



Minimalistic PHP Framework for Backends

User Guide v. 0.8

Overview

Rapyd is minimalistic php framework made to build applications in few lines of code.

Its based on a widespread pattern: MVC (Model, View, Controller).
Since 0.8 It support also HMVC (Hierarchical MVC)

It's inspired by CodeIgniter and KohanaPHP.
Rapyd is probably less mature, but give it a chance.

Features

- No compiling, no command line needed, just unzip, it should work.
- Widget oriented architecture (grids, forms, tables, etc..).
- Simple syntax, oriented to build simple (or complex) data GUI in few line of code.
- Modules support (each module is just "reply" of the application folder).
- you can integrate rapyd with any other php script (like wordpress) in easy way
- Nice URLs / SEO oriented (since 0.8 the framework does not require querystring).
- Really open source (MIT licensed)

Installation

System Requirements

1. PHP 5.1 or higher (required)
2. pdo-sqlite3 to run samples (it support also MySQL, postgresSQL)
3. apache webserver with mod_rewrite enabled (for really clean urls)

Configuration & Deployment

You must configure the application in /application/config.php (locale, path, db connection).
Rapyd comes with a Demo module (with several examples of data presentations and editing),
but remember, you should remove the demo folder on production.

URLs and Flow

Rapyd URLs

Rapyd uses URI (uniform resource identifiers) to determine which area and which action of your application to call.

So instead classic “php application” urls:

http://site.com/welcome.php
http://site.com/register.php?step=1

It use:

http://site.com/index.php/welcome/index
http://site.com/index.php/registration/step/1

(and with a simple .htaccess and mod_rewrite enabled “index.php” can be omitted)

In an mvc framework there is a system gateway: the only script called “directly” via http request, in our case this is the “index.php”, it include all needed php classes and process the request.

In the above examples rapyd will work as follows:

- instancing a class*: the controller “welcome”, then calling one of its methods: “index”
- instancing a class*: the controller “registration”, then calling its method “step”, passing the param “1” to it.

(must extend “rpd” class, and must be placed on the: /application/controllers folder)*

So, in a site driven by rapyd framework, to build a “page” you simply need to write a class declaration, with at least a method (that takes parameters or not) which print something.

This is enough but a bit poor, so in each MVC framework (where C is Controller) there are also **M**odels and **V**iews.

Models are classes where you should do db-stuffs, and Views are simple php/html files where you can perform echos and iterations, they're output “templates”.

A controller can instance and use models, then can pass data to a view and render the output.

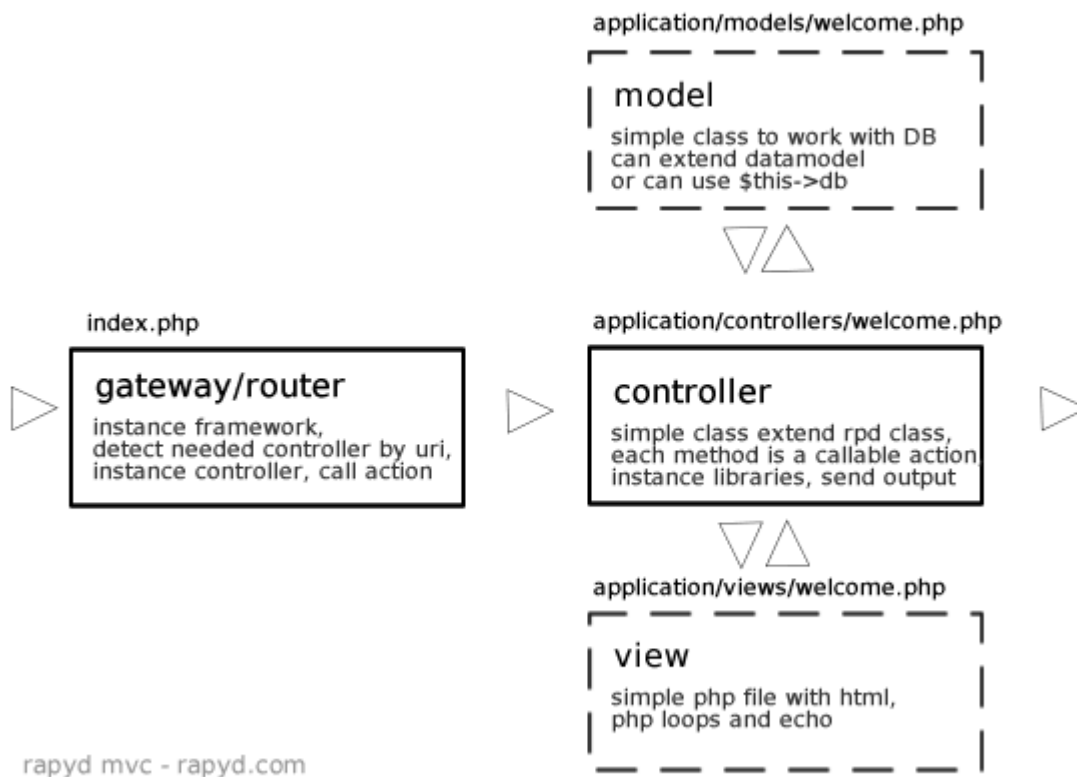
Application Flow

request

<http://website/index.php/welcome>

response

html output



rapyd mvc - rapyd.com

In Rapyd, and other MVC frameworks:

- All non static resources are served by a system gateway (index.php).
- The gateway detects by URI the needed resource and instances the right "Controller".
- A Controller is a simple class where each Method is a callable action.
- A Controller Method can send output by echo or print.. but if you need, it can instance models to perform db-queries, then include a view to send an output.
- A Controller should be placed in **/application/controllers**
- A Model should be placed in **/application/models**
- A View should be placed in **/application/views**

And the "flow" is:

request > **gateway** > **controller** [> **model**] [> **view**] > **output**

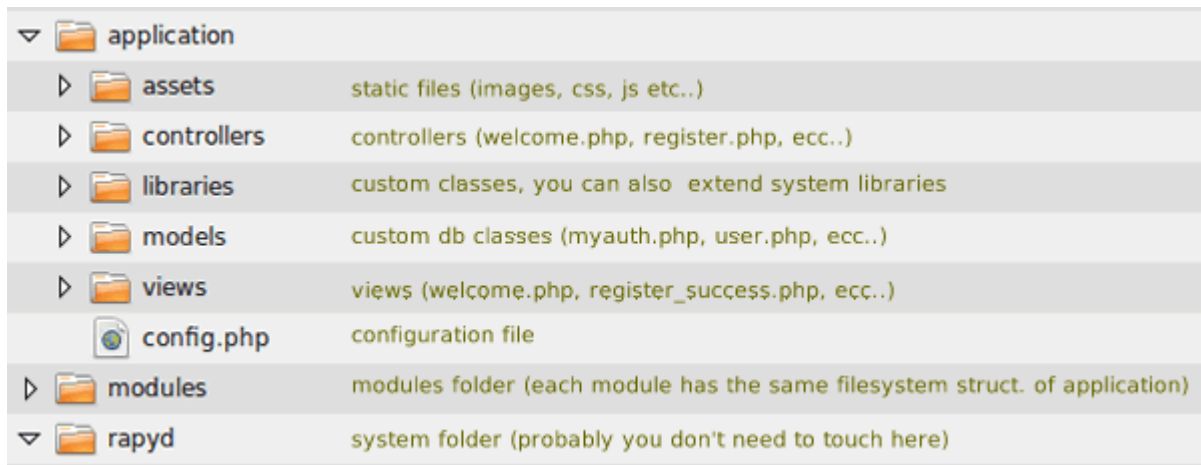
Note:

Rapyd also supports "modules" which are like "sub-applications", and cascade resource load.

Note:

In Rapyd, there are also, "Libraries" and "Helpers" (basically with similar concept of Models, except for their goal).

Filesystem



▼ application	
▶ assets	static files (images, css, js etc..)
▶ controllers	controllers (welcome.php, register.php, ecc..)
▶ libraries	custom classes, you can also extend system libraries
▶ models	custom db classes (myauth.php, user.php, ecc..)
▶ views	views (welcome.php, register_success.php, ecc..)
config.php	configuration file
▶ modules	modules folder (each module has the same filesystem struct. of application)
▼ rapyd	system folder (probably you don't need to touch here)

Rapyd Filesystem is clean, “system” is separated from “application” (usually you can upgrade by replacing “rapyd” folder).

Furthermore, you can isolate application pieces in “modules” folder (so you can for example organize a cms in modules).

If you need to customize “rapyd/libraries” you can build your extended version in “application/libraries”

Note:

Rapyd use class autoload, you do not have to worry about load classes before instancing or extending, Rapyd autoload will find and include all needed files (even if they are in a different folder/module).

Controller

A controller is a simple php class it must extend rpd superclass (so it can inherit `$this->db`, `$this->uri` and some other system libraries).

Conventions/rules:

- Class name must end with `_controller`
- Filename must be minuscapse, with the same class name (without `_controller` suffix)
- File must be deployed in `/application/controllers/` (or in `/modules/modulename/controllers/`)

Sample

```
<?php
class welcome_controller extends rpd {

    function index()
    {
        $vars = array('varname' => 'value');
        echo $this->view('welcome', $vars);
    }

    function test()
    {
        echo 'test';
    }

}
```

This request:

http://site.com/welcome/index

will parse the View 'welcome' then the result will be printed

This other request:

http://site.com/welcome/test

will output 'test'

View

A view is a simple php file with html, php loops, echos etc...

Conventions/rules:

- Filename must be minuscapse
- File must be deployed in /application/views/ (or in /modules/modulename/views/)

Sample

```
<html>
<head>
  <title>Welcome !</title>
</head>
<body>
  <h1>Welcome</h1>
  <p>This is a simple html file with some php var <?=$varname?> </p>
</body>
</html>
```

Note:

If you are thinking “where is smarty?.. I need a template engine!?”

my answer is .. No, there isn't a template engine, and you don't need it.. see below

Sample using view in view... and HMVC!

```
<?=rpd::view('header', $input_data)?>

<h1>Welcome</h1>
<p>This is a simple html file with some php var <?=$varname?> </p>

<h2>user info</h2>
<?=rpd::run('userbox','info', array('short_style'));>

<?=rpd::view('footer', $input_data)?>
```

TA-DA!..

rpd::view let you call another view, “\$input_data” is a wildcard, it means: pass all variables of current view to the subview. (typically you can use it for boxes, or header & footer)

rpd::run let you call \$controller->method(\$params);

so you can gain a great level of isolation. You can isolate modules, then perform integration in easy way. “run” is also the way you can bind rapyd & third party scripts like wordpress, phpbb.. etc..

Models / DB / Active Record

A Model is a class where you should do db stuffs. In example, you can see below how to perform simple queries, or use active record implementation (which is used by widgets). Using active record way is great to tokenize queries (for example to execute conditional statements) , it also do automatic escape on assigned values.

```
<?php
class articles_model extends rpd
{
    function __construct()
    {
        $this->db = rpd::$db;
    }

    //using simple query
    function get_categories()
    {
        $sql = "SELECT c.category_id as id, COUNT(a.article_id) as tot, c.*
        FROM articles_categories c LEFT JOIN articles a USING(category_id)
        WHERE c.public = 'y'
        GROUP BY c.category_id
        ORDER BY c.priority ASC ";
        $this->db->query($sql);

        if ($this->db->num_rows() > 0)
        {
            return $this->db->result_array();
        } else {
            return array();
        }
    }

    //using active record
    function get_articles($limit=10, $category="", $rand=false)
    {
        $this->db->select("a.*");
        $this->db->from("articles a");
        $this->db->where("a.public", "y");
        if (is_numeric($category))
        {
            $this->db->where("a.category_id", $category);
        }
        if ($rand)
            $this->db->orderby("RANDOM()");
        else
            $this->db->orderby("article_date", "desc");

        $this->db->get(null, $limit);
        if ($this->db->num_rows() > 0)
        {
            return $this->db->result_array();
        } else {
            return array();
        }
    }
}
}
```

Widgets

Ok, we now know how mvc works, and how to execute queries, but as you can see on the rapyd website or in the demo there are stuffs like grids, forms, etc.. they are “libraries” or widgets.

Rapyd is focused to build in few lines of code “CRUD” interfaces (Create, Read, Update and Delete), it's thinked to make backends, so it comes with a set of ready widgets for data-presentation and data-editing.

Each widget extend a basic “component”, and is designed to work together whit others, they share for example the same instance of “db”.

Widgets semantic

A widget has a behavior driven by “uri segments” (this is a change introduced in the 0.8, previous versions was driven by query string). For example:

<http://site.com/index.php/grid/index/pag/2>
http://site.com/index.php/grid/index/orderby/article_id/desc
http://site.com/index.php/grid/index/orderby/article_id/asc/pag/2

(this show how a datagrid work with offset and order)

<http://site.com/index.php/edit/index/show/1>
<http://site.com/index.php/edit/index/modify/1>
<http://site.com/index.php/edit/index/create/1>

(this show how a dataedit change editing status)

http://site.com/index.php/filtered_grid/index/search/1/orderby/article_id/desc

(this show how datafilter and datagrid work together: execute a search, then order results)

DataGrid widget

Article List			
ID	△▽	Title	Body
 7		Title 7	Body 74444
 8		Title 8	<p><strong
 9		Title 9	Body 9
 10		Title 10	Body 10 da
 11		pippo	C R A P<fo

[< Prima](#) [< 1](#) **[2](#)** [3](#) [4](#) [5](#) [6](#) [> Ultima](#) [>](#)

Sample

```
$grid = new datagrid_library();
$grid->label = 'Article List';
$grid->per_page = 5;

$grid->source('articles');
$grid->column('article_id', 'ID', true)->url('edit/show/{article_id}');
$grid->column('title', 'Title');
$grid->column('body', 'Body')->callback('escape', $this);

$grid->build();

$data['head'] = $this->head();
$data['content'] = $grid->output;
```

Properties

Property	Default Value	Options	Description
label	"	string	label to display for the grid
per_page		integer	the number of records to display per page

Methods

\$grid->source(\$source)

\$source - mixed may be a db-table name, a sql-query, a datafilter object, an associative array matrix

\$grid->column(\$pattern, \$label, \$is_orderby)

\$pattern - string field pattern, field name, or content of cells

\$label - string column label

\$orderby - boolean if column is sortable

\$grid->build()

build grid, fill \$grid->output

DataFilter widget

Article Filter

Title

Public Yes No

Sample

```
$filter = new datafilter_library();  
$filter->label = 'Article Filter';  
$filter->db->select("articles.*, authors.*");  
$filter->db->from("articles");  
$filter->db->join("authors", "authors.author_id=articles.author_id", "LEFT");  
$filter->field('input', 'title', 'Title')  
    ->attributes(array('style' => 'width:170px'));  
$filter->field('radiogroup', 'public', 'Public')  
    ->options(array("y"=>"Yes", "n"=>"No"));  
  
$filter->buttons('reset', 'search');  
$filter->build();  
  
$data['head'] = $this->head();  
$data['content'] = $grid->output;
```

Properties

Property	Default Value	Options	Description
label	"	string	label to display for the filter

Methods

\$filter->source(\$source)

\$source - mixed may be a db-table name, or a sql-query

\$filter->field(\$type, \$name, \$label)

\$type - string field type (input, password, checkboxgroup, checkbox, radiogroup, radio, dropdown, date, editor)
\$name - string field name, usually the db table field to append in the where clause
\$label - string label to display for the field

\$filter->field(\$type, \$name, \$label)->attributes(\$attributes)

\$attributes - assoc. array array of extra attributes to build for column

\$filter->field(\$type, \$name, \$label)->options(\$options)

\$options - array array of options (for field types like dropdown, radiogroup etc..)

\$filter->buttons(\$button1[, \$button2...])

\$button n - mixed name of buttons to build ('reset' and 'search' are the standard buttons available for datafilter)

\$filter->build()

build filter, fill \$filter->output

DataEdit widget

Manage Article Modifica

Title	Title 2
-------	---------

Public	Yes
--------	-----

Author	Jhon
--------	------

Date	01/04/2009
------	------------

Description

Body 2!! kjkjkjkjыыык

Torna all'elenco

Sample

```
$edit = new dataedit_library();
$edit->label = 'Manage Article';
$edit->back_url = $this->url('filtered_grid/index');

$edit->source('articles');
$edit->field('input','title','Title')->rule('trim','required');

$edit->field('radiogroup','public','Public')
    ->options(array("y"=>"Yes", "n"=>"No"));

$edit->field('dropdown','author_id','Author')
    ->options('SELECT author_id, firstname FROM authors')
    ->rule('required');
$edit->field('date','datefield','Date')
    ->attributes(array('style'=>'width: 80px'));

$edit->field('editor','body','Description')->rule('required');

$edit->buttons('modify','save','undo','back');

$edit->build();

$data['head'] = $this->head();
$data['content'] = $edit->output;
```

Properties

Property	Default Value	Options	Description
label	"	string	label to display for the filter
back_url	"	string	url of page where to go back (when we click on back button)

Methods

\$edit->source(\$source)

\$source	-	mixed	may be a "datamodel" object (or extended one), or the name of a db table (in this case dataedit will instance a new datamodel for us)
-----------------	---	-------	---

\$edit->field(\$type, \$name, \$label)

\$type	-	string	field type (input, password, checkboxgroup, checkbox, radiogroup, radio, dropdown, date, editor)
\$name	-	string	field name, usually the db table field to append in the where clause
\$label	-	string	label to display for the field

\$edit->field(\$type, \$name, \$label)->attributes(\$attributes)

\$attributes	-	assoc. array	array of extra attributes to build for column
---------------------	---	-----------------	---

\$edit->field(\$type, \$name, \$label)->rule(\$rule)

\$rule	-	mixed	\$rule can be 'required' or can be the name of a custom or native php function (like trim), it's possible to pass an array instead a single rule, or use a serialized syntax like 'required trim' using a pipe as separator
---------------	---	-------	---

\$edit->field(\$type, \$name, \$label)->options(\$options)

\$options	-	array	array of options (for field types like dropdown, radiogroup etc..)
------------------	---	-------	--

\$edit->buttons(\$button1[, \$button2...])

\$button n	-	mixed	name of buttons to build ('modify', 'save', 'undo' and 'back' are the standard buttons available for dataedit)
-------------------	---	-------	--

\$edit->build()

build dataedit, fill \$edit->output