Phalcon PHP Framework Internal Documentation

Release 1.0.0

Phalcon Team

Mar 15, 2017

Contents

Phalcon is not a traditional framework, it's written as an C extension for PHP to provide high performance. The purpose of this document is to explain how it is built internally. If you're interested in helping improve Phalcon, or simply understand how it works, this is the information you need.

CHAPTER 1

Other formats

- PDF
- HTML in one Zip
- ePub

CHAPTER 2

Table of Contents

General Considerations

Phalcon is a C extension for PHP developers. The way in which Phalcon is written is not the usual, if you compare the code of Phalcon with other C extensions you will see a considerable difference, this unusual way of write the extension has several objectives:

- Provide a code that is closer to be understood by developers PHP
- Create objects and components that act as close as possible to PHP objects
- Avoid dealing with low-level issues, whenever possible
- Help maintain a large base code like the Phalcon one

When writing code for Phalcon try to follow these guidelines:

- Writing the code in a PHP style, this will help to PHP developers to introspect easily understanding the internals of PHP objects. Reflection
- Help the Phalcon user to obtain more human backtraces.
- Don't repeat yourself, if PHP has a functionality already developed, why do not simply use it?

Phalcon API

We the creators of Phalcon, are mainly PHP programmers, we are lazy and do not want to deal 100% of the time with low-level details like segmentation faults or memory leaks. We believe that PHP language is incredible, and Phalcon is a contribution to all those things we do every day and it could be faster.

Moreover we want a fast and stable framework. For this reason, we have created the Phalcon API. The use of this API helps us to write C code in a PHP style. We have developed a number of functions to help the programmer to write code more interoperable with PHP in an easier way.

Phalcon API is based on the Zend API, but we have added more features to facilitate us the work. Phalcon is a very large project, frameworks need to be developed and improved every day, Phalcon API helps us write C code that is more stable and familiar to PHP developers.

If you're a PHP developer maybe you don't know C or you don't want to learn C, but after reading this guide you will find the Phalcon API very familiar to your knowledge.

Common Structures

To start explaining Phalcon, is necessary to understand a couple of low-level structures that are commonly used in the framework:

Zvals

Most variables used in the framework are Zvals. Each value within a PHP application is a zval. The Zvals are polymorphic structures, i.e. a zval can have any value (string, long, double, array, etc.). Inspecting some code you will see that most variables are declared Zvals. PHP has scalar data types (string, null, bool, long and double) and non-scalar data types (arrays and objects). There is a way to assign a value the zval according to the data type:

```
PHALCON_INIT_VAR(name);
ZVAL_STRING(name, "Sonny", 1);

PHALCON_INIT_VAR(number);
ZVAL_LONG(number, 12000);

PHALCON_INIT_VAR(price);
ZVAL_DOUBLE(price, 15.50);

PHALCON_INIT_VAR(nothing);
ZVAL_NULL(nothing);

PHALCON_INIT_VAR(is_alive);
ZVAL_BOOL(is_alive, false);
```

This is the internal structure of a zval:

```
typedef union _zvalue_value {
        long lval;                  /* long value */
        double dval;                /* double value */
        struct {
                char *val;
                int len;
        } str;
        HashTable *ht;              /* hash table value */
        zend_object_value obj;
} zvalue_value;

struct _zval_struct {
        /* Variable information */
        zvalue_value value;     /* value */
        zend_uint refcount__gc;
        zend_uchar type;    /* active type */
        zend_uchar is_ref__gc;
};
```

Most of the time you will not have to deal directly accessing the internal structure of a zval. Instead, the Zend API offers a comfortable syntax for altering its value or get information from it.

Previously we saw how to change their values change according to the type, let's see how to read their values:

```
long number = Z_LVAL_P(number_var);

char *str = Z_STRVAL_P(string_var);

int bool_value = Z_BVAL_P(bool_var);
```

If we want to know the data type of a zval:

```
int type = Z_TYPE_P(some_variable);
if (type == IS_STRING) {
        //Is string!
}
```

Zend Class Entries

Another common structure we used is the zend class entry. Every class in the C internal world has its own zend class entry. That structure help us describe a class, its name, method, default properties, etc.

```
//Get the class entry
class_entry = Z_OBJCE_P(this_ptr);

//Print the class name
fprintf(stdout, "%s", class_entry->name);
```

Memory Management

As you may know, the memory management in C is all manual. Within a PHP extension, all memory is managed by Zend Memory Manager, however, the management remains manual.

Manual memory management is a powerful tool that C offers. But, as PHP developers we are not used to this sort of thing. We like to just leave this task to the language without worrying about that.

Phalcon MM

To help in the creation of Phalcon, we have created the Phalcon Memory Manager (Phalcon MM) that basically tracks every variable allocated in order to free it before exit the active method:

For example:

```
PHP_METHOD(Phalcon_Some_Component, sayHello){

        zval *greeting = NULL;

        //Add a memory frame
        PHALCON_MM_GROW();

        PHALCON_INIT_VAR(greeting);
        ZVAL_STRING(greeting, "Hello!", 1);
```

```
        //Release all the allocated memory
        PHALCON_MM_RESTORE();
}
```

PHALCON_MM_GROW starts a memory frame in the memory manager, then PHALCON_INIT_VAR allocates memory for the variable greeting, before exit the method we call PHALCON_MM_RESTORE, this releases all the memory allocated from the last call to PHALCON_MM_GROW.

By this way, we can be sure that all the memory allocated will be freed, avoiding memory leaks. Let's pretend that we used 10-15 variables, releasing manually the memory of them can be tedious.

Others Kinds of Allocations

PHALCON_INIT_VAR only allocates memory for zvals, if we want to request memory for other structures then we must use the functions that provides the Zend API. One very important thing to know is that an extension in C must never use the standard functions malloc and free. The Zend API has its own functions that make the memory access safer.

```
char *some_word = (char *) emalloc(sizeof(char *) * 6);
memcpy(some_word, "Hello", 5);
some_word[5] = 0;
efree(some_word);
```

In short, emalloc allocates memory and efree frees it. If you forgot to make the efree you will get a memory leak.

Variables

Creation

Unlike PHP, each variable in C must be declared at the beginning of the function in which we are working. Also, as noted above, all variables must be initialized before the use. Even, it should be reset again when we change its value, for example:

```
//Declare the variable
zval *some_number = NULL;

//Initialize the variable and assign it a string value
PHALCON_INIT_VAR(some_number);
ZVAL_STRING(some_number, "one hundred", 1);

//Reinitialize the variable and change its value to long
PHALCON_INIT_NVAR(some_number);
ZVAL_LONG(some_number, 100);
```

If a variable is assigned within a cycle or it's re-assigned is important to initialize it to NULL in its declaration. By doing this, PHALCON_INIT_NVAR will know if the variable needs memory or it already have memory allocated.

Copy-on-Write

All the zval variables that we use in Phalcon are pointers. Each pointer points to its value in memory. Two pointers may eventually point to the same value in memory:

```
zval *a = NULL, *b = NULL;

PHALCON_INIT_VAR(a);
ZVAL_LONG(a, 1500);

b = a;

//Print both zvals print the same value
zend_print_zval(a, 1); // 1500
zend_print_zval(b, 1); // 1500
```

Changing the value of any of the two variables will change both pointers because their value are the same:

```
ZVAL_LONG(b, -10);

//Print both zvals print the same value
zend_print_zval(a, 1); // -10
zend_print_zval(b, 1); // -10
```

Now, imagine the following PHP code:

```
<?php

$a = 1500;
$b = $a;

echo $a, $b; // 1500 1500
```

This is practically the same as we did before, however, let's change the value of $b:

```
<?php

$a = 1500;
$b = $a;

echo $a, $b; // 1500 1500

$b = -10;

echo $a, $b; // 1500 -10
```

To our eyes, PHP has done what we did in C, but in reality there has been an internal process called copy-on-write. Pretend that our main memory is this:

```
// $a = "hello"
             +---------+----+
             |   0x1   | RC |
             +---------+----+
zval *a --> | "hello" |  1 |
             +---------+----+
```

The variable $a is pointing to a virtual memory address 0x1, also that memory have a reference counting of 1. It means that only one variable it's pointing to that memory address.

```
// $b = $a
             +---------+----+
             |   0x1   | RC |
             +---------+----+
```

```
zval *a --> | "hello" |  2 |
            +---------+----+
zval *b ---------^
```

Now, $b is equal to $a, now both variables are pointing to the same memory address 0x1. The reference counting is now 2, because two variables are pointing to the same memory slot. As you can see PHP is saving memory, although the variables have different names, they're pointing to the same value in memory so that we are not unnecessarily doubling its value.

```
// $b = 18
            +---------+----+  +---------+----+
            |   0x1   | RC |  |   0x2   | RC |
            +---------+----+  +---------+----+
zval *a --> | "hello" |  1 |  |    18   |  1 |
            +---------+----+  +---------+----+
zval *b ---------------------------^
```

We are changing the variable $b, to avoid changing the value of $a, PHP performs an internal process called "separation". In this process, PHP allocates memory for $b and reduces the reference count in $a to indicate that $b is not pointing anymore to $a.

Let's see how to write the above process using Zend API:

```
zval *a, *b;

ALLOC_INIT_ZVAL(a);
ZVAL_STRING(a, "hello", 1);

//b = a, so we increase the reference counting in "a"
b = a;
Z_ADDREF_P(a);

//Changing the value of b, b isn't pointing anymore to a so we decrease the reference␣
↪counting
Z_DELREF_P(b);

ALLOC_INIT_ZVAL(b);
ZVAL_LONG(b, 18);
```

It isn't a very complicated process, however properly maintain reference counts is another low-level task that can take us away from our true needs. In a large codebase like the Phalcon one, being aware of this can take a long time and wear us down.

With Phalcon API we should not worry about this:

```
zval *a = NULL, *b = NULL;

PHALCON_INIT_VAR(a);
ZVAL_STRING(a, "hello", 1);

PHALCON_CPY_WRT(b, a);

PHALCON_INIT_VAR(b);
ZVAL_LONG(b, 18);
```

Copying variables using PHALCON_CPY_WRT, we leave the task to Phalcon API of caring about the reference counting without worrying us about that.

Operations between Variables

PHP is a dynamic language, we can do almost any operation between two variables, regardless of type. Sometimes we do not know exactly the type of data that have the variables, using the Zend API we can do operations between them seamlessly:

```
//First variable is string but it's a numeric string
PHALCON_INIT_VAR(first_var);
ZVAL_STRING(first_var, "100.10", 1);

//Second variable is long
PHALCON_INIT_VAR(second_var);
ZVAL_LONG(second_var, 150);

//add_function will make the necessary conversions to produce the addition
PHALCON_INIT_VAR(result);
add_function(result, first_var, second_var);
```

Concatenation

Concatenation is one of the most common operations we do in PHP. However, using the Zend API can be tedious when oncatenating many values, for example:

```
// The following concatention using just Zend API:
//
// $month = "July"; $day = 1;
// $today = "Today is ".$month." ".$day;

PHALCON_INIT_VAR(month);
ZVAL_STRING(month, "2012", 1);

PHALCON_INIT_VAR(day);
ZVAL_LONG(day, 1);

PHALCON_INIT_VAR(today_is);
ZVAL_STRING(today_is, "Today is", 1);

PHALCON_INIT_VAR(first_part);
concat_function(first_part, today_is, month);

PHALCON_INIT_VAR(space);
ZVAL_STRING(space, " ", 1);

PHALCON_INIT_VAR(second_part);
concat_function(second_part, space, day);

PHALCON_INIT_VAR(today);
concat_function(today, first_part, second_part);
```

Another way to do that is use sprintf, in this case, you need to be completely sure that the variables have all string types:

```
char *final_string;
zval *final;

PHALCON_INIT_VAR(month);
```

```
ZVAL_STRING(month, "2012", 1);

PHALCON_INIT_VAR(day);
ZVAL_STRING(day, "1", 1);

final_string = emalloc(sizeof(char)*(Z_STRLEN_P(month)+Z_STRLEN_P(day)+12));
sprintf(final_string, "Today is %s %s", Z_STRVAL_P(month), Z_STRVAL_P(day));

PHALCON_INIT_VAR(final);
ZVAL_STRING(final, final_string, 0);
```

It's short, but if some of your variables aren't string you will get a segmentation fault or an unexpected behavior.

To help to solve this problem, we have created a set of macros to concatenate zvals and strings:

```
PHALCON_INIT_VAR(month);
ZVAL_STRING(month, "2012", 1);

PHALCON_INIT_VAR(day);
ZVAL_STRING(day, "1", 1);

PHALCON_INIT_VAR(today);
PHALCON_CONCAT_SVSV(today, "Today is", month, " ", day);
```

Other examples:

```
PHALCON_CONCAT_VV(result, month, day); //July1
PHALCON_CONCAT_VSV(result, month, ", ", day); //July, 1
PHALCON_CONCAT_SVSV(result, "Today is", month, " ", day); //July 1
PHALCON_CONCAT_SVSVSV(result, "Today is", month, " ", day, ", ", year); //July 1, 2012
```

S=String and V=Zval, just put the S and V to get the right concatenation macro. Easy, no?

Working with Arrays

Although the Zend API provides several functions for working with arrays, the Phalcon API has added others. Specifically helping to maintain the reference counting correctly:

One dimension Arrays

```
//Declare the variable
zval fruits = NULL;

PHALCON_INIT_VAR(fruits);
array_init(fruits);

//Adding items to the array
add_next_index_stringl(fruits, SL("apple"), 1);
add_next_index_stringl(fruits, SL("orange"), 1);
add_next_index_stringl(fruits, SL("lemon"), 1);
add_next_index_stringl(fruits, SL("banana"), 1);

//Get the first item in the array $fruits[0]
```

```
PHALCON_INIT_VAR(first_item);
phalcon_array_fetch_long(&first_item, fruits, 0, PH_NOISY_CC);
```

Mixing both string and number indexes:

```
//Let's create the following array using the Phalcon API
//$fruits = array(1, null, false, "some string", 15.20, "my-index" => "another string
↪");

PHALCON_INIT_VAR(fruits);
array_init(fruits);
add_next_index_long(fruits, 1);
add_next_index_null(fruits);
add_next_index_bool(fruits, 0);
add_next_index_stringl(fruits, SL("some string"), 1);
add_next_index_double(fruits, 15.2);
add_assoc_stringl_ex(fruits, SL("my-index")+1, SL("another string"), 1);

//Updating an existing index $fruits[2] = "other value";
phalcon_array_update_long_string(&fruits, 2, SL("other value"), PH_SEPARATE TSRMLS_
↪CC);

//Removing an existing index unset($fruits[1]);
phalcon_array_unset_long(fruits, 1);

//Removing an existing index unset($fruits["my-index"]);
phalcon_array_unset_string(fruits, SL("my-index")+1);
```

Calling Functions

Calling functions is another common action we do when create programs in PHP. Although many functions may be called directly using its internal function pointer in C, others can simply being called using the PHP userland. For example:

```
//$length = strlen(some_variable);

PHALCON_INIT_VAR(length);
PHALCON_CALL_FUNC_PARAMS_1(length, "strlen", some_variable);
```

The macro PHALCON_CALL_FUNC_PARAMS_1 calls functions that requires 1 parameter returning a value. There are another macros to call functions in the PHP userland:

```
//Calling substr() with its 3 arguments returning the substring into "part"
PHALCON_INIT_VAR(part);
PHALCON_CALL_FUNC_PARAMS_3(part, "substr", some_variable, start, length);

//Calling ob_start(), this function does not return anything
PHALCON_CALL_FUNC_NORETURN("ob_start");

//Closing a file with fclose
PHALCON_CALL_FUNC_PARAMS_1_NORETURN("fclose", file_handler);
```

As mentioned above, PHP already has many things going, the fact that an extension in C, does not mean we going to reinvent the wheel all over again develop. Also worth saying that we also try to avoid using very low-level features of

C, if PHP already has its own version. PHP functions help us get a behavior similar to that programming in the PHP would. In this way we avoid possible errors and as result the framework works as if it were PHP.

The following code opens a file in C and write something on it. Its functionality is limited because it only works on local files:

```
FILE * pFile;
pFile = fopen ("myfile.txt","w");
if (pFile != NULL) {
        fputs ("fopen example",pFile);
        fclose (pFile);
}
```

Now write the same code using the PHP userland:

```
PHALCON_INIT_VAR(mode);
ZVAL_STRING(mode, "w", 1);

PHALCON_INIT_VAR(file_handler);
PHALCON_CALL_FUNC_PARAMS_2(file_handler, "fopen", file_path, mode);

if (PHALCON_IS_NOT_FALSE(file_handler)) {

        PHALCON_INIT_VAR(text);
        ZVAL_STRING(text, "fopen example", 1);

        PHALCON_CALL_FUNC_PARAMS_2_NORETURN("fputs", file_handler, text);

        PHALCON_CALL_FUNC_PARAMS_1_NORETURN("fclose", file_handler);
}
```

Although both codes perform the same task, the former is more powerful as it could open a PHP stream, a file in a URL or a local file. We can also write the same code using the PHP API, without losing functionality. However, the above code is more familiar if we are primarily PHP developers.

The same code in PHP:

```
<?php

$fp = fopen($file_path, "w");
if($fp){
        fputs($fp, "fopen example");
        fclose($fp);
}
```

Objects Manipulation

Phalcon is a pure object-oriented framework for PHP. In this chapter, we explain how to make most common operations on objects using the Phalcon API.

Creation/Instantiation

Instantiate objects of the framework classes is easy:

```
//Instantiate a object from the Phalcon\Mvc\Router\Route class entry
PHALCON_INIT_VAR(route);
object_init_ex(route, phalcon_mvc_router_route_ce);

//Calling the constructor and passing a pattern as parameter
PHALCON_INIT_VAR(pattern);
ZVAL_STRING(pattern, "#^/([a-zA-Z0-9\\_]+)[/]{0,1}$#", 1);
PHALCON_CALL_METHOD_PARAMS_1_NORETURN(route, "__construct", pattern);
```

The above code is the same as doing in PHP:

```
<?php $route = new Phalcon\Mvc\Router\Route("#^/([a-zA-Z0-9\\_]+)[/]{0,1}$#");
```

Moreover, if is not a Phalcon class then objects must then initialized as follows:

```
zend_class_entry *reflection_ce;

//Obtain the ReflectionClass class entry, this will also call autoloaders
reflection_ce = zend_fetch_class(SL("ReflectionClass"), ZEND_FETCH_CLASS_AUTO TSRMLS_
↪CC);

//Instantiate the Reflection object
PHALCON_INIT_VAR(reflection);
object_init_ex(reflection, reflection_ce);

//Pass a class name as constructor's parameter
PHALCON_CALL_METHOD_PARAMS_1_NORETURN(reflection, "__construct", class_name);
```

Reading/Writing Properties

Writing scalar values:

```
//Create a stdClass object
PHALCON_INIT_VAR(employee);
object_init(employee);

// $employee->name = "Sonny"
phalcon_update_property_string(employee, SL("name"), "Sonny" TSRMLS_CC);

// $employee->age = 23
phalcon_update_property_long(employee, SL("age"), 23 TSRMLS_CC);

//Read the "name" property $name = $employee->name
PHALCON_INIT_VAR(name);
phalcon_read_property(&name, employee, SL("name"), PH_NOISY_CC);
```

Assigning other zvals to properties:

```
PHALCON_INIT_VAR(language);
ZVAL_STRING(language, "English", 1);

PHALCON_INIT_VAR(employee);
object_init(employee);

// $employee->language = $language
phalcon_update_property_zval(employee, SL("language"), language TSRMLS_CC);
```

Reading/Writing dynamical properties:

```
PHALCON_INIT_VAR(language);
ZVAL_STRING(language, "English", 1);

PHALCON_INIT_VAR(property);
ZVAL_STRING(property, "language", 1);

PHALCON_INIT_VAR(employee);
object_init(employee);

// $employee->$property = $language
phalcon_update_property_zval_zval(employee, property, language TSRMLS_CC);

// $user_language = $employee->$property
PHALCON_INIT_VAR(user_language);
phalcon_read_property_zval(&user_language, employee, property, PH_NOISY_CC);
```

Reading/Writing static properties:

```
//Updating a static member with a string zval
PHALCON_INIT_VAR(greeting);
ZVAL_STRING(greeting, "hello world", 1);
phalcon_update_static_property(SL("phalcon\\some\\component"), SL("_someString"),␣
↪greeting TSRMLS_CC);

//Updating a static member with a long zval
PHALCON_INIT_VAR(number);
ZVAL_LONG(number, 150);
phalcon_update_static_property(SL("phalcon\\some\\component"), SL("_someInteger"),␣
↪number TSRMLS_CC);

//Reading a static member
PHALCON_OBSERVE_VAR(number);
phalcon_read_static_property(&number, SL("phalcon\\some\\component"), SL("_someInteger
↪") TSRMLS_CC);
```

Calling Methods

As seen when calling functions, in Phalcon the methods are called in the PHP userland. Thanks to this, a Phalcon user can generate a backtrace and know exactly which components are involved in a given task. They also allows users to replace Phalcon components by PHP classes of their own. Additionally, like everything else we've seen, the way to make calls is familiar to PHP developers.

Instance methods

```
//Define the connection DSN
PHALCON_INIT_VAR(dsn);
ZVAL_STRING(dsn, "mysql:host=localhost;username=root;password=secret;dbname=some", 1);

//Get the PDO class entry
pdo_class_entry = zend_fetch_class(SL("PDO"), ZEND_FETCH_CLASS_AUTO TSRMLS_CC);

//Create a PDO instance passing the dsn to the constructor
```

```
PHALCON_INIT_VAR(pdo);
object_init_ex(pdo, pdo_class_entry);
PHALCON_CALL_METHOD_PARAMS_1_NORETURN(pdo, "__construct", dsn);

//Create a SQL statement
PHALCON_INIT_VAR(sql_statement);
ZVAL_STRING(sql_statement, "SELECT * FROM robots", 1);

//Call the "exec" method in the pdo object passing the sql_statement
PHALCON_INIT_VAR(success);
PHALCON_CALL_METHOD_PARAMS_1(success, pdo, "exec", sql_statement);
```

Static methods

```
PHALCON_INIT_VAR(some_variable);
ZVAL_STRING(some_variable, "invoices_detail", 1);

//Calling Phalcon\Text::camelize("invoices_detail")
PHALCON_INIT_VAR(camelized);
PHALCON_CALL_STATIC_PARAMS_1(camelized, "phalcon\\text", "camelize", some_variable);
```

Classes

As an object oriented framework, Phalcon is mostly composed of classes. The classes are organized into namespaces. The following are the steps to export a variable in the extension and also make it available for other classes inside the framework.

Registering methods and its arguments

In the dev/php_phalcon.h register in the first part of the file the pointer to the zend class entry. Let's pretend we're adding a Phalcon\Auth to the framework:

```
extern zend_class_entry *phalcon_auth_ce;
```

Then, we can add the methods prototypes, it's neccesary to add all the methods that compose our class:

```
PHP_METHOD(Phalcon_Auth, __construct);
PHP_METHOD(Phalcon_Auth, getIdentity);
PHP_METHOD(Phalcon_Auth, auth);
```

Later in the same file add the information of the arguments of each method. For example, let's define the class constructor only takes two arguments and they're mandatory:

```
ZEND_BEGIN_ARG_INFO_EX(arginfo_phalcon_auth__construct, 0, 0, 2)
        ZEND_ARG_INFO(0, adapter)
        ZEND_ARG_INFO(0, options)
ZEND_END_ARG_INFO()
```

Now let's attach methods to the class to which they belong:

```
PHALCON_INIT_FUNCS(phalcon_auth_method_entry){
        PHP_ME(Phalcon_Auth, __construct, arginfo_phalcon_auth__construct, ZEND_ACC_
↪PUBLIC)
        PHP_ME(Phalcon_Auth, getIdentity, NULL, ZEND_ACC_PUBLIC)
        PHP_ME(Phalcon_Auth, auth, NULL, ZEND_ACC_PUBLIC)
        PHP_FE_END
};
```

All this happens at dev/php_phalcon.h, if you check that file you will see that everything is registered right there.

Implementing a method

Each class has its own file .c file, in the case of Phalcon\Auth file would be dev/auth.c:

```
//PHP_METHOD(class_name, method_name)
PHP_METHOD(Phalcon_Auth, __construct){

        zval *adapter_name, *options = NULL;

        //Start a memory frame
        PHALCON_MM_GROW();

        //Receive the method parameters
        if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &adapter_name, &
↪options) == FAILURE) {
                PHALCON_MM_RESTORE();
                RETURN_NULL();
        }

        //Release the memory used
        PHALCON_MM_RESTORE();
}
```

With the above code we create the constructor of the class Phalcon\Auth, a method is defined with the macro PHP_METHOD, first we put the class name and then the name of the method, although Phalcon uses namespaces, class names have _ instead of \:

```
PHP_METHOD(Phalcon_Auth, __construct){
```

If the method has parameters we receive them using zend_parse_parameters:

```
if (zend_parse_parameters(ZEND_NUM_ARGS() TSRMLS_CC, "zz", &adapter_name, &options)␣
↪== FAILURE) {
        PHALCON_MM_RESTORE();
        RETURN_NULL();
}
```

If we do not receive the correct number of parameters will result in an error message. You see, there's an argument "zz" to receive the parameters, this indicates the type of data received and the number of them. In the above example that means that the method is receiving two parameters. If they were three zval then it should be "zzz".

Then the variables are received in respective order: &adapter_name, &options

Throwing Exceptions

When you throw an exception using the Phalcon API, the current flow of execution will be stopped, returning to the last PHP code block when a Phalcon method where called.

There are two ways to throw exceptions, the first, when the exception object only receives a string as parameter:

```
PHALCON_THROW_EXCEPTION_STR(phalcon_exception_ce, "Hey this is an exception");
```

Or building the exception manually and then throwing it:

```
PHALCON_INIT_VAR(exception_message);
PHALCON_CONCAT_SVS(exception_message, "Unable to insert into ", table, " without data
↪");

PHALCON_INIT_VAR(exception);
object_init_ex(exception, phalcon_db_exception_ce);

//The exception constructor must be manually called
PHALCON_CALL_METHOD_PARAMS_1_NORETURN(exception, "__construct", exception_message, PH_
↪CHECK);
phalcon_throw_exception(exception TSRMLS_CC);
return;
```

Compilation on Windows

Follow these instructions to build Phalcon in your Windows system. This guide is tested on Windows XP and Windows 7.

The Process Idea

Compiling Phalcon is performed as a part of PHP compilation. This is done in order to ensure, that Phalcon is built with exactly the same defines and options as PHP.

Before you Begin

Building Phalcon on Windows will require four things

- A properly set up build environment, including a compiler with the right SDK's and some binary tools used by the build system
- Prebuilt libraries and headers for third party libraries that PHP uses in the correct location
- The PHP source
- The Phalcon source

The Build Environment

This is the hardest part of the PHP windows build system to set up and will take up a lot of space on your hard drive - you need to have several GB of space free. Requirements

- Microsoft Visual C++, PHP officially supports building with Visual C++ 6.0 or with Visual C++ 9 (also known as Visual C++ 2008 just to be confusing). You can use the Express versions as well. MinGW and other compilers are NOT supported or even known to work. For more information and how to get the compiler see the supported versions.
- The correct Windows SDK or Platform SDK to match your compiler. see this page for the supported versions
- Various tools, see http://windows.php.net/downloads/php-sdk/ for binary versions of them

Setup Quick 'n' easy

- get visual studio 2008 (no matter what version - express, pro or others; all should work) and install it. Download: http://www.microsoft.com/visualstudio/en-us/products/2008-editions/express
- get and install windows sdk 6.1. Download http://www.microsoft.com/en-us/download/details.aspx?id=11310
- get a php 5.3 snapshot (do not extract yet!) Download: http://windows.php.net/download/
- get Phalcon source code: https://github.com/phalcon/cphalcon
- create the folder "c:\php-sdk"
- unpack the binary-tools.zip archive (http://windows.php.net/downloads/php-sdk/) into this directory, there should be one sub-directory called "bin" and one called "script"
- open the "windows sdk 6.1 shell" (it's available from the start menu group) and execute the following commands in it:

```
setenv /x86 /xp /release

cd c:\php-sdk\

bin\phpsdk_setvars.bat

bin\phpsdk_buildtree.bat php53dev
```

- now extract the snapshot from 3) to C:\php-sdk\php53dev\vc9\x86 with your favourite unpacker (winrar should handle it) so that the following directory gets created: C:\php-sdk\php53dev\vc9\x86\php5.3-xyz
- in the same directory (C:\php-sdk\php53dev\vc9\x86) there is a "deps" folder, extract any of your required libraries inside that folder (see http://wiki.php.net/internals/windows/libs) but make sure their top-level contains /include and /lib (some of them have an extra directory level in there)
- put prepared Phalcon source code to C:\php-sdk\php53dev\vc9\x86\php5.3-xyz\ext\phalcon folder. The prepared Phalcon source is located inside the "build*[type]*" folder of the Phalcon repository, where *type* can be "safe", "32bits" or "64bits". For more information on how to chose proper type or regenerate that source code (which is needed for development versions of Phalcon) read the file "build\README.md" in the Phalcon repository.
- run in the windows-sdk-shell:

```
cd C:\php-sdk\php53dev\vc9\x86\php5.3-xyz

buildconf
```

to get an overview of the compiling flags:

```
configure --help
```

create your configure command:

```
configure --disable-all --enable-cli --enable-phalcon=shared

nmake
```

The php_phalcon.dll after the compilation is located at: C:\php-sdk\php53dev\vc9\x86\php5.3-xyz\Release_TS\

This guide is based on this another guide: https://wiki.php.net/internals/windows/stepbystepbuild

License

Phalcon Framework and the Phalcon API is brought to you by the Phalcon Team! [Twitter - Google Plus - Github]