

# Estructura a gran escala de una aplicación Yii 2

Ricardo Pérez López

IES Doñana, curso 2019/2020



1. Introducción
2. Scripts de entrada
3. Aplicaciones
4. Componentes de aplicación
5. Controladores
6. Modelos
7. Resumen

# 1. Introducción

## 1.1. Introducción

## 1.1. Introducción

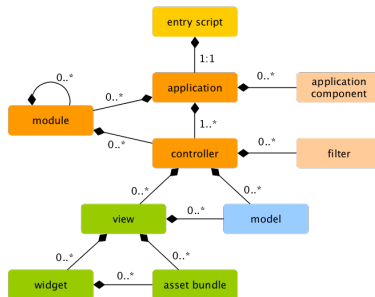


Figura 1: Estructura a gran escala de una aplicación Yii 2

## 2. Scripts de entrada

## 2.1. Scripts de entrada

## 2.1. 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.



## 2.2. Script de entrada de una aplicación web

## 2.2. 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`.

## 2.3. Script de entrada de una aplicación de consola

## 2.3. 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>]
```

## 2.4. ¿Qué hacen los scripts de entrada?

## 2.4. ¿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.

## 2.5. Ejemplo

## 2.5. Ejemplo

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

```
<?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();
```



## 3. Aplicaciones

## 3.1. Aplicaciones

## 3.1. 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`.

## 3.2. Configuración de aplicaciones

## 3.2. 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

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.

### 3.3. Propiedades de la aplicación

## 3.3. 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`).



## 3.4. Propiedades importantes de una aplicación

## 3.4. 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).

## 4. Componentes de aplicación

## 4.1. Componentes de aplicación



## 4.1. 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;
    },
  ],
]
```

## 4.2. Componentes de aplicación principales

## 4.2. 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.

## 5. Controladores

## 5.1. Controladores



## 5.1. 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.

## 5.2. Acciones

## 5.2. 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:

```
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\NotFoundException;  
        }  
  
        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.

```
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.

## 5.3. Rutas

## 5.3. 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:

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

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

## 6. Modelos

## 6.1. Modelos

## 6.1. 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.

**Etiquetas 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.

## 6.2. Atributos

## 6.2. 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:

```
$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`:

```
$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";
}
```

## 6.3. Definición de atributos



## 6.3. 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`:

```
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 sobrescribir 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.

## 6.4. Escenarios

## 6.4. 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:

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

o bien:

```
// 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 sobrescribiendo el método `yii\base\Model::scenarios()`:

```
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`:

```
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'],
        ];
    }
}
```

## 6.5. Reglas de validación

## 6.5. 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 sobrescribir para definir las reglas de validación del modelo.

Por ejemplo:

```
$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 sobrescribir el método `yii\base\Model::rules()` para que devuelva las reglas que deben satisfacer los atributos del modelo.

Ejemplo:

```
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:

```
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`.



## 6.6. Asignación masiva

## 6.6. 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`:

```
$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()`:

```
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.

## 7. Resumen

## 7. 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:

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

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

```
$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.

```
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:

```
public function escenarios()  
{  
    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.