HomePortals Modules Framework

By Oscar Arevalo
oarevalo@cfempire.com

Created: July 20th 2006
Last Update: August 7, 2006

Introduction

In HomePortals, content is described using HomePortals pages, which are XML documents that follow a given specification. This xml document tells HomePortals how to layout the page, what external resources to use and what content to render on the page. This content is defined via Modules. Modules are declared in the <modules> section of the xml document using <module> tags. The <module> tag takes several arguments that give information to HomePortals about how to locate the module, where and how it should be rendered, and more.

Modules are self contained elements that can be reused and placed anywhere on a page. These elements can range from simple html content to complex mini-applications with several views and actions. This document describes the framework provided by HomePortals for building these modules. The module framework allows modules to have a certain degree of pre-built functionality that facilitate the development of rich and complex modules.

The same module may appear several times on the same HomePortals page, having as only requirement that it ID attribute is not used again by any other module on the page. HomePortals uses this ID or ModuleID to uniquely identify each module. Each occurrence of a module is called a “module instance”, and all instances of the same module are said to belong to the same “module class”. When a page is built using the integrated Control Panel, then the HomePortals engine will be responsible of assigning the module ID and making sure they are unique.

All HomePortals Modules are defined as ColdFusion Components (CFCs) that extend Home.Components.baseModule. **baseModule** provides the basic interface for each module, being responsibility of each module to define views and implement the specific methods to provide its unique functionality and/or visual output.

Each HomePortals module is actually a small MVC-like application on its own, with the module CFC (along with the Home.Components.moduleController cfc) acting as the application controller. Each module may have one or more views associated with it, however only one view may be displayed on the module output area at any given time. HomePortals module framework provides access to a storage file personalized for each account which the module can use to store any necessary data. Nevertheless, developers are free to deal with data any way they choose within the module component.

HomePortals Module Lifecycle

When HomePortals is generating a page a sequence of events occurs in order to initialize and render a module. The following events happen for each module on the page:

The short version:

1. HomePortals creates a controller for the module cfc, which in turn instantiates the module cfc.
2. The module controller calls the method **`init()`** on the module cfc
3. HomePortals asks the module controller to render the module's "HTMLHead" view (if defined). The output of this view will be included within the Head section of the generated HTML document.
4. HomePortals asks the module controller to render the module's "default" view to display on the screen.

The long version:

1. HomePortals creates an instance of Home.Components.moduleController and calls its `Init()` method.
2. The moduleController in turn creates an instance of the module cfc
3. moduleController checks if configBeans (moduleConfigBean and contentStoreConfigBean) exist for this module on the current session, if so, then loads the configBeans from the persistent storage and assigns them to the module instance.
   If no configBeans exist for this module and session, then the moduleController creates default instances of the configuration beans and assigns them to the module.
4. moduleController passes a reference to itself to the new module instance (this.controller)
5. moduleController calls the method `Init()` on the module cfc.
6. moduleController stores the configBeans for this module into the permanent storage (either client or session scopes)
7. HomePortals asks the module controller to render the module's "HTMLHead" view (if defined). The output of this view will be included within the Head section of the generated HTML document.
8. HomePortals asks the module controller to render the module's "default" view to display on the screen.

From the module lifecycle described above, you could see that each module is expected to have an `Init()` method. Since every module cfc is required to extend Home.Components.baseModule a default implementation is always inherited; however, this baseModule only contains an empty stub for this method. You should overload the Init method to provide any initialization code your module may require and more importantly, to set the values of the configBeans in order to customize your module.

The following chapters will elaborate more on the configBeans and the baseModule.

Module Physical Structure

This section contains the requirements as well as suggested guidelines on how to create a module.

HomePortals does not require each module to have its own directory; however it is a very good practice to have a directory for each module or group of closely related modules. For example, when building an storefront, you may want to have separate modules for shopping cart, catalog, item search, etc; but they all may share certain resources, such as images, data-access components or web services, so it makes sense to have all these modules and resources bundled together.

Having said that, let us take a look at a sample directory structure for a module:

```
\Home
      \Modules
            \myFirstModule
                  myFirstModule.cfc
                  \views
                        vwMain.cfm
                        vwConfig.cfm
                  \layouts
                        lytMain.cfm
```

The previous directory structure assumes a default installation of HomePortals in the /Home directory. Also, by default, modules are stored within the HomePortals directory, but you may have a module stored anywhere on the server.

All module-related files are stored on the myFirstModule directory. The actual module is implemented by the myFirstModule.cfc file, which is a ColdFusion component that extends Home.Components.baseModule

The views directory is used to store the html blocks that will be rendered as the module visual content. This content is rendered when the page is created and/or requested directly from the browser by the module client. As the name implies, these html blocks are different than regular html pages, since they are not required to have tags like <html>, <body>, and <head>, since they will be actually inserted within the existing page.

The layouts directory is used to store layouts. Layout blocks are containers for views, they are used to display content that remain static while the view changes. The use of layouts is optional.

Besides these files and directories, module developers are free to add any additional resources as needed.

The Module Controller

As explained on the previous section, all modules are composed of a ColdFusion component and a set of views. However, the cfc by itself does not know very much of HomePortals and its environment. This is the role of the module controller.

The Module Controller (Home.Components.moduleController) is the component responsible for managing and interfacing with the actual module. Each HomePortals module has a corresponding module controller, which the module uses to get access to the HomePortals environment and services.

The way a module accesses its controller is by using the `this.controller` reference. This variable returns a handle to the controller for the current module. The controller has several functions available to the module to perform several tasks.

Refer to the component reference for a detailed list of available functions and their descriptions.

Defining a Module: the ConfigBeans

All HomePortals modules when created are basically copies of a default module, which requires them to be customized and configured in order to have an identity of its own.

The most important way a module is defined, is by its configBeans. These are special objects that are only used to hold settings or properties that characterize or define specific aspects of a module. The available properties of the configBeans determine the ways in which a module can be configured. HomePortals modules have two configBeans: the moduleConfigBean and the contentStoreConfigBean.

The moduleConfigBean is used to store the general settings for the module. It contains several methods to access the different properties that HomePortals requires for each module. Additionally, it may hold additional properties that are individual to each module. All modules should configure their own moduleConfigBean within their init method.

The contentStoreConfigBean is used to store the settings for the module's Content Store, which is a way of providing persistent storage for each module. This storage may be individual to each account, or may also be a fixed storage for anyone that uses the module. The Content Store is an optional service provided by HomePortals to its modules, so it is not required for a module to set the properties of the contentStoreConfigBean.

The configBeans for a module are always available regardless of the context in which a module is executed. They have semi-persistent storage, meaning that modules will have access to them as long as the user remains on the same page. This is important because HomePortals allows module code to be executed under two different contexts: first, during page generation time, and later when views or actions are called via remote requests from the browser window. When the user goes to a different page, then the configBeans will be dissolved and replaced with the configBeans for the new page. Note that configBeans are stored per user.

Both configBeans are accessed through the moduleController via the following functions:

- `this.controller.getModuleConfigBean()`
- `this.controller.getContentStoreConfigBean()`

Once you get a reference for each of the configBeans, you can start to use their own property access functions (setters/getters) to configure them. Any changes to a configBean are stored after the current module function finishes. This means that if you have an unhandled error after you set a configBean value, then it will not be saved and that change will be lost.

For the moduleConfig there are some values that will be set automatically by the moduleController, but there a few that must be set manually within the `Init()` function. The following table describes these properties:

Setting Name	Operation	Description
Module class name	setModuleClassName("…")	An identifier for the module class.
Default view	setView("default", "…")	Name of the initial view to display when rendering the page. Views are identified by the file name without the .cfm extension. All views must be located within the /views directory inside the directory where the module is located. This view is also displayed whenever a call is made to the render() function without explicitly passing a view name.
Default layout	setDefaultLayout("…")	Name of the default layout to use when rendering views. If this is empty, then no layout is used.
HTML Head view	setView("htmlHead", "…")	When set, this setting indicates the name of a view that will be rendered within the HTML Head section of the rendered page. Use this view to define script or style blocks that should only be rendered once regardless of the number of occurrences of the same module in a page.
Module Root	setModuleRoot("…")	This setting is required in order for the module to know its own location in the server, which is needed in order to access external files such as images. Since modules execute within the HomePortals engine, using paths relative to the location of the CFC will not work consistently for locating images or other resources.

Page Settings and Module Properties

The HomePortals module framework allows modules to be configured at two different levels. A module can have settings that affect the entire module class, which is all instances of any given module; and at the same time have settings that apply to only an individual instance of the module. For example, a module that checks an email inbox for new messages may have as a module class setting the URL of the mail server to check, and as module instance setting the specific username and password for which to check.

Module class settings, or "Module Properties", are stored on the moduleConfigBean and accessed via the `getProperty()` / `setProperty()` functions. These are name-value pairs that may be set within the `init()` method. The use of property functions is to allow the flexibility for the developer to implement the dynamic loading and changing of these values; for example, by having an external configuration file.

Module instance settings, or "Page Settings", are defined on the HomePortals page in which the module is contained. These settings are the attributes of the <module> tag in the HomePortals page. This means that all page settings are structured as name-value pairs. Page settings are automatically initialized by the moduleController when the module is created. To access these values at any time, use the functions `getPageSetting()` and `setPageSetting()`. Please note that these methods are only used to manipulate the values on the moduleConfigBean object; to make changes permanent on the actual HomePortals page, you must use the `savePageSettings()` method from the moduleController.

All these settings management functions allow you to create special “settings” views for your modules if needed.

ContentStore

The HomePortals module framework provides by default a mechanism to store data in the form of XML documents. This data storage functionality may be configured to use a fixed file per module class, or to use file that are individual to each HomePortals account. This functionality is provided through the contentStore component (Home.Components.contentStore)

To enable the contentStore on a module you need to setup the contentStoreConfigBean properties. The following table describes the typical properties you will need to set in order to configure the contentStore for your module.

Property	Operation	Description
url	setURL("…")	Path of the location of the xml document. If the file does not exist and the createStorage flag is on, then it will be created. If this is empty, then a file name will be generated using the defaultName argument, and will be stored on the Account directory for the current page. By default this is always empty.
rootNode	setRootNode("…")	Name of the root node of the xml file. The default is "dataStore".
defaultName	setDefaultName("..")	Default name for the content store file when no URL is explicitly given. The default for this property is "myContentStore.xml"
Description	setDescription("…")	Small description of this content store.
accountsRoot	setAccountsRoot("…")	Relative path to the location where HomePortals accounts are stored. Default is "/Accounts".
createStorage	setCreateStorage(…)	Flag to indicate if a contentStore file should be created in case the requested (or default) file does not exist.

These values may be set at any given time, and not necessarily on the module's `Init()` method; they can even be set dynamically at run-time. The contentStoreConfigBean will not be processed until you request the content store document by using the controller's `getContentStore()` method.

To provide certain degree of freedom to developers as to how the data is stored on the content store, the contentStore component only gives access to the raw xml object, and only provides the functionality for its retrieval and storage. The actual xml manipulation should be handled by each module.

To work with the contentStore document, use the `getContentStore()` method on the module controller. This method returns a reference to the contentStore for the current module already populated with the data. After you have a reference to the object, you may use the methods `getXMLData()` and `save()` to retrieve and save any changes. Both are methods of contentStore and use xml objects as return type and argument, respectively.

The following code snippet illustrates how to make a change to a content store:

```
<cfscript>
      myContentStore = this.controller.getContentStore();
      xmlDoc = myContentStore.getXMLData();
      xmlDoc.xmlRoot.xmlAttributes[“lastUpdate”] = lsDateFormat(Now());
      myContentStore.save(xmlDoc);
</cfscript>
```

Remote Server Requests

One of the defining characteristics of HomePortals is its support for remote server requests; these are requests made by the browser to execute server-side code without having to reload the current page. This behavior results in a radically different user experience when interacting with a web page, improving response time and giving an overall richer experience.

HomePortals module framework makes it easy for module developers to add this type of functionality to their modules without much effort. Each module in a HomePortals page has a corresponding JavaScript object that has several methods to "talk" to the server-side module code. This object is actually an instance of the moduleClient class (defined on */Home/Common/Javascript/moduleClient.js*) These methods allow the JavaScript client to call methods defined on the module cfc to perform operations and retrieve views to display on the module content area.

The name of the moduleClient instance is the current module ID. On the module cfc, this value is obtained by the function this.controller.getModuleID()`.`

The basic functions used for this type of functionality are: `getView()`, `getPopupView()`, `doAction()`, and `doFormAction()`

For example, to retrieve the view "main" on the blog module, you may use the following JavaScript code:

```
blog1.getView('main')
```

Where blog1 is the module ID.

The following table describes each of these functions in detail:

Method	**Description**
getView (view, target, [args])	Retrieves a server-side view and displays it on the element with *target* as its id attribute. The argument *args* is an optional structure that can be used to pass more arguments to the module. You may use this structure to pass any module-specific arguments that your view may require; However there are two special purpose arguments that will be used by the module: **layout** and **useLayout**. The first one is used to request that the view is rendered using a particular layout, and the second one is to force/avoid the use of layouts. This last argument is useful when a module has a default layout but you do not want to render this layout to be rendered on the current request.
getPopupView (view, [args])	Similar to getView, except that the retrieved view is displayed on a css pop-up window on the middle of the screen. The actual appearance of this window may be changed by overriding the default css classes used

	to style the window. Use closeWindow() to close this pop-up window.
doAction (action, [args])	This function is used to call methods on the module cfc. *Action* is used to identify the method to call. This method does not need to have "remote" access since it will be called using the moduleController object, which acts as a proxy. a*rgs* is an optional structure used to pass arguments to the method call.
doFormAction (action, frm)	This function is similar to doAction, except that instead of passing a structure with elements to pass as method arguments; the function takes a reference to a form, and all fields in this form will be passed to the module method.
closeWindow ()	This function closes the popup window opened with getPopupView()

Components Reference

Home.Components.BaseModule

Method	Return Type	Description
Init	Void	Overload this method to add initialization code for a module.
renderInclude	string	Returns the output of an included file. The included file is executed under the context of the current module. *Do not overload this method.*

Home.Components.moduleController

Method	Return Type	Description
Init (moduleID, [moduleClassLocation, modulePageSettings, pageHREF, isFirstInClass])	baseModule	Module constructor. Responsible for initializing internal variables as well as the module to be controlled. This method is always called internally by HomePortals.
getContentStore	contentStore	Returns a ContentStore object initialized for the current module.
getContentStoreConfigBean	contentStoreConfigBean	Returns the configBean for the module's contentStore
getModuleConfigBean	moduleConfigBean	Returns the configBean for the current module.
getModuleID	String	Returns the "moduleID" property of the current module. This name is used to identify the JavaScript module client.
getUserInfo	Struct	Returns a structure with information about the current user and the owner of the current page.
isFirstInClass	Boolean	Returns a flag informing whether this module instance is the first occurrence of this module class on the current page.
setEventToRaise (object, event, args)	Void	Adds a framework event to be raised on the client. A module may set any number of events to be raised, which will be done as soon as the current request finishes.
flushEventsToRaise	Void	Flushes the stack of events to raise.
setMessage (message)	Void	Sets a message to display on the module client.
getMessage	String	Returns the current message set to display.
flushMessage	Void	Clears the message.
setScript (script)	Void	Sets a JavaScript code block to be executed on the browser as soon as the current request finishes.
getScript	String	Returns the JavaScript code block.
flushScript	Void	Clears the JavaScript code block.
setErrorInfo (errorInfo)	Void	Sets a structure that will hold exception information. You may use the cfcatch for this structure. This is used along with the "error" view for customized error handling.

getErrorInfo	Struct	Returns the saved error information
flushErrorInfo	Void	Clears the saved error information.
savePageSettings		Saves page-level settings for this module. The current pageSettings on the moduleConfigBean are saved to the current HomePortals page.
Execute (action)	Void	Executes a method on the module. The method to execute is defined with the *action* argument. Any arguments to this method are passed along to the method called except the *action* argument.
render (view, layout, useLayout)	String	Renders the default view to display on the module content area. If layout is not empty, then the view is rendered using the given layout, if empty, then the default layout is used, unless there is no default layout or the useLayout flag is false.
renderHTMLHead	String	Returns the contents of the view selected as the HTML Head for this module. If no HTMLHead view is defined, then returns an empty string. The output of this module will be included on the <head> section of the resulting HTML document.
renderClientInit	String	Returns the JavaScript code for the initialization of the moduleClient javascript object.
renderMessage	String	Returns the JavaScript code to display a message on the module client.
renderRaiseEvents	String	Returns the JavaScript code to raise framework events on the module client.
renderScript	String	Returns any JavaScript code set to execute on the browser
renderError	string	Returns the content of the view defined for displaying errors. If not error view defined, then displays errors in a default format

Home.Components.contentStore

Method	Return Type	Description
Init (contentStoreConfigBean)	contentStore	Module constructor. Responsible for initializing internal variables, creating the storage document (if needed) and reading it.
Save(xmlDoc)	Void	Sets the given xml document as the content storage document.
getURL	String	Returns the location (as relative path) of the content store document.
getXMLData	XML	Returns the contents of the content store document as an xml object
getOwner	String	Returns the username of the account which created the content store document.
getCreateDate	String	Returns a string with the date in which the content store document was created.

Home.Components.moduleConfigBean

Method	Return Type	Description
getDefaultLayout	String	Returns the value of the DefaultLayout property.
getModuleClassLocation	String	Returns the value of the ModuleClassLocation property. This is the path to the module cfc relative to the website's root.
getModuleClassName	String	Returns the value of the ModuleClassName property. This is a name used to identify the current module class.
getModuleRoot	String	Returns the value of the ModuleRoot property. This is the path to directory where the module is located. This is useful when referencing images or similar resources.
getPageHREF	String	Returns the value of the PageHREF property. This is the location of the HomePortals page that contains the current module.
getPageSettings	Struct	Returns a structure of name-value pairs containing all module settings defined on the current HomePortals page.
getPageSetting(key, [default])	String	Returns the value of the requested page setting. If the setting (identified by *key*) does not exist, then returns the value of *default*.
getView(key)	String	Returns the name of the requested view type. The view type is identified by *key*. View types may be "default", "htmlHead" and "error".
getProperty(key, [default])	String	Returns the value of the requested module property. Properties are name-value pairs defined by each module.
setDefaultLayout(data)	Void	Sets the value of "DefaultLayout".
setModuleClassLocation(data)	Void	Sets the value of "ModuleClassLocation". Called internally by the module controller.
setModuleClassName(data)	Void	Sets the value of "ModuleClassName"
setModuleRoot(data)	Void	Sets the value of "ModuleRoot"
setPageHREF(data)	Void	Sets the value of "PageHREF". Called internally by the module controller.
setPageSettings(data)	Void	Sets the value of "PageSettings". Called internally by the module controller.
setPageSetting(key, data)	Void	Sets the value of a given page setting.
setView(key, data)	Void	Sets the name of a given view type (*key*). View types may be "default", "htmlHead" and "error".
setProperty(key, data)	Void	Sets the value of a given property.

Home.Components.contentStoreConfigBean

Method	Return Type	Description
getAccountsRoot	String	Returns the value of the AccountsRoot property. This is location where all account files are stored.
getDefaultName	String	Returns the value of the "DefaultName" property.

		This is the default name used for the content store document when an explicit location has not been given.
getDescription	String	Returns the value of the “Description” property. This is a brief description of the content store document.
getRootNode	String	Returns the value of the “RootNode” property. This is the name of the node to use as root node on the content store document.
getURL	String	Returns the value of the “URL” property. This is the location of the content store document.
getCreateStorage	String	Returns the value of the “CreateStorage” property. This is true/false value that determines whether a storage document should be created when no URL is given. The document is created on the current page’s owner account directory, and using the “DefaultName” property.
setAccountsRoot(data)	Void	Sets the value of “AccountsRoot”. Default is “/Accounts”.
setDefaultName(data)	Void	Sets the value of “DefaultName”. Default is “myContentStore.xml”.
setDescription(data)	Void	Sets the value of “Description”.
setRootNode(data)	Void	Sets the value of “RootNode”. Default is “dataStore”
setURL(data)	Void	Sets the value of “URL”.
setCreateStorage(data)	Void	Sets the value of “CreateStorage”. Default is “true”.