AModules3 Introduction [rev3]

◆

About

.. this document

I'm writing this document for PHP developers who are willing to develop using AModules3 libraries. There will be no detailed explanations about every function, but this document will help you to get started. This document consists of 3 parts: "overview", "for application developers" and "for library developers". Do not skip any part and read until you feel skilled enough. For further reading use class documentation and source code.

.. AModules3

AModules3 is a stable development framework written in PHP5. It comes with it's own concept which may change the way you will write your next PHP application. Main points of AModules3 are:

- On-Demand: you can use it for CLI application and it will not include any unnecessary code.
- Extensibility and flexibility: you can use it for CLI application, simple administration system, complex billing server, web community portal or simple website.
- No compromises: whatever you develop, you wouldn't waste productivity, your own time rewriting some essential things and you wouldn't loose any functionality.
- Complete: AModules3 have its own database abstraction layer, template system and many other things you will find essential in everyday use.

Apart from being just a library, AModules3 allows you to use it as a base for your own API framework. That means you may build your own framework on top of AModules3 and re-use it inside all the application you write.

.. me

6 years ago I have finished work on the first version of a-modules. Back then it was compatible with PHP3 and wasn't anything elegant. Over 3 years ago AModules2 were born. Each of those version had at least 20 web production applications built on top of them. Year ago I have started AModules3 which was was a total rewrite and brought many new concepts. At the same time it carried all the experience from it's predecessors to avoid possible mistakes.

Contents

Overview

This chapter guides through the basic concepts of AModules3 framework. If you are going to somehow work with AModules3, you should be familiar with it's concepts.

Class structure

Assuming that you already know what Class is and what is inheritance, it's rather straight-forward. Classes of AModules3 are located inside "lib" directory. Each file is name same as the class name. If class name contains underscores, they are translated into "/" when including. (class Lister is found inside lib/Lister.php. Class Form_Field is found inside lib/Form/Field.php).

AModules3 does not include any files manually. All the necessary class includes are done through __autoload() function.

MVC

According to Wikipedia, Model-View-Controller is a software architecture that separates an application's data model, user interface and control logic into three distinct components so that modifications to one component can be made with minimal impact to the others. There are many major libraries who encourage MVC structure such as:

- Cocoa (used in Mac OSX native apps)
- Microsoft Foundation Classes (MFC)
- Qt Toolkit (since QT4) / KDE4.

There wasn't any MVC library for Web application until now (except few unpopular ones and WebObjects). Other popular Web / PHP libraries and CMS engines try to do a similar job but most of them have serious conceptual mistakes. Most popular is that they do not separate Data from Logic.

Basically, what people call a CMS (Content Management System) uses a similar concept to MVC. It usually provides a great way to change templates and a nice way to manage data in the database, but when it comes to Logic - there are no easy way to change it. Face it - how difficult is to modify a CMS site to become a Document Repository or customize editing interface. Many sites are using Smarty, which combines small portion of logic with View (embedded inside template tags). That's a bad architecture in my opinion.

AModules3 holds the answer to a proper implementation of MVC architecture. Not only it properly separates them, but it allows any of them to be modified easily and with infinite flexibility. At the same time AModules3 is not just a supplementary library - it holds a solution to a complete universal framework of YOUR application.

Multi API

API is a main class, which is the first one to initialize and it takes care of all the other ones.

When I was using PEAR::DB long time ago, I was thinking to myself - why do I need all that crap? I never use 70% of the framework, why do I have to include that? It becomes a bad habit for many libraries to grow “fat” where most of the code is used on very rare occasions.

I decided that AModules3 would be different. At the moment of writing, there are at least 5 different APIs in AModules3. They are designed to contain only the stuff which is relevant for the task.

- ApiCLI - a bare-bones API used for CLI application. Does not provide session support, don’t send headers.
- ApiWeb - a bare-bones API for for Web applications.
- ApiAdmin - a full-featured API for administration system development.
- ApiPortal - an A-Portal framework implementation on top of AModules. Currently in development and might appear at later versions.
- ApiInstaller - a framework designed for “install” applications. You would want user to use such an application when he installs your package on his web-server. Currently in development.

All the API classes are as clean as possible. Even authentication is not implemented directly inside ApiAdmin - instead it’s a separate class.

When you will want to build your own application, you will need one API class. Normally you should create your own API by inheriting it from one of the provided classes. Then you redefine methods.

Many things I will be talking about in this document only applies to ApiAdmin or ApiWeb. The similar concept might be shared in other APIs.

Class types

All the classes in AModules3 are inherited from the following 3 classes:

- AbstractModel - mainly used as a base of APortal classes or in your own code.
- AbstractView - all the classes which draw something on the screen.
- AbstractController - used to create abstract layers or interfaces.

AModules3 does not have many Model classes, because those would be specific to your own application. However it provides many View classes you can use to represent information.

APIs are inherited from AbstractView (View instead of Controller for APImight seem illogical. This is only exception. Reason for exception is that we want API to render itself and use templates).

Some classes are designed as add-ons, which are optional but would provide extended functionality. For instance - Logger class would improve onscreen error display and would provide file logging capabilities. QuickSearch and Paginator classes may be added to Grid to

provide page-to-page navigation and quicksearch input box. Auth adds authentication to API.

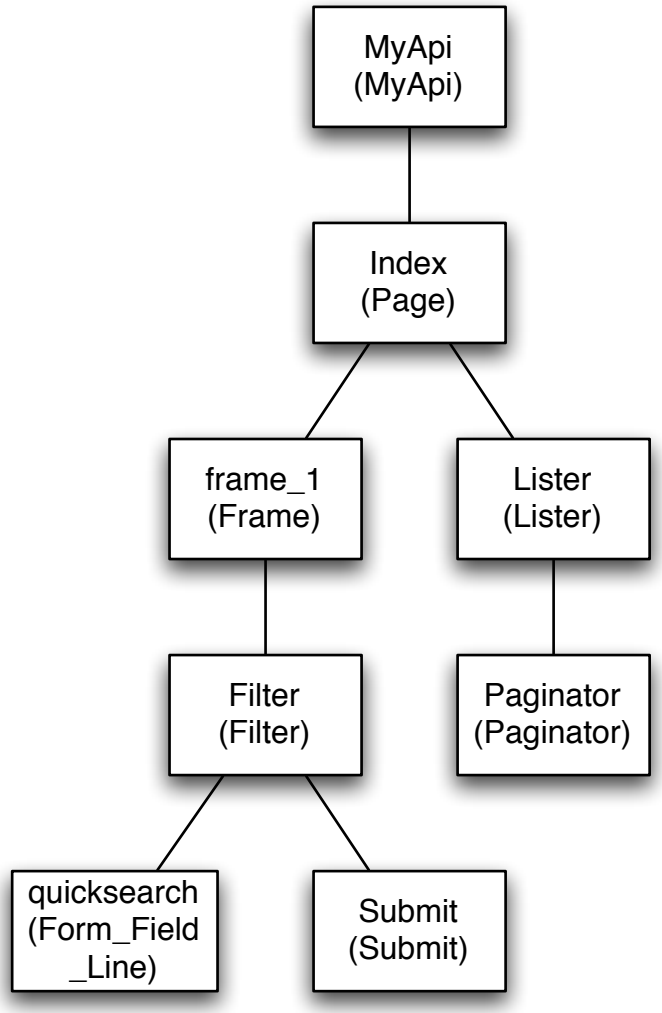
Runtime Object Tree

When your application is executed, it may create up to few dozens (x10) of objects. To keep track on those objects, they are organized inside a tree. Root object is always API. Any other object would have 2 links: $this->api and $this->owner, to point at root object and container object.

The only proper way to create objects in AModules3 is $this->add('Class_Name');. This would initialize new object as a child.

Each object have 2 names. Short name and full name. Short name looks like this: "Surname". Long name is composed from parent's name and child's short_name. It is unique and looks like MyApp_Index_Frame_Form_Surname.

You can see a typical Runtime Object Tree for ApiAdmin-based application on the right:



When you create new object using add() method, 2nd argument specifies a child's short_name. If you do not specify 2nd argument or leave it blank, then the class name will be used. There are cases, when it might be not unique. AModules3 will generate exception if you will try to create more than one View with the same name. The behavior is different for Model and Controller objects (when they short_name already exists). See documentation for add() method for more information. Here is a sample call from a "Index" object:

$this->add('Frame','frame_1');

Template system (SMlite)

I was looking at how Smarty - the most popular template engine works, when I realized it's not good for model of AModules3. We don't want the template to contain much logic. My opinion is that it's useless to re-write all the functions PHP can do in template engine. Why not use PHP instead?

SMlite, the default template engine in AModules3 uses concept of “regions”. You may find it similar to HTML / DOM. SMlite parses template and finds “region markers” inside it. Then it builds a region tree in the memory and allows to manipulate regions easily.

If you come to think about it - there are so many pages on the web with notes in HTML code which are similar to this:

```
<!-- header ends here -->
<!-- body starts here -->
```

SMlite have the same basic idea. But while normally you add those comments for your own reference, SMlite actually makes use of them. And the tags looks slightly different (to protect them from PHP-aware WYSIWYG editors):

```
<?header?>
 <head>..head code..
 </head>
<?/header?>
<?body?>
 ... here goes the body.

<?/body?>
```

SMlite allows to get content of any tag and replace that content. This works well when you want to add dynamic objects to replace parts of a static page.

Another important goal of SMlite is that it’s templates are easily readable in browser or HTML editor. Those usually hide <?..?> parts and display the rest of the page. In order to convert static HTML page into template all you need is to insert the tags. No other modifications are required.

Runtime Object Tree + SMLite template system = ?

From the previous 2 sections you know that objects and templates have tree-based structure. Those are joined together when objects are created. When sub-object is added, we specify name of the region for it. Dynamically generated View object will insert it’s output into specified region on the parent’s template.

For example, if you have <?Menu?> region defined on your template page, you can do the following:

```
$api->add(‘Menu’,null,’Menu’);
```

This code will add object of type Menu (with default short_name=‘Menu’) on the main page layout. When your application is executed, it will replace content of <?Menu?> region with an output generated by a new object.

Here is another example you can try in your application:

```
$api->add('LoremIpsum',null,'Content');
```

Class LoremIpsum generates few paragraphs of famous text-filler. It's output will be inserted into region <?Content?>.

Pages

A problem with PHP is that it inherits the "page" concept from static HTML files. The following URL will end up executing page "about.php".

```
http://yoursite.com/old_project/about.php
```

While it might seem a best way for PHP developers, it is a very complicated way of doings things. First it rejects Class / Object structure - 'about.php' would contain the code to display this particular page. It may use some includes, but the logic is inside the file. You can't inherit files (unless you are a Perl programmer).

AModules3 maps pages directly into Class names. The following URL:

```
http://yoursite.com/new_project/About
```

will be automatically rewritten by AModules3 into:

```
http://yoursite.com/new_project/main.php?page=About
```

All the execution is done through 'main.php' file. You can actually rename this file, it's only a small wrapper. Once API is initialized, it's responsible for doing necessary actions based on the page name. It's not Apache who finds the code to prepare your page now, but the API class. This makes much more sense.

For example, if you are using ApiAdmin, it automatically comes with Login, Logout and About pages. Of course you can redefine them, but normally you wouldn't do that.

AModules3 uses GET arguments too, which look like this:

```
http://yoursite.com/UserEdit&id=123
```

Because of mod_rewrite it does not use question mark. This might give some troubles on Apache+Windows (because of incorrectly implemented case handling in Windows). For compatibility you can use "main.php?page=UserEdit&id=123" form (without mod_rewrite). See documentation on ApiAdmin / getDestinationURL() for more information.

When ApiAdmin is executed, it would do the following:

1. Calculate page name and store in $api->page (in our case UserEdit)
2. Create new sub-object of class 'Page' and short_name 'UserEdit'
3. Execute function $api->page_UserEdit($page)

Normally you would place your initialization code inside page_UserEdit function, such as: $page->add('LoremIpsum',null,'Content');

There are more ways to define pages, read on.

How objects use templates

If you have worked with some other template engines, there is a chance you had to create separate files for each element. AModules3 is flexible. When you add() new sub-object, you can specify template to it. Paginator is a good example. If you use the following syntax to add paginator:

```
$this->add('Paginator',null,'paginator');
```

It will just work. Because Paginator knows where his template is located. But you can specify alternative template for it:

```
$this->add('Paginator',null,'paginator',array
('paginator','ajax_paginator'));
```

By specifying 4th argument to add() method, you can tell object which template it should use. There are few ways to specify a template, but here I used array. 1st argument specifies template name and second a region to use as template. SMlite will parse file "templates/paginator.html" and will use <?ajax_paginator?> contents for object rendering.

Sometimes the sub-element's template may be on the same page as the parent's. In this case you call it like this:

$this->add('Menu',null,'Menu','Menu');

add() will look for "Menu" region in $this object's template and would clone it for newly created Menu object.

Layouts of ApiAdmin

ApiAdmin uses standard template. It believes that it might find some tags in the standard template: Menu, Content, LeftSidebar, RightSidebar and InfoWindow. Those regions of shared (main) template are called Layout. ApiAdmin defines functions which are responsible for rendering those regions: layout_Content, layout_Menu.

If you are willing to add a dynamic banner on your page and if it does not fit into any of those regions, you would probably need to add one more method: layout_Banner(). Also initLayout() should be redefined and perform "addLayout()" call. That function insures you have both template region and layout_ function.

This structure allows you to specify a blank template for your API sometimes, and API wouldn't bother to initialize all those layouts because there are simply no place for them on your template. That's useful for popups and generic APIs.

init() and excternal initialization

Each object you add will have init() function. This function is called right when object is inserted to the page.

Normally you should redefine init() method to put some custom initializations for the object. You should call parent's init() first, then initialize the object in a proper way. For example for Grid you have to define a set of Columns. You should also specify data source.

However there is another way without customizing class. Instead of putting $this->addColumn into your custom class inherited from Grid, you may rather write like this:

```
$g=$this->add('Grid',null,'Content');
$g->addColumn('test');
```

Except some rare cases, it just does the same thing. Almost all objects of AModules3 can be used with external initialization. Of course it's more limited, but sometimes you don't need all that. Use external initialization when you do not plan to customize class itself.

memorize, learn, forget and recall

Those functions are very important. When you want to use $_SESSION, you should use those functions instead.

Why you mustn't use $_SESSION - because it's global variable and you will run into conflicts, if your class have several instances.

Value is remembered inside session. $this->memorize() would save value for current object only. That means - if the same object, on the same page, on same api will execute recall(), it will be able to access saved value. Other objects will not mess up with memorized data. If you want to share data between several objects, you can use $api->memorize().

Most common place where you want to use those functions is with optional arguments to your page. In this case you will want to use variable, which is initialized in the following way:

- if $_GET argument is present, it is used and remembered
- if $_GET is not specified, remembered value is used
- if no value is remembered and $_GET wasn't specified, use default value.

Function learn() is good for this purpose.

TODO - fix the way learn works()

Application Developer Introduction

Now, you have decided to create your application. 2nd part of this document will guide you though the practical usage of AModules3 for ApiAdmin based applications.

kick-start

There is a shell script located in tools/kickstart-admin.sh. This file initializes basic file structure for your application:

```
mkdir new_project
cd new_project
svn co https://adevel.com/amodules3/trunk amodules3
./amodules3/tools/kickstart-admin.sh
```

At this point you can open your web browser and look at the new directory:

```
http://yoursite.com/new_project/
```

Note, that older Apache server had a bug in it. If you don't specify trailing slash, the mod_rewrite will miserably fail and would display you scary error message. If you send URLs to other people, sometimes last slash might get lost. You might want to add page name at the end of the url, such as "Index".

```
http://yoursite.com/new_project/Index
```

You do not have to keep "amodules3" as subdirectory of your project. It can be located anywhere. Kick-start script will take care of using the right include. Also, by default the .htaccess file will use "wrap.php" for images, css files and javascript files. That would work even if your amodules3 located outside web-space.

However in our example we specifically created sub-directory amodules3 inside new_project. That means you can safely remove files containing reference to "wrap.php" from your .htaccess file. You can remove the wrap.php file itself.

Work by default philosophy

AModules3 work by default. That means you don't have to write anything to make it work. It just work. You write your code to customize it. When you will first visit your page, you will see a static page with Menu and Content on it. It's all up to you to replace. But before that, we need to create a page.

"Page" is really simple class. You can look at lib/Page.php and you will be amazed. Page is just a simple container for other. There are two ways to specify a Page:

First way is to simply create function "page_PageNameHere" inside your redefined Api class. However as your application grows, you may find it hard to have all that code in a single file

(where your Api class is described). That's why there is another way. You can create file inside page/PageNameHere.php and redefine it's init() method to achieve the same goal.

Skins in ApiAdmin

When you create a normal web application, you most probably would want to go with custom templates. If you are writing an administration interface which normally is only available to limited number of people, the unique page look is not so important anymore. ApiAdmin comes with skins. Skin is a set of templates which is used by elements such as forms, grids, menu and others.

There might me more than one skin to choose from. If you do not like default skin, you can change it by specifying 2nd argument to the API constructor.

$api = new ApiAdmin('myadmin','custom_skin');

skin templates will be searched inside amodules3/templates/custom_skin and then inside templates/custom_skin, so you can place them locally.

How to build applications on top of ApiAdmin

When using ApiAdmin, you don't have to worry about details. All you have to do is to put objects on your page and customize them.

First you need to create class inherited from ApiAdmin and redefine it's init() class. init() is where you can initialize something global, such as:

- Authorization (access restriction)
- Add namespaces (extensions/plugins to your application)
- Customize main template (add custom JS code)
- Remove or Disable some layout elements (remove 'Locator')

Earlier versions of AModules3 wanted you to initialize menu, database connection, config file from init(), but it's not required anymore. Everything is cleaned up at this point

ApiAdmin comes with the concept of "pages". .htaccess will translate the URL like this:

http://example.com/admin/EditClient&client_id=123

into main.php?page=EditClient&client_id=123

ApiAdmin will understand page name. First it will attempt to load class page_EditClient (must be inherited from Page class), which might be located in page/EditClient.php file. If this file does not exist, it will create object of class Page and then call

$api->page_EditClient($p);

You should define this function and put objects on the page. Since name says something about editing client, you would need to put a form here. Please refer to AModules3 Cookbook, which will give live examples on how to create forms, grids etc. Simplest page you can come up with is:

```
function page_EditClient($p){
    $p->add('LoremIpsum',null,'Content');
}
```

To get more information, you should read cookbook and put the code inside functions like this one.

Further reading

I have already mentioned AModules3 CookBook. You should also look at existing projects. See the samples and mini-apps included with AModules3 distribution.

Library Developer Introduction

Prepare yourself

If you are willing to understand or edit library classes, you will have to make sure you know PHP5 by 100%. Even experienced developers who looked through the code often found some constructions they couldn't understand or which made them wonder, how that could work.

I have wrote majority of the library code trying to make it as slick as possible, as fast as possible and as flexible as possible. Very often it's hard to catch my idea. If you do not understand meaning of some functions and no comments are available, please contact me instead of jumping into conclusions. I will comment the code and explain how function in question works.

Please read DEVELOPERS file, which contains guidelines to code editing. If you are editing code, you must match it to the rest of the library. Even if it does not seem proper to you, you should follow library guidelines.

How you shouldn't develop

I follow the rule, that only code which is essential to the task must be parsed. Even if ZendOptimizer would pre-parse the whole library, I think it's important for code structure to be clean from all possible modifications.

Let me give example. As we were working on the library, I haven't made any authorization class. Another developer decided to implement the Auth and he did. In the result, he had a rather big class, which had many interesting features - it was authenticating users against the database, it supported password reminder, it sent out email notifications and supported registration page. Of course those features were optional. I didn't liked the class and replaced it with BasicAuth.

BasicAuth implements only core authentication. It even don't use database, it just shows login screen and relies on function verifyCredintals(), which is rather dumb by default. The initial Auth was rewritten on top of this class.

With this example I want to point out, that initial implementation didn't follow rule of thumb in AModules3. It was too complicated. Why would I want to use this Auth class, if I need only 15% of it's functionality?

To summarize: Rule of thumb of AModules3 is that any class you create must serve a narrow function. If you want it to have more features, implement those as additional classes. Read this part of the introduction to learn how to do that. "All-in-one" classes will be kicked out from AModules3 and you will have to re-write them.

Booleans [new in rev3]

In AModules we use the following two values for a boolean values:

- 'Y' - true
- '' - false

This works with if() and it looks good if user sees this. It's also compatible with checkbox fields. Theoretically it also does not consume much memory and allows for possible custom values in the future with rather reasonable naming.

Use described scheme for any boolean values you might have in the database.

Hooks

Hooks allows you to specify a call-back function to be called from other object. Api have many hooks itself, to execute everything in the proper order.

For example hook 'pre-render' can be found in ApiWeb and ApiAdmin. If you want your function to be executed just before rendering, you should hook it up to 'pre-render'. Class Grid uses this 'hook' to pre-cache it's template, because it does not want to construct template for each line, it constructs it before rendering starts.

So if you are willing to add some optional or specific code to existing class, then place a hook in existing class, and put your code into separate class.

If you think about it - hooks insert code into classes, just like SMlite and Runtime Object Tree inserts HTML code inside parent's template.

Down-calls

Down-call allows to call method for each child in the Runtime Object Tree. Most popular example is rendering. Each object renders itself and places inside owner's template. Another example of down-call happens when form is submitted.

There are also up-calls used in error handling. It tries to execute function for each owner until someone handles it properly.

Error handling

AModules3 recognizes 4 types of messages: info, debug, error and exceptions. If your object calls $this->info('It's a nice weather today'); this information will be passed on to API, which will output it. Exceptions are caught by Api class and handled.

API classes have only basic capabilities of handling errors. However there is class Logger, which enhances this experience. $api->add('Logger') would make your error output look better, would allow you to save errors into files etc. Note that ApiAdmin initializes Logger automatically.

Namespaces

Namespaces allows you to create bundles with classes, which are plug-able into any application. ApiAdmin supports them - all you have to do is to call $this->addNamespace() from $api->init().

Depending on the namespaces, they may add additional menu items or controls on the page. Namespaces are good for:

- implementing help system
- implementing error reporting system
- splitting big application into parts
- implementing anything which is optional for your application

Good thing about namespaces that they can be removed easily and they can be ported to other application without no effort at all, just by copying folder.

Namespaces may add additional pages to your application which URL will look like:

http://example.com/admin/help;Contents

In this case, page Contents is implemented inside namespace "help". See sample 08-namespaces for practical instructions on implementing namespace.

Further reading

Please read source code :)