Share your information

Understanding and developing fetch functions

By Jérôme Vieilledent
http://www.lolart.net/en

Index

Table of Contents

1 Goal description

This tutorial will explain how template fetch functions work and how to develop custom ones. It will also show you how to call them from PHP.

2 Introduction

When developing templates in an eZ Publish project, use of fetch functions is a recurring task. You will of course mainly fetch content nodes/objects, or maybe infos about current user :

```
{def $myNode = fetch( 'content', 'node', hash( 'node_id', 2 ) )
     $userIsLoggedIn = fetch( 'user', 'current_user' )}
```

But sometimes you need to fetch pieces of information that are not covered by built-in functions (consume an external webservice, rows from *non-native* tables in your database, ...). Fortunately, fetch mechanism is extensible and it is fairly easy to add a new custom fetch function. Let's see how to do that...

3 Pre-requisites and target population

This tutorial is targeted to intermediate eZ Publish developers or beginners who has already understood template language and eZ Publish directory structure.

Basic PHP programming skills are also needed in order to understand fetch mechanism. MVC design pattern understanding can also be useful but is not required.

4 When to create a fetch function ?

When working your eZ project, you may ask yourself a few questions :

- *What is the difference between a fetch and a template operator ?*
- *When do I need to create a fetch function ?*

Well, let's try to answer those *vital* questions ;-).

4.1 *Difference between a fetch and template operator*

Well, that's right, the difference is thin as both can suit you in many situations. Here are signifi cant differences :

1. **Fetch functions can also be called from PHP** while this can be difficult for template operators
2. You should consider a **template operator as a data modifi er** (string or array operations for example), while this is not strictly the case for all built-in operators.
3. You can consider a fetch function as a model call in a MVC pattern.
4. **Fetch functions can substantially reduce the amount of template code** and increase its readability by **transferring complex code from template to your PHP** fetch function.
5. Fetch functions are much easier to implement :)

4.2 *When to create a fetch function*

Basically, you will need to develop a fetch function when you need to look for some data in your database, on your file system, in a partner's webservice… This is the main role for this feature, but not only. Indeed, as I said above, **with fetch functions you can drastically reduce template code complexity** .

Did you ever get bored of maintaining template code ? Have you ever dreamt of breakpoints driven debug instead of crappy *attribute('show')* ? Well, as a fetch function is written in PHP, you can transfer all your complex template code into a real function, get rid of template language limitations and take fully advantage of the real eZ Publish framework power. And of course, your fetch result can fully be exploited in your templates, so you can aggregate your data into an array, or even into PHP objects (but for that, you will need to implement some methods in your returned object's class; see the appendix for more explanations).

5 How it works

A fetch function is actually part of a module and is mainly composed of 2 fi les :

- Definition file (*function_definition.php*)
- Function collection class (class where your PHP functions will reside as static methods), that will be named ***<MyModule>*FunctionCollection** (replace *<MyModule>* by the name of your module)

You can get many examples in modules from eZ Publish kernel, take a look into *kernel/content/* folder for example (please note that every folders located under *kernel/*, except *classes/* and *common/*, contain a module).

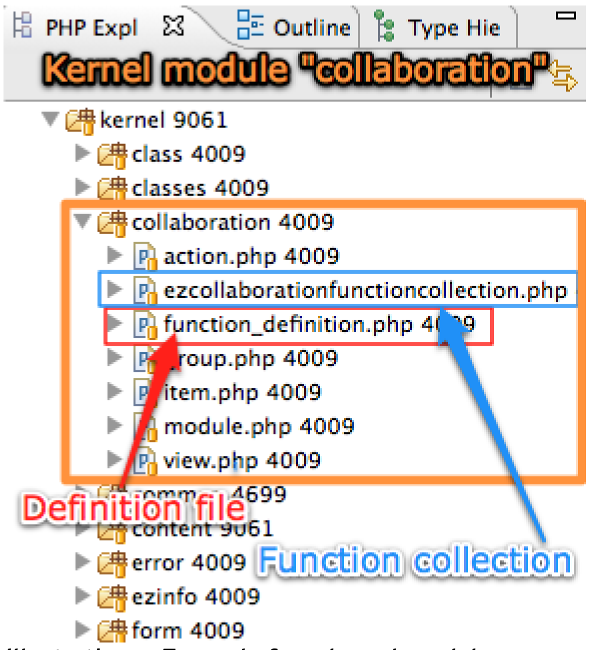

Illustration 1: Example from kernel module "collaboration"

5.1 The definition file

Basically, this file (*function_definition.php*) holds an associative array called *$FunctionList*. Keys of this array are fetch functions names and values are the defi nition as another associative array :

```
<?php
// function_definition.php
// Fetch definition file
$FunctionList = array();

$FunctionList['fetch_function_name'] = array(
    'name' => 'fetch_function_name', // Retype the name of your function
    'operation_types' => 'read', // Obsolete, for BC
    'call_method' => array(
        'class' => 'MyModuleFunctionCollection', // Function collection class
        'method' => 'getMyFetchFunctionResult' ), // Method to call
    'parameter_type' => 'standard', // Obsolete, for BC
    'parameters' => array( // Function parameters in the PHP function order
        array( 'name' => 'first_param',
               'type' => 'string',
                'required' => true ),
        array( 'name' => 'second_param',
                'type' => 'boolean',
                'required' => false,
                'default' => false )
    )
);
```

Above is a typical fetch definition file. You will find similar ones in the kernel or in extensions from the community (take a look into the *SQLIGeoloc* one).

5.2 *The function collection*

As I said before, your PHP fetch function will be contained in a so called **function collection**, which is a simple PHP class with static methods. Fetch code needs to be written in one of those static methods.

The main point to remember about these methods is the argument order. It must be strictly the same than declared in the definition file. Thus, in the example above, *first_param* will be mapped to the method's first argument, and *second_param* to the second.

Each method must return an associative array, with **result** as key and the value to be returned as value. If an error occurs, then it must return an associative array, with **error** as key and the error message as value.

```
<?php
/**
 * MyModule fetch functions
 */
class MyModuleFunctionCollection
{
       /**
        * Fetch function code.
        * You can do everything you want here.
        * Just return an associative array with 'result' as key and your result as value.
        * If an error is raised, return an associative array with 'error' as key and the
        * error message as value
        * @param string $myFirstParam first_param declared in function_definition.php
        * @param string $mySecondParam second_param declared in function_definition.php
        */
    public static function getMyFetchFunctionResult( $myFirstParam, $mySecondParam =
false )
       {
              try
```

```
            {
                  /*
                   * Do everything you want here to fetch/aggregate your data
                   * Associative arrays can be used in templates :
                   * {def $result = fetch( 'mymodule', 'fetch_function_name', hash(
                   *       'first_param', 'something',
                   *       'second_param', true()
                   * ) )}
                   * {$result.first_key} {* Will display "foo" *}
                   * {$result.second_key} {* Will display "bar" *}
                   */
                 $result = array(
                     'first_key'      => 'foo',
                     'second_key'     => 'bar'
                  );

                  return array( 'result' => $result );
            }
            catch( Exception $e )
            {
                  $errMsg = $e->getMessage();
                  eZDebug::writeError( $errMsg );
                  return array( 'error' => $errMsg );
            }
      }
}
```

6 Developing a custom fetch function

6.1 Inside a module

As fetch functions must be part of a module, the only requirement is to have a declared module inside of an active extension.

The *function_definition.php* file will reside inside the module folder, but the function collection class doesn't need to be in it since it will be automatically loaded thanks to the built-in class autoload system. Best practices tell us to create a *classes/* folder inside the extension and place our class here.

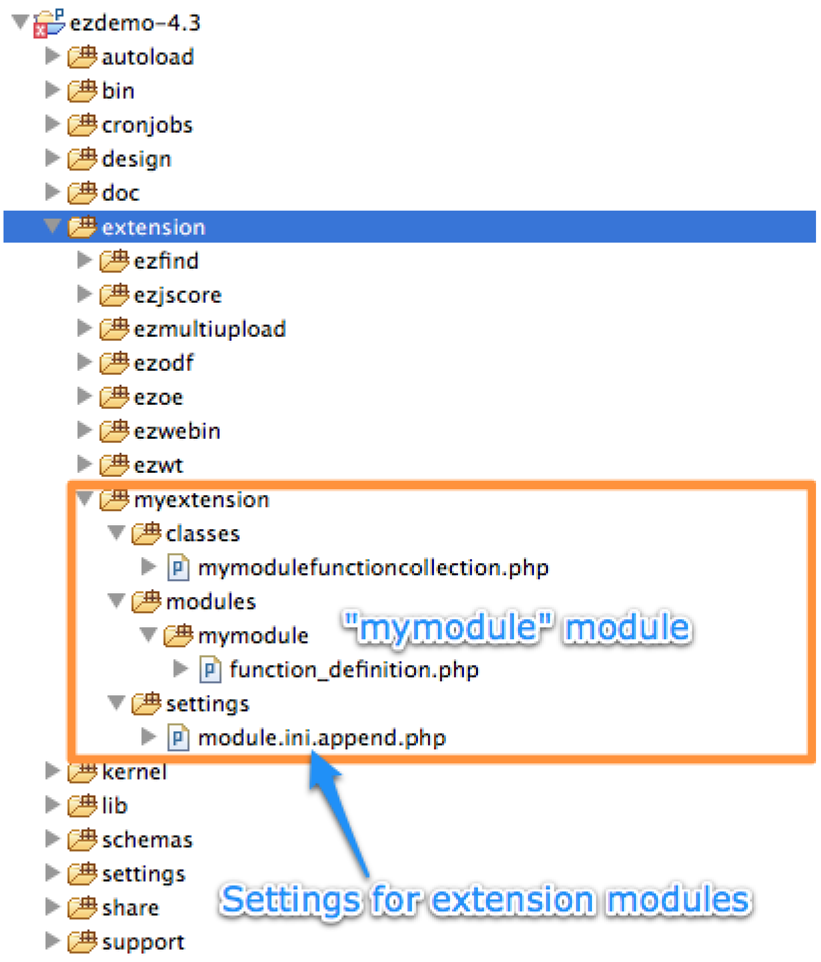
ezdemo-4.3
autoload
bin
cronjobs
design
doc
extension
ezfind
ezjscore
ezmultiupload
ezodf
ezoe
ezwebin
ezwt
myextension
classes
mymodulefunctioncollection.php
modules
mymodule
function_definition.php
settings
module.ini.append.php
kernel
lib
schemas
settings
share
support

Illustration 2: Example for "mymodule" module

So called *mymodule* module need to be declared in **extension/myextension/settings/module.ini.append.php** :

```
<?php /* #?ini charset="utf-8"?

[ModuleSettings]
ExtensionRepositories[]=myextension
ModuleList[]=mymodule

*/ ?>
```

And of course, be sure that your extension is activated in **settings/override/site.ini.append.php** (or in your siteaccess with *ActiveAccessExtensions*).

Et voilà ! You can now use your fetch function in templates :

```
{def $result = fetch( 'mymodule', 'fetch_function_name', hash(
         'first_param', 'something',
         'second_param', true()
     ) )}
{$result.first_key} {* Will display "foo" *}
{$result.second_key} {* Will display "bar" *}
```

6.2 *Using it in PHP*

One of the main advantages of fetch functions vs template operators is that they can easily be called in your PHP scripts. A handler class, **eZFunctionHandler**, with two shorthand methods are available for that :

1. **eZFunctionHandler::execute()** - same as *fetch* in templates
2. **eZFunctionHandler::executeAlias()** - same as *fetch_alias* in templates

```
<?php
$myResult = eZFunctionHandler::execute( 'mymodule', 'fetch_function_name', array(
      'first_param'       => 'something_else',
      'second_param'      => true
) );
echo $myResult['first_key']; // Will display "foo"
echo $myResult['second_key']; // Will display "bar"
```

7 Conclusion

Here is a quick recap :

- Developing fetch functions is an easy way to extend eZ Publish template language
- They can help you to simplify code in your templates and make it more effi cient
- They are easier to write than template operators
- They are callable from templates and from PHP

Please remember that templates should remain simple. As soon as you need to add too much complexity, consider developing a fetch function; you will be hands free and your code will be much easier to maintain.

Also note that you are not forced to write all code in the same PHP static method ! You can delegate part of your code into dedicated classes (Model in MVC pattern).

8 About the author : Jérôme Vieilledent

Jérôme is a completely self-educated web developer. He learnt PHP all by himself and started developing with eZ Publish in 2007. He is eZ Publish certifi ed and now works as a technical manager and expert at SQLi Agency in Paris.

9 License

10 Appendix

10.1 *Make a PHP object usable in templates*

While developing templates, you might have noticed that you can access most of every kernel objects attributes directly in templates :

```
{* $myNode will hold an eZContentObjectTreeNode object *}
{def $myNode = fetch( 'content', 'node', hash( 'node_id', 2 ) )}

{* Directly access to $myNode attributes *}
NodeID : {$myNode.node_id}<br />
Name : {$myNode.name}

{* Inspect $myNode *}
{$myNode|attribute( 'show', 2 )}
```

In order to be able to use a PHP object in templates such as eZContentObjectTreeNode in the example above, you will need to implement 3 methods :

- **attribute($attributeName)**

    - Accessor. Returns value of object attribute which name is *$attributeName*
- **hasAttribute($attributeName)**
    - Checks if object attribute *$attributeName* is available
- **attributes()**
    - Lists every available attributes (only names)

So, when you need to access an object attribute, eZ Publish first checks if it does exist via *hasAttribute()* method. If so, then it will use *attribute()* accessor to retrieve the value. Last *attributes()* method is only used when inspecting the object with *attribute* template operator.

<u>Example</u> :

```
<?php
class MyExampleClass
{
        /**
         * Attribute holder
         * @var array
         */
        private $attributes;

        public function __construct()
        {
                $this->attributes = array(
                        'first_attribute'   => 'foo',
                        'another_attribute' => 'bar'
                );
        }

        /**
         * Checks if $attributeName is a valid attribute
         * @return bool
         */
        public function hasAttribute( $attributeName )
        {
                return isset( $this->attributes[$attributeName] );
        }

        /**
```

```
         * Attribute accessor
         * @return mixed
         */
        public function attribute( $attributeName )
        {
               return $this->attributes[$attributeName];
        }

        /**
         * Returns all available attribute names
         * @return array
         */
        public function attributes()
        {
               return array_keys( $this->attributes );
        }
}
```

Template code :

```
{* Assume that $myObject is a MyExampleClass object *}
{$myObject.first_attribute}{* Will display 'foo' *}<br />
{$myObject.another_attribute}{* Will display 'bar' *}

{* Inspect $myNode *}
{$myNode|attribute( 'show', 2 )}
```