Estructura de una aplicación Yii 2

Ricardo Pérez López

IES Doñana, curso 2018-19

Sección 1

A pequeña escala

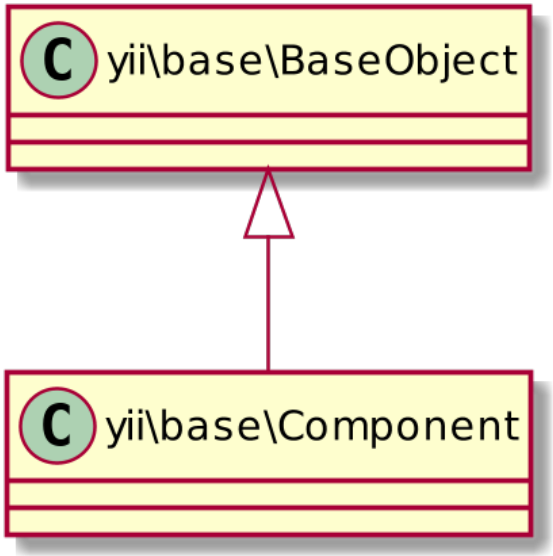
Sección 2

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 3

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:

```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)
```

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:

```php
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 4

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:

```php
use yii\base\BaseObject;

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

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

Algunas posibles configuraciones:

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

[ 'dos' => 18 ]
```

Se pueden aplicar a un objeto ya existente:

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

`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 5

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
<?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 6

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:

```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`.

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:

```php
$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:

```php
$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:

```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).

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:

```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
```

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:

```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
});
```

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:

```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.

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 7

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`:

```php
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 sobreescribe 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:

```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**.

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:

```php
// 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:

```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`.

Sección 8

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()`:

```php
// 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 9

Autoloading 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:

```php
// $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 10

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()`:

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

// registra "cache" usando un nombre de clase con el que se creará un componente:
$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:

```php
$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:

```php
$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 11

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:

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

Sección 12

A gran escala

Sección 13

Introducción

Introducción

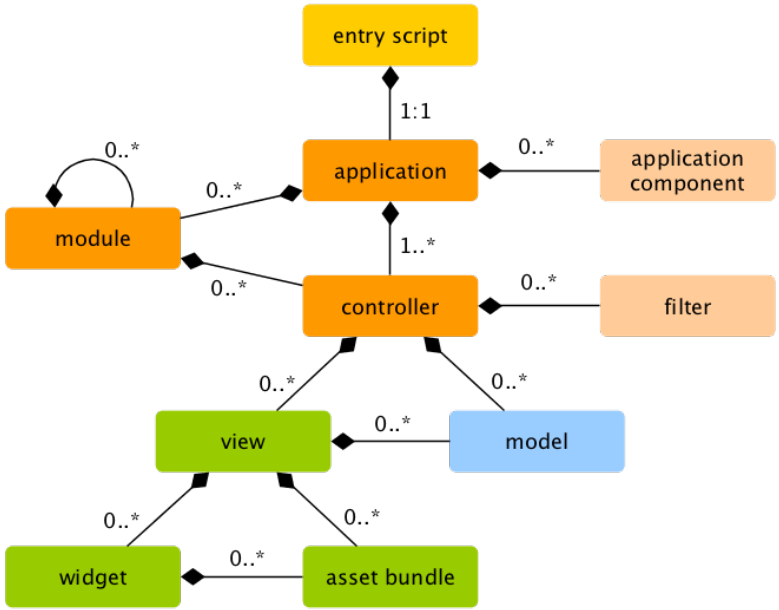


Figura 1: Estructura a gran escala de una aplicación Yii2

Sección 14

Scripts de entrada

Scripts de entrada

- Los **scripts de entrada** son el primer paso en el proceso de arranque de una aplicación.
- Una aplicación (ya sea web o de consola) tiene siempre un único script de entrada.
- Los usuarios finales realizan peticiones a los scripts de entrada, los cuales instanciarán un objeto aplicación y le redirigirá las peticiones a él.

Script de entrada de una aplicación web

- El script de entrada de una aplicación web se debe almacenar en un directorio accesible públicamente desde el exterior a través del servidor web, de forma que los usuarios finales pueda alcanzarlo.
- Normalmente se llama `index.php`, pero no es imprescindible.
- En la plantilla básica se encuentra en `web/index.php`.

Script de entrada de una aplicación de consola

- El script de entrada de una aplicación de consola se guarda normalmente en la ruta base de la aplicación y con el nombre `yii` (así ocurre en la plantilla básica).
- Debe tener permiso de ejecución para que los usuarios puedan ejecutar la aplicación de consola mediante el comando:

```
./yii <ruta> [argumentos] [opciones]
```

¿Qué hacen los scripts de entrada?

Principalmente, hacen lo siguiente:

- Define constantes globales.
- Registran el autoloader de Composer.
- Incluye el archivo de la clase Yii.
- Carga la configuración de la aplicación.
- Crea y configura una instancia de la aplicación.
- Llama a `yii\base\Application::run()` para procesar la petición entrante.

Ejemplo

Script de entrada de la aplicación web de la plantilla `yii2-app-basic`:

```php
<?php

defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

// registra autoloader de Composer
require __DIR__ . '/../vendor/autoload.php';

// incluye el archivo de la clase Yii
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

// carga la configuración de la aplicación
$config = require __DIR__ . '/../config/web.php';

// crea, configura y ejecuta la aplicación
(new yii\web\Application($config))->run();
```

Sección 15

Aplicaciones

Aplicaciones

- Las aplicaciones son los objetos que gobiernan la estructura general y el ciclo de vida de una aplicación Yii 2.
- Cada aplicación en Yii 2 contiene un único objeto aplicación.
- Ese objeto se crea en el script de entrada.
- Se puede acceder a él desde cualquier parte usando la expresión `\Yii::$app`.

Configuración de aplicaciones

- Cuando un script de entrada crea una aplicación, cargará su configuración y se la aplicará a la aplicación al instanciar el objeto aplicación:

```php
<?php

require __DIR__ . '/../vendor/autoload.php';
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

// carga la configuración de la aplicación desde el archivo config/web.php
$config = require __DIR__ . '/../config/web.php';

// instancia, configura y ejecuta la aplicación
(new yii\web\Application($config))->run();
```

- Como pasa con cualquier otra configuración, la configuración de la aplicación especifica cómo inicializar las propiedades del objeto aplicación.
- Como las configuraciones de aplicación suelen ser muy complejas, normalmente se mantienen en archivos de configuración separados, como el `config/web.php` del ejemplo anterior.
- Precisamente, `config/web.php` es el archivo principal de configuración de la aplicación web en la plantilla básica de Yii 2.

Propiedades de la aplicación

- Las aplicaciones tienen muchas propiedades importantes que se pueden configurar mediante una configuración de aplicación.
- Por ejemplo, las aplicaciones deben saber cómo y dónde cargar controladores, dónde guardar los archivos temporales, etc.
- Las dos únicas propiedades obligatorias son:
  - `id`: un identificador único que diferencia una aplicación de las demás.
  - `basePath`: especifica el directorio raíz de la aplicación (a donde apunta el alias `@app`).

Propiedades importantes de una aplicación

`components` Permite registrar (y configurar) en la aplicación los **componentes de aplicación** que alojará la misma (recuerda que una aplicación es un *localizador de servicios*). Por ejemplo:

```
[
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
        ],
    ],
]
```

Aquí se especifican dos componentes de aplicación: `cache` y `user`. Cada componente va asociado con la configuración que se usará al instanciar ese componente. Después se podrá acceder a ellos haciendo, por ejemplo, `\Yii::$app->cache`.

`aliases` Te permite definir un conjunto de alias usando un array. Por ejemplo:

```
[
    'aliases' => [
        '@nombre1' => 'ruta/del/alias1',
        '@nombre2' => 'ruta/del/alias2',
    ],
]
```

Equivale a hacer:

```
\Yii::setAlias('@nombre1', 'ruta/del/alias1')
\Yii::setAlias('@nombre2', 'ruta/del/alias2')
```

`bootstrap` Permite definir componentes que se cargarán durante el proceso de arranque de la aplicación.

Recordemos que un componente de aplicación, normalmente, no se carga hasta que se accede a él por primera vez indicando el ID del servicio asociado en `\Yii::$app->ID`. En cambio, a veces interesa que determinados componentes se carguen y se ejecuten **siempre** al iniciarse la aplicación:

```
[
    'bootstrap' => [
        'app\components\Profiler',
    ],
]
```

`language` Especifica el idioma en el que la aplicación deberá mostrar el contenido a los usuarios finales.

El valor predeterminado es `en`.

Para usar el español de España, lo correcto sería establecerlo a `es-ES`:

```
[
    'language' => 'es-ES',
]
```

`timeZone` Define la zona horaria con la que trabajará la aplicación a la hora de manipular fechas y horas.

El valor predeterminado es `UTC`, y así es como debería ser.

El componente de aplicación `formatter` (el encargado de formatear los datos para que el usuario los visualice) también tiene una propiedad `timeZone`, que sí debemos establecer a la zona horaria correcta (`Europe/Madrid` o la que el usuario tenga establecida en su perfil).

Sección 16

Componentes de aplicación

Componentes de aplicación

- Recordemos que las aplicaciones son *localizadores de servicios*.
- Las aplicaciones contienen los llamados **componentes de aplicación**, los cuales proporcionan diferentes servicios útiles durante el procesamiento de las peticiones.
- Por ejemplo:
  - El componente `urlManager` es responsable de encaminar (*enrutar*) las peticiones web a los controladores apropiados.
  - El componente `db` proporciona servicios relacionados con la base de datos.

- Cada componente de aplicación tiene un ID que lo identifica de forma única entre los demás componentes de la misma aplicación.
- Se puede acceder a un componente de aplicación mediante la expresión `\Yii::$app->ID`.
- Por ejemplo, se puede usar `\Yii::$app->db` para acceder a la conexión a la base de datos, o `\Yii::$app->cache` para obtener la caché principal registrada en la aplicación.
- Un componente de aplicación se crea la primera vez que se accede a él usando la expresión anterior. Los demás accesos posteriores devolverán la misma instancia sin crear otro objeto.

Como vimos anteriormente, los componentes de aplicación se definen (o *registran*) a través de la propiedad `components` de la configuración de la aplicación:

```
[
    'components' => [
        // registra el componente "cache" usando un nombre de clase:
        'cache' => 'yii\caching\ApcCache',

        // registra el componente "db" usando una configuración:
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'pgsql:host=localhost;dbname=demo',
            'username' => 'usuario',
            'password' => 'contraseña',
        ],

        // registra el componente "search" usando una función anónima:
        'search' => function () {
            return new app\components\SolrService;
        },
    ],
]
```

Componentes de aplicación principales

- Yii 2 define un conjunto de componentes de aplicación *principales* o predefinidos, con IDs fijos y configuraciones predeterminadas.
- Gracias a esos componentes principales, las aplicaciones pueden procesar las peticiones de los usuarios.
- Es posible configurar y personalizar esos componentes como si fueran componentes de aplicación normales. Cuando se configura un componente de aplicación principal, si no se indica la clase a usar, se usará su clase predeterminada.

Lista de componentes de aplicación principales:

- `assetManager`: gestiona los *asset bundles* y la publicación de *assets*.
- `db`: representa la conexión a la base de datos.
- `errorHandler`: gestiona los errores y excepciones de PHP.
- `formatter`: formatea los datos para que los visualicen los usuarios finales.
- `i18n`: se encarga de la traducción de mensajes y la internacionalización.
- `log`: gestiona los registros de la aplicación.
- `mailer`: se encarga de la composición y envío de e-mails.
- `response`: representa la respuesta que va a ser enviada al usuario final.
- `request`: representa la petición recibida del usuario final.
- `session`: representa la información de la sesión (sólo en aplicaciones web).
- `urlManager`: se encarga de la interpretación y creación de URLs.
- `user`: representa la información de autenticación de usuarios.
- `view`: gestiona el renderizado de las vistas.

Sección 17

Controladores

Controladores

- Los **controladores** son parte de la arquitectura **MVC**.
- Son instancias de subclases (directas o indirectas) de `yii\base\Controller`.
- Son los responsables de procesar las peticiones y generar las respuestas.

Acciones

- Los controladores se componen de **acciones**.
- La acción es la unidad mínima de ejecución en una aplicación MVC.
- Es decir, es la unidad más básica que un usuario final puede solicitar que se ejecute.
- Un controlador puede tener una o más acciones.

Ejemplo:

```php
namespace app\controllers;

use app\models\Post;

class PostController extends \yii\web\Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);
        if ($model === null) {
            throw new \yii\web\NotFoundHttpException;
        }

        return $this->render('view', [
            'model' => $model,
        ]);
    }
}
```

- En la acción `view` (definida por el método `actionView()`), se carga el modelo que corresponda al ID solicitado en `$id`. Si el modelo se carga correctamente, se muestra usando una vista llamada `view`. En caso contrario, se lanza una excepción.

```php
namespace app\controllers;

class PostController extends \yii\web\Controller
{
    public function actionCreate()
    {
        $model = new \app\models\Post;

        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {
            return $this->render('create', [
                'model' => $model,
            ]);
        }
    }
}
```

- En la acción `create` (definida por el método `actionCreate()`) se intenta rellenar una instancia nueva del modelo usando los datos de la petición y luego se guarda el modelo. Si hay éxito se redirige al navegador a la acción `view` pasándole el ID del modelo recién creado. En caso contrario, se muestra la vista `create` para que los usuarios puedan introducir los datos.

Rutas

- Los usuarios finales indican las acciones que desean ejecutar mediante las denominadas **rutas**.
- Una ruta es una cadena que consta de:

  El ID de un controlador Una cadena que identifica de forma única a un controlador de entre todos los controladores de la misma aplicación.

  El ID de una acción Una cadena que identifica de forma única a una acción de entre todas las acciones del mismo controlador.
- Las rutas toman la siguiente forma: `IDcontrolador/IDacción`.
- Por ejemplo: `socios/crear` (controlador `socios`, acción `crear`).

Para indicar una ruta, se utiliza el parámetro `r` enviado mediante GET.

Por tanto, si un usuario solicita la URL

`http://host/index.php?r=site/index`

se ejecutará la acción `index` del controlador `site`.

- Si la URL tuviera más parámetros, éstos se pasarían a la acción correspondiente.
- Por ejemplo, en la URL

  `http://host/index.php?r=site/index&page=3`

  se ejecutaría la acción `index` del controlador `site` y se le pasaría a dicha acción el parámetro `page` con el valor `3`.
- En la práctica, es como si el *framework* hiciera:

```php
(new \app\controllers\SiteController)->actionIndex(3);
```

  (suponiendo que el método `actionIndex()` de la clase `SiteController` recibe un parámetro de nombre `$page`).

Sección 18

Modelos

Modelos

- Los modelos son parte de la arquitectura MVC.
- Son objetos que representan datos, reglas y lógica de negocio.
- Las clases modelo se crean heredando (directa o indirectamente) de `yii\base\Model`.
- Dicha clase base proporciona muchas características útiles:
  - **Atributos**: representan los datos de negocio y se puede acceder a ellos como si fueran propiedades normales o elementos de un array.
  - **Etiqutas de atributos**: especifican las etiquetas con las que se visualizan los atributos.
  - **Asignación masiva**: permite asignar valores a varios atributos a la vez en un solo paso.
  - **Reglas de validación**: garantiza que los datos introducidos son válidos en función de unas reglas de validación indicadas.
  - **Exportación de datos**: permite exportar los datos del modelo en forma de arrays con formatos personalizables.

Atributos

- Los modelos representan los datos de negocio en forma de atributos.
- Cada atributo es como una propiedad públicamente accesible de un modelo.
- El método `yii\base\Model::attributes()` especifica qué atributos tiene una clase modelo.
- Se puede acceder a un atributo como si fuera una propiedad normal y corriente de un objeto:

```php
$modelo = new \app\models\ContactForm;

// "nombre" es un atributo de ContactForm
$modelo->nombre = 'ejemplo';
echo $modelo->nombre;
```

- También se pueden acceder a los atributos como si fueran elementos de un array, gracias a que `yii\base\Model` implementa las interfaces ArrayAccess y Traversable:

```php
$modelo = new \app\models\ContactForm;

// se accede a los atributos como si fueran elementos de un array:
$modelo['nombre'] = 'ejemplo';
echo $modelo['nombre'];

// Model es recorrible usando foreach:
foreach ($modelo as $nombreAtributo => $valor) {
    echo "$nombreAtributo: $valor\n";
}
```

Definición de atributos

- De entrada, si la clase modelo hereda directamente de `yii\base\Model`, *todas sus* ***variables miembro públicas no estáticas*** serán consideradas atributos.
- Por ejemplo, la siguiente clase modelo `ContactForm` tiene cuatro atributos: `nombre`, `correo`, `asunto` y `cuerpo`:

```php
namespace app\models;

class ContactForm extends \yii\base\Model
{
    public $nombre;
    public $correo;
    public $asunto;
    public $cuerpo;
}
```

- El modelo `ContactForm` se puede usar para contener los datos recibidos a través de un formulario HTML.

- Se puede sobreescribir el método `attributes()` de `yii\base\Model` para definir atributos de forma distinta.
- Ese método debe devolver los nombre de los atributos del modelo.
- Por ejemplo, `yii\db\ActiveRecord` lo hace devolviendo como nombres de atributos los nombres de las columnas de la tabla asociada con el modelo.
    - En este caso, y aunque hace falta algo más de *magia* para que acabe funcionando, al final se consigue que cada columna de la tabla aparezca como atributo del modelo correspondiente a esa tabla.

Escenarios

- Un modelo puede usarse en diferentes *escenarios*.
- Por ejemplo, se puede usar un modelo `User` para recoger los datos de entrada durante el *login* de un usuario, pero también se puede usar para registrar a un nuevo usuario.
- En escenarios diferentes, un modelo puede tener reglas y lógica de negocio diferentes.
- Por ejemplo, el atributo `correo` puede ser obligatorio para registrar a un nuevo usuario pero no para hacer *login*.

- Cada escenario define qué atributos se pueden **asignar masivamente** y qué **reglas de validación** se aplican:
  - La *asignación masiva* permite asignar valores a varios atributos al mismo tiempo en una sola operación.
  - Las *reglas de validación* determinan si los atributos de un modelo son válidos, o sea, si cumplen determinadas condiciones.

Escenario activo

- Los modelos usan la propiedad `yii\base\Model::scenario` para indicar en qué escenario se encuentran.
- Por defecto, un modelo sólo tiene un escenario, llamado `default`.
- Para cambiar el escenario de un modelo se puede hacer:

```php
// El escenario se establece con una propiedad:
$modelo = new User;
$modelo->scenario = User::SCENARIO_LOGIN;
```

o bien:

```php
// El escenario se establece con una configuración:
$modelo = new User(['scenario' => User::SCENARIO_LOGIN]);
```

- El escenario en el que se encuentra actualmente un modelo se denomina el **escenario activo** del modelo.

Definición de escenarios

- Los escenarios que soporta un modelo normalmente se definen en las *reglas de validación* del modelo.
- Sin embargo, también se pueden definir sobreescribiendo el método `yii\base\Model::scenarios()`:

```php
namespace app\models;

class User extends \yii\db\ActiveRecord
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTRO = 'registro';

    public function scenarios()
    {
        return [
            self::SCENARIO_LOGIN => ['nombre', 'password'],
            self::SCENARIO_REGISTRO => ['nombre', 'correo', 'password'],
        ];
    }
}
```

Atributos activos

- Los **atributos activos** de un escenario son aquellos que están sujetos a *validación* y (posiblemente también) a *asignación masiva* cuando el modelo se encuentra en dicho escenario.
- El método `scenarios()` debe devolver un array cuyas claves son los nombres de los escenarios y cuyos valores son los correspondientes atributos activos de ese escenario.
- La implementación predeterminada de ese método devuelve todos los escenarios y todos los atributos que aparecen alguna vez en una regla de validación.
- Si la implementación predeterminada no nos viniera bien, bastaría con cambiar el método `scenarios()`.

- En este ejemplo, los atributos `nombre` y `password` son activos en los escenarios `login` y `registro`, mientras que en este último escenario también está activo el atributo `correo`:

```php
namespace app\models;

class User extends \yii\db\ActiveRecord
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTRO = 'registro';

    public function scenarios()
    {
        return [
            self::SCENARIO_LOGIN => ['nombre', 'password'],
            self::SCENARIO_REGISTRO => ['nombre', 'correo', 'password'],
        ];
    }
}
```

Reglas de validación

- Cuando los datos de un modelo provienen de los usuarios finales, deben ser validados para garantizar que satisfacen ciertas reglas (llamadas **reglas de validación** o **reglas de negocio**).
- Por ejemplo, en el modelo `ContactForm` debemos asegurarnos de que ningún atributo está vacío y que el atributo `correo` contiene una dirección de correo válida.
- Si el valor de algún atributo no satisface sus reglas de validación, se deberían mostrar los mensajes de error apropiados para que los usuarios puedan corregir sus errores.

- Se puede llamar al método `yii\base\Model::validate()` para validar los datos introducidos:
  - Si no hay ningún error, el método devolverá `true`.
  - En caso contrario, guardará los mensajes de error en la propiedad `yii\base\Model::errors` y devolverá `false`.
- El método usará las reglas de validación declaradas en `yii\base\Model::rules()` para validar cada uno de los atributos que lo necesiten.
- Ese es el método que tenemos que sobreescribir para definir las reglas de validación del modelo.

Por ejemplo:

```php
$modelo = new \app\models\ContactForm;

// Rellena los atributos del modelo con la entrada del usuario:
$modelo->attributes = \Yii::$app->request->post('ContactForm');

if ($modelo->validate()) {
    // Todos los datos de entrada son válidos
} else {
    // La validación falló.
    // $errores guarda en un array los mensajes de error:
    $errores = $modelo->errors;
}
```

Declaración de reglas de validación

- Para declarar las reglas de validación asociadas a un modelo, hay que sobreescribir el método `yii\base\Model::rules()` para que devuelva las reglas que deben satisfacer los atributos del modelo.
- Ejemplo:

```php
public function rules()
{
    return [
        // nombre, correo, asunto y cuerpo son obligatorios:
        [['nombre', 'correo', 'asunto', 'cuerpo'], 'required'],

        // el correo debe ser una dirección de e-mail válida:
        ['correo', 'email'],
    ];
}
```

Reglas activas

- Las **reglas activas** son aquellas reglas que se aplican al *escenario activo*.
- En principio, todas las reglas se aplican a todos los escenarios.
- Sin embargo, podemos indicar que una regla sólo se aplique a un determinado escenario usando la opción `on`.

- Por ejemplo:

```php
public function rules()
{
    return [
        // nombre, correo, asunto y cuerpo son obligatorios:
        [['nombre', 'correo', 'asunto', 'cuerpo'], 'required'],

        // el correo debe ser una dirección de e-mail válida:
        ['correo', 'email', 'on' => User::ESCENARIO_CREAR],
    ];
}
```

- Aquí, la primera regla se aplicaría a todos los escenarios, mientras que la segunda regla sólo se aplicaría al escenario `crear`.
- Por tanto, decimos que la primera regla es una regla activa en todos los escenarios, y la segunda es una regla activa sólo en el escenario `crear`.

Asignación masiva

- La **asignación masiva** permite rellenar varios (o todos los) atributos de un modelo en una sola operación asignando los datos de entrada directamente a la propiedad `yii\base\Model::$attributes`:

```php
$modelo = new \app\models\ContactForm;
$modelo->attributes = \Yii::$app->request->post('ContactForm');

// Es equivalente a (mucho más largo y propenso a errores):

$modelo = new \app\models\ContactForm;
$datos = \Yii::$app->request->post('ContactForm', []);
$modelo->nombre = isset($datos['nombre']) ? $datos['nombre'] : null;
$modelo->correo = isset($datos['correo']) ? $datos['correo'] : null;
$modelo->asunto = isset($datos['asunto']) ? $datos['asunto'] : null;
$modelo->cuerpo = isset($datos['cuerpo']) ? $datos['cuerpo'] : null;
```

Atributos seguros

- La asignación masiva sólo se aplica a los **atributos seguros**.
- De entrada, un atributo activo en un escenario es seguro para ese escenario, salvo que se marque expresamente como inseguro (luego lo vemos).
  - Por tanto, un atributo puede ser activo pero inseguro. En ese caso, estará sujeto a las reglas de validación pero no admitirá asignación masiva.

- Como la implementación predeterminada de `yii\base\Model::scenarios()` devuelve todos los escenarios y atributos que aparecen en `yii\base\Model::rules()`, de entrada todos los atributos son seguros siempre y cuando aparezcan en alguna de las reglas de validación activas.
- Con esa implementación, si queremos que un atributo sea seguro pero no sabemos qué validación aplicarle, podemos marcarlo como `safe` en el método `rules()`:

```
public function rules()
{
    return [
        [['numero', 'codigo'], 'required'],
        ['created_at', 'safe'],
    ];
}
```

Atributos inseguros

- Al contrario, si queremos que un atributo activo sea **inseguro**, es decir, que esté sometido a reglas de validación pero que no admita asignación masiva, podemos marcarlo como tal usando una `!` delante de su nombre, en `scenarios()` o en `rules()`:

```php
public function rules()
{
    return [
        [['numero', 'codigo', '!created_at'], 'required'],
    ];
}
```

- Con la implementación predeterminada de `scenarios()`:
  - `numero`, `codigo` y `created_at` son los atributos activos.
  - Todos están sujetos a la regla de validación del `required`.
  - `created_at` no es seguro, por lo que no puede asignarse masivamente.

Resumen

- El **escenario activo** de un modelo es el escenario en el que se encuentra actualmente el modelo.
  - El escenario predeterminado es `default`.
  - La propiedad `scenario` contiene el escenario activo de ese modelo:

```php
echo $m->scenario; // Devuelve (por ejemplo) 'default'
```

  - Se puede cambiar el escenario activo cambiando el valor de esa propiedad:

```php
$m->scenario = 'crear';
```

- Las **reglas activas** en un escenario son las reglas de validación que se aplican en ese escenario.
  - Por defecto, todas las reglas son activas en todos los escenarios.
  - Con la opción `on` se pueden definir reglas que sólo son activas en un determinado escenario.

```php
public function rules()
{
    return [
        // Esta regla es activa en todos los escenarios:
        [['nombre', 'correo', 'asunto', 'cuerpo'], 'required'],

        // Esta regla sólo es activa en el escenario 'crear':
        ['correo', 'email', 'on' => User::ESCENARIO_CREAR],
    ];
}
```

- Los **atributos activos** en un escenario son los atributos que están sujetos a validación y (posiblemente también) a asignación masiva en ese escenario.
  - El método `scenarios()` devuelve los nombres de los atributos activos en cada escenario del modelo.
  - `$m->scenarios()['s']` devuelve los nombres de los atributos activos en el escenario `s` del modelo `$m`.
  - La implementación predeterminada del método `scenarios()` consulta el método `rules()`, localiza todos los atributos y escenarios que aparecen en él, y con esa información construye el array que devuelve.
  - Se puede cambiar dicha implementación por otra que devuelva directamente los escenarios y atributos activos que nos interesen.

- Los **atributos seguros** en un escenario son aquellos que permiten asignación masiva en ese escenario.
  - Según la implementación predeterminada de `scenarios()`, de entrada todos los atributos activos en un escenario son seguros en ese escenario.
  - En nuestra propia implementación de `scenarios()` o de `rules()`, podemos usar una `!` para marcar un atributo como *inseguro*, lo que haría que fuese activo pero no seguro, es decir, que estaría sujeto a validación pero no permitiría asignación masiva:

```php
public function scenarios()
{
    return [
        // El atributo created_at sería activo pero inseguro:
        'default' => ['numero', 'codigo', '!created_at'];
    ]
}
```

  - Por tanto, todos los atributos seguros son activos, pero no todos los atributos activos son seguros.

```
>>> $a = \app\models\Alquileres::findOne(1)
=> app\models\Alquileres {#250
     id: 1,
     socio_id: 1,
     pelicula_id: 1,
     created_at: "2018-01-16 10:08:19",
     devolucion: "2018-01-17 10:08:19",
   }
>>> $a->scenario
=> "default"
>>> $a->scenarios()
=> [
     "default" => [    // Hay un único escenario llamado 'default'
       "socio_id",     // Atributo activo en 'default'
       "pelicula_id",  // Atributo activo en 'default'
       "!created_at",  // Atributo activo pero inseguro en 'default'
     ],
   ]
>>> $a->attributes = ['created_at' => null];
>>> $a
=> app\models\Alquileres {#250
     id: 1,
     socio_id: 1,
     pelicula_id: 1,
     created_at: "2018-01-16 10:08:19", // No ha cambiado
     devolucion: "2018-01-17 10:08:19",
   }
```

```
>>> $a->validate()
=> true                 // Actualmente valida
>>> $a->created_at = null;
=> null                 // Asignar directamente siempre funciona
>>> $a
=> app\models\Alquileres {#250
     id: 1,
     socio_id: 1,
     pelicula_id: 1,
     created_at: null, // Ha cambiado el valor
     devolucion: "2018-01-17 10:08:19",
   }
>>> $a->validate()
=> false                // No valida, porque created_at debe ser !== null
```

- Aquí se ve que `created_at` es un atributo activo en el escenario `default` porque aparece en `scenarios()`, y por tanto está sujeto a las reglas de validación, pero no es un atributo seguro porque está marcado expresamente como inseguro, y por tanto no admite la asignación masiva.