

URL of the Article: http://entwickler.com/itr/online_artikel/psecom,id,382,nodeid,114.html

Issue: 04.2003

Zend Engine 2 - Internals Exposed

Behind the scenes of PHP5's revamped object model.

Harald Radi

ZendEngine2, the core of the upcoming PHP 5 version, not only has a totally revised userland object model but also a completely redesigned extension API. The new API makes it possible to quickly write simple OO extensions and also to hook deeply into the engine. In this article you will learn about the hooks available in the new Zend Engine and how to use them to overload various object manipulation actions.

PHP5 and the ZendEngine 2

At the time of writing this article PHP 5 is still at the alpha stage. Although feature complete the internal structures are not yet fully adapted and may still change. Nevertheless the material in this article gives an overview about the concepts in the new Zend Engine which definitely won't change. Just remember that one or another API function could have a parameter more or less at the time you use it, but it definitely won't be too hard to figure that out yourself.

What's new compared to PHP 4

This article only covers the parts of the Zend Engine that are interesting for extension developers. Explaining the whole list of changes would make up another article, thus non-relevant parts are not covered in detail.

Writing extensions in general is very much the same as for PHP 4, only the object overloading API has changed very drastically. OO extensions written for PHP 4 will not compile with PHP 5 anymore and the completely different concept requires that old extensions have to be rewritten from scratch.

Improved object handling: One of the major changes is how objects are treated by the new Zend Engine. In PHP4 dealing with objects was very cumbersome. You really had to take care whether to pass an object by value or explicitly by reference. Furthermore it was impossible to assign an object reference to a passed-by-reference function parameter.

Consider the following example:

```
<?php
/* ... */
```

```
function wed($bride, $groom)
{
    if ($bride->setHusband($groom)
    && $groom->setWife($bride)) {
        return true;
    } else {
        return false;
    }
}
```

```
wed($joanne, $joe);
print areMarried($joanne, $joe);
?>
```

One would expect Joanne and Joe to stay married after their wedding (at least for a while), but a check with `areMarried()` proves that they divorced immediately after their wedding. Although unexpected, the reason is simple. The function `wed()` does not see `$joanne` and `$joe` themselves but only copies of them. These copies are actually married, but they die as soon as the function returns, changes to objects in PHP 4 were therefore not persistent.

In PHP 5 objects are not copied anymore, instead a unique object handle is passed around. That handle is used by the engine to look up the actual object in the global object store, thus there is exactly one instance of an object except when it is explicitly cloned. Objects in PHP 5 are comparable with resources where the `zval` container also only holds the identifier and not the resource itself. The change of the object treatment should not have any serious implications in userland as this seems to be the more natural way of treating objects and is also quite common in other languages.

Improved object dereferencing: In PHP 4 it was not possible to directly dereference objects returned from methods or functions. In PHP 5, finally, code like `$object->method()->method()` or even `$object->method()->member->method()` will work. It will also be possible to directly dereference objects returned from function calls. This feature hopefully leads to better looking and more straightforward code. It can prevent certain programming errors and is also the natural syntax when interfacing with Java or COM objects.

Object cloning: In PHP 4 it was not up to the user to decide whether an object is duplicated or not. Whenever the engine had the impression that it should duplicate an object it made a bitwise copy of the underlying memory structure resulting in an identical replica of the originating object. Now, due to the revamped object handling described earlier, an object will never be duplicated by the engine automatically. To give the user the ability to duplicate an object intentionally the `__clone()` method was introduced.

The following code will create an exact clone of an object. The clone will contain the same properties initialized to the same values as in the original object, but will be an independent instance with no other relation to the original object besides its initial similarity:

```
<?php
$foo = new Bar();
$baz = $foo->__clone();

$baz->member = 4321; /* will not affect $foo */
?>
```

This method can be overloaded in userland as well as directly in an extension, so instead of copying everything only the relevant parts of an object can be copied.

An extension has to reopen resource handles in case the object holds something like a file pointer or a database connection. This is very important because otherwise the handle would be invalid whenever one of the two clones is destructed. The other instance would still be valid, but its underlying resource wouldn't be, PHP would probably then crash when trying to access that instance.

Object destructors: The lack of the ability to define a destructor for an object was criticised very often in the past.

Destructors are useful for cleaning up temporary files or resources, to log debug messages and lots of other things. PHP 5 finally presents a solution, destructors are now part of the language specification. When the last reference to an object is destroyed the object's destructor is called before the object is freed from memory. Unlike PEAR destructors, which are called at request shutdown (and therefore aren't real destructors), PHP 5 destructors are called immediately when an object gets destroyed.

PHP 5 provides no way to explicitly destruct an object, thus just like in Java, you have to ensure that all references are deleted when you don't need an object anymore and that you don't keep any references by accident, e.g. in an array or such. Important to know is that due to the nature of PHP the destructor might not always be called, there are still a few exceptions. For example, when a fatal error occurs the Engine will bail out without cleaning up everything properly.

Interfaces: As a way to expose multiple functionalities by a class while circumventing multiple inheritance, interfaces were implemented. This is a common methodology, well established in languages like Java or C#. Interfaces are a way to derive a class from one or more types without inheriting the type's implementation like you would if you derive from a base class.

```
<?php
interface MyInterface {
public function MyMethod();
}

class MyClass implements MyInterface {
public function MyMethod() {
/* ... */
}
}
?>
```

The interface MyInterface specifies that each class which implements it must have a method *MyMethod()*, but the actual implementation of that method is up to the class itself. A function expecting an argument to be an instance of a given interface can now safely assume that the method *MyMethod()* is there, it does not get any information about the actual implementation though. Interfaces are especially useful in conjunction with the recently introduced type hints that enable a developer to enforce that a function parameter is of a given class or implements a specific interface. Just like in a strict typed language, you can specify the type of a parameter in the function signature and the Zend Engine ensures that the function only gets called when the parameters match the expected types, otherwise it will bail out with a fatal error.

```
function bar(MyInterface $foo) {
/* $foo is an instance of MyInterface */
/* ... */
}
```

Namespaces: Due to PHP's popularity a lot of reusable code was written in the past and is now publicly available via various online code bases. One of the best known projects in this area is probably the PEAR project. The huge amount of functions and classes provided by such libraries made it increasingly difficult to avoid symbol name collisions. Namespaces in PHP 5 provide a way to avoid these collisions and manage those libraries in distinct namespaces. Unlike other languages, PHP 5 does not support nested namespaces, each namespace exists in the global scope and has no relation to any of the other namespaces. It is also not possible to instantiate a namespace just like in a few other languages. Beside the characters allowed in variable and function names, namespace names may also contain the `:` character as a separator. Even if `:` implies a semantic meaning at first glance it is really only syntactic sugar and a punctuation character like any other.

```
?php
namespace Foo {
const aConstant = 'someValue';
var $aVariable;
```

```
class Bar
{
/* ... */
}
```

```
function aFunction()
{
/* ... */
}
}
```

```
echo Foo::aConstant;
Foo::$aVariable = 'someValue';
```

```
$obj = new Foo::Bar();
```

```
Foo::aFunction();
```

```
?>
```

Exception handling: Exception handling is introduced with PHP 5 and is a very convenient tool when used correctly. That means it should only be used for handling errors and not for controlling the regular program flow.

The implementation of *try/throw/catch* looks similar to other programming languages. Any code which is inside a *try/catch* block can *throw* an exception and pass that exception to the closest catch block which catches exceptions of that type.

```
<?php
```

```
try {
    throw new Exception('Hello');
} catch (Exception $exception) {
    echo $exception;
}
?>
```

Visibility: PHP 5 introduces the PPP keywords *public/protected/private* to control the visibility of certain class members. Note that for performance reasons, no error message is emitted if illegal access to a private or protected member variable is attempted. If no keyword is specified the default visibility is public to ensure backwards compatibility. Therefore old code should run without any modifications.

Internal object representation

Internally an object is represented by a *zend_object_value* structure (Listing 1), which contains the object handle and a pointer to a *zend_object_handlers* structure (Listing 4) which will be described later. The handle is an unsigned integer which has to be unique to the object's class. That means it has to be clear enough for you to find the object's underlying data based on that handle.

Listing 1

```
struct _zend_object_value {
    zend_object_handle handle;
    zend_object_handlers *handlers;
};
```

The *zend_object_value* structure is part of the *zvalue_value* union (Listing 2) which is the value member of the well known *zval* structure, the internal representation of a variable in PHP. Now it should be clear to you what will happen whenever a *zval* container gets copied, only the unique handle and the pointer to the list of handlers will be copied, the object data itself remains untouched.

Listing 2

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

Listing 3

```
struct _zval_struct {
    /* Variable information */
    zvalue_value value;
    zend_uint refcount;
    zend_uchar type;
    zend_uchar is_ref;
};
```

The *zend_object_handlers* structure (Listing 4) contains all the existing hooks that can be used to react to object manipulations in userland. Don't be scared right now, you only have to implement a few of the hooks. The remaining hooks are either completely optional or have default handlers. Compared to the limited abilities in PHP 4 this is a huge improvement and also reflects the necessity that most parts of existing OO extensions have to be rewritten when ported from PHP 4 to PHP 5.

Listing 4

```
typedef struct _zend_object_handlers {
    /* general object functions */
    zend_object_add_ref_t add_ref;
    zend_object_del_ref_t del_ref;
    zend_object_delete_obj_t delete_obj;
    zend_object_clone_obj_t clone_obj;

    /* individual object functions */
    zend_object_read_property_t read_property;
    zend_object_write_property_t write_property;
    zend_object_get_property_ptr_t get_property_ptr;
    zend_object_get_property_zval_ptr_t get_property_zval_ptr;
    zend_object_get_t get;
    zend_object_set_t set;
    zend_object_has_property_t has_property;
    zend_object_unset_property_t unset_property;
```

```
zend_object_get_properties_t get_properties;
zend_object_get_method_t get_method;
zend_object_call_method_t call_method;
zend_object_get_constructor_t get_constructor;
zend_object_get_class_entry_t get_class_entry;
zend_object_get_class_name_t get_class_name;
zend_object_compare_t compare_objects;
} zend_object_handlers;
```

The structure all hooks and their signature are discussed here in detail.

add_ref

```
typedef void (*zend_object_add_ref_t)(zval *object TSRMLS_DC);
```

This handler is called whenever a new zval referring to that object is created. It does not create a new object, both zvals still refer to the same object. Example:

```
<?php
$foo = new Bar();
/* causes an add_ref call */
$baz = $foo;
/* causes another add_ref call */
?>
```

del_ref

```
typedef void (*zend_object_del_ref_t)(zval *object TSRMLS_DC);
```

This handler is called whenever a reference to that object is destroyed. This does not mean that the object should be destructed, only that one less zval points to that object. Example:

```
<?php
$foo = new Bar();
$baz = $foo;
$baz = null;
/* causes a del_ref call */
?>
```

delete_obj

```
typedef void (*zend_object_delete_obj_t)(zval *object TSRMLS_DC);
```

This handler is called after all references to that object are destroyed. It should clean up any resources and free the underlying data structures. After this function has been called the object's handle is no longer valid. Example:

```
<?php
$foo = new Bar();
$foo = null;
/* causes a delete_obj call after the del_ref call */
?>
```

clone_obj

```
typedef zend_object_value (*zend_object_clone_obj_t)(zval *object TSRMLS_DC);
```

This handler is called when a new object identical to an old one should be created. Unlike PHP4 this only happens when the user explicitly clones an object with the `__clone()` method. The handler is responsible for copying the underlying data and reinitializing resources a second time. Take care, if you only copy resource handles together with your underlying data structure these resources will get freed twice which will most probably result in a segmentation fault. Example:

```
<?php
$foo = new Bar();
$bar = $foo->__clone();
/* causes a clone_obj call */
?>
```

get_property_zval_ptr

```
typedef zval **(*zend_object_get_property_zval_ptr_t)(zval *object, zval *member TSRMLS_DC);
```

This handler is used to obtain a pointer to a modifiable zval for operations like `+=` or `++`. This should be used only if your object model stores properties as real zvals that can be modified from outside (e.g. standard PHP objects). Otherwise this handler should be `NULL` and the engine will use `read_property` and `write_property` instead. Example:

```
<?php
$foo = new Bar();
$foo->baz = 123;
/* causes a get_property_zval_ptr call */
$baz = $foo->baz;
/* causes another get_property_zval_ptr call */
?>
```

get_property_ptr

```
typedef zval **(*zend_object_get_property_ptr_t)(zval *object, zval *member TSRMLS_DC);
```

This handler is used to create a pointer to the property of the object, for future read-write access via the get and set handler. If your object's properties are stored as zval*, return the real pointer where the property is stored. If they aren't, the best way is to create a proxy object that is responsible for managing that property. How to create proxy objects is described later in this article. If you implemented the `get_property_zval_ptr` callback function you can set the `get_property_ptr` callback to the same function.

If you don't want to implement property referencing for your objects, you can set this handler to `NULL`. Example:

```
<?php
$foo = new Bar();
$baz = &$foo->baz;
/* causes a get_property_ptr call */
?>
```

read_property

`typedef zval (*zend_object_read_property_t)(zval *object, zval *member TSRMLS_DC);`

This handler is called when an object's property is requested. The returned value is read-only and not meant to be changed. This handler will only be called if `get_property_zval_ptr` is set to `NULL`. Example:

```
<?php
$foo = new Bar();
$baz = $foo->baz;
/* causes a read_property call */
?>
```

`write_property`

`typedef void (*zend_object_write_property_t)(zval *object, zval *member, zval *value TSRMLS_DC);`

This handler is used to assign property variables or to change them in operations like `+=` or `++` unless `get_property_zval_ptr` is defined. Example:

```
<?php
$foo = new Bar();
$foo->baz = 123;
/* causes a write_property call */
?>
```

`get`

`typedef zval* (*zend_object_get_t)(zval *property TSRMLS_DC);`

This handler is used to get an object value, most probably in combination with the result of the `get_property_ptr` function or when converting object value to one of the basic types.

Get and set handlers are used when engine needs to access the object as a value. Example:

```
<?php
$foo = new Bar();
$baz = &$foo->bar;
echo $baz;
/* causes a get call */
?>
```

`set`

`typedef void (*zend_object_set_t)(zval **property, zval *value TSRMLS_DC);`

The set handler is the counterpart to the `get` handler and is responsible for writing a value directly to an object. Example:

```
<?php
$foo = new Bar();
$baz = &$foo->baz;
$baz = 123;
/* causes a set call */
?>
```

`has_property`

`typedef int (*zend_object_has_property_t)(zval *object, zval *member, int check_empty TSRMLS_DC);`

This handler checks if that object has a certain property set and optionally if it is empty or not.

`unset_property`

`typedef void (*zend_object_unset_property_t)(zval *object, zval *member TSRMLS_DC);`

This handler is used to remove the value for a specified property of that object.

`get_properties`

`typedef HashTable* (*zend_object_get_properties_t)(zval *object TSRMLS_DC);`

This handler should return the list of properties of the object as a hash of zvals. Due to the unfinished reflection API, this and its related callback function are still likely to change.

`get_method`

`typedef union _zend_function* (*zend_object_get_method_t)(zval *object, char *method, int method_len TSRMLS_DC);`

This handler is used to find a method description by name. It should set the right type, function name and parameter mask for the method. If the type is `ZEND_OVERLOADED_FUNCTION`, the method is called via the `call_method` handler, otherwise the returned function is called directly. Example:

```
<?php
$foo = new Bar();
$foo->bar();
/* causes a get_method call */
?>
```

`call_method`

`typedef int (*zend_object_call_method_t)(char *method, INTERNAL_FUNCTION_PARAMETERS);`

This handler is a dispatcher function for any method call that couldn't be resolved by `get_method`. It is called with the method name and the full argument list of the actually called method. Parameters are passed like to any other Zend internal function.

`get_constructor`

`typedef union _zend_function* (*zend_object_get_constructor_t)(zval *object TSRMLS_DC);`

`get_constructor` performs the same operation as `get_method`, but for the object constructor. It returns a pointer to a `_zend_function` union that points to a function which should be called upon object construction.

`get_class_entry`

`typedef zend_class_entry* (*zend_object_get_class_entry_t)(zval *object TSRMLS_DC);`

This handler returns the `zend_class_entry` structure for the object in case it's a class object and not an ad-hoc object without a class.

`get_class_name`

`typedef int (*zend_object_get_class_name_t)(zval *object, char **class_name, zend_uint *class_name_len, int parent TSRMLS_DC);`

This handler is used to retrieve the class name of the object.

`compare_objects`

```
typedef int (*zend_object_compare_t)(zval *object1, zval *object2 TSRMLS_DC);
```

This handler is used to compare objects of the same class. This is used with the == operation whereas === compares objects by handles, i.e. it returns true if and only if it's really the same object. Note that objects from different object types cannot be compared.

Ad-Hoc objects

Unlike PHP 4 you can now create objects without a corresponding class entry. This is useful for returning objects from an extension function which mainly contains data without methods. Creating ad-hoc objects is probably quicker and much simpler, but you have to be aware that such objects can never be instantiated with the *new* operator, they can only be used as a return value.

You can create an ad-hoc object by assigning a handle and a pointer to the object handler's structure to a zval container and setting its type to *IS_OBJECT*.

Class objects

Creating class objects is a bit more difficult, although in return you gain a lot more functionality. They can be created with the *new* operator just like userland objects and they are fully supported by PHP's reflection API.

For each class you want to export you have to create a *zend_class_entry* structure, which is shown in Listing 5. Most members of that structure will be initialized by the *INIT_CLASS_ENTRY* macro, although you need to watch out for a few things though.

Listing 5

```
struct _zend_class_entry {
    char type;
    char *name;
    zend_uint name_length;
    struct _zend_class_entry *parent;
    int refcount;
    zend_bool constants_updated;
    zend_uint ce_flags;

    HashTable function_table;
    HashTable default_properties;
    HashTable properties_info;
    HashTable class_table;
    HashTable *static_members;
    HashTable constants_table;
    zend_function_entry *builtin_functions;
    struct _zend_class_entry *ns;

    union _zend_function *constructor;
    union _zend_function *destructor;
    union _zend_function *clone;
    union _zend_function *__get;
    union _zend_function *__set;
    union _zend_function *__call;

    /* handlers */
    zend_object_value (*create_object)
    (zend_class_entry *class_type TSRMLS_DC);

    zend_class_entry **interfaces;
    zend_uint num_interfaces;

    char *filename;
    zend_uint line_start;
    zend_uint line_end;
    char *doc_comment;
    zend_uint doc_comment_len;
};
```

The *INIT_CLASS_ENTRY* macro takes the class entry structure as the first and the class name as the second parameter.

The third parameter is optional and can be set to *NULL*. If set, it should point to a *function_entry[]* array which will be used as a list of class member functions.

```
INIT_CLASS_ENTRY(class_container, class_name, functions)
```

After initializing the class entry you need to assign the internal constructor function which is responsible for allocating the necessary memory for the object and the underlying data before it can be accessed. This is not the actual constructor of the object, you can think of it as the object allocator.

```
class_entry.create_object = ext_objects_new;
```

This is the minimum initialization for a class entry necessary to successfully create objects from it.

To make the engine aware of that class entry you need to register it with one of the two following functions. The former is used for basic registrations, the latter is used if you want to derive your class entry from an existing one.

```
ZEND_API zend_class_entry *zend_register_internal_class(zend_class_entry *class_entry TSRMLS_DC);
```

```
ZEND_API zend_class_entry *zend_register_internal_class_ex(zend_class_entry *class_entry, zend_class_entry
*parent_ce, char *parent_name TSRMLS_DC);
```

The object store

Since objects are no longer stored in the zval container itself, Zend introduced the object store to keep the actual object data and hand out unique object handles. The object store API mainly consists of three functions which are discussed below.

The first step is to register the object data at the object store. This is done with the `zend_objects_store_put` function, which expects pointers to a destructor and a clone function for that data. These pointers are optional and can be set to `NULL`. The function returns a unique handle that can be directly assigned to the `zend_object_value` structures handle member.

```
ZEND_API zend_object_handle zend_objects_store_put(void *object, zend_objects_store_dtor_t dtor, zend_objects_store_clone_t clone TSRMLS_DC);
```

The `zend_object_store_get_object` function can be used to retrieve the stored data back from the object store based on the `zval` containing the unique object handle.

```
ZEND_API void *zend_object_store_get_object(zval *object TSRMLS_DC);
```

And last but not least, `zend_objects_store_delete_obj` exists to tell the object store to remove a specific instance and clean up the allocated resources. This function will be called from an `objectsdelete_obj` handler.

```
ZEND_API void zend_objects_store_delete_obj(zval *object TSRMLS_DC);
```

Object proxying

As mentioned earlier it is possible for overloaded objects to return references to overloaded properties with the help of the engine's proxying capabilities. The Zend Engine exports a function that automatically creates a proxy for you:

```
ZEND_API zval **zend_object_create_proxy(zval *object, zval *member TSRMLS_DC)
```

This function expects the object and the name of the member that should be referenced as a `zval` container and directly returns the proxy object. The proxy object's `zend_object_handlers` structure differs completely from that of standard objects. It implements nothing but the `get` and `set` handler (Listing 6).

You can simply set the `get_property_ptr` callback to `zend_object_create_proxy` and the engine will take care of proxying for you.

Listing 6

```
static zend_object_handlers zend_object_proxy_handlers = {  
ZEND_OBJECTS_STORE_HANDLERS,
```

```
NULL, /* read_property */  
NULL, /* write_property */  
NULL, /* get_property_ptr */  
NULL, /* get_property_zval_ptr */  
zend_object_proxy_get, /* get */  
zend_object_proxy_set, /* set */  
NULL, /* has_property */  
NULL, /* unset_property */  
NULL, /* get_properties */  
NULL, /* get_method */  
NULL, /* call_method */  
NULL, /* get_constructor */  
NULL, /* get_class_entry */  
NULL, /* get_class_name */  
NULL /* compare_objects */  
};
```

Throwing Exceptions

Exceptions can not only be thrown in userspace but also from within PHP functions in C extensions. To be able to throw exceptions you first need to define a class entry for the class you later want to throw as an exception. This is best done somewhere in your extension's `PHP_MINIT_FUNCTION(ext)` function:

```
/* initialization */
```

```
zend_class_entry _ex_class_entry, *ex_class_entry;
```

```
INIT_CLASS_ENTRY(_ex_class_entry, "exception", NULL);
```

```
_ex_class_entry.create_object = zend_objects_new;
```

```
ex_class_entry = zend_register_internal_class(&_ex_class_entry TSRMLS_CC);
```

```
/* ... */
```

Now whenever a function in your extension does not terminate as expected you can throw an exception by executing the following code:

```
/* throw the exception */
```

```
zval *ex;
```

```
zend_object *ex_i;
```

```
ALLOC_ZVAL(ex);
```

```
INIT_PZVAL(ex);
```

```
Z_TYPE_P(ex) = IS_OBJECT;
```

```
ex->value.obj = zend_objects_new(&ex_i, ex_class_entry TSRMLS_CC);
```

```
EG(exception) = ex;
```

```
return FAILURE;
```

The `zend_objects_new` function is the allocator function of a standard PHP object and returns the `zend_object_value` of an ordinary PHP object - not an overloaded one.

Basic Example

A very small example that exports a meaningless class to userland can be found on the CD shipped with this issue. The class can be instantiated with the `new` operator and is reflected in all the means which PHP provides.

Example with Namespaces

If you exchange the `PHP_MINIT_FUNCTION(ext)` function from the `ext.c` file contained on your magazine CD with the one from Listing 7 your class will not be exported to the global scope but to a namespace scope, here to the namespace `Foo`. The example shows that by manually setting the `ce_flags` property of the `zend_class_entry` structure, you can change the class to be either an abstract class or an interface. Such a class would therefore not be instantiatable. This is sometimes very useful, e.g. if you want to export many different classes that are derived from a common base class you can protect the base class from being directly instantiated.

Listing 7

```
/* {{{ PHP_MINIT_FUNCTION(ext)
*/
PHP_MINIT_FUNCTION(ext)
{
    zend_namespace_ns, *ns;
    zend_class_entry_ext_class_entry;

    INIT_NAMESPACE(_ns, "Foo");
    ns = zend_register_namespace(&_ns TSRMLS_CC);

    INIT_CLASS_ENTRY(_ext_class_entry, "ext", ext_class_functions);
    _ext_class_entry.create_object = ext_objects_new;
    /* _ext_class_entry.ce_flags |= ZEND_ACC_INTERFACE; */
    /* _ext_class_entry.ce_flags |= ZEND_ACC_ABSTRACT_CLASS; */

    ext_class_entry = zend_register_internal_ns_class(&_ext_class_entry, NULL, ns, NULL TSRMLS_CC);

    /* copy the standard object handlers to
    you handler table */
    memcpy(&ext_object_handlers, zend_get_std_object_handlers(), sizeof(zend_object_handlers));
    /* replace the default clone handler with
    the object store clone handler */
    ext_object_handlers.clone_obj = zend_objects_store_clone_obj;

    return SUCCESS;
}
/* }}} */
```

Harald Radi is working as a consultant and software designer at nme (<http://www.nme.at>). He is the author of the `php-milter SAPI` and (co)maintainer of the `ADT`, `COM` and `RPC` extension. Beside additional work on other extensions he also contributes to the Zend Engine.

Links and Literature

- ▣ Zend Engine Version 2.0, Feature Overview and Design, Zend, www.zend.com/engine2/ZendEngine-2.0.pdf
- ▣ The Object-Oriented Evolution of PHP, Zeev Suraski, www.devx.com/webdev/Article/10007
- ▣ OBJECTS2_HOWTO, cvs.php.net/co.php/ZendEngine2/OBJECTS2_HOWTO
- ▣ ZEND_CHANGES, cvs.php.net/co.php/ZendEngine2/ZEND_CHANGES
- ▣ The namespaces RFC, Stig Sæther Bakken, cvs.php.net/co.php/ZendEngine2/RFCs/002.txt
- ▣ Thanks to all the PHP folks, especially Zeev Suraski and Marcus Börger, for answering various questions. Comments and Questions: forum.php-mag.net/4/3/zend2

© 2002 Software & Support Verlag GmbH. Reproduction has to be permitted by the publisher. All brands are usually registered trademarks of companies and organisations.

Questions? ... on the products of the Software & Support Verlag: info@entwickler.com

... on this website: webmaster@entwickler.com