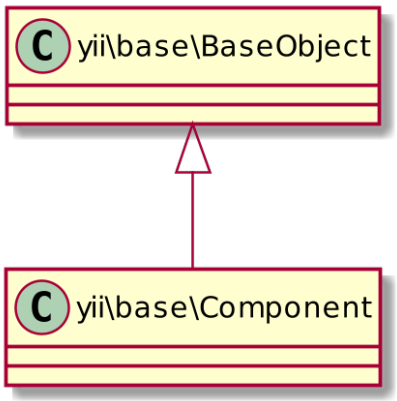
Estructura a pequeña escala de una aplicación Yii 2

Ricardo Pérez López

IES Doñana, curso 2018-19

1. Componentes



- `yii\base\BaseObject`:
  - Propiedades
  - Configurabilidad
- `yii\base\Component`:
  - Eventos
  - Comportamientos (*behaviors*)

- Aunque los componentes son muy potentes, son un poco más pesados que los objetos normales (consumen algo más de memoria y CPU).
- Si un componente no necesita eventos ni comportamientos, es mejor que la clase herede de `yii\base\BaseObject` en lugar de `yii\base\Component`, porque así los objetos serían tan eficientes como objetos normales de PHP, pero además tendrían propiedades y configurabilidad.

2. Propiedades

Por ejemplo:

```php
namespace app\components;

class Foo extends \yii\base\BaseObject
{
    private $_label;

    // El método getter:
    public function getLabel()
    {
        return $this->_label;
    }

    // El método setter:
    public function setLabel($value)
    {
        $this->_label = trim($value);
    }
}
```

Crea la propiedad `label`, accesible mediante `$foo->label`.

```php
$foo = new Foo;

echo $foo->label;     // Llama internamente a $foo->getLabel()

$foo->label = 'hola'; // Llama internamente a $foo->setLabel('hola');
```

Como `setLabel($value)` está definido como:

```php
public function setLabel($value)
{
    $this->_label = trim($value);
}
```

al asignarle una cadena a la propiedad se *trimeará* automáticamente, eliminando los espacios sobrantes:

```php
$foo->label = '    hola    ';   // Se guarda sin espacios sobrantes

echo $foo->label;               // Devuelve "hola" (sin espacios)
```

3. Configurabilidad

`Yii::createObject()` es como una especie de «súper `new`»:

- Crea instancias y las configura al mismo tiempo.
- Usa el *contenedor de inyección de dependencias* para resolver automáticamente las dependencias del objeto que se desea crear.

Por ello, `Yii::createObject()` se usa muchísimo más que `new` a lo largo de todo el código del framework y de la aplicación que hagamos con él.

4. Normas

```php
<?php

namespace yii\components\MiClase;

use yii\base\BaseObject;

class MiClase extends BaseObject
{
    public $prop1;
    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... inicialización antes de aplicar la configuración

        // Lo último que se hace es llamar al constructor del padre:
        parent::__construct($config);
    }

    public function init()
    {
        // Lo primero que se hace es llamar al init() del padre:
        parent::init();

        // ... inicialización después de aplicar la configuración
    }
}
```

5. Eventos

- Los eventos son un mecanismo que nos permite cambiar el comportamiento del *framework* sin tener que cambiar el código del propio *framework*.
- Esto es así porque el *framework* dispara ciertos eventos en ciertos momentos durante su ejecución, lo que podemos usar para vincular nuestro código y hacer que se ejecute en tales momentos.

Para poder disparar eventos o responder a eventos, la clase en cuestión debe ser subclase (directa o indirecta) de `yii\base\Component`.

La signatura del manejador de eventos es:

```php
function ($event) {
    // $event es un objeto de la clase yii\base\Event
    // (o una subclase de esta)
}
```

Ejercicio: consultar qué información contiene el objeto `$event`.

Los manejadores vinculados en otro objeto no se ejecutarán:

```php
$p = new Prueba;
$q = new Prueba;

// $p registra un manejador para el evento Prueba::EVENTO_HOLA:
$p->on(Prueba::EVENTO_HOLA, function ($event) {
    echo "Soy p";
});

// $q registra otro manejador para el mismo evento:
$q->on(Prueba::EVENTO_HOLA, function ($event) {
    echo "Soy q";
});

$p->trigger(Prueba::EVENTO_HOLA); // Muestra "Soy p" pero no "Soy q"
```

Aquí el evento lo dispara $p y por tanto no se ejecuta el manejador registrado por $q (el objeto $q no se entera).

Ejemplo:

```php
use yii\base\Event;

Event::on(Prueba::className(), Prueba::EVENTO_HOLA, function ($event) {
    echo "Hola";
});
```

A partir de ahora, todas las instancias (actuales y futuras) de la clase `Prueba` (y sus subclases) responderán al evento `Prueba::EVENTO_HOLA` mostrando un saludo en pantalla:

```php
$a = new Prueba;
$b = new Prueba;
$a->trigger(Prueba::EVENTO_HOLA);  // Saluda
$b->trigger(Prueba::EVENTO_HOLA);  // También saluda
```

Ejemplo:

```php
class Subclase extends Prueba
{
    // ...
}

$s = new Subclase;
$s->trigger(Prueba::EVENTO_HOLA);  // También saluda
```

Una instancia de `Subclase` dispara un evento que tiene vinculado un manejador en la clase `Prueba`, por lo que también se ejecuta dicho manejador.

En los manejadores de clase, se puede acceder al objeto que ha recibido el disparo del evento mediante la expresión `$event->sender`:

```php
use yii\base\Event;

Event::on(Prueba::className(), Prueba::EVENTO_HOLA, function ($event) {
    // $event->sender es el objeto que ha recibido el evento
});
```

Ejemplo:

```php
use yii\base\Event;

Event::on(Prueba::className(), Prueba::EVENTO_HOLA, function ($event) {
    var_dump($event->sender);  // Muestra "null"
});

// Aquí se dispara el evento de clase.
// Lo dispara directamente la clase Prueba, no una instancia concreta:
Event::trigger(Prueba::className(), Prueba::EVENT_HELLO);
```

Observa que, en este caso, `$event->sender` es `null`, ya que quien dispara el evento no es ninguna instancia concreta, sino una clase.

6. Comportamientos

Ejemplo de acoplamiento estático:

```php
class Usuario extends \yii\db\ActiveRecord
{
    public function behaviors()
    {
        return [
            // anónimo, sólo el nombre de la clase
            Comportamiento::className(),

            // con nombre, sólo el nombre de la case
            'comp2' => Comportamiento::className(),

            // anónimo, array de configuración
            [
                'class' => Comportamiento::className(),
                'prop1' => 'valor1',
                'prop2' => 'valor2',
            ],

            // con nombre, array de configuración
            'comp4' => [
                'class' => Comportamiento::className(),
                'prop1' => 'valor1',
                'prop2' => 'valor2',
            ],
        ];
    }
}
```

- Se le puede asociar un nombre a un comportamiento especificándolo en el array en la clave correspondiente a la configuración de ese comportamiento.
- En tal caso, al comportamiento se le denomina **comportamiento con nombre**.
- En el ejemplo anterior, hay dos comportamientos con nombre: `comp2` y `comp4`.
- Si un comportamiento no lleva asociado ningún nombre, se le denomina **comportamiento anónimo**.

Ejemplo:

```php
// "prop1" es una propiedad definida en la clase Comportamiento
echo $componente->prop1;
$componente->prop1 = $valor;

// pepe() es un método definido en la clase Comportamiento
$componente->pepe();
```

Como se ve, aunque `$componente` no tiene definda la propiedad `prop1` ni el método `pepe()`, puede usarlos como si fueran parte de la definición del componente gracias a que tiene acoplado el comportamiento `Comportamiento`.

7. Alias

- También se puede definir un alias usando otro alias (ya sea alias *raíz* o *derivado*):

```
Yii::setAlias('@pepejuan', '@pepe/juan');
```

Crea el alias *raíz* `@pepejuan` a partir del alias *derivado* `@pepe/juan`.

- Un alias raíz también puede contener barras (/). El método `Yii::getAlias()` es lo bastante inteligente para saber qué parte del alias es un alias raíz y así determinar correctamente la correspondiente ruta o URL:

```php
Yii::setAlias('@pepe', '/ruta/a/pepe');      // alias raíz
Yii::setAlias('@pepe/juan', '/ruta2/juan'); // alias raíz con barra
echo Yii::getAlias('@pepe/test/file.php');  // /ruta/a/pepe/test/file.php
echo Yii::getAlias('@pepe/juan/file.php');  // /ruta2/juan/file.php
```

- Si `@pepe/juan` no hubiese sido un alias raíz, la última sentencia habría devuelto `/ruta/a/pepe/juan/file.php`.

8. Autoloading de clases

9. Localizador de servicios

10. Contenedor de inyección de dependencias

- **Un contenedor de inyección de dependencias** es un objeto que sabe cómo instanciar y configurar objetos y todos los objetos de los que depende.
- Es una instancia de la clase `yii\di\Container`.
- Yii crea uno accesible a través de `Yii::$container`.
- `Yii::$container->set()` sirve para registrar una dependencia.
- `Yii::$container->get()` sirve para crear nuevos objetos:
  - A partir de un nombre de clase, interfaz o alias de dependencia.
  - Resuelve automáticamente las posibles dependencias y las inyecta en el nuevo objeto.
- Al llamar al método `Yii::createObject()`, éste llama a `Yii::$container->get()` para crear el nuevo objeto. Esto permite personalizar globalmente la inicialización de objetos.

Por ejemplo:

```php
class Pepe
{
    public function __construct(Juan $bar)
    {
    }
}

$pepe = $container->get('Pepe');

// equivale a lo siguiente:
$juan = new Juan;
$pepe = new Pepe($juan);
```

- Este es el código de `Yii::createObject()`, donde se aprecia que internamente usa `Yii::$container→get()` para instanciar objetos, resolviendo automáticamente las dependencias e inyectándolas en el objeto recién creado:

```php
public static function createObject($type, array $params = [])
{
    if (is_string($type)) {
        return static::$container→get($type, $params);
    } elseif (is_array($type) && isset($type['class'])) {
        $class = $type['class'];
        unset($type['class']);
        return static::$container→get($class, $params, $type);
    } elseif (is_callable($type, true)) {
        return static::$container→invoke($type, $params);
    } elseif (is_array($type)) {
        throw new InvalidConfigException('Object configuration must be an array containing a "c
    }
    throw new InvalidConfigException('Unsupported configuration type: ' . gettype($type));
}
```