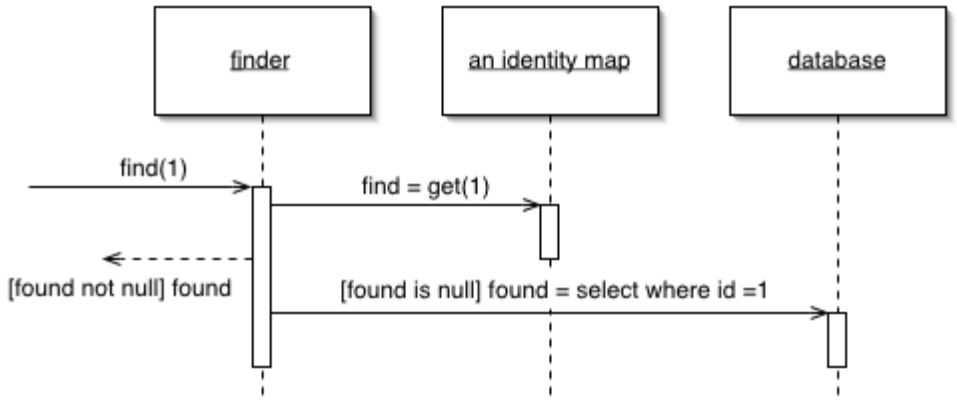
Identity Map et sauvegarde de contexte en PHP

Par Benjamin DELESPIERRE

Introduction

L'une des principales faiblesses de PHP est la perte de contexte entre deux requête: tous les objets instanciés sont détruits à la fin du script. Cela peut s'avérer très contraignant dans le cadre d'application où de gros objets sont générés à chaque appel comme c'est le cas notamment avec des objets modèles qui extraient des données depuis une base de donnée. Le design pattern Identity Map peut s'avérer très utile dans ce cas car il permet de mettre en cache les données pour limiter les accès aux divers média (base de données, fichiers).

D'après le célèbre *P of EAA* de Martin Fowler:
"[Identity Map] Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them."



(Référence : http://martinfowler.com/eaaCatalog/identityMap.html)
Comprenez qu'Identity Map s'assure que chaque objet n'est instancié qu'une seule fois en gardant toutes les instances dans une table de hachage et offre les assesseurs nécessaire pour obtenir les objets.

Dans cet article, nous allons aborder la mise en place du design pattern Identity Map et l'utiliser pour persister nos données entre plusieurs appels.

Ce qu'il faut savoir

Des notions de programmation orientée objet et de persistance en PHP sont impératives pour comprendre cet article. Une connaissance basique des principaux design patterns est souhaitable.
Cet article utilise des exemples pour PHP 5.3 minimum, il est toutefois possible de l'adapter à des version plus anciennes.

Créer une Identity Map

Une Identity Map est basiquement une table de hachage (en PHP nous utiliserons de simples tableaux associatifs) dans laquelle nous allons jeter nos instances. Nous allons pour cela nous servir de tokens (jetons) qui nous permettrons de retrouver nos petits.

L'identity map ne doit être instanciée qu'une et une seule fois dans tout le cycle de la requête, nous allons donc en faire un singleton. Pour cela nous allons nous munir d'une classe abstraite Singleton.

Décrire l'intérêt et l'usage des singleton est en dehors du cadre de cet article. Pour ceux qui ne connaissent pas ce design pattern, vous trouverez toutes les informations nécessaires sur wikipedia. Pour l'instant, retenez juste que le singleton caractérise une classe qui ne peut avoir qu'une seule instance.

```
<?php
interface Identifiable
{
   public static function who ();
}

abstract class Singleton implements Identifiable
{
   protected static $instance;

   abstract protected function __construct ();

   public static function singleton ()
   {
         if (!isset(self::$instance))
         {
               $class = static::who(); // PHP 5.3 late static binding
               self::$instance = new $class;
         }
         return self::$instance;
   }
}
```

La méthode statique *who* est définie dans une interface que nous nomerons Identifiable car

d’autres classes en auront rapidement besoin, vous verrez pourquoi plus loin.

Note: nous avons ici un cas d’usage du fameux Late Static Binding mis en place avec PHP 5.3 qui est utilisée pour choisir la classe appelée dans le cadre de l'héritage de méthodes statiques. Concrètement, cela signifie que la méthode *who* appellée dans la classe Singleton sera appellée dans le contexte de ses filles. Voir les exemples sur le site de php.

Maintenant, nous pouvons créer notre classe IdentityMap. Cette classe permettra la création d’instances d’autres classes (design pattern Factory), nous allons donc créer une interface pour modéliser cela.

```
<?php
interface Factory
{
   public static function factory($class, $args = array());
}

class SessionRegistry extends Singleton implements Factory
{
   const SESSION_KEY = 'instance_map';

   protected $map = array();

   protected function __construct ()
   {
         if (!isset($_SESSION[self::SESSION_KEY]))
               $_SESSION[self::SESSION_KEY] = array();

         $this->map = &$_SESSION[self::SESSION_KEY];
   }

   public function __destruct ()
   {
         $_SESSION[self::SESSION_KEY] = $this->map;
   }

   public static function who () { return __CLASS__; }

   public static function attach (Persistent $instance)
   {
         return self::$instance->map[$instance->getToken()] = $instance;
   }

   public static function attachAll ($instances)
   {
         foreach ($instances as $instance)
         {
               if (is_a($instance, "Persistent"))
```

```
                        self::attach($instance);
                else
                        throw new Exception("instance is not Persistent");
        }
        return self::$instance;
    }

    public static function detach ($obj)
    {
        if (is_string($obj)) // assume this is a token
        {
                self::$instance->map[$obj] = null;
        }
        elseif (is_a($obj, 'Persistent'))
        {
                self::$instance->map[$obj->getToken()] = null;
        }
        else
        {
                throw new BadMethodCallException("Object or string expected, " .
gettype($obj) . " given");
        }
    }

    public static function factory ($class, $args = array())
    {
        if (class_exists($class))
        {
                $reflect = new ReflectionClass($class);
                if ($reflect->isInstantiable() && $reflect->isSubclassOf("Persistent"))
               {
                        $token = $class::generateToken($args);
                        if (isset(self::$instance->map[$token]))
                        {
                                return self::$instance->map[$token];
                        }
                        else
                        {
                                return self::$instance->attach($reflect->newInstanceArgs($args)-
>setToken($token));
                        }
                }
                else
                {
                        throw new Exception("$class is not Persistent or cannot be
instanciated");
                }
        }
        else
        {
```

```
                throw new Exception("$class not found");
        }
    }

    public static function flush ()
    {
        self::$instance->map = array();
        return self::$instance;
    }
}
```

La classe s'instancie une seule fois (au travers de la méthode *singleton*, son constructeur étant protected) et importe les données recues de la variable de session à l'index défini dans sa constante interne SESSION_KEY, ce qui évite les problèmes de colision avec d'autres mécanismes utilisant la variable de session.

Cette classe est très simple, elle permet de créer de nouveaux objets avec la méthode *factory*, d'attacher un ou plusieurs objets avec le méthodes *attach* et *attachAll* et également de les détacher avec *detach*, enfin, elle permet de vider la map avec la méthode *flush*. La méthode statique *who* est définie ici et ne peut pas l'être dans sa mère car si c'était le cas, cette méthode renverrait toujours "Singleton" (au lieu de "IdentityMap"). La preuve par l'exemple:

```
class a { public static function hello () { echo __CLASS__; } }
class b extends a {}
b::hello(); // affiche 'a'

abstract class c { abstract public static function who(); public static function hello () { echo
static::who(); } }
class d extends c { public static function who () { return __CLASS__; } }
d::hello(); // affiche 'd'
```

Le destructeur de cette classe permet, lors que le script se termine, de sauvegarder l'IdentityMap sur la variable de session et de la récupérer au prochain appel via son constructeur, ainsi, les données sont persistantes d'une page à l'autre.

Penchons-nous plus en détail sur la méthode *factory* car c'est là que réside l'aspect le plus important de notre classe:

```
    public static function factory ($class, $args = array())
    {
        if (class_exists($class))
        {
            $reflect = new ReflectionClass($class);
            if ($reflect->isInstantiable() && $reflect->isSubclassOf("Persistent"))
        {
```

```
                        $token = $class::generateToken($args);
                        if (isset(self::$instance->map[$token]))
                        {
                                return self::$instance->map[$token];
                        }
                        else
                        {
                                return self::$instance->attach($reflect->newInstanceArgs($args)-
>setToken($token));
                        }
                }
                else
                {
                        throw new Exception("$class is not Persistent or cannot be
instanciated");
                }
        }
        else
        {
                throw new Exception("$class not found");
        }
  }
```

Cette méthode prends en paramètre le nom de la classe à instancier ainsi que les argument à passer à son constructeur. Son fonctionnement se déroule comme suit

1. On vérifie que la classe demandée existe, si ce n'est pas le cas, une exception est levée
2. On crée une classe de réflexion grâce à la classe PHP ReflexionClass ce qui va nous permettre d'obtenir des informations sur la classe que nous essayons d'instancier
3. On vérifie que la classe demandée est bien Persistante (voir ci-après) et qu'elle est instanciable, si tel n'est pas le cas, on lève une exception
4. On utilise les propriétés de dynamisme du langage pour appeler la méthode statique *generateToken* avec les arguments de la classe (voir ci-après) ceci va nous permettre d'obtenir le token (jeton) de l'instance demandée - on part du principe qu'une instance est identifiée par les paramètres de son constructeur et le nom de la classe (voir ci-après)
5. Si une instance portant ce token est trouvée dans l'IdentityMap, on renvoie sa référence, sinon, on instancie la classe, on met à jour la propriété instance_token de l'objet à l'aide de la méthode *setToken* et finalement on attache l'objet à l'IdentityMap avant de le retourner

Ainsi, on est sûr que chaque instance retournée par cette méthode est unique et si elle était déjà présente dans la variable de session, on l'utilise sans la ré-instancier.

Les Objets persistants

La classe IdentityMap à été conçue pour accueillir des objets qui étendent la classe Persistent.

Ces objet doivent donc posséder les méthodes permettant de générer leurs tokens de façon à ce qu'IdentityMap puisse les identifier:

```
abstract class Persistent implements Identifiable
{
	protected $instance_token;

	public function getToken ()
	{
		return $this->instance_token;
	}

	public function setToken ($token)
	{
		$this->instance_token = $token;
		return $this;
	}

	public static function generateToken ($arguments)
	{
		return static::who() . "_" . md5(serialize($arguments));
	}
}
```

On remarque que, comme Singleton, cette classe implémente l'interface identifiable, vous pouvez le déduire à l'usage de la méthode statique *who* au sein de la méthode statique *generateToken*.
Cette classe abstraite fournit à ses filles une propriété instance_token ainsi que ses accesseurs et la méthode statique qui permet de générer ce token et qui sera utilisée par IdentityMap pour identifier les objets.

Nous pouvons à présent créer notre première classe persistante:

```
class MyPersistentObject extends Persistent
{
	public    $a;
	protected $b;
	private   $c;

	public function __construct ($a, $b, $c)
	{
		echo __METHOD__ . "\n";
		$this->a = $a;
		$this->b = $b;
		$this->c = $c;
	}
```

```
	public static function who () { return __CLASS__; }

	public function __toString ()
	{
		return __CLASS__ . "[TOKEN:" . $this->getToken() . "] \{a:{$this->a}, b:{$this->b}, c:{$this->c}\}";
	}
}
```

Cette classe est suffisament simple pour se passer d'explication, elle étends la classe Persistent et donc doit définir la méthode static *who*. On se servira de *__toString* dans nos exemples pour obtenir un visuel de l'état de notre objet.

Comment ça marche ?

Nous allons tout simplement instancier nos objets, mais non pas en utilisant new mais en demandant à IdentityMap de les créer pour nous:

```
session_start(); // of course mandatory
if (isset($_REQUEST['destroy']))
{
	session_destroy();
	die("Session Détruite");
}
else
{
	echo "<pre>";
	var_dump($_SESSION);
	echo "</pre><hr />";
}

SessionRegistry::singleton();
echo "<pre>";
$MPO  = SR::Factory("MyPersistentObject", array(1,2,3));
$MPO2 = SR::Factory("MyPersistentObject", array(1,4,9));
$MPO3 = SR::Factory("MyPersistentObject", array(1,2,3));
echo $MPO  ."\n";
echo $MPO2 ."\n";
echo $MPO3 ."\n";
echo "</pre>";
```

Une première exécution produit:

```
array(0) {
}
```

```
MyPersistentObject::__construct
MyPersistentObject::__construct
MyPersistentObject[TOKEN:MyPersistentObject_262bbc0aa0dc62a93e350f1f7df792b9] \{a:1, b:2, c:3\}
MyPersistentObject[TOKEN:MyPersistentObject_4be3aa1bfc8428b0fbc819457ce6d409] \{a:1, b:4, c:9\}
MyPersistentObject[TOKEN:MyPersistentObject_262bbc0aa0dc62a93e350f1f7df792b9] \{a:1, b:2, c:3\}
```

La première partie nous montre que la variable de session est vide, mais on s'y attendait, ensuite, on peut voir les traces des exécutions des constructeurs de MyPersistentObject qui ne sont présentes que deux fois: la construction de MPO et MPO3 étants identiques, IdentityMap à simplement renvoyé la même référence de MPO vers MPO3. Nous pouvons contrôller par leurs tokens que les objets MPO et MPO3 sont bien identiques.

Effectuons un deuxième appel au script sans rien changer:

```
array(1) {
  ["instance_map"]=>
  array(2) {
    ["MyPersistentObject_262bbc0aa0dc62a93e350f1f7df792b9"]=>
    object(MyPersistentObject)#1 (4) {
      ["a"]=>
      int(1)
      ["b":protected]=>
      int(2)
      ["c":"MyPersistentObject":private]=>
      int(3)
      ["instance_token":protected]=>
      string(51) "MyPersistentObject_262bbc0aa0dc62a93e350f1f7df792b9"
    }
    ["MyPersistentObject_4be3aa1bfc8428b0fbc819457ce6d409"]=>
    object(MyPersistentObject)#2 (4) {
      ["a"]=>
      int(1)
      ["b":protected]=>
      int(4)
      ["c":"MyPersistentObject":private]=>
      int(9)
      ["instance_token":protected]=>
      string(51) "MyPersistentObject_4be3aa1bfc8428b0fbc819457ce6d409"
    }
  }
}
```

```
MyPersistentObject[TOKEN:MyPersistentObject_262bbc0aa0dc62a93e350f1f7df792b9] \{a:1, b:2, c:3\}
MyPersistentObject[TOKEN:MyPersistentObject_4be3aa1bfc8428b0fbc819457ce6d409] \{a:1, b:4, c:9\}
MyPersistentObject[TOKEN:MyPersistentObject_262bbc0aa0dc62a93e350f1f7df792b9] \{a:1, b:2, c:3\}
```

Cette fois, nous constatons que la variable de session est dûment remplie et on remarque avec satisfaction que les objets on bien été récupéré et n'ont dès lors pas été instanciés une seconde fois.

Un appel au script avec le paramère "destroy" nous permet de vider la variable de session pour retrouver l'état du script au départ.

Conclusion

Avec cette Identity Map, vous êtes désormais capable de conserver vos données d'une page à l'autre sans accéder manuellement à la variable de session. On peut dire qu'on à créé là un cache pour vos objets PHP.
Ce prototype, quoique simple, nous à permis de contourner une limitation majeure du langage. Il nous manque tout de même des wrappers de type Persistent pour les objets natif de PHP que nous ne pouvons étendre; avec un design pattern Décorateur.

Vos question, commentaires ou remarques sont les bienvenus: benjamin.delespier@gmail.com