Lucas Pereyra Follow
Apr 24, 2022 · 4 min read · List

PHP __sleep And __wakeup Magic Methods: How And When To Use Them?

PHP __sleep and __wakeup methods are *magic methods* (methods that PHP will invoke during special moments or cases) that PHP provides for dealing with serialization/deserialization of class instances (objects) during runtime. Serializing an object basically means creating a *string* representation of that object, so that it can be temporarily stored or saved (e.g. dumping it into a text file) and restored later.

PHP *serialize* function allows you to create a *string* representation of any object that can be restored in the future by means of the *unserialize* function. Furthermore, you can also serialize primitive types such as *ints, floats, arrays, stdClass instances* among others.

PHP serialize and unserialize functions in practice

Let's get into some code examples to help us illustrate how *serialize* and *unserialize* functions work. Take a look at the following snippet:

```php
<?php

class Greeting
{
    private string $firstName;
    private string $lastName;

    public function __construct(string $firstName, string $lastName)
    {
        $this->firstName = $firstName;
        $this->lastName = $lastName;
    }

    public function greet()
    {
        echo "Hello {$this->firstName} {$this->lastName}, how you're doing?" . PHP_EOL;
    }
}
```

PHP Serialization\Greeting.php hosted with ❤ by **GitHub** **view raw**

```
O:8:"Greeting":2:{s:19:"GreetingfirstName";s:4:"Jhon";s:18:"GreetinglastName";s:3:"Doe";
```

PHP Serialization\storage.txt hosted with ❤ by **GitHub** **view raw**

```php
<?php

require_once './Greeting.php';

$greeting = new Greeting('Jhon', 'Doe');
echo $greeting->greet(); // prints "Hello Jhon Doe, how you're doing?"

file_put_contents('storage.txt', serialize($greeting));
```

PHP Serialization\test1.php hosted with ❤ by **GitHub** **view raw**

Here I have a pretty basic class called *Greeting* that's used to greet somebody. Once we've instantiated this class, we can make use of the *greet* method to show a kind greeting through the CLI. Besides that, here I serialize the *$greeting* instance and put that result into the *storage.txt* file. By doing this you may get a strange *string* written into that file. Well, that's our *$greeting* instance *string* representation.

Let's suppose we have another file in the same directory, that runs the following:

```php
<?php

require_once './Greeting.php';

$serialized = file_get_contents('storage.txt');

$greeting = unserialize($serialized);

echo $greeting->greet(); // prints "Hello Jhon Doe, how you're doing?"
```

PHP Serialization\test2.php hosted with ❤ by GitHub view raw

Here I'm just reading the file content and using it to re-create the original *$greeting* instance as a PHP object which can be then invoked the *greet* method. Note that this would be executed in a different PHP script than the first one, so we could think we're communicating these two files by means of the *storage.txt* file and the serialization mechanism: we can store a snapshot of an in-memory object in a storage mechanism (e.g. a text file) from which we can restore it later. Also, note that in both files I needed to include the *Greeting* class definition, since it's necessary not only for creating the original instance, but for re-creating it during the *deserialization*.

Dealing with resources during serialization

PHP's *serialize* function can't create *string* representations of PHP resources. Hence, objects that hold any kind of resource types as private or public members can't be properly serialized and PHP will throw an error or behave rarely if we attempt to do so; like in the following example:

```php
<?php

class FileWrapper
{
    private $file;
    private $size;

    public function __construct(string $filename)
    {
        $this->size = filesize($filename);
        $this->file = fopen($filename, 'r+');
    }

    public function readFile() : string
    {
        return fread($this->file, $this->size);
    }

    public function __destruct()
    {
        fclose($this->file);
    }
}
```

PHP Serialization\FileWrapper.php hosted with ❤ by GitHub view raw

```php
<?php

require_once './FileWrapper.php';

$wrapper = new FileWrapper('test.txt');

echo $wrapper->readFile() . PHP_EOL;

$serialized = serialize($wrapper);

echo $serialized . PHP_EOL;

$unserializedWrapper = unserialize($serialized);

echo $unserializedWrapper->readFile();
```

PHP Serialization\test.php hosted with ❤ by GitHub view raw

When I run this code I get:

```
>Hello world from a simple .txt file
>O:11:"FileWrapper":2{s:17:"FileWrapperfile";i:0;s:17:"FileWrappersi
```

```
ze";i:35;}
>Warning: fread() expects parameter 1 to be resource, int given in
/usr/src/myapp/FileWrapper.php on line 16
>Warning: fclose() expects parameter 1 to be resource, int given in
/usr/src/myapp/FileWrapper.php on line 21
```

This is because the *file* property is being stored as a *resource type,* which makes PHP fail at serializing it. Any result that the PHP resource functions give us, shouldn't be directly serialized when trying to use the *serialize* function.

Okay, so, how do we tell PHP which properties should ignore and which should include when the *serialize* function is being invoked?

Using __sleep and __wakeup magic methods during serialization/deserialization

PHP's *__sleep* magic method should return an array holding all the object's properties that should be included in the serialized version of that object. Properties that are not included in that array, will have a default *null* value in the re-created class instance. PHP will call *__sleep* method before start serializing the object in order to know which properties should be included in the *string* representation.

On the other hand, the *__wakeup* method should contain any initialization code that should be executed once the object has been re-created. Initialization code could include restoring internal used resources, internal object relationships, restoring configuration variables, etc.

With this in mind, we could refactor the *FileWrapper* to make it *serializable*:

```php
<?php

class FileWrapper
{
    private $file = NULL;
    private $filename;

    public function __construct(string $filename)
    {
        $this->filename = $filename;
    }

    public function readFile() : string
    {
        return fread($this->file(), filesize($this->filename));
    }

    private function file()
    {
        if (is_null($this->file)) {
            $this->file = fopen($this->filename, 'r+');
        }
        return $this->file;
    }

    public function __destruct()
    {
        if (!is_null($this->file)) {
            fclose($this->file);
        }
    }

    public function __sleep()
    {
        return ['filename'];
    }
}
```

PHP Serialization\FileWrapper.php hosted with ❤ by GitHub view raw

Here I'm telling PHP to only store the *filename* property in the *string* representation of any *FileWrapper* instance. This, in conjunction with the *file()* access method, ensures that the file resource is only created and used during the current runtime.

To illustrate the usage of *__wakeup*, let's take a look at the following example:

```php
<?php

class TableAccess
{
    private $table;
    private $rowsLimit;
    private $connection = null;

    public function __construct(string $table, int $rowsLimit, $connection)
    {
        $this->table = $table;
        $this->rowsLimit = $rowsLimit;
        $this->connection = $connection;
    }

    public function queryAll()
    {
        $query = "SELECT * FROM {$this->table} LIMIT {$this->rowsLimit}";

        return $this->connection->execute($query)->get();
    }

    public function __sleep()
    {
        return ['table', 'rowsLimit'];
    }

    public function __wakeup()
    {
        $this->connection = DB::connectionInstance();
    }
}
```

PHP Serialization\TableAccess.php hosted with ❤ by **GitHub** **view raw**

Here I have a basic *TableAccess* object that makes use of a *Connection* instance, -which could be a singleton, as usually-, to access a database table and fetch some results. Since the *Connection* instance is likely to be or hold a *PHP resource type,* as if it stored any result of the *mysql_XXX* functions, we wouldn't serialize the *$connection* property (as shown in the *__sleep* method). Having said that, we'll need to initialize its value whenever we get a new fresh instance from the *unserialize* function, and that's what we're doing inside the *__wakeup* method.

string value representation of an in-memory class instance. By storing that string into a file or any other storage mechanism, we can access it later, hence accessing that class instance capabilities. Having said that, we could use the same in-memory object in two (or more) different execution threads (wow).

PHP's *__sleep* magic method should be used to let PHP know which private and public properties of an object should be included in the *string* representation that is obtained when calling the *serialize* function. PHP *resource types* properties should not be retrieved by this method, since PHP will be likely to fail or behave rarely either during serialization or deserialization. Moreover, any properties whose values could be easily calculated or obtained during deserialization, might be not included in the *__sleep*'s result so as to have a shorter *string* representation.

PHP's *__wakeup* magic method should be used to initialize the re-created object once the *unserialize* function has been called. Code that restores or re-creates PHP resource types should be here, as well as code that restores any other kind of necessary properties that were not returned by the *__sleep* method.

PHP Php Development Magic Method Php 7