Anatomy of the framework

- **/lib** contains the main classes of the framework.
- **/tmp** where the temporary files should go. *– better than the global /tmp folder*
- **/log** where the session log file goes.
- **/src** the applications folder.
  - **/src/apps :** contains all the apps
  - **/src/config** contains different config files for each environment
  - **/src/helpers :** shared helpers
  - **/src/layouts :** shared partial views
  - **/src/models :** shared data model blueprints
  - **/src/tools :** shared extra files/functions *– needs to be included manually*
- **/www** publication folder. The only needs are the index.php and .htaccess file ; after that, it's all about assets.

Anatomy of a config file

```
<env>
	<!-- email of the admin of the deployed framework instance. Used mainly for security mails  -->
	<admin>barbay.julien@gmail.com</admin>

	<!-- mysql database description  -->
	<database>
		<host>127.0.0.1</host>
		<name>mydb</name>
		<user>root</user>
		<pass>root</pass>
	</database>

	<!-- routing description  -->
	<routes domain='mydomain.com'>
		<!-- specify subdomain's app -->
		<subdomain name='www'>
			<!-- specify app folder and app slug in the url -->
			<module name='front' url='/'>
				<debug /> <!-- enables debug log to be append to the html -->
			</module>
		</subdomain>


		<!-- how to specify an other app in the framework -->

		<!-- 1/ other subdomain -->
		<subdomain name='api'>
			<module name='api' url='/' />
		</subdomain>

		<!-- 2/ specific app slug -->
		<subdomain name='www'>
			<!-- this needs to be append to the first 'www' subdomain node since it's using the same subdomain -->
			<module name='api' url='/api' />
		</subdomain>
	</routes>

	<!-- specific constant definition. each will be solved and defined as php constant -->
	<constants>
		<constant name='google_tag' value='UA000000' />
	</constants>
</env>
```

Anatomy of an app

in the /src/apps folder, you get this micro structure :

- **/myapp/functions.php** the starter file of the application
- **/myapp/controllers** where the controllers files are meant to be
- **/myapp/templates** where the templates files are meant to be

Framework's bootstrap

The publication folder contains the following index.php file :

```
<?php
	require('../lib/App.php');		//require the main file of the framework
		App::dispatch('dev');		//launches the app with the specified config file
?>
```

And works with the following htaccess rules :

```
##### INI SETTINGS
php_value session.name 'SSID'
php_value session.auto_start 0
php_value session.use_only_cookies 1
php_value session.cookie_httponly 1
php_value session.hash_function 1
php_value session.hash_bits_per_character 6

##### REWRITING
Options +FollowSymLinks
RewriteEngine On
RewriteBase /

# This means every physic file will be resolved
RewriteCond %{REQUEST_FILENAME} -f [OR]
RewriteCond %{REQUEST_FILENAME} -d
RewriteRule ^ - [L]

# If the first rule is not resolved, this one will be redirecting anything to the index file
RewriteRule ^.+$ index.php [L]
```

So, any url that is not pointing a valid file will be redirected to the index, which will launch our framework's main class.

Step by step rendering process

1/ Autoload and definitions
It will include ./load.php which defines some constants :

ROOT	/ (of the framework)
TMP	/tmp
WWW	/www
SRC	/src
APPS	/src/apps
LYT	/src/layouts
MDL	/src/models
HLP	/src/helpers

It will also load **any** file in the **MDL** folder.

2/ Config file loading
It opens the config file named with the argument passed in the dispatch function. Here, dispatch('dev') will load /src/configs/dev.xml

3/ XML parsing
Defines the DB properties as constants prefixed by DB_ and builds our routing data structure as described in the xml file

4/ Extra data definition
When the url is solved, 3 properties will be defined :

App::$real	The name of the application as written in the url
App::$name	The name of the folder of the application
App::$path	APPS.'/'.App::$name
TPL	App::$path.'/templates'

5/ i18n file loading
If the **/myapp/i18n.xml** file is found, it will be automatically loaded in App::$i18n.

6/ App starter file
If the **/myapp/functions.php** file is found, it will be automatically included.

If you want to to a pre-session start check, you can implement is special function in the functions.php file :

```
<?php function launch( ) { } ?>
```

You can also implement a pre dispatch function which will be called before any front controller handling.

```
<?php function pre_dispatch($path_as_array) { } ?>
```

7/ Session start
A special session manager starts a safe session

8a/ Front controller handling
You can provide your own front controller by writing the following function in the functions.php file :

```
<?php function handle($path_as_array) { return $post_dispatch_data; } ?>
```

If not defined, the default front controller implemented will take care of the routing.

8b/ Embedded front controller
The process splits the url by '/' and analyses the returned array.
The typical pattern looks like : protocol://subdomain.mydomain.com/{app_slug}/controller_slug/action_slug/{extra_parameters}

The controller slug matches the controller php file name in the /myapp/controllers folder. In no controller is provided, the "index" controller is used.
The action slug matches a function name that will be available right after the controller file include. If no action is provided, the "index" function is called.

Examples :

```
http://www.mydomain.com/
	include of /myapp/controllers/index.php
	call of index($parameters)

http://www.mydomain.com/products/all
	include of /myapp/controllers/products.php
	call of all($parameters)
```

If the controller or the action are not found, a 404 header is sent.
Therefor, a generic action can be called by implementing the following function in the controller :

```
<?php function wildcard($path_as_array) { return $post_dispatch_data; } ?>
```

This can be usefull for i18n slugs or special 404 handling.

9/ Do it your way
Lots of functions are available to help you writing your app.

App::render($tpl, $data = null) will be usefull to render a specific template.
The $data object will be passed into your template as the $view object.

App::partial($tpl, $datas) will render the specified template with each entry of the $datas array.
A specific 'partial_index' property will be added to help counting the occurences.

App::display($override = null) will end the rendering process and will loop through the override associative array to perform a key to value replacement.

App::link($to, $application = '') will generate and return a valid link to $to, according to the specified $application. If not, current will be used.

App::module($application) will generate and return a valid link to the base url of $application

App::redirect($url, $code = 302, $auto = true) will send a $code Location header to $url. if $auto, $url will be solved as App::link($url)

App::crossdomain($url) will send an Allow Access Control header to $url. This helps with ajax calls through differents subdomains

App::debug($content, $level = 0) will log $content into a special buffer which will be printed where the {DEBUG} keyword is when calling App::display

App::error($number, $msg = '') will send a $number header. It'll also die with an <h1></h1> and the $msg string.

10/ Display and post dispatch
When you're done with your controller, the returned data can be used somewhere.

You can implement a post dispatch function which will be called after any front controller handling.

```
<?php function pre_dispatch($post_dispatch_data) { } ?>
```

It's usefull if you want to group json_encode output or anything else.

Data models

There's an abtract model that can be extended to manage models. You can write your a simple model like this :

```
<?php
	class Admin extends Model
	{
		protected static $table = 'admin';				//mysql table
		protected static $fields = array(	'id'		=> 'int',	//mysql field => php type
							'login'		=> 'string',
							'password'	=> 'string'
						);

		public function __construct() {}
	}
?>
```

Then, the abstract Model takes care of everything.
You can use those functions :

Admin::get($elements = 'all', $order = '', $simplify = true) performs a custom SQL select statement.
$elements can be sereval types :

- If 'all' keyword, an * will be used for the SQL select.
- If numeric, the where clause will try to match the first field in the $fields array with $elements
- If associative array, the where clause will join any $key = $value with an AND statement
- If string, will be added to the where clause without any security checks

Examples :

- Admin::get('all') will process a SELECT * statement FROM admin
- Admin::get(1) will process a SELECT * statement FROM admin WHERE id = 1
- Admin::get(array('login' => 'root', 'password => 'root)) will process a SELECT * statement FROM admin WHERE login LIKE root AND password LIKE root

$order is a shortcut to the ORDER BY clause. There's no need to type 'ORDER BY'.

$simplify is a small boolean that tells the code to returns an array even if there's only one record to be returned (usefull for foeach loops).

$admin->save() will insert or update the instance depending on the value of the first field of $fields

$admin->delete() will perform a DELETE statement targeted on the instance

$admin->export() will clone the instance in a new non-typed object.

Session

A safe session manager is already implemented. The session start is automatically called in the process. It has some usefull functions :

Session::token() generates and returns a unique token. It will also be stored as 'token' flash var.

Session::ssid($object = null) returns a javascript string of the ssid.
Keywords are :

- 'serialize' SSID={session_id}
- '&serialize' &SSID={session_id}
- 'input' <input type='hidden' name='SSID' value='{session_id}' />
- null SSID : {session_id}

Otherwise, it will return $object .SSID = {session_id}

Session::get($key) it will work as a getter on the current session

Session::set($key, $value) it will work as a setter on the current session

Session::set($key, $value) it will work as a setter on the current session

Session::flash($key, $value = null, $ttl = 0) it will work as a one time getter/setter on the current session
If 1 argument is passed, it will act as a getter. Else, it will act as a setter. You can set the data freely, but can get it only once.
It will be destroyed after. If a $ttl is set, it will act as an expiry for the data.

Form validation

There are some validators embedded in the framework. These can be usefull to provide safe data validators.

You can call the global 'validate' function to return a new Validator instance and check it with the check($policy) function

Example :
validate($email, true)->check(Validator::$EMAIL)

The boolean tells the validation process if the field is mandatory or not.
The current policies available are :

- $NOT_EMPTY
- $ALPHANUMERIC
- $NUMERIC
- $ENCODABLE
- $ZIPCODE
- $EMAIL
- $DATE
- $DATETIME

You can create your own policies with passing a special array in the check function :
array('name' => 'mypolicy', 'regex' => '#^myregex+)$#i', 'filters' => array(FILTER_SANITIZE_STRING));