

# Conceptos fundamentales de Yii 2

Ricardo Pérez López

IES Doñana, curso 2018-19

- 1 Componentes
- 2 Propiedades
- 3 Configurabilidad
- 4 Normas
- 5 Eventos
- 6 Comportamientos
- 7 Alias
- 8 Autoloader de clases
- 9 Localizador de servicios
- 10 Contenedor de inyección de dependencias

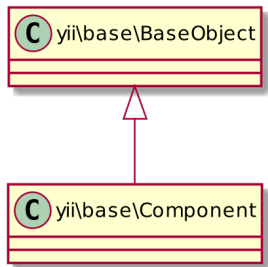
## Sección 1

# Componentes

# Componentes

- Los componentes son los principales bloques de construcción de una aplicación Yii.
- Los componentes son instancias de `yii\base\Component` (o una subclase suya).
- Se caracterizan por tener:
  - Propiedades
  - Configurabilidad
  - Eventos
  - Comportamientos (*behaviors*)

Las dos primeras características se heredan de `yii\base\BaseObject`.



- `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.

## Sección 2

# Propiedades

# Propiedades

- En PHP, a las variables miembro de una clase (variables de instancia) se las denomina también *propiedades*.
- Esas variables son parte de la definición de la clase, y se usan para representar el estado de una instancia de dicha clase.
- La clase `yii\base\BaseObject` de Yii 2 permite crear propiedades a partir de métodos *getter* y *setter*.
- Toda clase que herede (directa o indirectamente) de `yii\base\BaseObject` podrá definir propiedades de esa manera.



Por ejemplo:

```
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`.

```
$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:

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

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

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

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

## Propiedades de sólo lectura

Si definimos sólo el *getter* y no el *setter*, crearemos una **propiedad de sólo lectura**, por lo que podremos consultar su valor pero no cambiarlo:

```
class Prueba extends \yii\base\BaseObject
{
    public $_valor = 25;

    // Método getter (no hay setter):
    public function getValor()
    {
        return $this->_valor;
    }
}
```

```
$p = new Prueba;
echo $p->valor; // Devuelve 25
$p->valor = 30; // Da ERROR
```

## Sección 3

# Configurabilidad

# Configurabilidad

- Una instancia de la clase `yii\base\BaseObject` (o de una subclase suya) permite ser *configurada*.
- Una **configuración** es simplemente un array que contiene parejas de `clave => valor`, donde la `clave` representa el nombre de una propiedad (una cadena), y el `valor` es el valor que queremos asignarle a dicha propiedad.
- Se pueden usar para:
  - Asignar valores de forma masiva a las propiedades de un objeto usando `Yii::configure($objeto, $config)`.
  - Crear una instancia asignándole valores iniciales a sus propiedades usando `Yii::createObject($config)`.
  - Más posibilidades que iremos viendo en su momento.

# Asignación masiva

Supongamos la siguiente clase:

```
use yii\base\BaseObject;

class Prueba extends BaseObject
{
    public $uno;
    private $_dos;

    public function getDos()
    {
        return $this->_dos;
    }

    public function setDos($dos)
    {
        $this->_dos = $dos;
    }
}
```

Algunas posibles configuraciones:

```
[ 'uno' => 5, 'dos' => 7 ]
```

```
[ 'dos' => 18 ]
```

Se pueden aplicar a un objeto ya existente:

```
$p = new Prueba;

Yii::configure($p, [
    'uno' => 5,
    'dos' => 7,
]);

echo $p->uno; // Muestra "5"
echo $p->dos; // Muestra "7"
```

# Creación de nuevas instancias

- Una configuración también se puede usar para crear nuevas instancias y asignarle valores iniciales *en la misma operación* usando el método `Yii::createObject($config)`.
- Para ello es necesario que la configuración indique el nombre de la clase que se desea instanciar mediante un elemento con clave `'class'` (que además tiene que ser el primer elemento del array).
- Ejemplo:

```
$p = Yii::createObject([  
    'class' => 'Prueba',  
    'uno' => 4,  
    'dos' => 7,  
]);
```

- Se crea en `$p` una nueva instancia de la clase `Prueba` con los valores

`$p->uno = 4` y `$p->dos = 7`.



`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.

## Sección 4

### **Normas**

# Normas de creación de componentes

- Al heredar de `yii\base\Component` o `yii\base\BaseObject`, hay que seguir las siguientes normas:
  - Si se sobreescribe el constructor, el último parámetro debe ser `$config = []` y al final hay que pasárselo al constructor del padre.
  - Llamar siempre al constructor del padre al final del constructor sobreescrito.
  - Si se sobreescribe el método `yii\base\BaseObject::init()`, hay que llamar al `init()` del padre al comienzo del `init()` sobreescrito.

```
<?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
    }
}
```

## Sección 5

### **Eventos**

# Eventos

- Los eventos permiten inyectar nuestro código dentro de código ya existente en determinados puntos de ejecución.
- Se puede vincular un trozo de código a un evento de forma que, cuando el evento se dispare, se ejecutará el código automáticamente.
- Por ejemplo, un objeto que envíe correo puede disparar el evento `mensajeEnviado` cada vez que envíe un email. Si se desea hacer un seguimiento de los mensajes que se han enviado, se puede vincular el código de seguimiento al evento `mensajeEnviado`.

- 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`.



# Manejadores de eventos

- Un manejador de eventos es un *callable* de PHP que se ejecutará cuando se dispare el evento al que se haya vinculado.
- El *callable* puede ser:
  - Una función global de PHP especificada en forma de cadena (sin paréntesis): `'trim'`
  - Un método de instancia especificado como un array donde el primer elemento es el objeto y el segundo es el nombre del método como cadena (sin paréntesis): `[$objeto, 'metodo']`
  - Un método estático de clase especificado como un array donde el primer elemento es el nombre de la clase y el segundo es el nombre del método como cadena (sin paréntesis): `['NombreClase', 'metodo']`
  - Una función anónima: `function ($event) { ... }`

La signatura del manejador de eventos es:

```
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`.

## Vincular manejadores a eventos

Para **vincular un manejador a un evento en un objeto** se usa el método `yii\base\Component::on()` sobre ese objeto:

```
$p = new Prueba;

// Este manejador es una función global de PHP:
$p->on(Prueba::EVENTO_HOLA, 'funcion');

// Este manejador es un método de instancia:
$p->on(Prueba::EVENTO_HOLA, [$objeto, 'metodo']);

// Este manejador es un método estático de clase:
$p->on(Prueba::EVENTO_HOLA, ['\app\components\Pepe', 'metodo']);

// Este manejador es una función anónima:
$p->on(Prueba::EVENTO_HOLA, function ($event) {
    // Código que gestiona el evento
});
```

Cuando el objeto dispare el evento, se ejecutará el manejador vinculado en ese evento.

# Disparar eventos

- Los eventos se disparan llamando al método `yii\base\Component::trigger()` sobre el objeto que envía (o *dispara*) el evento:

```
$p->trigger(Prueba::EVENTO_HOLA);
```

- El objeto `$p` dispara el evento `Prueba::EVENTO_HOLA`, lo que provocará la ejecución de los manejadores que se hayan vinculado a ese evento en ese objeto (además de los *manejadores de clase*, que veremos luego).
- Si hay varios manejadores para el mismo evento, se ejecutarán en cadena en el orden en el que hayan sido vinculados.
- Un manejador puede hacer `$event->handled = true;` para romper la cadena y evitar que se ejecuten más manejadores de ese evento.

Los manejadores vinculados en otro objeto no se ejecutarán:

```
$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).

# Constantes para los eventos

- Es recomendable usar constantes de clase para representar los nombres de los eventos.
- En el ejemplo anterior, la constante `Prueba::EVENTO_HOLA` representa el evento `hola`.
- Esto tiene tres ventajas:
  - 1 Previene equivocaciones al teclear.
  - 2 Los hace más reconocible por el autocompletado de los editores.
  - 3 Resulta más fácil saber qué eventos soporta una clase simplemente mirando las constantes que tenga declaradas.

# Manejadores de clase

- Hasta ahora hemos vinculado manejadores a eventos *a nivel de instancia*, es decir, en objetos concretos.
- A veces, queremos que *todas* las instancias de una clase respondan de la misma forma a un determinado evento.
- En lugar de vincular un manejador de evento en cada instancia, podemos vincular el manejador *a nivel de clase*, es decir, en la propia clase.
- A estos eventos se los denomina **manejadores de clase**, a diferencia de los manejadores a nivel de instancia, que se denominan **manejadores de instancia**.
- Para ello, usamos el método estático `yii\base\Event::on()` indicando el nombre de la clase, el nombre del evento y el manejador del evento.

## Ejemplo:

```
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:

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



## También afecta a las subclases

- Los manejadores a nivel de clase se ejecutarán cuando se produzca un evento disparado por cualquier instancia de esa clase **o de cualquier subclase suya**.
- Dicho de otra forma: el evento se propagará hacia arriba en la jerarquía de herencia y provocará la ejecución de todos los manejadores de clase que encuentre vinculados a ese evento.
- Debido a eso, hay que tener cuidado para evitar la propagación de ese evento a más objetos de los necesarios.
- Por ejemplo, si vinculamos un manejador de evento a la clase `yii\base\BaseObject`, prácticamente *todas* las instancias de Yii 2 podrán responder a ese evento.

Ejemplo:

```
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`:

```
use yii\base\Event;
```

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

# Orden de los manejadores

- El mismo evento puede tener vinculados manejadores de instancia y manejadores de clase.
- En ese caso, al dispararse el evento se ejecutarán primero los manejadores de instancia y después los manejadores de clase (los manejadores de instancia siempre van primero).

## Eventos de clase

- Hasta ahora, los eventos han sido disparados por objetos concretos. A estos eventos se los denomina **eventos de instancia**.
- Podemos hacer que una clase también dispare eventos. Los eventos disparados por una clase se denominan **eventos de clase**.
- Los eventos de clase se disparan llamando al método `yii\base\Event::trigger()`, indicando el nombre de la clase y el nombre del evento.
- Los eventos de clase sólo provocan la ejecución de manejadores de clase, no de manejadores de instancia.

## Ejemplo:

```
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.

# Resumiendo

- Los eventos de instancia provocan la ejecución de manejadores de instancia y manejadores de clase.
- Los eventos de clase sólo provocan la ejecución de manejadores de clase.

## Sección 6

# Comportamientos



# Comportamientos

- Los comportamientos (o *behaviors*) permiten ampliar la funcionalidad de una clase sin afectar a su herencia.
- En otros lenguajes de programación se denominan *mixins*.
- Al *acoplar* un comportamiento a un componente se *inyectan* los métodos y las propiedades del comportamiento dentro del componente.
- El componente podrá usar esos métodos y propiedades como si estuvieran definidos en la clase del componente.
- Además, un comportamiento puede responder a los eventos disparados por el componente, lo que le permite alterar la ejecución normal del código del componente.

# Definición de comportamientos

Un comportamiento es una subclase (directa o indirecta) de `yii\base\Behavior`:

```
class Comportamiento extends \yii\base\Behavior
{
    public $prop1;
    private $_prop2;

    public function pepe()
    {
        // ...
    }

    public function getProp2()
    {
        return $this->_prop2;
    }

    public function setProp2($valor)
    {
        $this->_prop2 = $valor;
    }
}
```

- Este comportamiento define dos propiedades (`prop1` y `prop2`) y un método (`pepe()`).
- Cuando se acople a un componente, éste dispondrá de esas propiedades y ese método.

# Acoplar comportamientos a un componente

- Los comportamientos se pueden acoplar de forma *estática* o *dinámica*.
- El acoplamiento estático es el más usado.

# Acoplamiento estático de componentes

- Se sobrescribe el método `behaviors()` del componente al que se desea acoplar el comportamiento.
- El método `behaviors()` debe devolver un array de configuraciones de comportamientos.
- Cada configuración puede ser:
  - El nombre de una clase comportamiento, o
  - Un array de configuración.

## Ejemplo de acoplamiento estático:

```
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 clase
            '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**.

## Acoplamiento dinámico de comportamientos

Para acoplar un comportamiento dinámicamente, se llama al método `yii\base\Component::attachBehavior()` del componente al que se le va a acoplar el comportamiento:

```
// acopla un objeto comportamiento
$componente->attachBehavior('comp1', new Comportamiento);

// acopla una clase comportamiento
$componente->attachBehavior('comp2', Comportamiento::className());

// acopla un array de configuración
$componente->attachBehavior('comp3', [
    'class' => Comportamiento::className(),
    'prop1' => 'valor1',
    'prop2' => 'valor2',
]);
```

# Uso de comportamientos

- Para usar un comportamiento, primero hay que acoplarlo a un componente usando uno de los métodos que se han visto antes.
- A partir de ese momento, el componente podrá:
  - Acceder a cualquier variable de instancia pública o propiedad pública definida en el comportamiento como si estuvieran realmente definidas en el componente.
  - Invocar a cualquier método público definido en el comportamiento como si estuviera definido en el componente.



## Ejemplo:

```
// "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 definida 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`.

## Sección 7

### **Alias**

# Alias

- Representan rutas o URLs.
- Se usan para no tener que codificar rutas absolutas o URLs directamente en el proyecto.
- Empiezan por @.
- Yii 2 tiene varios alias predefinidos.
- Ejemplos:
  - `Yii::getAlias('@app')` → `"/home/ricardo/proyecto"`
  - `Yii::getAlias('@yii')` → `"/home/ricardo/proyecto/vendor/yiisoft/yii2"`

## Definiciones de alias

- Se puede definir un alias a una ruta o una URL usando `Yii::setAlias()`:

*// alias a una ruta*

```
Yii::setAlias('@pepe', '/ruta/a/pepe');
```

*// alias a una URL*

```
Yii::setAlias('@juan', 'http://www.ejemplo.com');
```

*// alias a un archivo concreto que contiene a la clase \pepe\Juan*

```
Yii::setAlias('@pepe/Juan.php', '/ruta/hasta/pepe/Juan.php');
```

# Alias derivados

- A partir de un alias, se puede derivar un nuevo alias sin tener que usar `Yii::setAlias()`, añadiendo una barra (/) seguido de uno o más segmentos de ruta.
- El alias definido con `Yii::setAlias()` se denomina **alias raíz**.
- Los alias que derivan de este se denominan **alias derivados**.
- Ejemplo:

```
Yii::setAlias('@pepe', 'ruta/a/pepe');
```

- `@pepe` es un *alias raíz* (se creó con `Yii::setAlias()`)
- `@pepe/juan/archivo.php` es un *alias derivado* (no se creó con `Yii::setAlias()`)

- 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`.

# Resolución de alias

- Se puede llamar a `Yii::getAlias()` para resolver (traducir) un alias raíz a la ruta o URL que representa.
- También se usa el mismo método para resolver alias derivados:

```
echo Yii::getAlias('@pepe');           // /ruta/a/pepe
echo Yii::getAlias('@juan');            // http://www.ejemplo.com
echo Yii::getAlias('@pepe/juan/fi.php'); // /ruta/a/pepe/juan/fi.php
```

- 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:

```
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`.



# Alias predefinidos

- `@yii`: El directorio base del framework.
- `@app`: El directorio base de la aplicación.
- `@runtime`: El directorio temporal de la aplicación. Por defecto vale `@app/runtime`.
- `@webroot`: El *DocumentRoot* de la aplicación (donde está almacenado el script de entrada).
- `@web`: La URL base de la aplicación. Tiene el mismo valor que `yii\web\Request::$baseUrl`.
- `@vendor`: El directorio `vendor` de **Composer**. Por defecto vale `@app/vendor`.
- `@bower`: El directorio raíz de los paquetes de **Bower**. Por defecto vale `@vendor/bower`.
- `@npm`: El directorio raíz de los paquetes de **npm**. Por defecto vale `@vendor/npm`.

## Sección 8

### **Autoloader de clases**

# Autoloader de clases

- Es un autoloader que cumple con el estándar **PSR-4**.
- Para que funcione, hay que seguir dos reglas:
  - Cada clase debe pertenecer a un espacio de nombres (como en `\pepe\juan\MiClase`).
  - Cada clase debe guardarse en un archivo individual cuya ruta se determina por el siguiente algoritmo:

```
// $clase es un nombre de clase totalmente cualificado sin \ inicial  
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $clase) . '.php');
```

- Al programar, hay que crear las clases en un espacio de nombres que cuelgue del raíz `app`. El autoloader la encontrará porque `@app` es un alias predefinido que siempre existe en Yii 2.

## Sección 9

### **Localizador de servicios**

# Localizador de servicios

- Es un objeto que sabe cómo proporcionar todo tipo de *servicios* (también llamados **componentes**) que pueda necesitar una aplicación.
- Dentro del localizador de servicios, cada componente existe como una única instancia identificada mediante un ID, que se usa para recuperar el componente de dentro del localizador de servicios.
- El localizador de servicios es una instancia de la clase `yii\di\ServiceLocator` (o una subclase).
- El más típico en Yii es el **objeto aplicación**, al que se accede mediante `\Yii::$app`. Los servicios que proporciona se denominan **componentes de aplicación**, como `request`, `response`, `db` o `urlManager`. Esos componentes se suelen definir mediante configuraciones.

## Registrar componentes

- Para usar un localizador de servicios, el primer paso es registrar componentes dentro de él.
- Un componente se puede registrar mediante `yii\di\ServiceLocator::set()`:

```
$locator = new yii\di\ServiceLocator;
```

*// registra "cache" usando un nombre de clase con el que se creará un component*

```
$locator->set('cache', 'yii\caching\ApcCache');
```

*// registra "db" usando una configuración con la que se creará un componente:*

```
$locator->set('db', [  
    'class' => 'yii\db\Connection',  
    'dsn' => 'mysql:host=localhost;dbname=demo',  
    'username' => 'root',  
    'password' => '',  
]);
```

*// registra "search" usando una función anónima que creará un componente:*

```
$locator->set('search', function () {  
    return new app\components\SolrService;  
});
```

# Usar componentes

- Una vez que el componente ha sido registrado, se puede acceder a él mediante su ID, usando una de estas dos formas:

```
$cache = $locator->get('cache');
```

*// o bien:*

```
$cache = $locator->cache;
```

- Como se ve, `yii\di\ServiceLocator` permite acceder a un componente como si fuera una propiedad, usando el ID del componente.
- Cuando se accede a un componente la primera vez, `yii\di\ServiceLocator` usará la información del registro del componente para crear una nueva instancia del componente y la devolverá.
- De ahí en adelante, si se vuelve a acceder al componente, el localizador de servicios devolverá siempre la misma instancia.

## Registro masivo de componentes

- Como los localizadores de servicios a menudo se crean mediante **configuraciones**, tienen una propiedad llamada `components` que permite configurar y registrar varios componentes a la vez:

```
$config = [
    // ...
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],
        'cache' => 'yii\caching\ApcCache',
        'search' => function () {
            $solr = new app\components\SolrService;
            // ... otras inicializaciones ...
            return $solr;
        },
    ],
];
$locator = new yii\di\ServiceLocator($config);
```



## Sección 10

# **Contenedor de inyección de dependencias**

# Contenedor de inyección de dependencias

Artículo interesante sobre *inversión de dependencias* e *inyección de dependencias*:

<http://raulavila.com/2015/03/principios-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.

## Ejemplo de uso

- Por ejemplo:

```
Yii::$container->set(yii\widgets\LinkPager::class, [  
    'maxButtonCount' => 5  
]);
```

- Cuando luego se quiera instanciar un objeto `yii\widgets\LinkPager` usando `Yii::createObject()`, se creará con `maxButtonCount = 5`:

```
$linkPager = Yii::createObject(yii\widgets\LinkPager::class);  
echo $linkPager->maxButtonCount; // devuelve 5
```

- Se habría obtenido el mismo resultado con:

```
$linkPager = Yii::$container->get(yii\widgets\LinkPager::class);  
echo $linkPager->maxButtonCount; // también devuelve 5
```

## Ejemplo de inyección de dependencias

- El contenedor de inyección de dependencias soporta la **inyección de constructores** usando tipos en los parámetros del constructor.
- Los tipos en los parámetros del constructor indican al contenedor de qué clases o interfaces depende el objeto que se va a crear.
- El contenedor intentará obtener instancias de las clases o interfaces de las que depende y luego las inyectará en el nuevo objeto a través del constructor.

Por ejemplo:

```
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:

```
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');
    }
    throw new InvalidConfigException('Unsupported configuration type: ' . gettype($type));
}
```