FreeAgent

For ColdFusion

By S. Isaac Dealey
info@autlabs.com

Contents

Acknowledgements

Great software is never the product of a single individual. Many thanks go to the creators and maintainers of the various open-source MVC frameworks for ColdFusion. Without them there would be far less collaboration in the ColdFusion community and there would be no cause for FreeAgent. It is their dedication that makes this possible.

What's a FreeAgent?

The FreeAgent project is a collection of components for ColdFusion that allows us to develop applications in such a way that they can be installed into virtually any of the MVC frameworks for ColdFusion (or deployed as a "standalone" application).

Let's start with an example. Joe is a ColdFusion developer and seeing how popular PHPBB is for PHP websites, he wants to create a similar open-source forum application for ColdFusion, that anyone with a ColdFusion server can run. (Or a Railo server or Open BlueDragon server.) So Joe starts about the task of outlining the application – looking at other forum applications, gathering requirements, making notes about important tasks the users might want to perform in a forum. In the process, the question arises, "should we use a framework?" There are lots of advantages to using MVC frameworks to develop web applications. They help organize our code, encourage separation between the model and the view, and many of them also include helpful widgets for performing common tasks like caching.

But with the use of a framework come some drawbacks as well. Which framework do you choose? Joe wants the new forum application to be something that anyone can use, so he doesn't want to make any decisions that would exclude large numbers of websites from running it. If he chooses no framework at all, that might result in a large user base for the forum, but there are liable to be a good number of companies where non-framework applications aren't allowed on their server. Other companies standardize on one framework or stack of frameworks. For example, one company may only work with CF on Wheels. Another company may only work with ColdSpring, ColdBox and Transfer, and still more companies may only work with Mach-II or Model-Glue. This means that if the new forum is developed in any single framework (or even no framework), it's liable to limit the appeal of the application and the user-base may remain small as a result. Even if Joe chooses a framework that's currently very popular in the community, there's still a possibility that single framework may later become less popular or potentially even become vaporware.

If Joe is very careful in managing the forum project, it's possible the team might be able to port it to several frameworks. Porting an application to different frameworks however requires a considerable investment of time, even if you're relatively familiar with each framework. And once the ports are done, then as the project manager, you've got to maintain several different parallel forks of the application, which could mean a notable "explosion" of code.

What if there's a way to have a *single* port of an application, and have it run in *all* the MVC frameworks for ColdFusion? What if this single port were no more difficult to develop than the same application written for a single framework? Joe's team could create the forum application with the widest possible appeal, making it useful for companies who standardized on any single framework, or even for companies where frameworks aren't normally used. And the team gets this broader appeal for free, with no extra work or extra maintenance required. Not only is this a win for Joe's forum team, broadening the appeal of the forum, it's also a win for all of the individual ColdFusion frameworks. It broadens the appeal of Mach-II, Model-Glue and Framework-One as well, since each of these frameworks now has a great forum application that people can install right away.

Enter FreeAgent for ColdFusion.

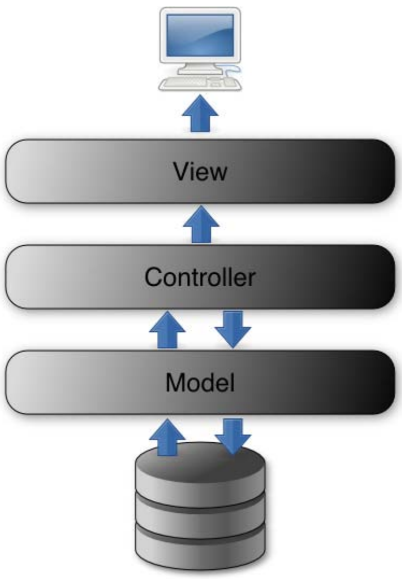
Is FreeAgent a Framework?

Actually, yes, FreeAgent is a framework. What's different about FreeAgent is the goal of the project. No other MVC framework for ColdFusion is designed specifically to provide portability to applications, allowing them to install into other MVC frameworks for ColdFusion. In truth, if we could have created FreeAgent without developing yet another MVC framework, we might have, but this turned out to be the simplest, easiest way to create the tool. What this framework does for us is eliminate the extra work (and code) that would normally be required to port an application to several different frameworks. This makes it just as easy to create a FreeAgent application as it is to create an application in any of the other frameworks.

Resources

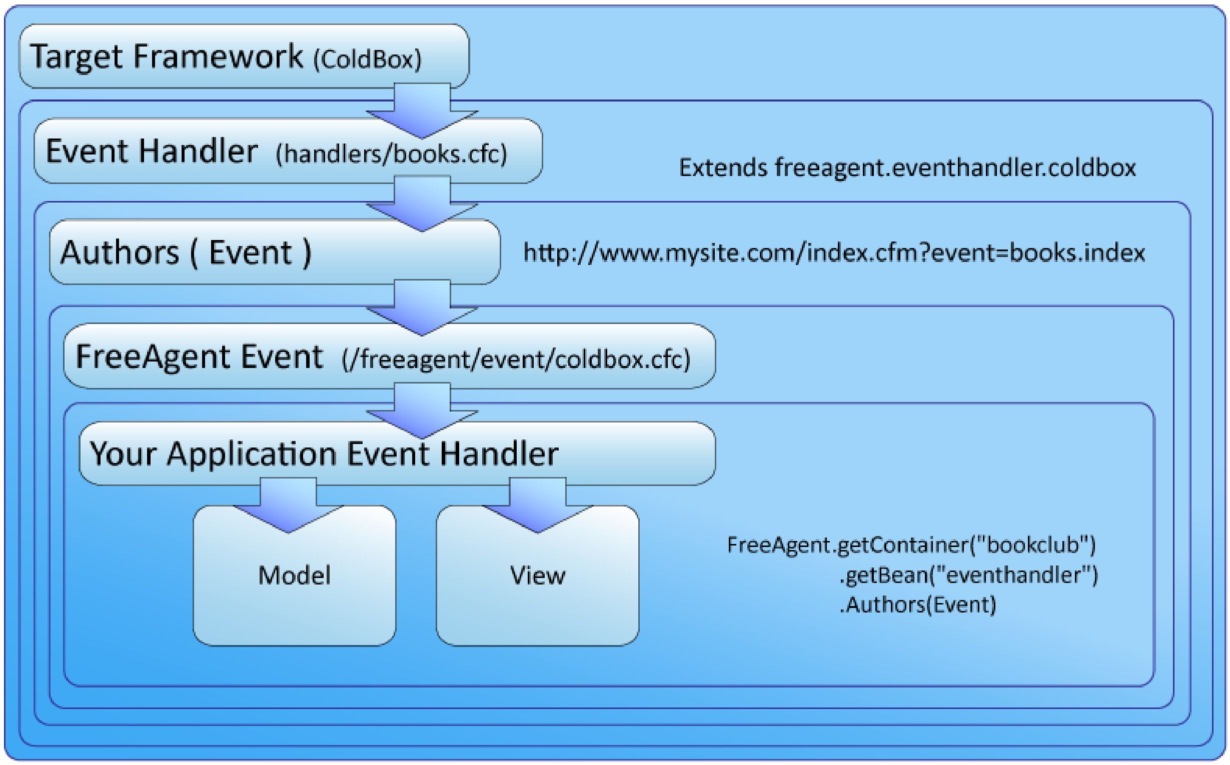
For additional support or to contribute to the FreeAgent project, join our discussion group at http://groups.google.com/group/free-agent.

How Does It Work?



Although the MVC frameworks for ColdFusion are sometimes rather complex internally, their theory is rather simple and much the same between frameworks. As a result we can hook into any of the existing MVC frameworks by taking advantage of those basic concepts. The most important concept is that all of these individual frameworks are "event-driven". This means that each framework has a method of modeling an "event", either as a structure variable or a component object, and an "event handler". The event-handler is a component object that acts as the controller, getting information from the application's domain-model objects and exposing them to the view later in the request. The implementation of the event and the event-handler (as well as the terminology) varies between frameworks, so with FreeAgent we simply provide a means of eliminating those variances between frameworks. This is done by creating an Event object which wraps the target framework's native event and use that to call the event-handler for our FreeAgent application. This way we can install the application into any given framework without modifying any of the application controller or view code, by using an adapter for the target framework's native event-handlers.

The diagram below shows the relationship between the target framework and your FreeAgent application. This diagram very closely resembles the flow of an application request if you're using any other framework. The diagram uses ColdBox as an example, so if you were using ColdBox, all the items in the diagram would be the same, except for the addition of the FreeAgent Event and Your Application Event Handler. When developing an application using ColdBox, the Event Handler at the top (handlers/books.cfc) *IS* your application event handler and connects directly to your model and view at the bottom. When developing the same application using FreeAgent, the ColdBox (or other framework) event handler becomes a simple adapter and your own application logic is moved down to the new event handler component at the bottom.


Target Framework (ColdBox)
Event Handler (handlers/books.cfc)
Extends freeagent.eventhandler.coldbox
Authors (Event)
http://www.mysite.com/index.cfm?event=books.index
FreeAgent Event (/freeagent/event/coldbox.cfc)
Your Application Event Handler
Model
View
FreeAgent.getContainer("bookclub")
.getBean("eventhandler")
.Authors(Event)

Don't panic. Although it may seem at first glance like this means a lot of additional work, it only seems that way. You don't actually need to write any code for the ColdBox (or other framework) event handler, you only need to write your own FreeAgent event handler CFC. The FreeAgent utilities handle all the middle parts of connecting the framework event handler to the FreeAgent event and executing your code. This means a small amount of configuration to install your application into each framework, but no real coding. The end result is that developing a FreeAgent application should be just as easy as developing an application for any individual framework, and should be particularly familiar for anyone currently working with ColdBox, FW/1 or Fusebox 5.

How small is that configuration? Here are examples for ColdBox, FW/1 and CF on Wheels frameworks. This code is taken from the book club sample application in the FreeAgent distribution.

ColdBox (handlers/books.cfc):

```
<cfcomponent extends="freeagent.eventhandler.coldbox">
     <cfset freeagent.container = "bookclub" />
</cfcomponent>
```

FW/1 (controllers/books.cfc):

```
<cfcomponent extends="freeagent.eventhandler.fw1">
     <cfset freeagent.container = "bookclub" />
</cfcomponent>
```

CF on Wheels (controllers/books.cfc):

```
<cfcomponent extends="freeagent.eventhandler.wheels">
     <cfset FreeAgentInit("bookclub") />
</cfcomponent>
```

As you can see, installing a FreeAgent application into any given framework is quite easy.

What Frameworks Are Supported?

Currently the following MVC frameworks are supported (alphabetical):

• CF on Wheels	http://www.cfwheels.org
• ColdBox	http://www.coldbox.org
• Fusebox 5	http://www.fusebox.org
• FW/1	http://fw1.riaforge.org
• Mach-II	http://www.mach-ii.com
• Model-Glue	http://www.model-glue.com
• onTap Framework	http://on.tapogee.com

If there is another MVC framework for ColdFusion you would like us to support, please join our discussion group at http://groups.google.com/group/free-agent.

Code Samples

After downloading FreeAgent, you will find a /sample directory in the distribution. In this directory are fully-functional copies of a sample application that uses the CFBOOKCLUB sample database distributed with the ColdFusion server. Code samples in this documentation will be generally derived from this sample application, which is found in the /sample/bookclub directory.

Getting Started

This section describes the essential requirements to get started building an application with FreeAgent. After reading this section you should know how to create a FreeAgent container, event-handler and how to use the LinkManager and AssetManager components to keep framework-specific content from leaking into your views.

Dependency Injection

Most new applications in today's ColdFusion community (at least those using frameworks), incorporate some form of dependency injection (DI) to reduce the amount of work needed to configure and load model components. In some cases this is ColdSpring or Lightwire – in other cases people develop their own dependency injection factories. This removes a lot of CreateObject() calls within the application and replaces them with Factory.getBean(). Although it may not seem at first like this is a big benefit, the abstraction helps to keep the application DRY, eliminating repetition of code and hiding implementation details like init-arguments and singleton status.

FreeAgent relies on the use of DI factories to organize code. Some other frameworks include DI integration like ColdBox and Model-Glue. These other frameworks generally assume that there is or will be a single DI factory for the entire ColdFusion application. Unlike these frameworks, FreeAgent assumes that there will be multiple factories. The reason for this assumption is simple: FreeAgent is about integrating applications together. With FreeAgent, the assumption is that every application is a sub-application within a larger site, intranet, extranet or suite of applications. For example you may have an extranet for your company which uses a particular framework like Mach-II or Mode-Glue and you may install several FreeAgent applications within that extranet, such as a discussion forum, a project management tool, and a job application tool for the HR department. It's likely that each of these FreeAgent applications is provided by a different group of ColdFusion developers. Some may be open source, others may be commercial. So assuming that these different applications would share a single DI factory would be at best problematic. What happens if two applications declare a bean with the same name? What happens if one project uses ColdSpring and the other project uses a custom DI factory? All of these potential problems are quickly and easily resolved simply by assuming that each application will have its own DI factory. This allows each project team to choose which DI factory they want to use without worrying about what any other project teams are doing.

Because each FreeAgent application will have its own DI factory, there must be a system for managing these factories. This is done by the FreeAgent CFC, which is a simple manager for DI factories. In order to manage the factories appropriately, they must be named and the potential differences between DI factories must be resolved to a single interface. To do this we've created "IoC Containers" which are found in the /freeagent/ioc directory. The container is responsible for ensuring that the DI factory has the same methods as any other factory, most importantly getBean() and containsBean(). (The term "bean" is an unfortunate legacy from java – it's meaningless, outside of being a pun on the name "Java", but has become the de facto standard for DI frameworks.) The IoC container is also responsible for knowing how to create the DI factory, but it doesn't actually create the factory immediately – it waits until the system requests a bean from that factory before loading it, to prevent unnecessary wait times during application start-up.

So to create your FreeAgent application, you'll need to configure the FreeAgent CFC with your DI factory. When the application loads, simply initialize a copy of the FreeAgent CFC and attach the containers for your FreeAgent applications to it.

Example:

```
<cfcomponent displayname=”application.cfc”>
<cffunction name=”onApplicationStart”>
     <cfscript>
     Var FreeAgent = CreateObject(“component”,”freeagent.freeagent”).init();
     FreeAgent.addContainer(“myapp”).init(“myapp.difactory”);
     </cfscript>
</cffunction>
</cfcomponent>
```

Remember that your FreeAgent container should have a descriptive name. This helps prevent the containers from colliding. Usually the name of your project, minus any spaces or punctuation, is a good way to name your container. So if we're creating a blog and our project is called "Shiny Blog", we should name the container "shinyblog". This container name should only be seen internally within the application – it should never leak out into URLs or other content available to your site visitors.

Good: xyzblog, xyzforum, xyzjobs

Bad: blog, forum, jobs

If you plan to use ColdSpring or LIghtwire, you'll find adapters for those frameworks in the IoC directory and can assign the container type using the 2nd argument of the addContainer method, for example, addContainer("myapp","freeagent.ioc.coldspringadapter"). This will create an IoC container specifically for a ColdSpring DI factory.

You'll also find several Application.cfc components in the /application directory of the FreeAgent distribution, which include some helper functions for loading FreeAgent. These application components aren't necessary, but are provided as an aid. Sometimes they're useful in integrating the FreeAgent initialization with the framework's built-in reloading system, which allows developers to reload the application with a URL parameter. The frameworks don't reload the FreeAgent containers without this extra integration because FreeAgent isn't part of the target framework.

IoC Manager Interface

These are public methods of the FreeAgent IoC Manager object. The methods you'll use most often (if at all) will be newContainer(), addContainer() and getContainer(). The remaining methods are useful for specific kinds of integration tasks, or tasks that can be automated by the framework.

getContainer(name): Returns a specific DI factory container by name.

listContainers(): Returns a comma delimited list of all loaded containers.

hasContainer(name): Indicates if a container has been loaded with a specific name.

hasVersion(name [, version, releasedate, buildnumber]): Checks to see if a container has been loaded with a specific name, then checks any provided release information. Returns true if the container indicates release information of equal or greater value. Returns false if the container isn't found or if release information indicates an earlier release or if the container doesn't have the requested release information.

addContainer(name, container): Attaches an already initialized DI factory container to the FreeAgent IoC manager. See also NewContainer() below.

detach(name): Removes a specified IoC container from the manager.

newContainer(name [, className]): Creates and attaches a new DI factory container and returns it for initialization. The container argument allows you to specify the path to an adapter which extends the default IoC container class, such as "freeagent.ioc.coldspringadapter" for ColdSpring or "freeagent.ioc.lightwireadapter" for Lightwire. See also addContainer() above.

findBean(beanName [,container]): Returns the name of a container which has a specified bean from a list of containers - ignores missing containers. If the list of containers is not supplied, all available containers are checked.

listBeanContainers(beanName): Returns a comma delimited list of the names of containers which have a bean matching a specified bean name. For example, listBeanContainers("TransferFactory") might return a list like "xyzblog,xyzforum" if both the blog and the forum application use the Transfer ORM and have a bean in their IoC factory named "TransferFactory".

resetContainers(): Resets the cache in all DI factories. This is useful if you need to restart or reload your application.

reset(): Drops all DI factory containers from the FreeAgent IoC manager. Does not perform any cleanup such as resetting cache for containers.

Writing an Event Handler

Writing a FreeAgent event handler is much like writing an event handler (or controller) for many other frameworks. There are two key differences from most frameworks. First, your event handler CFC will be created by your DI factory, not by the framework. Second, there are no mandated requirements for your event handler except that it must contain one public function for each event it will handle (it doesn't need to extend anything). Your application may have multiple event handlers (or multiple IoC containers), but the system assumes by convention that your application will default to a bean in your DI factory named "eventhandler". Because your event handler is managed by your own DI factory, you have complete control to provide it with init arguments or to load it with other objects from your DI factory. This also allows you to decide if the event handler will be a singleton or a transient object.

Event Methods

Each event in your application should be represented by a public function in one of your event handlers. This method receives one argument named "event", containing the FreeAgent event object. The event object contains a collection of parameters for the event (usually form and url parameters provided by the site visitor), as well as several methods for interacting with the view and/or the target framework.

Example:

```
<cffunction name="index" access="public" output="false">
     <cfargument name="event" type="any" required="true" />
     <cfset var rc = event.getCollection() />
     <cfset rc.qBook = instance.bookmanager.getAllBooks() />
</cffunction>
```

OnEventStart / OnEventEnd

Many other frameworks provide some method of executing code at the beginning and end of an event. ColdBox provides two methods called PreHandler/PostHandler and Fusebox provides PreFuseaction/PostFuseaction. FW/1 calls these same events Before and After and Model-Glue declares "before" and "after" nodes as part of an "event-type" in its XML config files. We've decided to keep this same feature in FreeAgent but have named the methods OnEventStart and OnEventEnd to preserve the naming conventions of the CFML language and Application.cfc.

Example:

```
<cffunction name="onEventStart" access="public">
     <cfargument name="event" type="any" required="true" />
     <cfset event.setValue("links",instance.linkmanager) />
     <cfset event.setValue("assets",instance.assetmanager) />
</cffunction>
```

OnMissingEvent

If an event handler doesn't have an event function with a specified name, FreeAgent throws an error indicating that the event could not be found. The exception to this rule is if you create a function in your event handler named "OnMissingEvent". If you declare this function then FreeAgent will call the onMissingEvent method, passing in the Event and EventName as arguments. This is an easy way to bypass controller logic for pure-content pages which may not need to execute any business-logic.

OnError

The FreeAgent system also allows you to declare some of your own error-handling for each event-handler by creating declaring an OnError method. Like the OnMissingEvent function, this method receives the Event and EventName, but also receives an Error argument, containing the trapped ColdFusion error structure.

The Event Object

The FreeAgent event object (/freeagent/event/genericevent.cfc) provides basic functionality related to event processing and may also improve integration with the target framework through subclassing. For example, the event object for the onTap framework (/freeagent/event/ontap.cfc) overrides the redirect() method to allow the framework to perform request cleanup before the browser is redirected to a new page. Here are the methods available through the Event object.

getCollection(): Returns the "request collection", a combination of form and URL parameters provided by the site visitor.

getValue(name): Returns a single parameter from the request collection.

setValue(name, value): Sets a parameter in the request collection.

valueExists(name): Indicates if a specific parameter is defined in the request collection.

redirect(URL): Sends the browser to a fully-qualified URL and may allow the framework to perform cleanup prior to the redirect.

setView(name): Sets the view template for the current event, relative to a directory for the FreeAgent IoC container holding the current event-handler. Exclude the .cfm extension from the name of the view template.

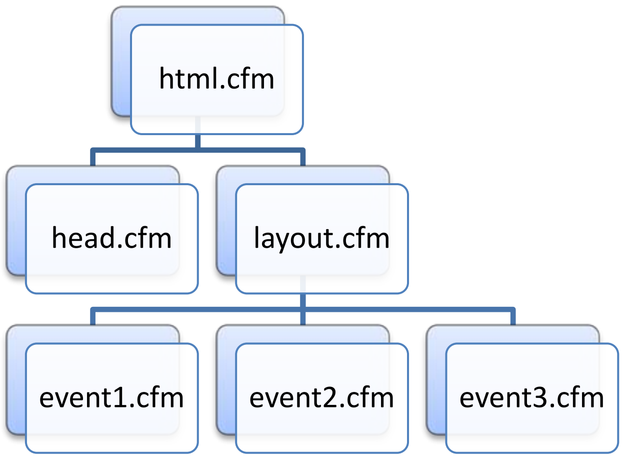
getView(): Returns the template name set for the current event - defaults to the name of the event.

isAjaxRequest(): Indicates if the current page request is a partial-page request from an AJAX call.

getFreeAgent(): Returns the FreeAgent IoC Manager for access to DI factories from other FreeAgent applications.

invokeFreeAgentEvent(EventName): Executes the event-handler method for a specified event. The name of the FreeAgent IoC container and event-handler bean are declared in the Event init method. In general this method will not be used in your code.

Rendering the View



The goal with a FreeAgent application is to allow installation into any framework. To meet this goal, the view must be devoid of any framework-specific methods calls that aren't part of the FreeAgent tools. Mostly this means abstracting the creation of links in the HTML using the FreeAgent LinkManager.cfc, however, there are a few other specific caveats.

Page Structure

As with many things, MVC frameworks for ColdFusion vary in their implementation of the view. Some provide "layouts", others don't, and still others require you to use layouts. The goal in defining a FreeAgent view structure has really been to avoid these kinds of systems as much as possible. For that reason, the skeleton applications included with the FreeAgent distribution divide the html view as seen in the diagram above.

Content Variables: Unlike some other frameworks, the current version of FreeAgent provides no method of placing rendered views into "content variables" or of caching the rendered content. So instead of executing a separate event and storing the output in a content variable which is then displayed in the layout, a FreeAgent application must use simple cfinclude tags instead.

HTML Head: It is important to separate any HTML head content for your application into its own template (head.cfm) for the inclusion of style sheet declarations and JavaScript libraries. The event object may or may not exist when the head template is executed, so the AssetManager used to generate these external file links should be collected directly from the FreeAgent container, using the variable Application.FreeAgent and then fetching your application container with the getContainer() method.

Example:

```
<cfset FA = Application.FreeAgent />
<cfset App = FA.getContainer(“myapp”) />
<cfset assets = App.getBean(“AssetManager”) />

<cfoutput>
     #style(“mystylesheet.css”)#
</cfoutput>
```

Layout.cfm: This template resembles layout templates for some other frameworks, allowing you to place headers, footers and sidebars around your entire sub-application. Content is placed within the layout using a simple cfinclude and the Event.getView() method.

AJAX Requests

If you're using jQuery to request AJAX partial pages, the framework will automatically detect the custom http-header provided by jQuery. Otherwise, you can use a URL or form parameter of "ajax=1" to tell the FreeAgent event that it is a partial page request. This is handled in the HTML template using the Event.isAjaxRequest() method, bypassing rendering of the HTML wrapper, head.cfm and the layout for the application.

LinkManager

The LinkManager object (/freeagent/linkmanager.cfc) is a tool for building links primarily to pages within your application. This is similar to URL-building utilities provided by other frameworks, such as the Mach-II BuildURL() function. These utilities were added to frameworks like Mach-II to abstract links in the application, for reasons similar to those of FreeAgent. Most of these frameworks allow you to change the name of the Event variable (some people like "event", others prefer "action", some prefer the compact "a" and the default for FuseBox used to be "fuseaction"). So it's helpful to have a simple function that encapsulates the creation of a link, so you don't have to worry about whether the event variable is named "action" or "fuseaction" or "a". These link-building methods also make it easier to convert to SES links using a routing tool like ColdCourse.

The FreeAgent LinkManager object takes link building one step further. In addition to allowing link-formatting to belong uniquely to the sub-application in question (your FreeAgent application), the LinkManager also provides a convenient way to add functionality within your application, and stub any complex links that might be used more than once within your interface to eliminate repetition. For example, if you have a URL with a lot of parameters, like "index.cfm?event=foo.bar&a=1&b=2&c=3" you might not want to have to type out all those URL parameters throughout your application. So with the LinkManager you can simply pre-define that URL so you don't have to declare the parameters, but you can append to or overwrite them as necessary.

Example:

```
<cfcomponent extends="freeagent.linkmanager">
     <cfset define(name="foo.bar",param="a=1&b=2&c=3") />
</cfcomponent>

<cfoutput>
<!--- display the URL with default parameters --->
<div>#linkManager.getLink("foo.bar")#</div>

<!--- add one parameter and overwrite one parameter --->
<div>
#linkManager.getLink(name="foo.bar",param="x=y&c=z")#
</div>
</cfoutput>
```

Parameters for links can always be passed to LinkManager methods either as an &-delimited string or as a structure of key-value pairs.

init(siteURL, linkPattern, eventVar, eventHandler, param): All of the values defined when initializing a LinkManager object are default values for the links it manages. All arguments are optional.

define(name [, eventName, param, eventVar, eventHandler, linkPattern, siteURL]): In most cases you'll only provide the first three arguments to this method. The remaining arguments are usually supplied by the defaults set in the init() method. Not all links must be defined. If a link is undefined, the LinkManager assumes that the provided link name is the event-name for the link, so the majority of links can remain undefined. This does however provide a convenient way to customize an application, by subclassing the link manager and using the subclass to create aliases for any links that need special handling.

getLink(name [, param]): Returns a complete URL string (not an HTML link tag). Any parameters defined for the link will be appended to or will overwrite default parameters defined for the link or for the LinkManager as a whole.

link(name, label [, param]): Returns an HTML link tag. The label argument defines the HTML to place within the link; usually text or an image tag. Additional arguments can be supplied to define additional attributes for the link tag. For example if you supply a "class" argument, the link tag will is created with a corresponding "class" attribute.

AssetManager

The AssetManager object (/freeagent/assetmanager.cfc) is similar to the LinkManager object, designed to abstract paths to external files that might be included in an HTML page, such as style sheets, JavaScript libraries, images and Flash movies.

More to come …

Menus and Navigation

The Menu object (/freeagent/menu.cfc) is a helper class designed to eliminate repetitive coding in site navigation. A helpful side-effect of this component is that developers who install your application into their own frameworks have a convenient method of customizing your application, by extending your menu objects.

More to come …

Security Façade

The SecurityFacade object (/freeagent/securityfacade.cfc) is a helper class designed to simplify the task of connecting various FreeAgent applications to a security layer of unknown origin. In other words, the application should be able to connect to and interact with your site's security layer, without knowing what your security layer is, who developed it or how it works. This will allow all of the FreeAgent applications installed in your site to use your security layer, regardless of who developed the FreeAgent application (instead of having each FreeAgent application contain its own separate security layer, which could be a real management nightmare).

More to come …

Internationalization

Translations

Time Zones

Names