Simple MVC Framework - Developer guide

Yoann Mikami

January 1, 2013

Using the framework

This package contains the files needed to build a web application using an MVC[1] oriented architecture.

This document aims at providing a guide on how to build a new web application, or maintain, understand existing ones.

Please refer to ➢*Install guide* for instructions on how to create the initial development workspace. A plain text version, *INSTALL*, is also available.

Contents

[1] (Model-View-Controller)

1 Getting started

1.1 Features

The purpose of this MVC framework is to stay as simple as it can possibly be. Thus, some advanced features that usual MVC framework provides are not available. However, it should have enough materials to build a simple web application, and also support Ajax requests. Below is an overview of the features the current framework provides :

- PHP 5 object-oriented MVC architecture
- Template-based layout using Savant3 (http://phpsavant.com)
  Support of view helpers and view filters
- i18n[2] using gettext
- Single-point of entry provided by a bootstrap, request dispatch through a front dispatcher
- Logical separation (namespace) of applications sharing physical resources
- Hierarchy-based views. Built-in breadcrumbs, navigation menu builder
- Logical mapping of URIs to *all* physical site resources (controllers, views, images...)
- Separation of static and dynamic resources to allow easy deployment and maintenance
- CSS and javascript aggregation : all CSS and javascript served as a single HTTP request
- init plugins, view plugins, resource plugins
- Configuration file in YAML format
- Built-in support for MySQL and MongoDB databases

1.2 Documentation coverage

Each class' phpdoc should give much more details on how to use them, but this document aims at providing generic general guidelines on how to :

- Overview on how this framework handles requests. See page 5).
- Create models, and how to use them. See page 16.
- Create a new controller. See page 20.
- Create a view, assign it a layout and store specific metadata. See page 29.
- Create view helpers, view filters (Savant3 template engine feature). See page 25.
- Create init plugins, view plugins. See page 31.
- Handle various resources. See page 36.

[2] Internationalization

2 Framework basics

2.1 Framework structure

The framework should have the following structure :

- LIB_DIR (current directory).
  - config-default sample configuration file. Configuration is detailed in the Configuration appendix.
  - controllers location for controllers used by the framework.
  - doc The documentation in LaTeXformat.
  - docroot Contains a sample bootstrap PHP file (see ➢*Install guide*).
  - INSTALL ➢*Installation instructions* (plain text) : should be read first.
  - layouts location for template files.
  - libs core PHP classes.
    - [*see below*]
  - opt 3rd party dependencies packages.
  - README.md The Markdown file generated from the LaTeX documentation.
  - views location for view files used by the framework.

ⓘ

- The structure is very similar to the web application structure described in the ➢*Install guide* document.
- Framework currently supplies the following controllers :
  - `stylesheets` (manages stylesheets to be loaded in web pages)
  - `scripts` (manages scripts/files to be included in web pages)

  As well as their associated views in the *views* folder.
- There is currently a single layout provided : *nil.phtml*: only renders the view content. It is used by both stylesheets and scripts views.
- *libs* contains the core files of the framework. All the framework features are provided by the classes defined in this folder.

2.2 Library structure

The *libs* folder has the following structure (some files are omitted) :

- libs
  - interface Iface_XXX interfaces.
  - mvc Default implementation of MVC interfaces.
  - plugins init and view plugins.
    - Init Plugin_Init_XXX plugins : plugins executed before request is processed.
    - View Plugin_View_XXX plugins : plugins executed before view is rendered.
    - Resource Plugin_Resource_XXX plugins : plugins executed before a resource (image, file...) is output.
  - savant3
    - Savant3

- resources Savant3 resources : Savant3_Filter_XXX filters and Savant3_Plugin_XXX plugins (available during view rendering).
- utils Various utility classes.
- Init.php Application single entry-point.
- MVC.php Abstract factory to manage MVC core classes instances.
- Utils.php Handy functions not available in PHP.

2.3 Framework request dispatching process

This section will assume that the application is a web application accessed through a web browser.
It is also assumed that it has two namespaces : front and admin.
The application is accessed through *www.mysite.com* for the front side, and *admin.mysite.com* for the admin side.

The request dispatching process will be explained assuming the above. Application setup follows folder organization described in ➢*Install guide*. It is split into four subfolders under the webroot (e.g. httpdocs) :

- mysite_docroot
- mysite_lib
- mysite_resources
- mysite_site

mysite_lib contains this framework. *mysite_site* contains all the web application files. *mysite_resources* has *logs*, *cache*, *upload* resources folders. Finally, *mysite_docroot* contains the two bootstraps for both namespaces. If apache configuration is available, then the VirtualHost configuration can set the `docroot` for both *www.mysite.com* and *admin.mysite.com* to mysite_docroot . It will contain both bootstrap files and *.htaccess*. On shared servers where server configuration is limited, an *.htaccess* file can be placed under the webroot; its content would be similar, bootstrap files location excluded.

2.3.1 Apache virtual host / .htaccess URL rewriting

Apache is responsible for redirecting the incoming request based on the URL to the proper bootstrap. Here, the front top page is requested so configuration in the VirtualHost section of Apache config, or the *.htaccess* in the docroot under the "*site root dir*" do the trick. *(This documentation leaves out the details on how to configure VirtualHost for Apache).* If *.htaccess* is located under the "*site root dir*", it will look like the following :

Listing 1: .htaccess sample using mod_rewrite

```
RewriteEngine On

RewriteCond %{SERVER_NAME} ^www\.mysite\.com
RewriteCond %{REQUEST_URI} !www\.mysite\.com\.php
RewriteRule ^(.*)$ www.mysite.com.php/$1 [L]

RewriteCond %{SERVER_NAME} ^admin\.mysite\.com
```

```
RewriteCond %{REQUEST_URI} !admin\.mysite\.com\.php
RewriteRule ^(.*)$ admin.mysite.com.php/$1 [L]
```

Should the *.htaccess* be located under webroot instead, the lines

```
RewriteRule ^(.*)$ www.mysite.com.php/$1 [L]

RewriteRule ^(.*)$ admin.mysite.com.php/$1 [L]
```

would become

```
RewriteRule ^(.*)$ mysite\_docroot/www.mysite.com.php/$1 [L]

RewriteRule ^(.*)$ mysite\_docroot/admin.mysite.com.php/$1 [L]
```

The key is to use mod_rewrite so that any incoming request on server name *www.mysite.com|admin.mysite.com* should be redirected to their respective PHP bootstrap file. The initial request URI is appended to it so the URI */foo/bar/* would be rewritten as resp. *www.mysite.com.php/foo/bar* and *admin.mysite.com.php/foo/bar*.

ⓘ bootstrap filenames are arbitrary : they could be named whatever they want. However, should it be different than the SERVER_NAME value, the constant NAMESPACE will need to be defined with a value matching the base URL. The constant can be defined either in the bootstrap file, or in a PreRun plugin (See Pre-run object). Anyhow, it should be defined when `Init::run()` is called.

2.3.2 bootstrapping

The bootstrap file is a small script that merely calls the framework initialization class. It also defines some required constants assumed to be valid in the framework. The first two mandatory constants are DATA_DIR and LIB_DIR: the former should be the server absolute path to the "*site root dir*" while the latter should be the absolute path to the "*framework root dir*". The script should end by including *LIB_DIR/libs/Init.php* and executing `Init::main()`.

In this case, *www.mysite.com.php* will have the following :

Listing 2: Bootstrap code sample

```
define ('DATA_DIR', realpath($_SERVER['DOCUMENT_ROOT'] . '../mysite_site/') . '/');
define ('LIB_DIR' , realpath($_SERVER['DOCUMENT_ROOT'] . '/../mysite_lib/') . '/');
define ('BOOTSTRAP_FILE', basename(__FILE__)); // contains www.mysite.com.php
include (LIB_DIR . 'libs/Init.php');
Init::main();
```

If DOCUMENT_ROOT points to mysite_docroot; otherwise, ../mysite_site and ../mysite_lib should be replaced with /mysite_site and mysite_lib.

The third required constant, namely BOOTSTRAP_FILE, should contain the filename of the script used as a bootstrap file. the PHP magic constants ___FILE___ contains this the full path to the current filename, so it can be used for that matter like above, using **basename**.

Optionally, NAMESPACE can be supplied, which will define the default value assigned to the application namespace during the initialization. (See below).

When not user-defined, the NAMESPACE defaults to the URI server name (`$_SERVER['SERVER_NAME']` value). If the namespace name differs from the latter, it should be defined by the user before calling the bootstrap init.

The following line would be needed if while the server name is *www.mysite.com.php*, the namespace used in the config was mysite_ns :

```
define ('NAMESPACE', 'mysite_ns');
```

From now on, the framework takes charge of the request processing and starts initializing itself.
Static call `Init::main()` instantiates a new `Init` class, runs any pre-run object if available, then runs `$init->run()` on itself.

2.3.3 Pre-run object

Before `$init->run()` is called, `Init::main()` calls `$init->preRun();`
This method will look for a specific classname, namely `Plugin_PreRun` in the current realm of the autoloader (check `Init` header documentation for the included paths).
If such a class exists, and it defines the `Iface_PreRun` interface, it is instantiated then its `process()` method is ran.
Any pre-initialization stuff can be done in this class. However, note that no application specific data is available yet, such as configuration, locale...; thus, application config. dependent code should not be placed there. It can be useful to define required constant NAMESPACE, BOOTSTRAP_FILE... based on specific server-side conditions.
For instance, the a third-party library such as WURFL could be used to perform user agent detection of mobile VS PC browsers, and define the NAMESPACE constant accordingly.

If no pre-run action was found, or after it has completed, the actual initialization can be performed.

2.3.4 Run Init

The `Init` class is the entry point for all requests. It performs the following initializations before processing the request :

- Extract request namespace : If not supplied as a NAMESPACE constant, defaults to `$_SERVER['SERVER_NAME']` : In this case, it will be www.mysite.com.
- Starts an internal timer : Can be used to time the request processing time.
- Loads application config : *config.yaml*, located under *DATA_DIR/config*.
  It reads the configuration values assigned to the current namespace, aka www.mysite.com , so *config.yaml* should have a section with the specified name. It should also extend the 'default' namespace.

The namespace defined in the configuration file should be the namespace determined above with . (dot) and - (hyphen) transformed into _ (underbar). This is due to the YAML format which do not accept these two characters as a property key definition.

The configuration for namespace www_mysite_com is the following :
(Refer to the Configuration appendix for configuration values) :

```
default:
  system:
    debug: false
    timezone: Asia/Tokyo
    locales: en_US,ja_JP
    loglevel: ERR
    plugins:
      init:
        - A
        - B
      view:
        - C
        - D
  directories:
    resources:
    skin: skins
    lang: langs
    upload: upload
    log: logs
    cache: cache

www_mysite_com:
  overrides: default
  system:
    debug: true
    locales: ja_JP
    loglevel: DEBUG
```

- Sets debug flag : flag can be turned on/off in the configuration, for each namespace. Outputs additional debug information in error pages (such as exception stack trace).
- Inits resources : from the configuration, all resource folders values are retrieved and set to the `Utils_ResourceLocator` static class : it is responsible to map all application URLs to physical resources on the filesystem.
- Inits timezone : configurable in the configuration file.
- Inits request : init plugins are processed at this stage. See Init plugins chapter, or section below. Doing this early allows plugins to perform preprocessing on locales, translation files... i.e. anything init after this point.
- Inits locale : configurable in the configuration file. Initial locale defaults to whatever the browser settings are.
- Inits translation file : gettext "*curr_locale*".mo file is expected to be found in the folder pointed by the `langs` property in the configuration file.
  It is namespaced, so in this case the class will look for
  mysite_site/langs/www_mysite_com/ja_JP.mo.
- Inits log file : the `log` folder should be writable by the apache user/group (usually www-data or apache).
  The file *mysite_resources/logs/www_mysite_com.log* will be created if such file does not exist yet, or the current one will be updated. All log write requests from that point will now be directed to that file, based on the log level also configured in the configuration file.

If all previous steps succeed, then the request is parsed. Should a step fail, the init script ends with an "Init error" message sent back to the browser.

2.3.5 Run init plugins (init request)

Init plugins are ran in the order they are declared in the configuration file. That is, in this example, plugin A is processed then B.
Init plugins allows an application to plug functionality and/or initialization routines not supplied in the `Init`. This framework supplies 3 init plugins that can be enabled as needed :

mysql Provides a connection to a MySQL database given the connection properties specified in the configuration file. See page 32

mongoDB Provides a connection to a MongoDB database given the connection properties specified in the configuration file. See page 32

session Provides a session object (`Zend_Session`), with an optional session timeout that can be defined in the configuration file. See page 33

An application can provide its own plugins easily;
They need to be located under *<site root dir>/libs/plugins/Init/*. The process to do so is detailed in page 32.

An init plugin is passed the site configuration, so it can defines its own configuration properties to be set in it. It can also set new parameters in the global registry.

2.3.6 Parse request

Request to be dispatched is extracted from the `$_SERVER['PHP_SELF']` value : it should contain the bootstrap filename, previously defined through the constant `BOOTSTRAP_FILE`, followed by a '/' and the request.

In this case `PHP_SELF` value should be/www.mysite.com.php/', as the top page is requested. After extraction, the original request ('/') is retrieved.
An error is shown and the process is ended if the namespace cannot be found in `PHP_SELF` value.

2.3.7 Handle static resources

The process checks whether current request is a static resource. If it is, then it is served using the proper `mimetype` and ends; i.e. if request is a static resource, the front dispatcher is not called and no view will be rendered. See Handling resources.

2.3.8 Request dispatch to controllers

If resource was not a static resource, as it is the case for this example (top page), the actual dispatch processing is delegated to the front dispatcher.

The front dispatcher is a singleton class, namely `FrontDispatcher`, part of the SimpleMVC . It is responsible for mapping a (non static resource) request to a controller class and dispatch the request to it if found.

ⓘ This explicit use of `FrontController` (detailed page 12 makes a heavy dependency between the framework core and the SimpleMVC package. A future version may address this, by making a front dispatcher interface which would be implemented by the former, and add the dispatcher instance retrieval method to the `MVC` class. Refer to Managing MVC classes regarding the MVC package.

For each applicable controller found (if any), the request is passed. The following 4 base cases can happen :

- An internal redirection is requested during the controller process.
- An external redirect is performed within the controller.
- An exception is raised during the controller process.
- The controller ends normally. The view to render should be set.

For the first case, a new dispatch call is made to the front dispatcher again, to parse the designated new request. This process is repeated as long as the controller(s) processed request for internal redirects, potentially leading to *infinite loops*.
For the second case, the controller will never actually return and the redirection will be executed at once. The front dispatcher does not take the reins back.
For the third case, a "404 error view" will be requested for display if exception raised is a `PageNotFoundException`. If `RequestException`, a "request error view" will be rendered. Otherwise, an init error message will be shown to the user (no view rendered). For the last case, the view set within the controller is retrieved and is set for render, if it exists.

Before view is rendered, view plugins are executed. They are passed the view object, and can perform pre-rendering operations.

2.3.9 Run view plugins

View plugins are then ran in the order they are declared in the configuration file. That is, in this example, plugin C is processed then D.
View plugins allows an application to perform pre-rendering routines that uses view metadata . This framework does not supply any view plugins.

An application can provide its own plugins easily;
They need to be located under *<site root dir>/libs/plugins/View/*. The process to do so is detailed in page 34.

2.3.10 Render the view

The last step in the framework request dispatching process is to render the view associated with current request. By default, the view canonical name is the same as the controller's, but the latter can set it if needed.

The request dispatching process is then over.

3 Introduction to the SimpleMVC framework

SimpleMVC is as its name implies, a simple MVC framework used by this framework. Its architecture is simple on purpose, and does not define a complex class hierarchy. Instead, it defines three main interfaces :

- `Iface_Model`
- `Iface_View`
- `Iface_Controller`

and the corresponding class implementations :

- `SimpleMysqlModel` | `SimpleMongoDBModel`
- `SimpleView`
- `SimpleController` | `AjaxController`

There is currently no additional abstraction layer between the database and the models, to keep it simple. Thus, two distinct model classes are supplied for two databases : MySQL and MongoDB. Similarily, `AjaxController` subclasses `SimpleController` and adds support for XMLHTTPRequests.

In addition, `Layout` is the templating engine rendering class, used by `SimpleView` for rendering.
`FrontDispatcher` dispatches the request to the appropriate controller if any, then can render the resulting view.

Finally, the factory class `SimpleMVCFactory` manages all M,V,C classes and glues SimpleMVC to the `MVC` static class.
It implements the `Iface_MVC` interface.

`MVC` is the main class to be used by applications to instantiate models, views, controllers objects.
It implements an abstract factory pattern, currently limited to SimpleMVC support.

3.1 Notes to developers

All classes (not limited to SimpleMVC) use an NVI[3] approach in their API : all public methods are declared `final`, and delegate the process to a `protected` method `doXxx`, `xxx` being the public method name.

When subclassing SimpleMVC classes such as models or controllers, only the `doXXX` methods can thus be overridden/implemented.

[3]Non Virtual Interface

3.2 Managing MVC classes

Any instantiation can be done through the `MVC` static class, like the following :

Listing 3: Retrieving MVC classes

```
$m = MVC::model('/my/model/name/');
$v = MVC::view('/my/view/name/');
$c = MVC::controller('/my/controller/name/');
```

Each parameter is a canonical name to a view, model or controller. This name is mapped by the factory to an actual file/classname based on a set of rules.
These differ for M, V, and C, but they commonly depend on on application namespaces.

View, model or controller files need to be located to whatever the canonical name, once resolved, would point at.
SimpleMVC resolving rules are explained later in this document.

Except for models which are usually used in controllers, views and controllers should rarely be instantiated by the application developer. `FrontDispatcher` is in charge of instantiating controllers and views, while `SimpleView` instances can instantiate other view instances.

3.3 FrontDispatcher

`FrontDispatcher` is a singleton serving as an entry-point to request dispatching. It instantiates controllers whose canonical names match the current request and delegates the dispatching process to each of them.

3.3.1 Canonical name to controller instance mapping

The base directories for controllers are given by `Utils_ResourceLocator::controllerDirs()`. One is returned for each defined namespace, and for both the "*site root dir*" and "*framework root dir*" (the latter does not define namespaces).

Given the application structure defined in Framework request dispatching process section, the following mapping will be performed, in this order, if *admin.mysite.com/foo/bar/* is requested :

Request value is parsed to produce (alphanumeric string, list of values) pair(s) :

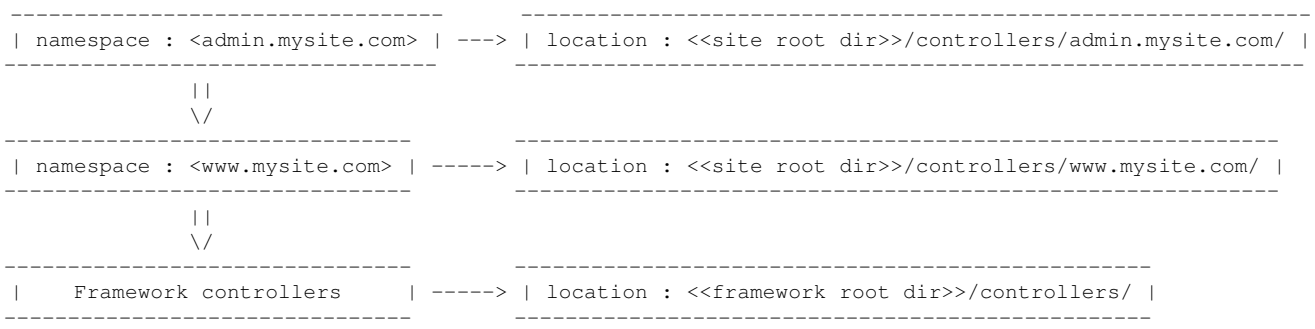
- '/' yields ("", empty list) (i.e. empty string)
- '/foo/' yields ("foo", empty list)
- '/foo/bar' yields ("foobar", empty list) and ("foo", list("bar"))
- ...

ⓘ This first (filtered, stripped of non alnum characters) value becomes both the controller canonical name to look for and the default view canonical name to render at the end of the process.
The second value (the list) can be request parameters when using SEO friendly URLs. More on this in the section (Get the request controller(s)).
Thus, '/foo% /bar/baz' would yield ("foobarbaz", empty list) and ("foo", list("bar", "baz")).

The controller classname is also deducted from the filtered value and becomes :

- N/A (no controller assigned for the empty string)
- `FooController`
- `FoobarController` and `FooController`

The rule is **`ucfirst`**`($filtered).'Controller'`.



> ⓘ '-' characters in the namespace are replaced by '_', so the namespaced folders should be named accordingly.

For each of the folders generated above, generated controller name 'foobar' will be appended :

- *<site root dir>/controllers/admin.mysite.com/foobar/*
- *<site root dir>/controllers/www.mysite.com/foobar/*
- *<framework root dir>/controllers/foobar/*

For each of the folders above, `FoobarController.php` is looked up until such a file is found :

- *<site root dir>/controllers/admin.mysite.com/foobar/FoobarController.php*
- *<site root dir>/controllers/www.mysite.com/foobar/FoobarController.php*
- *<framework root dir>/controllers/foobar/FoobarController.php*

Found file should contain the PHP class definition of `FoobarController`, which should implement `Iface_SimpleController`.
This whole lookup process above (except the request name filtering) is performed through the single call : `MVC::controller('foobar')`.

3.3.2 Get the "default" controller

For every request, the front dispatcher will always look for a "default" (canonical name) controller first. Each namespace can define one "default" keywordcontroller, and if found, will be processed first. Inheritance also applies so if namespace www.mysite.com defines a "default" controller, admin.mysite.com will also use it unless it has its own.
Such controller can be useful to perform operations available to all application pages globally.

The default controller location follows the same rules as the above. Thus the following folders are looked for a `DefaultController` class in this order :

- *<site root dir>/controllers/admin.mysite.com/default/*
- *<site root dir>/controllers/www.mysite.com/default/*
- *<framework root dir>/controllers/default/*

> ⓘ This framework does not provide a default controller, thus the third lookup will always return nothing.

The "default" controller is not mandatory : if not defined in any namespace, the dispatcher will continue anyway.

3.3.3 Get the request controller(s)

The next controller candidate that is looked for is the current request, defining its canonical name ; if found, then it is executed after the "default" if any. The current request controller can be either, or both of the following :

- Full filtered request, no request parameter extracted from URL ⟹ `FoobarController` defined under *<any controller dir>/foobar*.
- First part of filtered request, the rest being treated as request parameters, assuming a so-called SEO-friendly URL.
  ⟹ `FooController` defined under *<any controller dir>/foo*, with the list of parameters `[bar]` stored.

Both, either or none of these controller may be found; in case of none, no controller is assigned to current request, and the dispatch process ends at this point (the view is rendered as is). In case of both, more specific controller (FooBar) is always processed before the other one (Foo). Having several controllers stacked up may be useful in somes cases when a generic action has to be performed in every page and avoid code duplication. When dealing with several namespaces, a controller can be defined for request /a/* (`AController`) in namespace foo then another controller in namespace bar could be defined for the specific request /a/b/ (`ABController`, to add some specific behavior in this particular case.

> ⓘ `ABController` will always be performed first, so it has its limitations (e.g. code in `AController` must be performed *first*). Also, overlapping can be troublesome if internal redirects are set in `ABController` as `AController` would end up not being performed at all! (Similarily with the default controller, which is ran before any other one).

3.3.4 Dispatch requests to controllers

Default view canonical name is first set on the new controller instance, i.e. the filtered request value. Then the `SimpleController` main entry point for dispatching is the function `dispatch` is called : it returns itself upon return (fluent interface).
The view to render, if changed during dispatch, is retrieved using `view()`.
The function `isForward()` tells whether current request should perform an internal redirect; if so, then `FrontDispatcher::dispatchRequest()` is called again with the retrieved view name as the new request. This process is reiterated while internal redirects are set.

If a redirect is requested by current controller, it overrides any controller waiting to be processed after the current one; e.g. "default", then "foo" controllers are pushed into the stack but "default" asks for a redirect. When defining controllers, programmer should be careful with overlapping values and redirections. Some code expected to run in some later controller may not eventually due to some internal redirect!! Refer to note in Get the request controller(s) section.

For each controller instance, a `requestParams` instance variable is set with the extracted (default empty) values from the URL. The controller may then use them to perform what is needed based on those values.

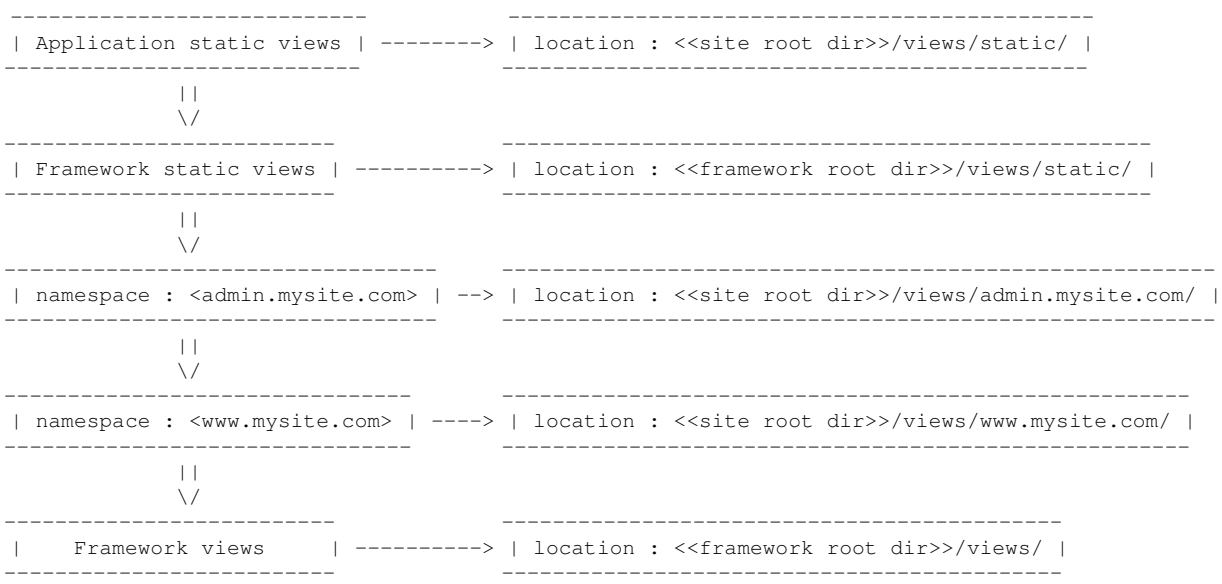
3.3.5 Canonical name to view instance mapping

Once dispatching is over, and before return, the front dispatcher tries to instantiate the view (whose canonical name defaults to the controller canonical name if no controller set it or no controller was processed) related to current request.

The base directories for views are given by `Utils_ResourceLocator::viewDirs()`. One is returned for each defined namespace, and for both the "*site root dir*" and "*framework root dir*" (the latter does not define namespaces).

SimpleMVC implementation also prepends two additional directories before this list, for static view files.
Those static view files should contain views shared across all application namespaces, usually e.g. 404 page not found error page. . . .

The following assumes the controller named 'foobar' above was found and changed the default view name from foobar to barbaz.



> ⓘ '-' characters in the namespace are replaced by '_', so the namespaced folders should be named accordingly.

For each of the folders above, *barbaz.phtml* is looked up until such a file is found :

- *<site root dir>/views/static/barbaz.phtml*
- *<framework root dir>/views/static/barbaz.phtml*
- *<site root dir>/views/admin.mysite.com/barbaz.phtml*
- *<site root dir>/views/www.mysite.com/barbaz.phtml*
- *<framework root dir>/views/barbaz.phtml*

This whole lookup process above is performed through the single call : `MVC::view('barbaz')`.

> ⓘ While a controller not found does not produce an error, should view *barbaz.phtml* be not found in any of the paths above, a `PageNotFoundException` will be raised, which is processed by the function `Init::parseRequest` : the view named '404' will then be rendered. Should that fail again (e.g. no such view available either), a plain text error message will be rendered.

3.4 Managing Models

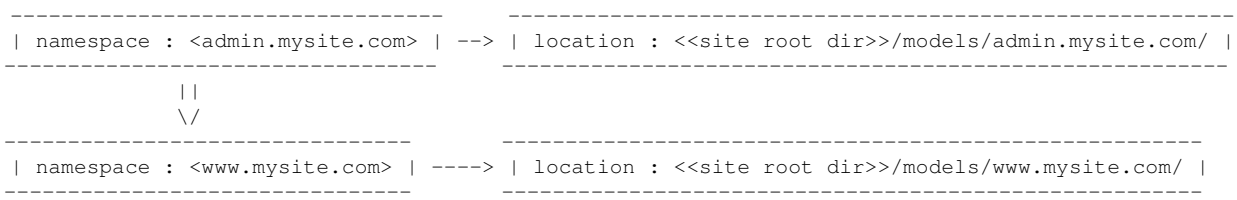
3.4.1 Canonical name to model instance mapping

The base directories for models are given by `Utils_ResourceLocator::modelDirs()`. One is returned for each defined namespace under "*site root dir*".

Contrary to views and controllers, models do not define a hierarchy, so model folders are flat.

Given the application structure defined in Framework request dispatching process section, the following mapping will be performed, in this order, if model named 'foobar' is requested :

The controller classname is deduced from this value and is `Model_Foobar`. The rule is `'Model_'.ucfirst($filtered)`.



> ⓘ '-' characters in the namespace are replaced by '_', so the namespaced folders should be named accordingly.
> The framework does not define models, which is a very application specific concept. Thus, it does not even look under the "*framework root dir*" for models.

For each of the folders above, *Foobar.php* is looked up until such a file is found :

- *<site root dir>/models/admin.mysite.com/Foobar.php*
- *<site root dir>/models/www.mysite.com/Foobar.php*

Note that the "Model_" is not part of the filename: it is auto-appended in the `Init` class by using `Zend_Autoloader`. Found file however should contain the PHP class definition of `Model_Foobar`, which should implement `Iface_SimpleModel`.
The model class can also extend one of the two supplied models implementing `Iface_SimpleModel` in SimpleMVC : `SimpleMysqlModel` or `SimpleMongoDBModel`, if connectivity to either database is required. This whole lookup process above (except the request name filtering) is performed through the single call : `MVC::model('foobar')`.

3.4.2 Creating a model

As stated before, there are two ways to create a new model :

Implement Iface_Model `Iface_Model` defines the basic CRUD[4] methods to be implemented.
Some methods require a valid instance (i.e. `$this` must refer to an actual record in the system) while some can work on any instance of the model object. Such methods return a valid instance.

Model properties may be defined the way the developer wants: getter/setter method, through magic methods (`__get`/`__set`) method, or plain public class variables: the interface does not enforce this.
`SimpleMysqlModel` and `SimpleMongoDBModel` use the latter approach.

The model class should be named "Model_MyModelName"; its canonical name will then be 'mymodelname', and can be instantiated with `MVC::model('mymodelname')`.
The file should be located in one of the model folders as previously detailed.

[4]Create-Read-Update-Delete : the four basic operations on objects

- `insert($data, $options=null)`
  Inserts a new entry using data `$data` and using `$options`. Fluent interface, so should return `$this`.
  Can be called on any `$this` instance; the latter is expected to be mutated by this method so that inserted data are accessible through the object after the call.

- `update($data, $options=null)`
  Updates (`$this`) record with data from `$data` and using `$options`. Fluent interface, so should return `$this`.
  Can be called on *valid* `$this` instances only. `$this` is expected to be mutated by this method so that updated data are accessible through the object after the call.

- `remove($crits=array(), $options=null)`
  Removes record(s) matching criteria defined in `$crits`. Fluent interface, so should return `$this`.
  Can be called on *valid* `$this` instances only.

- `findOne($data=array())`
  Finds 0 or 1 record matching specified data defined in `$data`. Returns false if not found, or a new instance of `$this` for specified record.
  Can be called on any `$this` instance.

- `find($data=array(), $sort=array(), $limit=0, $offset=0)`
  Finds all records matching specified data defined in `$data`.
  Can be optionally sorted using `$sort` criteria.
  A subset of `$limit` records starting at `$offset` can be returned.
  Returns empty array if nothing found, or an array of model instances of matching records.
  Can be called on any `$this` instance.

- **`count`**`($data=array())`
  Returns the number of records matching specified data defined in `$data`.
  The number is the number of records that would be returned by find without `$limit` or `$offset` set (unfiltered).

Extend SimpleMysqlModel | SimpleMongoDBModel

ⓘ To use Mysql|MongoDB models, the init plugin 'mysql'|'mongodb' must be enabled in the configuration file, and the relevant configuration properties for either database set properly.

Unique ID Both model implementations have a unique ID named `_id` : generally `AUTO_INCREMENT` for Mysql and `UID` for MongoDB. By default, the function `getUniqueIds()` returns an array with a single element `_id` in it. Should a model have another way to define uniqueness, it can override `doGetUniqueIds`. This function is internally used by the `update` method to apply update data on the current record only.
For MongoDB model, `_id` is mandatory and will thus always be non empty. On the other hand, Mysql model may have empty `_id`.

Defining properties Both model implementations adopt the public class variable approach to define model properties : `_id` is the only built-in such variable already defined for all models. All other properties should be defined through a set of `public $my_prop`.

> ⓘ Reflection is used internally to extract properties from the class. Thus, only *properties* should be public!

Hydrating data Hydration occurs before/after CRUD operations. It's a process that allows to format/convert data passed as a parameter when necessary, or set some data for system defined values (e.g. creation date...).

The following are defined and can be overridden for MongoDB models :

before insert/update : `doHydrateInsertData($data)`**:array**

before findOne/find : `doHydrateFindCriteria($data)`**:array**

after insert/update/findOne/find : `doHydrateMapData($data)`**:array**

In addition, the related public methods (hydrateData, hydrateFindCriteria) perform some additional formatting on the `_id` field :

- Before `insert`, _id is converted from a string to a `MongoId` object
- Before `find`/`findOne`, _id is converted if it is a string to a `MongoId` object

The following are defined and can be overridden for Mysql models :

before insert : `doHydrateInsertData($data)`**:array**

before update : `doHydrateUpdateData($data)`**:array**

before findOne/find : `doHydrateFindCriteria($data)`**:array**

after insert/update/findOne/find : `doHydrateMapData($data)`**:array**

> ⓘ `doHydrateFindData` creates bound parameters for query. Therefore, it *must be called last using `parent::doHydrateFindCriteria` if overridden!*

Registry objects MongoDB init plugin registers two objects in the registry : 'mongo_db' (used by the associated model) and 'mongo_db_admin' (not used but available to the developer). The former is a connection to the DB name defined in the configuration file, while the latter is a global object that allows to do admin operations for MongoDB.

Mysql init plugin registers a single object in the registry that is used by the associated model : 'mysql_db'.

Initialization MongoDB model defines two init methods : `dbInit($db)` and `init()`, called when a new instance is created. The former allows to set global properties on the given `$db` object (e.g. `ensureIndex`), while the latter allows to perform other initializations (e.g. assign default values to properties...).

Mysql model defines the `init()` method, called when a new instance is created. It can be used to perform various initializations (e.g. assign default values to properties...).

Define the target collection/table MongoDB model defines the abstract method `getDbSection()`, which shall return the collection name current model instance is operating on.

Mysql model defines the abstract method `getTable()`, which shall return the table name current model instance is operating on.

3.5 Managing controllers

3.5.1 Canonical name to controller instance mapping

Refer to Canonical name to controller instance mapping regarding the mapping rules.

3.5.2 Creating a controller

The SimpleMVC framework supplies two base controller classes that the developer can extend :

- SimpleController, for basic GET/POST support (No Ajax)
- AjaxController, which adds XMLHttpRequest (Ajax) requests to the above. Response are sent in JSON format, and the dispatch method does not return (script ends after JSON data are sent back).

The class should be named "MycontrollerController" (only the first letter is uppercased, this is not Camel case!), which would then become instantiable through its canonical name 'mycontroller' and instantiated with `MVC::controller('mycontroller')`.

It is also possible to create a new controller from scratch by implementing the `Iface_Controller` or the `Iface_AjaxController` interface.

Implement Iface_Controller | Iface_AjaxController The methods to define are :

- `dispatch()`
  Dispatches the request and performs associated action. Fluent interface, so should return `$this`.
  Upon return, the `view()` method should return the view canonical name to render, `isForward()` method should return true if internal redirect is needed, and perform any external redirect needed without returning.
  Should not return when implementing an ajax controller.
- `redirect($to)`
  Performs an external redirect to specified URL. This method does not return.

- `isForward()`
  Returns true if current controller process requests an internal redirect.
- `getHeader($header)`
  Returns the specified HTTP header value.
- `getRequestMethod()`
  Returns 'GET', 'POST'... for current request.
- `getPost()`
  Returns the POST data as an array (may be sanitized in the process).
- `getParams()`
  Returns the GET data as an array (may be sanitized in the process).
- `isPost()`
  Returns true when current request method is POST.
- `isGet()`
  Returns true when current request method is GET.
- `setView($view)`
  Sets the view canonical name to render when this controller ends its process.
- `view()`
  Returns currently set view canonical name.
- `isXmlHttpRequest()` [AjaxController only]
  Returns true when current request method is an Ajax request.
- `getXmlHttpResponse()` [AjaxController only]
  Returns the JSON response to send back to the client.

Extends SimpleController | AjaxController

Implementing get/post/ajax dispatchers

Main function `dispatch` called by `FrontController` further dispatches the request based on the request method; currently, GET and POST are supported with `doGetDispatch` and `doPostDispatch` methods.
For ajax requests, if controller is a subclass of `AjaxController`, a special `doAjaxDispatch` method is also provided.

Default implementation for GET and POST dispatches raise a `RequestException` (invalid request).

Controllers should override any of these methods to prevent the exception from being raised. When using `AjaxController`, `doAjaxDispatch` will be called regardless of GET/POST request method. Should an error occur during the request process, an exception should be raised; `RequestException` for request related errors, or other exceptions for system related errors. . . .

Each controller may access the `requestParams` instance variable, which is an array (list) of parameters extracted from the URL, if one is using SEO-friendly URLs.

Passing request data to views Controllers can store data which are meant to be available to the target view.
This can be done either through direct assignment on the controller instance, e.g. `$this->myvar = 'foo'`, or through the `addRequestData($key, $value` method.
Similarily, those data can be retrieved from within the controller through the direct accessor syntax `if ($this->myvar === 'foo')`, or through the `getRequestData($key=null)` method. The latter returns all (key,value) pairs if its $key argument is null.

When in the view, all data stored this way are accessible through the direct accessor syntax, like view metadata.
`<?php if ($this->myvar === 'foo'): echo $this->myvar; endif; ?>`.

Important : Some keys have special meaning as views can't be used as request data keys (this will cause the view to fail during rendering). These values are :

- content
- filters
- layout
- locale
- location
- order
- title
- view

ⓘ Example shown use a string as the value, but any datatype can be stored: objects, arrays...

About POST/Redirect/GET pattern Post/Redirect/Get is a design pattern which helps avoiding duplicate form submission and allow user agents to behave more intuitively with bookmarks and the refresh button. Basically, each POST request generates a redirect (303) to the same page after it finishes processing the request.

The `SimpleController` has built-in support for this design; to enable it, the subclass controller should call `setRedirectOnPost(true)` in the constructor. This behavior is not enabled by default, so without the flag set to true, the POST is performed normally.
In the case where it is enabled, `dispatch` method takes care of it by detecting the request type. When a POST request is detected, the following is performed :

- Delegates the process to `doDispatch` (`doPostDispatch` will be called at this point)
- Creates a new temporary session namespace associated with that controller
- Stores into the newly created session all (key,value) pairs stored as request data, returned by `getRequestData`. This is necessary as a 303 redirect will generate a new request, and thus all request data will be lost in the next request. Only session data are persisted.
- Perform a 303 redirect to specified view (it must be handled by the same controller or the session namespace will not match!)
- (New request) Checks whether request is a GET and has a valid namespace session named after the controller
- If true, then gets all (key,value) pairs previously stored in this namespace session and put them back into the controller as request data using `addRequestData`
- Unset the namespace session

ⓘ This process relies on sessions, therefore `Zend_Session` is used, regardless of whether the session plugin is active or not. Moreover, the latter can be used without overlapping because namespaces are different.

3.6 Managing layouts

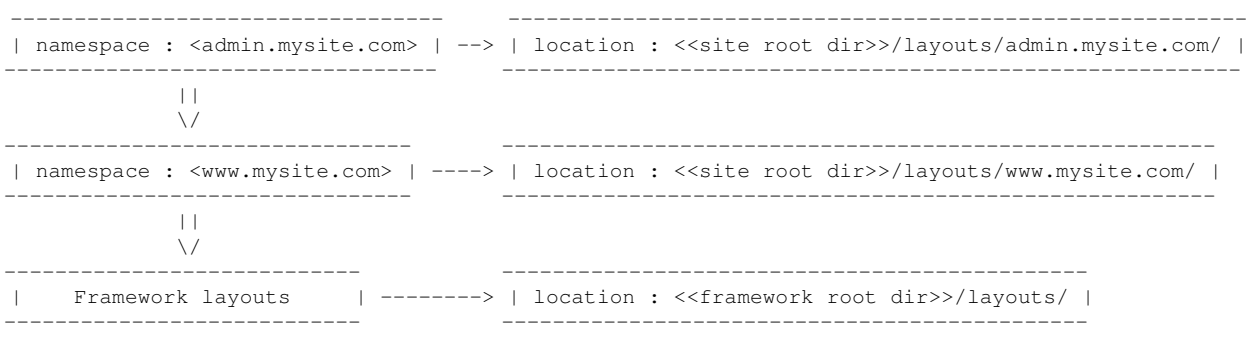
3.6.1 About layouts

A layout in this framework is a wrapper around the Savant3 templating engine; it is comosed of the `Layout` class, and provide additional features to `Savant3`, such as coupling with `SimpleView` objects. Thus, layouts and views are tighly coupled together. Both are phtml files. They are located inside one of the directories returned by `Utils_ResourceLocator::layoutDirs()`.
The framework provides a single layout : *nil.phtml*, which corresponds to the empty template : only the view content is displayed.

3.6.2 Layouts location

The base directories for layouts are given by `Utils_ResourceLocator::layoutDirs()`. One is returned for each defined namespace, and for both the "*site root dir*" and "*framework root dir*" (the latter does not define namespaces).

Layouts are instantiated by view instances. The mandatory '`layout`' metadata in the view header (Refer to View header) has the name of the layout to wrap the view with. Given the application structure defined in Framework request dispatching process section, the following mapping will be performed, in this order, if layout metadata value is 'top' :



> ⓘ '-' characters in the namespace are replaced by '_', so the namespaced folders should be named accordingly.

For each of the folders generated above, layout filename top.phtml will be appended :

- *<site root dir>/layouts/admin.mysite.com/top.phtml*
- *<site root dir>/layouts/www.mysite.com/top.phtml/*
- *<framework root dir>/layouts/top.phtml/*

Found file shall contain the whole content to be displayed. Everytime a `SimpleView` is instantiated, a new `Layout` object is also created. Calling `$view->render()` actually delegates the rendering process to its underlying layout. If the latter includes the view content into its own, the PHP code embedded into the view will be ran *in the context of the layout*, i.e. `$this` will refer to the `Layout` instance inside the view code. It also means that `$this->view` is the view object itself inside a view phtml.

3.6.3 Writing the layout content

Savant3 template plugins can be used inside the layout phtml code.

The view metadata defined in its header can be accessed through the `view` property : `$this->view`, which is always set . The snippet below shows a very basic example of a layout :

Listing 4: Simple layout sample

```
<!doctype html>
<html>
  <div id="header">
    <?php echo $this->header(); ?>
  </div>
<?php if (isset($this->view->isRoot())): ?>
  <div id="top-banner">
    <?php echo $this->xl8('Welcome to my site!'); ?>
  </div>
<?php endif; ?>
  <div id="main">
    <span class="title"><?php echo $this->title(); ?>
    <?php echo $this->content; ?>
  </div>
  <div id="footer">
    <?php echo $this->footer(); ?>
  </div>
</html>
```

The simple listing above shows how template plugins are called (`$this->header()`, `$this->footer()`, `$this->xl8()` and `$this->title()`) : each returns some content. It is thus possible to create different layouts and still use common parts through such plugins.

> ⓘ `xl8` and `title` are plugins supplied with the framework. See Built-in template plugins for the list.

The `view` property is also used in a conditional code to show a banner when the current layout is displayed for the root view (top of hierarchy).
The property `content` is always set in `Layout` instances, and contains the *interpreted* resulting output from the view.

The view `content` is interpreted *before* the layout is rendered. This allows the view to modify the environment prior to rendering, such as adding new CSS/javascript to the page. Echoing it in the layout is embedding the view into it. If the content is never echoed in the file, the view content will never be shown. It is also possible to embed the view content multiple times.

3.6.4 Template filters and plugins

Template filters and plugins are features provided by the Savant3 templating engine and reused as is.

Filters are ran after the rendering on the generated output to perform post-processing on the text, in the specified order, and returned output becomes the input for the next filter until they are all processed.
Plugins are components that can be ran inside the layout code (or the view code, as it is ran in the same context), which can do various things such as returning pieces of code. As opposed to the ones found in the layout content, template plugins inside views are processed before the layout is rendered, as stated above.

Specify global filters Global filters are applied to all layouts or views. New global filters can be added using `Layout::addGlobalFilters(array())`.
Parameter is an array of filter names to priorities.

Specify individual view filters View filters are defined in the view header and apply when that view is rendered only. No priority can be defined for view filters: thus, they all have the default priority, i.e. 50.

View Filters is referenced through the "filters" YAML property, and is an array of filter names. See next paragraph for a sample.

Filter Priority Filters can be assigned a priority from 0 to 100, default being 50. Before filters are processed, they are sorted by descending priority (100 is highest, 0 is lowest). Filters specified in the View header always have the default priority. The order view filters will be ran is the order they were declared. All global filters with a priority of 50 will be ran before view filters.[5]

The following graph summarizes priorities; `[G]` means global filter, `[V]` means view filter.

[5] Global and view filters are the same classes, only the declaration is different

Listing 5: "Set global and view filters"

```
Layout::addGlobalFilters(array('filter1'=>100, 'filter2'=>80, 'filter4'=>50,
                 'filter3'=>50, 'filter7'=>20, 'filter8'=>0));

// .... in the view header .... //
/*
---
filters:
  - filter6
  - filter5
---
*/
```

```
------------------------------
| [G] Filter1 priority : 100 |
------------------------------
               ||
               \/
------------------------------
| [G] Filter2 priority : 80 |
------------------------------
               ||
               \/
------------------------------
| [G] Filter4 priority : 50 |
------------------------------
               ||
               \/
------------------------------
| [G] Filter3 priority : 50 |
------------------------------
               ||
               \/
------------------------------
| [V] Filter6 priority : 50 |
------------------------------
               ||
               \/
------------------------------
| [V] Filter5 priority : 50 |
------------------------------
               ||
               \/
------------------------------
| [G] Filter7 priority : 20 |
------------------------------
               ||
               \/
---------------------------
| [G] Filter8 priority : 0 |
---------------------------
```

Filter / Plugin locations `Layout` sets the filter and plugin locations internally; the path is the same as the one used in Savant3.

This framework supplies some filters and plugins as well, so they can be found at two locations :

- framework : "*framework root dir*"/libs/savant3/Savant3/resources/
- site : "*site root dir*"/libs/savant3/Savant3/resources/

Any filter/plugin saved in either folder will be automatically become available to layouts and views.

Creating a new filter Savant3 template filters should subclass `Savant3_Filter`. The new filter classname is also expected to be of the form `Savant3_Filter_Foo`, where foo will be the filter name used to specify global/view filters, as seen above.

Filters need to implement a single static method : `static public function filter( $buffer):string`. Input is the rendered layout+view content, and the output should be the filtered input.
Filters may not modify the input at all (i.e. return it as is), but rather perform some post-process after the rendering.

The following snippet shows the "SiteBase" filter supplied in the framework, which rewrites all absolute HTML links such as href (A elements), action (FORM elements), src (IMG elements) by prepending the site base string when defined (i.e. site top is not accessible through the docroot, e.g. located at *http://www.myhost.com/mysite/* instead of *http://www.mysite.com/*). File is *Savant3_Filter_SiteBase.php*.

Listing 6: "siteBase filter"

```
final class Savant3_Filter_SiteBase extends Savant3_Filter
{
    static function filter($buffer)
    {
        $buffer = preg_replace('!(href|action|src)=(["\'])/([^"\']*)(["\'])!',
                 '$1=$2'.Utils_Url::getSiteBase().'$3$4', $buffer);
        $buffer = preg_replace('!(["\'])/(skin|upload|script)/!',
                 '$1'.Utils_Url::getSiteBase().'$2/', $buffer);
        return $buffer;
    }
}
```

This filter can be added to the global/view filters by specifying "siteBase" as the filter name.

Creating a new plugin Savant3 template plugins should subclass `Savant3_Plugin`. The new plugin classname is also expected to be of the form `Savant3_Plugin_Foo`, where foo will be the plugin name referenced in layouts/views with the code `$this->foo()`.

As opposed to filters, which are internally ran by the Savant3 engine and thus have a fixed method prototype, plugins are called by the template/view developer and can therefore have any parameter needed.

Plugins need to implement a single method : `public function xxx(...)`, where xxx is the plugin name, which induces its classname as well. `Savant3_Plugin_FooBar` needs to have a public method called `fooBar` to be recognized by the engine. Parameters and return value is not fixed: some plugins may return a string or some content while others may buffer the output and display it directly, and thus return nothing. The Savant3 `eprint` is such plugin.

The following snippet shows the "xl8" template plugin, which gets a gettext translation in the current locale for specified text :

Listing 7: "Code for the Xl8 template plugin"

```
final class Savant3_Plugin_xl8 extends Savant3_Plugin
{
```

```
    public function xl8($str, $escape=true)
    {
        static $xl8 = null;
        if (is_null($xl8)) {
            $xl8 = Zend_Registry::get('Zend_Translate');
        }
        $str = $xl8->_($str);
        if ($escape) {
            $str = $this->Savant->escape($str);
        }
        return $str;
    }
}
```

Built-in template filters This framework provides the following template filters :

compress : compresses CSS content (remove comments, strip whitespaces/newlines...)

siteBase : append subdirectory to absolute urls defined in href, action, src... if site top page is not under the docroot

Built-in template plugins This framework provides the following template plugins :

breadcrumbs : Builds an array of current view ancestors, from current to root. Can be used to display breadcrumbs

menu : Returns the root view object and all its descendents whose `location` metadata is set to specified value.
Menus can be built using this plugin : i.e. all views that must appear in the same menu may have the same location value. Those that don't will not be in the child list.

partial : Interpret PHP code from input string, and return the result. Similar to **`include`**, only for string instead of files.

scripts : Processes all registered javascript files/scripts, aggregate all files into one and return HTML content which load all javascript from the page. See Managing stylesheets and scripts.

selfUrl : Returns an absolute URL to current page.

stylesheets : Aggregates all registeed CSS and return HTML content which load all CSS from the page. See Managing stylesheets and scripts.

timer : Stops the timer started by `Init::main()` and returns an HTML comments enclosing the timer elapsed time.

title : Returns the view localized title/subtitle. Expects metadata "title" (possibly i18n formatted) in the view header. If "subtitle" is set as well, it is appended to the title. The result is escaped and returned.

tzTime : Returns specified timestamp into a localized time using current timezone (from the configuration file).

xl8 : Translates specified string to current locale using gettext. - breadcrumbs - menu

3.7 Managing views

A view is a phtml[6] file located under the views directories. Views are hierarchized based on their relative path from the views root directory. PHP code can be embedded directly into the view (using `<?php ... ?>`), or through template plugins (See page 25). This hierarchy allows to build a site map, breadcrumbs, or navigation menus. The framework provides template plugins to generate those automatically.

3.7.1 Canonical name to view instance mapping

Refer to Canonical name to view instance mapping regarding the mapping rules.

3.7.2 View hierarchy

`SimpleView` instances have 0 or 1 parent and 0 or more children. Parent is null if the view instance refers to the top of the hierarchy, while children are empty if it is a leaf.

Given the application structure defined in Framework request dispatching process section, below is a view directory structures sample alongwith the associated view hierarchy :

- *"site root dir"*
  - views Views root dir.
    - www.mysite.com namespace.
      - index.phtml Root. cname: /. parent: null. children: [/foo/,/baz/]
      - foo foo subtree
        - index.phtml cname: /foo/. parent: /. children: [/foo/bar/,/foo/baz/]
        - bar.phtml cname: /foo/bar/. parent: /foo/. children: []
        - baz baz subtree
          - index.phtml cname: /foo/baz/. parent: /foo/. children: [/foo/baz/aaa/]
          - aaa.phtml cname: /foo/baz/aaa/. parent: /foo/baz/. children: []
      - baz.phtml cname: /baz/. parent: /. children: []
    - admin.mysite.com namespace
      - baz.phtml cname: /baz/. parent: /. children: []
      - foo foo subtree
        - baz baz subtree
          - index.phtml cname: /foo/baz/. parent: /foo/. children: [/foo/baz/aaa/]
          - aaa.phtml cname: /foo/baz/aaa/. parent: /foo/baz/. children: []
    - static static views
      - 404.phtml cname: /404/. parent: /. children: []
      - rerror.phtml cname: /rerror/. parent: /. children: []

Each level is expected to have at least an *index.phtml*. Any other file located at the same level as the latter will become its children.

View canonical name is the file hierarchy minus the `.phtml` extension; e.g. *.../foo/bar.phtml* $\Longrightarrow$ /foo/bar/.

However, for *index.phtml* files, the whole filename is dropped; e.g. *.../foo/index.phtml* $\Longrightarrow$ /foo/.

[6] HTML with embedded PHP code

As a consequence, view canonical names to view filename is not a 1-1 mapping : e.g. /foo/ can refer to either *. . . /foo.phtml*, or *. . . /foo/index.phtml*.

3.7.3 View header

Every view has a header in YAML format. YAML is a human-readable format; data structure hierarchy is achieved through outline indentation :

Listing 8: "YAML format sample"

```
---
# Comment
property: value  # Scalar
list:      # List of elements
 - one
 - two
 - three
array:       # Associative array
 sub1: value   # Scalar
 subarr:     # Nested array
  sub2: value
  sub3:
   - four
   - five
---
```

A view file needs to have the header at the top of the file. Header is delimited using three hyphens '-'. The view content starts after the closing hyphens triplet.

Currently, the framework requires the following properties to be set in every header :

layout The layout name to render the view into. See Layouts location.

Additionally, the framework recognizes the following properties if set :

title Defines the title of the current view. Can be displayed using the title savant plugin. Can be internationalized (See i18n in YAML files/view headers).

filters A *list* of template filters to apply on the view. See Template filters and plugins.

order A number defining the position of current view among its siblings. Used by menu and breadcrumbs template plugins, `SimpleView::children()` to order view children. *index.phtml* located at the same depth will be compared to each other, while non *index.phtml* files located within a given subdirectory will be compared to each other.

location An arbitrary value that the developer can use with the menu template plugin: only views whose `location` property is set and match specified value will be returned when getting view descendents.

3.7.4 Creating a view file

- Create the .phtml file under the desired location : static file => "*site root dir*" static folder. other => "*site root dir*" views folder.
- Add a YAML header and the required "`layout`" property. (Other properties may be added as needed).

- The view content is everything below the header. All values previously stored in the global registry are available. It is a way for controllers to communicate with the upcoming view, by storing data to the registry that they can later retrieve.

3.7.5 View object

Though the developer manages views as phtml files, `SimpleView` implementing the `Iface_View` interface, wraps the file and takes care of extracting header info, content, and provide some utility methods, including `parent()` and `children()`. For the complete API, refer to the PHPdoc.

A static utility class for views called `Utils_Views` also provides some functions that handles view objects. It is mainly used internally, but they can also be useful in some cases. The PHPdoc provide detailed explanations on their usage.

4 Plugins

This framework currently has the following types of plugins :

- Init plugins, called during the library initialization and before request is parsed by `FrontController`
- View plugins, called after controller dispatch and before view is rendered.
- Resource plugins, called for each site resource (skin, scripts and upload).

Plugins can be enabled on a namespace basis using the configuration file; The format is :

Listing 9: "Config format for plugins"

```
<namespace>:
  system:
    plugins:
      init:   # list of init plugins to enable
        - myplug1
        - myplug2
      view:   # list of view plugins to enable
        - myplug1
        - myplug2
      resources: # list of resource plugins to enable
        image:
          - myplug1
          - myplug2
```

Init plugins can be located either in
<site root dir>/libs/plugins/Init or *<framework root dir>/libs/plugins/Init*.

View plugins can be located either in
<site root dir>/libs/plugins/View or *<framework root dir>/libs/plugins/View*.

Resource plugins can be located either in
<site root dir>/libs/plugins/Resource or *<framework root dir>/libs/plugins/Resource*.

4.1 Init plugins

Init plugins shall implement the `Iface_Plugin_Init` interface. The classname should be `Plugin_Init_Foo`, where Foo becomes the init plugin name used in the configuration file (e.g. myplug1/myplug2 above).

There is a single method to implement for these plugins :

```
public function process(Utils_Config $config, $namespace, $request): void:
```

- **$config** : The configuration object. Init plugins may require their own configuration values, thus they can retrieve them through this object.
- **$namespace** : The current request namespace : the configuration object handles namespace resolution automatically, but plugins may want to know which namespace is currently active.
- **$request** : The current request : should the init plugin need to do something based on the request.

4.1.1 Built-in init plugins

This framework supplies the following init plugins :

Mysql init plugin

Purpose Establish a connection to a Mysql database.

Configuration The following properties are needed in the configuration file, and their default values if not supplied :

Listing 10: "Mysql config"

```
<namespace>:
  mysql:
    host: <string>|'localhost' # The database host to connect to
    name: <string>|'default'   # The database name
    user: <string>|''          # The database username to connect with
    pass: <string>|''          # The password for supplied username, if any
```

Registry values The registry property mysql_db, associated with the Mysql DB connection object, is stored in the registry.
This init plugin is required if the application models subclass `SimpleMysqlModel`.

MongoDB init plugin

Purpose Establish a connection to a MongoDB database.

Configuration The following properties are needed in the configuration file, and their default values if not supplied :

Listing 11: "MongoDB config"

```
<namespace>:
  mongoDB:
    host: <string>|'localhost' # The database host to connect to
    port: <integer>|27017      # The database port to connect to
    name: <string>|'default'   # The database name
```

Registry values The registry property mongo_db, associated with the MongoDB connection object to specified database, is stored in the registry.
In addition, the registry property mongo_db_admin is also stored: it is the global MongoDB object which can perform system-wide operations. This init plugin is required if the application models subclass `SimpleMongoDBModel`.

Session init plugin

Purpose Creates a client session.

Configuration If the following property is defined in the configuration file, the session will time out after the specified value (in seconds) has elapsed:

Listing 12: "Session config"

```
<namespace>:
  system:
    session_timeout: <integer> # The value in seconds for the session timeout
```

Registry values The registry property session, associated with the session object, is stored in the registry.
The session object (`Zend_Session` uses getter/setter magic functions. The following snippet shows a simple example on how to use the session object:

Listing 13: "Code using session"

```
/*
In the controller :
-------------------
*/
$session = Zend_Registry::get('session'); // No check , assume it exists. In reality,
    isRegistered should be used.
if ($session) {
  $session->foo = 'bar';
}
// ....
/*
In the view :
-------------
*/
<div><?php echo $session->foo; ?></div>
```

4.2 View plugins

View plugins shall implement the `Iface_Plugin_View` interface. The classname should be `Plugin_View_Foo`, where Foo becomes the view plugin name used in the configuration file (e.g. myplug1/myplug2 above).

There is a single method to implement for these plugins :

```
public function process(Iface_View $view): void :
```

- **$view** : The view object about to be rendered.

This framework currently does not provide any view plugin.

4.3 Resource plugins

Resource plugins shall implement the `Iface_Plugin_Resource` interface. The classname should be `Plugin_Resource_Foo`, where Foo becomes the resource plugin name used in the configuration file (e.g. myplug1/myplug2 above).

There is a single method to implement for these plugins :

```
public function process(string $resource, string $mime): void :
```

- **$resource** : The filename to the resource about to be output
- **$mime** : The mime type identified for this resource

Additional parameters may be supplied depending on the type of resource being handled. Resource are loaded during the init process using `Utils_ResourceLoader::outputResource` . It extracts the resource type to render using the mime type; the type is the first part of the string (until the slash /); i.e. image/jpeg will yield image as the resource type.

Resource types are then used to look for a specific `Utils_ResourceLoader_Abstract` implementation targeting it.
Convention is to name the `ResourceLoader` after the resource type it handles. For instance, image resource types are handled through `Utils_ResourceLoader_Image`, supplied by the library.

If no specific loader can be found, the default one is used : `Utils_ResourceLoader_Default` .

This framework currently does not provide any resource plugin.

4.3.1 Image resource loader

Loader used for image type resources. The list of plugins to apply are defined in the configuration under
`[namespace]->system->plugins->resources->image`.
It passes a third parameter to the `process` method of each plugin: a `WideImage` instance for the specified resource.
Each plugin should also return the new updated `WideImage` instance, or the original one if nothing was changed which is chained to the next plugin and so on.
The return value from the last plugin is then used to output the image.

5 Handling translations

Translations files are namespaced and located under the *langs* folder.
Format used is gettext. It has a .po file, which is plain text, and a .mo file, which is a gettext compiled file.
Free applications such as poedit (http://www.poedit.net/) are very good and can be used to do the translations.

.po/.mo files should be named after the *locale they define* : thus, the japanese translations would be in a file named *ja_JP.po*, giving once compiled *ja_JP.mo*.

As the *langs* folder is namespaced, it is possible to provide different translation files for each namespace. However, the framework does *not* merge the content of all files found in each namespace. Therefore, all translation files in all namespaces should contain all translations, including those that do not change.

ⓘ When using poedit, it is possible to set some po headers that will allow auto update of new/old translation within the application. A good way to do so is to recursively look into "*site root dir*" *views*, *libs*, *layouts* for the keyword "xl8", which is the template plugin for translating a string. It can appear in layouts, views and other plugins, hence those three specific folders. Once it is set, all new translations will be added to the file automatically during synchronization, while removed ones will disappear.
PHP parser type in poedit setup may also need to be changed to add **.phtml* file types to the list of processable files.

5.1 i18n in YAML files/view headers

YAML properties i18n strings can be automatically extracted by software such as poedit provided xgettext.pl and yaml-perl packages are installed.
A YAML property can be declared to be an i18n string by enclosing it with single quotes and prepending the _ character, e.g. `myproperty: _'My string'`.
A script like the following can be created (to be used by poedit automatic extraction feature to the po file). This script can be found in the *config-default* folder. It should be placed to wherever the poedit program can find it :

Listing 14: "Script to extract i18n strings from YAML by poedit"

```
#!/bin/bash
out=$1;
shift;
sed -n '/---/,/---/ {p}' "$@" | xgettext.pl -o `basename $out` -p `dirname $out` -P yaml=*
```

Add a new YAML parser in poedit configuration in Edit > Preferences > Parsers Tab : "YAML" : parser command = "*path/to/script/above/script.sh %o %F*".
Apply this to **.yaml;*.phtml* file types. (The above script will extract the YAML header only from input files).

`Layout` handles the _'...' syntax and will extract the inner string automatically. The latter will be what is returned when querying the view object for this property.

The *title* savant3 plugin treats the "title" and "subtitle" properties from the view YAML header as an i18n string.

6 Handling resources

Currently, the framework consider as static resources the following :

- script files (e.g. javascript). Part of the skin resources, but can be served using a different URL (for third party javascript tools such as ckeditor).
- skin files (e.g. CSS files, javascript, graphics...).
- user uploads (under the "*resources root dir*" folder).

If the request starts with /script/, /skin/ or /upload/, the process will assume what comes next is resp. a javascript file, a skin file, or an uploaded file.

All resources follow the same pattern regarding the file location : each request prefix gets mapped to a base folder under "*resources root dir*" or "*site root dir*", under which namespace folders may be defined, and the remaining path is a direct mapping of the request -minus the prefix- :

/skin/ : Maps to *<site root dir>/skins/<namespace>/*
⟹ */skin/img/top/01.jpg* : Maps to *<site root dir>/skins/<namespace>/img/top/01.jpg*
.
⟹ */skin/css/top.css* : Maps to *<site root dir>/skins/<namespace>/css/top.css*.

/script/ : Maps to *<site root dir>/skins/<namespace>/scripts/*
⟹ This folder is a shortcut to /skin/scripts.

/upload/ : Maps to *<resources root dir>/upload/*
⟹ The upload folder is not namespaced.
e.g. */upload/img/my_img.jpg* : Maps to *<resources root dir>/upload/img/my_img.jpg*.

ⓘ Contrary to other types of resources, skin resources can have a "default" namespace folder underneath it : it may thus contain non skin-specific resources (i.e. shared among all namesapces).

6.1 Managing skins

Skin resources shall contain everything layouts and views use for look'n'feel: this includes stylesheets (CSS), javascript, images....

ⓘ Though planned (and easy to add) but not yet available in the framework is a user-switchable skin functionality : it would be possible to change the look'n'feel or a site by changing the current skin.

ⓘ As explained in section about Managing stylesheets and scripts, stylesheets and scripts managed through `Utils_ResourceLoader` are later rendered via controllers supplied.
The associated stylesheets view will look for stylesheets files in */skin/css/<stylesheet file>/* : thus, it is expected the application to have a "css" folder in the *skins* folder, as shown in the example before.
The scripts are looked for into the "/script/", so similarily, a *scripts* should be available in the application *skins* folder.

6.2 Managing user uploads

Though no editor is supplied with the framework, third party tools such as CKEditor can provide media upload through their WYSIWYG editors. Usually, such software require a special folder where all files are uploaded and stored.
The *uploads* folder in the "*resources root dir*" is meant for this. It has been tested with CKEditor 3.x with a modified version of the fckeditor filemanager for the file upload.

7 Managing stylesheets and scripts

This framework has a utility called `Utils_ResourceLoader` that manages stylesheets and scripts. Until the layout is rendered, controllers, views, view and init plugins, template filter s and plugins can add either resource to the stack. During the layout rendering, all stacked resources will be aggregated through special controllers supplied by the framework, so that a single request will retrieve everything. In addition, stylesheets are also *compressed* on-the-fly.

7.1 Adding stylesheets

A new stylesheet is added by using `Utils_ResourceLoader::addStylesheet($filename, $media='screen')`. `$filename` is the basename of a *PHP* script located in the folder designated by */skin/css/*.
`$media` is a comma-separated list of valid CSS viewports. e.g. screen, print, handheld.... The following snippet is a view plugin that adds stylesheets to the stack based on the view layout to be rendered :

Listing 15: "Sample to add stylesheets to the stack"

```
$layout = $view->layout;
// Try to add stylesheet named after the layout template
Utils_ResourceLoader::addStylesheet($layout, 'screen,print');
// Add print only stylesheet
Utils_ResourceLoader::addStylesheet('print', 'print');
$b = new Browser();
// Add IE specific stylesheet
if ($b->getBrowser() === Browser::BROWSER_IE) {
  Utils_ResourceLoader::addStylesheet('ie', 'screen,print');
}
```

The code above will require a stylesheet to be named after the layout the view is rendered into. It also adds a "print" media stylsheet, named print and an Internet Explorer specific stylesheet if detected browser is IE. Thus, the three following files will have to be available :

- *<site root dir>/skins/<namespace>/css/$layout.php*
- *<site root dir>/skins/<namespace>/css/print.php*
- *<site root dir>/skins/<namespace>/css/ie.php*

Stylesheet files are in fact PHP files because :

- They are not included directly as is in the layout, and thus there's no constraint on them being .css files.
- By using PHP scripts, it is possible to build CSS whose content dynamically changes based on values stored in the registry, the request parameters. . . .
  It also allows to make changes more easily when dealing with fixed width designs where each div, span. . . width/height depend on their parent values. PHP allows to include calculations from base values so that any change somewhere is propagated everywhere.
- by using **`include`**s, it is possible to split the stylesheet into smaller files, and reuse common parts while keeping specific ones in separate filenames.

7.2 Adding scripts

Scripts can be added to the resource stack, using `Utils_ResourceLoader::addScript($fileOrScript, $embedded=`**`false`**`, $prio=50)`. `$fileOrScript` can be either :

- The basename of a ".js" script file; this is the default : the content of the specified file will be included. Needs `$embedded` to be false.
- javascript code : Instead of including the content of a file, the content is this parameter directly. Needs `$embedded` to be true.

`$embedded` false if `$fileOrScript` is a file, true if some code. `$prio` is a priority : if higher, the file/embedded code will be output before the lower priority ones.

The following snippet is a view plugin that adds scripts to the stack based on the view layout to be rendered :

Listing 16: "Sample to add scripts to the stack"

```
$layout = $view->layout;
Utils_ResourceLoader::addScript('jquery-1.4.2.min');
if ('top' === $layout) {
  Utils_ResourceLoader::addScript('top');
}
Utils_ResourceLoader::addScript('site');
Utils_ResourceLoader::addScript('$(document).ready(MyLIB.lib.init);', true);
Utils_ResourceLoader::addScript('$(window).load(MyLIB.lib.load);', true);
```

The code above adds jquery scripts to the stack as well as the site main javascript file. If the view layout is "top", the file top is also added. Finally, two embedded code are added, which resp. call `MyLIB.lib.init` and `MyLIB.lib.load` on document `onReady` and window `onLoad` events. Thus, the following files will have to be available :

- *<site root dir>/skins/<namespace>/scripts/jquery-1.4.2.min.js*
- *<site root dir>/skins/<namespace>/scripts/top.js*
- *<site root dir>/skins/<namespace>/scripts/site.js*

7.3 Render stylesheets and scripts in layouts

This framework supplies two template plugins to render stacked stylesheets and scripts into a layout : stylesheets and scripts.

7.3.1 stylesheets template plugin

All stacked stylesheets are retrieved using `Utils_ResourceLoader::getStylesheets()`. Each stylesheet will become a query string parameter to the stylesheets controller. The key is the stylesheet name; the value is the list of media. The returned string is the generated query.

Thus the typical use of the plugin is the following :

```
<link rel="stylesheet" href="<?php echo $this->stylesheets(); ?>" type="text/css" media="all" />
```

`media="all"` is fine because each file defines its own media targets and will be included using the CSS syntax. `@media screen,print ...`

For files added in section Adding stylesheets, the following URL would be generated (after HTML encoding) :

```
/stylesheets?<layout>=screen,print&amp;print=print&amp;ie=screen,print
```

Thus, a single request to the stylesheets controller will render all stylesheets into a single `link`.
The controller just sets a new registry property with the query string parameters.
The view retrieves this parameter, and for each of them, tries to **`include`** the associated php file located in the correct folder. If a file is not found, the view ignores it and continue with the next one.

7.3.2 scripts template plugin

All stacked scripts are retrieved using `Utils_ResourceLoader::getScripts()`. They are then sorted by descending order of priority.
Files and embedded code are grouped separately :

- Files yields a query string similar to the stylesheets one : `/scripts?<script1>&<script2>...`.

- Embedded code are all wrapped inside a `<script></script>` element.

The returned string will be two `<script>` elements : one with a `src` attribute pointing at the generated query string URL above, another with all the embedded code.

Thus the typical use of the plugin is the following :

```
<body>
<div>...</div>
<div id="footer">...</div>
<?php $this->scripts(); ?>
</body>
```

Putting the scripts at the end of the page improves performance.

For files added in section Adding scripts, the following URL would be generated (after HTML encoding) :

```
/scripts?jquery-1.4.2.min&amp;top&amp;site
```

Thus, a single request to the scripts controller will render the content of all specified js files into the output of the `script`.
The controller just sets a new registry property with the query string parameters.
The view retrieves this parameter, and for each of them, tries to **`file_get_contents`** the associated .js file located in the correct folder. If a file is not found, the view ignores it and continue with the next one.

A Configuration

Listing 17: Configuration values

```
default:                        #Provides with default values for all properties for subclassing
    namespaces
  system:                       #System configuration
    debug: true|false           #Turn debug on/off
    timezone:  [TIMEZONE]       #Valid timezone name. Eg: UTC , Asia/Tokyo
    locales: [locale, ...]      #Supported locales. Eg: en_US, ja_JP
    loglevel: [LOGLEVEL]        #From 0 to 7, or one of EMERG, ALERT, CRIT, ERR, WARN, NOTICE,
        INFO, DEBUG
    session_timeout: [seconds] #If session plugin enabled, number of seconds until session
        timeouts
    charset: UTF-8              #Default encoding used
  plugins:                      #Defines init|view plugins to enable for current namespace
    init:                       #List of init plugins to enable
      - <plugin>
      - ...
    view:                       #List of view plugins to enable
      - <plugin>
      - ...
    resources:                  #List of resource plugins to enable
      - <plugin>
      - ...
  mysql:                        #If mysql plugin enabled, provides Mysql DB connection info
    host: [host]                #Mysql host (default : localhost)
    user: [user]                #Mysql user (default : '')
    pass: [password]            #Mysql user password (default : '')
    name: [db name]             #DB name to connect to (default : 'default')
  mongoDB:                      #If mongoDB plugin enabled, provides MongoDB connection info
    host: [host]                #MongoDB host (default : localhost)
    port: [port number]         #MongoDB server port (default : 27017)
    name: [db name]             #DB name to connect to (default : 'default')
  directories:                  #Server directory names
    resources: [resourcesdir]  #Name for the resources folder
    skin: [skindir]             #Name for the skins folder. Under [site root dir (DATA_DIR)]
    lang: [langdir]             #Name for the langs folder. Under [site root dir (DATA_DIR)]
    upload: [uploaddir]         #Name for the upload folder. Under [resourcesdir]
    log: [logsdir]              #Name for the logs folder. Under [resourcesdir]
    cache: [cachedir]           #Name for the cache folder. Under [resourcesdir]

<namespace>:
  overrides: <namespace>|default # The namespace it overriddes. Should always override 'default'
      if none.
  ... overridde properties ...

...
```

[**END OF DOCUMENT**]