Web::class documentation

Index

1. Introduction to Web::class

1.1. What is Web::class?

Web::class is a fast and lightweight CMS for power users, used to create static and dynamic web pages.

Features:

- Database-driven multi-user core
- Templating support (Twig)
- Lua support for dynamic scripting
- Database support (via extensions)
- Send e-mails via web pages (Swift Mailer)
- Translation infrastructure
- PHP external application support
- Can be extended via extensions

1.2. Architecture

Web::class is a multi-user system, but can also be used only by one user.

Directory structure

- *apps/*: In this directory, you can store any normal PHP application.
- *cache/*: Directory for cache files. Web::class compiles certain files (such as templates or translation lists) into PHP files, in order to speed up the execution time. Web::class creates those files automatically when necessary.
- *class/*: This is the Web::class application and usually this does not need to be modified by the user, nor should it be writable by the webserver.
- *docs/*: Directory containing this document and also the API documentation of Web::class.
- *links/*: Quick links to user directories.
- *tools/*: Example databases.
- *users/*: User files, such as their templates, translation lists, etc...
- *path.php*: Main routing file, where all web requests start.

How a web page is displayed

Once a web request is made to Web::class, for example **http://www.example.com/examplepage**, it is routed to the file **path.php**, via the **rewrite** directives configured for the webserver.

path.php loads **class/Web.class.php**, which in turn is the responsible party to load the web page requested.

First the domain name, including the subdomain, is checked against the **domains** database. So, **www.example.com** is **NOT** the same as **example.com**. If this is required, you can use the built-in redirect PHP application to redirect non-www to www pages (or viceversa).

If the domain is not found in the database, the file **users/0/templates/webinactive.tpl** is loaded, and presented to the browser. Likewise, if the website is disabled in the database, the same page is loaded.

Note that the special user **0** stores those files not belonging to any other user. We will further discuss at the installation section regarding this special user.

If the domain is found, its database connection information will be extracted from the master database. This information will then be used for all database connections.

Finally, Web::class checks the **templates** table and matches the name, in this case **/examplepage**, with the database. If it finds the page, it will be loaded and displayed accordingly. If it is not found, a special page named “404” will be loaded from the same database, if available. If the page’s defined access level is greater than the current user’s level, a special page “403” will be loaded from the database, if available.

Additionally, Web::class checks the **apps** database, and if it finds a matching item (via regexp), the PHP application it points to will be launched. Apps have priority over pages. Therefore, if an app matches first, it will be executed and the page discarded.

All user files are stored under the **users/** directory. Each user has a directory in the form of the user name (or ID if you want to use numbers), and below there is one directory for every domain and protocol. So, in our example this would be **users/user1/example.com/http/**.

Additionally, a soft link needs to be created in the **links/** directory. In our example, it would be a link called **example.com** which points to **users/user1/example.com/**. This is necessary so that assets such as style sheets and images are loaded.

Note: If www.example.com and example.com ought to be the same page, it is customary to make a symbolic link from www.example.com to example.com on the file system.

A user can have the following types of files in his personal folder, beneath the protocol folder:

- *templates/*: Stores the template files, in Twig format. In our example, the file would be **examplepage.tpl**.
- *translations/*: If the site should be localized, this would store translation lists.
- *images/,styles/,scripts/*: These directories store images, CSS style sheets, and Javascript files, if they are used by the website.
- *documents/*: This directory can contain any file, and is served by Web::class as-is when a user accesses **example.com/documents/documentname**.

You can find detailed examples and instructions in the example website that comes with the Web::class distribution.

1.3. Installation

So as to install Web::class, your server needs to meet the following system requirements:

- Apache Web Server >= 2.2.22 or NGINX >= 1.1.9
- PHP >= 5.3.10

And the default for databases (but you could use another database):

- MySQL >= 5.5 and the MySQL extension (included)

Note that the versions displayed above are orientative. Web::class may work with older (minor) versions too, but the versions above are the ones that have been tested to work.

Web::class may work along with other databases, such as Postgres for instance, although it would need a driver to be able to connect to the database. Drivers can be provided by extensions, see chapter “7” - “Extending Web::class”.

Other web servers than Apache and NGINX also work, but need converted rewrite rules (as found in the rewrite_apache and rewrite_nginx files). These are not included. If you have written such rules, please open a bug on the bug tracker to have them included (see http://www.relamp.tk/ for links).

In addition, you will need to have the following PHP packages installed:

- Twig templating engine (http://twig.sensiolabs.org/) >= 1.10.3
- Swiftmailer (http://swiftmailer.org/) >= 4.2.1
- PHPLua (http://repo.or.cz/w/phplua.git) >= git 4a505d59bcfd36a2d82613c5c35f786d8ae1476e

Again, the versions listed above are orientative. Web::class may work with older versions, but these are known to work.

Once these dependencies are met (the install script will check for these), you can install Web::class.

1. Copy all the files into a folder on your web server.
2. Make sure the **cache/** folder is writable by your web server. It does **NOT** need to be writable by any other user, under normal circumstances.
3. Make sure the **users/0** folder is writable by your web server. This is required for the installation only, and should be reverted afterwards.
4. Open **http(s)://<yourwebaddress>/install.php** to launch the installation process.

During the installation process, you will have to enter your database information (skip this step if you intend to not use MySQL), so keep this ready. Please try to avoid using the database “root” user. While Web::class tries its best to avoid problems, unknown bugs still may occur. It is best to create a new user for Web::class.

The installation also includes a demo database and files. This feature is recommended for first time users, as it offers a more comprehensive overview of all features than this documentation. If you just want to install Web::class and use your own files/backup, you may delete the demo database and files.

Note: Don't forget to apply the rewrite rules at your web server as described in the rewrite_apache and rewrite_nginx files; otherwise, Web::class will not be able to load any assets!

Once Web::class is installed, you can start creating web pages!

2. Creating your first page

2.1. What is a page?

A page consists of one document, or section, within the website. With other words, let's say that a website that is made of two URLs (example.com/page1 and example.com/page2) has two "pages".

To create a new page, two steps (or three, but more on that later in section 3) are required:

1. Insert the page information into the website database.
2. Create the template (the current HTML document).

Also, of course you will need to upload other elements specified in your HTML code, such as CSS stylesheets, Javascript code files, images and other documents.

All files have to be included in the user's personal path and below the website's main directory, e.g. **users/user1/example.com/http/.** (we've already seen that in the "1.2" - "Architecture" chapter).

2.2. The database part

For each website you want to handle by means of Web::class, you first need to "register" and "enable" the new website within the master database (usually called "webclass") in the "domains" table. You can use several tools such as **PHPMyAdmin** (MySQL), or similar, to edit your database and create new records.

In the "domains" table you will have to provide correct details in these fields:

- *domain*: This has to be the **exact** domain name that the website will use. If you need to have two domains, you will need to create two rows. Note that **example.com** and **www.example.com** is NOT the same in this case.
- *username*: This is the user name this website belongs to.
- *protocol*: HTTP for normal port 80 HTTP, HTTPS for port 443 SSL. If the same website should be accessible on both ports, you need to create two rows.
- *rootdir*: This is usually the same as the user name. Only for special installations this should be changed, but is not recommended.
- *database*: The complete name of the database where content info can be found. A standard convention is to use **user1_http_www_example_com**. If you have created one or more rows for the same website, you can use the same database name here on all rows if the page should be the same.
- *timezone*: This is the time zone setting for this website. Different websites can have different time zones, so it is possible to host different applications for different time zones on the same server. It is expressed in PHP TZ format: http://php.net/manual/en/timezones.php.
- *admin*: If this flag is set to 1 (true), then this website will be set as an "administrator" site, meaning it will have access to the master database (webclass). If not, set to 0 (false), which is the option selected for regular users.
- *active*: It controls whether the website is currently active. Set to 1 (true) for activation, and to 0 (false) for deactivation. If set to 0, Web::class behaves as if the website didn't exist and nobody can access it. Instead, the template **users/0/templates/webinactive.tpl** is displayed to the browser.

- *securecookie*: Controls wheter the default session cookie will be sent over all connections (0) or only HTTPS connections (1). This will effectively set the 'secure' flag on the cookie in the client browser.

Once you have set up the master database, you will need to set up the slave database and tables. Make sure you use the same name as the one you provided in the **database** field for the new database.

Now, in order to create a page for the new website, you will need to provide the details of the page in the slave database, in a table called “templates”. This is the location where all the pages go.

You have to fill in all the following fields:

- *match*: This is the PHP regular expression match. To match the “/page1” page for example, you would use #^/page1$#”. See the PHP documentation for more information: http://php.net/pcre.
- *linkurl*: If this page is linked to, this will be the url of the link, for example “/page1”.
- *title*: This is the page title. While you can typically use anything here, you would usually type the page title, the HTML “<title>”-tag.
- *keywords*: The words for the meta tag “keywords” go here.
- *leftmenu*: If the page is going to have a secondary sub-menu, “leftmenu” specifies when to highlight the links to the pages.
- *topmenu*: Same as above, just for the primary main-menu.
- *description*: An optional description for the page. Alternatively, it can be used in the page template.
- *metadescription*: Text that goes into the meta “description” tag.
- *file*: The template file to use. It is relative to the **users/username/domain/protocol/templates/** directory.
- *mastertemplate*: This template functions like a master page, e.g. it integrates the main website header in all pages. It is relative to the **users/username/domain/protocol/templates/** directory.
- *menuid*: The ID of the menu you are implementing. You can have multiple menus used within different pages.
- *statistics*: We need to specify the template used for statistic tracking, e.g. Google Analytics. It is relative to the **users/username/domain/protocol/templates/** directory.
- *sitemap*: If the sitemap app is used, we should indicate whether this page is included in the sitemap. Type 1 (true) to include it or 0 (false) to hide it.
- *requiredaccesslevel*: Specifies the necessary access level to view this page. It can be useful along with login and permission systems.

Example

We will provide plenty of examples as we go through this manual!

We first need to create a new website in the master database, “domains” table:

- *domain*: We will use “example.com”. Please modify this as needed for your installation, any domain can be used. We will also use “www.example.com”, so we need to create two identical rows and just change the “domain” field.
- *userid*: Since this is a new installation, we will use “user1” as user name. Remember that the user 0 is reserved to Web::class, so it can not be used. We will also create a new directory under **users/** on the web server, and name it “user1”.
- *protocol*: For protocol we will use “HTTP”.
- *rootdir*: We will use the same name as the folder of the user name, so “user1”.
- *database*: As slave database name, we will use “user1_http_www_example_com”.
- *timezone*: Set this to the desired time zone setting. Here, we will use “Europe/Madrid”.

- *admin*: This is not an admin page, so we set this to “0”.
- *active*: We need to activate this website by setting this to “1”.
- *securecookie*: Since we are not using HTTPS, set this to “0”.

Now that Web::class knows about our new website, if we type **example.com** or **www.example.com**, we will no longer see the standard “web inactive” message. However, there will be errors as we do not yet have any pages.

2.3. The master template part

When working on a web site, the header or footer (or both) are usually common to all pages. One option would be to copy & paste those into all pages of your web site, but it would be considered tedious work. Also, should your header or footer require any changes, you would have to go through all pages manually.

This is where the master template comes in. Like a “master page” in an office application, the master template is automatically embedded on all pages by Web::class, if it exists.

To make your life easier, Web::class also provides a few variables that can be used in your master template:

- *{{ pageData.title }}*: This variable will replace the page title from the database.
- *{{ pageData.metadescription }}*: This variable will replace the “metadescription” field from the database.
- *{{ pageData.keywords }}*: This variable will replace the “keywords” field from the database.
- *{{ systemData.now|date('Y') }}*: This is a special variable. “systemData.now” will get the current timestamp (Unix), and the “date” function will format it for display. PHP “date” format rules apply. Those are so-called “Twig expressions”. To learn more about the Twig templating engine, visit: http://twig.sensiolabs.org/.
- *{% include pageData.file ~ ".tpl" %}*: This expression will include the page itself after collecting the page name from the database. This is the most important expression, because it will be responsible for your page to be displayed.
- *{% include pageData.statistics ~ ".tpl" %}*: Same as above, but including the “statistics” file from the database.

As you see, all of the above are “Twig expressions”. Twig is a fast and lightweight templating language. To learn how to use it, we recommend you now check out http://twig.sensiolabs.org/.

Standard variables

Web::class provides some variables to your document which you can use in your templates and Lua scripts.

- *pageData*: Holds an array with all fields of the current page row (“template” table in your database).
- *systemData*: Holds an array with three variables: “now” is the current UNIX timestamp, “request” is the complete URL requested, “post_data” is the POST (and GET) variables array, “remoteip” is the IP address of the user browsing the site.
- *userData*: Holds the user’s session array. Web::class will automatically create a session for each visitor, which you can use to store and retrieve values between pages.

Example

Moving forward with our example, we will make a simple master template, like this:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang="en" dir="ltr">
```

```
<head>
    <title>{{ pageData.title }}</title>
    <meta name="description" content="{{ pageData.metadescription }}" />
    <meta name="keywords" content="{{ pageData.keywords }}" />
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
</head>

<body>
        {% include pageData.file ~ ".tpl" %}
</body>
</html>
```

Save this file under **users/user1/example.com/http/templates/master.tpl** (change the path accordingly if needed).

2.4. The template part

Now that we have created our master template, we can go ahead and create our individual pages. As you already know, for each page, we have to make an entry in the slave database; within the “templates” table:

- *match*: Type “#^/page1$#”.
- *linkurl*: Type “/page1”.
- *title*: Type “Title of Page 1”.
- *keywords*: As keywords, we’ll use “page, 1”
- *leftmenu*: Since we have no menu, this will be blank.
- *topmenu*: Same as above, leave this blank.
- *description*: An optional description for the page. In our case, we do not need this.
- *metadescription*: Type “This is page 1 of our website”.
- *file*: The template file to use. Type “page1”.
- *mastertemplate*: Here we specify our previously created master page. Type “master”.
- *menuid*: We do not have a menu, so this is blank.
- *statistics*: We do not have statistics yet, so leave this blank. Here you would ideally put the name of your template containing statistics code, e.g. Google Analytics.
- *sitemap*: Set this to “1”.
- *requiredaccesslevel*: We want everybody to see this page, so we set this to “0”.

Now that we have created our database row, we also need to create the template file on the server. Use the following code:

```
<p>This is page 1!</p>
```

Save it into **users/user1/example.com/http/templates/page1.tpl** (change the path accordingly if needed).

Right now, if you type into your web browser **example.com/page1**, you will see the new page appear. It is that easy!

2.5. Linking pages

Of course, a normal web site has more than one page, which are usually linked together. Web::class provides a linking mechanism that allows you to put links into your page, without having to worry about name changes. These go into the “links” table of the slave database.

These fields are necessary for the new table called “links”:

- *linkname*: This is the name of the link you will later use in your templates/scripts to refer to this link. Note that more than one link can be named the same, if you are using translations. We will later get to this point in section “4” - ”Translating your page”.
- *language*: When using translations, you would put in the language code of this link. We will later get to this in section “4” - “Translating your page”.
- *menuid*: This is to “group” different links into a menu, which later can be displayed on the page. In the next section we’ll get to this.
- *text*: This is the name of the link that may be displayed to the user. Note that you do not have to use this, but it is recommended if you later want to use translations.
- *templateid*: This is the ID of the template. This link will open (field “id” of the “templates” table).
- *visible*: Set this to “1” (true) to display the link, “0” (false) if you want to hide it. This property makes only sense if the link is included in a menu group.
- *prio*: A number which gives the link priority in the menu, e.g. you can reorder links this way, from lower to higher. This property is only effective if the link is included in a menu group.
- *displayaccesslevel*: Access levels so as to display this link. Multiple access levels can be specified by using a semicolon “;”. For the remaining access levels, the link will be hidden.

Example

We want to create another page in addition to the one we have just set up. For users to be able to reach that page, we’ll need to make a link to it from the first page we created.

First, create a second page following the steps we’ve learned in the previous section, for example, “page2”.

Next, we’ll create a new row in the “links” table:

- *linkname*: We’ll call it “page2”.
- *language*: We will not use translations, so this should be blank.
- *menuid*: We are not including the link in a menu, so leave it blank.
- *text*: We’ll call it “Link to page 2”.
- *templateid*: Type in here the ID of the template where you want to link to. If you have so far followed this tutorial, you will know it should be “2”.
- *visible*: This should be “1”, though it will not have any effect right now.
- *prio*: Set this to “0”, as we do not need to sort the links now.
- *displayaccesslevel*: Set this to “0”, so every user can see the link.

Now that we have defined our link, we will also need to display it. For that, we need to “load” it first via a Lua script. Don’t worry about the details now - you’ll learn how to use Lua in section “3” - “Using LUA to script your page”.

For now, create a file called “page1.lua” in the same directory you placed “page1.tpl” (change the path if necessary), and type in the following content:

```
-- Executed before the page loads
function on_load(args)

        -- Load links needed in page
        a = {}
        a["lpage2"] = link("page2")
        return a

end

--
-- Switch between functions
--
return _G[funcToCall](funcArgs)
```

Now, in your “page1.tpl”, we can use this link:

```
<p><a href="{{ lpage2.url }}">{{ lpage2.name }}</a></p>
```

Now refresh the page and you should see the new link working.

2.6. Creating a page menu

Many web pages have a menu that appears on every page. Web::class provides two built-in menu levels:

- *top menu*: This is the main menu of the website or the menu that is normally displayed at the top of the website.
- *left menu*: This is a sub-menu for the main menu items. Note that each main menu (“top menu”) item may have its own left menu items.

To create a menu, we must first define a menu ID. We do this by changing the “templates” table:

- *menuid*: This is the name we use to define which menu the page should display.
- *topmenu*: This is the link ID (“id” field at the “links” table) that is used to identify the page in the links. This is also used to highlight the links in the menu later.

We also need to modify some fields in the “links” table:

- *menuid*: Each link included in a menu needs to have the same name, and it needs to match the name of the “templates” table.
- *prio*: We can use this option to order links in a menu as per priority. Starting from 0, each higher number is placed below (or on the right-hand side) the lower number links.

Example

We will now add a “top menu” to our example web page. For this, let’s modify both “page1” and “page2” on the “templates” table:

- *menuid*: For both, type “mainmenu”.
- *topmenu*: Create a new link in the “links” table for “page 1”, and fill in the IDs of the links corresponding to each page.

Now, we will also need to modify the “links” table:

- *menuid*: Type “mainmenu” for both links.
- *prio*: “0” for “page 1”, “1” for “page 2”, as we want to display the link for “page 1” before the link to “page 2”.

We will also need to include this menu in our templates (both “page 1” and “page 2”:

```
<p>{% for link in topMenu %}
        {% if link.high %}
                <li class="toplink_high"><a href="{{ link.url }}">{{ link.text }}</a></li>
        {% else %}
                <li class="toplink"><a href="{{ link.url }}">{{ link.text }}</a></li>
        {% endif %}
{% endfor %}</p>
```

Now if you refresh the page, you should see two links for the menu. If you also add the two styles (toplink_high and toplink) as CSS, you will even see the links being highlighted!

So what about the “left menu”?

Including a “left menu” is even easier! For each page you want to include a “left menu”, you only need to create the links in the “links” table, then fill the “leftmenu” field in the “templates” table.

You can include more than one link by separating the link IDs using a semicolon “;”, e.g. “0;1”.

To include the “left menu” on a template page, use a code like this:

```
{% for link in leftMenu %}
        {% if link.high %}
                <div class="menulink_high"><a href="{{ link.url }}">{{ link.text }}</a></div>
        {% else %}
                <div class="menulink"><a href="{{ link.url }}">{{ link.text }}</a></div>
        {% endif %}
{% endfor %}
```

2.7. Apps

Under certain circumstances, you may want the web page to do something not easily possible with Lua, or you need to include an external PHP script.

In Web::class, an “app” is an arbitrary PHP script that can be called from any URL, or various URL’s with regular expressions.

“Apps” are stored in the **apps/** directory of the Web::class distribution, where some example apps are already included:

- *langredirect.php*: This app can redirect the user’s browser to a language-specific page of your website. Arguments: default (default link ID if no language), en (link ID of given language), es (link ID of given language), etc...
- *redirect.php*: This app can redirect the user to any given URL. Arguments: url (URL to redirect to).
- *sitemap.php*: When called, it outputs a XML sitemap. Arguments: None.

To define an “app”, you need to fill in the “apps” table of the slave database:

- *appname*: Name of the application, **must** have the same name as the PHP file, without the extension.
- *arguments*: Many “apps” have arguments. Arguments are key=value pairs separated by a semicolon “;”.
- *match*: This is the PHP regular expression match. To match the root “/” page for example, you would use “#^/$#”. See the PHP documentation for further information: http://php.net/pcre.

Example

Suppose we wanted to redirect our domain root “/” to language-specific sub pages. We have an English page (called “en”, template ID “1”), and a French page (called “fr”, template ID “2”). If the web browser has another language, we will redirect the user to the English version.

We would put this into the “apps” table:

- *appname*: Type “langredirect”.
- *arguments*: Type “default=1;en=1;fr=2”.
- *match*: Type “#^/$#”.

This way, each time a user opens the root page of our website, this application is called.

Refer to the included example “apps” and the Web::class API documentation so as to find out more on how to write other apps.

3. Using LUA to script your page

3.1. What is LUA?

Lua is a powerful, fast, lightweight, embeddable scripting language. Web::class uses Lua in order to provide scripting support for pages.

Since Web::class is a multi-user system, it makes it possible to allow your users to “script” their page, without needing to implement difficult OS permissions and restrictions. The Lua engine in Web::class runs sandboxed, so no user can touch the files and databases of another user.

We suggest you now take a look at the Lua web page to learn more about this language: http://www.lua.org/.

3.2. How to use LUA in different pages

Web::class calls Lua at different levels:

- *master template*: For each master template you use in your web page, Web::class may call a Lua script file. Note that the Lua file needs to have the same name as the master template, except for the extension name which should be “.lua”.
- *templates*: For each template in your application, Web::class can also call its corresponding “.lua” file. If no “.lua” file exists with the same name as the template, Lua will be skipped at this stage.

In order to use Lua for your page, each Lua file needs to include the so called “global” function. This is a function that Web::class will use to call the different functions contained in the file and should be declared like this:

```
--
-- Switch between functions
--
return _G[funcToCall](funcArgs)
```

This will allow Web::class to call your function (“funcToCall”), pass the arguments to the function as array (“funcArgs”), and return the result to your page (if any).

In addition, Web::class can also execute a function when the template is loaded (both master and other templates), with the “on_load” function, for example, to load a link:

```
-- Executed before the page loads
function on_load(args)
```

```
        -- Load links needed in page
        a = {}
        a["lpage2"] = link("page2")
        return a

end
```

3.3. Built-in functions

Web::class provides several built-in Lua functions you can use to script your page. Note that these are only basic, so you most likely will need to extend Web::class to include more functions (we will cover this in chapter “7” - “Extending Web::class”).

If there are any basic functions not included yet, you can send a patch via our bug tracker to have that function included. Note that only basic functions will be included (see http://www.relamp.tk for links).

The functions included are:

- *translate*: Arguments: $transid. Will check the translation files for $transid, and return the translated string.
- *exit*: Arguments: $status. Will call the PHP exit() function, with the return status $status. Must use POSIX numeric return codes.
- *header*: Arguments: $header. Will call the PHP header() function. http://php.net/manual/en/function.header.php.
- *strtotime*: Arguments: $time, (optional) $timestamp. Will turn a string containing a time/date pattern into a UNIX timestamp. You can optionally pass a timestamp, else the current system time will be used. See http://php.net/strtotime.
- *date*: Arguments: $format, (optional) $timestamp. Will turn a UNIX timestamp into a readable time/date string according to the $format specified. See http://php.net/manual/en/function.date.php.
- *setuserdata*: Arguments: $data, $varname. Will add an arbitrary $varname=$data pair to the user’s session. Note that even if a user is not logged in, Web::class still creates a session so you can store variables in it. You can get the variables in the user’s session with the “session” variable. You can also call this function from your template directly, e.g. {% set sessionreturndata = setuserdata('content','myname') %}.
- *mail*: Arguments: $text, $subject, $sendername, $senderemail, $to, (optional) $html. Sends an e-mail containing the text $text, from $sendername <$senderemail> to $to. If the optional $html is given, the message will have an HTML and text version (MIME).
- *opendir*: Arguments: $path, (bool) $recursive, (bool) $removeextensions. This will list all files in $path, optionally including subdirectories and files if $recursive is set to true. If $removextensions is set to 1 (true), extensions will be removed from file names. Note that a user can only open files under his directory, e.g. “users/user1”. Escaping “..” is not allowed, and will be blocked by Web::class.
- *link*: Arguments: $linkid. Loads a link from the database, “links” table.
- *url*: Arguments: $uri, $method(GET|POST), (optional) (array) $query, (optional) (array) $headers. This function can be used to do HTTP/HTTPS GET and POST requests. Needs the PHP cURL extension installed. You can optionally give $query array containing the query string and a $header array containing additional headers to send.
- *json_decode*: Arguments: $json. This function can be used to decode a JSON string.
- *base64_encode*: Arguments: $text. This function can be used to base64-encode a string.
- *hmac*: Arguments: $algo, $data, $key, (optional) (bool) $rawoutput. This function will take $data and sign it with $key, via the algorithm $algo. Is the same as the PHP hash_hmac function: http://es1.php.net/hash_hmac.

Note that all functions are supposed to include just a single argument, i.e. the “args” argument, which is an array of the real arguments passed.

3.4. Calling LUA functions in templates

There are two ways of using Lua functions in your templates:

- *Cache*: Allows you to execute a function and cache its result. This means that the result is stored in the generated PHP code, so each time this page is displayed by anyone, the cached result will be displayed.
- *No-cache*: Allows you to execute a function each time the template is executed, e.g. the page is accessed. This allows for dynamic pages.

To call a function in the page’s .lua file, simply use the Twig “lua” tag (please refer to Twig’s documentation on the use of tags), for example:

```
{% set arguments = ["Argument 1",1234] %}
{% lua nocache returnval functioninluafile arguments %}
```

The “nocache” field indicated we do not want to cache the result. To cache the result, leave this out.

We use the “set” tag to set the arguments for this function call, and as required, we specify the parameters as an array.

The function’s return value (if any) will be stored in the “returnval” variable, and can later be used in the template.

4. Translating your page

4.1. Translation files

Web::class includes a translation infrastructure. This means you only have to write templates and Lua scripts once, but can have several languages displayed.

To begin with, you need to decide on how to “name” the languages included. For example, you could use **example.com/en/mypage** or even **example.com/english/mypage**. Since it is possible to use the same template for several pages, you can easily point **example.com/es/mipagina** to the same template, so you can have both templates and URLs translated (oh! - and links can be translated automatically too)!

As you see, the language tag placed after your domain, in this case “en” or “english”, will determine which language is called, and the template will be the next part. It is important for you to decide this, since all pages will need to follow the same rules.

To actually create translations, we use the so-called “translation” files. These files will be stored under **users/username/domain/protocol/translations/** (example: **users/user1/example.com/http/translations/**), with the name **messages.<language>.yml**, and a syntax like this:

```
# This is a comment
mypage1.text1: 'Hello there!'
mypage2.link2: 'Click here!'
```

In this case, note that we have chosen to give our translations useful names, like “mypage1.text1”, however, you can use any variable name you like here, for example, “myPage1Text1”. These files are in YAML syntax, so you should now check out the YAML language documentation: http://www.yaml.org/.

4.2. Translating templates

Once you have created your translation files, Web::class will automatically find and make them available for you to use. Both in your “master” and other templates, you can use the “translate” function to put the new strings into your template:

```
<p>{{ translate("mypage1.text1") }}</p>
```

Will generate a page saying **Hello there!**.

4.3. Translating strings in LUA scripts

Sometimes, you will need to put some text strings into your Lua scripts. While this is not recommended for design reasons, it often becomes necessary.

Luckily, you can use the same “translate” function in your Lua scripts with ease, for example:

```
-- Translate a string
mynewtext = translate("mypage1.text1")
```

Of course, the translate function could be coupled with other functions as well:

```
-- Use a translated string in a session variable
setuserdata(translate("mypage2.link2"),"myvariable")
```

4.4. Translating links

Of course, links can also be translated. Even better, you do not have to change anything in your templates! (Provided, of course, you did not hard-code the link texts into the templates ;-)).

For this, you have a “language” field in the “links” table of the database. Just fill this info and create copies of the link with different “language” values according to the number of languages used.

Now, just call the link in Lua as usual:

```
-- Load a link
a = {}
a["lmylink"] = link("mylink")
return a
```

And use them in your templates as usual:

```
<a href="{{ lmylink.url }}">{{ lmylink.name }}</a>
```

5. Working with databases on your website

5.1. How databases work in Web::class

In addition of being a database-driven application itself, Web::class also provides several ways for your pages to access a database. Drivers can be provided by Web::class extensions to support many different databases - SQL or NoSQL, to accommodate as much as possible the requirements of many websites.

Currently, only the MySQL extensions ships by default with Web::class, but other drivers may be available. Check http://www.relamp.tk/ for downloads.

Lua scripts can SELECT/INSERT/UPDATE/DELETE database values through the “db_query” Lua function provided by Web:class, which wraps the driver.

5.2. Using databases in LUA scripts

Each driver has its own configuration, so check in the driver’s documentation on how to configure the connection.

Note that the MySQL extension already provides an example in the included **confing.php_dist** file.

db_query

This function executes “SELECT/INSERT/UPDATE/DELETE” queries on the database, and has only three arguments:

- *query*: This is the WCQL (Web::class Query Language) query to execute.
- *parameters*: An array of parameters to pass to the query (see “positional” or “named” parameters in the WCQL documentation. Optional.
- *ismaster*: This is usually not included, and only intended for admin pages (see “Architecture” chapter). If this is set to 1 (true), it will allow querying of the master database. *Note: Only works if the page is an admin page!*

For example, we could do:

```
-- Get things from the database
parameters = {}
parameters["myparameter"] = myvariableinlua
parameters["myparameter1"] = 1234

returnarray = db_query("SELECT MyTable (*) WHERE myfield = :myparameter AND myfield1 = :myparameter1", parameters)
```

Remember that in Lua the array index “0” is invalid, so the first index of the resulting array is “1”.

Web::class Query Language

The Web::class Query Language (WCQL) is a subset of the well-known SQL language. However, due to needing support for any kind of data storage, it is limited in functionality, for example, you can not do JOIN’s or ORDER BY’s.

While this may seem limiting at first, Web::class is built to be able to be highly scaled; and a traditional JOIN model is not very scalable.

Instead, try to “denormalize” your dataset, which might require more “updates” to your tables(s) and some data duplication, but it is very scalable.

Selecting data

We can execute a SELECT statement like this:

```
-- Select from database
parameters = {}
parameters["mycondition"] = "a string";
parameters["myothercondition"] = 123;

returnarray = db_query("SELECT MyTable (myfield1, myfield2) WHERE mycondition = :mycondition AND myothercondition = :myothercontidion", parameters)
```

Note that you can select all fields of a database row/object by specifying a single asterisk “(*)” instead of the fields to query.

Inserting data

We can execute an INSERT statement like this:

```
-- Insert into database
fields = {}
fields["myfield1"] = "a string"
fields["myfield2"] = 123
fields["myfield3"] = "something more"

returnvalue = db_query("INSERT MyTable SET myfield1 = :myfield1 AND myfield2 = :myfield2 AND myfield3 = :myfield3", fields)
```

Note that this function only returns a value specifying if the query has been executed successfully (true) or not (false).

Updating data

We can execute an UPDATE statement like this:

```
-- Update database
parameters = {}
parameters["myfield1"] = "a string";
parameters["myfield2"] = 123;
parameters["myfield3"] = 456;
parameters["myfield4"] = "b string";

returnvalue = db_query("UPDATE MyTable SET myfield1 = :myfield1 AND myfield2 = :myfield2 WHERE myfield3 = :myfield3 AND myfield4 = :myfield4", parameters)
```

Note that this function only returns a value specifying if the query has been executed successfully (true) or not (false).

Deleting data

We can execute a DELETE statement like this:

```
-- Delete from database
parameters = {}
parameters["myfield1"] = "a string";
parameters["myfield2"] = 123;

returnvalue = db_query("DELETE MyTable WHERE myfield1 = :myfield1 AND myfield2 = :myfield2", parameters)
```

Note that this function only returns a value specifying if the query has been executed successfully (true) or not (false).

Operators

For WHERE operations, the following operators are supported:

- >=: Greater or equal than.
- <=: Less or equal than.
- >: Greater than.
- <: Less than.
- =: Equal.

6. Security features

6.1. Introduction

Web::class has some nice security features already built-in. For example, by default Web:class uses a sandboxed Lua interpreter in order to hinder users’ access to another user’s files and database.

Nonetheless, there are not just features for administrators, but some other functionalities for websites that run on Web::class as well.

6.2. Request cleaning

By default, Web::class will strip HTML tags from your POST and GET variables. If this is not desired, you can disable this function in the configuration file **users/0/config.php**.

Note that Web::class will **NOT** automatically quote your POST and GET variables, as different applications have different needs. If you need to quote input strings, please do this on a case-by-case basis, for example, in your Lua script.

Note: Unsafe input strings can be safely passed to all built-in Lua functions of Web::class, as it will take care of quoting the inputs for e.g. database operations.

6.3. CSRF protection

Web::class also features built-in CSRF protection for HTML forms. Web::class will automatically generate a random key on each page load and provide it to your websites.

To use it in a form you can do:

```
<input type="hidden" name="key" value="{{ userData['csrfkey'] }}" />
```

Then in your Lua code you can check it, for example, like this:

```
if args['key'] == nil then
        errorMessage = 'Key not correct'
else
        if args['key'] ~= session['last-csrfkey'] then
                errorMessage = 'Key not correct'
        end
end
```

7. Extending Web::class

7.1 What are extensions?

Extensions allow you to provide:

- New functions for your Lua scripts.
- Drivers for using Web::class with another database.

Extensions are classes providing these functions, written in PHP.

7.2. Writing an extension

All extensions are stored under the **users/0/extensions/** directory, and are thus accessible to all users. Web::class does not limit which user can use the exported Lua or database functions. This way, you may control which user has access to certain functions.

Extension classes have a name syntax of <extensionName>.extension.php, and have the same (or very similar) base code:

```
<?php

class extensionNameextension {

        // Constructor
        public function __construct($Web) {
                // Important if you want to access Web::class!
                $this->Web = $Web;

        }

}
```

Note that the extension class has to end with the word “extension”.

Web::class passes itself as first parameter of your constructor function. This allows you to access the Web::class API (see the API documentation under the **docs/api/** directory of the Web::class distribution on how to use it).

7.3. Exporting functions to LUA scripts

To export functions for our Lua scripts, we need to define a new function:

```
// Return a list of LUA commands
public function getLuaCommands() {

        return array(
                'myfunctioninlua' => 'myfunctioninphp',
        );

}
```

Web::class will call the function “getLuaCommands()” upon initializing the extension, so it **must** be included in the class. If you do not need to provide any functions to Lua scripts (for example, if this is only a database driver), it should return an empty array.

This function is expected to return an array of functions which should be available to Lua scripts. We used a quite simple function here, but it could be much more complicated, for example, to provide access rules.

Each function needs to be given a name as the “key” of the array, that is the name the Lua scripts will use to call it (in this example, it is “myfunctioninlua”). The “value” of the associative array is the function how it is called in PHP, in your class. This can have a different name, for example, to avoid name clashes with built-in PHP functions - which also could be exported here.

The function may return any data type - Web::class will make sure Lua can use it later on - for example, by shifting an array result so no “0” key is used (as this is invalid for Lua).

As an example, our function could do:

```
function myfunctioninphp($myname) {

        return 'Hello '.$myname.'!';

}
```

Now, Lua scripts could call this function:

```
-- Call our exported function
returnval = myfunctioninlua("John")
```

Also templates can call it via the “lua” tag and using a proxy function in your template’s Lua code:

```
{% lua nocache returnval myfunctionintemplatelua "John" %}
{{ myreturnval }}
```

7.4. Creating a database driver

Web::class can be extended to use another driver as database backend. Due to its simple WCQL (Web::class Query Language), it is possible to create a driver for almost anything - SQL databases, NoSQL databases, API's, etc...

For this, you need to provide a couple of functions in your PHP extension class, and configure Web::class to use your custom extension for database operations.

Configuration

In Web::class, all database drivers are extensions, and have the following syntax in the **config.php** file:

```
// Database settings - please change to your host
$this->settings['DataBase']['Driver'] = 'mysqlextension';
$this->settings['DataBase']['Options'] = array(
                                        'host' => '127.0.0.1',
                                        'user' => '',
                                        'password' => '',
                                        'database' => 'webclass', // Default value
                                        'charset' => 'utf8' // Optional but recommended
);
```

The example above is the supplied configuration for the MySQL driver. However, your driver can have other options - Web::class will pass the complete "Options" key to your extension class, so you can use whatever options you need.

Functions in your extension

Your extension class will need to provide some functions to Web::class:

- *void createConnection($options, $ismaster)*: Web::class will pass the **config.php** array in the "$options" parameter. The parameter "$ismaster" indicates to connect to either the master or slave databases. Should you not be able to connect to the database, your class should throw an exception.
- *void closeConnection()*: This is called when you should close the database connection. Do only throw an exception here if your database needs to close correctly to secure the data, otherwise ignore any errors.
- *mixed escape($value, $ismaster)*: This function is called when Web::class expects you to escape user input. Generally, you'll need to support strings, integer and doubles. Return the escaped value. You can optionally use the "$ismaster" parameter to use the correct database for the escape function.
- *string escapeField($value, $ismaster)*: This function is called when Web::class expects you to escape a field name. Return the escaped value. You can optionally use the "$ismaster" parameter to use the correct database for the escape function.
- *mixed db_query($query, $ismaster)*: This function is called with the complete WCQL query. Web::class will also again pass the "$ismaster" value so you can direct the query to the appropriate master or slave database.

The $query array

Web::class will pass you queries in an already formatted "$query" array. This array is completely syntax-checked and escaped, so you only need to handle the query logic in this function.

The “$query” array has some common keys always present:

- *type*: This parameter indicates the query type: T_SELECT, T_INSERT, T_UPDATE, T_DELETE.
- *of*: This parameter indicates the table/object the query is operating on.

Other parameters will depend on the query type.

T_SELECT

This query is used to retrieve rows/values from the database, and can only have two more keys:

- *fields*: An array of fields to return. Can be a single element “*” meaning return all fields.
- *where*: This is an array of “where” clauses to append to the query. Since WCQL is based on a subset of SQL; it relates to what a “where” clause in the SQL query would do.

More than one key/value keys can appear, with different operators. However, between two where clauses, there is always an “AND” operator. “OR” queries are not supported by WCQL.

For completeness, the first key/value array will have the “operation” field set to “T_WHERE”, and all others to “T_AND”, so you can implement the correct order, if it matters in your database.

Field/value comparators can be either <, >, =, >=, <=.

An example array would be:

```
array(
        "type" = "T_SELECT",
        "of" = "MyTable",
        "fields" = array(
                        myfield1,
                        myfield2,
                        ),
        "where" = array(
                        array(
                                "operation" => "T_WHERE",
                                "field" => "myfield1",
                                "operator" => "=",
                                "parameter" => "a string",
                                ),
                        array(
                                "operation" => "T_AND",
                                "field" => "myfield2",
                                "operator" => "<",
                                "parameter" => 123,
                                ),
                        ),
        );
```

This would be the array for the following WCQL query:

```
SELECT MyTable (myfield1, myfield2) WHERE myfield1 = "a string" AND myfield2 < 123
```

T_INSERT

This query is used to insert (**NOT** update, updating should fail) rows/values into the database, and can only have one more key:

- *set*: An array of fields/values to insert.

More than one array of elements can appear, but only the operator “=” is valid in the query, all other operators will be ignored by Web::class.

An example array would be:

```
array(
        "type" = "T_INSERT",
        "of" = "MyTable",
        "set" = array(
                        array(
                                "field" => "myfield1",
                                "parameter" => "a string",
                                ),
                        array(
                                "field" => "myfield2",
                                "parameter" => 123,
                                ),
                        ),
        );
```

This would be the array for the following WCQL query:

```
INSERT MyTable SET myfield1 = "a string" AND myfield2 = 123
```

T_UPDATE

This query is used to update (**NOT** insert, inserting should fail) rows/values in the database, and can only have two more keys:

- *set*: An array of fields/values to insert.
- *where*: This is an array of “where” clauses to append to the query.

The “set” and “where” keys are identical to what was explained above, and share the same restrictions.

An example array would be:

```
array(
        "type" = "T_UPDATE",
        "of" = "MyTable",
        "set" = array(
                        array(
                                "field" => "myfield1",
                                "parameter" => "a string",
                                ),
                        array(
                                "field" => "myfield2",
                                "parameter" => 123,
                                ),
                        ),
        "where" = array(
                        array(
                                "operation" => "T_WHERE",
                                "field" => "myfield1",
                                "operator" => "=",
                                "parameter" => "b string",
                                ),
                        array(
                                "operation" => "T_AND",
                                "field" => "myfield2",
                                "operator" => "<",
                                "parameter" => 432,
                                ),
                        ),
        );
```

This would be the array for the following WCQL query:

```
UPDATE MyTable SET myfield1 = "a string" AND myfield2 = 123 WHERE myfield1 = "b string" AND myfield2 = 432
```

T_DELETE

This query is used to delete rows/values from the database, and can only have one more key:

- *where*: This is an array of “where” clauses to append to the query.

The “where” key is identical to what was explained above, and shares the same restrictions.

An example array would be:

```
array(
        "type" = "T_DELETE",
        "of" = "MyTable",
        "where" = array(
                        array(
                                "operation" => "T_WHERE",
                                "field" => "myfield1",
                                "operator" => "=",
                                "parameter" => "a string",
                                ),
                        array(
                                "operation" => "T_AND",
                                "field" => "myfield2",
                                "operator" => "<",
                                "parameter" => 123,
                                ),
                        ),
        );
```

This would be the array for the following WCQL query:

```
DELETE MyTable WHERE myfield1 = "a string" AND myfield2 = 123
```

7.5. Conclusion

As you see, you can extend Web::class to suit your user’s needs!

7.6. Need help?

Thank you for taking time to read this manual! For assistance, check http://www.relamp.tk!