question. So, in ACMS there is nothing which is the "contents" of the object which different from the object's "attributes". EVERYTHING is meta-data (aka *attributes* aka *fields* aka *whatever*).

An object can describe (in as finely granular fashion as you want) anything: text, url, images, logic, whatever. The entire content is written automatically by ACMS in an outside repository location as a special ascii file which is readable by Rebol.

The .objects file only contains a "hash" block, where each hash index refers to the file where the actual data of the object is preserved. Each element in the hash block refers to only four important attributes of the object:

title,
author,
the *date of creation* and
date of modification.

As you can see above, the .objects file really does not contain the entire data. That is stored elsewhere, off the web path. That is why we said that the .objects file contains a list of **shorter* representations* of the objects.

The title can be used for so that the object can be quickly described in lists, menus etc. without first looking into the object's data file. The author name can be used to check permissions, and dates can be used for sorting as per dates, accessing journal entries, etc.

I personally see no particular advantage in MySql or any database system for storing the object's data. Moreover, as the objects at a particular node in the OOPs tree are only relevant for that particular point, they would always be considerably smaller than the entire set of objects handled by ACMS.

However, if someone does want the data of the objects to be stored in a database, instead of text; ACMS can be modified to easily handle that requirement. In fact, I can even see a transparent connection to a version control system such as subversion for the data repository too.

The meta-data of an object need not be defined in the object. But they can be given to the object in question via the ACMS inheritence mechanism.

A meta-data is nothing but the properties and methods accumulated either by direct instantiation within the object or by inherience, which describes the object in question.

Inheritence in ACMS
If you note carefully, inheritience used in ACMS is very similar to standard inheritence mechanism of OOPs but there are significant differences: Each point in the tree decides what the NEXT level of the tree can or cannot inherit. Each node can override what it had inherited from its parent "class".

But there are no "abstract" classes in ACMS. Like many simple OOPs systems, ACMS has only single inheritence. Like in standard OOPs languages, ACMS can have both methods (Rebol funcs) and properties (just plain data-types) as values for the class and object variables.

In ACMS one can define additional attributes (meta-data) even at the granular level of the object also; even if they (or their declarations) were absent in the class (Whereas in C++, Java, etc. all attributes must be declared in the class definition)

In fact, one of the desgin development decision that an ACMS developer will have to undertake is to decide which of the meta-data attributes should be instantianted only within the object and which should be inherited via the OOPs mechanism.

Objects Persistence
At each node in the OOPs tree (which as explained before is also the web-tree) a block of objects are written. The initial creation of the objects are NOT done in the *.objects* file. In fact the .objects file is never touched by hand. In order to maintain object persistence and consistency; there are two other files that are used by ACMS to do CRUD operations... well just the "C"(Create), and "D" (Delete) operations are handled. "R"(Read) operations are anyway done separately by ACMS and "U" (Update) is managed by a "D" operation followed by a "C" operation.

These two helper files are named thus, and are kept in the same folder as the .objects file they are supposed to effect. These two files are transient: Once their work is over, they are deleted automatically by ACMS.

.c.objects (for create)
.d.objects (for delete)

Every object is identified by a non-duplicated global Id. (Even after an object is deleted, the Id is never reused) Deleting objects are done by simply enumerating the id numbers of the object to be deleted by ACMS into the .d.objects file as a simple string of integers delimited by either a space or by comma. For e.g. The following numbers in .d.objects would mean that objects with Id 43535 and 5353535 will be deleted if they exist at that node.

43535, 5353535

By the way, the object ID numbers are unique universally across the ACMS site. So it is trivial to determine which node contains a particular object. So searching should be very fast, irrespective of the data structures that the ACMS programmer may have invented.

The .objects file will be automatically constructed and will contain references to the required "objects" after processing the .c.objects file. An ACMS object is usually created and added directly to the "objects" block by adding as many blocks as required into the *.c.objects* file. For e.g. these two blocks

```
[
  [title: "Hello" contents: "world"]
```

```
      [title: "Name" contents: "Sabu Francis"]
     ]
```

will eventually create two objects in the .objects file. Each of the above block is called an "object def block".

IMPORTANT: *It is EXTREMELY important that each object is put inside its own block, and the complete list of objects is itself inside one block as shown above. This is true even if there is only one new object that is being created.*

If the ID is also present, then that object def block is used to "edit" (instead of create) an object having the same ID.

Internally, the way ACMS handles creation of objects are as follows:

a) load .d.objects and keep one block ("delBlock") of Object IDs that need to be deleted
b) load .objects file and load it as one temp block ("tempBlock2") then foreach elem
   in tempBlock2 check if the element ID is a member of the delBlock. If so, then skip that
   elem, else append the element to the another temp block "objs"
c) erase tempBlock2
d) load .c.objects, create object from each object definition, serialize the object definition
   using "mold/all", save the serialized version into a data file in the repository, and add
   the element ID to "objs" along with its "title".

   Many of the object definitions in the .c.objects file will not have the object ID, in which
   case a unique object ID will be first assigned. But when one wishes to edit an object, then the
   ID of that object must be present in the object definition. In such a case, the object having
   the same ID in the "objs" be over-written with the new one. (So, that particular definition
   in the .c.objects becomes an "Update" rather than a "Create")
e) overwrite (or make afresh if not existing) the .objects file with the "objs" hash block. Note
   that the actual data of each object is stored separately in the data repository. What is present
   in the .objects file is only the object ID and a title (which is not expcted to be very big)
   Hence loading .objects file at any node of the OOPS tree should not take time.

The above 5 steps are handled automagically by ACMS. As a ACMS programmer, all you have to ensure is that the .c.objects and the .d.objects files are created as per your requirement, and then have some mechanism to reload the page. ACMS will recreate the correct .objects file automagically at that point and silently remove the .c.objects and .d.objects files from that node.

In fact, ACMS has a neat library function which can create the .c.objects file from a CGI block, *even though ACMS is unaware of the internal details of the object being created!* All the programmer has to do is to send the inputs from a form to ACMS, and ACMS will simply extract each value and create the required fields and write the *.c.objects* file! Next time the same node is read by ACMS, the *.objects will be re-created with the new objects.*

ACMS Core funcs
To create objects, we have a short-cut func called "appObj" to do the creation. That is internally used by ACMS to create an object.

When the ACMS developer writes the following object def in the .c.objects file

```
[
 [title: "My Beautiful Home Page"
  contents: "Hmmm I am a nice chap"]
]
```

... what ACMS actually does is to invoke the following func:

```
appObj [title: "My Beautiful Home Page"
    contents: "Hmmm I am a nice chap"
       ]
```

Note once again: In the .c.objects file, each object def is placed within square brackets [] and then the entire set is also placed within square brackets ... even if one object is being created.

"appObj" is called foreach element in the .c.objects file. The ACMS developer will never really call appObj. It is effectively a "private" function of the acms class (if you are looking at this from a C++ or Java point of view) and it is called internally by ACMS to pick up the object defs. Of course, the ACMS developer would have to initiate the process somewhere when performing his side of the story. (i.e. develop a CGI controller for his website that uses the ACMS system, writing Javascript files, etc.)

ACMS core library

appObj and *saveToRepository* funcs for ACMS are squirelled away along with other funcs, etc. as part of the core functions of ACMS. As stated in the introduction, an ACMS based website can be develped either in traditional CGI programming or using frameworks such as the Magic! system.

In case of the latter, these ACMS funcs will be used by the Magic! system. (So then ACMS becomes actually an extended version of Magic!) You can extend ACMS further by writing any number of further "layers" for specialist web-sites.

Rest of the docs assumes that ACMS is used as separate CGI apps, instead of being part of Magic! (In fact you can combine both approaches too on the same site. ... hmmm... should I have told you this earlier?)

Late binding

In ACMS, other than the "title", "auth" and "date" attributes of an object, everything is "late-bound" i.e. it is formed just as the object is queried. This ensures that ACMS deals with very little data at any given point in time. Any dynamic changes that are required would also happen gracefully. When a particular view is rendered in the M-V-C system of ACMS, actually only the objects that were referred in that view will be loaded into memory. If the view is satisfied just by the title and/or author's name and/or creation date of the object then the actual data file of the object will not even be queried.

All meta-data that gets passed down to a node in the OOPs tree using the "inheritence" mechanism are also late-bound. The meta-data for ALL the objects at a particular node is one and the same (to begin with). Hence, the loading of the inherited meta-data happens only once, and after that meta-data is loaded, each object will use that accumulated meta-data as its prototype. However, each object can have its own meta-data too; and so the ACMS developer can even over-ride the meta-data picked up from the class files, for some specific objects.

Future: A caching mechanism for inherited meta-data will be implemented.

Granularity of objects

An "object" can have as finely granular structure as you require. Which means the logical parts of the object can be described using different attributes (meta-data) of the object. For e.g. If you had to describe the article written by an author, then you may want to describe the object that represents an article as

```
[
author: "Sabu Francis"
title: "My fantastic article"
abstract: "Nothing but the truth"
contents: "Woweeee. There is nothing here. And that is the truth!"
keywords: "nothing, truth"
]
```

As stated before, the meta-data of an object could be instantiated in the object itself. However, some may be passed down to objects via the OOPS inheritence mechanism. The ACMS designer has to decide; as per the site being created, which of the meta-data attributes should be within the object itself (And therefore stored in the object's data file) and which should be put into the various class files on the OOPs tree. Some of those attributes could even be function definitions, for some complex logic handling of the object.

Objects at Nodes
At any given node, to prevent confusion, there should be objects only of one kind (i.e. one class). This is reflected in the facts that there can be only one file called ".inheritable.class" and one called ".nonInheritable.class" This is not a strict rule, but if you ignore it wildly then you would need to override lots of attributes even at the object level for some objects. That can lead to spaghetti coding.

The basic premise in an ACMS site is that each folder (aka node on the OOPs tree) logically contains the same kind of objects.

If you need to have another kind then you need to create another sub-folder (node in the OOPs tree). This can be useful in many situations. For e.g. If you are creating a medical journal publishing lots of articles, this folder path (OOPs branch) would be useful

/diseases/gastro-intestinal/children/appendicitis

* So at /diseases you can invent attributes that apply to "disease" in general
* at /diseases/gastro-intestinal you write down attributes for gastro-intestinal issues
* at /diseases/gastro-intestinal/children you can invent attributes that apply to "children"... attributes regarding "disease" and "disease/gastro-intestinal" will get naturally inherited from the parent node

and so on so forth.

Now there are a lot of OOPs design issues that need to be resolved beforehand. Some may not like that folder structure and instead use the following:

/diseases/children/gastro-intestinal/appendicitis

some may say this one is the correct:

/diseases/gastro-intestinal/appendicitis/children

etc. There is no single right answer. Such issues would be faced in any OOPs implementation.

Of course, if there are no attributes that need to be described specifically at a particular node, you can easily leave the .inheritable.class and the .nonInheritable.class files empty in that node. or those files may even be absent in those nodes.

Cloned objects
After reading the .objects file from a particular node in the OOPs tree, an ACMS developer can also add "objects" to the block by reading the .objects file from another node (depending of course on the particular need of the ACMS based site)

Those objects are called "clones". As the .objects file contain the reference to the data file in the repository where said object's definition exists, ACMS can read them in as quickly as the .objects file in the current node. However, the inherited meta-data of the object at any point is strictly as per the OOPS tree. That means, when you read an .objects file from another location (say 'b') other than where it was naturally located (say 'a'), it will load the inherited meta-data from 'b' instead of from 'a'.

If you want an object from another node on the OOPs tree to include the inherited meta-data of that location ('a') you would need to take the user to that point 'a' (i.e. users navigate to that location 'a' by clicking at an appropriate url referring to the originating node on the website)

This way of creating clones can be cleverly used by the ACMS developer to use the SAME object data for different kind of usages, without ever repeating the object data. For e.g. An ACMS website can have a url path for normal visitors (who don't get editing rights) and another url path for editors (who has editing rights) but both refer to the same data.

A regular use of this capability is to clone navigation objects such as menus and side-bars for use at various locations of a website.

What meta-data?
ACMS provides a very minimal set of core functions to ensure that the creation of objects at the right location happens in the correct manner using the OOPs inheritence mechanism. The nature of the meta-data (methods and properties of OOPs) in these objects can very well be the prerogative of the particular ACMS installation.

You can invent and put anything you like into the classes and the objects. One way to invent meta-data is to imagine the fields of the HTML form that would be used to create the object. For e.g. If your web site gets articles for a journal, then each article can have the following fields in the creation form:

Abstract:
Body:
References:
Keywords:

You now have reasons to "invent" four meta-data attributes (Fields) for your objects. Other than above, ACMS anyway stores the author's name, title and date. So you get those attributes too to "enrich" your object. Once the attributes are decided, the other the decision to be made is whether those attributes are to be placed at the object level or at the level of the class. That would be upto you. As in any OOPs system, if you use the classes intelligently; the work to be done at individual object level would be reduced.

Contents as meta-data
If you notice an "object" stores EVERYTHING (both meta-data and contents) as fields (or attributes). (This is not the approach taken by XML.) The advantage of storing contents as if it is part of the meta-data is that you could even "inherit" the object's contents, or provide default contents to objects. This is extremely useful for say things like the text for objects that represents Navigation elements such as menu items, etc. Repetition is completely avoided, and such text need not be even present in the .rhtml templates that would be used to create the website. The M-V-C system picks up such objects to do its Magic! inside the template as required.

Accessing various values of an object is simple

```
object/contents ;;; this will yield the contents of the object.
            ;;;If it is a function, then that function is executed to get the contents

object/title ;;;this yields its title
```

and so on and so forth

Precautions for meta-data
1. Make sure that all meta-data are assigned to variables. This is exactly as the XML grammar of having name=value in the attribute list of elements. In our case, "name" is the variable name. If you don't specify variable names and only specify the values that the variables must hold, there can be "side-effects" that Rebol may do on its own during the object's creation. (E.g. If the value was a function call)
2. Some of the meta-data would be read from visitor inputs and then stored into the various files (.inherited.class, .nonInherited.class, .objects, etc) Hence it is extremely important to "sanitize" them lest the user introduces some Rebol code into them. One way to do it is to use the Rebol construct function on the CGI variables
3. If a meta-data attribute is a function call then make sure that you don't actually call the function when writing the attribute. For e.g. if you want an object to have access to a function say "do_uppercaseTitle" written somewhere else then don't create a meta-data attribute like this

```
doit: do_uppercaseTitle self/title ;;; won't work!
```

instead the following is the correct way:

```
doit: func [] [do_uppercaseTitle self/title ]
```

When we now want the object to call do_uppercaseTitle , we'll be invoking it through the object thus:

```
theObject/doit
```

Assuming do_uppercaseTitle changes the case of a string to upper case, in the above example it will return an uppercase string of the object's own title. This is of course the power of encapsulation.

Revisiting objects
A traditional hand-written HTML files based website is tree shaped, with each node of the tree containing a sub-set of all the HTML files in the site. The web designer would determine which set of files goes into which node depending on some semantic pertaining to the site. Maybe all the articles written in a web magazine would be bunched up into a sub-folder in the web-tree called "articles" (what else can it be named?), all the files will be another folder called "files" (but of course) and so on and so forth.

ACMS closely follows this convention. Many CMS largely ignores the directories and the various internal URLs of a website are created on those CMS using redirection tricks of the .htaccess files (and other techniques) Not so in ACMS. In ACMS, if there is a URL called http://www.acms-example.com/here/there/everywhere you can be rest assured that there is a folder "branch" of the web tree as given in the path "/here/there/everywhere" So it is very intuitive. No tricks whatsoever. And moreover, you can even protect those folders using the usual .htaccess authentication methods without ever opening up your ACMS code.

Creating "Bread crumbs" is also a no-brainer because it is very easy to construct the bread-crumbs using the branch of the tree which the user is viewing. A kernel function in ACMS creates the string containing these bread-crumbs.

URLs and OOPs tree
As it may be evident now, there is a direct co-relation between URLs within the site and the OOPs tree that is used in ACMS. However, it is not always necessary that EVERY folder in the web-tree should also reflect as a usable URL. For e.g. you may want to create objects that describe navigation elements of the site in a folder called "navigation" thus

```
/user-interface/navigation
```

Now the ACMS developer may neither be interested in letting the user go to the url "/user-interface" nor "/user-interface/navigation" He may have simply created those folders only for the OOPs mechanism. In such a case, just put .htaccess files with visitor restrictions which will prevent web-visitors from accessing those folders. In fact, you can give partial restrictions too: Web-designers who are given responsibility to develop navigation objects may want to modify content through WebDAV in the files of those folders, and so they could be given access.

Rebol anyway picks up the files separately (not through Apache) so the objects and classes present in any folder on the web-site can still be used by ACMS.

So what is different?
ACMS separates the data quite clearly from the way the data is presented. Much more than what CSS would do. Then there is no repetition of logic. OOPs inheritence down the folder path can be very natural and one can let the inheritance feature take care of mundane things.

It is extremely easy to distribute work in an ACMS website. Encapsulation and separation of data from the view allows user-interface developers, content developers, backend system developers, all to work together without interfering with each other. The M-V-C system helps this even further.

M-V-C
ACMS is a Model-View-Controller based system. What we have described so far is essentially just the "model" part of it. If you notice, we have not really indicated how the data is actually going to be

displayed. Hmmm... for that matter, we have not described what is stored by the "model". You can invent any meta-data that can go into the "model". And we've told you this enough number of times.

Typically a team using ACMS would be "publishing" the meta-data attributes on the site that the team members are developing -- so that everyone knows what the semantic of the site is; including how inheritence happens along various branches of the site. Those handling the "visual aspects" (aka themes in other CMS) would only need to know some of the meta-data semantic that effects the look of the site. This will be done on a node to node basis, so even within the site they can distribute their work. They would not even get into the programming.

A team can easily be divided into three to reflect M-V-C: One portion handling the "controllers", one portion handling the "view" and the third handling the "model" The last (the "model" person) is very important: Data structures and meta-structures help decide what others can do and they need to be invented first before the ACMS based website can be used. The "view" person will need to learn the meta-data attributes of the objects and embed instructions for those into the HTML (or .rhtml if Magic! is used) templates.

As in any M-V-C system, the controller sits centrally to swing the data over from ACMS to the views and back from the views into ACMS. In CGI based websites, the controller is split into two parts: One part is one or more CGI applications written in Rebol and the second part is written as Javascript module or modules. That javascript module(s) are called at appropriate locations in the view. (Therfore the Javascript works at the client end and the CGI is at the server end) The two parts of the controller "shake hands" with each other using JSON formatted strings.

If the Magic! route is used then the the Views are created using the .rhtml templates of Magic! and the controller is directly written (or called from) within the .rhtml templates. Separate CGI programs need not be written.

Views
A view consists of one and only one html file. There can be multiple views at a node. Each view being handled by its own html file. This is analogous to the old system of having html files. The difference here is that the HTML files are actually empty! Each html file (for a particular view) contains all the good looking design that the designer wants but without ANY content. The designer will have to adhere a simple calling convention at the header of the html file, which can be easily learnt.

ACMS will then "pour" the data into various DIV (and other blocks defined by the designer) present inside the HTML file. The end user gets a pretty neat looking browsing experience. At each node in the web-tree, there will be one special javascript file called "view.js" which has additional functions that will swing the data values over into the DIV blocks of the HTML file. Of course, the "view.js" file would need to be written by a programmer who knows the jquery system. He will need to sit with the designer and both would need to agree on the naming convention of the DIV blocks. The person writing "view.js" need not know the looks of the design itself. It won't matter if there is a DIV block called #topmenu which is placed on the right-side of the browser instead of the top. "view.js" will still do its work.

Each view can pull in object information from the current node (i.e. the folder wherever the html file is residing) or any number of other nodes from the web-tree. The designer can control that by some instructions at the top of the HTML file.

So much for data to flow into the browser from ACMS.

Views would also be needed to handle data flows in the other direction: i.e. to send data from browser to be stored by ACMS. This is done by a small glue function in "view.js" and simply writing a regular HTML form. Nothing else is needed. ACMS will automatically create the required ACMS objects for each set of data sent by that form. The website architect can decide which kind of objects are needed, and the designer can make the form fields accordingly. Then he has to just place ONE (yes, ONE) function call in "view.js" so that the form data is sent back to ACMS and stored as an object. There are almost no restrictions on the form fields that can be used. Just some hidden fields are required per form so that ACMS knows what to do when it receives the form information. If the form is used to make a "login" then it is another function call.

Documentation to be continued... please give feedback to Sabu Francis: webmaster@sabufrancis.com