Guía Definitiva de Yii 2.0

http://www.yiiframework.com/doc/guide

Qiang Xue,
Alexander Makarov,
Carsten Brandt,
Klimov Paul,
and
many contributors from the Yii community

Español translation provided by:
Antonio Ramirez,
Daniel Gómez Pan,
Enrique Matías Sánchez (Quique),
'larnu',
Luciano Baraglia

Índice general

Capítulo 1

Introducción

1.1. ¿Qué es Yii?

Yii es un framework de PHP de alto rendimiento, basado en componentes para desarrollar aplicaciones web modernas en poco tiempo. El nombre Yii significa "simple y evolutivo" en chino. También se puede considerar como el acrónimo de ***Yes It Is*** (que en inglés significa ***Sí, lo es***)!

1.1.1. ¿En qué es mejor Yii?

Yii es un framework genérico de programación web, lo que significa que se puede utilizar para desarrollar todo tipo de aplicaciones web en PHP. Debido a su arquitectura basada en componentes y a su sofisticada compatibilidad de caché, es especialmente apropiado para el desarrollo de aplicaciones de gran envergadura, como páginas web, foros, sistemas de gestión de contenidos (CMS), proyectos de comercio electrónico, servicios web compatibles con la arquitectura REST y muchos más.

1.1.2. ¿Cómo se compara Yii con otros frameworks?

Si estás ya familiarizado con otros framework, puedes apreciar como se compara Yii con ellos:

- Como la mayoría de los framework de PHP, Yii implementa el patrón de diseño MVC (Modelo-Vista-Controlador) y promueve la organización de código basada en este patrón.
- La filosofía de Yii consiste en escribir el código de manera simple y elegante, sin sobrediseñar nunca por el mero hecho de seguir un patrón de diseño determinado.
- Yii es un framework completo (full stack) que provee muchas características probadas y listas para usar, como los constructores de consultas y la clase ActiveRecord para las bases de datos relacionales y

NoSQL, la compatibilidad con la arquitectura REST para desarrollar API, la compatibilidad de caché en varios niveles y muchas más.
- Yii es extremadamente extensible. Puedes personalizar o reemplazar prácticamente cualquier pieza de código de base, como se puede también aprovechar su sólida arquitectura de extensiones para utilizar o desarrollar extensiones distribuibles.
- El alto rendimiento es siempre la meta principal de Yii.

Yii no es un proyecto de un sola persona, detrás de Yii hay un sólido equipo de desarrollo[1], así como una gran comunidad en la que numerosos profesionales contribuyen constantemente a su desarrollo. El equipo de desarrollo de Yii se mantiene atento a las últimas tendencias de desarrollo web, así como a las mejores prácticas y características de otros frameworks y proyectos. Las buenas prácticas y características más relevantes de otros proyectos se incorporan regularmente a la base del framework y se exponen a través de interfaces simples y elegantes.

1.1.3. Versiones de Yii

Actualmente existen dos versiones principales de Yii: la versión 1.1 y la versión 2.0. Para la versión 1.1, que es de la generación anterior, actualmente solo se ofrece mantenimiento. La versión 2.0 está completamente reescrita y adopta las últimas tecnologías y protocolos, incluidos Composer, PSR, namespaces, traits, etc. La versión 2.0 representa la actual generación del framework y su desarrollo recibirá el principal esfuerzo en los próximos años. Esta guía está basada principalmente en la versión 2.0. del framework.

1.1.4. Requisitos y Prerequisitos

Yii 2.0 requiere PHP 7.3.0 o una versión posterior y corre de mejor manera en la última versión de PHP. Se pueden encontrar requisitos más detallados de características individuales ejecutando el script de comprobación incluido en cada lanzamiento de Yii.

Para utilizar Yii se requieren conocimientos básicos de programación orientada a objetos (POO), porque el framework Yii se basa íntegramente en esta tecnología. Yii 2.0 hace uso también de las últimas características de PHP, como namespaces[2] y traits[3]. Comprender estos conceptos te ayudará a entender mejor Yii 2.0.

[1] https://www.yiiframework.com/team/
[2] https://www.php.net/manual/es/language.namespaces.php
[3] https://www.php.net/manual/es/language.oop5.traits.php

1.2. Actualizar desde Yii 1.1

Existen muchas diferencias entre las versiones 1.1 y 2.0 de Yii ya que el framework fue completamente reescrito en su segunda versión. Como resultado, actualizar desde la versión 1.1 no es tan trivial como actualizar entre versiones menores. En esta guía encontrarás las diferencias más grandes entre estas dos versiones.

Si no has utilizado Yii 1.1 antes, puedes saltarte con seguridad esta sección e ir directamente a "Comenzando con Yii".

Es importante anotar que Yii 2.0 introduce más características de las que van a ser cubiertas en este resumen. Es altamente recomendado que leas a través de toda la guía definitiva para aprender acerca de todas ellas. Hay muchas posibilidades de que algo que hayas desarrollado anteriormente para extender Yii, sea ahora parte del núcleo de la librería.

1.2.1. Instalación

Yii 2.0 adopta íntegramente Composer[4], el administrador de paquetes de facto de PHP. Tanto la instalación del núcleo del framework como las extensiones se manejan a través de Composer. Por favor consulta la sección Comenzando con la Aplicación Básica para aprender a instalar Yii 2.0. Si quieres crear extensiones o transformar extensiones de Yii 1.1 para que sean compatibles con Yii 2.0, consulta la sección Creando Extensiones de la guía.

1.2.2. Requerimientos de PHP

Yii 2.0 requiere PHP 5.4 o mayor, lo que es un gran progreso ya que Yii 1.1 funcionaba con PHP 5.2. Como resultado, hay muchas diferencias a nivel del lenguaje a las que deberías prestar atención. Abajo hay un resumen de los mayores cambios en relación a PHP:

- Namespaces[5].
- Funciones anónimas[6].
- La sintaxis corta de Arrays `[...elementos...]` es utilizada en vez de `array(...elementos...)`.
- Etiquetas cortas de `echo`. Ahora en las vistas se usa `<?=`. Esto se puede utilizar desde PHP 5.4.
- SPL - Biblioteca estándar de PHP[7].
- Enlace estático en tiempo de ejecución[8].
- Fecha y Hora[9].

[4] https://getcomposer.org/
[5] https://www.php.net/manual/es/language.namespaces.php
[6] https://www.php.net/manual/es/functions.anonymous.php
[7] https://www.php.net/manual/es/book.spl.php
[8] https://www.php.net/manual/es/language.oop5.late-static-bindings.php
[9] https://www.php.net/manual/es/book.datetime.php

- Traits[10].
- intl[11]. Yii 2.0 utiliza la extensión `intl` de PHP como soporte para internacionalización.

1.2.3. Namespace

El cambio más obvio en Yii 2.0 es el uso de namespaces. Casi todas las clases del núcleo utilizan namespaces, ej., `yii\web\Request`. El prefijo "C" no se utiliza más en los nombre de clases. El esquema de nombres sigue la estructura de directorios. Por ejemplo, `yii\web\Request` indica que el archivo de la clase correspondiente `web/Request.php` está bajo el directorio de Yii framework.

(Puedes utilizar cualquier clase del núcleo sin necesidad de incluir el archivo que la contiene, gracias al autoloader de Yii.)

1.2.4. Componentes y Objetos

Yii 2.0 parte la clase `CComponent` de 1.1 en dos clases: `yii\base\BaseObject` y `yii\base\Component`. La clase `BaseObject` es una clase base que permite definir propiedades de object a través de getters y setters. La clase `Component` extiende de `BaseObject` y soporta eventos y comportamientos.

Si tu clase no necesita utilizar las características de eventos o comportamientos, puedes considerar usar `BaseObject` como clase base. Esto es frecuente en el caso de que las clases que representan sean estructuras de datos básicas.

1.2.5. Configuración de objetos

La clase `BaseObject` introduce una manera uniforme de configurar objetos. Cualquier clase descendiente de `BaseObject` debería declarar su constructor (si fuera necesario) de la siguiente manera para que puede ser adecuadamente configurado:

```
class MyClass extends \yii\base\BaseObject
{
    public function __construct($param1, $param2, $config = [])
    {
        // ... se aplica la inicialización antes de la configuración

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();
```

[10] https://www.php.net/manual/es/language.oop5.traits.php
[11] https://www.php.net/manual/es/book.intl.php

```
        // ... se aplica la inicialización después de la configuración
    }
}
```

En el ejemplo de arriba, el último parámetro del constructor debe tomar un array de configuración que contiene pares clave-valor para la inicialización de las propiedades al final del mismo. Puedes sobrescribir el método `init()` para realizar el trabajo de inicialización que debe ser hecho después de que la configuración haya sido aplicada.

Siguiendo esa convención, podrás crear y configurar nuevos objetos utilizando un array de configuración:

```
$object = Yii::createObject([
    'class' => 'MyClass',
    'property1' => 'abc',
    'property2' => 'cde',
], [$param1, $param2]);
```

Se puede encontrar más detalles acerca del tema en la sección Configuración.

1.2.6. Eventos

En Yii 1, los eventos eran creados definiendo un método `on` (ej., `onBeforeSave`). En Yii 2, puedes utilizar cualquier nombre de evento. Ahora puedes disparar un evento utilizando el método `trigger()`:

```
$event = new \yii\base\Event;
$component->trigger($eventName, $event);
```

Para conectar un manejador a un evento, utiliza el método `on()`:

```
$component->on($eventName, $handler);
// Para desconectar el manejador, utiliza:
// $component->off($eventName, $handler);
```

Hay muchas mejoras en lo que respecta a eventos. Para más detalles, consulta la sección Eventos.

1.2.7. Alias

Yii 2.0 extiende el uso de alias tanto para archivos/directorios como URLs. Yii 2.0 ahora requiere que cada alias comience con el carácter `@`, para diferenciarlos de rutas o URLs normales. Por ejemplo, el alias `@yii` corresponde al directorio donde Yii se encuentra instalado. Los alias están soportados en la mayor parte del núcleo. Por ejemplo, `yii\caching\FileCache::$cachePath` puede tomar tanto una ruta de directorios normal como un alias.

Un alias está estrechamente relacionado con un namespace de la clase. Se recomienda definir un alias por cada namespace raíz, y así poder utilizar el autoloader de Yii sin otra configuración. Por ejemplo, debido a que `@yii` se refiere al directorio de instalación, una clase como `yii\web\Request` puede ser auto-cargada. Si estás utilizando una librería de terceros, como Zend Framework, puedes definir un alias `@Zend` que se refiera al directorio de instalación de ese framework. Una vez realizado esto, Yii será capaz de auto-cargar cualquier clase de Zend Framework también.

Se puede encontrar más detalles del tema en la sección Alias.

1.2.8. Vistas

El cambio más significativo con respecto a las vistas en Yii 2 es que la variable especial `$this` dentro de una vista ya no se refiere al controlador o widget actual. En vez de eso, `$this` ahora se refiere al objeto de la *vista*, un concepto nuevo introducido en Yii 2.0. El objeto *vista* es del tipo `yii\web\View`, que representa la parte de las vistas en el patrón MVC. Si quieres acceder al controlador o al widget correspondiente desde la propia vista, puedes utilizar `$this->context`.

Para renderizar una vista parcial (partial) dentro de otra vista, se utiliza `$this->render()`, no `$this->renderPartial()`. La llamada a `render` además tiene que ser mostrada explícitamente a través de `echo`, ya que el método `render()` devuelve el resultado de la renderización en vez de mostrarlo directamente. Por ejemplo:

```
echo $this->render('_item', ['item' => $item]);
```

Además de utilizar PHP como el lenguaje principal de plantillas (templates), Yii 2.0 está también equipado con soporte oficial de otros dos motores de plantillas populares: Smarty y Twig. El motor de plantillas de Prado ya no está soportado. Para utilizar esos motores, necesitas configurar el componente `view` de la aplicación, definiendo la propiedad `View::$renderers`. Por favor consulta la sección Motores de Plantillas para más detalles.

1.2.9. Modelos

Yii 2.0 utiliza `yii\base\Model` como modelo base, algo similar a `CModel` en 1.1. La clase `CFormModel` ha sido descartada por completo. Ahora, en Yii 2 debes extender de `yii\base\Model` para crear clases de modelos basados en formularios.

Yii 2.0 introduce un nuevo método llamado `scenarios()` para declarar escenarios soportados, y para indicar bajo que escenario un atributo necesita ser validado, puede ser considerado seguro o no, etc. Por ejemplo:

```
public function scenarios()
{
```

```
    return [
        'backend' => ['email', 'role'],
        'frontend' => ['email', '!role'],
    ];
}
```

En el ejemplo anterior, se declaran dos escenarios: `backend` y `frontend`. Para el escenario `backend` son considerados seguros ambos atributos, `email` y `role`, y pueden ser asignados masivamente. Para el escenario `frontend`, `email` puede ser asignado masivamente mientras `role` no. Tanto `email` como `role` deben ser validados utilizando reglas (rules).

El método `rules()` aún es utilizado para declara reglas de validación. Ten en cuenta que dada la introducción de `scenarios()`, ya no existe el validador `unsafe`.

En la mayoría de los casos, no necesitas sobrescribir `scenarios()` si el método `rules()` especifica completamente los escenarios que existirán, y si no hay necesidad de declarar atributos inseguros (`unsafe`).

Para aprender más detalles de modelos, consulta la sección Modelos.

1.2.10. Controladores

Yii 2.0 utiliza `yii\web\Controller` como controlador base, similar a `CWebController` en Yii 1.1. `yii\base\Action` es la clase base para clases de acciones.

El impacto más obvio de estos cambios en tu código es que que cada acción del controlador debe devolver el contenido que quieres mostrar en vez de mostrarlo directamente:

```
public function actionView($id)
{
    $model = \app\models\Post::findOne($id);
    if ($model) {
        return $this->render('view', ['model' => $model]);
    } else {
        throw new \yii\web\NotFoundHttpException;
    }
}
```

Por favor, consulta la sección Controladores para más detalles acerca de los controladores.

1.2.11. Widgets

Yii 2.0 utiliza `yii\base\Widget` como clase base de los widgets, similar a `CWidget` en Yii 1.1.

Para obtener mejor soporte del framework en IDEs, Yii 2.0 introduce una nueva sintaxis para utilizar widgets. Los métodos estáticos `begin()`, `end()`, y `widget()` fueron incorporados, y deben utilizarse así:

```php
use yii\widgets\Menu;
use yii\widgets\ActiveForm;

// Ten en cuenta que debes pasar el resultado a "echo" para mostrarlo
echo Menu::widget(['items' => $items]);

// Pasando un array para inicializar las propiedades del objeto
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge']],
]);
... campos del formulario aquí ...
ActiveForm::end();
```

Consulta la sección Widgets para más detalles.

1.2.12. Temas

Los temas funcionan completamente diferente en Yii 2.0. Ahora están basados en un mecanismo de mapeo de rutas, que mapea la ruta de un archivo de la vista de origen a uno con un tema aplicado. Por ejemplo, si el mapeo de ruta de un tema es `['/web/views' => '/web/themes/basic']`, entonces la versión con el tema aplicado del archivo `/web/views/site/index.php` será `/web/themes/basic/site/index.php`. Por esta razón, ahora los temas pueden ser aplicados a cualquier archivo de la vista, incluso una vista renderizada fuera del contexto de un controlador o widget.

Además, el componente `CThemeManager` ya no existe. En cambio, `theme` es una propiedad configurable del componente `view` de la aplicación.

Consulta la sección Temas para más detalles.

1.2.13. Aplicaciones de Consola

Las aplicaciones de consola ahora están organizadas en controladores, tal como aplicaciones Web. Estos controladores deben extender de `yii\console\Controller`, similar a `CConsoleCommand` en 1.1.

Para correr un comando de consola, utiliza `yii <ruta>`, donde `<ruta>` se refiere a la ruta del controlador (ej. `sitemap/index`). Los argumentos anónimos adicionales son pasados como parámetros al método de la acción correspondiente del controlador, mientras que los argumentos especificados son pasados de acuerdo a las declaraciones en `yii\console\Controller::options()`.

Yii 2.0 soporta la generación automática de información de ayuda de los comandos a través de los bloques de comentarios del archivo.

Por favor consulta la sección Comandos de Consola para más detalles.

1.2.14. I18N

Yii 2.0 remueve el formateador de fecha y números previamente incluido en favor del módulo de PHP PECL intl[12].

La traducción de mensajes ahora es ejecutada vía el componente `i18n` de la aplicación. Este componente maneja un grupo de mensajes origen, lo que te permite utilizar diferentes mensajes basados en categorías.

Por favor, consulta la sección Internacionalización para más información.

1.2.15. Filtros de Acciones

Los filtros de acciones son implementados a través de comportamientos. Para definir un nuevo filtro personalizado, se debe extender de `yii\base\ActionFilter`. Para utilizar el filtro, conecta la clase del filtro al controlador como un comportamiento. Por ejemplo, para utilizar el filtro `yii\filters\AccessControl`, deberías tener el siguiente código en el controlador:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => 'yii\filters\AccessControl',
            'rules' => [
                ['allow' => true, 'actions' => ['admin'], 'roles' => ['@']],
            ],
        ],
    ];
}
```

Consulta la sección Filtrando para una mayor información acerca del tema.

1.2.16. Assets

Yii 2.0 introduce un nuevo concepto llamado *asset bundle* que reemplaza el concepto de script package encontrado en Yii 1.1.

Un asset bundle es una colección de archivos assets (ej. archivos JavaScript, archivos CSS, imágenes, etc.) dentro de un directorio. Cada asset bundle está representado por una clase que extiende de `yii\web\AssetBundle`. Al registrar un asset bundle a través de `yii\web\AssetBundle::register()`, haces que los assets de dicho bundle sean accesibles vía Web. A diferencia de Yii 1, la página que registra el bundle contendrá automáticamente las referencias a los archivos JavaScript y CSS especificados en el bundle.

Por favor, consulta la sección Manejando Assets para más detalles.

[12] https://pecl.php.net/package/intl

1.2.17. Helpers

Yii 2.0 introduce muchos helpers estáticos comúnmente utilizados, incluyendo:

- `yii\helpers\Html`
- `yii\helpers\ArrayHelper`
- `yii\helpers\StringHelper`
- `yii\helpers\FileHelper`
- `yii\helpers\Json`

Por favor, consulta la sección Información General de Helpers para más detalles.

1.2.18. Formularios

Yii 2.0 introduce el concepto de *campo* (field) para construir formularios utilizando `yii\widgets\ActiveForm`. Un campo es un contenedor que consiste en una etiqueta, un input, un mensaje de error y/o texto de ayuda. Un campo es representado como un objeto `ActiveField`. Utilizando estos campos, puedes crear formularios más legibles que antes:

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <div class="form-group">
        <?= Html::submitButton('Login') ?>
    </div>
<?php yii\widgets\ActiveForm::end(); ?>
```

Por favor, consulta la sección Creando Formularios para más detalles.

1.2.19. Constructor de Consultas

En Yii 1.1, la generación de consultas a la base de datos estaba dividida en varias clases, incluyendo `CDbCommand`, `CDbCriteria`, y `CDbCommandBuilder`. Yii 2.0 representa una consulta a la base de datos en términos de un objeto `Query` que puede ser convertido en una declaración SQL con la ayuda de `QueryBuilder` detrás de la escena. Por ejemplo:

```
$query = new \yii\db\Query();
$query->select('id, name')
      ->from('user')
      ->limit(10);

$command = $query->createCommand();
$sql = $command->sql;
$rows = $command->queryAll();
```

Lo mejor de todo, dichos métodos de generación de consultas pueden ser también utilizados mientras se trabaja con Active Record.

Consulta la sección Constructor de Consultas para más detalles.

1.2.20. Active Record

Yii 2.0 introduce muchísimos cambios con respecto a Active Record. Los dos más obvios se relacionan a la generación de consultas y al manejo de relaciones.

La clase de Yii 1.1 `CDbCriteria` es reemplazada por `yii\db\ActiveQuery` en Yii 2. Esta clase extiende de `yii\db\Query`, y por lo tanto hereda todos los métodos de generación de consultas. Para comenzar a generar una consulta, llamas al método `yii\db\ActiveRecord::find()`:

```
// Recibe todos los clientes *activos* y ordenados por su ID:
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

Para declarar una relación, simplemente define un método getter que devuelva un objeto `ActiveQuery`. El nombre de la propiedad definida en el getter representa el nombre de la relación. Por ejemplo, el siguiente código declara una relación `orders` (en Yii 1.1, las relaciones se declaraban centralmente en el método `relations()`):

```
class Customer extends \yii\db\ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany('Order', ['customer_id' => 'id']);
    }
}
```

Ahora puedes utilizar `$customer->orders` para acceder a las órdenes de la tabla relacionada. También puedes utilizar el siguiente código para realizar una consulta relacional 'sobre la marcha' con una condición personalizada:

```
$orders = $customer->getOrders()->andWhere('status=1')->all();
```

Cuando se utiliza la carga temprana (eager loading) de la relación, Yii 2.0 lo hace diferente de 1.1. En particular, en 1.1 una declaración JOIN sería creada para seleccionar tanto los registros de la tabla primaria como los relacionados. En Yii 2.0, dos declaraciones SQL son ejecutadas sin utilizar un JOIN: la primera trae todos los modelos primarios, mientras que la segunda trae los registros relacionados utilizando como condición la clave primaria de los primarios.

En vez de devolver objetos `ActiveRecord`, puedes conectar el método `asArray()` mientras generas una consulta que devuelve un gran número de registros. Esto causará que el resultado de la consulta sea devuelto como arrays, lo que puede reducir significativamente la necesidad de tiempo de CPU y memoria si el número de registros es grande. Por ejemplo:

```
$customers = Customer::find()->asArray()->all();
```

Otro cambio es que ya no puedes definir valores por defecto a los atributos a través de propiedades públicas. Si lo necesitaras, debes definirlo en el método `init` de la clase del registro en cuestión.

```
public function init()
{
    parent::init();
    $this->status = self::STATUS_NEW;
}
```

Anteriormente, solía haber algunos problemas al sobrescribir el constructor de una clase ActiveRecord en 1.1. Estos ya no están presentes en Yii 2.0. Ten en cuenta que al agregar parámetros al constructor podrías llegar a tener que sobrescribir `yii\db\ActiveRecord::instantiate()`.

Hay muchos otros cambios y mejoras con respecto a ActiveRecord. Por favor, consulta la sección Active Record para más detalles.

1.2.21. Active Record Behaviors

En 2.0, hemos eliminado la clase del comportamiento base `CActiveRecordBehavior`. Si desea crear un comportamiento Active Record, usted tendrá que extender directamente de `yii\base\Behavior`. Si la clase de comportamiento debe responder a algunos eventos propios, usted tiene que sobrescribir los métodos `events()` como se muestra a continuación,

```
namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
        // ...
    }
}
```

1.2.22. User e IdentityInterface

La clase `CWebUser` de 1.1 es reemplazada por `yii\web\User`, y la clase `CUserIdentity` ha dejado de existir. En cambio, ahora debes implementar `yii\web\IdentityInterface` el cual es mucho más directo de usar. El template de proyecto avanzado provee un ejemplo así.

Consulta las secciones Autenticación, Autorización, y Template de Proyecto Avanzado[13] para más detalles.

1.2.23. Manejo de URLs

El manejo de URLs en Yii 2 es similar al de 1.1. Una mejora mayor es que el manejador actual ahora soporta parámetros opcionales. Por ejemplo, si tienes una regla declarada como a continuación, entonces coincidirá tanto con `post/popular` como con `post/1/popular`. En 1.1, tendrías que haber creado dos reglas diferentes para obtener el mismo resultado

```
[
    'pattern' => 'post/<page:\d+>/<tag>',
    'route' => 'post/index',
    'defaults' => ['page' => 1],
]
```

Por favor, consulta la sección Documentación del Manejo de URLs para más detalles.

Un cambio importante en la convención de nombres para rutas es que los nombres en CamelCase de controladores y acciones ahora son convertidos a minúsculas y cada palabra separada por un guión, por ejemplo el id del controlador `CamelCaseController` será `camel-case`. Consulta la sección acerca de IDs de controladores y IDs de acciones para más detalles.

1.2.24. Utilizar Yii 1.1 y 2.x juntos

Si tienes código en Yii 1.1 que quisieras utilizar junto con Yii 2.0, por favor consulta la sección Utilizando Yii 1.1 y 2.0 juntos.

[13] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide-es/README.md

Capítulo 2

Primeros pasos

2.1. Qué necesita saber

La curva de aprendizaje de Yii no es tan empinada como en otros *frameworks* en PHP, pero todavía hay algunas cosas que debería aprender antes de empezar con Yii.

2.1.1. PHP

Yii es un *framework* (base estructurada de desarrollo) en PHP, así que asegúrese de leer y comprender la referencia del lenguaje[1]. Al desarrollar con Yii deberá escribir código de manera orientada a objetos, así que asegúrese de estar familiarizado con clases y objetos[2] así como con espacios de nombres[3].

2.1.2. Programación orientada a objetos

Se requiere una comprensión básica de la programación orientada a objetos. Si no está familiarizado con ella, diríjase a alguno de los muchos tutoriales disponibles, como el de tuts+[4].

Observe que cuanto más complicada sea su aplicación, más conceptos avanzados de la POO deberá aprender para gestionar con éxito esa complejidad.

2.1.3. Línea de órdenes y composer

Yii usa profusamente el gestor de paquetes *de facto* de PHP, Composer[5], así que asegúrese de leer y comprender su guía[6]. Si no está familiarizado con

[1] https://www.php.net/manual/es/langref.php
[2] https://www.php.net/manual/es/language.oop5.basic.php
[3] https://www.php.net/manual/es/language.namespaces.php
[4] https://code.tutsplus.com/tutorials/object-oriented-php-for-beginners--net-12762
[5] https://getcomposer.org/
[6] https://getcomposer.org/doc/01-basic-usage.md

el uso de la línea de órdenes, es hora de empezar a probarla. Una vez que aprenda los fundamentos, nunca querrá trabajar sin ella.

2.2. Instalar Yii

Puedes instalar Yii de dos maneras, utilizando el administrador de paquetes Composer[7] o descargando un archivo comprimido. La forma recomendada es la primera, ya que te permite instalar nuevas extensions o actualizar Yii con sólo ejecutar un comando.

La instalación estándar de Yii cuenta tanto con el framework como un template de proyecto instalados. Un template de proyecto es un proyecto Yii funcional que implementa algunas características básicas como: login, formulario de contacto, etc. El código está organizado de una forma recomendada. Por lo tanto, puede servir como un buen punto de partida para tus proyectos.

En esta y en las próximas secciones, describiremos cómo instalar Yii con el llamado *Template de Proyecto Básico* y cómo implementar nuevas características por encima del template. Yii también provee otro template llamado Template de Proyecto Avanzado[8] qué es mejor para desarrollar aplicaciones con varios niveles en el entorno de un equipo de desarrollo.

> **Información:** El Template de Proyecto Básico es adecuado para desarrollar el 90 porciento de las aplicaciones Web. Difiere del Template de Proyecto Avanzado principalmente en cómo está organizado el código. Si eres nuevo en Yii, te recomendamos utilizar el Template de Proyecto Básico por su simplicidad pero funcionalidad suficiente.

2.2.1. Instalando via Composer

Si aún no tienes Composer instalado, puedes hacerlo siguiendo las instrucciones que se encuentran en getcomposer.org[9]. En Linux y Mac OS X, se ejecutan los siguientes comandos:

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

En Windows, tendrás que descargar y ejecutar Composer-Setup.exe[10].

Por favor, consulta la Documentación de Composer[11] si encuentras algún problema o deseas obtener un conocimiento más profundo sobre su utilización.

[7] https://getcomposer.org/

[8] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/README.md

[9] https://getcomposer.org/download/

[10] https://getcomposer.org/Composer-Setup.exe

[11] https://getcomposer.org/doc/

Si ya tienes composer instalado, asegúrate de tener una versión actualizada. Puedes actualizar Composer ejecutando el comando `composer self-update`

Teniendo Composer instalado, puedes instalar Yii ejecutando los siguientes comandos en un directorio accesible vía Web:

```
composer global require "fxp/composer-asset-plugin:^1.4.1"
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

El primer comando instala composer asset plugin[12], que permite administrar dependencias de paquetes bower y npm a través de Composer. Sólo necesitas ejecutar este comando una vez. El segundo comando instala Yii en un directorio llamado `basic`. Puedes elegir un nombre de directorio diferente si así lo deseas.

> **Nota:** Durante la instalación, Composer puede preguntar por tus credenciales de acceso de Github. Esto es normal ya que Composer necesita obtener suficiente límite de acceso de la API para traer la información de dependencias de Github. Para más detalles, consulta la documentación de Composer[13].
>
> **Consejo:** Si quieres instalar la última versión de desarrollo de Yii, puedes utilizar uno de los siguientes comandos, que agregan una opción de estabilidad[14]:
>
> ```
> composer create-project --prefer-dist --stability=dev
> yiisoft/yii2-app-basic basic
> ```
>
> Ten en cuenta que la versión de desarrollo de Yii no debería ser utilizada en producción ya que podría romper tu código actual.

2.2.2. Instalar desde un Archivo Comprimido

Instalar Yii desde un archivo comprimido involucra tres pasos:

1. Descargar el archivo desde yiiframework.com[15].
2. Descomprimirlo en un directorio accesible vía Web.
3. Modificar el archivo `config/web.php` introduciendo una clave secreta para el ítem de configuración `cookieValidationKey` (esto se realiza automáticamente si estás instalando Yii a través de Composer):

```
// !!! insert a secret key in the following (if it is empty) - this is
required by cookie validation
'cookieValidationKey' => 'enter your secret key here',
```

[12] https://github.com/fxpio/composer-asset-plugin
[13] https://getcomposer.org/doc/articles/troubleshooting.md#api-rate-limit-and-oauth-tokens
[14] https://getcomposer.org/doc/04-schema.md#minimum-stability
[15] https://www.yiiframework.com/download/yii2-basic

2.2.3. Otras Opciones de Instalación

Las instrucciones anteriores muestran cómo instalar Yii, lo que también crea una aplicación Web lista para ser usada. Este es un buen punto de partida para la mayoría de proyectos, tanto grandes como pequeños. Es especialmente adecuado si recién estás aprendiendo a utilizar Yii.

Pero también hay otras opciones de instalación disponibles:

- Si sólo quieres instalar el núcleo del framework y entonces crear una nueva aplicación desde cero, puedes seguir las instrucciones explicadas en Generando una Aplicación desde Cero.
- Si quisieras comenzar con una aplicación más avanzada, más adecuada para un entorno de desarrollo de equipo, deberías considerar instalar el Template de Aplicación Avanzada.

2.2.4. Verificando las Instalación

Una vez finalizada la instalación, o bien configura tu servidor web (mira la sección siguiente) o utiliza el servidor web incluido en PHP[16] ejecutando el siguiente comando de consola estando parado en el directorio `web` de la aplicación:

```
php yii serve
```

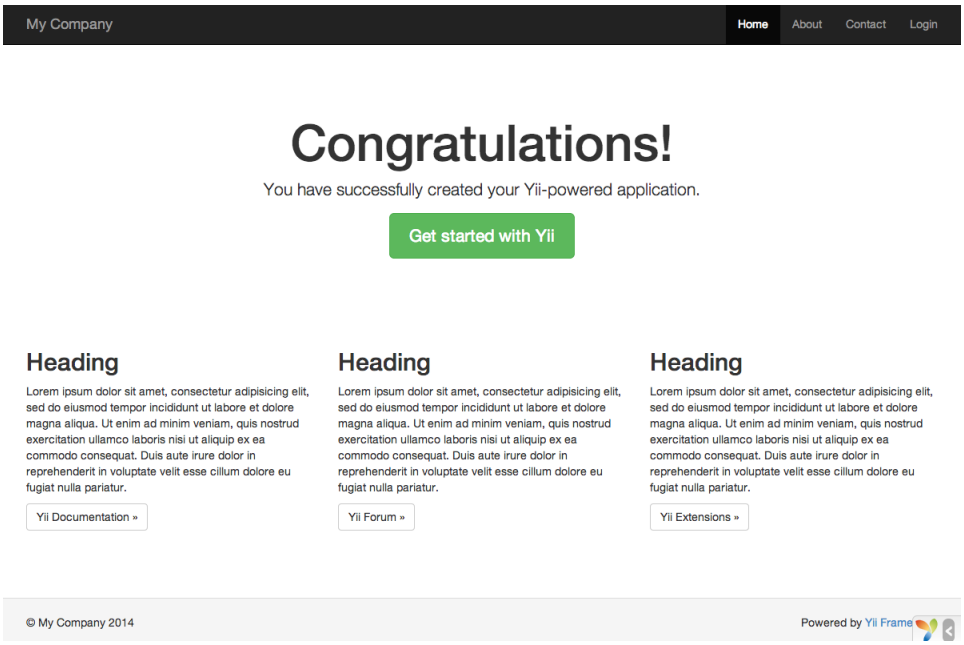
> **Nota:** Por defecto el servidor HTTP escuchará en el puerto 8080. De cualquier modo, si el puerto está en uso o deseas servir varias aplicaciones de esta manera, podrías querer especificar qué puerto utilizar. Sólo agrega el argumento –port:

```
php yii serve --port=8888
```

Puedes utilizar tu navegador para acceder a la aplicación instalada de Yii en la siguiente URL:

```
http://localhost:8080/.
```

[16] https://www.php.net/manual/es/features.commandline.webserver.php



Deberías ver la página mostrando "Congratulations!" en tu navegador. Si no ocurriera, por favor chequea que la instalación de PHP satisfaga los requerimientos de Yii. Esto puedes hacerlo usando cualquiera de los siguientes procedimientos:

- Copiando `/requirements.php` a `/web/requirements.php` y visitando la URL `http://localhost/basic/requirements.php` en tu navegador
- Corriendo los siguientes comandos:

```
cd basic
php requirements.php
```

Deberías configurar tu instalación de PHP para que satisfaga los requisitos mínimos de Yii. Lo que es más importante, debes tener PHP 5.4 o mayor. También deberías instalar la Extensión de PHP PDO[17] y el correspondiente driver de base de datos (como `pdo_mysql` para bases de datos MySQL), si tu aplicación lo necesitara.

2.2.5. Configurar Servidores Web

> **Información:** Puedes saltear esta sección por ahora si sólo estás probando Yii sin intención de poner la aplicación en un servidor de producción.

La aplicación instalada siguiendo las instrucciones mencionadas debería estar lista para usar tanto con un servidor HTTP Apache[18] como con un servidor HTTP Nginx[19], en Windows, Mac OS X, o Linux utilizando PHP 5.4 o

[17] https://www.php.net/manual/es/pdo.installation.php
[18] https://httpd.apache.org/
[19] https://nginx.org/

mayor. Yii 2.0 también es compatible con HHVM[20] de Facebook. De todos modos, hay algunos casos donde HHVM se comporta diferente del PHP oficial, por lo que tendrás que tener cuidados extra al utilizarlo.

En un servidor de producción, podrías querer configurar el servidor Web para que la aplicación sea accedida a través de la URL `https://www.example.com/index.php` en vez de `https://www.example.com/basic/web/index.php`. Tal configuración requiere apuntar el document root de tu servidor Web a la carpeta `basic/web`. También podrías querer ocultar `index.php` de la URL, como se describe en la sección Parseo y Generación de URLs. En esta sub-sección, aprenderás a configurar tu servidor Apache o Nginx para alcanzar estos objetivos.

> **Información:** Al definir `basic/web` como document root, también previenes que los usuarios finales accedan al código privado o archivos con información sensible de tu aplicación que están incluidos en los directorios del mismo nivel que `basic/web`. Denegando el acceso es una importante mejora en la seguridad.

> **Información:** En caso de que tu aplicación corra en un entorno de hosting compartido donde no tienes permisos para modificar la configuración del servidor Web, aún puedes ajustar la estructura de la aplicación para mayor seguridad. Por favor consulta la sección Entorno de Hosting Compartido para más detalles.

Configuración Recomendada de Apache

Utiliza la siguiente configuración del archivo `httpd.conf` de Apache dentro de la configuración del virtual host. Ten en cuenta que deberás reemplazar `path/to/basic/web` con la ruta real a `basic/web`.

```
# Definir el document root como "basic/web"
DocumentRoot "path/to/basic/web"

<Directory "path/to/basic/web">
    # utiliza mod_rewrite para soporte de URLs amigables
    RewriteEngine on
    # Si el directorio o archivo existe, utiliza la petición directamente
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    # Sino, redirige la petición a index.php
    RewriteRule . index.php

    # ...otras configuraciones...
</Directory>
```

[20] https://hhvm.com/

Configuración Recomendada de Nginx

Para utilizar Nginx[21], debes instalar PHP como un FPM SAPI[22]. Utiliza la siguiente configuración de Nginx, reemplazando `path/to/basic/web` con la ruta real a `basic/web` y `mysite.test` con el hostname real a servir.

```
server {
    charset utf-8;
    client_max_body_size 128M;

    listen 80; ## listen for ipv4
    #listen [::]:80 default_server ipv6only=on; ## listen for ipv6

    server_name mysite.test;
    root        /path/to/basic/web;
    index       index.php;

    access_log  /path/to/basic/log/access.log;
    error_log   /path/to/basic/log/error.log;

    location / {
        # Redireccionar a index.php todo lo que no sea un archivo real
        try_files $uri $uri/ /index.php$is_args$args;
    }

    # descomentar para evitar el procesamiento de llamadas de Yii a archivos
    estáticos no existente
    #location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.html;

    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass   127.0.0.1:9000;
        #fastcgi_pass unix:/var/run/php5-fpm.sock;
        try_files $uri =404;
    }

    location ~ /\.(ht|svn|git) {
        deny all;
    }
}
```

Al utilizar esta configuración, también deberías definir `cgi.fix_pathinfo=0` en el archivo `php.ini`, y así evitar muchas llamadas innecesarias del sistema a `stat()`.

Ten en cuenta también que al correr un servidor HTTPS, deberás agregar `fastcgi_param HTTPS on;` así Yii puede detectar propiamente si la conexión es

[21] https://wiki.nginx.org/
[22] https://www.php.net/manual/es/install.fpm.php

segura.

2.3. Corriendo Aplicaciones

Después de haber instalado Yii, tienes una aplicación totalmente funcional a la que se puede acceder a través de la URL `https://hostname/basic/web/index.php` o `https://hostname/index.php`, dependiendo de tu configuración. Esta sección será una introducción a la funcionalidad incluida de la aplicación, cómo se organiza el código, y cómo la aplicación maneja los requests en general.

> **Información:** Por simplicidad, en el transcurso de este tutorial "Para Empezar", se asume que has definido `basic/web` como el document root de tu servidor Web, y configurado la URL de acceso a tu aplicación para que sea `https://hostname/index.php` o similar. Dependiendo de tus necesidades, por favor ajusta dichas URLs.

Ten en cuenta que a diferencia del framework en sí, después de que el template de proyecto es instalado, este es todo tuyo. Eres libre de agregar o eliminar código modificar todo según tu necesidad.

2.3.1. Funcionalidad

La aplicación básica contiene 4 páginas:

- página principal, mostrada cuando se accede a la URL `https://hostname/index.php`,
- página "Acerca de (About)",
- la página "Contacto (Contact)", que muestra un formulario de contacto que permite a los usuarios finales contactarse vía email,
- y la página "Login", que muestra un formulario para loguearse que puede usarse para autenticar usuarios. Intenta loguearte con "admin/admin", y verás que el elemento "Login" del menú principal cambiará a "Logout".

Estas páginas comparten un encabezado y un pie. El encabezado contiene una barra con el menú principal que permite la navegación entre las diferentes páginas.

También deberías ver una barra en la parte inferior de la ventana del navegador. Esta es la útil herramienta de depuración provista por Yii para registrar y mostrar mucha información de depuración, tal como los mensajes de log, response status, las consultas ejecutadas a la base de datos, y más.

Adicionalmente a la aplicación web, hay un script de consola llamado `yii`, localizado en el directorio base de la aplicación. El script puede ser utilizado para ejecutar tareas de fondo y tareas de mantenimiento de la aplicación, las cuales son descritas en la Sección de Aplicación de Consola.

2.3.2. Estructura de la aplicación

Los archivos y directorios más importantes en tu aplicación son (asumiendo que la raíz de la aplicación es `basic`):

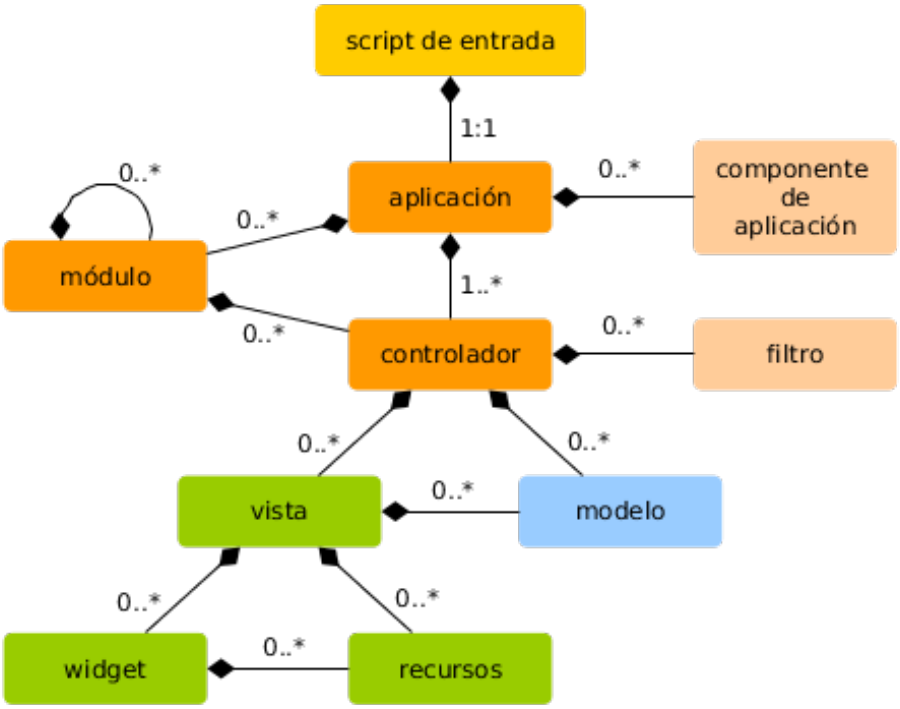
```
basic/                  base path de la aplicación
    composer.json       archivo utilizado por Composer, describe información
    de sus paquetes y librerías
    config/             contiene la configuración de las aplicaciones (y
    otras)
        console.php     configuración de la aplicación de consola
        web.php         configuración de la aplicación web
    commands/           contiene las clases de comandos de consola
    controllers/        contiene las clases de los controladores
    models/             contienes las clases del modelo
    runtime/            contiene archivos generados por Yii en tiempo de
    ejecución, como archivos de log y cache
    vendor/             contiene los paquetes y librerías instalados por
    Composer, incluyendo el propio núcleo de Yii
    views/              contiene los archivos de vistas (templates)
    web/                raíz web de la aplicación, contiene los archivos
    accesibles vía Web
        assets/         contiene los assets publicados (javascript y css)
        por Yii
        index.php       el script de entrada (o bootstrap) de la aplicación
    yii                 el script de ejecución de los comandos de consola de
    Yii
```

En general, los archivos de la aplicación pueden ser divididos en dos: aquellos bajo `basic/web` y aquellos bajo otros directorios. Los primeros pueden accederse directo por HTTP (ej., en un navegador), mientras que los últimos no pueden ni deben ser accedidos así.

Yii implementa el patrón de diseño modelo-vista-controlador (MVC)[23], que es reflejado en la estructura de directorios utilizada. El directorio `models` contiene todas las clases del modelo, el directorio `views` contiene todas las vistas (templates), y el directorio `controllers` contiene todas las clases de controladores.

El siguiente diagrama muestra la estructura estática de una aplicación.

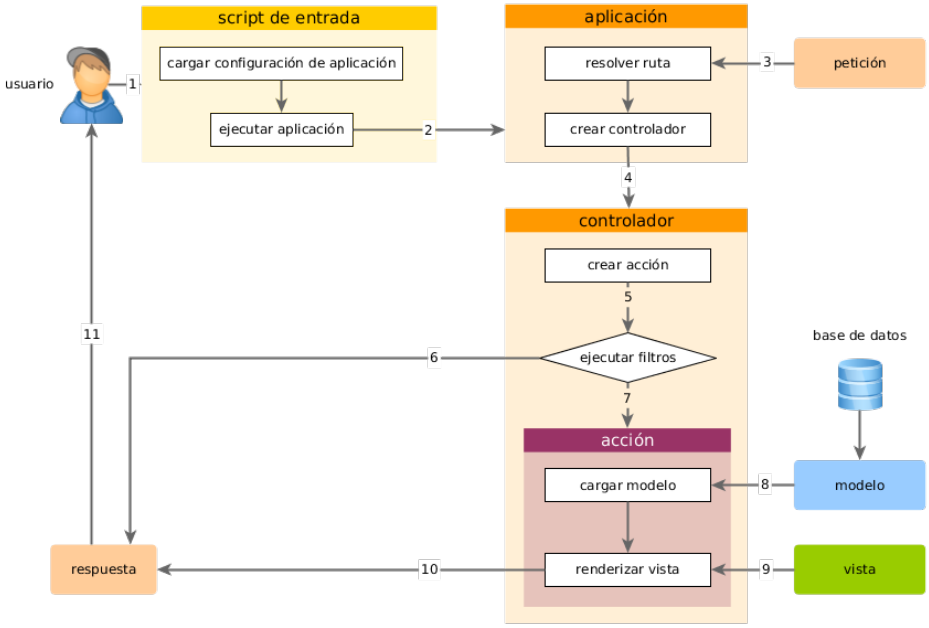
[23] https://wikipedia.org/wiki/Model-view-controller



Cada aplicación tiene un script de entrada `web/index.php` que es el único script PHP accesible vía web. El script de entrada toma una petición (request) entrante y crea una instancia de una aplicación para manejarlo. La aplicación resuelve la petición (request) con la ayuda de sus componentes, y la envía al resto de los elementos MVC. Los widgets son usados en las vistas para ayudar a construir elementos de interfaz complejos y dinámicos.

2.3.3. Ciclo de Vida de una Petición (Request)

El siguiente diagrama muestra cómo una aplicación maneja una petición.



1. Un usuario realiza una petición al script de entrada `web/index.php`.
2. El script de entrada carga la configuración de la aplicación y crea una instancia de la aplicación para manejar la consulta.
3. La aplicación resuelve la ruta solicitada con la ayuda del componente request de la aplicación.
4. La aplicación crea una instancia de un controlador para manejar la petición.
5. El controlador crea una instancia de una acción y ejecuta los filtros de dicha acción.
6. Si alguno de los filtros falla, la acción es cancelada.
7. Si todos los filtros pasan, la acción es ejecutada.
8. La acción carga datos del modelo, posiblemente de la base de datos.
9. La acción renderiza una vista, pasándole los datos del modelo cargado.
10. El resultado de la renderización es pasado al componente response de la aplicación.
11. El componente response envía el resultado de la renderización al navegador del usuario.

2.4. Diciendo Hola

Esta sección describe cómo crear la típica página "Hola Mundo" (Hello World en inglés) en tu aplicación. Para lograr este objetivo, vas a crear una acción y una vista:

- La aplicación enviará la petición de la página a la acción
- y la acción regresará el render de la vista que muestra la palabra "Hola" al usuario final.

A lo largo de este tutorial, aprenderás tres cosas:

1. Cómo crear una acción para responder peticiones (request),
2. Cómo crear una vista para armar el contenido de la respuesta, y
3. Cómo una aplicación envía peticiones a las acciones.

2.4.1. Creando una Acción

Para la tarea "Hola", crearás una acción `say` que lee un parámetro `message` de la petición y muestra este mensaje de vuelta al usuario. Si la petición no provee un parámetro `message`, la acción mostrará el mensaje por defecto "Hola".

> **Información:** Las acciones son objetos que los usuarios finales pueden utilizar directamente para su ejecución. Las acciones están agrupadas por controladores (controllers). El resultado de la ejecución de una acción es la respuesta que el usuario final recibirá.

Las acciones deben ser declaradas en controladores. Para simplificar, puedes declarar la acción `say` en el controlador `SiteController` existente. Este controlador está definido en el archivo de clase `controllers/SiteController.php`. Aquí está el inicio de la nueva acción:

```
<?php

namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    // ...código existente...

    public function actionSay($message = 'Hola')
    {
        return $this->render('say', ['message' => $message]);
    }
}
```

En el código de arriba, la acción `say` está definida por un método llamado `actionSay` en la clase `SiteController`. Yii utiliza el prefijo `action` para diferenciar los métodos de acciones de otros métodos en las clases de los controladores. El nombre que le sigue al prefijo `action` se mapea al ID de la acción.

Cuando se trata de nombrar las acciones, debes entender como Yii trata los ID de las acciones. Los ID de las acciones siempre son referenciados en minúscula. Si un ID de acción requiere múltiples palabras, estas serán concatenadas con guiones (ej., `crear-comentario`). Los nombres de los métodos de las acciones son mapeados a los ID de las acciones removiendo los guiones, colocando en mayúscula la primera letra de cada palabra, y colocando el prefijo `action` al resultado. Por ejemplo, el ID de la acción `crear-comentario` corresponde al nombre de método de acción `actionCrearComentario`.

El método de acción en nuestro ejemplo toma un parámetro `$message`, el cual tiene como valor por defecto `"Hola"` (de la misma manera que se coloca un valor por defecto a un argumento en cualquier función o método en PHP). Cuando una aplicación recibe una petición y determina que la acción `say` es responsable de manejar dicha petición, la aplicación llenará el parámetro con el parámetro que tenga el mismo nombre en la petición. En otras palabras, si la petición incluye un parámetro `message` con el valor de `"Adios"`, la variable `$message` dentro de la acción será sustituida por este valor.

Dentro del método de acción, `render()` es llamado para hacer render (mostrar o visualizar) un archivo vista (template) llamado `say`. El parámetro `message` tambien es pasado al view para que pueda ser utilizado ahí. El resultado es devuelto al método de la acción. Ese resultado será recibido por la aplicación y mostrado al usuario final en el navegador (como parte de una página HTML completa).

2.4.2. Creando una Vista

Las vistas son scripts que escribes para generar una respuesta de contenido. Para la tarea "Hola", vas a crear una vista `say` que imprime el parámetro `message` recibido desde el método action, y pasado por la acción a la vista:

```
<?php
use yii\helpers\Html;
?>
<?= Html::encode($message) ?>
```

La vista `say` debe ser guardada en el archivo `views/site/say.php`. Cuando el método `render()` es llamado en una acción, buscará un archivo PHP llamado `views/ControllerID/NombreVista.php`.

Nota que en el código de arriba, el parámetro `message` es procesado por `HTML-encoded` antes de ser impreso. Esto es necesario ya que el parámetro

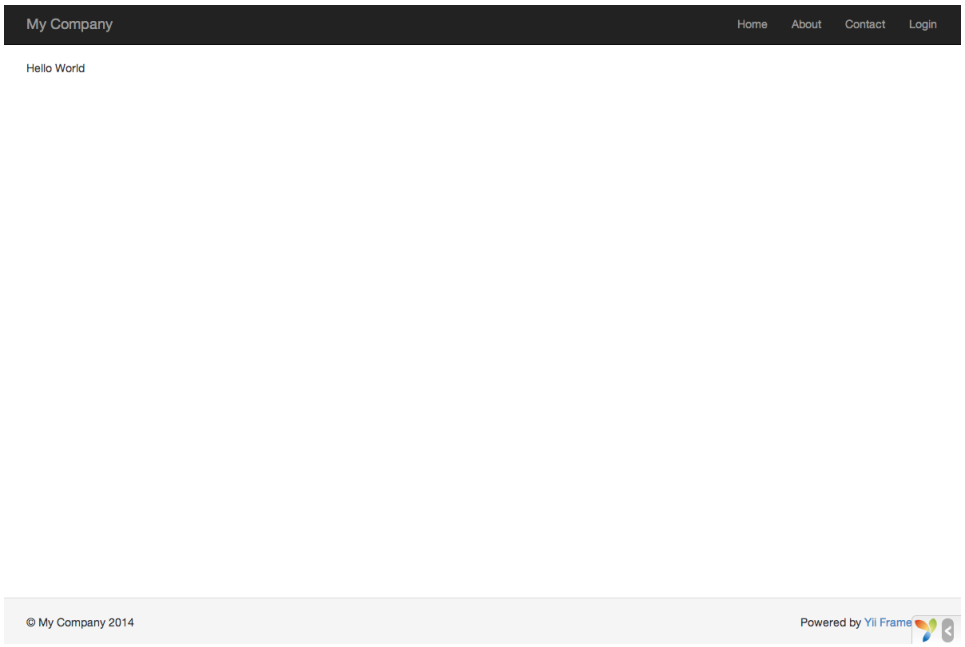
viene de un usuario final, haciéndolo vulnerable a ataques cross-site scripting (XSS)[24] pudiendo inyectar código de Javascript malicioso dentro del parámetro.

Naturalmente, puedes colocar mas contenido en la vista `say`. El contenido puede consistir de etiquetas HTML, texto plano, e inclusive código PHP. De hecho, la vista `say` es sólo un script PHP que es ejecutado por el método `render()`. El contenido impreso por el script de la vista será regresado a la aplicación como la respuesta del resultado. La aplicación a cambio mostrará el resultado al usuario final.

2.4.3. Probándolo

Después de crear la acción y la vista, puedes acceder a la nueva página abriendo el siguiente URL:

```
https://hostname/index.php?r=site%2Fsay&message=Hello+World
```



Esta URL resultará en una página mostrando "Hello World". La página comparte el mismo encabezado y pie de página de las otras páginas de la aplicación.

Si omites el parámetro `message` en el URL, verás que la página muestra sólo "Hola". Esto es porque `message` es pasado como un parámetro al método `actionSay()`, y cuando es omitido, el valor por defecto `"Hola"` será utilizado.

> **Información:** La nueva página comparte el mismo encabezado y pie de página que otras páginas porque el método `render()` au-

[24] https://es.wikipedia.org/wiki/Cross-site_scripting

> tomáticamente inyectará el resultado de la vista `say` en el layout, que en este caso está localizada en `views/layouts/main.php`.

El parámetro `r` en el URL de arriba requiere más explicación. Se refierea a route (ruta), y es el ID amplio y único de una aplicación que refiere a una acción. El formato de las rutas es `ControllerID/ActionID`. Cuando la aplicación recibe una petición, revisará este parámetro, utilizando la parte del `ControllerID` para determinar cual clase de controlador será inicializado para manejar la petición. Entonces, el controlador utilizará la parte `ActionID` para determinar cual acción debe ser inizializada para hacer realmente el trabajo. En este ejemplo, la ruta `site/say` será respondida por la clase controlador `SiteController` y la acción `say`. Como resultado, el método `SiteController::actionSay()` será llamado para manejar el requerimiento.

> **Información:** Al igual que las acciones, los controladores tambien tienen ID únicos que los identifican en una aplicación. Los ID de los Controladores utilizan las mismas reglas de nombrado que los ID de las acciones. Los nombres de las clases de los controladores son derivados de los ID de los controladores removiendo los guiones de los ID, colocando la primera letra en mayúscula en cada palabra, y colocando el sufijo `Controller` al resultado. Por ejemplo, el ID del controlador `post-comentario` corresponde al nombre de clase del controlador `PostComentarioController`.

2.4.4. Resumen

En esta sección, has tocado las partes del controlador y la vista del patrón de diseño MVC. Has creado una acción como parte de un controlador para manejar una petición específica. Y también has creado una vista para armar el contenido de la respuesta. En este simple ejemplo, ningún modelo ha sido involucrado ya que el único dato que fue utilizado fue el parámetro `message`.

También has aprendido acerca de las rutas en Yii, que actúan como puentes entre la petición del usuario y las acciones del controlador.

En la próxima sección, aprenderás como crear un modelo, y agregar una nueva página que contenga un formulario HTML.

2.5. Trabajando con Formularios

En esta sección, describiremos como crear una nueva página para solicitar información de los usuarios. La página mostrará un formulario con un campo de input para el nombre y un campo de input para el email. Después de recibir estos datos del usuario, la página le mostrará la información de vuelta al usuario para la confirmación.

Para lograr este objetivo, además de crear una acción y dos vistas, también crearás un modelo.

A través de este tutorial, aprenderás

- Cómo crear un modelo para representar los datos ingresados por un usuario;
- Cómo declarar reglas para validar los datos ingresado por los usuarios;
- Cómo construir un formulario HTML en una vista.

2.5.1. Creando un Modelo

Para representar los datos ingresados por un usuario, crea una clase modelo `EntryForm` cómo se muestra abajo y guarda la clase en el archivo `models/EntryForm.php`. Por favor, visita la sección Autocargando Clases para obtener más detalles acerca de la convención de nombres de los archivos de clase.

```
<?php

namespace app\models;

use yii\base\Model;

class EntryForm extends Model
{
    public $name;
    public $email;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['email', 'email'],
        ];
    }
}
```

La clase se extiende a partir de `yii\base\Model`, que es una clase base que provee Yii y es comúnmente utilizada para representar datos de formularios.

La clase contiene dos miembros públicos, `name` y `email`, que son utilizas para mantener los datos ingresados por el usuario. También contiene el método llamado `rules()` que regresa un conjunto de reglas utilizadas para validar los datos. Las reglas de validación declaradas arriba indican que

- ambos datos, tanto el `name` como el `email`, son requeridos;
- el dato `email` debe ser una dirección de correo válida.

Si tienes un objeto `EntryForm` llenado con los datos ingresados por el usuario, puedes llamar su `validate()` para disparar (trigger) la validación de los datos. Un fallo en la validación de los datos se mostrará en la propiedad `hasErrors`, y a través de `errors` puedes aprender cuales son los errores de validación que tiene el modelo.

2.5.2. Creando una Acción

Luego, crea una acción `entry` en el controlador `site`, como lo hiciste en la sección anterior.

```
<?php

namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\EntryForm;

class SiteController extends Controller
{
    // ...código existente...

    public function actionEntry()
    {
        $model = new EntryForm;

        if ($model->load(Yii::$app->request->post()) && $model->validate())
        {
            // validar los datos recibidos en el modelo

            // aquí haz algo significativo con el modelo ...

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // la página es mostrada inicialmente o hay algún error de
            validación
            return $this->render('entry', ['model' => $model]);
        }
    }
}
```

La acción primero crea un objeto `EntryForm`. Luego intenta poblar el modelo con los datos del `$_POST` que es proporcionado por Yii a través de `yii\web\Request::post()`. Si el modelo es llenado satisfactoriamente (ej., el usuario ha enviado el formulario HTML), llamará a `validate()` para asegurarse que los datos ingresados son válidos.

Si todo está bien, la acción mostrará una vista llamada `entry-confirm` para confirmar con el usuario que acepta los datos que ha ingresado. De otra manera, la vista `entry` será mostrada, y mostrará el formulario HTML junto con los mensajes de error de validación (si es que hay alguno).

> **Información:** La expresión `Yii::$app` representa la instancia de la aplicación que es un singleton globalmente accesible. También es un service locator (localizador de servicio) que provee los componentes, tales como `request`, `response`, `db`, etc. para soportar

funcionalidades específicas. En el código de arriba, el componente `request` es utilizado para acceder los datos `$_POST`.

2.5.3. Creando Vistas

Finalmente, crea dos vistas llamadas `entry-confirm` y `entry` que sean mostradas por la acción `entry`, tal y como fue descrito en la última sub-sección.

La vista `entry-confirm` simplemente muestra los datos de name y email. Ésta debe ser guardada como el archivo `views/site/entry-confirm.php`.

```
<?php
use yii\helpers\Html;
?>
<p>You have entered the following information:</p>

<ul>
    <li><label>Name</label>: <?= Html::encode($model->name) ?></li>
    <li><label>Email</label>: <?= Html::encode($model->email) ?></li>
</ul>
```

La vista `entry` muestra un formulario HTML. Debe ser guardado como el archivo `views/site/entry.php`.

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
?>
<?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'name') ?>

    <?= $form->field($model, 'email') ?>

    <div class="form-group">
        <?= Html::submitButton('Submit', ['class' => 'btn btn-primary']) ?>
    </div>

<?php ActiveForm::end(); ?>
```

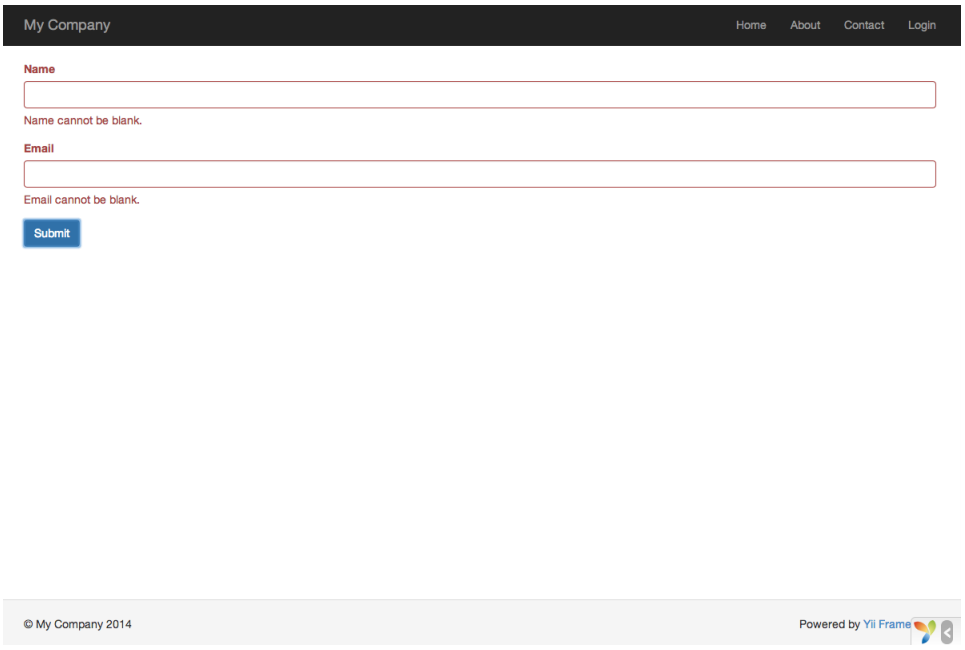
La vista utiliza un poderoso widget llamado `ActiveForm` para construir el formulario HTML. Los métodos `begin()` y `end()` del widget muestran, respectivamente, las etiquetas de apertura y cierre del formulario. Entre las llamadas de los dos métodos, los campos de input son creados por el método `field()`. El primer campo input es del dato "name", y el segundo del dato "email". Después de los campos de input, el método `yii\helpers\Html::submitButton()` es llamado para general el botón de submit (enviar).

2.5.4. Probándolo

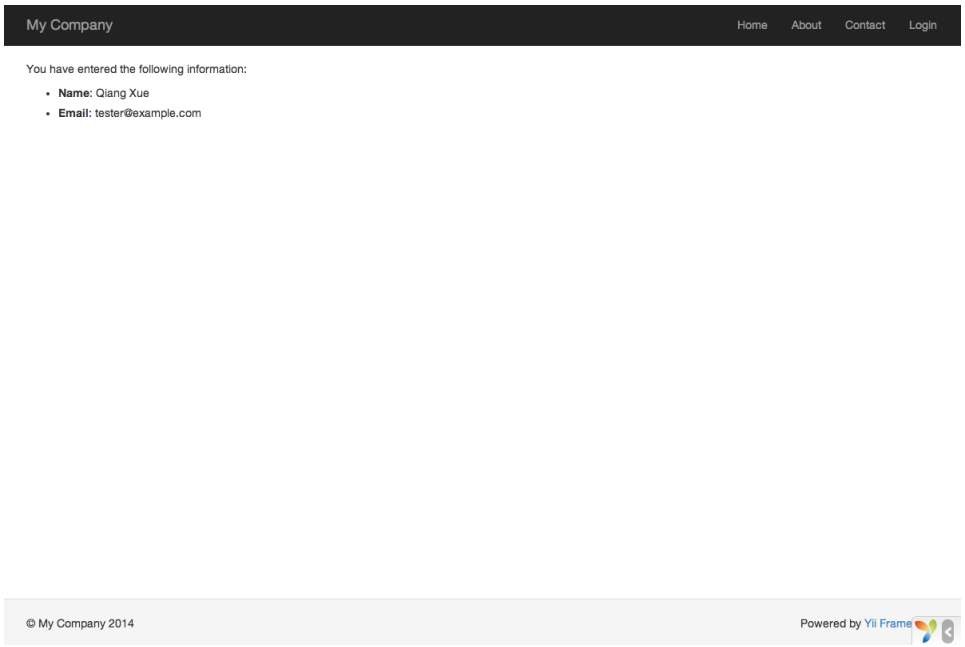
Para ver cómo funciona, utiliza tu navegador para ir al siguiente URL:

```
https://hostname/index.php?r=site/entry
```

Verás una página que muestra un formulario con dos campos de input. Adelante de cada campo de input, será mostrada también una etiqueta indicando que dato necesitas ingresar. Si haces click en el botón de envío (Submit) sin ingresar nada, o si ingresas una dirección de correo inválida, verás un mensaje de error que se mostrará al lado del campo que tiene problemas.



Después de ingresar un nombre y dirección de correo válidos y haciendo click en el botón de envío (Submit), verás una nueva página mostrando los datos que acabas de ingresar.



Magia Explicada

Te estarás preguntando cómo funciona toda esa automatización del formulario HTML, porque parece casi mágico que pueda mostrar una etiqueta para cada campo de input y mostrar los mensajes de error si no ingresas los datos correctamente sin recargar la página.

Si, la validación de los datos se realiza en el lado del cliente utilizando JavaScript así como también en el lado del servidor. `yii\widgets\ActiveForm` es lo suficientemente inteligente como para extraer las reglas de validación que has declarado en `EntryForm`, convertirlas en código Javascript, y utilizar el JavaScript para realizar la validación de los datos. En caso de que hayas deshabilitado JavaScript en tu navegador, la validación se realizará igualmente en el lado del servidor, como se muestra en el método `actionEntry()`. Esto garantiza la validez de los datos en cualquier circunstancias.

Las etiquetas de los campos de input son generados por el método `field()` basado en los nombres de las propiedades del modelo. Por ejemplo, la etiqueta `Name` será generada de la propiedad `name`. Puedes personalizar una etiqueta con el siguiente código:

```
<?= $form->field($model, 'name')->label('Tu Nombre') ?>
<?= $form->field($model, 'email')->label('Tu Email') ?>
```

> **Información:** Yii provee muchos widgets para ayudarte a construir rápidamente vistas complejas y dinámicas. Como aprenderás más adelante, escribir un nuevo widget es extremadamente

fácil. Puedes convertir mucho del código de tus vistas en widgets reutilizables para simplificar el desarrollo de las vistas en un futuro.

2.5.5. Resumen

En esta sección, has tocado cada parte del patrón de diseño MVC. Ahora has aprendido a crear una clase modelo para representar los datos del usuario y validarlos.

También has aprendido como obtener datos de los usuarios y como mostrarlos de vuelta. Esta es una tarea que puede tomarte mucho tiempo cuando estás desarrollando una aplicación. Yii provee poderosos widgets para hacer muy fácil esta tarea.

En la próxima sección, aprenderás como trabajar con bases de datos que son necesarias en casi cualquier aplicación.

2.6. Trabajar con Bases de Datos

En esta sección, explicaremos cómo crear una nueva página para mostrar datos de países traídos de una tabla de la base de datos llamada `country`. Para lograr este objetivo, configurarás una conexión a la base de datos, crearás una clase Active Record, una acción y una vista.

A lo largo de este tutorial, aprenderás a

- configurar una conexión a la base de datos;
- definir una clase Active Record;
- realizar consultas a la base de datos utilizando la clase Active Record;
- mostrar datos en una vista con paginación incluida.

Ten en cuenta que para finalizar esta sección, deberás tener al menos conocimientos básicos y experiencia con bases de datos. En particular, deberás ser capaz de crear una base de datos y saber ejecutar consultas SQL usando alguna herramienta de cliente de base de datos.

2.6.1. Preparar una Base de Datos

Para empezar, crea una base de datos llamada `yii2basic` de la cual tomarás los datos en la aplicación. Puedes elegir entre una base de datos SQLite, MySQL, PostgreSQL, MSSQL u Oracle, dado que Yii incluye soporte para varios motores. Por simplicidad, usaremos MySQL en la siguiente descripción.

A continuación, crea una tabla llamada `country` e inserta algunos datos de ejemplo. Puedes utilizar las siguientes declaraciones SQL.

```
CREATE TABLE `country` (
  `code` CHAR(2) NOT NULL PRIMARY KEY,
  `name` CHAR(52) NOT NULL,
```

```
  `population` INT(11) NOT NULL DEFAULT '0'
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `country` VALUES ('AU','Australia',24016400);
INSERT INTO `country` VALUES ('BR','Brazil',205722000);
INSERT INTO `country` VALUES ('CA','Canada',35985751);
INSERT INTO `country` VALUES ('CN','China',1375210000);
INSERT INTO `country` VALUES ('DE','Germany',81459000);
INSERT INTO `country` VALUES ('FR','France',64513242);
INSERT INTO `country` VALUES ('GB','United Kingdom',65097000);
INSERT INTO `country` VALUES ('IN','India',1285400000);
INSERT INTO `country` VALUES ('RU','Russia',146519759);
INSERT INTO `country` VALUES ('US','United States',322976000);
```

Al final, tendrás una base de datos llamada `yii2basic`, y dentro de esta, una tabla llamada `country` con diez registros en ella.

2.6.2. Configurar una conexión a la Base de Datos

Asegúrate de tener instalado la extensión de PHP PDO[25] y el driver de PDO para el motor que estés utilizando (ej. `pdo_mysql` para MySQL). Este es un requisito básico si tu aplicación va a utilizar bases de datos relacionales.

Abre el archivo `config/db.php` y ajusta el contenido dependiendo de la configuración a tu base de datos. Por defecto, el archivo contiene el siguiente contenido:

```
<?php

return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

El archivo `config/db.php` representa la típica configuración basada en archivos. Este archivo de configuración en particular especifica los parámetros necesarios para crear e inicializar una instancia de `yii\db\Connection` a través de la cual puedes realizar consultas SQL contra la base de datos subyacente.

La conexión a la base de datos realizada anteriormente puede ser accedida mediante `Yii::$app->db`.

> **Información:** El archivo `config/db.php` será incluido en el archivo principal de configuración `config/web.php`, el cual especifica cómo la instancia de la aplicación debe ser inicializada. Para más información, consulta la sección Configuraciones.

[25] https://www.php.net/manual/es/book.pdo.php

Si necesitas trabajar con bases de datos cuyo soporte no está incluído en Yii, revisa las siguientes extensiones:

- Informix[26]
- IBM DB2[27]
- Firebird[28]

2.6.3. Crear un Active Record

Para representar y extraer datos de la tabla `country`, crea una clase Active Record llamada `Country` y guárdala en el archivo `models/Country.php`.

```
<?php

namespace app\models;

use yii\db\ActiveRecord;

class Country extends ActiveRecord
{
}
```

La clase `Country` extiende de `yii\db\ActiveRecord`. No necesitas escribir ningún código dentro de ella! Con tan sólo el código de arriba, Yii adivinará la tabla correspondiente a la clase desde su nombre.

> **Información:** Si no se puede realizar un emparejamiento entre el nombre de la clase y la tabla, puedes sobrescribir el método `yii\db\ActiveRecord::tableName()` para especificar explícitamente el nombre de la tabla asiciada.

Utilizando la clase `Country`, puedes manipular los datos de la tabla `country` fácilmente, como se muestra en los siguiente ejemplos:

```
use app\models\Country;

// obtiene todos los registros de la tabla country ordenándolos por "name"
$countries = Country::find()->orderBy('name')->all();

// obtiene el registro cuya clave primaria es "US"
$country = Country::findOne('US');

// muestra "United States"
echo $country->name;

// cambia el nombre del país a "U.S.A." y lo guarda en la base de datos
$country->name = 'U.S.A.';
$country->save();
```

[26] https://github.com/edgardmessias/yii2-informix
[27] https://github.com/edgardmessias/yii2-ibm-db2
[28] https://github.com/edgardmessias/yii2-firebird

> **Información:** Active Record es una potente forma de acceder y manipular datos de una base de datos de una manera orientada a objetos. Puedes encontrar información más detallada acerca de Active Record. Además de Active Record, puedes utilizar un método de acceso de bajo nivel llamado Data Access Objects.

2.6.4. Crear una Acción

Para mostrar el país a los usuarios, necesitas crear una acción. En vez de hacerlo en el controlador `site` como lo hiciste en las secciones previas, tiene más sentido crear un nuevo controlador que englobe todas las acciones de manipulación de datos de la tabla country. Llama a este nuevo controlador `CountryController` y define una acción `index` en él, como se muestra a continuación:

```
<?php

namespace app\controllers;

use yii\web\Controller;
use yii\data\Pagination;
use app\models\Country;

class CountryController extends Controller
{
    public function actionIndex()
    {
        $query = Country::find();

        $pagination = new Pagination([
            'defaultPageSize' => 5,
            'totalCount' => $query->count(),
        ]);

        $countries = $query->orderBy('name')
            ->offset($pagination->offset)
            ->limit($pagination->limit)
            ->all();

        return $this->render('index', [
            'countries' => $countries,
            'pagination' => $pagination,
        ]);
    }
}
```

Guarda el código anterior en el archivo `controllers/CountryController.php`.

La acción `index` llama a `Country::find()` para generar una consulta a la base de datos y traer todos los datos de la tabla `country`. Para limitar la cantidad de registros traídos en cada petición, la consulta es paginada con la

ayuda de un objeto `yii\data\Pagination`. El objeto `Pagination` sirve para dos propósitos:

- Define las cláusulas `offset` y `limit` de la consulta SQL para así sólo devolver una sola página de datos (5 registros por página como máximo).
- Es utilizado en la vista para mostrar un paginador que consiste en una lista de botones que representan a cada página, tal como será explicado en la siguiente sub-sección.

Al final, la acción `index` renderiza una vista llamada `index` y le pasa los datos de países así como la información de paginación relacionada.

2.6.5. Crear una Vista

Bajo el directorio `views`, crea primero un sub-directorio llamado `country`. Este será usado para contener todas las vistas renderizadas por el controlador `country`. Dentro del directorio `views/country`, crea un archivo llamado `index.php` con el siguiente contenido:

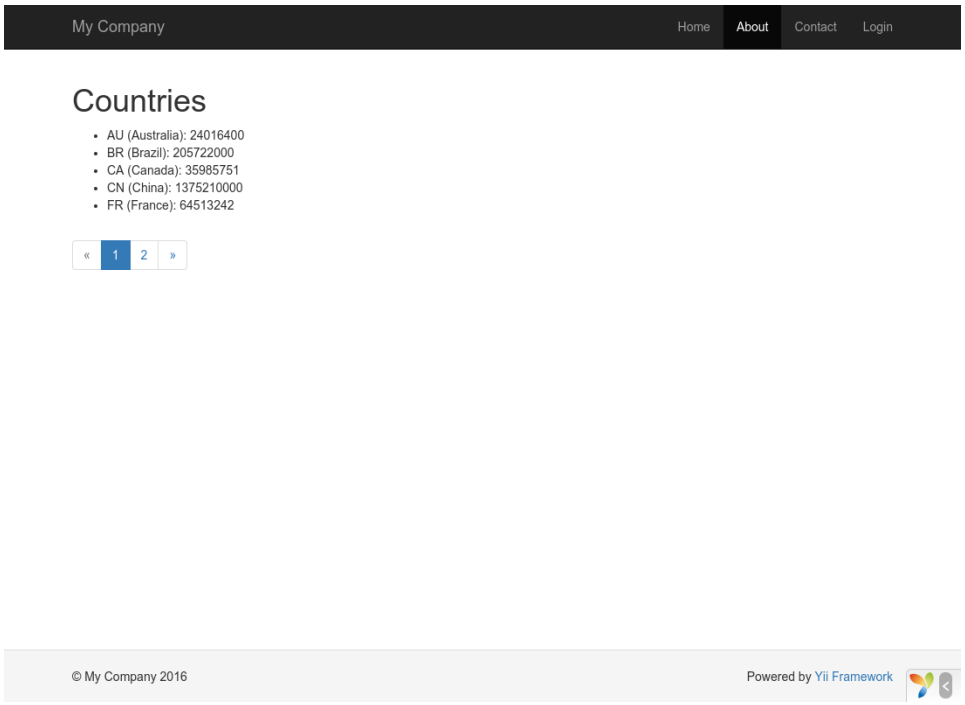
```
<?php
use yii\helpers\Html;
use yii\widgets\LinkPager;
?>
<h1>Países</h1>
<ul>
<?php foreach ($countries as $country): ?>
    <li>
        <?= Html::encode("{$country->name} ({$country->code})") ?>:
        <?= $country->population ?>
    </li>
<?php endforeach; ?>
</ul>

<?= LinkPager::widget(['pagination' => $pagination]) ?>
```

La vista consiste en dos partes. En la primera, los datos de países son recorridos y renderizados como una lista HTML. En la segunda parte, un widget `yii\widgets\LinkPager` es renderizado usando la información de paginación pasada desde la acción. El widget `LinkPager` muestra una lista de botones que representan las páginas disponibles. Haciendo click en cualquiera de ellas mostrará los datos de países de la página correspondiente.

2.6.6. Probándolo

Para ver cómo funciona, utiliza a la siguiente URL en tu navegador:

```
https://hostname/index.php?r=country%2Findex
```



Verás una página que muestra cinco países. Y debajo de todos los países, verás un paginador con cuatro botones. Si haces click en el botón "2", verás que la página muestra otros cinco países de la base de datos. Observa más cuidadosamente y verás que la URL en el navegador cambia a

```
https://hostname/index.php?r=country%2Findex&page=2
```

Entre bastidores, `Pagination` está realizando su magia.

- Inicialmente, `Pagination` representa la primera página, que agrega a la consulta SQL a la base de datos con la cláusula `LIMIT 5 OFFSET 0`. Como resultado, los primeros cinco países serán traídos y mostrados.
- El widget `LinkPager` renderiza los botones de páginas usando las URLs creadas por `Pagination`. Las URLs contendrán el parámetro `page` representando los números de páginas.
- Si haces click en el botón "2", se lanza y maneja una nueva petición a la ruta `country/index`. `Pagination` lee el parámetro `page` y define el número de página actual como "2". Por consiguiente, la consulta a la base de datos tendrá la cláusula `LIMIT 5 OFFSET 5` y devolverá los siguientes cinco países para mostrar.

2.6.7. Resumen

En esta sección has aprendido cómo trabajar con una base de datos. También has aprendido cómo traer y mostrar datos paginados con la ayuda de `yii\data\Pagination` y `yii\widgets\LinkPager`.

En la siguiente sección, aprenderás a utilizar la poderosa herramienta de generación de código llamada Gii, para ayudarte a implementar rápidamente algunas características comunes, como crear operaciones de Alta-Baja-Modificación (ABM, o CRUD en inglés) de los datos guardados en la base de datos. De hecho, el código que acabas de escribir fue generado automáticamente a través de esta herramienta.

2.7. Generando Código con Gii

En esta sección, explicaremos cómo utilizar Gii para generar código que automáticamente implementa algunas de las características más comunes de una aplicación. Para lograrlo, todo lo que tienes que hacer es ingresar la información de acuerdo a las instrucciones mostradas en la páginas web de Gii.

A lo largo de este tutorial, aprenderás

- Cómo activar Gii en tu aplicación;
- Cómo utilizar Gii para generar una clase Active Record;
- Cómo utilizar Gii para generar el código que implementa las operaciones ABM de una tabla de la base de datos.
- Cómo personalizar el código generado por Gii.

2.7.1. Comenzando con Gii

Gii está provisto por Yii en forma de módulo. Puedes habilitar Gii configurándolo en la propiedad `modules` de la aplicación. Dependiendo de cómo hayas creado tu aplicación, podrás encontrar que el siguiente código ha sido ya incluido en el archivo de configuración `config/web.php`:

```
$config = [ ... ];

if (YII_ENV_DEV) {
    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = [
        'class' => 'yii\gii\Module',
    ];
}
```

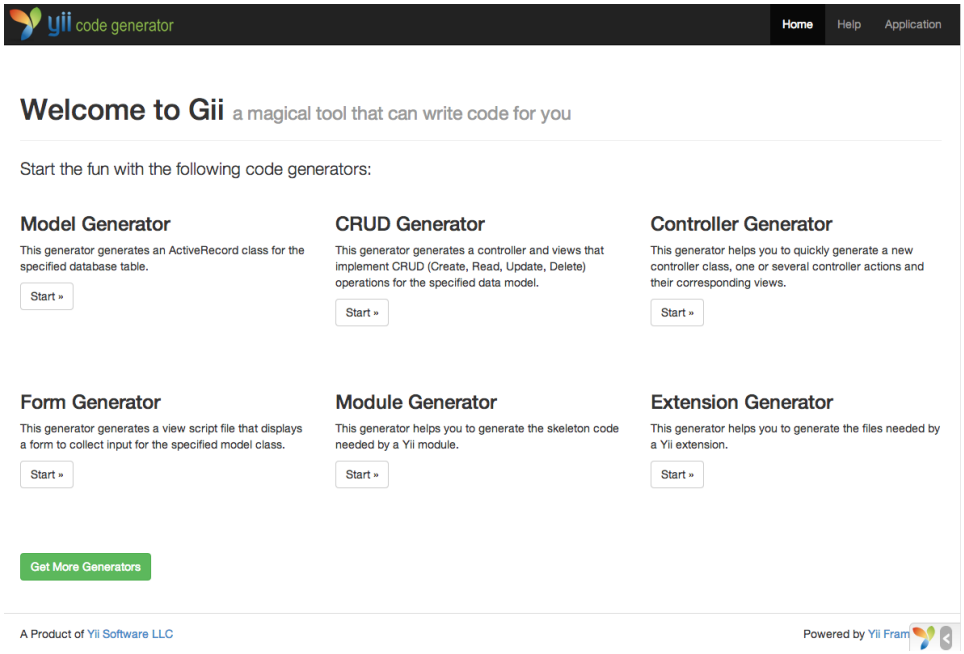
La configuración dice que al estar en el entorno de desarrollo, la aplicación debe incluir el módulo llamado `gii`, cuya clase es `yii\gii\Module`.

Si chequeas el script de entrada `web/index.php` de tu aplicación, encontrarás la línea que esencialmente define la constante `YII_ENV_DEV` como verdadera `-true`.

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

De esta manera, tu aplicación habrá habilitado Gii, y puedes acceder al módulo a través de la siguiente URL:

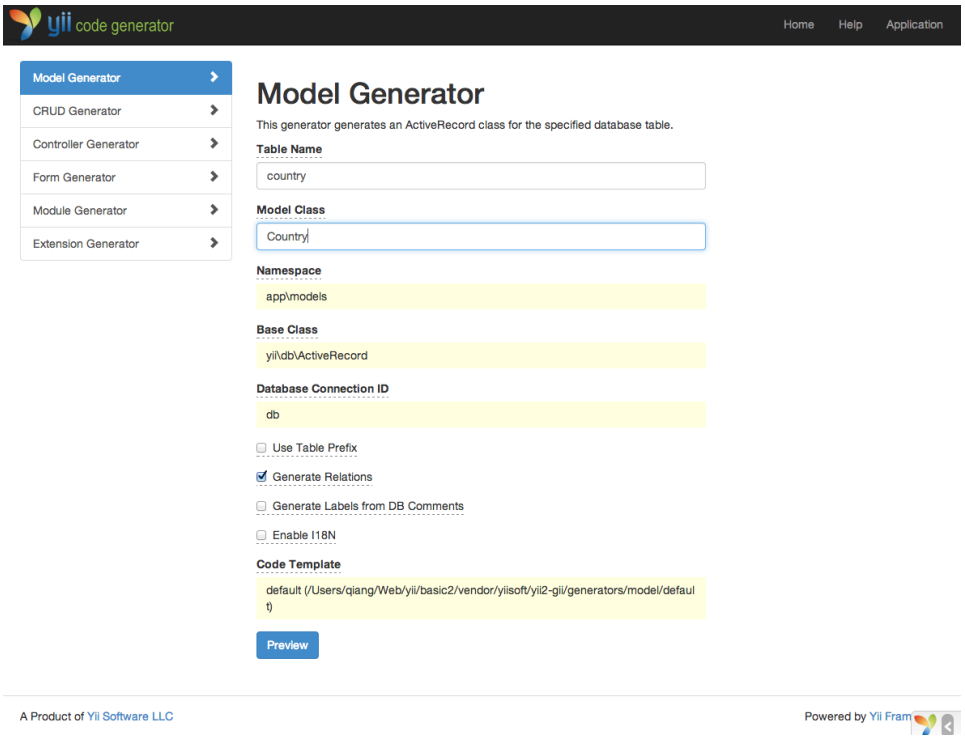
```
https://hostname/index.php?r=gii
```



2.7.2. Generando una Clase Active Record

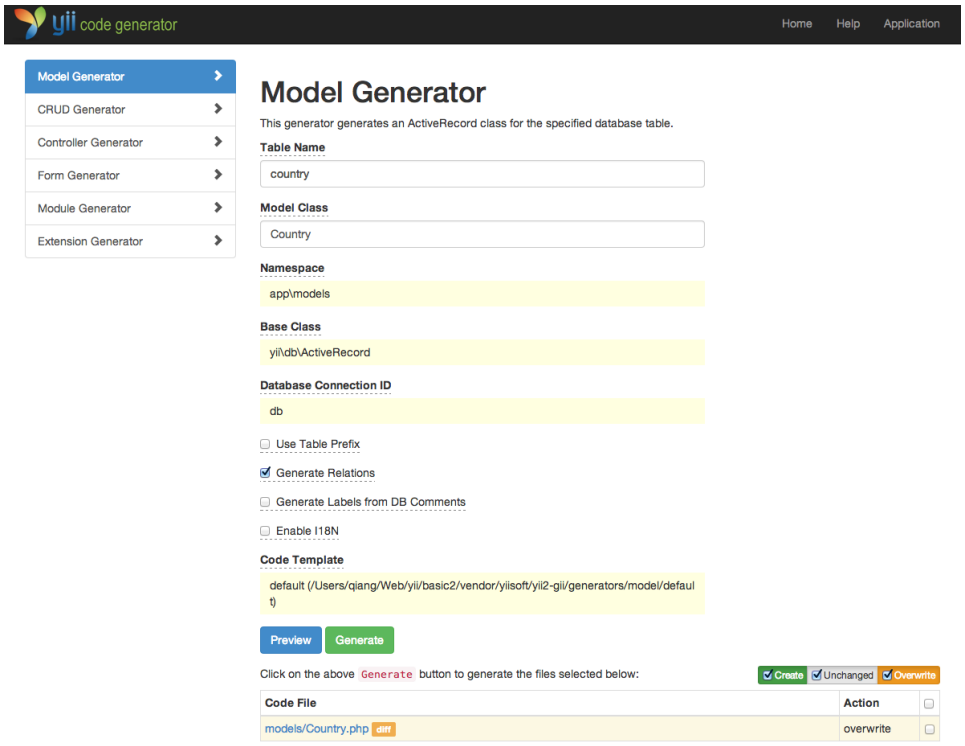
Para poder generar una clase Active Record con Gii, selecciona "Model Generator" (haciendo click en el vínculo que existe en la página inicial del modulo Gii). Después, completa el formulario de la siguiente manera,

- Table Name: `country`
- Model Class: `Country`



Haz click el el botón "Preview". Verás que `models/Country.php` está mostrado listado como la clase resultante que ha de ser creada. Puedes hacer click en el nombre de la clase para previsualizar su contenido.

Al utilizar Gii, si habías creado previamente el mismo archivo y puede ser sobrescrito, si haces click en el botón `diff` cercano al nombre del archivo, verás las diferencias entre el código a ser generado y la versión existente del mismo.



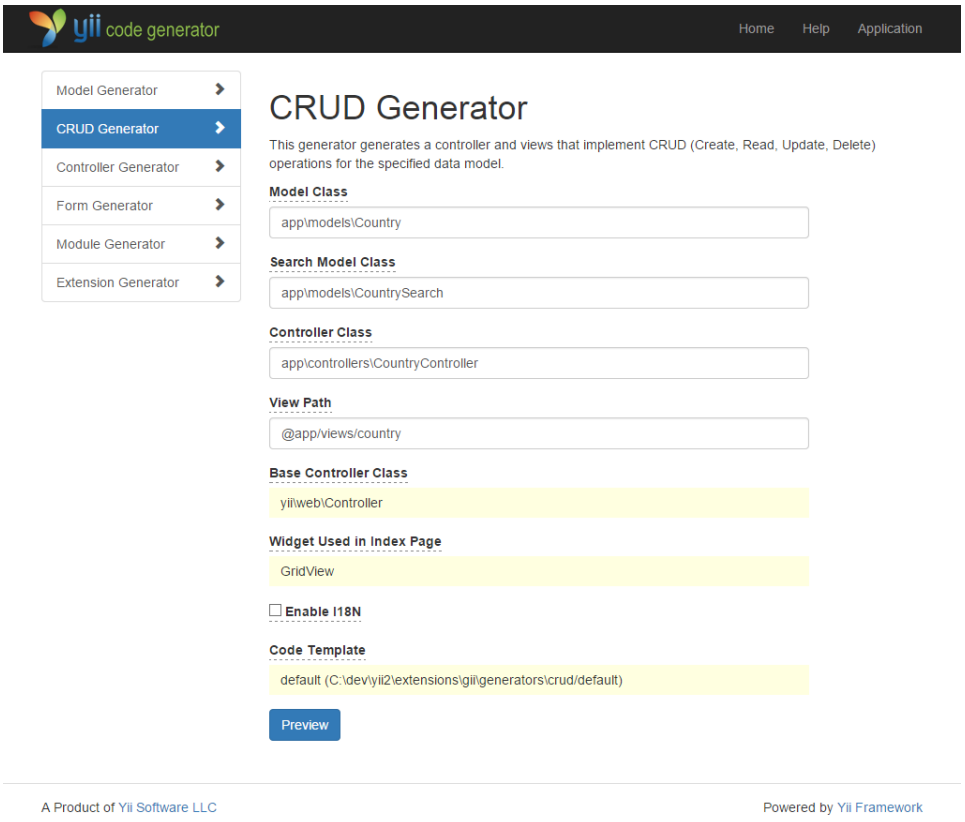
Para sobrescribir un archivo existente, marca el checkbox que se encuentra al lado de "overwrite" y posteriormente haz click en el botón "Generate".

Después, verás una página de confirmación indicando que el código ha sido generado correctamente y tu archivo `models/Country.php` ha sido sobrescrito con el nuevo código generado.

2.7.3. Generando código de ABM (CRUD en inglés)

En computación, CRUD es el acrónimo de Crear, Obtener, Actualizar y Borrar (del inglés: Create, Read, Update y Delete) representando la cuatro funciones con datos más comunes en la mayoría de sitios Web. El acrónimo ABM es Altas, Bajas y Modificaciones. Para generar un ABM, selecciona "CRUD Generator" y completa el formulario de esta manera:

- Model Class: `app\models\Country`
- Search Model Class: `app\models\CountrySearch`
- Controller Class: `app\controllers\CountryController`



Al hacer click en el botón "Preview" verás la lista de archivos a ser generados.

Si has creado previamente los archivos `controllers/CountryController.php` y `views/country/index.php` (en la sección sobre bases de datos de esta guía), asegúrate de seleccionar el checkbox "overwrite" para reemplazarlos. (Las versiones anteriores no disponían de un soporte ABM (CRUD) completo.)

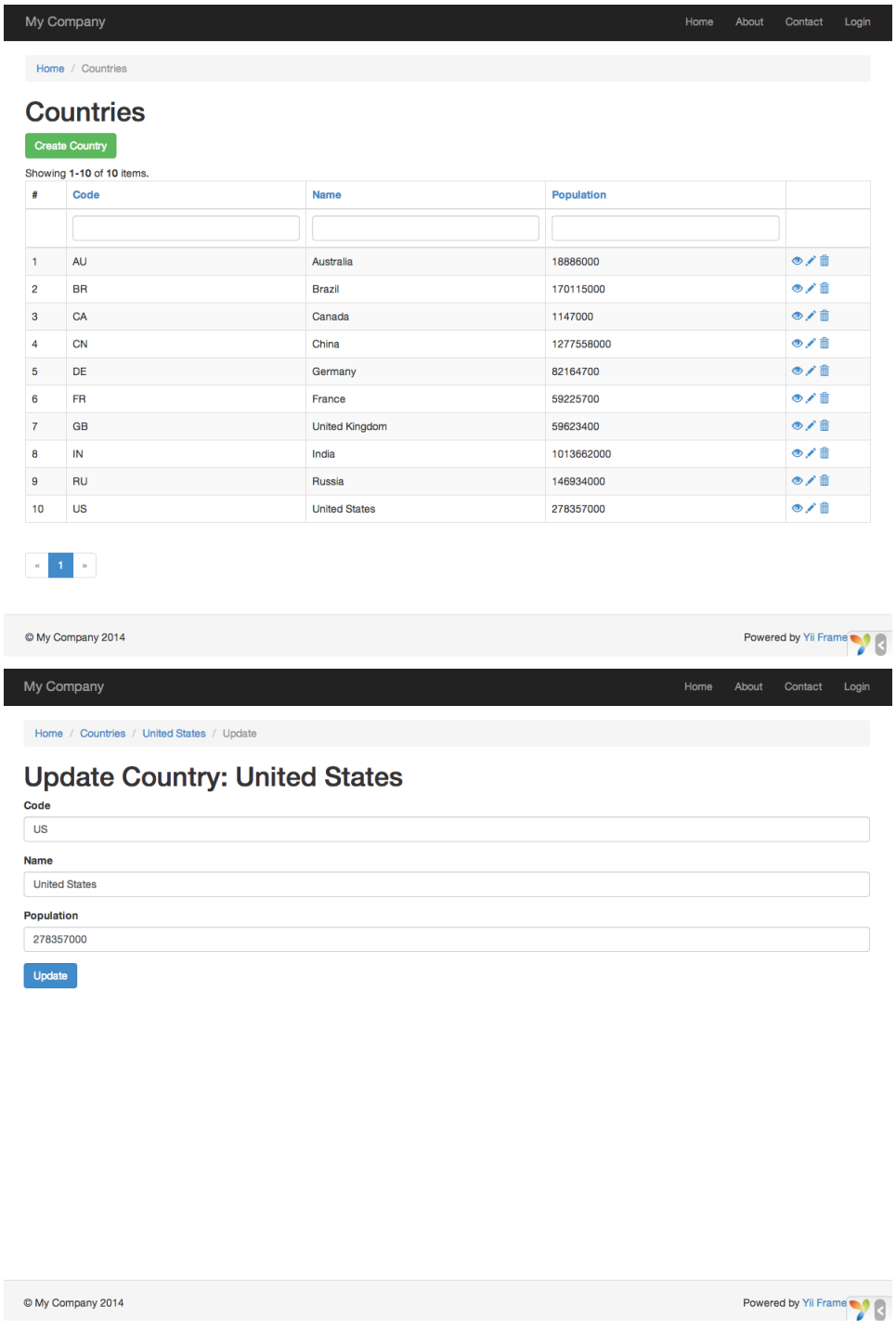
2.7.4. Probándolo

Para ver cómo funciona, accede desde tu navegador a la siguiente URL:

```
https://hostname/index.php?r=country/index
```

Verás una grilla de datos mostrando los países de la base de datos. Puedes ordenar la grilla o filtrar los resultados escribiendo alguna condición en los encabezados de las columnas.

Por cada país mostrado en la grilla, puedes elegir entre visualizar el registro, actualizarlo o eliminarlo. Puedes incluso hacer click en el botón "Create Country" que se encuentra sobre la grilla y así cargar un nuevo país en la base de datos.



La siguiente es la lista de archivos generados por Gii, en el caso de que quieras inspeccionar cómo el ABM ha sido generado, o por si desearas personalizarlos:

- Controlador: `controllers/CountryController.php`

- Modelos: `models/Country.php` y `models/CountrySearch.php`
- Vistas: `views/country/*.php`

> **Información:** Gii está diseñado para ser una herramienta altamente configurable. Utilizándola con sabiduría puede acelerar enormemente la velocidad de desarrollo de tu aplicación. Para más detalles, consulta la sección Gii.

2.7.5. Resumen

En esta sección, has aprendido a utilizar Gii para generar el código que implementa completamente las características de un ABM de acuerdo a una determinada tabla de la base de datos.

2.8. Mirando Hacia Adelante

Si has leído el capítulo "Comenzando con Yii" completo, has creado una aplicación completa en Yii. En el proceso, has aprendido cómo implementar algunas características comúnmente necesitadas, tales como obtener datos del usuario a través de formularios HTML, traer datos desde la base de datos, y mostrar datos utilizando paginación. También has aprendido a utilizar Gii[29] para generar código automáticamente. Utilizar Gii para la generación de código transforma la carga en el proceso de tu desarrollo Web en una tarea tan simple como solamente completar unos formularios.

Esta sección resumirá los recursos disponibles de Yii que te ayudarán a ser más productivo al utilizar el framework.

- Documentación
  - La Guía Definitiva[30]: Como su nombre lo indica, la guía define precisamente cómo debería trabajar Yii y provee guías generales acerca de su utilización. Es el tutorial más importante de Yii, y el que deberías leer antes de escribir cualquier código en Yii.
  - La Referencia de Clases[31]: Esta especifica el uso de cada clase provista por Yii. Debería ser utilizada principalmente cuando estás escribiendo código y deseas entender el uso de una clase, método o propiedad en particular. El uso de la referencia de clases es mejor luego de un entendimiento contextual del framework.
  - Los Artículos de la Wiki[32]: Los artículos de la wiki son escritos por usuarios de Yii basados en sus propias experiencias. La mayoría de ellos están escritos como recetas de cocina, y muestran cómo resolver problemas particulares utilizando Yii. Si bien la calidad

[29] https://github.com/yiisoft/yii2-gii/blob/master/docs/guide/README.md
[30] https://www.yiiframework.com/doc-2.0/guide-README.html
[31] https://www.yiiframework.com/doc-2.0/index.html
[32] https://www.yiiframework.com/wiki/?tag=yii2

de estos puede no ser tan buena como la de la Guía Definitiva, son útiles ya que cubren un espectro muy amplio de temas y puede proveer a menudo soluciones listas para usar.
    - Libros[33]
- Extensiones[34]: Yii puede hacer alarde de una librería de miles de extensiones contribuidas por usuarios, que pueden fácilmente conectadas a tu aplicación, haciendo que el desarrollo de la misma sea todavía más fácil y rápido.
- Comunidad
    - Foro: https://forum.yiiframework.com/
    - Chat IRC: El canal #yii en la red Libera (ircs://irc.libera.chat:6697/yii)
    - Chat Gitter: https://gitter.im/yiisoft/yii2
    - GitHub: https://github.com/yiisoft/yii2
    - Facebook: https://www.facebook.com/groups/yiitalk/
    - Twitter: https://twitter.com/yiiframework
    - LinkedIn: https://www.linkedin.com/groups/yii-framework-1483367
    - Stackoverflow: https://stackoverflow.com/questions/tagged/yii2

[33] https://www.yiiframework.com/books
[34] https://www.yiiframework.com/extensions/

Capítulo 3

Estructura de una aplicación

3.1. Información general

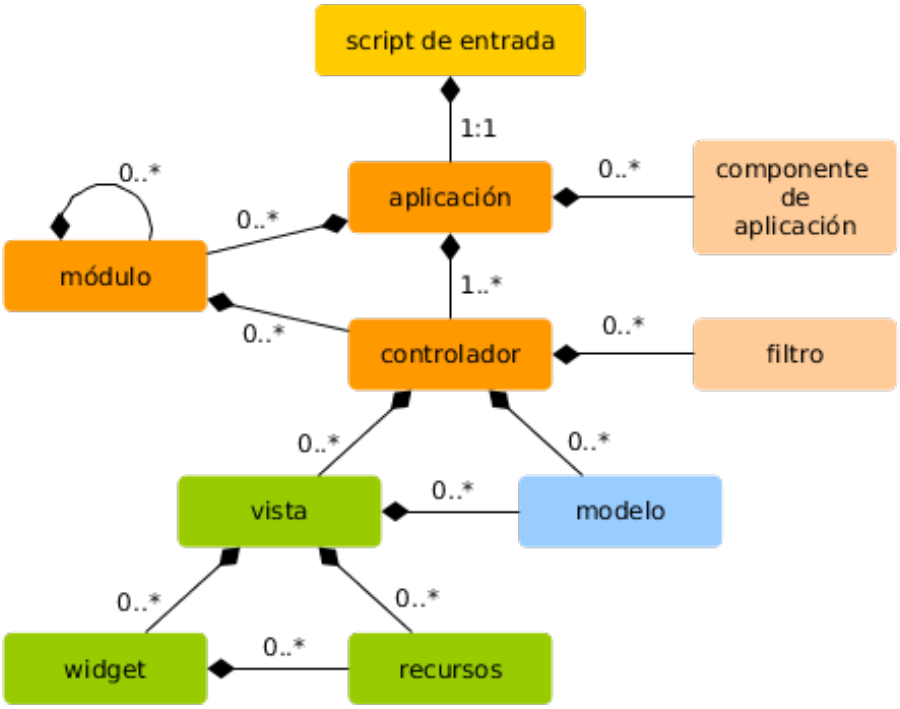
Las aplicaciones realizadas con Yii están organizadas de acuerdo al patrón de diseño modelo-vista-controlador (MVC)[1]. Los modelos representan datos, la lógica de negocios y sus reglas; las vistas son la representación de salida de los modelos; y finalmente, los controladores que toman datos de entrada y los convierten en instrucciones para los modelos y vistas.

Además de MVC, las aplicaciones Yii también tienen las siguientes entidades:

- scripts de entrada: Existen scripts PHP directamente accesibles a los usuarios finales. Son los responsables de comenzar el ciclo de manejo de una solicitud.
- aplicaciones: Son objetos accesibles globalmente que gestionan y coordinan los componentes de la aplicación con el fin de atender las diferentes solicitudes.
- componentes de la aplicación: Son los objetos registrados con la aplicación, y proporcionan varios servicios para cumplir las solicitudes.
- módulos: Son paquetes auto-contenidos los cuales por si solos poseen estructura MVC. Una aplicación puede estar organizada en términos de múltiples módulos.
- filtros: Representan el código que debe ser invocado antes y despues de la ejecución de cada solicitud por los controladores.
- widgets: Son objetos que pueden ser embebidos en las Vistas. Pueden contener lógica del controlador y ser reutilizados en múltiples vistas.

El siguiente esquema muestra la estructura estática de una aplicación:

[1] https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador



3.2. Scripts de Entrada

Los scripts de entrada son el primer eslabón en el proceso de arranque de la aplicación. Una aplicación (ya sea una aplicación Web o una aplicación de consola) tiene un único script de entrada. Los usuarios finales hacen peticiones al script de entrada que instancia instancias de aplicación y remite la petición a estos.

Los scripts de entrada para aplicaciones Web tiene que estar alojado bajo niveles de directorios accesibles para la Web de manera que puedan ser accesibles para los usuarios finales. Normalmente se nombra como `index.php`, pero también se pueden usar cualquier otro nombre, los servidores Web proporcionados pueden localizarlo.

El script de entrada para aplicaciones de consola normalmente está alojado bajo la ruta base de las aplicaciones y es nombrado como `yii` (con el sufijo `.php`). Estos deberían ser ejecutables para que los usuarios puedan ejecutar las aplicaciones de consola a través del comando `./yii <ruta> [argumentos] [opciones]`.

El script de entrada principalmente hace los siguientes trabajos:

- Definir las constantes globales;
- Registrar el cargador automático de Composer[2];

[2] https://getcomposer.org/doc/01-basic-usage.md#autoloading

- Incluir el archivo de clase `Yii`;
- Cargar la configuración de la aplicación;
- Crear y configurar una instancia de aplicación;
- Llamar a `yii\base\Application::run()` para procesar la petición entrante.

3.2.1. Aplicaciones Web

El siguiente código es el script de entrada para la Plantilla de Aplicación web Básica.

```
<?php

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

// registrar el cargador automático de Composer
require __DIR__ . '/../vendor/autoload.php';

// incluir el fichero de clase Yii
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';

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

// crear, configurar y ejecutar la aplicación
(new yii\web\Application($config))->run();
```

3.2.2. Aplicaciones de consola

De la misma manera, el siguiente código es el script de entrada para la aplicación de consola:

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 *
 * @link https://www.yiiframework.com/
 * @copyright Copyright (c) 2008 Yii Software LLC
 * @license https://www.yiiframework.com/license/
 */

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

// registrar el cargador automático de Composer
require __DIR__ . '/vendor/autoload.php';

// incluir el fichero de clase Yii
require __DIR__ . '/vendor/yiisoft/yii2/Yii.php';
```

```
// cargar la configuración de la aplicación
$config = require __DIR__ . '/config/console.php';

$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

3.2.3. Definición de Constantes

El script de entrada es el mejor lugar para definir constantes globales. Yii soporta las siguientes tres constantes:

- `YII_DEBUG`: especifica si la aplicación se está ejecutando en modo depuración. Cuando esta en modo depuración, una aplicación mantendrá más información de registro, y revelará detalladas pilas de errores si se lanza una excepción. Por esta razón, el modo depuración debería ser usado principalmente durante el desarrollo. El valor por defecto de 'YII_DEBUG' es falso.
- `YII_ENV`: especifica en que entorno se esta ejecutando la aplicación. Se puede encontrar una descripción más detallada en la sección Configuraciones. El Valor por defecto de `YII_ENV` es `'prod'`, que significa que la aplicación se esta ejecutando en el entorno de producción.
- `YII_ENABLE_ERROR_HANDLER`: especifica si se habilita el gestor de errores proporcionado por Yii. El valor predeterminado de esta constante es verdadero.

Cuando se define una constante, a menudo se usa código como el siguiente:

```
defined('YII_DEBUG') or define('YII_DEBUG', true);
```

que es equivalente al siguiente código:

```
if (!defined('YII_DEBUG')) {
    define('YII_DEBUG', true);
}
```

Claramente el primero es más breve y fácil de entender.

La definición de constantes debería hacerse al principio del script de entrada para que pueda tener efecto cuando se incluyan otros archivos PHP.

3.3. Aplicaciones

Las `Applications` (aplicaciones) son objetos que gobiernan la estructura total y el ciclo de vida de las aplicaciones hechas en Yii. Cada aplicación Yii contiene un objeto `Application` que es creado en el script de entrada y es globalmente accesible a través de la expresión `\Yii::$app`.

> **Información:** Dependiendo del contexto, cuando decimos "una aplicación", puede significar tanto un objeto Application o un sistema desarrollado en Yii.

Hay dos tipos de aplicaciones: `aplicaciones Web` y `aplicaciones de consola`. Como el nombre lo indica, la primera maneja principalmente Web requests mientras que la última maneja requests (peticiones) de la línea de comandos.

3.3.1. Configuraciones de las Aplicaciones

Cuando un script de entrada crea una aplicación, cargará una configuración y la aplicará a la aplicación, como se muestra a continuación:

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

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

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

Principalmente, las configuraciones de una aplicación especifican como inicializar las propiedades de un objeto `application`. Debido a que estas configuraciones suelen ser complejas, son usualmente guardadas en archivos de configuración, como en el archivo `web.php` del ejemplo anterior.

3.3.2. Propiedades de la Aplicación

Hay muchas propiedades importantes en la aplicación que deberían configurarse en en la configuración de la aplicación. Estas propiedades suelen describir el entorno en el cual la aplicación está corriendo. Por ejemplo, las aplicaciones necesitan saber cómo cargar controladores, dónde guardar archivos temporales, etc. A continuación, resumiremos esas propiedades.

Propiedades Requeridas

En cualquier aplicación, debes configurar al menos dos propiedades: `id` y `basePath`.

`id` La propiedad `id` especifica un ID único que diferencia una aplicación de otras. Es mayormente utilizada a nivel programación. A pesar de que no es un requerimiento, para una mejor interoperabilidad, se recomienda utilizar sólo caracteres alfanuméricos.

`basePath` La propiedad `basePath` especifica el directorio raíz de una aplicación. Es el directorio que alberga todos los archivos protegidos de un sistema. Bajo este directorio, tendrás normalmente sub-directorios como `models`, `views`, `controllers`, que contienen el código fuente correspondiente al patrón MVC.

Puedes configurar la propiedad `basePath` usando la ruta a un directorio o un alias. En ambas formas, el directorio debe existir, o se lanzará una excepción. La ruta será normalizada utilizando la función `realpath()`.

La propiedad `basePath` es utilizada a menudo derivando otras rutas (ej. la ruta `runtime`). Por esta razón, un alias llamado `@app` está predefinido para representar esta ruta. Rutas derivadas pueden ser entonces creadas a partir de este alias (ej. `@app/runtime` para referirse al directorio `runtime`).

Propiedades Importantes

Las propiedades descritas en esta subsección a menudo necesita ser configurada porque difieren entre las diferentes aplicaciones.

`aliases` Esta propiedad te permite definir un grupo de alias en términos de un array (matriz). Las claves del array son los nombres de los alias, y los valores su correspondiente definición. Por ejemplo:

```
[
    'aliases' => [
        '@name1' => 'path/to/path1',
        '@name2' => 'path/to/path2',
    ],
]
```

Esta propiedad está provista de tal manera que puedas definir alias en términos de configuraciones de la aplicación en vez de llamadas al método `Yii::setAlias()`.

`bootstrap` Esta es una propiedad importante. Te permite definir un array de los componentes que deben ejecutarse durante el `proceso de` de la aplicación. Por ejemplo, si quieres personalizar las reglas de URL de un módulo, podrías listar su ID como un elemento de este array.

Cada componente listado en esta propiedad puede ser especificado en cualquiera de los siguientes formatos:

- el ID de un componente como está especificado vía `components`.
- el ID de un módulo como está especificado vía `modules`.
- un nombre de clase.
- un array de configuración.

Por ejemplo:

```
[
    'bootstrap' => [
        // un ID de componente o de módulo
        'demo',

        // un nombre de clase
        'app\components\TrafficMonitor',
```

```
        // un array de configuración
        [
            'class' => 'app\components\Profiler',
            'level' => 3,
        ]
    ],
]
```

Durante el proceso de `bootstrapping`, cada componente será instanciado. Si la clase del componente implementa `yii\base\BootstrapInterface`, también se llamará a su método `bootstrap()`.

Otro ejemplo práctico se encuentra en la configuración del Template de Aplicación Básica, donde los módulos `debug` y `gii` son configurados como componentes `bootstrap` cuando la aplicación está corriendo en un entorno de desarrollo,

```
if (YII_ENV_DEV) {
    // ajustes en la configuración del entorno 'dev' (desarrollo)
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';

    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = 'yii\gii\Module';
}
```

> **Nota:** Agregar demasiados componentes `bootstrap` degradará la performance de tu aplicación debido a que por cada request, se necesita correr el mismo grupo de componentes. Por lo tanto, utiliza componentes `bootstrap` con criterio.

`catchAll` Esta propiedad está solamente soportada por `aplicaciones Web`. Especifica la acción de controlador que debería manejar todos los requests (peticiones) del usuario. Es mayormente utilizada cuando una aplicación está en "modo de mantenimiento" y necesita que todas las peticiones sean capturadas por una sola acción.

La configuración es un array cuyo primer elemento especifica la ruta de la acción. El resto de los elementos del array (pares clave-valor) especifica los parámetros a ser enviados a la acción. Por ejemplo:

```
[
    'catchAll' => [
        'offline/notice',
        'param1' => 'value1',
        'param2' => 'value2',
    ],
]
```

`components` Esta es la propiedad más importante. Te permite registrar una lista de componentes llamados componentes de aplicación que puedes utilizar en otras partes de tu aplicación. Por ejemplo:

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

Cada componente de la aplicación es un par clave-valor del array. La clave representa el ID del componente, mientras que el valor representa el nombre de la clase del componente o una configuración.

Puedes registrar cualquier componente en una aplicación, y el componente puede ser globalmente accedido utilizando la expresión `\Yii::$app->ComponentID`.

Por favor, lee la sección Componentes de la Aplicación para mayor detalle.

`controllerMap` Esta propiedad te permite mapear un ID de controlador a una clase de controlador arbitraria. Por defecto, Yii mapea ID de controladores a clases de controladores basado en una convención (ej. el ID `post` será mapeado a `app\controllers\PostController`). Configurando esta propiedad, puedes saltear esa convención para controladores específicos. En el siguiente ejemplo, `account` será mapeado a `app\controllers\UserController`, mientras que `article` será mapeado a `app\controllers\PostController`.

```
[
    'controllerMap' => [
        'account' => 'app\controllers\UserController',
        'article' => [
            'class' => 'app\controllers\PostController',
            'enableCsrfValidation' => false,
        ],
    ],
]
```

Las claves de este array representan los ID de los controladores, mientras que los valores representan los nombres de clase de dichos controladores o una configuración.

`controllerNamespace` Esta propiedad especifica el `namespace` bajo el cual las clases de los controladores deben ser ubicados. Por defecto es `app\controllers`.

Si el ID es `post`, por convención el controlador correspondiente (sin `namespace`) será `PostController`, y el nombre completo (cualificado) de la clase `app\controllers\PostController`.

Las clases de controladores pueden ser ubicados también en sub-directorios del directorio correspondiente a este `namespace`. Por ejemplo, dado el ID de controlador `admin/post`, el nombre completo de la clase sería `app\controllers\admin\PostController`.

Es importante que el nombre completo de la clase del controlador sea auto-cargable y el `namespace` actual de la clase coincida con este valor. De otro modo, recibirás un error "Page Not Found" ("Página no Encontrada") cuando accedas a la aplicación.

En caso de que quieras romper con la convención cómo se comenta arriba, puedes configurar la propiedad controllerMap.

`language` Esta propiedad especifica el idioma en el cual la aplicación debería mostrar el contenido a los usuarios. El valor por defecto de esta propiedad es `en`, referido a English. Deberías configurar esta propiedad si tu aplicación necesita soporte multi-idioma.

El valor de esta propiedad determina varios aspectos de la internacionalización, incluido la traducción de mensajes, formato de fecha y números, etc. Por ejemplo, el widget `yii\jui\DatePicker` utilizará el valor de esta propiedad para determinar en qué idioma el calendario debe ser mostrado y cómo dar formato a la fecha.

Se recomienda que especifiques el idioma en términos de una Código de idioma IETF[3]. Por ejemplo, `en` se refiere a English, mientras que `en-US` se refiere a English (United States).

Se pueden encontrar más detalles de este aspecto en la sección Internacionalización.

`modules` Esta propiedad especifica los módulos que contiene la aplicación.

Esta propiedad toma un array con los nombre de clases de los módulos o configuraciones con las claves siendo los IDs de los módulos. Por ejemplo:

```
[
    'modules' => [
        // módulo "booking" especificado con la clase del módulo
        'booking' => 'app\modules\booking\BookingModule',

        // módulo "comment" especificado usando un array de configuración
        'comment' => [
            'class' => 'app\modules\comment\CommentModule',
            'db' => 'db',
        ],
```

[3] https://es.wikipedia.org/wiki/C\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\endgroup\relax\let\ignorespaces\relax\accent1o\egroup\spacefactor\accent@spacefactordigo_de_idioma_IETF

```
    ],
]
```

Por favor consulta la sección Módulos para más detalles.

`name` Esta propiedad especifica el nombre de la aplicación que será mostrado a los usuarios. Al contrario de `id`, que debe tomar un valor único, el valor de esta propiedad existe principalmente para propósito de visualización y no tiene porqué ser única.

No siempre necesitas configurar esta propiedad si en tu aplicación no va a ser utilizada.

`params` Esta propiedad especifica un array con parámetros accesibles desde cualquier lugar de tu aplicación. En vez de usar números y cadenas fijas por todos lados en tu código, es una buena práctica definirlos como parámetros de la aplicación en un solo lugar y luego utilizarlos donde los necesites. Por ejemplo, podrías definir el tamaño de las imágenes en miniatura de la siguiente manera:

```
[
    'params' => [
        'thumbnail.size' => [128, 128],
    ],
]
```

Entonces, cuando necesites acceder a esa configuración en tu aplicación, podrías hacerlo utilizando el código siguiente:

```
$size = \Yii::$app->params['thumbnail.size'];
$width = \Yii::$app->params['thumbnail.size'][0];
```

Más adelante, si decides cambiar el tamaño de las miniaturas, sólo necesitas modificarlo en la configuración de la aplicación sin necesidad de tocar el código que lo utiliza.

`sourceLanguage` Esta propiedad especifica el idioma en el cual la aplicación está escrita. El valor por defecto es `'en-US'`, referido a English (United States). Deberías configurar esta propiedad si el contenido de texto en tu código no está en inglés.

Como la propiedad language, deberías configurar esta propiedad siguiendo el Código de idioma IETF[4]. Por ejemplo, `en` se refiere a English, mientras que `en-US` se refiere a English (United States).

Puedes encontrar más detalles de esta propiedad en la sección Internacionalización.

[4] https://es.wikipedia.org/wiki/C\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\let\begingroup\endgroup\relax\let\ignorespaces\relax\accent1o\egroup\spacefactor\accent@spacefactordigo_de_idioma_IETF

`timeZone` Esta propiedad es provista como una forma alternativa de definir el `time zone` de PHP por defecto en tiempo de ejecución. Configurando esta propiedad, escencialmente estás llamando a la función de PHP date_default_timezone_set()[5]. Por ejemplo:

```
[
    'timeZone' => 'America/Los_Angeles',
]
```

`version` Esta propiedad especifica la versión de la aplicación. Es por defecto `'1.0'`. No hay total necesidad de configurarla si tu no la usarás en tu código.

Propiedades Útiles

Las propiedades especificadas en esta sub-sección no son configuradas normalmente ya que sus valores por defecto estipulan convenciones comunes. De cualquier modo, aún puedes configurarlas en caso de que quieras romper con la convención.

`charset` Esta propiedad especifica el `charset` que la aplicación utiliza. El valor por defecto es `'UTF-8'`, que debería ser mantenido tal cual para la mayoría de las aplicaciones a menos que estés trabajando con sistemas legados que utilizan muchos datos no-unicode.

`defaultRoute` Esta propiedad especifica la ruta que una aplicación debería utilizar si el `request` no especifica una. La ruta puede consistir el ID de un submódulo, el ID de un controlador, y/o el ID de una acción. Por ejemplo, `help`, `post/create`, `admin/post/create`. Si el ID de la acción no se especifica, tomará el valor por defecto especificado en `yii\base\Controller::$defaultAction`.

Para `aplicaciones Web`, el valor por defecto de esta propiedad es `'site'`, lo que significa que el controlador `SiteController` y su acción por defecto serán usados. Como resultado, si accedes a la aplicación sin especificar una ruta, mostrará el resultado de `app\controllers\SiteController::actionIndex()`.

Para `aplicaciones de consola`, el valor por defecto es `'help'`, lo que significa que el comando `yii\console\controllers\HelpController::actionIndex()` debería ser utilizado. Como resultado, si corres el comando `yii` sin proveer ningún argumento, mostrará la información de ayuda.

`extensions` Esta propiedad especifica la lista de extensiones que se encuentran instaladas y son utilizadas por la aplicación. Por defecto, tomará el array devuelto por el archivo `@vendor/yiisoft/extensions.php`. El archivo `extensions.php` es generado y mantenido automáticamente cuando utilizas

[5] https://www.php.net/manual/es/function.date-default-timezone-set.php

Composer[6] para instalar extensiones. Por lo tanto, en la mayoría de los casos no necesitas configurarla.

En el caso especial de que quieras mantener las extensiones a mano, puedes configurar la propiedad como se muestra a continuación:

```
[
    'extensions' => [
        [
            'name' => 'nombre de la extensión',
            'version' => 'número de versión',
            'bootstrap' => 'BootstrapClassName',  // opcional, puede ser
            también un array de configuración
            'alias' => [  // opcional
                '@alias1' => 'to/path1',
                '@alias2' => 'to/path2',
            ],
        ],

        // ... más extensiones como las de arriba ...

    ],
]
```

Como puedes ver, la propiedad toma un array de especificaciones de extensiones. Cada extensión es especificada mediante un array que consiste en los elementos `name` y `version`. Si una extensión necesita ser ejecutada durante el proceso de `bootstrap`, un elemento `bootstrap` puede ser especificado con un nombre de clase o un array de configuración. Una extensión también puede definir algunos alias.

`layout` Esta propiedad especifica el valor del `layout` por defecto que será utilizado al renderizar una vista. El valor por defecto es `'main'`, y se refiere al archivo `main.php` bajo el `layout path` definido. Si tanto el `layout path` y el `view path` están utilizando los valores por defecto, el archivo `layout` puede ser representado con el alias `@app/views/layouts/main.php`.

Puedes configurar esta propiedad con el valor `false` si quieres desactivar el `layout` por defecto, aunque esto sería un caso muy raro.

`layoutPath` Esta propiedad especifica el lugar por defecto donde deben buscarse los archivos `layout`. El valor por defecto es el sub-directorio `layouts` bajo el `view path`. Si el `view path` usa su valor por defecto, el `layout path` puede ser representado con el alias `@app/views/layouts`.

Puedes configurarlo como un directorio o utilizar un alias.

[6] https://getcomposer.org

`runtimePath` Esta propiedad especifica dónde serán guardados los archivos temporales, como archivos de log y de cache, pueden ser generados. El valor por defecto de esta propiedad es el alias `@app/runtime`.

Puedes configurarlo como un directorio o utilizar un alias. Ten en cuenta que el directorio debe tener permisos de escritura por el proceso que corre la aplicación. También este directorio debe estar protegido de ser accedido por usuarios finales, ya que los archivos generados pueden tener información sensible.

Para simplificar el acceso a este directorio, Yii trae predefinido el alias `@runtime` para él.

`viewPath` Esta propiedad especifica dónde están ubicados los archivos de la vista. El valor por defecto de esta propiedad está representado por el alias `@app/views`. Puedes configurarlo como un directorio o utilizar un alias.

`vendorPath` Esta propiedad especifica el directorio `vendor` que maneja Composer[7]. Contiene todas las librerías de terceros utilizadas por tu aplicación, incluyendo el núcleo de Yii. Su valor por defecto está representado por el alias `@app/vendor`.

Puedes configurarlo como un directorio o utilizar un alias. Cuando modificas esta propiedad, asegúrate de ajustar la configuración de Composer en concordancia.

Para simplificar el acceso a esta ruta, Yii trae predefinido el alias `@vendor`.

`enableCoreCommands` Esta propiedad está sólo soportada por `aplicaciones de consola`. Especifica si los comandos de consola incluidos en Yii deberían estar habilitados o no. Por defecto está definido como `true`.

3.3.3. Eventos de la Aplicación

Una aplicación dispara varios eventos durante su ciclo de vida al manejar un `request`. Puedes conectar manejadores a dichos eventos en la configuración de la aplicación como se muestra a continuación:

```
[
    'on beforeRequest' => function ($event) {
        // ...
    },
]
```

El uso de la sintáxis `on nombreEvento` es descrita en la sección Configuraciones.

Alternativamente, puedes conectar manejadores de eventos durante el `proceso de bootstrapping` después de que la instancia de la aplicación es creada. Por ejemplo:

[7] `https://getcomposer.org`

```
\Yii::$app->on(\yii\base\Application::EVENT_BEFORE_REQUEST, function
($event) {
    // ...
});
```

`EVENT_BEFORE_REQUEST`

Este evento es disparado *before* (antes) de que la aplicación maneje el `request`. El nombre del evento es `beforeRequest`.

Cuando este evento es disparado, la instancia de la aplicación ha sido configurada e inicializada. Por lo tanto es un buen lugar para insertar código personalizado vía el mecanismo de eventos para interceptar dicho manejo del `request`. Por ejemplo, en el manejador del evento, podrías definir dinámicamente la propiedad `yii\base\Application::$language` basada en algunos parámetros.

`EVENT_AFTER_REQUEST`

Este evento es disparado *after* (después) de que una aplicación finaliza el manejo de un `request` pero *before* (antes) de enviar el `response` (respuesta). El nombre del evento es `afterRequest`.

Cuando este evento es disparado, el manejo del `request` está finalizado y puedes aprovechar para realizar algún post-proceso del mismo o personalizar el `response` (respuesta).

Ten en cuenta que el componente `response` también dispara algunos eventos mientras está enviando el contenido a los usuarios finales. Estos eventos son disparados *after* (después) de este evento.

`EVENT_BEFORE_ACTION`

Este evento es disparado *before* (antes) de ejecutar cualquier acción de controlador. El nombre de este evento es `beforeAction`.

El parámetro evento es una instancia de `yii\base\ActionEvent`. Un manejador de eventos puede definir la propiedad `yii\base\ActionEvent::$isValid` como `false` para detener la ejecución de una acción. Por ejemplo:

```
[
    'on beforeAction' => function ($event) {
        if (..alguna condición..) {
            $event->isValid = false;
        } else {
        }
    },
]
```

Ten en cuenta que el mismo evento `beforeAction` también es disparado por módulos y [controladores)(structure-controllers.md). Los objectos aplicación

son los primeros en disparar este evento, seguidos por módulos (si los hubiera), y finalmente controladores. Si un manejador de eventos define `yii\base\ActionEvent::$isValid` como `false`, todos los eventos siguientes NO serán disparados.

`EVENT_AFTER_ACTION`

Este evento es disparado *after* (después) de ejecutar cualquier acción de controlador. El nombre de este evento es `afterAction`.

El parámetro evento es una instancia de `yii\base\ActionEvent`. A través de la propiedad `yii\base\ActionEvent::$result`, un manejador de eventos puede acceder o modificar el resultado de una acción. Por ejemplo:

```
[
    'on afterAction' => function ($event) {
        if (..alguna condición...) {
            // modificar $event->result
        } else {
        }
    },
]
```

Ten en cuenta que el mismo evento `afterAction` también es disparado por módulo y [controladores)(structure-controllers.md). Estos objetos disparan el evento en orden inverso que los de `beforeAction`. Esto quiere decir que los controladores son los primeros en dispararlo, seguido por módulos (si los hubiera), y finalmente aplicaciones.

3.3.4. Ciclo de Vida de una Aplicación

Cuando un script de entrada está siendo ejecutado para manejar un `request`, una aplicación experimenta el siguiente ciclo de vida:

1. El script de entrada carga el array de configuración de la aplicación.
2. El script de entrada crea una nueva instancia de la aplicación:
   - Se llama a `preInit()`, que configura algunas propiedades de alta prioridad de la aplicación, como `basePath`.
   - Registra el `manejador de errores`.
   - Configura las propiedades de la aplicación.
   - Se llama a `init()` con la subsiguiente llamada a `bootstrap()` para correr componentes `bootstrap`.
3. El script de entrada llama a `yii\base\Application::run()` para correr la aplicación:
   - Dispara el evento `EVENT_BEFORE_REQUEST`.

- Maneja el `request`: lo resuelve en una route (ruta) con los parámetros asociados; crea el módulo, controlador y objetos acción como se especifica en dicha ruta; y entonces ejecuta la acción.
- Dispara el evento `EVENT_AFTER_REQUEST`.
- Envía el `response` (respuesta) al usuario.

4. El script de entrada recibe el estado de salida de la aplicación y completa el proceso del `request`.

3.4. Componentes de la Aplicación

Las aplicaciones son service locators (localizadores de servicios). Ellas albergan un grupo de los llamados *componentes de aplicación* que proveen diferentes servicios para procesar el `request` (petición). Por ejemplo, el componente `urlManager` es responsable por rutear Web `requests` (peticiones) a los controladores apropiados; el componente `db` provee servicios relacionados a base de datos; y así sucesivamente.

Cada componente de la aplicación tiene un ID que lo identifica de forma inequívoca de otros en la misma aplicación. Puedes acceder a un componente de la aplicación con la siguiente expresión:

```
\Yii::$app->ComponentID
```

Por ejemplo, puedes utilizar `\Yii::$app->db` para obtener la `conexión a la base de datos`, y `\Yii::$app->cache` para obtener el `cache primario` registrado con la aplicación.

Estos componentes pueden ser cualquier objeto. Puedes registrarlos configurando la propiedad `yii\base\Application::$components` en las configuraciones de la aplicación. Por ejemplo:

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

        // registra el componente "db" utilizando un array de configuración
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],

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

```
    ],
]
```

> **Información:** A pesar de que puedes registrar tantos componentes como desees, deberías hacerlo con criterio. Los componente de la aplicación son como variables globales. Abusando demasiado de ellos puede resultar en un código más difícil de mantener y testear. En muchos casos, puedes simplemente crear un componente local y utilizarlo únicamente cuando sea necesario.

3.4.1. Componentes del Núcleo de la Aplicación

Yii define un grupo de componentes del *núcleo* con IDs fijos y configuraciones por defecto. Por ejemplo, el componente `request` es utilizado para recolectar información acerca del `request` del usuario y resolverlo en una ruta; el componente `db` representa una conexión a la base de datos a través del cual realizar consultas a la misma. Es con ayuda de estos componentes del núcleo que Yii puede manejar los `request` del usuario.

A continuación, hay una lista de componentes predefinidos en el núcleo. Puedes configurarlos y personalizarlos como lo haces con componentes normales de la aplicación. Cuando configuras un componente del núcleo, si no especificas su nombre de clase, la clase por defecto será utilizada.

- `assetManager`: maneja los `assets bundles` y su publicación. Consulta la sección Menajando Assets para más detalles.
- `db`: representa una conexión a la base de datos a través de la cual puedes realizar consultas a la misma. Ten en cuenta que cuando configuras este componente, debes especificar el nombre de clase así como otras propiedades requeridas por el mismo, como `yii\db\Connection::$dsn`. Por favor consulta la sección Data Access Objects para más detalles.
- `errorHandler`: maneja errores y excepciones de PHP. Por favor consulta la sección Handling Errors para más detalles.
- `yii\base\Formatter`: da formato a los datos cuando son mostrados a los usuarios. Por ejemplo, un número puede ser mostrado usando un separador de miles, una fecha en una forma extensa. Por favor consulta la sección Formato de Datos para más detalles.
- `i18n`: soporta traducción y formato de mensajes. Por favor consulta la sección Internacionalización para más detalles.
- `log`: maneja a dónde dirigir los logs. Por favor consulta la sección Logging para más detalles.
- `yii\swiftmailer\Mailer`: soporta construcción y envío de emails. Por favor consulta la sección Enviando Emails para más detalles.
- `response`: representa la respuesta enviada a los usuarios. Por favor consulta la sección Responses para más detalles.

- `request`: representa el `request` recibido de los usuarios. Por favor consulta la sección Requests para más detalles.
- `session`: representa la información de sesión. Este componente sólo está disponible en `alpicaciones Web`. Por favor consulta la sección Sessions and Cookies para más detalles.
- `urlManager`: soporta el parseo y generación de URLs. Por favor consulta la sección URL Parsing and Generation para más detalles.
- `user`: representa la información e autenticación del usuario. Este componente sólo está disponible en `aplicaciones Web` Por favor consulta la sección Autenticación para más detalles.
- `view`: soporta el renderizado de las vistas. Por favor consulta la sección Vistas para más detalles.

3.5. Controladores

Los controladores son parte del patrón o arquitectura MVC[8]. Son objetos que extienden de `yii\base\Controller` y se encargan de procesar los `requests` (consultas) generando `responses` (respuestas). Particularmente, después de tomar el control desde las aplicaciones, los controladores analizarán los datos que entran en el `request`, los pasan a los modelos, inyectan los modelos resultantes a las vistas, y finalmente generan los `responses` (respuestas) de salida.

3.5.1. Acciones

Los Controladores están compuestos por *acciones* que son las unidades más básicas a las que los usuarios pueden dirigirse y solicitar ejecución. Un controlador puede tener una o múltiples acciones.

El siguiente ejemplo muestra un controlador `post` con dos acciones: `view` y `create`:

```
namespace app\controllers;

use Yii;
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

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

[8] https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador

```
        }

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

    public function actionCreate()
    {
        $model = new 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 `view` (definida en el método `actionView()`), el código primero carga el modelo de acuerdo el ID del modelo solicitado; Si el modelo es cargado satisfactoriamente, lo mostrará usando una vista llamada `view`. Si no, arrojará una excepción.

En la acción `create` (definida por el método `actionCreate()`), el código es similar. Primero intenta poblar el modelo usando datos del `request` y guardarlo. Si ambas cosas suceden correctamente, se redireccionará el navegador a la acción `view` con el ID del modelo recientemente creado. De otro modo mostrará la vista `create` a través de la cual el usuario puede completar los campos necesarios.

3.5.2. Routes

Los usuarios ejecutan las acciones a través de las llamadas *routes* (rutas). una ruta es una cadena que consiste en las siguientes partes:

- un ID de módulo: este existe solamente si el controlador pertenece a un módulo que no es de la aplicación;
- un ID de controlador: una cadena que identifica exclusivamente al controlador entre todos los controladores dentro de la misma aplicación (o el mismo módulo si el controlador pertenece a uno);
- un ID de acción: una cadena que identifica exclusivamente a la acción entre todas las acciones del mismo controlador.

Las rutas pueden usar el siguiente formato:

```
ControllerID/ActionID
```

o el siguiente formato si el controlador pertenece a un módulo:

```
ModuleID/ControllerID/ActionID
```

Entonces si un usuario solicita la URL `https://hostname/index.php?r=site/index`, la acción `index` del controlador `site` será ejecutado. Para más detalles acerca de cómo las son resueltas en acciones, por favor consulta la sección Routing.

3.5.3. Creando Controladores

En `aplicaciones Web`, los controladores deben extender de `yii\web\Controller` o cualquier clase hija. De forma similar los controladores de `aplicaciones de consola`, deben extender de `yii\console\Controller` o cualquier clase hija de esta. El siguiente código define un controlador `site`:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
}
```

IDs de Controladores

Normalmente, un controlador está diseñado para manejar los `requests` de acuerdo a un tipo de recurso. Por esta razón, los IDs de controladores son a menudo sustantivos de los tipos de recurso que están manejando. Por ejemplo, podrías utilizar `article` como el ID de un controlador que maneja datos de artículos.

Por defecto, los IDs de controladores deberían contener sólo estos caracteres: letras del Inglés en minúscula, dígitos, guiones bajos y medios, y barras. Por ejemplo, `article`, `post-comment`, `admin/post-comment` son todos IDs de controladores válidos, mientras que `article?`, `PostComment`, `admin\post` no lo son.

Los guiones en un ID de controlador son utilizados para separar palabras, mientras que las barras diagonales lo son para organizar los controladores en sub-directorios.

Nombres de Clases de Controladores

Los nombres de clases de controladores pueden ser derivados de los IDs de acuerdo a las siguientes reglas:

- Transforma la primera letra de cada palabra separada por guiones en mayúscula. Nota que si el ID del controlador contiene barras, esta regla sólo aplica a la porción después de la última barra dentro del ID.
- Elimina guiones y reemplaza cualquier barra diagonal por barras invertidas.

- Agrega el sufijo `Controller`.
- Agrega al principio el `controller namespace`.

A continuación mostramos algunos ejemplos, asumiendo que el `controller namespace` toma el valor por defecto: `app\controllers`:

- `article` deriva en `app\controllers\ArticleController`;
- `post-comment` deriva en `app\controllers\PostCommentController`;
- `admin/post-comment` deriva en `app\controllers\admin\PostCommentController`.

Las clases de controladores deben ser autocargables. Por esta razón, en los ejemplos anteriores, la clase del controlador `article` debe ser guardada en un archivo cuyo alias alias es `@app/controllers/ArticleController.php`; mientras que el controlador `admin/post-comment` debería estar en `@app/controllers/admin/PostCommentController.php`.

> **Información:** En el último ejemplo, `admin/post-comment`, demuestra cómo puedes poner un controlador bajo un sub-directorio del `controller namespace`. Esto es útil cuando quieres organizar tus controladores en varias categorías pero sin utilizar módulos.

Controller Map

Puedes configurar `controller map` (mapeo de controladores) para superar las restricciones de los IDs de controladores y sus nombres de clase descritos arriba. Esto es principalmente útil cuando estás utilizando un controlador de terceros del cual no tienes control alguno sobre sus nombres de clase.

Puedes configurar `controller map` en la configuración de la aplicación de la siguiente manera:

```
[
    'controllerMap' => [
        [
            // declara el controlador "account" usando un nombre de clase
            'account' => 'app\controllers\UserController',

            // declara el controlador "article" utilizando un array de
            configuración
            'article' => [
                'class' => 'app\controllers\PostController',
                'enableCsrfValidation' => false,
            ],
        ],
    ],
]
```

Controller por Defecto

Cada aplicación tiene un controlador por defecto especificado a través de la propiedad `yii\base\Application::$defaultRoute`. Cuando un `request` no especifica una ruta, se utilizará la ruta especificada en esta propiedad.

Para `aplicaciones Web`, el valor es `'site'`, mientras que para `aplicaciones de consola` es `help`. Por lo tanto, si la URL es `https://hostname/index.php`, significa que el `request` será manejado por el controlador `site`.

Puedes cambiar el controlador por defecto con la siguiente configuración de la aplicación:

```
[
    'defaultRoute' => 'main',
]
```

3.5.4. Creando Acciones

Crear acciones puede ser tan simple como definir un llamado *método de acción* en una clase controlador. Un método de acción es un método *public* cuyo nombre comienza con la palabra `action`. El valor de retorno de uno de estos métodos representa los datos de respuesta (response) a ser enviado a los usuarios. El siguiente código define dos acciones: `index` y `hello-world`:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actionIndex()
    {
        return $this->render('index');
    }

    public function actionHelloWorld()
    {
        return 'Hola Mundo!';
    }
}
```

IDs de Acciones

Una acción está a menudo diseñada para realizar una manipulación particular de un recurso. Por esta razón, los IDs de acciones son usualmente verbos, como `view` (ver), `update` (actualizar), etc.

Por defecto, los IDs de acciones deberían contener estos caracteres solamente: letras en Inglés en minúsculas, dígitos, guiones bajos y barras. Los guiones en un ID de acción son utilizados para separar palabras. Por ejemplo, `view`, `update2`, `comment-post` son todos IDs válidos, mientras que `view?` y `Update` no lo son.

Puedes crear acciones de dos maneras: acciones en línea (inline) o acciones independientes (standalone). Una acción en línea es definida como un método en la clase del controlador, mientras que una acción independiente

es una clase que extiende `yii\base\Action` o sus clases hijas. Las acciones en línea son más fáciles de crear y por lo tanto preferidas si no tienes intenciones de volver a utilizarlas. Las acciones independientes, por otro lado, son principalmente creadas para ser reutilizadas en otros controladores o para ser redistribuidas como extensiones.

Acciones en Línea

Como acciones en línea nos referimos a acciones que son definidas en términos de métodos como acabamos de describir.

Los nombre de métodos de acciones derivan de los IDs de acuerdo al siguiente criterio:

- Transforma la primera letra de cada palabra del ID de la acción a mayúscula;
- Elimina guiones;
- Prefija la palabra `action`.

Por ejemplo, `index` se vuelve `actionIndex`, y `hello-world` se vuelve `actionHelloWorld`.

> **Nota:** Los nombres de los métodos de acción son *case-sensitive* (distinguen entre minúsculas y mayúsculas). Si tienes un método llamado `ActionIndex`, no será considerado como un método de acción, y como resultado, solicitar la acción `index` resultará en una excepción. También ten en cuenta que los métodos de acción deben ser `public`. Un método `private` o `protected` NO define un método de acción.

Las acciones en línea son las más comúnmente definidas ya que requieren muy poco esfuerzo de creación. De todos modos, si planeas reutilizar la misma acción en diferentes lugares, o quieres redistribuir una acción, deberías considerar definirla como un *acción independiente*.

Acciones Independientes

Las acciones independientes son acciones definidas en términos de clases de acción que extienden de `yii\base\Action` o cualquiera de sus clases hijas. Por ejemplo, en Yii se encuentran las clases `yii\web\ViewAction` y `yii\web\ErrorAction`, de las cuales ambas son acciones independientes.

Para utilizar una acción independiente, debes declararla en el *action map* (mapeo de acciones) sobrescribiendo el método `yii\base\Controller::actions()` en tu controlador de la siguiente manera:

```
public function actions()
{
    return [
        // declara la acción "error" utilizando un nombre de clase
        'error' => 'yii\web\ErrorAction',
```

```
        // declara la acción "view" utilizando un array de configuración
        'view' => [
            'class' => 'yii\web\ViewAction',
            'viewPrefix' => '',
        ],
    ];
}
```

Como puedes ver, el método `actions()` debe devolver un array cuyas claves son los IDs de acciones y sus valores los nombres de clases de acciones o configuraciones. Al contrario de acciones en línea, los IDs de acciones independientes pueden contener caracteres arbitrarios, mientras sean declarados en el método `actions()`.

Para crear una acción independiente, debes extender de `yii\base\Action` o una clase hija, e implementar un método `public` llamado `run()`. El rol del método `run()` es similar al de un método de acción. Por ejemplo:

```
<?php
namespace app\components;

use yii\base\Action;

class HelloWorldAction extends Action
{
    public function run()
    {
        return "Hola Mundo!";
    }
}
```

Resultados de Acción

El valor de retorno de una método de acción o del método `run()` de una acción independiente son significativos. Este se refiere al resultado de la acción correspondiente.

El valor devuelto puede ser un objeto response que será enviado como respuesta a los usuarios.

- Para `aplicaciones Web`, el valor de retorno pueden ser también datos arbitrarios que serán asignados a `yii\web\Response::$data` y más adelante convertidos a una cadena representando el cuerpo de la respuesta.
- Para `aplicaciones de consola`, el valor de retorno puede ser también un entero representando el `status de salida` de la ejecución del comando.

En los ejemplos mostrados arriba, los resultados de las acciones son todas cadenas que serán tratadas como el cuerpo de la respuesta a ser enviado a los usuarios. El siguiente ejemplo demuestra cómo una acción puede redirigir

el navegador del usuario a una nueva URL devolviendo un objeto `response` (debido a que el método `redirect()` devuelve un objeto `response`):

```
public function actionForward()
{
    // redirige el navegador del usuario a https://example.com
    return $this->redirect('https://example.com');
}
```

Parámetros de Acción

Los métodos de acción para acciones en línea y el método `run()` de acciones independientes pueden tomar parámetros, llamados *parámetros de acción*. Sus valores son obtenidos del `request`. Para `aplicaciones Web`, el valor de cada parámetro de acción es tomado desde `$_GET` usando el nombre del parámetro como su clave; para `aplicaciones de consola`, estos corresponden a los argumentos de la línea de comandos.

En el siguiente ejemplo, la acción `view` (una acción en línea) declara dos parámetros: `$id` y `$version`.

```
namespace app\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    public function actionView($id, $version = null)
    {
        // ...
    }
}
```

Los parámetros de acción serán poblados como se muestra a continuación para distintos `requests`:

- `https://hostname/index.php?r=post/view&id=123`: el parámetro `$id` tomará el valor `'123'`, mientras que `$version` queda como `null` debido a que no hay un parámetro `version` en la URL.
- `https://hostname/index.php?r=post/view&id=123&version=2`: los parámetros `$id` y `$version` serán llenados con `'123'` y `'2'`, respectivamente.
- `https://hostname/index.php?r=post/view`: se lanzará una excepción `yii\web\BadRequestHttpException` dado que el parámetro `$id` es requerido pero no es provisto en el `request`.
- `https://hostname/index.php?r=post/view&id[]=123`: una excepción `yii\web\BadRequestHttpException` será lanzada porque el parámetro `$id` está recibiendo un valor inesperado, el array `['123']`.

Si quieres que un parámetro de acción acepte un array como valor, deberías utilizar el `type-hinting` (especificación de tipo) `array`, como a continuación:

```
public function actionView(array $id, $version = null)
{
    // ...
}
```

Ahora si el `request` es `https://hostname/index.php?r=post/view&id[]=123`, el parámetro `$id` tomará el valor de `['123']`. Si el `request` es `https://hostname/index.php?r=post/view&id=123`, el parámetro `$id` recibirá aún el mismo array como valor ya que el valor escalar `'123'` será convertido automáticamente en array.

Los ejemplos de arriba muestran principalmente como funcionan los parámetros de acción de una aplicación Web. Para aplicaciones de consola, por favor consulta la sección Comandos de Consola para más detalles.

Acción por Defecto

Cada controlador tiene una acción por defecto especificada a través de la propiedad `yii\base\Controller::$defaultAction`. Cuando una ruta contiene sólo el ID del controlador, implica que se está solicitando la acción por defecto del controlador especificado.

Por defecto, la acción por defecto (valga la redundancia) definida es `index`. Si quieres cambiar dicho valor, simplemente sobrescribe esta propiedad en la clase del controlador, como se muestra a continuación:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $defaultAction = 'home';

    public function actionHome()
    {
        return $this->render('home');
    }
}
```

3.5.5. Ciclo de Vida del Controlador

Cuando se procesa un `request`, la aplicación creará un controlador basado en la ruta solicitada. El controlador entonces irá a través del siguiente ciclo de vida para completar el `request`:

1. El método `yii\base\Controller::init()` es llamado después de que el controlador es creado y configurado.

2. El controlador crea un objecto `action` basado en el ID de acción solicitado:

- Si el ID de la acción no es especificado, el `ID de la acción por defecto` será utilizado.
- Si el ID de la acción es encontrado en el `mapeo de acciones`, una acción independiente será creada;
- Si el ID de la acción es coincide con un método de acción, una acción en línea será creada;
- De otra manera, se lanzará una excepción `yii\base\InvalidRouteException`.

3. El controlador llama secuencialmente al método `beforeAction()` de la aplicación, al del módulo (si el controlador pertenece a uno) y al del controlador.
   - Si alguna de las llamadas devuelve `false`, el resto de los llamados subsiguientes a `beforeAction()` serán saltados y la ejecución de la acción será cancelada.
   - Por defecto, cada llamada al método `beforeAction()` lanzará un evento `beforeAction` al cual le puedes conectar un manejador.
4. El controlador ejecuta la acción:
   - Los parámetros de la acción serán analizados y poblados con los datos del `request`;
5. El controlador llama secuencialmente al método `afterAction()` del controlador, del módulo (si el controlador pertenece a uno) y de la aplicación.
   - Por defecto, cada llamada al método `afterAction()` lanzará un evento `afterAction` al cual le puedes conectar un manejador.
6. La aplicación tomará el resultado de la acción y lo asignará al response.

3.5.6. Buenas Prácticas

En una aplicación bien diseñada, los controladores son a menudo muy pequeños con cada acción conteniendo unas pocas líneas de código. Si tu controlador se torna muy complejo, es usualmente un indicador de que deberías realizar una refactorización y mover algo de código a otras clases.

En resumen, los controladores

- pueden acceder a los datos del request;
- puede llamar a métodos del modelo y otros componentes con data del `request`;
- pueden utilizar vistas para componer `responses`;
- NO debe procesar datos del 'request - esto debe ser realizado en los modelos;
- deberían evitar insertar HTML o cualquier código de presentación - para esto están las vistas.

3.6. Modelos

Los modelos forman parte de la arquitectura MVC[9]. Son objetos que representan datos de negocio, reglas y lógica.

Se pueden crear clases modelo extendiendo a `yii\base\Model` o a sus clases hijas. La clase base `yii\base\Model` soporta muchas características útiles:

- Atributos: representan los datos de negocio y se puede acceder a ellos como propiedades normales de un objeto o como elementos de un array;
- Etiquetas de atributo: especifica la etiqueta a mostrar para los atributos;
- Asignación masiva: soporta la asignación múltiple de atributos en un único paso;
- validación: asegura la validez de los datos de entrada basándose en reglas declaradas;
- Exportación de datos: permite que los datos del modelo sean exportados en términos de arrays con formatos personalizables.

La clase 'modelo' también es una base para modelos más avanzados, tales como Active Records.

> **Información:** No es obligatorio basar las clases modelo en `yii\base\Model`. Sin embargo, debido a que hay muchos componentes de Yii construidos para dar soporte a `yii\base\Model`, por lo general, es la clase base preferible para un modelo.

Atributos

Los modelos representan los datos de negocio en términos de *atributos*. Cada atributos es como una propiedad públicamente accesible de un modelo. El método `yii\base\Model::attributes()` especifica qué atributos tiene la clase modelo.

Se puede acceder a un atributo como se accede a una propiedad de un objeto normal.

```
$model = new \app\models\ContactForm;

// "name" es un atributo de ContactForm
$model->name = 'example';
echo $model->name;
```

También se puede acceder a los atributos como se accede a los elementos de un array, gracias al soporte para ArrayAccess[10] y ArrayIterator[11] que brinda `yii\base\Model`:

[9] https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador
[10] https://www.php.net/manual/es/class.arrayaccess.php
[11] https://www.php.net/manual/es/class.arrayiterator.php

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

// acceder a atributos como elementos de array
$model['name'] = 'example';
echo $model['name'];

// iterar entre atributos
foreach ($model as $name => $value) {
    echo "$name: $value\n";
}
```

Definir Atributos

Por defecto, si un modelo extiende directamente a `yii\base\Model`, todas sus variables miembro no estáticas son atributos. Por ejemplo, la siguiente clase modelo 'ContactForm' tiene cuatro atributos: 'name', 'email', 'subject', 'body'. El modelo 'ContactForm' se usa para representar los datos de entrada recibidos desde un formulario HTML.

```php
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;
}
```

Se puede sobrescribir `yii\base\Model::attributes()` para definir los atributos de diferente manera. El método debe devolver los nombres de los atributos de un modelo. Por ejemplo `yii\db\ActiveRecord` lo hace devolviendo el nombre de las columnas de la tabla de la base de datos asociada como el nombre de sus atributos. Hay que tener en cuenta que también puede necesitar sobrescribir los métodos mágicos como `__get()`, `__set()` de modo que se puede acceder a los atributos como a propiedades de objetos normales.

Etiquetas de atributo

Cuando se muestran valores o se obtienen entradas para atributos, normalmente se necesita mostrar etiquetas asociadas a los atributos. Por ejemplo, dado un atributo con nombre 'segundoApellido', es posible que se quiera mostrar la etiqueta 'Segundo Apellido' ya que es más fácil de interpretar por el usuario final en lugares como campos de formularios y en mensajes de error.

Se puede obtener la etiqueta de un atributo llamando a `yii\base\Model::getAttributeLabel()`. Por ejemplo:

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

// muestra "Name"
echo $model->getAttributeLabel('name');
```

Por defecto, una etiqueta de atributo se genera automáticamente a partir del nombre de atributo. La generación se hace con el método `yii\base\Model::generateAttributeLabel()`. Este convertirá los nombres de variables de tipo camel-case en múltiples palabras con la primera letra de cada palabra en mayúsculas. Por ejemplo 'usuario' se convertirá en 'Nombre', y 'primerApellido' se convertirá en 'Primer Apellido'. Si no se quieren usar las etiquetas generadas automáticamente, se puede sobrescribir `yii\base\Model::attributeLabels()` a una declaración de etiquetas de atributo especifica. Por ejemplo:

```php
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;

    public function attributeLabels()
    {
        return [
            'name' => 'Your name',
            'email' => 'Your email address',
            'subject' => 'Subject',
            'body' => 'Content',
        ];
    }
}
```

Para aplicaciones con soporte para múltiples idiomas, se puede querer traducir las etiquetas de los atributos. Esto se puede hacer en el método `attributeLabels()`, como en el siguiente ejemplo:

```php
public function attributeLabels()
{
    return [
        'name' => \Yii::t('app', 'Your name'),
        'email' => \Yii::t('app', 'Your email address'),
        'subject' => \Yii::t('app', 'Subject'),
        'body' => \Yii::t('app', 'Content'),
    ];
}
```

Incluso se puede definir etiquetas de atributo condicionales. Por ejemplo, basándose en el escenario en que se esta usando el modelo, se pueden devolver diferentes etiquetas para un mismo atributo.

> **Información:** Estrictamente hablando, los atributos son parte de las vistas. Pero declarar las etiquetas en los modelos, a menudo, es muy conveniente y puede generar a un código muy limpio y reutilizable.

3.6.1. Escenarios

Un modelo puede usarse en diferentes *escenarios*. Por ejemplo, un modelo 'Usuario', puede ser utilizado para recoger entradas de inicio de sesión de usuarios, pero también puede usarse para generar usuarios. En diferentes escenarios, un modelo puede usar diferentes reglas de negocio y lógica. Por ejemplo, un atributo 'email' puede ser requerido durante un registro de usuario, pero no ser necesario durante el inicio de sesión del mismo.

Un modelo utiliza la propiedad `yii\base\Model::$scenario` para mantener saber en qué escenario se esta usando. Por defecto, un modelo soporta sólo un escenario llamado 'default'. El siguiente código muestra dos maneras de establecer el escenario en un modelo.

```
// el escenario se establece como una propiedad
$model = new User;
$model->scenario = 'login';

// el escenario se establece mediante configuración
$model = new User(['scenario' => 'login']);
```

Por defecto, los escenarios soportados por un modelo se determinan por las reglas de validación declaradas en el modelo. Sin embargo, se puede personalizar este comportamiento sobrescribiendo el método `yii\base\Model::scenarios()`, como en el siguiente ejemplo:

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    public function scenarios()
    {
        return [
            'login' => ['username', 'password'],
            'register' => ['username', 'email', 'password'],
        ];
    }
}
```

> **Información:** En el anterior y en los siguientes ejemplos, las clases modelo extienden a `yii\db\ActiveRecord` porque el uso de múltiples escenarios normalmente sucede con clases de Active Records.

El método 'scenarios()' devuelve un array cuyas claves son el nombre de escenario y los valores correspondientes a los *atributos activos*. Un atributo activo puede ser asignado masivamente y esta sujeto a validación. En el anterior ejemplo, los atributos 'username' y 'password' están activados en el escenario 'login'; mientras que en el escenario 'register', el atributo 'email' esta activado junto con 'username' y 'password'.

La implementación por defecto de los 'scenarios()' devolverá todos los escenarios encontrados en el método de declaración de las reglas de validación `yii\base\Model::rules()`. Cuando se sobrescribe 'scenarios()', si se quiere introducir nuevos escenarios además de los predeterminados, se puede hacer como en el siguiente ejemplo:

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    public function scenarios()
    {
        $scenarios = parent::scenarios();
        $scenarios['login'] = ['username', 'password'];
        $scenarios['register'] = ['username', 'email', 'password'];
        return $scenarios;
    }
}
```

La característica escenario se usa principalmente en las validaciones y por la asignación masiva de atributos. Aunque también se puede usar para otros propósitos. Por ejemplo, se pueden declarar etiquetas de atributo diferentes basándose en el escenario actual.

3.6.2. Reglas de Validación

Cuando un modelo recibe datos del usuario final, estos deben ser validados para asegurar que cumplan ciertas reglas (llamadas *reglas de validación*, también conocidas como *reglas de negocio*). Por ejemplo, dado un modelo 'ContactForm', se puede querer asegurar que ningún atributo este vacío y que el atributo 'email' contenga una dirección de correo válida. Si algún valor no cumple con las reglas, se debe mostrar el mensaje de error apropiado para ayudar al usuario a corregir estos errores.

Se puede llamar a `yii\base\Model::validate()` para validar los datos recibidos. El método se usará para validar las reglas declaradas en `yii\base`

`\Model::rules()` para validar cada atributo relevante. Si no se encuentran errores, se devolverá `true`. De otro modo, este almacenará los errores en la propiedad `yii\base\Model::$errors` y devolverá falso. Por ejemplo:

```
$model = new \app\models\ContactForm;

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

if ($model->validate()) {
    // todas las entradas validadas
} else {
    // validación fallida: $errors es un array que contiene los mensajes de
    error
    $errors = $model->errors;
}
```

Para declarar reglas de validación asociadas a un modelo, se tiene que sobrescribir el método `yii\base\Model::rules()` para que devuelva las reglas que los atributos del modelo deben satisfacer. El siguiente ejemplo muestra las reglas de validación declaradas para el modelo 'ContactForm'.

```
public function rules()
{
    return [
        // name, email, subject y body son atributos requeridos
        [['name', 'email', 'subject', 'body'], 'required'],

        // el atribuido email debe ser una dirección de correo electrónico
        válida
        ['email', 'email'],
    ];
}
```

Una regla puede usarse para validar uno o más atributos, y un atributo puede validarse por una o múltiples reglas. Por favor refiérase a la sección Validación de entrada para obtener más detalles sobre cómo declarar reglas de validación.

A veces, solamente se quiere aplicar una regla en ciertos escenarios. Para hacerlo, se puede especificar la propiedad 'on' de una regla, como en el siguiente ejemplo:

```
public function rules()
{
    return [
        // username, email y password son obligatorios en el escenario
        "register"
        [['username', 'email', 'password'], 'required', 'on' => 'register'],

        // username y password son obligatorios en el escenario "login"
        [['username', 'password'], 'required', 'on' => 'login'],
```

```
    ];
}
```

Si no se especifica la propiedad 'on', la regla se aplicará en todos los escenarios. Se llama a una regla *regla activa* si esta puede aplicarse en el `scenario` actual.

Un atributo será validado si y sólo si es un atributo activo declarado en 'scenarios()' y esta asociado con una o más reglas activas declaradas en 'rules()'.

3.6.3. Asignación Masiva

La asignación masiva es una buena forma de rellenar los atributos de un modelo con las entradas de usuario en una única línea de código. Rellena los atributos de un modelo asignando los datos de entrada directamente a las propiedades de `yii\base\Model::$attributes`. Los siguientes dos ejemplos son equivalentes, ambos intentan asignar los datos enviados por el usuario final a través de un formulario a los atributos del modelo 'ContactForm'. Claramente, el primero, que usa la asignación masiva, es más claro y menos propenso a errores que el segundo:

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

```
$model = new \app\models\ContactForm;
$data = \Yii::$app->request->post('ContactForm', []);
$model->name = isset($data['name']) ? $data['name'] : null;
$model->email = isset($data['email']) ? $data['email'] : null;
$model->subject = isset($data['subject']) ? $data['subject'] : null;
$model->body = isset($data['body']) ? $data['body'] : null;
```

Atributos Seguros

La asignación masiva sólo se aplica a los llamados *atributos seguros* qué son los atributos listados en `yii\base\Model::scenarios()` para el actual `scenario` del modelo. Por ejemplo, si en el modelo 'User' tenemos la siguiente declaración de escenario, entonces cuando el escenario actual sea 'login', sólo los atributos 'username' y 'password' podrán ser asignados masivamente. Cualquier otro atributo permanecerá intacto

```
public function scenarios()
{
    return [
        'login' => ['username', 'password'],
        'register' => ['username', 'email', 'password'],
    ];
}
```

> **Información:** La razón de que la asignación masiva sólo se aplique a los atributos seguros es debida a que se quiere controlar qué atributos pueden ser modificados por los datos del usuario final. Por ejemplo, si el modelo 'User' tiene un atributo 'permission' que determina los permisos asignados al usuario, se quiere que estos atributos sólo sean modificados por administradores desde la interfaz backend.

Debido a que la implementación predeterminada de `yii\base\Model::scenarios()` devolverá todos los escenarios y atributos encontrados en `yii\base\Model::rules()`, si no se sobrescribe este método, significa que un atributo es seguro mientras aparezca en una de las reglas de validación activas.

Por esta razón, se proporciona un validador especial con alias 'safe' con el que se puede declarar un atributo como seguro sin llegar a validarlo. Por ejemplo, las siguientes reglas declaran que los atributos 'title' y 'description' son atributos seguros.

```
public function rules()
{
    return [
        [['title', 'description'], 'safe'],
    ];
}
```

Atributos Inseguros

Como se ha descrito anteriormente, el método `yii\base\Model::scenarios()` sirve para dos propósitos: determinar qué atributos deben ser validados y determinar qué atributos son seguros. En situaciones poco comunes, se puede querer validar un atributo pero sin marcarlo como seguro. Se puede hacer prefijando el signo de exclamación '!' delante del nombre del atributo cuando se declaran en 'scenarios()', como el atributo 'secret' del siguiente ejemplo:

```
public function scenarios()
{
    return [
        'login' => ['username', 'password', '!secret'],
    ];
}
```

Cuando el modelo esté en el escenario 'login', los tres atributos serán validados. Sin embargo, sólo los atributos 'username' y 'password' se asignarán masivamente. Para asignar un valor de entrada al atribuido 'secret', se tendrá que hacer explícitamente como en el ejemplo:

```
$model->secret = $secret;
```

3.6.4. Exportación de Datos

A menudo necesitamos exportar modelos a diferentes formatos. Por ejemplo, se puede querer convertir un conjunto de modelos a formato JSON o Excel. El proceso de exportación se puede dividir en dos pasos independientes. En el primer paso, se convierten los modelos en arrays; en el segundo paso, los arrays se convierten a los formatos deseados. Nos puede interesar fijarnos en el primer paso, ya que el segundo paso se puede lograr mediante un formateador de datos genérico, tal como `yii\web\JsonResponseFormatter`. La manera más simple de convertir un modelo en un array es usar la propiedad `yii\base\Model::$attributes`. Por ejemplo:

```
$post = \app\models\Post::findOne(100);
$array = $post->attributes;
```

Por defecto, la propiedad `yii\base\Model::$attributes` devolverá los valores de *todos* los atributos declarados en `yii\base\Model::attributes()`.

Una manera más flexible y potente de convertir un modelo en un array es usar el método `yii\base\Model::toArray()`. Su funcionamiento general es el mismo que el de `yii\base\Model::$attributes`. Sin embargo, este permite elegir que elementos de datos, llamados *campos*, queremos poner en el array resultante y elegir como debe ser formateado. De hecho, es la manera por defecto de exportar modelos en desarrollo de servicios Web RESTful, tal y como se describe en Formatos de Respuesta.

Campos

Un campo es simplemente un elemento nombrado en el array resultante de ejecutar el método `yii\base\Model::toArray()` de un modelo. Por defecto, los nombres de los campos son equivalentes a los nombres de los atributos. Sin embargo, se puede modificar este comportamiento sobrescribiendo el método `fields()` y/o el método `extraFields()`. Ambos métodos deben devolver una lista de las definiciones de los campos. Los campos definidos mediante 'fields()' son los campos por defecto, esto significa que 'toArray()' devolverá estos campos por defecto. El método 'extraFields()' define campos adicionalmente disponibles que también pueden devolverse mediante 'toArray()' siempre y cuando se especifiquen a través del parámetro '$expand'. Por ejemplo, el siguiente código devolverá todos los campos definidos en 'fields()' y los campos 'prettyName' y 'fullAdress' si estos están definidos en 'extraFields()'.

```
$array = $model->toArray([], ['prettyName', 'fullAddress']);
```

Se puede sobrescribir 'fields()' para añadir, eliminar, renombrar o redefinir campos. El valor devuelto por 'fields()' debe se un array. Las claves del array son los nombres de los campos, y los valores son las correspondientes

definiciones de los campos que pueden ser nombres de propiedades/atributos o funciones anónimas que devuelvan los correspondientes valores de campo. En el caso especial en que un nombre de un campo es el mismo a su definición de nombre de atributo, se puede omitir la clave del array. Por ejemplo:

```
// lista explícitamente cada campo, es mejor usarlo cuando nos queremos
asegurar
// de que los cambios en la tabla de la base de datos o los atributos del
modelo
// no modifiquen los campos(para asegurar compatibilidades para versiones
anteriores de API)
public function fields()
{
    return [
        // el nombre del campo es el mismo que el nombre de atributo
        'id',

        // el nombre del campo es "email", el nombre de atributo
        correspondiente es "email_address"
        'email' => 'email_address',

        // El nombre del campo es "name", su valor esta definido por una
        llamada de retorno PHP
        'name' => function () {
            return $this->first_name . ' ' . $this->last_name;
        },
    ];
}

// filtrar algunos campos, es mejor usarlo cuando se quiere heredar la
implementación del padre
// y discriminar algunos campos sensibles.
public function fields()
{
    $fields = parent::fields();

    // elimina campos que contengan información sensible.
    unset($fields['auth_key'], $fields['password_hash'],
    $fields['password_reset_token']);

    return $fields;
}
```

Aviso: debido a que por defecto todos los atributos de un modelo serán incluidos en el array exportado, se debe examinar los datos para asegurar que no contienen información sensible. Si existe dicha información, se debe sobrescribir 'fields()' para filtrarla. En el anterior ejemplo, se filtra 'aut_key', 'password_hash' y 'password_reset_token'.

3.6.5. Mejores Prácticas

Los modelos son los lugares centrales para representar datos de negocio, reglas y lógica. Estos a menudo necesitan ser reutilizados en diferentes lugares. En una aplicación bien diseñada, los modelos normalmente son más grandes que los controladores.

En resumen, los modelos:

- pueden contener atributos para representar los datos de negocio;
- pueden contener reglas de validación para asegurar la validez e integridad de los datos;
- pueden contener métodos que para implementar la lógica de negocio;
- NO deben acceder directamente a peticiones, sesiones, u otro tipo de datos de entorno. Estos datos deben ser inyectados por los controladores en los modelos.
- deben evitar embeber HTML u otro código de presentación – esto es mejor hacerlo en las vistas;
- evitar tener demasiados escenarios en un mismo modelo.

Generalmente se puede considerar la última recomendación cuando se estén desarrollando grandes sistemas complejos. En estos sistemas, los modelos podrían ser muy grandes debido a que podrían ser usados en muchos lugares y por tanto contener muchos conjuntos de reglas y lógicas de negocio. A menudo esto desemboca en un código muy difícil de mantener ya que una simple modificación en el código puede afectar a muchos sitios diferentes. Para mantener el código más fácil de mantener, se puede seguir la siguiente estrategia:

- Definir un conjunto de clases modelo base que sean compartidas por diferentes aplicaciones o módulos. Estas clases modelo deben contener el conjunto mínimo de reglas y lógica que sean comunes para todos sus usos.
- En cada aplicación o módulo que use un modelo, definir una clase modelo concreta que extienda a la correspondiente clase modelo base. La clase modelo concreta debe contener reglas y lógica que sean específicas para esa aplicación o módulo.

Por ejemplo, en la Plantilla de Aplicación Avanzada, definiendo una clase modelo base 'common\models\Post'. Después en la aplicación front end, definiendo y usando una clase modelo concreta 'frontend\models\Post' que extienda a 'common\models\Post'. Y de forma similar en la aplicación back end, definiendo 'backend\models\Post'. Con esta estrategia, nos aseguramos que el código de 'frontend\models\Post' es específico para la aplicación front end, y si se efectúa algún cambio en el, no nos tenemos que preocupar de si el cambio afectará a la aplicación back end.

3.7. Vistas

Las Vistas (views) son una parte de la arquitectura MVC[12]. Estas son el código responsable de presentar los datos al usuario final. En una aplicación Web, las vistas son usualmente creadas en términos de *templates* que son archivos PHP que contienen principalmente HTML y PHP. Estas son manejadas por el componente de la aplicación `view`, el cual provee los métodos comúnmente utilizados para facilitar la composición y renderizado. Por simplicidad, a menudo nos referimos a los templates de vistas o archivos de templates como vistas.

3.7.1. Crear Vistas

Como fue mencionado, una vista es simplemente un archivo PHP que mezcla código PHP y HTML. La siguiente es una vista que muestra un formulario de login. Como puedes ver, el código PHP utilizado es para generar contenido dinámico, como el título de la página y el formulario mismo, mientras que el código HTML organiza estos elementos en una página HTML mostrable.

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $form yii\widgets\ActiveForm */
/* @var $model app\models\LoginForm */

$this->title = 'Login';
?>
<h1><?= Html::encode($this->title) ?></h1>

<p>Por favor completa los siguientes campos para loguearte:</p>

<?php $form = ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('Login') ?>
<?php ActiveForm::end(); ?>
```

Dentro de una vista, puedes acceder a la variable `$this` referida al `componente view` que maneja y renderiza la vista actual.

Además de `$this`, puede haber otras variables predefinidas en una vista, como `$form` y `$model` en el ejemplo anterior. Estas variables representan los datos que son *inyectados* a la vista desde el controlador o algún otro objeto que dispara la renderización de la vista.

[12] https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador

> **Consejo:** La lista de variables predefinidas están listadas en un bloque de comentario al principio de la vista así pueden ser reconocidas por las IDEs. Esto es también una buena manera de documentar tus propias vistas.

Seguridad

Al crear vistas que generan páginas HTML, es importante que codifiques (encode) y/o filtres los datos provenientes de los usuarios antes de mostrarlos. De otro modo, tu aplicación puede estar expuesta a ataques tipo cross-site scripting[13].

Para mostrar un texto plano, codifícalos previamente utilizando `yii\helpers\Html::encode()`. Por ejemplo, el siguiente código aplica una codificación del nombre de usuario antes de mostrarlo:

```
<?php
use yii\helpers\Html;
?>

<div class="username">
    <?= Html::encode($user->name) ?>
</div>
```

Para mostrar contenido HTML, utiliza `yii\helpers\HtmlPurifier` para filtrarlo antes. Por ejemplo, el siguiente código filtra el contenido del post antes de mostrarlo en pantalla:

```
<?php
use yii\helpers\HtmlPurifier;
?>

<div class="post">
    <?= HtmlPurifier::process($post->text) ?>
</div>
```

> **Consejo:** Aunque HTMLPurifier hace un excelente trabajo al hacer la salida más segura, no es rápido. Deberías considerar el aplicar un caching al resultado de aplicar el filtro si tu aplicación requiere un gran desempeño (performance).

Organizar las Vistas

Así como en controladores y modelos, existen convenciones para organizar las vistas.

- Para vistas renderizadas por controladores, deberían colocarse en un directorio tipo `@app/views/ControllerID` por defecto, donde `ControllerID` se

[13] https://es.wikipedia.org/wiki/Cross-site_scripting

refiere al ID del controlador. Por ejemplo, si la clase del controlador es `PostController`, el directorio sería `@app/views/post`; Si fuera `PostCommentController`, el directorio sería `@app/views/post-comment`. En caso de que el controlador pertenezca a un módulo, el directorio sería `views/ControllerID` bajo el `directorio del módulo`.
- Para vistas renderizadas por un widget, deberían ser puestas en un directorio tipo `WidgetPath/views` por defecto, donde `WidgetPath` se refiere al directorio que contiene a la clase del widget.
- Para vistas renderizadas por otros objetos, se recomienda seguir una convención similar a la utilizada con los widgets.

Puedes personalizar estos directorios por defecto sobrescribiendo el método `yii\base\ViewContextInterface::getViewPath()` en el controlador o widget necesario.

3.7.2. Renderizando Vistas

Puedes renderizar vistas desde controllers, widgets, o cualquier otro lugar llamando a los métodos de renderización de vistas. Estos métodos comparten una firma similar, como se muestra a continuación:

```
/**
 * @param string $view nombre de la vista o ruta al archivo, dependiendo del
 método de renderización utilizado
 * @param array $params los datos pasados a la vista
 * @return string el resultado de la renderización
 */
methodName($view, $params = [])
```

Renderizando en Controladores

Dentro de los controladores, puedes llamar al siguiente método del controlador para renderizar una vista:

- `render()`: renderiza la vista nombrada y aplica un layout al resultado de la renderización.
- `renderPartial()`: renderiza la vista nombrada sin ningún layout aplicado.
- `renderAjax()`: renderiza la vista nombrada sin layout, e inyecta todos los scripts y archivos JS/CSS registrados. Esto sucede usualmente en respuestas a peticiones AJAX.
- `renderFile()`: renderiza la vista especificada en términos de la ruta al archivo o alias.
- `renderContent()`: renderiza un string fijo, inscrustándolo en el layout actualmente aplicable. Este método está disponible desde la versión 2.0.1.

Por ejemplo:

```php
namespace app\controllers;

use Yii;
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

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

        // renderiza una vista llamada "view" y le aplica el layout
        return $this->render('view', [
            'model' => $model,
        ]);
    }
}
```

Renderizando en Widgets

Dentro de widgets, puedes llamar a cualquier de los siguientes métodos de widget para renderizar una vista.

- `render()`: renderiza la vista nombrada.
- `renderFile()`: renderiza la vista especificada en términos de ruta al archivo o alias.

Por ejemplo:

```php
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class ListWidget extends Widget
{
    public $items = [];

    public function run()
    {
        // renderiza una vista llamada "list"
        return $this->render('list', [
            'items' => $this->items,
        ]);
    }
}
```

Renderizar en Vistas

Puedes renderizar una vista dentro de otra vista llamando a algunos de los siguientes métodos provistos por el `componente view`:

- `render()`: renderiza la vista nombrada.
- `renderAjax()`: renderiza la vista nombrada e inyecta todos los archivos y scripts JS/CSS. Esto sucede usualmente en respuestas a las peticiones AJAX.
- `renderFile()`: renderiza la vista especificada en términos de ruta al archivo o alias.

Por ejemplo, el siguiente código en una vista renderiza el template `_overview.php` encontrado en el mismo directorio de la vista renderizada actualmente. Recuerda que la variable `$this` en una vista se refiere al componente `view`:

```
<?= $this->render('_overview') ?>
```

Renderizar en Otros Lugares

En cualquier lugar, puedes tener acceso al componente `view` utilizando la expresión `Yii::$app->view` y entonces llamar a los métodos previamente mencionados para renderizar una vista. Por ejemplo:

```
// muestra el template "@app/views/site/license.php"
echo \Yii::$app->view->renderFile('@app/views/site/license.php');
```

Vistas Nombradas

Cuando renderizas una vista, puedes especificar el template utilizando tanto el nombre de la vista o la ruta/alias al archivo. En la mayoría de los casos, utilizarías la primera porque es más concisa y flexible. Las *vistas nombradas* son vistas especificadas mediante un nombre en vez de una ruta al archivo o alias.

Un nombre de vista es resuelto a su correspondiente ruta de archivo siguiendo las siguientes reglas:

- Un nombre de vista puede omitir la extensión del archivo. En estos casos se utilizará `.php` como extensión del archivo. Por ejemplo, el nombre de vista `about` corresponde al archivo `about.php`.
- Si el nombre de la vista comienza con doble barra (`//`), la ruta al archivo correspondiente será `@app/views/ViewName`. Esto quiere decir que la vista es buscada bajo el `ruta de vistas de la aplicación`. Por ejemplo, `//site/about` será resuelto como `@app/views/site/about.php`.
- Si el nombre de la vista comienza con una barra simple `/`, la ruta al archivo de la vista utilizará como prefijo el nombre de la vista con el `view path` del módulo utilizado actualmente. Si no hubiera módulo activo se utilizará `@app/views/ViewName`. Por ejemplo, `/user/create` será

resuelto como `@app/modules/user/views/user/create.php` si el módulo activo es `user`. Si no hubiera módulo activo, la ruta al archivo será `@app/views/user/create.php`.

- Si la vista es renderizada con un `context` y dicho contexto implementa `yii\base\ViewContextInterface`, la ruta al archivo se forma utilizando como prefijo la `ruta de vistas` del contexto de la vista. Esto principalmente aplica a vistas renderizadas en controladores y widgets. Por ejemplo, `about` será resuelto como `@app/views/site/about.php` si el contexto es el controlador `SiteController`.
- Si la vista es renderizada dentro de otra vista, el directorio que contiene la otra vista será prefijado al nuevo nombre de la vista para formar la ruta a la vista. Por ejemplo, `item` sera resuelto como `@app/views/post/item` si está siendo renderizado desde la vista `@app/views/post/index.php`.

De acuerdo a las reglas mencionadas, al llamar a `$this->render('view')` en el controlador `app\controllers\PostController` se renderizará el template `@app/views/post/view.php`, mientras que llamando a `$this->render('_overview')` en la vista renderizará el template `@app/views/post/_overview.php`.

Acceder a Datos en la Vista

Hay dos modos posibles de acceder a los datos en la vista: push (inyectar) y pull (traer).

Al pasar los datos como segundo parámetro en algún método de renderización, estás utilizando el modo push. Los datos deberían ser representados como un array de pares clave-valor. Cuando la vista está siendo renderizada, la función PHP `extract()` será llamada sobre este array así se extraen las variables que contiene a la vista actual. Por ejemplo, el siguiente código de renderización en un controlador inyectará dos variables a la vista `report`: `$foo = 1` y `$bar = 2`.

```
echo $this->render('report', [
    'foo' => 1,
    'bar' => 2,
]);
```

El modo pull obtiene los datos del `componente view` u otros objetos accesibles en las vistas (ej. `Yii::$app`). Utilizando el código anterior como ejemplo, dentro de una vista puedes acceder al objeto del controlador a través de la expresión `$this->context`. Como resultado, te es posible acceder a cualquier propiedad o método del controlador en la vista `report`, tal como el ID del controlador como se muestra a continuación:

```
El ID del controlador es: <?= $this->context->id ?>
```

Para acceder a datos en la vista, normalmente se prefiere el modo push, ya que hace a la vista menos dependiente de los objetos del contexto. La

contra es que tienes que construir el array manualmente cada vez, lo que podría volverse tedioso y propenso al error si la misma vista es compartida y renderizada desde diferentes lugares.

Compartir Datos Entre las Vistas

El `componente view` provee la propiedad `params` para que puedas compartir datos entre diferentes vistas.

Por ejemplo, en una vista `about`, podrías tener el siguiente código que especifica el segmento actual del breadcrumbs (migas de pan).

```
$this->params['breadcrumbs'][] = 'Acerca de Nosotros';
```

Entonces, en el archivo del layout, que es también una vista, puedes mostrar el breadcrumbs utilizando los datos pasados a través de `params`:

```
<?= yii\widgets\Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ?
    $this->params['breadcrumbs'] : [],
]) ?>
```

3.7.3. Layouts

Los layouts son un tipo especial de vista que representan partes comunes de otras múltiples vistas. Por ejemplo, las páginas de la mayoría de las aplicaciones Web comparten el mismo encabezado y pie de página. Aunque puedes repetirlos en todas y cada una de las vistas, una mejor forma es hacerlo sólo en el layout e incrustar el resultado de la renderización de la vista en un lugar apropiado del mismo.

Crear Layouts

Dado que los layouts son también vistas, pueden ser creados de manera similar a las vistas comunes. Por defecto, los layouts son guardados en el directorio `@app/views/layouts`. Para layouts utilizados dentro de un módulo, deberían ser guardados en el directorio `views/layouts` bajo el `directorio del módulo`. Puedes personalizar el directorio de layouts por defecto configurando la propiedad `yii\base\Module::$layoutPath` de la aplicación o módulos.

El siguiente ejemplo muestra cómo debe verse un layout. Ten en cuenta que por motivos ilustrativos, hemos simplificado bastante el código del layout. En la práctica, probablemente le agregues más contenido, como tags en el `head`, un menú principal, etc.

```
<?php
use yii\helpers\Html;

/* @var $this yii\web\View */
```

```
/* @var $content string */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <?= Html::csrfMetaTags() ?>
    <title><?= Html::encode($this->title) ?></title>
    <?php $this->head() ?>
</head>
<body>
<?php $this->beginBody() ?>
    <header>Mi Compañía</header>
    <?= $content ?>
    <footer>&copy; 2014 - Mi Compañía</footer>
<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

Como puedes ver, el layout genera los tags HTML comunes a todas las páginas. Dentro de la sección `<body>`, el layout imprime la variable `$content`, que representa el resultado de la renderización del contenido de cada vista y es incrustado dentro del layout cuando se llama al método `yii\base\Controller::render()`.

La mayoría de layouts deberían llamar a los siguientes métodos (como fue mostrado recién). Estos métodos principalmente disparan eventos acerca del proceso de renderizado así los scripts y tags registrados en otros lugares pueden ser propiamente inyectados en los lugares donde los métodos son llamados.

- `beginPage()`: Este método debería ser llamado bien al principio del layout. Esto dispara el evento `EVENT_BEGIN_PAGE`, el cual indica el comienzo de la página.
- `endPage()`: Este método debería ser llamado al final del layout. Esto dispara el evento `EVENT_END_PAGE`, indicando el final de la página.
- `head()`: Este método debería llamarse dentro de la sección `<head>` de una página HTML. Esto genera un espacio vacío que será reemplazado con el código del head HTML registrado (ej. link tags, meta tags) cuando una página finaliza el renderizado.
- `yii\base\View::beginBody()`: Este método debería llamarse al principio de la sección `<body>`. Esto dispara el evento `EVENT_BEGIN_BODY` y genera un espacio vacío que será reemplazado con el código HTML registrado (ej. JavaScript) que apunta al principio del body.
- `yii\base\View::endBody()`: Este método debería llamarse al final de la sección `<body>`. Esto dispara el evento `EVENT_END_BODY`, que genera un espacio vacío a ser reemplazado por el código HTML registrado (ej. JavaScript) que apunta al final del body.

Acceder a Datos en Layouts

Dentro de un layout, tienes acceso a dos variables predefinidas: `$this` y `$content`. La primera se refiere al componente `view`, como en cualquier vista, mientras que la última contiene el resultado de la renderización del contenido de la vista que está siendo renderizada al llamar al método `render()` en los controladores.

Si quieres acceder a otros datos en los layouts, debes utilizar el modo pull que fue descrito en la sub-sección Accediendo a Datos en la Vista. Si quieres pasar datos desde al contenido de la vista a un layout, puedes utilizar el método descrito en la sub-sección Compartiendo Datos Entre las Vistas.

Utilizar Layouts

Como se describe en la sub-sección Renderizando en Controllers, cuando renderizas una vista llamando al método `render()` en un controlador, al resultado de dicha renderización le será aplicado un layout. Por defecto, el layout `@app/views/layouts/main.php` será el utilizado.

Puedes utilizar un layout diferente configurando la propiedad `yii\base\Application::$layout` o `yii\base\Controller::$layout`. El primero se refiere al layout utilizado por todos los controladores, mientras que el último sobrescribe el layout en controladores individuales. Por ejemplo, el siguiente código hace que el controlador `post` utilice `@app/views/layouts/post.php` como layout al renderizar sus vistas. Otros controladores, asumiendo que su propiedad `layout` no fue modificada, utilizarán `@app/views/layouts/main.php` como layout.

```
namespace app\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    public $layout = 'post';

    // ...
}
```

Para controladores que pertencen a un módulo, puedes también configurar la propiedad `layout` y así utilizar un layout en particular para esos controladores.

Dado que la propiedad `layout` puede ser configurada en diferentes niveles (controladores, módulos, aplicación), detrás de escena Yii realiza dos pasos para determinar cuál es el archivo de layout siendo utilizado para un controlador en particular.

En el primer paso, determina el valor del layout y el módulo de contexto:

- Si la propiedad `yii\base\Controller::$layout` no es `null`, la utiliza como valor del layout y el `módulo` del controlador como el módulo de contexto.
- Si `layout` es `null`, busca a través de todos los módulos ancestros del controlador y encuentra el primer módulo cuya propiedad `layout` no es `null`. Utiliza ese módulo y su valor de `layout` como módulo de contexto y como layout seleccionado. Si tal módulo no puede ser encontrado, significa que no se aplicará ningún layout.

En el segundo paso, se determina el archivo de layout actual de acuerdo al valor de layout y el módulo de contexto determinado en el primer paso. El valor de layout puede ser:

- un alias de ruta (ej. `@app/views/layouts/main`).
- una ruta absoluta (ej. `/main`): el valor del layout comienza con una barra. El archivo de layout actual será buscado bajo el `layout path` de la aplicación, que es por defecto `@app/views/layouts`.
- una ruta relativa (ej. `main`): El archivo de layout actual será buscado bajo el `layout path` del módulo de contexto, que es por defecto el directorio `views/layouts` bajo el `directorio del módulo`.
- el valor booleano `false`: no se aplicará ningún layout.

Si el valor de layout no contiene una extensión de tipo de archivo, utilizará por defecto `.php`.

Layouts Anidados

A veces podrías querer anidar un layout dentro de otro. Por ejemplo, en diferentes secciones de un sitio Web, podrías querer utilizar layouts diferentes, mientras que todos esos layouts comparten el mismo layout básico que genera la estructura general de la página en HTML5. Esto es posible llamando a los métodos `beginContent()` y `endContent()` en los layouts hijos como se muestra a continuación:

```
<?php $this->beginContent('@app/views/layouts/base.php'); ?>

...contenido del layout hijo aquí...

<?php $this->endContent(); ?>
```

Como se acaba de mostrar, el contenido del layout hijo debe ser encerrado dentro de `beginContent()` y `endContent()`. El parámetro pasado a `beginContent()` especifica cuál es el módulo padre. Este puede ser tanto un archivo layout como un alias.

Utilizando la forma recién mencionada, puedes anidar layouts en más de un nivel.

Utilizar Blocks

Los bloques te permiten especificar el contenido de la vista en un lugar y mostrarlo en otro. Estos son a menudo utilizados junto a los layouts. Por ejemplo, puedes definir un bloque un una vista de contenido y mostrarla en el layout.

Para definir un bloque, llamas a `beginBlock()` y `endBlock()`. El bloque puede ser accedido vía `$view->blocks[$blockID]`, donde `$blockID` se refiere al ID único que le asignas al bloque cuando lo defines.

El siguiente ejemplo muestra cómo utilizar bloques para personalizar partes especificas del layout in una vista.

Primero, en una vista, define uno o varios bloques:

```
...

<?php $this->beginBlock('block1'); ?>

...contenido de block1...

<?php $this->endBlock(); ?>

...

<?php $this->beginBlock('block3'); ?>

...contenido de block3...

<?php $this->endBlock(); ?>
```

Entonces, en la vista del layout, renderiza los bloques si están disponibles, o muestra un contenido por defecto si el bloque no está definido.

```
...
<?php if (isset($this->blocks['block1'])): ?>
    <?= $this->blocks['block1'] ?>
<?php else: ?>
    ... contenido por defecto de block1 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block2'])): ?>
    <?= $this->blocks['block2'] ?>
<?php else: ?>
    ... contenido por defecto de block2 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block3'])): ?>
    <?= $this->blocks['block3'] ?>
```

```
<?php else: ?>
    ... contenido por defecto de block3 ...
<?php endif; ?>
...
```

3.7.4. Utilizar Componentes de Vista

Los componentes de vista proveen características relacionadas a las vistas. Aunque puedes obtener componentes de vista creando instancias individuales de `yii\base\View` o sus clases hijas, en la mayoría de los casos utilizarías el componente `view` del a aplicación. Puedes configurar este componente en la configuración de la aplicación como a continuación:

```
[
    // ...
    'components' => [
        'view' => [
            'class' => 'app\components\View',
        ],
        // ...
    ],
]
```

Los componentes de vista proveen las siguientes características útiles, cada una descrita en mayor detalle en su propia sección:

- temas: te permite desarrollar y cambiar el tema (theme) de tu sitio Web.
- caché de fragmentos: te permite guardar en cache un fragmento de una página Web.
- manejo de scripts del cliente: soporte para registro y renderización de CSS y JavaScript.
- manejo de asset bundle: soporte de registro y renderización de asset bundles.
- motores de template alternativos: te permite utilizar otros motores de templates, como Twig[14] o Smarty[15].

Puedes también utilizar frecuentemente el siguiente menor pero útil grupo de características al desarrollar páginas Web.

Definiendo Títulos de Página

Toda página Web debería tener un título. Normalmente el tag de título es generado en layout. De todos modos, en la práctica el título es determinado en el contenido de las vistas más que en layouts. Para resolver este problema, `yii\web\View` provee la propiedad `title` para que puedas pasar información del título desde el contenido de la vista a los layouts.

[14] https://twig.symfony.com/
[15] https://www.smarty.net/

Para utilizar esta característica, en cada contenido de la vista, puedes definir el título de la siguiente manera:

```
<?php
$this->title = 'Título de mi página';
?>
```

Entonces en el layout, asegúrate de tener el siguiente código en la sección `<head>` de la página:

```
<title><?= Html::encode($this->title) ?></title>
```

Registrar Meta Tags

Las páginas Web usualmente necesitan generar varios meta tags necesarios para diferentes grupos. Cómo los títulos de página, los meta tags aparecen en la sección `<head>` y son usualmente generado en los layouts.

Si quieres especificar cuáles meta tags generar en las vistas, puedes llamar a `yii\web\View::registerMetaTag()` dentro de una de ellas, como se muestra a continuación:

```
<?php
$this->registerMetaTag(['name' => 'keywords', 'content' => 'yii, framework,
php']);
?>
```

El código anterior registrará el meta tag "keywords" a través del componente view. El meta tag registrado no se renderiza hasta que finaliza el renderizado del layout. Para entonces, el siguiente código HTML será insertado en el lugar donde llamas a `yii\web\View::head()` en el layout, generando el siguiente HTML:

```
<meta name="keywords" content="yii, framework, php">
```

Ten en cuenta que si llamas a `yii\web\View::registerMetaTag()` varias veces, esto registrará varios meta tags, sin tener en cuenta si los meta tags son los mismo o no.

Para asegurarte de que sólo haya una instancia de cierto tipo de meta tag, puedes especificar una clave al llamar al método. Por ejemplo, el siguiente código registra dos meta tags "description", aunque sólo el segundo será renderizado.

```
$this->registerMetaTag(['name' => 'description', 'content' => 'Este es mi
sitio Web cool hecho con Yii!'], 'description');
$this->registerMetaTag(['name' => 'description', 'content' => 'Este sitio
Web es sobre mapaches graciosos.'], 'description');
```

Registrar Link Tags

Tal como los meta tags, los link tags son útiles en muchos casos, como personalizar el ícono (favicon) del sitio, apuntar a una fuente de RSS o delegar OpenID a otro servidor. Puedes trabajar con link tags, al igual que con meta tags, utilizando `yii\web\View::registerLinkTag()`. Por ejemplo, en el contenido de una vista, puedes registrar un link tag como se muestra a continuación:

```
$this->registerLinkTag([
    'title' => 'Noticias en Vivo de Yii',
    'rel' => 'alternate',
    'type' => 'application/rss+xml',
    'href' => 'https://www.yiiframework.com/rss.xml/',
]);
```

El resultado del código es el siguiente:

```
<link title="Noticias en Vivo de Yii" rel="alternate"
type="application/rss+xml" href="https://www.yiiframework.com/rss.xml/">
```

Al igual que con `registerMetaTags()`, puedes especificar una clave al llamar a `registerLinkTag()` para evitar registrar link tags repetidos.

3.7.5. Eventos de Vistas

Los `componentes de vistas` disparan varios eventos durante el proceso de renderizado de la vista. Puedes responder a estos eventos para inyectar contenido a la vista o procesar el resultado de la renderización antes de que sea enviada al usuario final.

- `EVENT_BEFORE_RENDER`: disparado al principio del renderizado de un archivo en un controlador. Los manejadores de este evento pueden definir `yii\base\ViewEvent::$isValid` como `false` para cancelar el proceso de renderizado.
- `EVENT_AFTER_RENDER`: disparado luego de renderizar un archivo con la llamada de `yii\base\View::afterRender()`. Los manejadores de este evento pueden obtener el resultado del renderizado a través de `yii\base\ViewEvent::$output` y modificar esta propiedad para cambiar dicho resultado.
- `EVENT_BEGIN_PAGE`: disparado por la llamada a `yii\base\View::beginPage()` en layouts.
- `EVENT_END_PAGE`: disparado por la llamada a `yii\base\View::endPage()` en layouts.
- `EVENT_BEGIN_BODY`: disparado por la llamada a `yii\web\View::beginBody()` en layouts.
- `EVENT_END_BODY`: disparado por la llamada a `yii\web\View::endBody()` en layouts.

Por ejemplo, el siguiente código inyecta la fecha actual al final del body de la página:

```
\Yii::$app->view->on(View::EVENT_END_BODY, function () {
    echo date('Y-m-d');
});
```

3.7.6. Renderizar Páginas Estáticas

Con páginas estáticas nos referimos a esas páginas cuyo contenido es mayormente estático y sin necesidad de acceso a datos dinámicos enviados desde los controladores.

Puedes generar páginas estáticas utilizando un código como el que sigue dentro de un controlador:

```
public function actionAbout()
{
    return $this->render('about');
}
```

Si un sitio Web contiene muchas páginas estáticas, resultaría tedioso repetir el mismo código en muchos lados. Para resolver este problema, puedes introducir una acción independiente llamada `yii\web\ViewAction` en el controlador. Por ejemplo,

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'page' => [
                'class' => 'yii\web\ViewAction',
            ],
        ];
    }
}
```

Ahora, si creamos una vista llamada `about` bajo el directorio `@app/views/site/pages`, serás capáz de mostrarla en la siguiente URL:

```
http://localhost/index.php?r=site%2Fpage&view=about
```

El parámetro `GET view` le comunica a `yii\web\ViewAction` cuál es la vista solicitada. La acción entonces buscará esta vista dentro de `@app/views/site/pages`. Puedes configurar la propiedad `yii\web\ViewAction::$viewPrefix` para cambiar el directorio en el que se buscarán dichas páginas.

3.7.7. Buenas Prácticas

Las vistas son responsables de la presentación de modelos en el formato que el usuario final desea. En general, las vistas

- deberían contener principalmente sólo código de presentación, como HTML, y PHP simple para recorrer, dar formato y renderizar datos.
- no deberían contener código que realiza consultas a la base de datos. Ese tipo de código debe ir en los modelos.
- deberían evitar el acceso directo a datos del `request`, como `$_GET` y/o `$_POST`. Esto es una responsabilidad de los controladores. Si se necesitan datos del `request`, deben ser inyectados a la vista desde el controlador.
- pueden leer propiedades del modelo, pero no debería modificarlas.

Para hacer las vistas más manejables, evita crear vistas que son demasiado complejas o que contengan código redundante. Puedes utilizar estas técnicas para alcanzar dicha meta:

- utiliza layouts para representar secciones comunes (ej. encabezado y footer de la página).
- divide una vista compleja en varias más simples. Las vistas pequeñas pueden ser renderizadas y unidas una mayor utilizando los métodos de renderización antes descritos.
- crea y utiliza widgets como bloques de construcción de la vista.
- crea y utilizar helpers para transformar y dar formato a los datos en la vista.

3.8. Filtros

Los Filtros (filters) son objetos que se ejecutan antes y/o después de las acciones de controlador. Por ejemplo, un filtro de control de acceso puede ejecutarse antes de las acciones para asegurar que un usuario final tiene permitido acceder a estas; un filtro de compresión de contenido puede ejecutarse después de las acciones para comprimir el contenido de la respuesta antes de ser enviado al usuario final.

Un filtro puede consistir en un pre-filtro (lógica de filtrado aplicada *antes* de las acciones) y/o un post-filtro (lógica de filtro aplicada *después* de las acciones).

3.8.1. Uso de Filtros

Los filtros son esencialmente un tipo especial de comportamientos (behaviors). Por lo tanto, usar filtros es lo mismo que uso de comportamientos. Se pueden declarar los filtros en una clase controlador sobrescribiendo el método `behaviors()` como en el siguiente ejemplo:

```
public function behaviors()
{
```

```
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index', 'view'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

Por defecto, los filtros declarados en una clase controlador, serán aplicados en *todas* las acciones de este controlador. Sin embargo, se puede especificar explícitamente en que acciones serán aplicadas configurando la propiedad `only`. En el anterior ejemplo, el filtro 'HttpCache' solo se aplica a las acciones 'index' y 'view'. También se puede configurar la propiedad `except` para prevenir que ciertas acciones sean filtradas.

Además de en los controladores, se pueden declarar filtros en módulos o aplicaciones. Una vez hecho, los filtros serán aplicados a *todas* las acciones de controlador que pertenezcan a ese modulo o aplicación, a menos que las propiedades `only` y `except` sean configuradas como se ha descrito anteriormente.

> **Nota:** Cuando se declaran filtros en módulos o aplicaciones, deben usarse rutas en lugar de IDs de acciones en las propiedades `only` y `except`. Esto es debido a que los IDs de acciones no pueden especificar acciones dentro del ámbito de un modulo o una aplicación por si mismos.

Cuando se configuran múltiples filtros para una misma acción, se aplican de acuerdo a las siguientes reglas:

- Pre-filtrado
  - Aplica filtros declarados en la aplicación en orden de aparición en `behaviors()`.
  - Aplica filtros declarados en el modulo en orden de aparición en `behaviors()`.
  - Aplica filtros declarados en el controlador en orden de aparición en `behaviors()`.
  - Si hay algún filtro que cancele la ejecución de la acción, los filtros(tanto pre-filtros como post-filtros) posteriores a este no serán aplicados.
- Ejecución de la acción si pasa el pre-filtro.
- Post-filtrado
  - Aplica los filtros declarados en el controlador en el controlador en orden inverso al de aparición en `behaviors()`.

- Aplica los filtros declarados en el modulo en orden inverso al de aparición en `behaviors()`.
- Aplica los filtros declarados en la aplicación en orden inverso al de aparición en `behaviors()`.

3.8.2. Creación de Filtros

Para crear un nuevo filtro de acción, hay que extender a `yii\base\ActionFilter` y sobrescribir los métodos `beforeAction()` y/o `afterAction()`. El primero será ejecutado antes de la acción mientras que el segundo lo hará una vez ejecutada la acción. El valor devuelto por `beforeAction()` determina si una acción debe ejecutarse o no. Si el valor es falso, los filtros posteriores a este serán omitidos y la acción no será ejecutada.

El siguiente ejemplo muestra un filtro que registra el tiempo de ejecución de una acción:

```
namespace app\components;

use Yii;
use yii\base\ActionFilter;

class ActionTimeFilter extends ActionFilter
{
    private $_startTime;

    public function beforeAction($action)
    {
        $this->_startTime = microtime(true);
        return parent::beforeAction($action);
    }

    public function afterAction($action, $result)
    {
        $time = microtime(true) - $this->_startTime;
        Yii::debug("Action '{$action->uniqueId}' spent $time second.");
        return parent::afterAction($action, $result);
    }
}
```

3.8.3. Filtros del Núcleo

Yii proporciona una serie de filtros de uso general, que se encuentran principalmente en `yii\filters` namespace. En adelante introduciremos estos filtros brevemente.

`AccessControl`

AccessControl proporciona control de acceso simple basado en un conjunto de `rules`. En concreto, antes de ejecutar una acción, AccessControl

examinará la lista de reglas y encontrará la primera que concuerde con las actuales variables de contexto(tales como dirección IP de usuario, estado de inicio de sesión del usuario, etc.). La regla que concuerde dictara si se permite o deniega la ejecución de la acción solicitada. Si ninguna regla concuerda, el acceso será denegado.

El siguiente ejemplo muestra como habilitar el acceso a los usuarios autenticados a las acciones 'create' y 'update' mientras deniega a todos los otros usuarios el acceso a estas dos acciones.

```
use yii\filters\AccessControl;

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::class,
            'only' => ['create', 'update'],
            'rules' => [
                // permitido para usuarios autenticados
                [
                    'allow' => true,
                    'roles' => ['@'],
                ],
                // todo lo demás se deniega por defecto
            ],
        ],
    ];
}
```

Para conocer más detalles acerca del control de acceso en general, refiérase a la sección de Autorización

Filtros del Método de Autenticación

Los filtros del método de autenticación se usan para autenticar a un usuario utilizando varios métodos, tales como la Autenticación de acceso básico HTTP[16], Oauth 2[17]. Estas clases de filtros se encuentran en el espacio de nombres `yii\filters\auth`.

El siguiente ejemplo muestra como usar `yii\filters\auth\HttpBasicAuth` para autenticar un usuario usando un token de acceso basado en el método de Autenticación de acceso básico HTTP. Tenga en cuenta que para que esto funcione, la clase `user identity class` debe implementar el método `findIdentityByAccessToken()`.

```
use yii\filters\auth\HttpBasicAuth;
```

[16] https://es.wikipedia.org/wiki/Autenticaci%C3%B3n_de_acceso_b%C3%A1sica
[17] https://oauth.net/2/

```
public function behaviors()
{
    return [
        'basicAuth' => [
            'class' => HttpBasicAuth::class,
        ],
    ];
}
```

Los filtros del método de autenticación se usan a menudo para implementar APIs RESTful. Para más detalles, por favor refiérase a la sección Autenticación RESTful.

`ContentNegotiator` El filtro ContentNegotiator da soporte a la negociación del formato de respuesta y a la negociación del idioma de la aplicación. Este determinara el formato de respuesta y/o el idioma examinando los parámetros 'GET' y 'Accept' del encabezado HTTP.

En el siguiente ejemplo, el filtro ContentNegotiator se configura para soportar los formatos de respuesta 'JSON' y 'XML', y los idiomas Ingles(Estados Unidos) y Alemán.

```
use yii\filters\ContentNegotiator;
use yii\web\Response;

public function behaviors()
{
    return [
        [
            'class' => ContentNegotiator::class,
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ];
}
```

Los formatos de respuesta y los idiomas a menudo precisan ser determinados mucho antes durante el ciclo de vida de la aplicación. Por esta razón, ContentNegotiator esta diseñado de tal manera que se pueda usar como componente de bootstrapping así como de filtro. Por ejemplo, ContentNegotiator se puede configurar en la configuración de la aplicación como en el siguiente ejemplo:

```
use yii\filters\ContentNegotiator;
use yii\web\Response;
```

```
[
    'bootstrap' => [
        [
            'class' => ContentNegotiator::class,
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ],
];
```

> **Información:** En el caso que el tipo preferido de contenido y el idioma no puedan ser determinados por una petición, será utilizando el primer elemento de formato e idioma de la lista `formats` y `lenguages`.

HttpCache

HttpCache implementa un almacenamiento caché del lado del cliente utilizando las cabeceras HTTP 'Last-Modified' y 'Etag'. Por ejemplo:

```
use yii\filters\HttpCache;

public function behaviors()
{
    return [
        [
            'class' => HttpCache::class,
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

Para conocer más detalles acerca de HttpCache refiérase a la sección almacenamiento caché HTTP.

PageCache

PageCache implementa una caché por parte del servidor de paginas enteras. En el siguiente ejemplo, se aplica PageCache a la acción 'index' para generar una cache de la pagina entera durante 60 segundos como máximo o

hasta que el contador de entradas de la tabla 'post' varíe. También se encarga de almacenar diferentes versiones de la pagina dependiendo del idioma de la aplicación seleccionado.

```
use yii\filters\PageCache;
use yii\caching\DbDependency;

public function behaviors()
{
    return [
        'pageCache' => [
            'class' => PageCache::class,
            'only' => ['index'],
            'duration' => 60,
            'dependency' => [
                'class' => DbDependency::class,
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
            'variations' => [
                \Yii::$app->language,
            ]
        ],
    ];
}
```

Por favor refiérase a Caché de Páginas para obtener más detalles acerca de como usar PageCache.

RateLimiter

RateLimiter implementa un algoritmo de para limitar la tasa de descarga basándose en leaky bucket algorithm[18]. Este se utiliza sobre todo en la implementación de APIs RESTful. Por favor, refiérase a la sección limite de tasa para obtener más detalles acerca de el uso de este filtro.

VerbFilter

VerbFilter comprueba que los métodos de las peticiones HTTP estén permitidas para las acciones solicitadas. Si no están permitidas, lanzara una excepción de tipo HTTP 405. En el siguiente ejemplo, se declara VerbFilter para especificar el conjunto típico métodos de petición permitidos para acciones CRUD.

```
use yii\filters\VerbFilter;

public function behaviors()
{
    return [
```

[18] https://en.wikipedia.org/wiki/Leaky_bucket

```
        'verbs' => [
            'class' => VerbFilter::class,
            'actions' => [
                'index'  => ['get'],
                'view'   => ['get'],
                'create' => ['get', 'post'],
                'update' => ['get', 'put', 'post'],
                'delete' => ['post', 'delete'],
            ],
        ],
    ];
}
```

Cors

CORS[19] es un mecanismo que permite a diferentes recursos (por ejemplo: fuentes, JavaScript, etc) de una pagina Web ser solicitados por otro dominio diferente al dominio que esta haciendo la petición. En particular las llamadas AJAX de JavaScript pueden utilizar el mecanismo XMLHttpRequest. De otro modo esta petición de dominio cruzado seria prohibida por los navegadores Web, por la misma pollita de seguridad de origen. CORS establece la manera en que el navegador y el servidor pueden interaccionar para determinar si se permite o no la petición de dominio cruzado. El filtro `Cors filter` puede ser definido antes de los filtros Autenticación / Autorización para asegurar que las cabeceras de CORS siempre serán enviadas.

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::class,
        ],
    ], parent::behaviors());
}
```

El filtrado CORS puede ser ajustado utilizando la propiedad 'cors'.

- `cors['Origin']`: array utilizado para definir los orígenes permitidos. Puede ser `['*']` (everyone) o `['https://www.myserver.net', 'https://www.myotherserver.com']`. Por defecto `['*']`.
- `cors['Access-Control-Request-Method']`: array de los verbos permitidos como `['GET', 'OPTIONS', 'HEAD']`. Por defecto `['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS']`.
- `cors['Access-Control-Request-Headers']`: array de las cabeceras permitidas. Puede ser `['*']` todas las cabeceras o algunas especificas `['X-Request-With']`. Por defecto `['*']`.

[19] https://developer.mozilla.org/es/docs/Web/HTTP/CORS

- `cors['Access-Control-Allow-Credentials']`: define si la petición actual puede hacer uso de credenciales. Puede ser `true`, `false` o `null` (not set). Por defecto `null`.
- `cors['Access-Control-Max-Age']`: define el tiempo de vida del la petición pref-flight. Por defecto `86400`. Por ejemplo, habilitar CORS para el origen: `https://www.myserver.net` con métodos `GET`, `HEAD` y `OPTIONS`:

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::class,
            'cors' => [
                'Origin' => ['https://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD',
                'OPTIONS'],
            ],
        ],
    ], parent::behaviors());
}
```

Se pueden ajustar las cabeceras de CORS sobrescribiendo los parámetros por defecto de una acción. Por ejemplo añadir `Access-Control-Allow-Credentials` a la acción 'login', se podría hacer así:

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::class,
            'cors' => [
                'Origin' => ['https://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD',
                'OPTIONS'],
            ],
            'actions' => [
                'login' => [
                    'Access-Control-Allow-Credentials' => true,
                ]
            ]
        ],
    ], parent::behaviors());
}
```

3.9. Widgets

Los *widgets* son bloques de código reutilizables que se usan en las vistas para crear elementos de interfaz de usuario complejos y configurables, de forma orientada a objetos. Por ejemplo, un *widget* de selección de fecha puede generar un selector de fechas bonito que permita a los usuarios seleccionar una fecha. Todo lo que hay que hacer es insertar el siguiente código en una vista:

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget(['name' => 'date']) ?>
```

Yii incluye un buen número de *widgets*, tales como `formulario activo`, `menú`, *widgets* de jQuery UI[20], y *widgets* de Twitter Bootstrap[21]. A continuación presentaremos las nociones básicas de de los *widgets*. Por favor, refiérase a la documentación de la API de clases si quiere aprender más acerca del uso de un *widget* en particular.

3.9.1. Uso de los *widgets*

Los *widgets* se usan principalmente en las vistas. Se puede llamar al método `yii\base\Widget::widget()` para usar un *widget* en una vista. El método toma un *array* de configuración para inicializar el *widget* y devuelve la representación resultante del *widget*. Por ejemplo, el siguiente código inserta un *widget* de selección de fecha configurado para usar el idioma ruso y guardar la selección en el atributo `from_date` de `$model`.

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget([
    'model' => $model,
    'attribute' => 'from_date',
    'language' => 'ru',
    'dateFormat' => 'php:Y-m-d',
]) ?>
```

Algunos *widgets* pueden coger un bloque de contenido que debería encontrarse entre la invocación de `yii\base\Widget::begin()` y `yii\base\Widget::end()`. Por ejemplo, el siguiente código usa el *widget* `yii\widgets\ActiveForm` para generar un formulario de inicio de sesión. El *widget* generará las etiquetas `<form>` de apertura y cierre donde se llame a `begin()` y `end()` respectivamente. Cualquier cosa que este en medio se representará tal cual.

[20] https://www.yiiframework.com/extension/yiisoft/yii2-jui
[21] https://www.yiiframework.com/extension/yiisoft/yii2-bootstrap

```
<?php
use yii\widgets\ActiveForm;
use yii\helpers\Html;
?>

<?php $form = ActiveForm::begin(['id' => 'login-form']); ?>

    <?= $form->field($model, 'username') ?>

    <?= $form->field($model, 'password')->passwordInput() ?>

    <div class="form-group">
        <?= Html::submitButton('Login') ?>
    </div>

<?php ActiveForm::end(); ?>
```

Hay que tener en cuenta que, a diferencia de `yii\base\Widget::widget()` que devuelve la representación resultante del *widget*, el método `yii\base\Widget::begin()` devuelve una instancia del *widget*, que se puede usar para generar el contenido del *widget*.

> Nota: Algunos *widgets* utilizan un búfer de salida[22] para ajustar el contenido rodeado al invocar `yii\base\Widget::end()`. Por este motivo se espera que las llamadas a `yii\base\Widget::begin()` y `yii\base\Widget::end()` tengan lugar en el mismo fichero de vista. No seguir esta regla puede desembocar en una salida distinta a la esperada.

Configuración de las variables globales predefinidas

Las variables globales predefinidas de un *widget* se pueden configurar por medio del contenedor de inyección de dependencias:

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

Consulte la sección "Uso práctico" de la Guía del contenedor de inyección de dependencias para más detalles.

3.9.2. Creación de *widgets*

Para crear un *widget*, extienda la clase `yii\base\Widget` y sobrescriba los métodos `yii\base\Widget::init()` y/o `yii\base\Widget::run()`. Normalmente el método `init()` debería contener el código que inicializa las propiedades del *widget*, mientras que el método `run()` debería contener el código que genera la representación resultante del *widget*. La representación

[22] https://www.php.net/manual/es/book.outcontrol.php

resultante del método `run()` puede pasarse directamente a `echo` o devolverse como una cadena.

En el siguiente ejemplo, `HelloWidget` codifica en HTML y muestra el contenido asignado a su propiedad `message`. Si la propiedad no está establecida, mostrará «Hello World» por omisión.

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
{
    public $message;

    public function init()
    {
        parent::init();
        if ($this->message === null) {
            $this->message = 'Hello World';
        }
    }

    public function run()
    {
        return Html::encode($this->message);
    }
}
```

Para usar este *widget*, simplemente inserte el siguiente código en una vista:

```
<?php
use app\components\HelloWidget;
?>
<?= HelloWidget::widget(['message' => 'Good morning']) ?>
```

Abajo se muestra una variante de `HelloWidget` que toma el contenido insertado entre las llamadas a `begin()` y `end()`, lo codifica en HTML y posteriormente lo muestra.

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
{
    public function init()
    {
        parent::init();
        ob_start();
    }
```

```
    public function run()
    {
        $content = ob_get_clean();
        return Html::encode($content);
    }
}
```

Como se puede observar, el búfer de salida de PHP es iniciado en `init()` para que toda salida entre las llamadas de `init()` y `run()` puede ser capturada, procesada y devuelta en `run()`.

> **Información:** Cuando llame a `yii\base\Widget::begin()`, se creará una nueva instancia del *widget* y se llamará a su método `init()` al final del constructor del *widget*. Cuando llame a `yii\base\Widget::end()`, se invocará el método `run()` y el resultado que devuelva será pasado a `echo` por `end()`.

El siguiente código muestra cómo usar esta nueva variante de `HelloWidget`:

```
<?php
use app\components\HelloWidget;
?>
<?php HelloWidget::begin(); ?>

    contenido que puede contener <etiqueta>s

<?php HelloWidget::end(); ?>
```

A veces, un *widget* puede necesitar representar un gran bloque de contenido. Aunque que se podría incrustar el contenido dentro del método `run()`, es preferible ponerlo dentro de una vista y llamar al método `yii\base\Widget::render()` para representarlo. Por ejemplo:

```
public function run()
{
    return $this->render('hello');
}
```

Por omisión, las vistas para un *widget* deberían encontrarse en ficheros dentro del directorio `WidgetPath/views`, donde `WidgetPath` representa el directorio que contiene el fichero de clase del *widget*. Por lo tanto, el ejemplo anterior representará el fichero de vista `@app/components/views/hello.php`, suponiendo que la clase del *widget* se encuentre en `@app/components`. Se puede sobrescribir el método `yii\base\Widget::getViewPath()` para personalizar el directorio que contiene los ficheros de vista del *widget*.

3.9.3. Buenas prácticas

Los *widgets* son una manera orientada a objetos de reutilizar código de las vistas.

Al crear *widgets*, debería continuar suguiendo el patrón MVC. En general, se debería mantener la lógica en las clases del widget y la presentación en las vistas.

Los *widgets* deberían diseñarse para ser autosuficientes. Es decir, cuando se use un *widget*, se debería poder ponerlo en una vista sin hacer nada más. Esto puede resultar complicado si un *widget* requiere recursos externos, tales como CSS, JavaScript, imágenes, etc. Afortunadamente Yii proporciona soporte para paquetes de recursos (*asset bundles*) que se pueden utilizar para resolver este problema.

Cuando un *widget* sólo contiene código de vista, es muy similar a una vista. De hecho, en este caso, su única diferencia es que un *widget* es una clase redistribuible, mientras que una vista es sólo un simple script PHP que prefiere mantener dentro de su aplicación.

3.10. Módulos

Los módulos son unidades de software independientes que consisten en modelos, vistas, controladores, y otros componentes de apoyo. Los usuarios finales pueden acceder a los controladores de un módulo cuando éste está instalado en la aplicación. Por éstas razones, los módulos a menudo se consideran como mini-aplicaciones. Los módulos difieren de las aplicaciones en que los módulos no pueden ser desplegados solos y tienen que residir dentro de aplicaciones.

3.10.1. Creación de Módulos

Un módulo está organizado de tal manera que contiene un directorio llamado `base path` del módulo. Dentro de este directorio, hay subdirectorios tales como 'controllers', 'models', 'views', que contienen controladores, modelos, vistas y otro código, exactamente como una aplicación. El siguiente ejemplo muestra el contenido dentro de un módulo:

```
forum/
    Module.php                  archivo clase módulo
    controllers/                contiene archivos de la clase controlador
        DefaultController.php   archivo clase controlador por defecto
    models/                     contiene los archivos de clase modelo
    views/                      contiene las vistas de controlador y los
    archivos de diseño
        layouts/                contiene los archivos de diseño de las
        vistas
```

```
        default/                contiene los archivos de vista del
        DefaultController
            index.php           archivo de vista del index
```

Clases Módulo

Cada módulo debe tener una única clase módulo que extiende a `yii\base\Module`. La clase debe encontrarse directamente debajo del `base path` y debe ser autocargable. Cuando se está accediendo a un módulo, se creará una única instancia de la clase módulo correspondiente. Como en las instancias de aplicación, las instancias de módulo se utilizan para compartir datos y componentes de código dentro de los módulos.

El siguiente ejemplo muestra como podría ser una clase módulo.

```php
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->params['foo'] = 'bar';
        // ...  otro código de inicialización ...
    }
}
```

Si el método 'init()' contiene mucho código de inicialización de las propiedades del módulo, también se puede guardar en términos de configuración y cargarlo con el siguiente código 'init()':

```php
public function init()
{
    parent::init();
    // inicializa el módulo con la configuración cargada desde config.php
    \Yii::configure($this, require __DIR__ . '/config.php');
}
```

donde el archivo de configuración 'config.php' puede contener el siguiente contenido, similar al de configuraciones de aplicación.

```php
<?php
return [
    'components' => [
        // lista de configuraciones de componente
    ],
    'params' => [
        // lista de parámetros
    ],
];
```

Controladores en Módulos

Cuando se crean controladores en un modelo, una convención es poner las clases controlador debajo del sub-espacio de nombres de 'controllers' del espacio de nombres de la clase módulo. Esto también significa que los archivos de la clase controlador deben ponerse en el directorio 'controllers' dentro del `base path` del módulo. Por ejemplo, para crear un controlador 'post' en el módulo 'forum' mostrado en la última subdivisión, se debe declarar la clase controlador de la siguiente manera:

```
namespace app\modules\forum\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    // ...
}
```

Se puede personalizar el espacio de nombres de las clases controlador configurando la propiedad `yii\base\Module::$controllerNamespace`. En el caso que alguno de los controladores esté fuera del espacio de nombres, se puede hacer accesible configurando la propiedad `yii\base\Module::$controllerMap`, similar a como se hace en una aplicación.

Vistas en Módulos

Las vistas en un módulo deben alojarse en el directorio 'views' dentro del módulo del `base path`. Las vistas renderizadas por un controlador en el módulo, deben alojarse en el directorio 'views/ControllerID', donde el 'ControllerID' hace referencia al ID del controlador. Por ejemplo, si la clase controlador es 'PostController', el directorio sería 'views/post' dentro del `base path` del módulo.

Un modulo puede especificar un layout que se aplica a las vistas renderizadas por los controladores del módulo. El layout debe alojarse en el directorio 'views/layouts' por defecto, y se puede configurar la propiedad `yii\base\Module::$layout` para apuntar al nombre del layout. Si no se configura la propiedad 'layout', se usar el layout de la aplicación.

3.10.2. Uso de los Módulos

Para usar un módulo en una aplicación, simplemente se tiene que configurar la aplicación añadiendo el módulo en la propiedad `modules` de la aplicación. El siguiente ejemplo de la configuración de la aplicación usa el modelo 'forum':

```
[
    'modules' => [
```

```
        'forum' => [
            'class' => 'app\modules\forum\Module',
            // ... otras configuraciones para el módulo ...
        ],
    ],
]
```

La propiedad `modules` contiene un array de configuraciones de módulo. Cada clave del array representa un *ID de módulo* que identifica de forma única el módulo de entre todos los módulos de la aplicación, y el correspondiente valor del array es la configuración para crear el módulo.

Rutas

De Igual manera que el acceso a los controladores en una aplicación, las rutas se utiliza para dirigirse a los controladores en un módulo. Una ruta para un controlador dentro de un módulo debe empezar con el ID del módulo seguido por el ID del controlador y el ID de la acción. Por ejemplo, si una aplicación usa un módulo llamado 'forum', la ruta 'forum/post/index' representaría la acción 'index' del controlador 'post' en el módulo. Si la ruta sólo contiene el ID del módulo, entonces la propiedad `yii\base\Module::$defaultRoute` que por defecto es 'default', determinara que controlador/acción debe usarse. Esto significa que la ruta 'forum' representaría el controlador 'default' en el módulo 'forum'.

Acceder a los Módulos

Dentro de un módulo, se puede necesitar obtener la instancia de la clase módulo para poder acceder al ID del módulo, componentes del módulo, etc. Se puede hacer usando la siguiente declaración:

```
$module = MyModuleClass::getInstance();
```

Dónde 'MyModuleClass' hace referencia al nombre de la clase módulo en la que estemos interesados. El método 'getInstance()' devolverá la instancia actualmente solicitada de la clase módulo. Si no se solicita el módulo, el método devolverá nulo. Hay que tener en cuenta que si se crea una nueva instancia del módulo, esta será diferente a la creada por Yii en respuesta a la solicitud.

> **Información:** Cuando se desarrolla un módulo, no se debe dar por sentado que el módulo usará un ID fijo. Esto se debe a que un módulo puede asociarse a un ID arbitrario cuando se usa en una aplicación o dentro de otro módulo. Para obtener el ID del módulo, primero se debe usar el código del anterior ejemplo para obtener la instancia y luego el ID mediante '$modeule->id'.

También se puede acceder a la instancia de un módulo usando las siguientes declaraciones:

```
// obtiene el modulo hijo cuyo ID es "forum"
$module = \Yii::$app->getModule('forum');

// obtiene el módulo al que pertenece la petición actual
$module = \Yii::$app->controller->module;
```

El primer ejemplo sólo es útil cuando conocemos el ID del módulo, mientras que el segundo es mejor usarlo cuando conocemos los controladores que se están solicitando.

Una vez obtenida la instancia del módulo, se puede acceder a parámetros o componentes registrados con el módulo. Por ejemplo:

```
$maxPostCount = $module->params['maxPostCount'];
```

Bootstrapping Módulos

Puede darse el caso en que necesitemos que un módulo se ejecute en cada petición. El módulo `yii\debug\Module` es un ejemplo. Para hacerlo, tenemos que listar los IDs de los módulos en la propiedad `bootstrap` de la aplicación.

Por ejemplo, la siguiente configuración de aplicación se asegura de que el módulo 'debug' siempre se cargue:

```
[
    'bootstrap' => [
        'debug',
    ],

    'modules' => [
        'debug' => 'yii\debug\Module',
    ],
]
```

3.10.3. Módulos anidados

Los módulos pueden ser anidados sin límite de niveles. Es decir, un módulo puede contener un módulo y éste a la vez contener otro módulo. Nombramos *padre* al primero mientras que al segundo lo nombramos *hijo*. Los módulos hijo se tienen que declarar en la propiedad `modules` de sus módulos padre. Por ejemplo:

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
```

```
        parent::init();

        $this->modules = [
            'admin' => [
                // debe considerarse usar un nombre de espacios más corto!
                'class' => 'app\modules\forum\modules\admin\Module',
            ],
        ];
    }
}
```

En un controlador dentro de un módulo anidado, la ruta debe incluir el ID de todos los módulos antecesores. Por ejemplo, la ruta 'forum/admin/dashboard/index' representa la acción 'index' del controlador 'dashboard' en el módulo 'admin' que es el módulo hijo del módulo 'forum'.

> **Información:** El método `getModule()` sólo devuelve el módulo hijo que pertenece directamente a su padre. La propiedad `yii\base\Application::$loadedModules` contiene una lista de los módulos cargados, incluyendo los hijos directos y los anidados, indexados por sus nombres de clase.

3.10.4. Mejores Prácticas

Es mejor usar los módulos en grandes aplicaciones en las que sus funcionalidades puedan ser divididas en diferentes grupos, cada uno compuesto por funcionalidades directamente relacionadas. Cada grupo de funcionalidades se puede desarrollar como un módulo que puede ser desarrollado y mantenido por un programador o equipo específico.

Los módulos también son una buena manera de reutilizar código a nivel de grupo de funcionalidades. Algunas funcionalidades de uso común, tales como la gestión de usuarios o la gestión de comentarios, pueden ser desarrollados como módulos para que puedan ser fácilmente reutilizados en futuros proyectos.

3.11. Assets

Un asset en Yii es un archivo al que se puede hacer referencia en una página Web. Puede ser un archivo CSS, un archivo JavaScript, una imagen o un archivo de video, etc. Los assets se encuentran en los directorios públicos de la web y se sirven directamente por los servidores Web.

A menudo es preferible gestionar los assets mediante programación. Por ejemplo, cuando se usa el widget `yii\jui\DatePicker` en una página, éste incluirá automáticamente los archivos CSS y JavaScript requeridos, en vez de tener que buscar los archivos e incluirlos manualmente. Y cuando se actualice el widget a una nueva versión, ésta usará de forma automática la

nueva versión de los archivos asset. En este tutorial, se describirá la poderosa capacidad que proporciona la gestión de assets en Yii.

3.11.1. Asset Bundles

Yii gestiona los assets en unidades de *asset bundle*. Un asset bundle es simplemente un conjunto de assets localizados en un directorio. Cuando se registra un asset bundle en una vista, éste incluirá los archivos CSS y JavaScript del bundle en la página Web renderizada.

3.11.2. Definición de Asset Bundles

Los asset bundles son descritos como clases PHP que extienden a `yii\web\AssetBundle`. El nombre del bundle es simplemente su correspondiente nombre de la classe PHP que debe ser autocargable. En una clase asset bundle, lo más habitual es especificar donde se encuentran los archivos asset, que archivos CSS y JavaScript contiene el bundle, y como depende este bundle de otros bundles.

El siguiente código define el asset bundle principal que se usa en la plantilla de aplicación básica:

```
<?php

namespace app\assets;

use yii\web\AssetBundle;

class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.css',
    ];
    public $js = [
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

La anterior clase `AppAsset` especifica que los archivos asset se encuentran en el directorio `@webroot` que corresponde a la URL `@web`; el bundle contiene un único archivo CSS `css/site.css` y ningún archivo JavaScript; el bundle depende de otros dos bundles: `yii\web\YiiAsset` y `yii\bootstrap\BootstrapAsset`. A continuación se explicarán más detalladamente las propiedades del `yii\web\AssetBundle`:

- `sourcePath`: especifica el directorio raíz que contiene los archivos asset en el bundle. Si no, se deben especificar las propiedades `basePath` y `baseUrl`, en su lugar. Se pueden usar alias de ruta.
- `basePath`: especifica el directorio Web público que contiene los archivos assets de este bundle. Cuando se especifica la propiedad `sourcePath`, el gestor de assets publicará los assets de este bundle en un directorio Web público y sobrescribirá la propiedad en consecuencia. Se debe establecer esta propiedad si los archivos asset ya se encuentran en un directorio Web público y no necesitan ser publicados. Se pueden usar alias de ruta.
- `baseUrl`: especifica la URL correspondiente al directorio `basePath`. Como en `basePath`, si se especifica la propiedad `sourcePath`, el gestor de assets publicara los assets y sobrescribirá esta propiedad en consecuencia. Se pueden usar alias de ruta.
- `js`: un array lista los archivos JavaScript que contiene este bundle. Tenga en cuenta que solo deben usarse las barras invertidas "/" como separadores de directorios. Cada archivo Javascript se puede especificar en uno de los siguientes formatos:
  - una ruta relativa que represente un archivo local JavaScript (ej. `js/main.js`). La ruta actual del fichero se puede determinar anteponiendo `yii\web\AssetManager::$basePath` a la ruta relativa, y la URL actual de un archivo puede ser determinada anteponiendo `yii\web\AssetManager::$baseUrl` a la ruta relativa.
  - una URL absoluta que represente un archivo JavaScript externo. Por ejemplo, `https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js` o `//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js`.
- `css`: un array que lista los archivos CSS que contiene este bundle. El formato de este array es el mismo que el de `js`.
- `depends`: un array que lista los nombres de los asset bundles de los que depende este asset bundle (para explicarlo brevemente).
- `jsOptions`: especifica las opciones que se enviarán al método `yii\web\View::registerJsFile()` cuando se le llame para registrar *todos* los archivos JavaScript de este bundle.
- `cssOptions`: especifica las opciones que se enviarán al método `yii\web\View::registerCssFile()` cuando se le llame para registrar *todos* los archivos CSS de este bundle.
- `publishOptions`: especifica las opciones que se enviarán al método `yii\web\AssetManager::publish()` cuando se le llame para publicar los archivos de los assets fuente a un directorio Web. Solo se usa si se especifica la propiedad `sourcePath`.

Ubicación de los Assets

Según la localización de los assets, se pueden clasificar como:

- assets fuente (source assets): los assets se encuentran junto con el código fuente PHP, al que no se puede acceder directamente a través de la Web. Para usar los assets fuente en una página, deben ser copiados en un directorio público y transformados en los llamados assets publicados. El proceso se llama *publicación de assets* que será descrito a continuación.
- assets publicados (published assets): los archivos assets se encuentran en el directorio Web y son accesibles vía Web.
- assets externos (external assets): los archivos assets se encuentran en un servidor Web diferente al de la aplicación.

Cuando se define una clase asset bundle, si se especifica la propiedad `sourcePath`, significa que cualquier asset listado que use rutas relativas será considerado como un asset fuente. Si no se especifica la propiedad, significa que los assets son assets publicados (se deben especificar `basePath` y `baseUrl` para hacerle saber a Yii dónde se encuentran.)

Se recomienda ubicar los assets que correspondan a la aplicación en un directorio Web para evitar publicaciones de assets innecesarias. Por esto en el anterior ejemplo `AppAsset` especifica `basePath` en vez de `sourcePath`.

Para las extensiones, por el hecho de que sus assets se encuentran junto con el código fuente, en directorios que no son accesibles para la Web, se tiene que especificar la propiedad `sourcePath` cuando se definan clases asset bundle para ellas.

> **Nota:** No se debe usar `@webroot/assets` como `source path`. Este directorio se usa por defecto por el `asset manager` para guardar los archivos asset publicados temporalmente y pueden ser eliminados.

Dependencias de los Asset

Cuando se incluyen múltiples archivos CSS o JavaScript en una página Web, tienen que cumplir ciertas órdenes para evitar problemas de sobrescritura. Por ejemplo, si se usa un widget jQuery UI en una página Web, tenemos que asegurarnos de que el archivo JavaScript jQuery se incluya antes que el archivo JavaScript jQuery UI. A esto se le llama ordenar las dependencias entre archivos.

Las dependencias de los assets se especifican principalmente a través de la propiedad `yii\AssetBundle::depends`. En el ejemplo `AppAsset`, el asset bundle depende de otros dos asset bundles `yii\web\YiiAsset` y `yii\bootstrap\BootstrapAsset`, que significa que los archivos CSS y JavaScript en `AppAsset` se incluirán *después* que los archivos de los dos bundles dependientes.

Las dependencias son transitivas. Esto significa, que si un bundle A depende de un bundle B que depende de C, A dependerá de C, también.

Opciones de los Assets

Se pueden especificar las propiedades `cssOptions` y `jsOptions` para personalizar la forma en que los archivos CSS y JavaScript serán incluidos en una página. Los valores de estas propiedades serán enviadas a los métodos `yii\web\View::registerCssFile()` y `yii\web\View::registerJsFile()`, respectivamente cuando las vistas los llamen para incluir los archivos CSS y JavaScript.

> **Nota:** Las opciones que se especifican en una clase bundle se aplican a *todos* los archivos CSS/JavaScript de un bundle. Si se quiere usar diferentes opciones para diferentes archivos, se deben crear assets bundles separados y usar un conjunto de opciones para cada bundle.

Por ejemplo, para incluir una archivo CSS condicionalmente para navegadores que como IE9 o anteriores, se puede usar la siguiente opción:

```
public $cssOptions = ['condition' => 'lte IE9'];
```

Esto provoca que un archivo CSS dentro de un bundle sea incluido usando los siguientes tags HTML:

```
<!--[if lte IE9]>
<link rel="stylesheet" href="path/to/foo.css">
<![endif]-->
```

Para envolver el tag del enlace con `<noscript>` se puede usar el siguiente código:

```
public $cssOptions = ['noscript' => true];
```

Para incluir un archivo JavaScript en la sección cabecera (head) de una página (por defecto, los archivos JavaScript se incluyen al final de la sección cuerpo(body)), se puede usar el siguiente código:

```
public $jsOptions = ['position' => \yii\web\View::POS_HEAD];
```

Por defecto, cuando un asset bundle está siendo publicado, todos los contenidos del directorio especificado por `yii\web\AssetBundle::$sourcePath` serán publicados. Puedes personalizar este comportamiento configurando la propiedad `publishOptions`. Por ejemplo, públicar solo uno o unos pocos subdirectorios de `yii\web\AssetBundle::$sourcePath`, puedes hacerlo de la siguiente manera en la clase asset bundle:

```php
<?php
namespace app\assets;

use yii\web\AssetBundle;

class FontAwesomeAsset extends AssetBundle
{
    public $sourcePath = '@bower/font-awesome';
    public $css = [
        'css/font-awesome.min.css',
    ];

    public function init()
    {
        parent::init();
        $this->publishOptions['beforeCopy'] = function ($from, $to) {
            $dirname = basename(dirname($from));
            return $dirname === 'fonts' || $dirname === 'css';
        };
    }
}
```

El ejemplo anterior define un asset bundle para el "fontawesome" package[23]. Especificando la opción de publicación `beforeCopy`, solo los subdirectorios `fonts` y `css` serán publicados.

Bower y NPM Assets

La mayoría de paquetes JavaScript/CSS se gestionan con Bower[24] y/o NPM[25]. Si tu aplicación o extensión usa estos paquetes, se recomienda seguir los siguientes pasos para gestionar los assets en la librería:

1. Modificar el archivo `composer.json` de tu aplicación o extensión e introducir el paquete en la lista `require`. Se debe usar `bower-asset/PackageName` (para paquetes Bower) o `npm-asset/PackageName` (para paquetes NPM) para referenciar la librería.

2. Crear una clase asset bundle y listar los archivos JavaScript/CSS que se planea usar en la aplicación o extensión. Se debe especificar la propiedad `sourcePath` como `@bower\PackageName` o `@npm\PackageName`. Esto se debe a que Composer instalará el paquete Bower o NPM en el correspondiente directorio de este alias.

   Nota: Algunos paquetes pueden distribuir sus archivos en subdirectorios. Si es el caso, se debe especificar el subdirectorio como valor del `sourcePath`. Por ejemplo, `yii\web\JqueryAsset` usa `@bower/jquery/dist` en vez de `@bower/jquery`.

[23] https://fontawesome.com/
[24] https://bower.io/
[25] https://www.npmjs.com/

3.11.3. Uso de Asset Bundles

Para usar un asset bundle, debe registrarse con una vista llamando al método `yii\web\AssetBundle::register()`. Por ejemplo, en plantilla de vista se puede registrar un asset bundle como en el siguiente ejemplo:

```
use app\assets\AppAsset;
AppAsset::register($this);  // $this representa el objeto vista
```

> **Información:** El método `yii\web\AssetBundle::register()` devuelve un objeto asset bundle que contiene la información acerca de los assets publicados, tales como `basePath` o `baseUrl`.

Si se registra un asset bundle en otro lugar, se debe proporcionar la vista necesaria al objeto. Por ejemplo, para registrar un asset bundle en una clase widget, se puede obtener el objeto vista mediante `$this->view`.

Cuando se registra un asset bundle con una vista, por detrás, Yii registrará todos sus asset bundles dependientes. Y si un asset bundle se encuentra en un directorio inaccesible por la Web, éste será publicado a un directorio Web público. Después cuando la vista renderice una página, se generarán las etiquetas (tags) `<link>` y `<script>` para los archivos CSS y JavaScript listados en los bundles registrados. El orden de estas etiquetas será determinado por las dependencias entre los bundles registrados y los otros assets listados en las propiedades `yii\web\AssetBundle::$css` y `yii\web\AssetBundle::$js`.

Personalización de Asset Bundles

Yii gestiona los asset bundles a través de un componente de aplicación llamado `assetManager` que está implementado por `yii\web\AssetManager`. Configurando la propiedad `yii\web\AssetManager::$bundles`, se puede personalizar el comportamiento (behavior) de un asset bundle. Por ejemplo, de forma predeterminada, el asset bundle `yii\web\Jquery` , utiliza el archivo `jquery.js` desde el paquete Bower instalado. Para mejorar la disponibilidad y el rendimiento se puede querer usar la versión alojada por Google. Ésta puede ser obtenida configurando `assetManager` en la configuración de la aplicación como en el siguiente ejemplo:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => [
                    'sourcePath' => null,   // no publicar el bundle
                    'js' => [
                        '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',
```

```
                ]
            ],
        ],
    ],
],
];
```

Del mismo modo, se pueden configurar múltiples asset bundles a través de `yii\web\AssetManager::$bundles`. Las claves del array deben ser los nombres de clase (sin la primera barra invertida) de los asset bundles, y los valores del array deben ser las correspondientes configuraciones de arrays.

> **Consejo:** Se puede elegir condicionalmente que assets se van a usar en un asset bundle. El siguiente ejemplo muestra como usar `jquery.js` en el entorno de desarrollo y `jquery.min.js` en los otros casos:
>
> ```
> 'yii\web\JqueryAsset' => [
>     'js' => [
>         YII_ENV_DEV ? 'jquery.js' : 'jquery.min.js'
>     ]
> ],
> ```

Se puede deshabilitar uno o más asset bundles asociando `false` a los nombres de los asset bundles que se quieran deshabilitar. Cuando se registra un asset bundle deshabilitado con una vista, ninguno de sus bundles dependientes será registrado, y la vista tampoco incluirá ningún asset del bundle en la página que se renderice. Por ejemplo, para deshabilitar `yii\web\JqueryAsset`, se puede usar la siguiente configuración:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => false,
            ],
        ],
    ],
];
```

Además se pueden deshabilitar *todos* los asset bundles asignando `false` a `yii\web\AssetManager::$bundles`.

Mapeo de Assets (Asset Mapping)

A veces se puede querer "arreglar" rutas de archivos incorrectos/incompatibles usadas en múltiples asset bundles. Por ejemplo, el bundle A usa `jquery.min.js` con versión 1.11.1, y el bundle B usa `jquery.js` con versión 2.11.1. Mientras

que se puede solucionar el problema personalizando cada bundle, una forma más fácil, es usar la característica *asset map* para mapear los assets incorrectos a los deseados. Para hacerlo, se tiene que configurar la propiedad `yii\web\AssetManager::$assetMap` como en el siguiente ejemplo:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'assetMap' => [
                'jquery.js' =>
                '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',
            ],
        ],
    ],
];
```

Las claves de `assetmMap` son los nombres de los assets que se quieren corregir, y los valores son las rutas de los assets deseados. Cuando se registra un asset bundle con una vista, cada archivo de asset relativo de `css` y `js` serán contrastados con este mapa. Si se detecta que alguna de estas claves es la última parte de un archivo asset (prefijado con `yii\web\AssetBundle::$sourcePath`, si esta disponible), el correspondiente valor reemplazará el asset y será registrado con la vista. Por ejemplo, un archivo asset `mi/ruta/a/jquery.js` concuerda con la clave `jquery.js`.

> **Nota:** Sólo los assets especificados usando rutas relativas están sujetos al mapeo de assets. Y las rutas de los assets destino deben ser tanto URLs absolutas o rutas relativas a `yii\web\AssetManager::$basePath`.

Publicación de Asset

Como se ha comentado anteriormente, si un asset bundle se encuentra en un directorio que no es accesible por la Web, este asset será copiado a un directorio Web cuando se registre el bundle con una vista. Este proceso se llama *publicación de assets*, y se efectúa automáticamente por el `asset manager`.

De forma predeterminada, los assets se publican en el directorio `@webroot/assets` cuando corresponden a la URL `@web\assets`. Se puede personalizar esta ubicación configurando las propiedades `basePath` y `baseUrl`.

En lugar de publicar los assets copiando archivos, se puede considerar usar enlaces simbólicos, si tu SO (sistema operativo) y servidor Web lo permiten. Esta característica se puede habilitar estableciendo el valor de `linkAssets` en `true`.

```
return [
    // ...
```

```
    'components' => [
        'assetManager' => [
            'linkAssets' => true,
        ],
    ],
];
```

Con la anterior configuración, el gestor de assets creará un enlace simbólico a la ruta de origen del asset bundle cuando éste sea publicado. Esto es más rápido que copiar archivos y también asegura que siempre estén actualizados.

3.11.4. Los Asset Bundles más Comunes

El código del núcleo de Yii tiene definidos varios asset bundles. Entre ellos, los siguientes bundles son los más usados y pueden referenciarse en códigos de aplicaciones o extensiones.

- `yii\web\YiiAsset`: Principalmente incluye el archivo `yii.js` que implementa un mecanismo de organización de código JavaScript en los módulos. También proporciona soporte especial para los atributos `data-method` y `data-confirm` y otras característica útiles.
- `yii\web\JqueryAsset`: Incluye el archivo `jquery.js` desde el paquete Bower jQuery.
- `yii\bootstrap\BootstrapAsset`: Incluye el archivo CSS desde el framework Twitter Bootstrap.
- `yii\bootstrap\BootstrapPluginAsset`: Incluye el archivo JavaScript desde el framework Twitter Bootstrap para dar soporte a los plugins JavaScript de Bootstrap.
- `yii\jui\JuiAsset`: Incluye los archivos CSS y JavaScript desde la librería jQuery UI.

Si el código depende de jQuery, jQuery UI o Bootstrap, se pueden usar estos asset bundles predefinidos en lugar de crear versiones propias. Si la configuración predeterminada de estos bundles no satisface las necesidades, se puede personalizar como se describe en la subsección Personalización de Asset Bundles.

3.11.5. Conversión de Assets

En lugar de escribir código CSS y/o JavaScript directamente, los desarrolladores a menudo escriben código usando una sintaxis extendida y usan herramientas especiales para convertirlos en CSS/JavaScript. Por ejemplo, para código CSS se puede usar LESS[26] o SCSS[27]; y para JavaScript se puede usar TypeScript[28].

[26] https://lesscss.org
[27] https://sass-lang.com/
[28] https://www.typescriptlang.org/

Se pueden listar los archivos asset con sintaxis extendida (extended syntax) en `css` y `js` en un asset bundle. Por ejemplo:

```
class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.less',
    ];
    public $js = [
        'js/site.ts',
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

Cuando se registra uno de estos asset bundles en una vista, el `asset manager` ejecutará automáticamente las herramientas pre-procesadoras para convertir los assets de sintaxis extendidas reconocidas en CSS/JavaScript. Cuando la vista renderice finalmente una página, se incluirán los archivos CSS/JavaScript en la página, en lugar de los assets originales en sintaxis extendidas.

Yii usa las extensiones de archivo para identificar que sintaxis extendida se está usando. De forma predeterminada se reconocen las siguientes sintaxis y extensiones de archivo.

- LESS[29]: `.less`
- SCSS[30]: `.scss`
- Stylus[31]: `.styl`
- CoffeeScript[32]: `.coffee`
- TypeScript[33]: `.ts`

Yii se basa en las herramientas pre-procesadoras instalada para convertir los assets. Por ejemplo, para usar LESS[34] se debe instalar el comando pre-procesador `lessc`.

Se pueden personalizar los comandos de los pre-procesadores y las sintaxis extendidas soportadas configurando `yii\web\AssetManager::$converter` como en el siguiente ejemplo:

```
return [
    'components' => [
        'assetManager' => [
```

[29] https://lesscss.org/
[30] https://sass-lang.com/
[31] https://stylus-lang.com/
[32] https://coffeescript.org/
[33] https://www.typescriptlang.org/
[34] https://lesscss.org/

```
            'converter' => [
                'class' => 'yii\web\AssetConverter',
                'commands' => [
                    'less' => ['css', 'lessc {from} {to} --no-color'],
                    'ts' => ['js', 'tsc --out {to} {from}'],
                ],
            ],
        ],
    ],
];
```

En el anterior ejemplo se especifican las sintaxis extendidas soportadas a través de la propiedad `yii\web\AssetConverter::$commands`. Las claves del array son los nombres de extensión de archivo (sin el punto), y los valores del array las extensiones de archivo resultantes y los comandos para realizar la conversión de assets. Los tokens `{from}` y `{to}` en los comandos se reemplazarán por las rutas de origen de los archivos asset y las rutas de destino de los archivos asset.

> **Información:** Hay otras maneras de trabajar con las assets de sintaxis extendidas, además de la descrita anteriormente. Por ejemplo, se pueden usar herramientas generadoras tales como grunt[35] para monitorear y convertir automáticamente los assets de sintaxis extendidas. En este caso, se deben listar los archivos CSS/JavaScript resultantes en lugar de los archivos de originales.

3.11.6. Combinación y Compresión de Assets

Una página web puede incluir muchos archivos CSS y/o JavaScript. Para reducir el número de peticiones (requests) HTTP y el tamaño total de descarga de estos archivos, una práctica común es combinar y comprimir uno o varios archivos, y después incluir los archivos comprimidos en las páginas Web.

>Información: La combinación y compresión de assets es habitualmente necesario cuando una aplicación se encuentra en modo de producción. En modo de desarrollo, es más conveniente usar los archivos CSS/JavaScript originales por temas relacionados con el debugging.

En el siguiente ejemplo, se muestra una propuesta para combinar y comprimir archivos asset sin necesidad de modificar el código de la aplicación.

1. Buscar todos los asset bundles en la aplicación que se quieran combinar y comprimir.

2. Dividir estos bundles en uno o más grupos. Tenga en cuenta que cada bundle solo puede pertenecer a un único grupo.

[35] https://gruntjs.com/

3. Combina/Comprime los archivos CSS de cada grupo en un único archivo. Hace lo mismo para los archivos JavaScript.

4. Define un nuevo asset bundle para cada grupo:
   - Establece las propiedades `css` y `js` para que sean los archivos CSS y JavaScript combinados, respectivamente.
   - Personaliza los asset bundles en cada grupo configurando sus propiedades `css` y `js` para que sean el nuevo asset bundle creado para el grupo.

Usando este propuesta, cuando se registre un asset bundle en una vista, se genera un registro automático del nuevo asset bundle para el grupo al que pertenece el bundle original. Y como resultado, los archivos combinados/comprimidos se incluyen en la página, en lugar de los originales.

Un Example

Vamos a usar un ejemplo para explicar la propuesta anterior.

Asumiendo que la aplicación tenga dos páginas X e Y. La página X utiliza el asset bundle A, B y C mientras que la página Y usa los asset bundles B, C y D.

Hay dos maneras de dividir estos asset bundles. Uno es usar un único grupo que incluye todos los asset bundles, el otro es poner (A, B y C) en el Grupo X, y (B, C, D) en el grupo Y. ¿Cuál es mejor? El primero tiene la ventaja de que las dos páginas comparten los mismos archivos CSS y JavaScript combinados, que producen una caché HTTP más efectiva. Por otra parte, por el hecho de que un único grupo contenga todos los bundles, los archivos JavaScript serán más grandes y por tanto incrementan el tiempo de transmisión del archivo inicial. En este ejemplo, se usará la primera opción, ej., usar un único grupo que contenga todos los bundles.

> **Información:** Dividiendo los asset bundles en grupos no es una tarea trivial. Normalmente requiere un análisis de los datos del tráfico real de varios assets en diferentes páginas. Al principio, se puede empezar con un único grupo para simplificar.

Se pueden usar herramientas existentes (ej. Closure Compiler[36], YUI Compressor[37]) para combinar y comprimir todos los bundles. Hay que tener en cuenta que los archivos deben ser combinados en el orden que satisfaga las dependencias entre los bundles. Por ejemplo, si el Bundle A depende del B que depende a su vez de C y D, entonces, se deben listar los archivos asset empezando por C y D, seguidos por B y finalmente A.

[36] https://developers.google.com/closure/compiler/
[37] https://github.com/yui/yuicompressor/

Después de combinar y comprimir obtendremos un archivo CSS y un archivo JavaScript. Supongamos que se llaman `all-xyz.css` y `all-xyz.js`, donde `xyz` representa un timestamp o un hash que se usa para generar un nombre de archivo único para evitar problemas con la caché HTTP.

Ahora estamos en el último paso. Configurar el `asset manager` como en el siguiente ejemplo en la configuración de la aplicación:

```
return [
    'components' => [
        'assetManager' => [
            'bundles' => [
                'all' => [
                    'class' => 'yii\web\AssetBundle',
                    'basePath' => '@webroot/assets',
                    'baseUrl' => '@web/assets',
                    'css' => ['all-xyz.css'],
                    'js' => ['all-xyz.js'],
                ],
                'A' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'B' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'C' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'D' => ['css' => [], 'js' => [], 'depends' => ['all']],
            ],
        ],
    ],
];
```

Como se ha explicado en la subsección Personalización de Asset Bundles, la anterior configuración modifica el comportamiento predeterminado de cada bundle. En particular, el Bundle A, B, C y D ya no tendrán ningún archivo asset. Ahora todos dependen del bundle `all` que contiene los archivos combinados `all-xyz.css` y `all-xyz.js`. Por consiguiente, para la Página X, en lugar de incluir los archivos originales desde los bundles A, B y C, solo se incluirán los dos archivos combinados; pasa lo mismo con la Página Y.

Hay un último truco para hacer que el enfoque anterior se adapte mejor. En lugar de modificar directamente el archivo de configuración de la aplicación, se puede poner el array del personalización del bundle en un archivo separado y que se incluya condicionalmente este archivo en la configuración de la aplicación. Por ejemplo:

```
return [
    'components' => [
        'assetManager' => [
            'bundles' => require __DIR__ . '/' . (YII_ENV_PROD ?
            'assets-prod.php' : 'assets-dev.php'),
        ],
    ],
];
```

Es decir, el array de configuración del asset bundle se guarda en `asset-prod.php` para el modo de producción, y `assets-del.php` para los otros modos.

Uso del Comando

Yii proporciona un comando de consola llamado `asset` para automatizar el enfoque descrito.

Para usar este comando, primero se debe crear un archivo de configuración para describir que asset bundle se deben combinar y cómo se deben agrupar. Se puede usar el sub-comando `asset/template` para generar una plantilla primero y después modificarla para que se adapte a nuestras necesidades.

```
yii asset/template assets.php
```

El comando genera un archivo llamado `assets.php` en el directorio actual. El contenido de este archivo es similar al siguiente código:

```
<?php
/**
 * Configuration file for the "yii asset" console command.
 * Note that in the console environment, some path aliases like '@webroot'
and '@web' may not exist.
 * Please define these missing path aliases.
 */
return [
    // Ajustar comando/callback para comprimir los ficheros JavaScript:
    'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_file
    {to}',
    // Ajustar comando/callback para comprimir los ficheros CSS:
    'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o
    {to}',
    // La lista de assets bundles para comprimir:
    'bundles' => [
        // 'yii\web\YiiAsset',
        // 'yii\web\JqueryAsset',
    ],
    // Asset bundle para la salida de compresión:
    'targets' => [
        'all' => [
            'class' => 'yii\web\AssetBundle',
            'basePath' => '@webroot/assets',
            'baseUrl' => '@web/assets',
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
        ],
    ],
    // Configuración del Asset manager:
    'assetManager' => [
    ],
];
```

Se debe modificar este archivo para especificar que bundles plantea combinar en la opción `bundles`. En la opción `targets` se debe especificar como se deben dividir entre los grupos. Se puede especificar uno o más grupos, como se ha comentado.

> **Nota:** Debido a que los alias `@webroot` y `@web` no están disponibles en la aplicación de consola, se deben definir explícitamente en la configuración.

Los archivos JavaScript se combinan, comprimen y guardan en `js/all-{hash}.js` donde {hash} se reemplaza con el hash del archivo resultante.

Las opciones `jsCompressor` y `cssCompressor` especifican los comandos de consola o llamadas PHP (PHP callbacks) para realizar la combinación/compresión de JavaScript y CSS. De forma predeterminada Yii usa Closure Compiler[38] para combinar los archivos JavaScript y YUI Compressor[39] para combinar archivos CSS. Se deben instalar las herramientas manualmente o ajustar sus configuraciones para usar nuestras favoritas.

Con el archivo de configuración, se puede ejecutar el comando `asset` para combinar y comprimir los archivos asset y después generar un nuevo archivo de configuración de asset bundles `asset-prod.php`:

```
yii asset assets.php config/assets-prod.php
```

El archivo de configuración generado se puede incluir en la configuración de la aplicación, como se ha descrito en la anterior subsección.

> **Información:** Usar el comando `asset` no es la única opción de automatizar el proceso de combinación y compresión. Se puede usar la excelente herramienta de ejecución de tareas grunt[40] para lograr el mismo objetivo.

3.12. Extensiones

Las extensiones son paquetes de software redistribuibles diseñados especialmente para ser usados en aplicaciones Yii y proporcionar características listas para ser usadas. Por ejemplo, la extensión yiisoft/yii2-debug añade una practica barra de herramientas de depuración (debug toolbar) al final de cada página de la aplicación para ayudar a comprender más fácilmente como se han generado las páginas. Se pueden usar extensiones para acelerar el proceso de desarrollo. También se puede empaquetar código propio para compartir nuestro trabajo con otra gente.

> **Información:** Usamos el termino "extensión" para referirnos a los paquetes específicos de software Yii. Para propósitos generales los paquetes de software pueden usarse sin Yii, nos referiremos a ellos usando los términos "paquetes" (package) o "librerías" (library).

[38] https://developers.google.com/closure/compiler/
[39] https://github.com/yui/yuicompressor/
[40] https://gruntjs.com/

3.12.1. Uso de Extensiones

Para usar una extension, primero tenemos que instalarla. La mayoría de extensiones se usan como paquetes Composer[41] que se pueden instalar mediante los dos simples siguientes pasos:

1. modificar el archivo `composer.json` de la aplicación y especificar que extensiones (paquetes Composer) se quieren instalar.

2. ejecutar `composer install` para instalar las extensiones especificadas.

Hay que tener en cuenta que es necesaria la instalación de Composer[42] si no la tenemos instalada.

De forma predeterminada, Composer instala los paquetes registrados en Packagist[43] que es el repositorio más grande de paquetes Composer de código abierto (open source). Se pueden buscar extensiones en Packagist. También se puede crear un repositorio propio[44] y configurar Composer para que lo use. Esto es práctico cuando se desarrollan extensiones privadas que se quieran compartir a través de otros proyectos.

Las extensiones instaladas por Composer se almacenan en el directorio `BasePath/vendor`, donde `BasePath` hace referencia a la ruta base (base path) de la aplicación. Ya que Composer es un gestor de dependencias, cuando se instala un paquete, también se instalarán todos los paquetes de los que dependa.

Por ejemplo, para instalar la extensión `yiisoft/yii2-imagine`, modificamos el archivo `composer.json` como se muestra a continuación:

```
{
    // ...

    "require": {
        // ... otras dependencias

        "yiisoft/yii2-imagine": "~2.0.0"
    }
}
```

Después de la instalación, debemos encontrar el directorio `yiisoft/yii2-imagine` dentro del directorio `BasePath/vendor`. También debemos encontrar el directorio `imagine/imagine` que contiene sus paquetes dependientes instalados.

> **Información:** La extensión `yiisoft/yii2-imagine` es una extensión del núcleo (core) desarrollada y mantenida por el equipo de

[41] https://getcomposer.org/
[42] https://getcomposer.org/
[43] https://packagist.org/
[44] https://getcomposer.org/doc/05-repositories.md#repository

desarrollo de Yii. Todas las extensiones del núcleo se hospedan en Packagist[45] y son nombradas como `yiisoft/yii2-xyz`, donde `zyz` varia según la extensión.

Ahora ya podemos usar las extensiones instaladas como si fueran parte de nuestra aplicación. El siguiente ejemplo muestra como se puede usar la clase `yii\imagine\Image` proporcionada por la extensión `yiisoft/yii2-imagine`:

```
use Yii;
use yii\imagine\Image;

// genera una miniatura (thumbnail) de la imagen
Image::thumbnail('@webroot/img/test-image.jpg', 120, 120)
    ->save(Yii::getAlias('@runtime/thumb-test-image.jpg'), ['quality' =>
    50]);
```

Información: Las clases de extensiones se cargan automáticamente gracias a autocarga de clases de Yii.

Instalación Manual de Extensiones

En algunas ocasiones excepcionales es posible que tengamos que instalar alguna o todas las extensiones manualmente, en lugar de utilizar Composer. Para lograrlo, debemos:

1. descargar los archivos de la extensión y descomprimirlos en la carpeta `vendor`.
2. instalar la clase de autocarga proporcionada por las extensiones, si existe.
3. descargar e instalar todas las extensiones dependientes como siguiendo estas mismas instrucciones.

Si una extensión no proporciona clase de autocarga pero sigue el estándar PSR-4[46], se puede usar la clase de autocarga proporcionada por Yii para cargar automáticamente las clases de las extensiones. Todo lo que se tiene que hacer es declarar un alias de raíz (root) para las extensiones del directorio raíz. Por ejemplo, asumiendo que tenemos instalada una extensión en el directorio `vendor/mycompany/myext`, y las clases de extensión se encuentran en el namespace `myext`, entonces podemos incluir el siguiente código en nuestra configuración de aplicación:

```
[
    'aliases' => [
        '@myext' => '@vendor/mycompany/myext',
    ],
]
```

[45] https://packagist.org/
[46] https://www.php-fig.org/psr/psr-4/

3.12.2. Creación de Extensiones

Podemos considerar la creación de una extensión cuando tengamos la necesidad de compartir nuestro código. Cada extensión puede contener el código que se desee, puede ser una clase de ayuda (helper class), un widget, un módulo, etc.

Se recomienda crear una extensión como paquetes de Composer[47] para que sea se pueda instalarse más fácilmente por los otros usuarios, como se ha descrito en la anterior subsección.

Más adelante se encuentran los pasos básicos que deben seguirse para crear una extensión como paquete Composer.

1. Crear un proyecto para la extensión y alojarlo en un repositorio con VCS (Sistema de Control de Versiones), como puede ser github.com[48]. El trabajo de desarrollo y el mantenimiento debe efectuarse en este repositorio.

2. En el directorio raíz del repositorio debe encontrarse el archivo `composer.json` que es requerido por Composer. Se pueden encontrar más detalles en la siguiente subsección.

3. Registrar la extensión en un repositorio de Composer como puede ser Packagist[49], para que los otros usuarios puedan encontrarlo e instalarla mediante Composer.

Cada paquete de Composer tiene que tener un archivo `composer.json` en su directorio raíz. El archivo contiene los metadatos relacionados con el paquete. Se pueden encontrar especificaciones completas acerca de este fichero en el Manual de Composer[50]. El siguiente ejemplo muestra el archivo `composer.json` para la extensión `yiisoft/yii2-imagine`:

```
{
    // nombre del paquete
    "name": "yiisoft/yii2-imagine",

    // tipo de paquete
    "type": "yii2-extension",

    "description": "The Imagine integration for the Yii framework",
    "keywords": ["yii2", "imagine", "image", "helper"],
    "license": "BSD-3-Clause",
    "support": {
```

[47] https://getcomposer.org/
[48] https://github.com
[49] https://packagist.org/
[50] https://getcomposer.org/doc/01-basic-usage.md#composer-json-project-setup

```
        "issues":
        "https://github.com/yiisoft/yii2/issues?labels=ext%3Aimagine",
        "forum": "https://forum.yiiframework.com/",
        "wiki": "https://www.yiiframework.com/wiki/",
        "irc": "ircs://irc.libera.chat:6697/yii",
        "source": "https://github.com/yiisoft/yii2"
    },
    "authors": [
        {
            "name": "Antonio Ramirez",
            "email": "amigo.cobos@gmail.com"
        }
    ],

    // dependencias del paquete
    "require": {
        "yiisoft/yii2": "~2.0.0",
        "imagine/imagine": "v0.5.0"
    },

    // especificaciones de la autocarga de clases
    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

Nombre del Paquete Cada paquete Composer debe tener un nombre de paquete que identifique de entre todos los otros paquetes. El formato del nombre del paquete es `nombreProveedor/nombreProyecto`. Por ejemplo, el nombre de paquete `yiisoft/yii2-imagine`, el nombre del proveedor es `yiisoft` y el nombre del proyecto es `yii2-imagine`.

NO se puede usar el nombre de proveedor `yiisoft` ya que está reservado para el usarse para el código del núcleo (core) de Yii.

Recomendamos usar el prefijo `yii2-` al nombre del proyecto para paquetes que representen extensiones de Yii 2, por ejemplo, `minombre/yii2-miwidget`. Esto permite ver a los usuarios más fácilmente si un paquete es una extensión de Yii 2.

Tipo de Paquete Es importante que se especifique el tipo del paquete de la extensión como `yii2-extension` para que el paquete pueda ser reconocido como una extensión de Yii cuando se esté instalando.

Cuando un usuario ejecuta `composer install` para instalar una extensión, el archivo `vendor/yiisoft/extensions.php` se actualizará automáticamente para incluir la información acerca de la nueva extensión. Desde este archivo, las aplicaciones Yii pueden saber que extensiones están instaladas. (se puede acceder a esta información mediante `yii\base\Application::$extensions`).

Dependencias La extensión depende de Yii (por supuesto). Por ello se debe añadir (`yiisoft/yii2`) a la lista en la entrada `required` del archivo `composer.json`. Si la extensión también depende de otras extensiones o de terceras (third-party) librerías, también se deberán listar. Debemos asegurarnos de anotar las restricciones de versión apropiadas (ej. `1.*`, `@stable`) para cada paquete dependiente. Se deben usar dependencias estables en versiones estables de nuestras extensiones.

La mayoría de paquetes JavaScript/CSS se gestionan usando Bower[51] y/o NPM[52], en lugar de Composer. Yii utiliza el Composer asset plugin[53] para habilitar la gestión de estos tipos de paquetes a través de Composer. Si la extensión depende de un paquete Bower, se puede, simplemente, añadir la dependencia de el archivo `composer.json` como se muestra a continuación:

```
{
    // dependencias del paquete
    "require": {
        "bower-asset/jquery": ">=1.11.*"
    }
}
```

El código anterior declara que tela extensión depende del paquete Bower `jquery`. En general, se puede usar `bower-asset/NombrePaquete` para referirse al paquete en `composer.json`, y usar `npm-asset/NombrePaquete` para referirse a paquetes NPM. Cuando Composer instala un paquete Bower o NPM, de forma predeterminada los contenidos de los paquetes se instalarán en `@vendor/bower/NombrePaquete` y `@vendor/npm/Packages` respectivamente. Podemos hacer referencia a estos dos directorios usando los alias `@bower/NombrePaquete` and `@npm/NombrePaquete`.

Para obtener más detalles acerca de la gestión de assets, puede hacerse referencia a la sección Assets.

Autocarga de Clases Para que se aplique la autocarga a clases propias mediante la autocarga de clases de Yii o la autocarga de clases de Composer, debemos especificar la entrada `autoload` en el archivo `composer.json` como se puede ver a continuación:

```
{
    // ....

    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

[51] https://bower.io/
[52] https://www.npmjs.com/
[53] https://github.com/fxpio/composer-asset-plugin

Se pueden añadir una o más namespaces raíz y sus correspondientes rutas de archivo.

Cuando se instala la extensión en una aplicación, Yii creara un alias para todos los namespaces raíz, que harán referencia al directorio correspondiente del namespace. Por ejemplo, la anterior declaración `autoload` corresponderá a un alias llamado `@yii/imagine`.

Prácticas Recomendadas

Dado que las extensiones están destinadas a ser utilizadas por otras personas, a menudo es necesario hacer un esfuerzo extra durante el desarrollo. A continuación presentaremos algunas practicas comunes y recomendadas para la creación de extensiones de alta calidad.

Namespaces Para evitar colisiones de nombres y permitir que las clases usen la autocarga en extensiones propias, se deben usar namespaces y nombres de clase siguiendo el estándar PSR-4[54] o el estándar PSR-0[55].

Los namespaces de clases propias deben empezar por `NombreProveedor\NombreExtension` donde `NombreExtension` es similar al nombre del paquete pero este no debe contener el prefijo `yii2-`. Por ejemplo, para la extensión `yiisoft/yii2-imagine`, usamos `yii\imagine` como namespace para sus clases.

No se puede usar `yii`, `yii2` o `yiisoft` como nombre de proveedor. Estos nombres están reservados para usarse en el código del núcleo de Yii.

Clases de Bootstrapping A veces, se puede querer que nuestras extensiones ejecuten algo de código durante el proceso de bootstrapping de una aplicación. Por ejemplo, queremos que nuestra extensión responda a un evento `beginRequest` de la aplicación para ajustar alguna configuración de entorno. Aunque podemos indicar a los usuarios de la extensión que añadan nuestro gestor de eventos para que capture `beginRequest`, es mejor hacerlo automáticamente.

Para llevarlo a cabo, podemos crear una *clase de bootstrpping* para implementar `yii\base\BootstrapInterface`. Por ejemplo,

```
namespace myname\mywidget;

use yii\base\BootstrapInterface;
use yii\base\Application;

class MyBootstrapClass implements BootstrapInterface
{
    public function bootstrap($app)
    {
```

[54] https://www.php-fig.org/psr/psr-4/
[55] https://www.php-fig.org/psr/psr-0/

```
        $app->on(Application::EVENT_BEFORE_REQUEST, function () {
            // do something here
        });
    }
}
```

Entonces se tiene que añadir esta clase en la lista del archivo `composer.json` de la extensión propia como se muestra a continuación,

```
{
    // ...

    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

Cuando se instala la extensión en la aplicación, Yii automáticamente instancia la clase de bootstrapping y llama a su método `bootstrap()` durante el proceso de bootstrapping para cada petición.

Trabajar con Bases de Datos Puede darse el caso en que la extensión necesite acceso a bases de datos. No se debe asumir que las aplicaciones que usen la extensión siempre usarán `Yii::$db` como conexión de BBDD. Se debe crear una propiedad `db` para las clases que requieran acceso a BBDD. La propiedad permitirá a los usuarios de nuestra extensión elegir que conexión quieren que use nuestra extensión. Como ejemplo, se puede hacer referencia a la clase `yii\caching\DbCache` y observar como declara y utiliza la propiedad `db`.

Si nuestra extensión necesita crear tablas especificas en la BBDD o hacer cambios en el esquema de la BBDD, debemos:

- proporcionar migraciones para manipular el esquema de la BBDD, en lugar de utilizar archivos con sentencias SQL;
- intentar hacer las migraciones aplicables a varios Sistemas de Gestión de BBDD;
- evitar usar Active Record en las migraciones.

Uso de Assets Si nuestra aplicación es un widget o un módulo, hay posibilidades de que requiera assets para poder funcionar. Por ejemplo, un modulo puede mostrar algunas páginas de que contengan archivos JavaScript y/o CSS. Debido a que los archivos de las extensiones se encuentran en la misma ubicación y no son accesibles por la Web cuando se instalan en una aplicación, hay dos maneras de hacer los assets accesibles vía Web:

- pedir a los usuarios que copien manualmente los archivos assets en un directorio público de la Web.

- declarar un asset bundle dejar que el mecanismo de publicación se encargue automáticamente de copiar los archivos que se encuentren en el asset bundle a un directorio Web público.

Recomendamos el uso de la segunda propuesta para que la extensión sea más fácil de usar para usuarios. Se puede hacer referencia a la sección Assets para encontrar más detalles acerca de como trabajar con ellos.

Internacionalización y Localización Puede que las extensiones propias se usen en aplicaciones que den soporte a diferentes idiomas! Por ello, si nuestra extensión muestra contenido a los usuarios finales, se debe intentar internacionalizar y localizar la extensión. En particular,

- Si la extensión muestra mensajes destinados a usuarios finales, los mensajes deben mostrarse usando `Yii::t()` para que puedan ser traducidos. Los mensajes dirigidos a desarrolladores (como mensajes de excepciones internas) no necesitan ser traducidos.
- Si la extensión muestra números, fechas, etc., deben ser formateados usando `yii\i18n\Formatter` siguiendo las reglas de formato adecuadas.

Se pueden encontrar más detalles en la sección internacionalización.

Testing Para conseguir que las aplicaciones propias se ejecuten sin problemas y no causen problemas a otros usuarios, deben ejecutarse test a las extensiones antes de ser publicadas al público.

Se recomienda crear varios casos de prueba (test cases) para probar el código de nuestra extensión en lugar de ejecutar pruebas manuales. Cada vez que se vaya a lanzar una nueva versión, simplemente podemos ejecutar estos casos de prueba para asegurarnos de que todo está correcto. Yii proporciona soporte para testing que puede ayudar a escribir pruebas unitarias (unit tests), pruebas de aceptación (acceptance tests) y pruebas de funcionalidad (functionality tests), más fácilmente. Se pueden encontrar más detalles en la sección Testing.

Versiones Se debe asignar un número de versión cada vez que se lance una nueva distribución. (ej. `1.0.1`). Recomendamos seguir la práctica Versionamiento Semántico[56] para determinar que números se deben usar.

Lanzamientos Para dar a conocer nuestra extensión a terceras personas, debemos lanzara al público.

Si es la primera vez que se realiza un lanzamiento de una extensión, debemos registrarla en un repositorio Composer como puede ser Packagist[57]. Después de estos, todo lo que tenemos que hacer es crear una etiqueta (tag)

[56] `https://semver.org/lang/es/`
[57] `https://packagist.org/`

(ej. `v1.0.1`) en un repositorio con VCS (Sistema de Control de Versiones) y notificarle al repositorio Composer el nuevo lanzamiento. Entonces la gente podrá encontrar el nuevo lanzamiento y instalar o actualizar la extensión a mediante el repositorio Composer.

En los lanzamientos de una extensión, además de archivos de código, también se debe considerar la inclusión los puntos mencionados a continuación para facilitar a otra gente el uso de nuestra extensión:

- Un archivo léame (readme) en el directorio raíz: describe que hace la extensión y como instalarla y utilizarla. Recomendamos que se escriba en formato Markdown[58] y llamarlo `readme.md`.
- Un archivo de registro de cambios (changelog) en el directorio raíz: enumera que cambios se realizan en cada lanzamiento. El archivo puede escribirse en formato Markdown y llamarlo `changelog.md`.
- Un archivo de actualización (upgrade) en el directorio raíz: da instrucciones de como actualizar desde lanzamientos antiguos de la extensión. El archivo puede escribirse en formato Markdown y llamarlo `upgrade.md`.
- Tutoriales, demostraciones, capturas de pantalla, etc: son necesarios si nuestra extensión proporciona muchas características que no pueden ser detalladas completamente en el archivo `readme`.
- Documentación de API: el código debe documentarse debidamente para que otras personas puedan leerlo y entenderlo fácilmente. Más información acerca de documentación de código en archivo de Objetos de clase[59]

  Información: Los comentarios de código pueden ser escritos en formato Markdown. La extensión `yiisoft/yii2-apidoc` proporciona una herramienta para generar buena documentación de API basándose en los comentarios del código.

  Información: Aunque no es un requerimiento, se recomienda que la extensión se adhiera a ciertos estilos de codificación. Se puede hacer referencia a estilo de código del núcleo del framework[60] para obtener más detalles.

3.12.3. Extensiones del Núcleo

Yii proporciona las siguientes extensiones del núcleo que son desarrolladas y mantenidas por el equipo de desarrollo de Yii. Todas ellas están registradas en Packagist[61] y pueden ser instaladas fácilmente como se describe en la subsección Uso de Extensiones

[58] https://daringfireball.net/projects/markdown/

[59] https://github.com/yiisoft/yii2/blob/master/framework/base/BaseObject.php

[60] https://github.com/yiisoft/yii2/blob/master/docs/internals/core-code-style.md

[61] https://packagist.org/

- yiisoft/yii2-apidoc[62]:proporciona un generador de documentación de APIs extensible y de de alto rendimiento.
- yiisoft/yii2-authclient[63]:proporciona un conjunto de clientes de autorización tales como el cliente OAuth2 de Facebook, el cliente GitHub OAuth2.
- yiisoft/yii2-bootstrap[64]: proporciona un conjunto de widgets que encapsulan los componentes y plugins de Bootstrap[65].
- yiisoft/yii2-debug[66]: proporciona soporte de depuración para aplicaciones Yii. Cuando se usa esta extensión, aparece una barra de herramientas de depuración en la parte inferior de cada página. La extensión también proporciona un conjunto de páginas para mostrar información detallada de depuración.
- yiisoft/yii2-elasticsearch[67]: proporciona soporte para usar Elasticsearch[68]. Incluye soporte básico para realizar consultas/búsquedas y también implementa patrones de Active Record que permiten y permite guardar los `active records` en Elasticsearch.
- yiisoft/yii2-faker[69]: proporciona soporte para usar Faker[70] y generar datos automáticamente.
- yiisoft/yii2-gii[71]: proporciona un generador de código basado den Web altamente extensible y que puede usarse para generar modelos, formularios, módulos, CRUD, etc. rápidamente.
- yiisoft/yii2-httpclient[72]: provides an HTTP client.
- yiisoft/yii2-imagine[73]: proporciona funciones comunes de manipulación de imágenes basadas en Imagine[74].
- yiisoft/yii2-jui[75]: proporciona un conjunto de widgets que encapsulan las iteraciones y widgets de JQuery UI[76].
- yiisoft/yii2-mongodb[77]: proporciona soporte para utilizar MongoDB[78]. incluye características como consultas básicas, Active Record, migraciones, caching, generación de código, etc.

[62] https://github.com/yiisoft/yii2-apidoc
[63] https://github.com/yiisoft/yii2-authclient
[64] https://github.com/yiisoft/yii2-bootstrap
[65] https://getbootstrap.com/
[66] https://github.com/yiisoft/yii2-debug
[67] https://github.com/yiisoft/yii2-elasticsearch
[68] https://www.elastic.co/
[69] https://github.com/yiisoft/yii2-faker
[70] https://github.com/fzaninotto/Faker
[71] https://github.com/yiisoft/yii2-gii
[72] https://github.com/yiisoft/yii2-httpclient
[73] https://github.com/yiisoft/yii2-imagine
[74] https://imagine.readthedocs.org/
[75] https://github.com/yiisoft/yii2-jui
[76] https://jqueryui.com/
[77] https://github.com/yiisoft/yii2-mongodb
[78] https://www.mongodb.com/

- yiisoft/yii2-redis[79]: proporciona soporte para utilizar redis[80]. incluye características como consultas básicas, Active Record, caching, etc.
- yiisoft/yii2-smarty[81]: proporciona un motor de plantillas basado en Smarty[82].
- yiisoft/yii2-sphinx[83]: proporciona soporte para utilizar Sphinx[84]. incluye características como consultas básicas, Active Record, code generation, etc.
- yiisoft/yii2-swiftmailer[85]: proporciona características de envío de correos electrónicos basadas en swiftmailer[86].
- yiisoft/yii2-twig[87]: proporciona un motor de plantillas basado en Twig[88].

[79] https://github.com/yiisoft/yii2-redis
[80] https://redis.io/
[81] https://github.com/yiisoft/yii2-smarty
[82] https://www.smarty.net/
[83] https://github.com/yiisoft/yii2-sphinx
[84] https://sphinxsearch.com
[85] https://github.com/yiisoft/yii2-swiftmailer
[86] https://swiftmailer.symfony.com/
[87] https://github.com/yiisoft/yii2-twig
[88] https://twig.symfony.com/

Capítulo 4

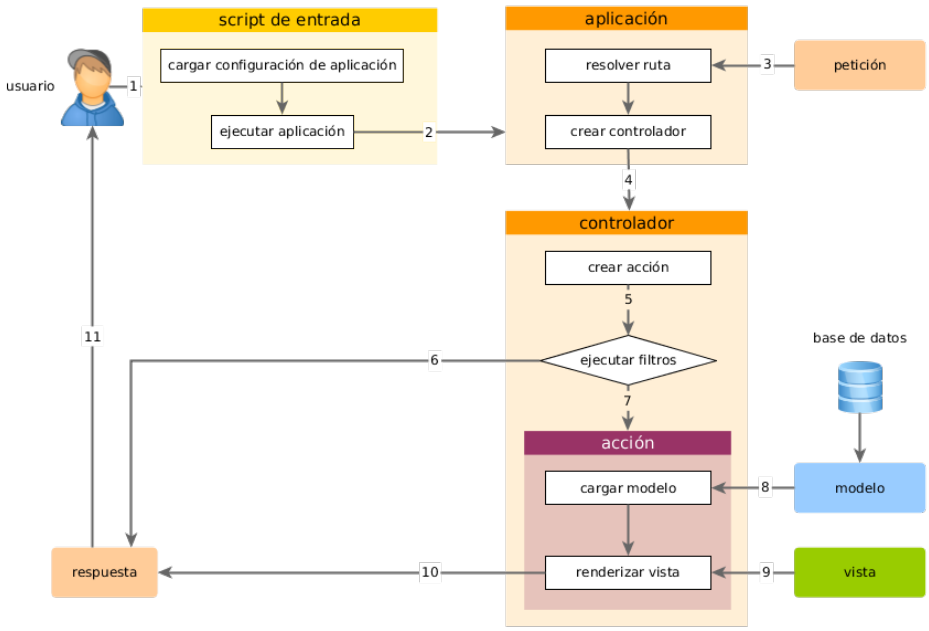
Gestión de las peticiones

4.1. Información General

Cada vez que una aplicación Yii gestiona una petición, se somete a un flujo de trabajo similar.

1. Un usuario hace una petición al script de entrada 'web/index.php'.
2. El script de entrada carga la configuración y crea una instancia de la aplicación para gestionar la petición.
3. La aplicación resuelve la ruta solicitada con la ayuda del componente petición de la aplicación.
4. La aplicación crea una instancia del controlador para gestionar la petición.
5. El controlador crea una instancia de la acción y ejecuta los filtros para la acción.
6. Si algún filtro falla, se cancela la acción.
7. Si pasa todos los filtros, se ejecuta la acción.
8. La acción carga un modelo de datos, posiblemente de la base de datos.
9. La acción renderiza una vista, proporcionándole el modelo de datos.
10. El resultado renderizado se devuelve al componente respuesta de la aplicación.
11. El componente respuesta envía el resultado renderizado al navegador del usuario.

El siguiente diagrama muestra como una aplicación gestiona una petición.



En esta sección, se describirá en detalle cómo funcionan algunos de estos pasos.

4.2. Bootstrapping

El Bootstrapping hace referencia al proceso de preparar el entorno antes de que una aplicación se inicie para resolver y procesar una petición entrante. El se ejecuta en dos lugares: el script de entrada y la aplicación.

En el script de entrada, se registran los cargadores automáticos de clase para diferentes librerías. Esto incluye el cargador automático de Composer a través de su fichero 'autoload.php' y del cargador automático de Yii a través del fichero de clase 'Yii'. El script de entrada después carga la configuración de la aplicación y crea una instancia de la aplicación.

El constructor de la aplicación, ejecuta el siguiente trabajo de bootstrapping:

Llama a `preInit()`, que configura algunas propiedades de alta prioridad de la aplicación, como `basePath`. Registra el `error handler`. Inicializa las propiedades de aplicación usando la configuración de la aplicación dada. Llama a `init()` que a su vez llama a `bootstrap()` para ejecutar componentes de bootstrapping. Incluye el archivo de manifiesto de extensiones 'vendor/yiisoft/extensions.php' Crea y ejecuta componentes de bootstrap declarados por las extensiones. Crea y ejecuta componentes de aplicación y/o módulos que se declaran en la propiedad bootstrap de la aplicación.

Debido a que el trabajo de bootstrapping se tiene que ejecutar antes

de gestionar *todas* las peticiones, es muy importante mantener este proceso ligero y optimizado lo máximo que sea posible.

Intenta no registrar demasiados componentes de bootstrapping. Un componente de bootstrapping sólo es necesario si tiene que interaccionar en todo el ciclo de vida de la gestión de la petición. Por ejemplo, si un modulo necesita registrar reglas de análisis de URL adicionales, se debe incluirse en la propiedad bootstrap para que la nueva regla de URL tenga efecto antes de que sea utilizada para resolver peticiones.

En modo de producción, hay que habilitar la cache bytecode, así como APC[1], para minimizar el tiempo necesario para incluir y analizar archivos PHP.

Algunas grandes aplicaciones tienen configuraciones de aplicación muy complejas que están dividida en muchos archivos de configuración más pequeños.

4.3. Enrutamiento y Creación de URLS

Cuando una aplicación Yii empieza a procesar una URL solicitada, lo primero que hace es convertir la URL en una ruta. Luego se usa la ruta para instanciar la acción de controlador correspondiente para gestionar la petición. A este proceso se le llama *enrutamiento*.

El proceso inverso se llama *creación de URLs*, y crea una URL a partir de una ruta dada y unos parámetros de consulta (query) asociados. Cuando posteriormente se solicita la URL creada, el proceso de enrutamiento puede resolverla y convertirla en la ruta original con los parámetros asociados.

La principal pieza encargada del enrutamiento y de la creación de URLs es `URL manager`, que se registra como el componente de aplicación `urlManager`. El `URL manager` proporciona el método `parseRequest()` para convertir una petición entrante en una ruta y sus parámetros asociados y el método `createUrl()` para crear una URL a partir de una ruta dada y sus parámetros asociados.

Configurando el componente `urlManager` en la configuración de la aplicación, se puede dotar a la aplicación de reconocimiento arbitrario de formatos de URL sin modificar el código de la aplicación existente. Por ejemplo, se puede usar el siguiente código para crear una URL para la acción `post/view`:

```
use yii\helpers\Url;

// Url::to() llama a UrlManager::createUrl() para crear una URL
$url = Url::to(['post/view', 'id' => 100]);
```

Dependiendo de la configuración de `urlManager`, la URL generada puede asemejarse a alguno de los siguientes (u otro) formato. Y si la URL creada se solicita posteriormente, se seguirá convirtiendo en la ruta original y los valores de los parámetros.

[1] https://www.php.net/manual/es/book.apc.php

```
/index.php?r=post/view&id=100
/index.php/post/100
/posts/100
```

4.3.1. Formatos de URL

El `URL manager` soporta dos formatos de URL: el formato predeterminado de URL y el formato URL amigable (pretty URL).

El formato de URL predeterminado utiliza un parámetro de consulta llamado `r` para representar la ruta y los parámetros normales de la petición para representar los parámetros asociados con la ruta. Por ejemplo, la URL `/index.php?r=post/view&id=100` representa la ruta `post/view` y 100 es el valor del parámetro `id` de la consulta. El formato predeterminado de URL no requiere ningún tipo de configuración para `URL manager` y funciona en cualquier configuración de servidor Web.

El formato de URL amigable utiliza la ruta adicional a continuación del nombre del script de entrada (entry script) para representar la ruta y los parámetros de consulta. Por ejemplo, La ruta en la URL `/index.php/post/100` es `/post/100` que puede representar la ruta `post/view` y el parámetro de consulta `id` 100 con una `URL rule` apropiada. Para poder utilizar el formato de URL amigable, se tendrán que diseñar una serie de `URL rules` de acuerdo con el requerimiento actual acerca de como deben mostrarse las URLs.

Se puede cambiar entre los dos formatos de URL conmutando la propiedad `enablePrettyUrl` del `URL manager` sin cambiar ningún otro código de aplicación.

4.3.2. Enrutamiento

El Enrutamiento involucra dos pasos. El primero, la petición (request) entrante se convierte en una ruta y sus parámetros de consulta asociados. En el segundo paso, se crea la correspondiente acción de controlador para la ruta convertida para que gestione la petición.

Cuando se usa el formato predefinido de URL, convertir una petición en una ruta es tan simple como obtener los valores del parámetro de consulta `GET` llamado `r`.

Cuando se usa el formato de URL amigable, el `URL manager` examinará las `URL rules` registradas para encontrar alguna que pueda convertir la petición en una ruta. Si no se encuentra tal regla, se lanzará una excepción de tipo `yii\web\NotFoundHttpException`.

Una vez que la petición se ha convertido en una ruta, es el momento de crear la acción de controlador identificada por la ruta. La ruta se desglosa en múltiples partes a partir de las barras que contenga. Por ejemplo, `site/index` será desglosado en `site` e `index`. Cada parte is un ID que puede hacer referencia a un modulo, un controlador o una acción. Empezando por la primera

parte de la ruta, la aplicación, sigue los siguientes pasos para generar (si los hay), controladores y acciones:

1. Establece la aplicación como el modulo actual.
2. Comprueba si el `controller map` del modulo actual contiene un ID actual. Si lo tiene, se creará un objeto controlador de acuerdo con la configuración del controlador encontrado en el mapa, y se seguirá el Paso 5 para gestionar la parte restante de la ruta.
3. Comprueba si el ID hace referencia a un modulo listado en la propiedad `modules` del módulo actual. Si está listado, se crea un modulo de acuerdo con la configuración encontrada en el listado de módulos, y se seguirá el Paso 2 para gestionar la siguiente parte de la ruta bajo el contexto de la creación de un nuevo módulo.
4. Trata el ID como si se tratara de un ID de controlador y crea un objeto controlador. Sigue el siguiente paso con la parte restante de la ruta.
5. El controlador busca el ID en su `action map`. Si lo encuentra, crea una acción de acuerdo con la configuración encontrada en el mapa. De otra forma, el controlador intenta crear una acción en linea definida por un método de acción correspondiente al ID actual.

Si ocurre algún error entre alguno de los pasos anteriores, se lanzará una excepción de tipo `yii\web\NotFoundHttpException`, indicando el fallo de proceso de enrutamiento.

Ruta Predeterminada

Cuando una petición se convierte en una ruta vacía, se usa la llamada *ruta predeterminada*. Por defecto, la ruta predeterminada es `site/index`, que hace referencia a la acción `index` del controlador `site`. Se puede personalizar configurando la propiedad `defaultRoute` de la aplicación en la configuración de aplicación como en el siguiente ejemplo:

```
[
    // ...
    'defaultRoute' => 'main/index',
];
```

Ruta

A veces, se puede querer poner la aplicación Web en modo de mantenimiento temporalmente y mostrar la misma pagina de información para todas las peticiones. Hay varias maneras de lograr este objetivo. Pero una de las maneras más simples es configurando la propiedad `yii\web\Application::$catchAll` como en el siguiente ejemplo de configuración de aplicación:

```
[
    // ...
        'catchAll' => ['site/offline'],
];
```

Con la anterior configuración, se usar la acción `site/offline` para gestionar todas las peticiones entrantes.

La propiedad `catchAll` debe tener un array cuyo primer elemento especifique una ruta, y el resto de elementos (pares nombre-valor) especifiquen los parámetros ligados a la acción.

4.3.3. Creación de URLs

Yii proporciona un método auxiliar (helper method) `yii\helpers\Url::to()` para crear varios tipos de URLs a partir de las rutas dadas y sus parámetros de consulta asociados. Por ejemplo,

```
use yii\helpers\Url;

// crea una URL para la ruta: /index.php?r=post/index
echo Url::to(['post/index']);

// crea una URL para la ruta con parámetros: /index.php?r=post/view&id=100
echo Url::to(['post/view', 'id' => 100]);

// crea una URL interna: /index.php?r=post/view&id=100#contentecho
Url::to(['post/view', 'id' => 100, '#' => 'content']);

// crea una URL absoluta: https://www.example.com/index.php?r=post/index
echo Url::to(['post/index'], true);

// crea una URL absoluta usando el esquema https:
https://www.example.com/index.php?r=post/index
echo Url::to(['post/index'], 'https');
```

Hay que tener en cuenta que en el anterior ejemplo, asumimos que se está usando el formato de URL predeterminado. Si habilita el formato de URL amigable, las URLs creadas serán diferentes, de acuerdo con las `URL rules` que se usen.

La ruta que se pasa al método `yii\helpers\Url::to()` es context sensitive. Esto quiere decir que puede ser una ruta *relativa* o una ruta *absoluta* que serán tipificadas de acuerdo con las siguientes reglas:

- Si una ruta es una cadena vacía, se usará la `route` solicitada actualmente.
- Si la ruta no contiene ninguna barra `/`, se considerará que se trata de un ID de acción del controlador actual y se le antepondrá el valor `uniqueId` del controlador actual.

- Si la ruta no tiene barra inicial, se considerará que se trata de una ruta relativa al modulo actual y se le antepondrá el valor `uniqueId` del modulo actual.

Por ejemplo, asumiendo que el modulo actual es `admin` y el controlador actual es `post`,

```
use yii\helpers\Url;

// la ruta solicitada: /index.php?r=admin/post/index
echo Url::to(['']);

// una ruta relativa solo con ID de acción: /index.php?r=admin/post/index
echo Url::to(['index']);

// una ruta relativa: /index.php?r=admin/post/index
echo Url::to(['post/index']);

// una ruta absoluta: /index.php?r=post/index
echo Url::to(['/post/index']);
```

El método `yii\helpers\Url::to()` se implementa llamando a los métodos `createUrl()` y `createAbsoluteUrl()` del `URL manager`. En las próximas sub-secciones, explicaremos como configurar el `URL manager` para personalizar el formato de las URLs generadas.

El método `yii\helpers\Url::to()` también soporta la creación de URLs NO relacionadas con rutas particulares. En lugar de pasar un array como su primer paramento, se debe pasar una cadena de texto. Por ejemplo,

```
use yii\helpers\Url;

// la URL solicitada actualmente: /index.php?r=admin/post/index
echo Url::to();

// una URL con alias: https://example.comYii::setAlias('@example',
'https://example.com/');
echo Url::to('@example');

// una URL absoluta: https://example.com/images/logo.gif
echo Url::to('/images/logo.gif', true);```
```

Además del método `to()`, la clase auxiliar `yii\helpers\Url` también proporciona algunos otros métodos de creación de URLs. Por ejemplo,

```
use yii\helpers\Url;

// URL de la página inicial: /index.php?r=site/index
echo Url::home();

// la URL base, útil si la aplicación se desarrolla en una sub-carpeta de la
carpeta raíz (root) Web
echo Url::base();
```

```
// la URL canónica de la actual URL solicitada// visitar
https://en.wikipedia.org/wiki/Canonical_link_element
echo Url::canonical();

// recuerda la actual URL solicitada y la recupera más tarde
requestsUrl::remember();
echo Url::previous();
```

4.3.4. Uso de URLs Amigables

Para utilizar URLs amigables, hay que configurar el componente `ulrManager` en la configuración de la aplicación como en el siguiente ejemplo:

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => true,
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

La propiedad `enablePrettyUrl` es obligatoria ya que alterna el formato de URL amigable. El resto de propiedades son opcionales. Sin embargo, la anterior configuración es la más común.

- `showScriptName`: esta propiedad determina si el script de entrada debe ser incluido en las URLs generadas. Por ejemplo, en lugar de crear una URL `/index.php/post/100`, estableciendo la propiedad con valor `true`, la URL que se generará sera `/post/100`.
- `enableStrictParsing`: esta propiedad determina si se habilita la conversión de petición estricta, si se habilita, la URL solicitada tiene que encajar al menos con uno de las `rules` para poder ser tratada como una petición valida, o se lanzará una `yii\web\NotFoundHttpException`. Si la conversión estricta esta deshabilitada, cuando ninguna de las `rules` coincida con la URL solicitada, la parte de información de la URL se tratará como si fuera la ruta solicitada.
- `rules`: esta propiedad contiene una lista de las reglas que especifican como convertir y crear URLs. Esta es la propiedad principal con la que se debe trabajar para crear URLs que satisfagan el formato de un requerimiento particular de la aplicación.

  Nota: Para ocultar el nombre del script de entrada en las URLs generadas, además de establecer el `showScriptName` a falso, pue-

de ser necesaria la configuración del servidor Web para que identifique correctamente que script PHP debe ejecutarse cuando se solicita una URL que no lo especifique. Si se usa el servidor Web Apache, se puede utilizar la configuración recomendada descrita en la sección de Instalación.

Reglas de URL

Una regla de URL es una instancia de `yii\web\UrlRule` o de una clase hija. Cada URL consiste en un patrón utilizado para cotejar la parte de información de ruta de las URLs, una ruta, y algunos parámetros de consulta. Una URL puede usarse para convertir una petición si su patrón coincide con la URL solicitada y una regla de URL pude usarse para crear una URL si su ruta y sus nombres de parámetros coinciden con los que se hayan dado.

Cuando el formato de URL amigables está habilitado, el `URL manager` utiliza las reglas de URL declaradas en su propiedad `rules` para convertir las peticiones entrantes y crear URLs. En particular, para convertir una petición entrante, el `URL manager` examina las reglas en el orden que se han declarado y busca la *primera* regla que coincida con la URL solicitada. La regla que coincide es la que se usa para convertir la URL en una ruta y sus parámetros asociados. De igual modo, para crear una URL, el `URL manager` busca la primera regla que coincida con la ruta dad y los parámetros y la utiliza para crear una URL.

Se pueden configurar las `yii\web\UrlManager::$rules` como un array con claves, siendo los patrones y las reglas sus correspondientes rutas. Cada pareja patrón-ruta construye una regla de URL. Por ejemplo, la siguiente configuración de configuración de `rules` declara dos reglas de URL. La primera regla coincide con una URL `posts` y la mapea a la ruta `post/index`. La segunda regla coincide con una URL que coincida con la expresión regular `post/(\d+)` y la mapea a la ruta `post/view` y el parámetro llamado `id`.

```
[
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

Información; El patrón en una regla se usa para encontrar coincidencias en la parte de información de la URL. Por ejemplo, la parte de información de la ruta `/index.php/post/100?source=ad` es `post/100` (la primera barra y la ultima son ignoradas) que coincide con el patrón `post/(\d+)`.

Entre la declaración de reglas de URL como pares de patrón-ruta, también se pueden declarar como arrays de configuración. Cada array de configuración se usa para configurar un único objeto de tipo regla de URL. Este proceso

se necesita a menudo cuando se quieren configurar otras propiedades de la regla de URL. Por ejemplo,

```
[
    // ... otras reglas de URL ...

    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

De forma predeterminada si no se especifica la opción `class` en la configuración de una regla, se utilizará la clase predeterminada `yii\web\UrlRule`.

Parameters Asociativos

Una regla de URL puede asociarse a una determinado grupo de parámetros de consulta que se hayan sido especificados en el patrón con el formato `<ParamName:RegExp>`, donde `ParamName` especifica el nombre del parámetro y `RegExp` es una expresión regular opcional que se usa para encontrar los valores de los parámetros. Si no se especifica `RegExp` significa que el parámetro debe ser una cadena de texto sin ninguna barra.

> **Nota:** Solo se pueden especificar expresiones regulares para los parámetros. La parte restante del patrón se considera texto plano.

Cuando se usa una regla para convertir una URL, esta rellenara los parámetros asociados con los valores que coincidan con las partes correspondientes de la URL, y estos parámetros serán accesibles posteriormente mediante `$_GET` por el componente de aplicación `request`. Cuando se usa una regla para crear una URL, esta obtendrá los valores de los parámetros proporcionados y los insertara donde se hayan declarado los parámetros.

Vamos a utilizar algunos ejemplos para ilustrar como funcionan los parámetros asociativos. Asumiendo que hemos declarado las siguientes tres URLs:

```
[
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
    'posts/<year:\d{4}>/<category>' => 'post/index',
]
```

Cuando se usen las reglas para convertir URLs:

- `/index.php/posts` se convierte en la ruta `post/index` usando la primera regla;
- `/index.php/posts/2014/php` se convierte en la ruta `post/index`, el parámetro `year` cuyo valor es 2014 y el parámetro `category` cuyo valor es `php` usando la tercera regla;

- `/index.php/post/100` se convierte en la ruta `post/view` y el parámetro `id` cuyo valor es 100 usando la segunda regla;
- `/index.php/posts/php` provocara una `yii\web\NotFoundHttpException` cuando `yii\web\UrlManager::$enableStrictParsing` sea `true`, ya que no coincide ninguno de los parámetros . Si `yii\web\UrlManager::$enableStrictParsing` es `false` (valor predeterminado), se devolverá como ruta la parte de información `posts/php`.

Y cuando las se usen las reglas para crear URLs:

- `Url::to(['post/index'])` genera `/index.php/posts` usando la primera regla;
- `Url::to(['post/index', 'year' => 2014, 'category' => 'php'])` genera `/index.php/posts/2014/php` usando la tercera regla;
- `Url::to(['post/view', 'id' => 100])` genera `/index.php/post/100` usando la segunda regla;
- `Url::to(['post/view', 'id' => 100, 'source' => 'ad'])` genera `/index.php/post/100?source=ad` usando la segunda regla. Debido a que el parámetro `source` no se especifica en la regla, se añade como un parámetro de consulta en la URL generada.
- `Url::to(['post/index', 'category' => 'php'])` genera `/index.php/post/index?category=php` no usa ninguna de las reglas. Hay que tener en cuenta que si no se aplica ninguna de las reglas, la URL se genera simplemente añadiendo la parte de información de la ruta y todos los parámetros como parte de la consulta.

Parametrización de Rutas

Se pueden incrustar nombres de parámetros en la ruta de una regla de URL. Esto permite a la regla de URL poder ser usada para que coincida con varias rutas. Por ejemplo, la siguiente regla incrusta los parámetros `controller` y `action` en las rutas.

```
[
    '<controller:(post|comment)>/<id:\d+>/<action:(create|update|delete)>'
    => '<controller>/<action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/view',
    '<controller:(post|comment)>s' => '<controller>/index',
]
```

Para convertir una URL `index.php/comment/100/create`, se aplicará la primera regla, que establece el parámetro `controller` a `comment` y el parámetro `action` a `create`. Por lo tanto la ruta `<controller>/<action>` se resuelve como `comment/create`.

Del mismo modo, para crear una URL para una ruta `comment/index`, se aplicará la tercera regla, que crea una URL `/index.php/comments`.

> **Información:** Mediante la parametrización de rutas es posible reducir el numero de reglas de URL e incrementar significativamente el rendimiento del `URL manager`.

De forma predeterminada, todos los parámetros declarados en una regla son requeridos. Si una URL solicitada no contiene un parámetro en particular, o si se esta creando una URL sin un parámetro en particular, la regla no se aplicará. Para establecer algunos parámetros como opcionales, se puede configurar la propiedad de `defaults` de una regla. Los parámetros listados en esta propiedad son opcionales y se usarán los parámetros especificados cuando estos no se proporcionen.

En la siguiente declaración de reglas, los parámetros `page` y `tag` son opcionales y cuando no se proporcionen, se usarán los valores 1 y cadena vacía respectivamente.

```
[
    // ... otras reglas ...
    [
        'pattern' => 'posts/<page:\d+>/<tag>',
        'route' => 'post/index',
        'defaults' => ['page' => 1, 'tag' => ''],
    ],
]
```

La regla anterior puede usarse para convertir o crear cualquiera de las siguientes URLs:

- `/index.php/posts`: `page` es 1, `tag` es ''.
- `/index.php/posts/2`: `page` es 2, `tag` es ''.
- `/index.php/posts/2/news`: `page` es 2, `tag` es `'news'`.
- `/index.php/posts/news`: `page` es 1, `tag` es `'news'`.

Sin usar ningún parámetro opcional, se tendrían que crear 4 reglas para lograr el mismo resultado.

Reglas con Nombres de Servidor

Es posible incluir nombres de servidores Web en los parámetros de las URLs. Esto es practico principalmente cuando una aplicación debe tener distintos comportamientos paro diferentes nombres de servidores Web. Por ejemplo, las siguientes reglas convertirán la URL `https://admin.example.com/login` en la ruta `admin/user/login` y `https://www.example.com/login` en `site/login`.

```
[
    'https://admin.example.com/login' => 'admin/user/login',
    'https://www.example.com/login' => 'site/login',
]
```

También se pueden incrustar parámetros en los nombres de servidor para extraer información dinámica de ellas. Por ejemplo, la siguiente regla convertirá la URL `https://en.example.com/posts` en la ruta `post/index` y el parámetro `language=en`.

```
[
    'http://<language:\w+>.example.com/posts' => 'post/index',
]
```

> **Nota:** Las reglas con nombres de servidor NO deben incluir el subdirectorio del script de entrada (entry script) en sus patrones. Por ejemplo, is la aplicación se encuentra en `https://www.example.com/sandbox/blog`, entonces se debe usar el patrón `https://www.example.com/posts` en lugar de `https://www.example.com/sandbox/blog/posts`. Esto permitirá que la aplicación se pueda desarrollar en cualquier directorio sin la necesidad de cambiar el código de la aplicación.

Sufijos de URL

Se puede querer añadir sufijos a las URLs para varios propósitos. Por ejemplo, se puede añadir `.html`a las URLs para que parezcan URLs para paginas HTML estáticas; también se puede querer añadir `.json` a las URLs para indicar el tipo de contenido que se espera encontrar en una respuesta (response). Se puede lograr este objetivo configurando la propiedad `yii\web\UrlManager::$suffix` como en el siguiente ejemplo de configuración de aplicación:

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => true,
            'suffix' => '.html',
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

La configuración anterior permitirá al `URL manager` reconocer las URLs solicitadas y a su vez crear URLs con el sufijo `.html`.

> **Consejo:** Se puede establecer `/` como el prefijo de URL para que las URLs finalicen con una barra.

> **Nota:** Cuando se configura un sufijo de URL, si una URL solicitada no tiene el sufijo, se considerará como una URL desconocida. Esta es una practica recomendada para SEO (optimización en motores de búsqueda).

A veces, se pueden querer usar sufijos diferentes para URLs diferentes. Esto se puede conseguir configurando la propiedad `suffix` de una regla de URL individual. Cuando una regla de URL tiene la propiedad establecida, anulará el sufijo estableciendo a nivel de `URL manager`. Por ejemplo, la siguiente configuración contiene una regla de URL personalizada que usa el sufijo `.json` en lugar del sufijo global `.html`.

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => true,
            'suffix' => '.html',
            'rules' => [
                // ...
                [
                    'pattern' => 'posts',
                    'route' => 'post/index',
                    'suffix' => '.json',
                ],
            ],
        ],
    ],
]
```

Métodos HTTP

Cuando se implementan APIs RESTful, normalmente se necesita que ciertas URLs se conviertan en otras de acuerdo con el método HTTP que se esté usando. Esto se puede hacer fácilmente prefijando los métodos HTTP soportados como los patrones de las reglas. Si una regla soporta múltiples métodos HTTP, hay que separar los nombres de los métodos con comas. Por ejemplo, la siguiente regla usa el mismo patrón `post/<id:\d+>` para dar soporte a diferentes métodos HTTP. Una petición para `PUT post/100` se convertirá en `post/create`, mientras que una petición `GET post/100` se convertirá en `post/view`.

```
[
    'PUT,POST post/<id:\d+>' => 'post/create',
    'DELETE post/<id:\d+>' => 'post/delete',
    'post/<id:\d+>' => 'post/view',
]
```

> **Nota:** Si una regla de URL contiene algún método HTTP en su patrón, la regla solo se usará para aplicar conversiones. Se omitirá cuando se llame a `URL manager` para crear URLs.

> **Consejo:** Para simplificar el enrutamiento en APIs RESTful, Yii proporciona una clase de reglas de URL `yii\rest\UrlRule` especial que es bastante eficiente y soporta ciertas características como pluralización de IDs de controladores. Para conocer más detalles, se puede visitar la sección Enrutamiento acerca de el desarrollo de APIs RESTful.

Personalización de Reglas

En los anteriores ejemplos, las reglas de URL se han declarado principalmente en términos de pares de patrón-ruta. Este es un método de acceso directo que se usa a menudo. En algunos escenarios, se puede querer personalizar la regla de URL configurando sus otras propiedades, tales como `yii\web\UrlRule::$suffix`. Esto se puede hacer usando una array completo de configuración para especificar una regla. El siguiente ejemplo se ha extraído de la subsección Sufijos de URL.

```
[
    // ... otras reglas de URL ...

    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

> **Información:** De forma predeterminada si no se especifica una opción `class` para una configuración de regla, se usará la clase predeterminada `yii\web\UrlRule`.

Adición de Reglas Dinámicamente

Las reglas de URL se pueden añadir dinámicamente en el `URL manager`. A menudo se necesita por módulos redistribubles que se encargan de gestionar sus propias reglas de URL. Para que las reglas añadidas dinámicamente tenga efecto durante el proceso de enrutamiento, se deben añadir durante la etapa bootstrapping. Para los módulos, esto significa que deben implementar `yii\base\BootstrapInterface` y añadir las reglas en el método `bootstrap()` como en el siguiente ejemplo:

```
public function bootstrap($app)
{
    $app->getUrlManager()->addRules([
        // declaraciones de reglas aquí
    ], false);
}
```

Hay que tener en cuenta se deben añadir estos módulos en `yii\web\Application::bootstrap()` para que puedan participar en el proceso de bootstrapping

Creación de Clases de Reglas

A pesar del hecho de que de forma predeterminada la clase `yii\web\UrlRule` lo suficientemente flexible para la mayoría de proyectos, hay situaciones en las que se tiene que crear una clase de reglas propia. Por ejemplo, en un sitio Web de un concesionario de coches, se puede querer dar soporte a las URL con el siguiente formato `/Manufacturer/Model`, donde tanto `Manufacturer` como `Model` tengan que coincidir con algún dato almacenado una tabla de la base de datos. De forma predeterminada, la clase regla no puede gestionar estas reglas ya que se base en patrones estáticos declarados.

Podemos crear la siguiente clase de reglas de URL para solucionar el problema.

```
namespace app\components;

use yii\web\UrlRuleInterface;
use yii\base\BaseObject;

class CarUrlRule extends BaseObject implements UrlRuleInterface
{

    public function createUrl($manager, $route, $params)
    {
        if ($route === 'car/index') {
            if (isset($params['manufacturer'], $params['model'])) {
                return $params['manufacturer'] . '/' . $params['model'];
            } elseif (isset($params['manufacturer'])) {
                return $params['manufacturer'];
            }
        }
        return false;  // no se aplica esta regla
    }

    public function parseRequest($manager, $request)
    {
        $pathInfo = $request->getPathInfo();
        if (preg_match('%^(\w+)(/(\w+))?$%', $pathInfo, $matches)) {
            // comprueba $matches[1] y $matches[3] para ver
            // si coincide con un *manufacturer* y un *model* en la base de
            datos
            // Si coinciden, establece $params['manufacturer'] y/o
            $params['model']
            // y devuelve ['car/index', $params]
        }
        return false;  // no se aplica la regla
    }
}
```

Y usa la nueva clase de regla en la configuración de `yii\web\UrlManager::$rules`:

```
[
    // ... otras reglas ...

    [
        'class' => 'app\components\CarUrlRule',
        // ... configura otras propiedades ...
    ],
]
```

4.3.5. Consideración del Rendimiento

Cuando se desarrolla una aplicación Web compleja, es importante optimizar las reglas de URL para que tarden el mínimo tiempo posible en convertir las peticiones y crear URLs.

Usando rutas parametrizadas se puede reducir el numero de reglas de URL que a su vez significa una mejor en el rendimiento.

Cuando se convierten o crean URLs, el `URL manager` examina las reglas de URL en el orden en que han sido declaradas. Por lo tanto, se debe tener en cuenta el orden de las reglas de URL y anteponer las reglas más especificas y/o las que se usen más a menudo.

Si algunas URLs comparten el mismo prefijo en sus patrones o rutas, se puede considerar usar `yii\web\GroupUrlRule` ya que puede ser más eficiente al ser examinado por `URL manager` como un grupo. Este suele ser el caso cuando una aplicación se compone por módulos, y cada uno tiene su propio conjunto de reglas con un ID de módulo para sus prefijos más comunes.

4.4. Peticiones

Las peticiones (requests) hechas a una aplicación son representadas como objetos `yii\web\Request` que proporcionan información como parámetros de la petición, cabeceras HTTP, cookies, etc. Dada una petición, se puede acceder al objeto request correspondiente a través del componente de aplicación `request` que, por defecto, es una instancia de `yii\web\Request`. En esta sección se describirá como hacer uso de este componente en las aplicaciones.

4.4.1. Parámetros de Request

Para obtener los parámetros de la petición, se puede llamar a los métodos `get()` y `post()` del componente `request`. Estos devuelven los valores de `$_GET` y `$_POST`, respectivamente. Por ejemplo:

```
$request = Yii::$app->request;
```

```php
$get = $request->get();
// equivalente a: $get = $_GET;

$id = $request->get('id');
// equivalente a: $id = isset($_GET['id']) ? $_GET['id'] : null;

$id = $request->get('id', 1);
// equivalente a: $id = isset($_GET['id']) ? $_GET['id'] : 1;

$post = $request->post();
// equivalente a: $post = $_POST;

$name = $request->post('name');
// equivalente a: $name = isset($_POST['name']) ? $_POST['name'] : null;

$name = $request->post('name', '');
// equivalente a: $name = isset($_POST['name']) ? $_POST['name'] : '';
```

> **Información:** En lugar de acceder directamente a `$_GET` y `$_POST` para obtener los parámetros de la petición, es recomendable que se obtengan mediante el componente `request` como en el ejemplo anterior. Esto facilitará la creación de tests ya que se puede simular una componente de request con datos de peticiones personalizados.

Cuando se implementan APIs RESTful, a menudo se necesita obtener parámetros enviados desde el formulario a través de PUT, PATCH u otros métodos de request. Se pueden obtener estos parámetros llamando a los métodos `yii\web\Request::getBodyParam()`. Por ejemplo:

```php
$request = Yii::$app->request;

// devuelve todos los parámetros
$params = $request->bodyParams;

// devuelve el parámetro "id"
$param = $request->getBodyParam('id');
```

> **Información:** A diferencia de los parámetros `GET`, los parámetros enviados desde el formulario a través de `POST`, `PUT`, `PATCH`, etc. se envían en el cuerpo de la petición. El componente `request` convierte los parámetros cuando se acceda a él a través de los métodos descritos anteriormente. Se puede personalizar la manera en como los parámetros se convierten configurando la propiedad `yii\web\Request::$parsers`.

4.4.2. Métodos de Request

Se puede obtener el método HTTP usado por la petición actual a través de la expresión `Yii::$app->request->method`. Se proporcionan un conjunto de

propiedades booleanas para comprobar si el método actual es de un cierto tipo. Por ejemplo:

```
$request = Yii::$app->request;

if ($request->isAjax) { // la request es una request AJAX }
if ($request->isGet)  { // el método de la request es GET }
if ($request->isPost) { // el método de la request es POST }
if ($request->isPut)  { // el método de la request es PUT }
```

4.4.3. URLs de Request

El componente `request` proporciona muchas maneras de inspeccionar la URL solicitada actualmente.

Asumiendo que la URL que se está solicitando es `https://example.com/admin/index.php/product?id=100`, se pueden obtener varias partes de la URL explicadas en los siguientes puntos:

- `url`: devuelve `/admin/index.php/product?id=100`, que es la URL sin la parte de información del host.
- `absoluteUrl`: devuelve `https://example.com/admin/index.php/product?id=100`, que es la URL entera, incluyendo la parte de información del host.
- `hostInfo`: devuelve `https://example.com`, que es la parte de información del host dentro de la URL.
- `pathInfo`: devuelve `/product`, que es la parte posterior al script de entrada y anterior al interrogante (query string)
- `queryString`: devuelve `id=100`, que es la parte posterior al interrogante.
- `baseUrl`: devuelve `/admin`, que es la parte posterior a la información del host y anterior al nombre de script de entrada.
- `scriptUrl`: devuelve `/admin/index.php`, que es la URL sin la información del la ruta ni la query string.
- `serverName`: devuelve `example.com`, que es el nombre del host dentro de la URL.
- `serverPort`: devuelve 80, que es el puerto que usa el servidor web.

4.4.4. Cabeceras HTTP

Se pueden obtener la información de las cabeceras HTTP a través de `header collection` devueltas por la propiedad `yii\web\Request::$headers`. Por ejemplo:

```
// $headers es un objeto de yii\web\HeaderCollection
$headers = Yii::$app->request->headers;

// devuelve el valor Accept de la cabecera
$accept = $headers->get('Accept');

if ($headers->has('User-Agent')) { // la cabecera contiene un User-Agent }
```

El componente `request` también proporciona soporte para acceder rápidamente a las cabeceras usadas más comúnmente, incluyendo:

- `userAgent`: devuelve el valor de la cabecera `User-Agen`.
- `contentType`: devuelve el valor de la cabecera `Content-Type` que indica el tipo MIME de los datos del cuerpo de la petición.
- `acceptableContentTypes`: devuelve los tipos de contenido MIME aceptado por los usuarios, ordenados por puntuación de calidad. Los que tienen mejor puntuación, se devolverán primero.
- `acceptableLanguages`: devuelve los idiomas aceptados por el usuario. Los idiomas devueltos son ordenados según su orden de preferencia. El primer elemento representa el idioma preferido.

Si la aplicación soporta múltiples idiomas y se quiere mostrar las páginas en el idioma preferido por el usuario, se puede usar el método de negociación de idioma `yii\web\Request::getPreferredLanguage()`. Este método obtiene una lista de idiomas soportados por la aplicación, comparados con `acceptableLanguages`, y devuelve el idioma más apropiado.

> **Consejo:** También se puede usar el filtro `ContentNegotiator` para determinar diatónicamente el content type y el idioma que debe usarse en la respuesta. El filtro implementa la negociación de contenido en la parte superior de las propiedades y métodos descritos anteriormente.

4.4.5. Información del cliente

Se puede obtener el nombre del host y la dirección IP de la máquina cliente a través de `userHost` y `userIP`, respectivamente. Por ejemplo:

```
$userHost = Yii::$app->request->userHost;
$userIP = Yii::$app->request->userIP;
```

4.5. Respuestas

Cuando una aplicación finaliza la gestión de una petición (request), genera un objeto `response` y lo envía al usuario final. El objeto response contiene información tal como el código de estado (status code) HTTP, las cabeceras (headers) HTTP y el cuerpo (body). El objetivo final del desarrollo de una aplicación Web es esencialmente construir objetos response para varias peticiones.

En la mayoría de casos principalmente se debe tratar con componentes de aplicación de tipo `response` que, por defecto, son una instancia de `yii\web\Response`. Sin embargo, Yii permite crear sus propios objetos `response` y enviarlos al usuario final tal y como se explica a continuación.

En esta sección, se describirá como generar y enviar respuestas a usuarios finales.

4.5.1. Códigos de Estado

Una de las primeras cosas que debería hacerse cuando se genera una respuesta es indicar si la petición se ha gestionado correctamente. Esto se indica asignando la propiedad `yii\web\Response::$statusCode` a la que se le puede asignar cualquier valor válido dentro de los códigos de estado HTTP[2]. Por ejemplo, para indicar que la petición se ha gestionado correctamente, se puede asignar el código de estado a 200, como en el siguiente ejemplo:

```
Yii::$app->response->statusCode = 200;
```

Sin embargo, en la mayoría de casos nos es necesario asignar explícitamente el código de estado. Esto se debe a que el valor por defecto de `yii\web\Response::$statusCode` es 200. Y si se quiere indicar que la petición ha fallado, se puede lanzar una excepción HTTP apropiada como en el siguiente ejemplo:

```
throw new \yii\web\NotFoundHttpException;
```

Cuando el error handler captura una excepción, obtendrá el código de estado de la excepción y lo asignará a la respuesta. En el caso anterior, la excepción `yii\web\NotFoundHttpException` está asociada al estado HTTP 404. En Yii existen las siguientes excepciones predefinidas.

- `yii\web\BadRequestHttpException`: código de estado 400.
- `yii\web\ConflictHttpException`: código de estado 409.
- `yii\web\ForbiddenHttpException`: código de estado 403.
- `yii\web\GoneHttpException`: código de estado 410.
- `yii\web\MethodNotAllowedHttpException`: código de estado 405.
- `yii\web\NotAcceptableHttpException`: código de estado 406.
- `yii\web\NotFoundHttpException`: código de estado 404.
- `yii\web\ServerErrorHttpException`: código de estado 500.
- `yii\web\TooManyRequestsHttpException`: código de estado 429.
- `yii\web\UnauthorizedHttpException`: código de estado 401.
- `yii\web\UnsupportedMediaTypeHttpException`: código de estado 415.

Si la excepción que se quiere lanzar no se encuentra en la lista anterior, se puede crear una extendiendo `yii\web\HttpException`, o directamente lanzando un código de estado, por ejemplo:

```
throw new \yii\web\HttpException(402);
```

4.5.2. Cabeceras HTTP

Se puede enviar cabeceras HTTP modificando el `header collection` en el componente `response`. Por ejemplo:

[2] https://tools.ietf.org/html/rfc2616#section-10

```php
$headers = Yii::$app->response->headers;

// añade una cabecera Pragma. Las cabeceras Pragma existentes NO se
sobrescribirán.
$headers->add('Pragma', 'no-cache');

// asigna una cabecera Pragma. Cualquier cabecera Pragma existente será
descartada.
$headers->set('Pragma', 'no-cache');

// Elimina las cabeceras Pragma y devuelve los valores de las eliminadas en
un array
$values = $headers->remove('Pragma');
```

> **Información:** Los nombres de las cabeceras case insensitive, es decir, no discriminan entre mayúsculas y minúsculas. Además, las nuevas cabeceras registradas no se enviarán al usuario hasta que se llame al método `yii\web\Response::send()`.

4.5.3. Cuerpo de la Respuesta

La mayoría de las respuestas deben tener un cuerpo que contenga el contenido que se quiere mostrar a los usuarios finales.

Si ya se tiene un texto de cuerpo con formato, se puede asignar a la propiedad `yii\web\Response::$content` del response. Por ejemplo:

```php
Yii::$app->response->content = 'hello world!';
```

Si se tiene que dar formato a los datos antes de enviarlo al usuario final, se deben asignar las propiedades `format` y `data`. La propiedad `format` especifica que formato debe tener `data`. Por ejemplo:

```php
$response = Yii::$app->response;
$response->format = \yii\web\Response::FORMAT_JSON;
$response->data = ['message' => 'hello world'];
```

Yii soporta a los siguientes formatos de forma predeterminada, cada uno de ellos implementado por una clase `formatter`. Se pueden personalizar los formatos o añadir nuevos sobrescribiendo la propiedad `yii\web\Response::$formatters`.

- `HTML`: implementado por `yii\web\HtmlResponseFormatter`.
- `XML`: implementado por `yii\web\XmlResponseFormatter`.
- `JSON`: implementado por `yii\web\JsonResponseFormatter`.
- `JSONP`: implementado por `yii\web\JsonResponseFormatter`.

Mientras el cuerpo de la respuesta puede ser mostrado de forma explicita como se muestra a en el anterior ejemplo, en la mayoría de casos se puede asignar implícitamente por el valor de retorno de los métodos de acción. El siguiente, es un ejemplo de uso común:

```
public function actionIndex()
{
    return $this->render('index');
}
```

La acción `index` anterior, devuelve el resultado renderizado de la vista `index`. El valor devuelto será recogido por el componente `response`, se le aplicará formato y se enviará al usuario final.

Por defecto, el formato de respuesta es `HTML`, sólo se debe devolver un string en un método de acción. Si se quiere usar un formato de respuesta diferente, se debe asignar antes de devolver los datos. Por ejemplo:

```
public function actionInfo()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
    return [
        'message' => 'hello world',
        'code' => 100,
    ];
}
```

Como se ha mencionado, además de usar el componente de aplicación `response` predeterminado, también se pueden crear objetos response propios y enviarlos a los usuarios finales. Se puede hacer retornando un objeto en el método de acción, como en el siguiente ejemplo:

```
public function actionInfo()
{
    return \Yii::createObject([
        'class' => 'yii\web\Response',
        'format' => \yii\web\Response::FORMAT_JSON,
        'data' => [
            'message' => 'hello world',
            'code' => 100,
        ],
    ]);
}
```

> **Nota:** Si se crea un objeto response propio, no se podrán aprovechar las configuraciones asignadas para el componente `response` en la configuración de la aplicación. Sin embargo, se puede usar la inyección de dependencias para aplicar la configuración común al nuevo objeto response.

4.5.4. Redirección del Navegador

La redirección del navegador se basa en el envío de la cabecera HTTP `Location`. Debido a que esta característica se usa comúnmente, Yii proporciona soporte especial para ello.

Se puede redirigir el navegador a una URL llamando al método `yii\web\Response::redirect()`. El método asigna la cabecera de `Location` apropiada con la URL proporcionada y devuelve el objeto response él mismo. En un método de acción, se puede acceder a él mediante el acceso directo `yii\web\Controller::redirect()` como en el siguiente ejemplo:

```
public function actionOld()
{
    return $this->redirect('https://example.com/new', 301);
}
```

En el ejemplo anterior, el método de acción devuelve el resultado del método `redirect()`. Como se ha explicado antes, el objeto response devuelto por el método de acción se usará como respuesta enviándola al usuario final.

En otros sitios que no sean los métodos de acción, se puede llamar a `yii\web\Response::redirect()` directamente seguido por una llamada al método `yii\web\Response::send()` para asegurar que no habrá contenido extra en la respuesta.

```
\Yii::$app->response->redirect('https://example.com/new', 301)->send();
```

> **Información:** De forma predeterminada, el método `yii\web\Response::redirect()` asigna el estado de respuesta al código de estado 302 que indica al navegador que recurso solicitado está *temporalmente* alojado en una URI diferente. Se puede enviar un código de estado 301 para expresar que el recurso se ha movido de forma *permanente*.

Cuando la petición actual es de una petición AJAX, el hecho de enviar una cabecera `Location` no causará una redirección del navegador automática. Para resolver este problema, el método `yii\web\Response::redirect()` asigna una cabecera `X-Redirect` con el valor de la URL de redirección. En el lado del cliente se puede escribir código JavaScript para leer la esta cabecera y redireccionar el navegador como corresponda.

> **Información:** Yii contiene el archivo JavaScript `yii.js` que proporciona un conjunto de utilidades comunes de JavaScript, incluyendo la redirección de navegador basada en la cabecera `X-Redirect`. Por tanto, si se usa este fichero JavaScript (registrándolo *asset bundle* `yii\web\YiiAsset`), no se necesitará escribir nada más para tener soporte en redirecciones AJAX.

4.5.5. Enviar Archivos

Igual que con la redirección, el envío de archivos es otra característica que se basa en cabeceras HTTP especificas. Yii proporciona un conjunto de métodos para dar soporte a varias necesidades del envío de ficheros. Todos ellos incorporan soporte para el rango de cabeceras HTTP.

- `yii\web\Response::sendFile()`: envía un fichero existente al cliente.
- `yii\web\Response::sendContentAsFile()`: envía un string al cliente como si fuera un archivo.
- `yii\web\Response::sendStreamAsFile()`: envía un *file stream* existente al cliente como si fuera un archivo.

Estos métodos tienen la misma firma de método con el objeto response como valor de retorno. Si el archivo que se envía es muy grande, se debe considerar usar `yii\web\Response::sendStreamAsFile()` porque es más efectivo en términos de memoria. El siguiente ejemplo muestra como enviar un archivo en una acción de controlador.

```
public function actionDownload()
{
    return \Yii::$app->response->sendFile('ruta/del/fichero.txt');
}
```

Si se llama al método de envío de ficheros fuera de un método de acción, también se debe llamar al método `yii\web\Response::send()` después para asegurar que no se añada contenido extra a la respuesta.

```
\Yii::$app->response->sendFile('ruta/del/fichero.txt')->send();
```

Algunos servidores Web tienen un soporte especial para enviar ficheros llamado *X-Sendfile*. La idea es redireccionar la petición para un fichero a un servidor Web que servirá el fichero directamente. Como resultado, la aplicación Web puede terminar antes mientras el servidor Web envía el fichero. Para usar esta funcionalidad, se puede llamar a `yii\web\Response::xSendFile()`. La siguiente lista resume como habilitar la característica `X-Sendfile` para algunos servidores Web populares.

- Apache: X-Sendfile[3]
- Lighttpd v1.4: X-LIGHTTPD-send-file[4]
- Lighttpd v1.5: X-Sendfile[5]
- Nginx: X-Accel-Redirect[6]
- Cherokee: X-Sendfile and X-Accel-Redirect[7]

4.5.6. Enviar la Respuesta

El contenido en una respuesta no se envía al usuario hasta que se llama al método `yii\web\Response::send()`. De forma predeterminada, se llama a este método automáticamente al final de `yii\base\Application::run()`.

[3] https://tn123.org/mod_xsendfile
[4] https://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
[5] https://redmine.lighttpd.net/projects/lighttpd/wiki/X-LIGHTTPD-send-file
[6] https://www.nginx.com/resources/wiki/start/topics/examples/xsendfile/
[7] https://www.cherokee-project.com/doc/other_goodies.html#x-sendfile

Sin embargo, se puede llamar explícitamente a este método forzando el envío de la respuesta inmediatamente.

El método `yii\web\Response::send()` sigue los siguientes pasos para enviar una respuesta:

1. Lanza el evento `yii\web\Response::EVENT_BEFORE_SEND`.
2. Llama a `yii\web\Response::prepare()` para convertir el `response data` en `response content`.
3. Lanza el evento `yii\web\Response::EVENT_AFTER_PREPARE`.
4. Llama a `yii\web\Response::sendHeaders()` para enviar las cabeceras HTTP registradas.
5. Llama a `yii\web\Response::sendContent()` para enviar el contenido del cuerpo de la respuesta.
6. Lanza el evento `yii\web\Response::EVENT_AFTER_SEND`.

Después de llamar a `yii\web\Response::send()` por primera vez, cualquier llamada a este método será ignorada. Esto significa que una vez se envíe una respuesta, no se le podrá añadir más contenido.

Como se puede observar, el método `yii\web\Response::send()` lanza varios eventos útiles. Al responder a estos eventos, es posible ajustar o decorar la respuesta.

4.6. Sesiones (Sessions) y Cookies

Las sesiones y las cookies permiten la persistencia de datos a través de múltiples peticiones de usuario. En PHP plano, debes acceder a ellos a través de las variables globales `$_SESSION` y `$_COOKIE`, respectivamente. Yii encapsula las sesiones y las cookies como objetos y por lo tanto te permite acceder a ellos de manera orientada a objetos con estupendas mejoras adicionales.

4.6.1. Sesiones

Como las peticiones y las respuestas, puedes acceder a las sesiones vía el componente de la aplicación `session` el cual es una instancia de `yii\web\Session`, por defecto.

Abriendo y cerrando sesiones

Para abrir y cerrar una sesión, puedes hacer lo siguiente:

```php
$session = Yii::$app->session;

// comprueba si una sesión está ya abierta
if ($session->isActive) ...

// abre una sesión
$session->open();

// cierra una sesión
$session->close();

// destruye todos los datos registrados por la sesión.
$session->destroy();
```

Puedes llamar a `open()` y `close()` múltiples veces sin causar errores. Esto ocurre porque internamente los métodos verificarán primero si la sesión está ya abierta.

Accediendo a los datos de sesión

Para acceder a los datos almacenados en sesión, puedes hacer lo siguiente:

```php
$session = Yii::$app->session;

// devuelve una variable de sesión. Los siguientes usos son equivalentes:
$language = $session->get('language');
$language = $session['language'];
$language = isset($_SESSION['language']) ? $_SESSION['language'] : null;

// inicializa una variable de sesión. Los siguientes usos son equivalentes:
$session->set('language', 'en-US');
$session['language'] = 'en-US';
$_SESSION['language'] = 'en-US';

// remueve la variable de sesión. Los siguientes usos son equivalentes:
$session->remove('language');
unset($session['language']);
unset($_SESSION['language']);

// comprueba si una variable de sesión existe. Los siguientes usos son
equivalentes:
if ($session->has('language')) ...
if (isset($session['language'])) ...
if (isset($_SESSION['language'])) ...

// recorre todas las variables de sesión. Los siguientes usos son
equivalentes:
foreach ($session as $name => $value) ...
foreach ($_SESSION as $name => $value) ...
```

> **Información:** Cuando accedas a los datos de sesión a través del componente `session`, una sesión será automáticamente abierta si

> no lo estaba antes. Esto es diferente accediendo a los datos de sesión a través de `$_SESSION`, el cual requiere llamar explícitamente a `session_start()`.

Cuando trabajas con datos de sesiones que son arrays, el componte `session` tiene una limitación que te previene directamente de modificar un elemento del array. Por ejemplo,

```
$session = Yii::$app->session;

// el siguiente código no funciona
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// el siguiente código funciona:
$session['captcha'] = [
    'number' => 5,
    'lifetime' => 3600,
];

// el siguiente código también funciona:
echo $session['captcha']['lifetime'];
```

Puedes usar las siguientes soluciones para arreglar este problema:

```
$session = Yii::$app->session;

// directamente usando $_SESSION (asegura te de que
Yii::$app->session->open() ha sido llamado)
$_SESSION['captcha']['number'] = 5;
$_SESSION['captcha']['lifetime'] = 3600;

// devuelve el valor del array, lo modifica y a continuación lo guarda
$captcha = $session['captcha'];
$captcha['number'] = 5;
$captcha['lifetime'] = 3600;
$session['captcha'] = $captcha;

// usa un ArrayObject en vez de un array
$session['captcha'] = new \ArrayObject;
...
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// almacena los datos en un array con un prefijo común para las claves
$session['captcha.number'] = 5;
$session['captcha.lifetime'] = 3600;
```

Para un mejor rendimiento y legibilidad del código, recomendamos la última solución. Es decir, en vez de almacenar un array como una única variable de sesión, almacena cada elemento del array como una variable de sesión que comparta el mismo prefijo clave con otros elementos del array.

Personalizar el almacenamiento de sesión

Por defecto la clase `yii\web\Session` almacena los datos de sesión como ficheros en el servidor. Yii también provee de las siguientes clases de sesión que implementan diferentes almacenamientos de sesión:

- `yii\web\DbSession`: almacena los datos de sesión en una tabla en la base de datos.
- `yii\web\CacheSession`: almacena los datos de sesión en una caché con la ayuda de la configuración del componente caché.
- `yii\redis\Session`: almacena los datos de sesión usando redis[8] como medio de almacenamiento.
- `yii\mongodb\Session`: almacena los datos de sesión en MongoDB[9].

Todas estas clases de sesión soportan los mismos métodos de la API. Como consecuencia, puedes cambiar el uso de diferentes almacenamientos de sesión sin la necesidad de modificar el código de tu aplicación que usa sesiones.

> **Nota:** si quieres acceder a los datos de sesión vía `$_SESSION` mientras estás usando un almacenamiento de sesión personalizado, debes asegurar te que la sesión está ya empezada por `yii\web\Session::open()`. Esto ocurre porque los manipuladores de almacenamiento de sesión personalizado son registrados sin este método.

Para aprender como configurar y usar estas clases de componentes, por favor consulte la documentación de la API. Abajo está un ejemplo que muestra como configurar `yii\web\DbSession` en la configuración de la aplicación para usar una tabla en la base de datos como almacenamiento de sesión:

```
return [
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',
            // 'db' => 'mydb',  // el identificador del componente de
            aplicación DB connection. Por defecto'db'.
            // 'sessionTable' => 'my_session', // nombre de la tabla de
            sesión. Por defecto 'session'.
        ],
    ],
];
```

También es necesario crear la siguiente tabla de la base de datos para almacenar los datos de sesión:

```
CREATE TABLE session
(
```

[8] https://redis.io/
[9] https://www.mongodb.com/

```
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

donde 'BLOB' se refiere al BLOB-type de tu DBMS preferida. Abajo está el tipo BLOB que puedes usar para algunos DBMS populares:

- MySQL: LONGBLOB
- PostgreSQL: BYTEA
- MSSQL: BLOB

> **Nota:** De acuerdo con la configuración de php.ini `session.hash_function`, puedes necesitar ajustar el tamaño de la columna `id`. Por ejemplo, si `session.hash_function=sha256`, deberías usar el tamaño 64 en vez de 40.

Flash Data

Flash data es una clase especial de datos de sesión que, una vez se inicialice en la primera petición, estará sólo disponible durante la siguiente petición y automáticamente se borrará después. Flash data es comúnmente usado para implementar mensajes que deberían ser mostrados una vez a usuarios finales, tal como mostrar un mensaje de confirmación después de que un usuario envíe un formulario con éxito.

Puedes inicializar y acceder a flash data a través del componente de aplicación `session`. Por ejemplo,

```
$session = Yii::$app->session;

// Petición #1
// inicializa el mensaje flash nombrado como "postDeleted"
$session->setFlash('postDeleted', 'You have successfully deleted your
post.');

// Petición #2
// muestra el mensaje flash nombrado "postDeleted"
echo $session->getFlash('postDeleted');

// Petición #3
// $result será `false` ya que el mensaje flash ha sido borrado
automáticamente
$result = $session->hasFlash('postDeleted');
```

Al igual que los datos de sesión regulares, puede almacenar datos arbitrarios como flash data.

Cuando llamas a `yii\web\Session::setFlash()`, sobrescribirá cualquier Flash data que tenga el mismo nombre. Para añadir un nuevo flash data a el/los existes con el mismo nombre, puedes llamar a `yii\web\Session::addFlash()`. Por ejemplo:

```php
$session = Yii::$app->session;

// Petición #1
// añade un pequeño mensaje flash bajo el nombre de "alerts"
$session->addFlash('alerts', 'You have successfully deleted your post.');
$session->addFlash('alerts', 'You have successfully added a new friend.');
$session->addFlash('alerts', 'You are promoted.');

// Petición #2
// $alerts es un array de mensajes flash bajo el nombre de "alerts"
$alerts = $session->getFlash('alerts');
```

> **Nota:** Intenta no usar a la vez `yii\web\Session::setFlash()` con `yii\web\Session::addFlash()` para flash data del mismo nombre. Esto ocurre porque el último método elimina el flash data dentro del array así que puedes añadir un nuevo flash data con el mismo nombre. Como resultado, cuando llamas a `yii\web\Session::getFlash()`, puedes encontrarte algunas veces que te está devolviendo un array mientras que otras veces te está devolviendo un string, esto depende del orden que invoques a estos dos métodos.

4.6.2. Cookies

Yii representa cada cookie como un objeto de `yii\web\Cookie`. Tanto `yii\web\Request` como `yii\web\Response` mantienen una colección de cookies vía la propiedad de llamada `cookies`. La colección de cookie en la antigua representación son enviadas en una petición, mientras la colección de cookie en esta última representa las cookies que van a ser enviadas al usuario.

Leyendo Cookies

Puedes recuperar las cookies en la petición actual usando el siguiente código:

```php
// devuelve la colección de cookie (yii\web\CookieCollection) del componente
"request"
$cookies = Yii::$app->request->cookies;

// devuelve el valor "language" de la cookie. Si la cookie no existe,
retorna "en" como valor por defecto.
$language = $cookies->getValue('language', 'en');

// una manera alternativa de devolver el valor "language" de la cookie
if (($cookie = $cookies->get('language')) !== null) {
    $language = $cookie->value;
}

// puedes también usar $cookies como un array
```

```php
if (isset($cookies['language'])) {
    $language = $cookies['language']->value;
}

// comprueba si hay una cookie con el valor "language"
if ($cookies->has('language')) ...
if (isset($cookies['language'])) ...
```

Enviando Cookies

Puedes enviar cookies a usuarios finales usando el siguiente código:

```php
// devuelve la colección de cookie (yii\web\CookieCollection) del componente
"response"
$cookies = Yii::$app->response->cookies;

// añade una nueva cookie a la respuesta que se enviará
$cookies->add(new \yii\web\Cookie([
    'name' => 'language',
    'value' => 'zh-CN',
]));

// remueve una cookie
$cookies->remove('language');
// equivalente a lo siguiente
unset($cookies['language']);
```

Además de `name`, `value` las propiedades que se muestran en los anteriores ejemplos, la clase `yii\web\Cookie` también define otras propiedades para representar toda la información posible de las cookies, tal como `domain`, `expire`. Puedes configurar estas propiedades según sea necesario para preparar una cookie y luego añadirlo a la colección de cookies de la respuesta.

> **Nota:** Para mayor seguridad, el valor por defecto de `yii\web\Cookie::$httpOnly` es `true`. Esto ayuda a mitigar el riesgo del acceso a la cookie protegida por script desde el lado del cliente (si el navegador lo soporta). Puedes leer el httpOnly wiki article[10] para más detalles.

Validación de la Cookie

Cuando estás leyendo y enviando cookies a través de los componentes `request` y `response` como mostramos en las dos últimas subsecciones, cuentas con el añadido de seguridad de la validación de cookies el cual protege las cookies de ser modificadas en el lado del cliente. Esto se consigue con la firma de cada cookie con una cadena hash, el cual permite a la aplicación saber si una cookie ha sido modificada en el lado del cliente o no. Si es así, la cookie no será accesible a través de `cookie collection` del componente `request`.

[10] https://owasp.org/www-community/HttpOnly

> **Información:** Si falla la validación de una cookie, aún puedes acceder a la misma a través de `$_COOKIE`. Esto sucede porque librerías de terceros pueden manipular de forma propia las cookies, lo cual no implica la validación de las mismas.

La validación de cookies es habilitada por defecto. Puedes desactivar lo ajustando la propiedad `yii\web\Request::$enableCookieValidation` a `false`, aunque se recomienda encarecidamente que no lo haga.

> **Nota:** Las cookies que son directamente leídas/enviadas vía `$_COOKIE` y `setcookie()` no serán validadas.

Cuando estás usando la validación de cookie, puedes especificar una `yii\web\Request::$cookieValidationKey` el cual se usará para generar los strings hash mencionados anteriormente. Puedes hacerlo mediante la configuración del componente `request` en la configuración de la aplicación:

```
return [
    'components' => [
        'request' => [
            'cookieValidationKey' => 'fill in a secret key here',
        ],
    ],
];
```

> **Información:** `cookieValidationKey` es crítico para la seguridad de tu aplicación. Sólo debería ser conocido por personas de confianza. No lo guardes en sistemas de control de versiones.

4.7. Gestión de Errores

Yii incluye un `error handler` que permite una gestión de errores mucho más práctica que anteriormente. En particular, el gestor de errores de Yii hace lo siguiente para mejorar la gestión de errores:

- Todos los errores no fatales (ej. advertencias (warning), avisos (notices)) se convierten en excepciones capturables.
- Las excepciones y los errores fatales de PHP se muestran con una pila de llamadas (call stack) de información detallada y lineas de código fuente.
- Soporta el uso de acciones de controlador dedicadas para mostrar errores.
- Soporta diferentes formatos de respuesta (response) de errores.

El `error handler` esta habilitado de forma predeterminada. Se puede deshabilitar definiendo la constante `YII_ENABLE_ERROR_HANDLER` con valor `false` en el script de entrada (entry script) de la aplicación.

4.7.1. Uso del Gestor de Errores

El `error handler` se registra como un componente de aplicación llamado `errorHandler`. Se puede configurar en la configuración de la aplicación como en el siguiente ejemplo:

```
return [
    'components' => [
        'errorHandler' => [
            'maxSourceLines' => 20,
        ],
    ],
];
```

Con la anterior configuración, el numero del lineas de código fuente que se mostrará en las páginas de excepciones será como máximo de 20.

Como se ha mencionado, el gestor de errores convierte todos los errores de PHP no fatales en excepciones capturables. Esto significa que se puede usar el siguiente código para tratar los errores PHP:

```
use Yii;
use yii\base\ErrorException;

try {
    10/0;
} catch (ErrorException $e) {
    Yii::warning("Division by zero.");
}

// la ejecución continua ...
```

Si se quiere mostrar una página de error que muestra al usuario que su petición no es válida o no es la esperada, se puede simplemente lanzar una excepción de tipo `HTTP exception`, como podría ser `yii\web\NotFoundHttpException`. El gestor de errores establecerá correctamente el código de estado HTTP de la respuesta y usará la vista de error apropiada para mostrar el mensaje.

```
use yii\web\NotFoundHttpException;

throw new NotFoundHttpException();
```

4.7.2. Personalizar la Visualización de Errores

El `error handler` ajusta la visualización del error conforme al valor de la constante `YII_DEBUG`. Cuando `YII_DEBUG` es `true` (es decir, en modo depuración (debug)), el gestor de errores mostrara las excepciones con una pila detallada de información y con lineas de código fuente para ayudar a depurar. Y cuando la variable `YII_DEBUG` es `false`, solo se mostrará el mensaje de error para prevenir la revelación de información sensible de la aplicación.

> **Información:** Si una excepción es descendiente de `yii\base\UserException`, no se mostrará la pila de llamadas independientemente del valor de `YII_DEBUG`. Esto es debido a que se considera que estas excepciones se deben a errores cometidos por los usuarios y los desarrolladores no necesitan corregirlas.

De forma predeterminada, el `error handler` muestra los errores usando dos vistas:

- `@yii/views/errorHandler/error.php`: se usa cuando deben mostrarse los errores SIN la información de la pila de llamadas. Cuando `YII_DEBUG` es falos, este es el único error que se mostrara.
- `@yii/views/errorHandler/exception.php`: se usa cuando los errores deben mostrarse CON la información de la pila de llamadas.

Se pueden configurar las propiedades `errorView` y `exceptionView` el gestor de errores para usar nuestros propias vistas para personalizar la visualización de los errores.

Uso de Acciones de Error

Una mejor manera de personalizar la visualización de errores es usar un acción de error dedicada. Para hacerlo, primero se debe configurar la propiedad `errorAction` del componente `errorHandler` como en el siguiente ejemplo:

```
return [
    'components' => [
        'errorHandler' => [
            'errorAction' => 'site/error',
        ],
    ]
];
```

La propiedad `errorAction` vincula una ruta a una acción. La configuración anterior declara que cuando un error tiene que mostrarse sin la pila de información de llamadas, se debe ejecutar la acción `site/error`.

Se puede crear una acción `site/error` como se hace a continuación,

```
namespace app\controllers;

use Yii;
use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'error' => [
```

```
                'class' => 'yii\web\ErrorAction',
            ],
        ];
    }
}
```

El código anterior define la acción `error` usando la clase `yii\web\ErrorAction` que renderiza un error usando la vista llamada `error`.

Además, usando `yii\web\ErrorAction`, también se puede definir la acción `error` usando un método de acción como en el siguiente ejemplo,

```
public function actionError()
{
    $exception = Yii::$app->errorHandler->exception;
    if ($exception !== null) {
        return $this->render('error', ['exception' => $exception]);
    }
}
```

Ahora se debe crear un archivo de vista ubicado en `views/sites/error.php`. En este archivo de vista, se puede acceder a las siguientes variables si se define el error como un `yii\web\ErrorAction`:

- `name`: el nombre del error;
- `message`: el mensaje del error;
- `exception`: el objeto de excepción a través del cual se puede obtener más información útil, tal como el código de estado HTTP, el código de error, la pila de llamadas del error, etc.

  Información: Tanto la plantilla de aplicación básica como la plantilla de aplicación avanzada, ya incorporan la acción de error y la vista de error.

  Nota: Si necesitas redireccionar en un gestor de error, hazlo de la siguiente manera: `php Yii::$app->getResponse()->redirect($url)->send(); return; `

Personalizar el Formato de Respuesta de Error

El gestor de errores muestra los errores de siguiente la configuración del formato de las respuestas. Si el [[yii\web\Response::format response format]] es `html`, se usará la vista de error o excepción para mostrar los errores tal y como se ha descrito en la anterior subsección. Para otros tipos de formatos de respuesta, el gestor de errores asignara la representación del array de la excepción a la propiedad `yii\web\Response::$data` que posteriormente podrá convertirse al formato deseado. Por ejemplo, si el formato de respuesta es `json`, obtendremos la siguiente respuesta:

```
HTTP/1.1 404 Not Found
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
    "code": 0,
    "status": 404
}
```

Se puede personalizar el formato de respuestas de error respondiendo al evento `beforeSend` del componente `response` en la configuración de la aplicación:

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
    ],
];
```

El código anterior reformateará la respuesta de error como en el siguiente ejemplo:

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "success": false,
    "data": {
        "name": "Not Found Exception",
        "message": "The requested resource was not found.",
        "code": 0,
        "status": 404
    }
}
```

4.8. Registro de anotaciones

Yii proporciona un poderoso framework dedicado al registro de anotaciones (logging) que es altamente personalizable y extensible. Usando este framework se pueden guardar fácilmente anotaciones (logs) de varios tipos de mensajes, filtrarlos, y unificarlos en diferentes destinos que pueden ser archivos, bases de datos o emails.

Usar el framework de registro de anotaciones de Yii involucra los siguientes pasos:

- Registrar mensajes de las anotaciones en distintos lugares del código;
- Configurar los destinos de las anotaciones en la configuración de la aplicación para filtrar y exportar los mensajes de las anotaciones;
- Examinar los mensajes filtrados de los las anotaciones exportadas para diferentes destinos (ej. Yii debugger).

En esta sección, se describirán principalmente los dos primeros pasos.

4.8.1. Anotación de Messages

Registrar mensajes de anotación es tan simple como llamar a uno de los siguientes métodos de registro de anotaciones.

- `Yii::debug()`: registra un mensaje para trazar el funcionamiento de una sección de código. Se usa principalmente para tareas de desarrollo.
- `Yii::info()`: registra un mensaje que transmite información útil.
- `Yii::warning()`: registra un mensaje de advertencia que indica que ha sucedido algo inesperado.
- `Yii::error()`: registra un error fatal que debe ser investigado tan pronto como sea posible.

Estos métodos registran mensajes de varios *niveles de severidad* y *categorías*. Comparten el mismo registro de función `function ($message, $category = 'application')`, donde `$message` representa el mensaje del registro que tiene que ser registrado, mientras que `$category` es la categoría del registro de mensaje. El código del siguiente ejemplo registra la huella del mensaje para la categoría `application`:

```
Yii::debug('start calculating average revenue');
```

> **Información:** Los mensajes de registro pueden ser tanto cadenas de texto como datos complejos, como arrays u objetos. Es responsabilidad de los destinos de registros tratar los mensajes de registro de manera apropiada. De forma predeterminada, si un mensaje de registro no es una cadena de texto, se exporta como si fuera un string llamando a `yii\helpers\VarDumper::export()`.

Para organizar mejor y filtrar los mensajes de registro, se recomienda especificar una categoría apropiada para cada mensaje de registro. Se puede elegir

un sistema de nombres jerárquicos por categorías que facilite a los destino de registros el filtrado de mensajes basándose en categorías. Una manera simple pero efectiva de organizarlos es usar la constante predefinida (magic constant) de PHP `__METHOD__` como nombre de categoría. Además este es el enfoque que se usa en el código del núcleo (core) del framework Yii. Por ejemplo,

```
Yii::debug('start calculating average revenue', __METHOD__);
```

La constante `__METHOD__` equivale al nombre del método (con el prefijo del nombre completo del nombre de clase) donde se encuentra la constante. Por ejemplo, es igual a la cadena `'app\controllers\RevenueController::calculate'` si la linea anterior de código se llamara dentro de este método.

> **Información:** Los métodos de registro de anotaciones descritos anteriormente en realidad son accesos directos al método `log()` del `logger object` que es un singleton accesible a través de la expresión `Yii::getLogger()`. Cuando se hayan registrado suficientes mensajes o cuando la aplicación haya finalizado, el objeto de registro llamará `message dispatcher` para enviar los mensajes de registro registrados a los destiinos de registros.

4.8.2. Destino de Registros

Un destino de registro es una instancia de la clase `yii\log\Target` o de una clase hija. Este filtra los mensajes de registro por sus niveles de severidad y sus categorías y después los exporta a algún medio. Por ejemplo, un `database target` exporta los mensajes de registro filtrados a una tabla de base de datos, mientras que un `email target` exporta los mensajes de registro a una dirección de correo electrónico específica.

Se pueden registrar múltiples destinos de registros en una aplicación configurándolos en la aplicación de componente `log` dentro de la configuración de aplicación, como en el siguiente ejemplo:

```
return [
    // el componente log tiene que cargarse durante el proceso de
    bootstrapping
    'bootstrap' => ['log'],

    'components' => [
        'log' => [
            'targets' => [
                [
                    'class' => 'yii\log\DbTarget',
                    'levels' => ['error', 'warning'],
                ],
                [
                    'class' => 'yii\log\EmailTarget',
```

```
                    'levels' => ['error'],
                    'categories' => ['yii\db\*'],
                    'message' => [
                        'from' => ['log@example.com'],
                        'to' => ['admin@example.com',
                        'developer@example.com'],
                        'subject' => 'Database errors at example.com',
                    ],
                ],
            ],
        ],
    ],
];
```

> **Nota:** El componente `log` debe cargarse durante el proceso de bootstrapping para que pueda enviar los mensajes de registro a los destinos inmediatamente. Este es el motivo por el que se lista en el array `bootstrap` como se muestra más arriba.

En el anterior código, se registran dos destinos de registros en la propiedad `yii\log\Dispatcher::$targets`

- el primer destino gestiona los errores y las advertencias y las guarda en una tabla de la base de datos;
- el segundo destino gestiona mensajes los mensajes de error de las categorías cuyos nombres empiecen por `yii\db\` y los envía por email a las direcciones `admin@example.com` y `developer@example.com`.

Yii incluye los siguientes destinos. En la API de documentación se pueden referencias a estas clases e información de configuración y uso.

- `yii\log\DbTarget`: almacena los mensajes de registro en una tabla de la base de datos.
- `yii\log\EmailTarget`: envía los mensajes de registro a direcciones de correo preestablecidas.
- `yii\log\FileTarget`: guarda los menajes de registro en archivos.
- `yii\log\SyslogTarget`: guarda los mensajes de registro en el syslog llamando a la función PHP `syslog()`.

A continuación, se describirá las características más comunes de todos los destinos de registros.

Filtrado de Mensajes

Se pueden configurar las propiedades `levels` y `categories` para cada destino de registros, con estas se especifican los niveles de severidad y las categorías de mensajes que deberán procesar sus destinos.

La propiedad `levels` es un array que consta de uno o varios de los siguientes valores:

- `error`: correspondiente a los mensajes registrados por `Yii::error()`.
- `warning`: correspondiente a los mensajes registrados por `Yii::warning()`.

- `info`: correspondiente a los mensajes registrados por `Yii::info()`.
- `trace`: correspondiente a los mensajes registrados por `Yii::debug()`.
- `profile`: correspondiente a los mensajes registrados por `Yii::beginProfile()` y `Yii::endProfile()`, que se explicará más detalladamente en la subsección Perfiles.

Si no se especifica la propiedad `levels`, significa que el destino procesará los mensajes de *cualquier* nivel de severidad.

La propiedad `categories` es un array que consta de categorías de mensaje o patrones. El destino sólo procesará mensajes de las categorías que se puedan encontrar o si coinciden con algún patrón listado en el array. Un patrón de categoría es un nombre de categoría al que se le añade un asterisco `*` al final. Un nombre de categoría coincide con un patrón si empieza por el mismo prefijo que el patrón. Por ejemplo, `yii\db\Command::execute` y `yii\db\Command::query` que se usan como nombres de categoría para los mensajes registrados en la clase `yii\db\Command`, coinciden con el patrón `yii\db\*`.

Si no se especifica la propiedad `categories`, significa que el destino procesará los mensajes de *todas* las categorías.

Además añadiendo las categorías en listas blancas (whitelisting) mediante la propiedad `categories`, también se pueden añadir ciertas categorías en listas negras (blacklist) configurando la propiedad `except`. Si se encuentra la categoría de un mensaje o coincide algún patrón con esta propiedad, NO será procesada por el destino.

La siguiente configuración de destinos especifica que el destino solo debe procesar los mensajes de error y de advertencia de las categorías que coincidan con alguno de los siguientes patrones `yii\db\*` o `yii\web\HttpException:*`, pero no con `yii\web\HttpException:404`.

```
[
    'class' => 'yii\log\FileTarget',
    'levels' => ['error', 'warning'],
    'categories' => [
        'yii\db\*',
        'yii\web\HttpException:*',
    ],
    'except' => [
        'yii\web\HttpException:404',
    ],
]
```

> **Información:** Cuando se captura una excepción de tipo HTTP por el gestor de errores, se registrará un mensaje de error con el nombre de categoría con formato `yii\web\HttpException:ErrorCode`. Por ejemplo, la excepción `yii\web\NotFoundHttpException` causará un mensaje de error del tipo `yii\web\HttpException:404`.

Formato de los Mensajes

Los destinos exportan los mensajes de registro filtrados en cierto formato. Por ejemplo, is se instala un destino de registros de la calse `yii\log\FileTarget`, encontraremos un registro similar en el archivo de registro `runtime/log/app.log`:

```
2014-10-04 18:10:15 [::1][][-][trace][yii\base\Module::getModule] Loading
module: debug
```

De forma predeterminada los mensajes de registro se formatearan por `yii\log\Target::formatMessage()` como en el siguiente ejemplo:

```
Timestamp [IP address][User ID][Session ID][Severity Level][Category]
Message Text
```

Se puede personalizar el formato configurando la propiedad `yii\log\Target::$prefix` que es un PHP ejecutable y devuelve un prefijo de mensaje personalizado. Por ejemplo, el siguiente código configura un destino de registro anteponiendo a cada mensaje de registro el ID de usuario (se eliminan la dirección IP y el ID por razones de privacidad).

```
[
    'class' => 'yii\log\FileTarget',
    'prefix' => function ($message) {
        $user = Yii::$app->has('user', true) ? Yii::$app->get('user') :
        null;
        $userID = $user ? $user->getId(false) : '-';
        return "[$userID]";
    }
]
```

Además de prefijos de mensaje, destinos de registros también añaden alguna información de contexto en cada lote de mensajes de registro. De forma predeterminada, se incluyen los valores de las siguientes variables globales de PHP: `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION` y `$_SERVER`. Se puede ajustar el comportamiento configurando la propiedad `yii\log\Target::$logVars` con los nombres de las variables globales que se quieran incluir con el destino del registro. Por ejemplo, la siguiente configuración de destino de registros especifica que sólo se añadirá al mensaje de registro el valor de la variable `$_SERVER`.

```
[
    'class' => 'yii\log\FileTarget',
    'logVars' => ['_SERVER'],
]
```

Se puede configurar `logVars` para que sea un array vacío para deshabilitar totalmente la inclusión de información de contexto. O si se desea implementar un método propio de proporcionar información de contexto se puede sobrescribir el método `yii\log\Target::getContextMessage()`.

Nivel de Seguimiento de Mensajes

Durante el desarrollo, a veces se quiere visualizar de donde proviene cada mensaje de registro. Se puede lograr configurando la propiedad `traceLevel` del componente `log` como en el siguiente ejemplo:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [...],
        ],
    ],
];
```

La configuración de aplicación anterior establece el `traceLevel` para que sea 3 si `YII_DEBUG` esta habilitado y 0 si esta deshabilitado. Esto significa que si `YII_DEBUG` esta habilitado, a cada mensaje de registro se le añadirán como mucho 3 niveles de la pila de llamadas del mensaje que se este registrando; y si `YII_DEBUG` está deshabilitado, no se incluirá información de la pila de llamadas.

> **Información:** Obtener información de la pila de llamadas no es trivial. Por lo tanto, sólo se debe usar esta característica durante el desarrollo o cuando se depura la aplicación.

Liberación (Flushing) y Exportación de Mensajes

Como se ha comentado anteriormente, los mensajes de registro se mantienen en un array por el `logger object`. Para limitar el consumo de memoria de este array, el componente encargado del registro de mensajes enviará los mensajes registrados a los destinos de registros cada vez que el array acumule un cierto número de mensajes de registro. Se puede personalizar el número configurando la propiedad `flushInterval` del componente `log`:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 100,   // el valor predeterminado es 1000
            'targets' => [...],
        ],
    ],
];
```

> **Información:** También se produce la liberación de mensajes cuando la aplicación finaliza, esto asegura que los destinos de los registros reciban los mensajes de registro.

Cuando el `logger object` libera los mensajes de registro enviándolos a los destinos de registros, estos no se exportan inmediatamente. La exportación de mensajes solo se produce cuando un destino de registros acumula un cierto número de mensajes filtrados. Se puede personalizar este número configurando la propiedad `exportInterval` de un destinos de registros individual, como se muestra a continuación,

```
[
    'class' => 'yii\log\FileTarget',
    'exportInterval' => 100,  // el valor predeterminado es 1000
]
```

Debido al nivel de configuración de la liberación y exportación de mensajes, de forma predeterminada cuando se llama a `Yii::debug()` o cualquier otro método de registro de mensajes, NO veremos el registro de mensaje inmediatamente en los destinos de registros. Esto podría ser un problema para algunas aplicaciones de consola de ejecución prolongada (long-running). Para hacer que los mensajes de registro aparezcan inmediatamente en los destinos de registro se deben establecer `flushInterval` y `exportInterval` para que tengan valor 1 como se muestra a continuación:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 1,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'exportInterval' => 1,
                ],
            ],
        ],
    ],
];
```

> **Nota:** El uso frecuente de liberación y exportación puede degradar el rendimiento de la aplicación.

Conmutación de Destinos de Registros

Se puede habilitar o deshabilitar un destino de registro configuración su propiedad `enabled`. Esto se puede llevar a cabo a mediante la configuración del destino de registros o con la siguiente declaración PHP de código:

```
Yii::$app->log->targets['file']->enabled = false;
```

El código anterior requiere que se asocie un destino como `file`, como se muestra a continuación usando las claves de texto en el array `targets`:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'targets' => [
                'file' => [
                    'class' => 'yii\log\FileTarget',
                ],
                'db' => [
                    'class' => 'yii\log\DbTarget',
                ],
            ],
        ],
    ],
];
```

Creación de Nuevos Destinos

La creación de nuevas clases de destinos de registro es muy simple. Se necesita implementar el método `yii\log\Target::export()` enviando el contenido del array `yii\log\Target::$messages` al medio designado. Se puede llamar al método `yii\log\Target::formatMessage()` para formatear los mensajes. Se pueden encontrar más detalles de destinos de registros en las clases incluidas en la distribución de Yii.

4.8.3. Perfilado de Rendimiento

El Perfilado de rendimiento es un tipo especial de registro de mensajes que se usa para medir el tiempo que tardan en ejecutarse ciertos bloques de código y encontrar donde están los cuellos de botella de rendimiento. Por ejemplo, la clase `yii\db\Command` utiliza el perfilado de rendimiento para encontrar conocer el tiempo que tarda cada consulta a la base de datos.

Para usar el perfilado de rendimiento, primero debemos identificar los bloques de código que tienen que ser perfilados, para poder enmarcar su contenido como en el siguiente ejemplo:

```
\Yii::beginProfile('myBenchmark');

... Empieza el perfilado del bloque de código ...

\Yii::endProfile('myBenchmark');
```

Donde `myBenchmark` representa un token único para identificar el bloque de código. Después cuando se examine el resulte del perfilado, se podrá usar este token para encontrar el tiempo que ha necesitado el correspondiente bloque de código.

Es importante asegurarse de que los pares de `beginProfile` y `endProfile` estén bien anidados. Por ejemplo,

```
\Yii::beginProfile('block1');

    // código que será perfilado

    \Yii::beginProfile('block2');
        // más código para perfilar
    \Yii::endProfile('block2');

\Yii::endProfile('block1');
```

Si nos dejamos el `\Yii::endProfile('block1')` o lo intercambiamos `\Yii::endProfile('block1')` con `\Yii::endProfile('block2')`, el perfilado de rendimiento no funcionará.

Se registra un mensaje de registro con el nivel de severidad `profile` para cada bloque de código que se haya perfilado. Se puede configurar el destino del registro para reunir todos los mensajes y exportarlos. El depurador de Yii incluye un panel de perfilado de rendimiento que muestra los resultados de perfilado.

Capítulo 5

Conceptos clave

5.1. Componentes

Los componentes son los principales bloques de construcción de las aplicaciones Yii. Los componentes son instancias de `yii\base\Component` o de una clase extendida. Las tres características principales que los componentes proporcionan a las otras clases son:

- Propiedades
- Eventos
- Comportamientos

Por separado y combinadas, estas características hacen que las clases Yii sean mucho mas personalizables y sean mucho más fáciles de usar. Por ejemplo, el incluido `yii\jui\DatePicker`, un componente de la interfaz de usuario, puede ser utilizado en una vista para generar un DatePicker interactivo:

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'ru',
    'name'  => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

Las propiedades del widget son fácilmente modificables porque la clase se extiende de `yii\base\Component`.

Mientras que los componentes son muy potentes, son un poco más pesados que los objetos normales, debido al hecho de que necesitan más memoria y tiempo de CPU para poder soportar eventos y comportamientos en particular. Si tus componentes no necesitan estas dos características, deberías considerar extender tu componente directamente de `yii\base\BaseObject` en vez de `yii\base\Component`. De esta manera harás que tus componentes

sean mucho más eficientes que objetos PHP normales, pero con el añadido soporte para propiedades.

Cuando extiendes tu clase de `yii\base\Component` o `yii\base\BaseObject`, se recomienda que sigas las siguientes convenciones:

- Si sobrescribes el constructor, especifica un parámetro `$config` como el *último* parámetro del constructor, y después pasa este parámetro al constructor padre.
- Siempre llama al constructor padre al *final* de su propio constructor.
- Si sobrescribes el método `yii\base\BaseObject::init()`, asegúrese de llamar la implementación padre de `init` *al principio* de su método `init`.

Por ejemplo:

```
namespace yii\components\MyClass;

use yii\base\BaseObject;

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

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

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();

        // ... inicialización después de la configuración esta siendo
        aplicada
    }
}
```

Siguiendo esas directrices hará que tus componentes sean configurables cuando son creados. Por ejemplo:

```
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
// alternativamente
$component = \Yii::createObject([
    'class' => MyClass::class,
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

> **Información:** Mientras que el enfoque de llamar `Yii::createObject()` parece mucho más complicado, es mucho más potente debido al

hecho de que se implementa en la parte superior de un contenedor de inyección de dependencia.

La clase `yii\base\BaseObject` hace cumplir el siguiente ciclo de vida del objeto:

1. Pre-inicialización en el constructor. Puedes establecer los valores predeterminados de propiedades aquí.
2. Configuración del objeto a través de `$config`. La configuración puede sobrescribir los valores prdeterminados dentro del constructor.
3. Post-inicialización dentro de `init()`. Puedes sobrescribir este método para realizar comprobaciones de validez y normalización de las propiedades.
4. Llamadas a métodos del objeto.

Los tres primeros pasos ocurren dentro del constructor del objeto. Esto significa que una vez obtengas la instancia de un objeto, ésta ha sido inicializada para que puedas utilizarla adecuadamente.

5.2. Propiedades

En PHP, las variables miembro de clases también llamadas *propiedades*, son parte de la definición de la clase, y se usan para representar el estado de una instancia de la clase (ej. para diferenciar una instancia de clase de otra). A la práctica, a menudo, se puede querer gestionar la lectura o escritura de las propiedades de algunos momentos. Por ejemplo, se puede querer eliminar los espacios en blanco (trim) de una cadena de texto cada vez que esta se asigne a una propiedad de tipo `label`. Se *podría* usar el siguiente código para realizar esta tarea:

```
$object->label = trim($label);
```

La desventaja del código anterior es que se tendría que ejecutar `trim()` en todas las partes del código que pudieran establecer la propiedad `label`. Si en el futuro, la propiedad `label` tiene que seguir otro funcionamiento, como por ejemplo que la primera letra tiene que estar en mayúsculas, se tendrán que modificar todas las secciones de código que asignen el valor a la propiedad `label`. La repetición de código conlleva a bugs, y es una practica que se tiene que evitar en la medida de lo posible.

Para solventar este problema, Yii introduce la clase base llamada `yii\base\BaseObject` que da soporte a la definición de propiedades basada en los métodos de clase *getter* y *setter*. Si una clase necesita más funcionalidad, debe extender a la clase `yii\base\BaseObject` o a alguna de sus hijas.

> **Información:** Casi todas las clases del núcleo (core) en el framework Yii extienden a `yii\base\BaseObject` o a una de sus clases hijas. Esto significa que siempre que se encuentre un getter o un setter en una clase del núcleo, se puede utilizar como una propiedad.

Un método getter es un método cuyo nombre empieza por la palabra `get`: un metodo setter empieza por `set`. El nombre añadido detrás del prefijo `get` o `set` define el nombre de la propiedad. Por ejemplo, un getter `getLabel()` y/o un setter `setLabel()` definen la propiedad `label`, como se muestra a continuación:

```
namespace app\components;

use yii\base\BaseObject;

class Foo extends BaseObject
{
    private $_label;

    public function getLabel()
    {
        return $this->_label;
    }

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

(Para ser claros, los métodos getter y setter crean la propiedad `label`, que en este caso hace una referencia interna al nombre de atributo privado `_label`.)

Las propiedades definidas por los getter y los setters se pueden usar como variables de clase miembro. La principal diferencia radica en que cuando esta propiedad se lea, se ejecutará su correspondiente método getter; cuando se asigne un valor a la propiedad, se ejecutará el correspondiente método setter. Por ejemplo:

```
// equivalente a $label = $object->getLabel();
$label = $object->label;

// equivalente a $object->setLabel('abc');
$object->label = 'abc';
```

Una propiedad definida por un getter sin un setter es de tipo *sólo lectura*. Si se intenta asignar un valor a esta propiedad se producirá una excepción de tipo `InvalidCallException`. Del mismo modo que una propiedad definida con un setter pero sin getter será de tipo *sólo escritura*, cualquier intento de lectura de esta propiedad producirá una excepción. No es común tener variables de tipo sólo escritura.

Hay varias reglas especiales y limitaciones en las propiedades definidas mediante getters y setters:

- Los nombres de estas propiedades son *case-insensitive*. Por ejemplo, `$object->label` y `$object->Label` son la misma. Esto se debe a que los nombres de los métodos en PHP son case-insensitive.
- Si el nombre de una propiedad de este tipo es igual al de una variable miembro de la clase, la segunda tendrá prioridad. Por ejemplo, si la anterior clase `Foo` tiene la variable miembro `label`, entonces la asignación `$object->label = 'abc'` afectará a la *variable miembro* 'label'; no se ejecutará el método setter `setLabel()`.
- Estas variables no soportan la visibilidad. No hay diferencia en definir los métodos getter o setter en una propiedad public, protected, o private.
- Las propiedades sólo se pueden definir por getters y setters *no estáticos*. Los métodos estáticos no se tratarán de la misma manera.

Volviendo de nuevo al problema descrito al principio de la guía, en lugar de ejecutar `trim()` cada vez que se asigne un valor a `label`, ahora `trim()` sólo necesita ser invocado dentro del setter `setLabel()`. I si se tiene que añadir un nuevo requerimiento, para que `label` empiece con una letra mayúscula, se puede modificar rápidamente el método `setLabel()` sin tener que modificar más secciones de código. El cambio afectará a cada asignación de `label`.

5.3. Eventos

Los eventos permiten inyectar código dentro de otro código existente en ciertos puntos de ejecución. Se pueden adjuntar código personalizado a un evento, cuando se lance (triggered), el código se ejecutará automáticamente. Por ejemplo, un objeto mailer puede lanzar el evento `messageSent` cuando se envía un mensaje correctamente. Si se quiere rastrear el correcto envío del mensaje, se puede, simplemente, añadir un código de seguimiento al evento `messageSent`.

Yii introduce una clase base `yii\base\Component` para soportar eventos. Si una clase necesita lanzar un evento, este debe extender a `yii\base\Component` o a una clase hija.

5.3.1. Gestor de Eventos

Un gestor de eventos es una llamada de retorno PHP (PHP callback)[1] que se ejecuta cuando se lanza el evento al que corresponde. Se puede usar cualquier llamada de retorno de las enumeradas a continuación:

- una función de PHP global especificada como una cadena de texto (sin paréntesis), ej. `'trim'`;

[1] https://www.php.net/manual/es/language.types.callable.php

- un método de objeto especificado como un array de un objeto y un nombre de método como una cadena de texto (sin paréntesis), ej. `[$object, 'methodNAme']`;
- un método de clase estático especificado como un array de un nombre de clase y un método como una cadena de texto (sin paréntesis), ej. `[$class, 'methodName']`;
- una función anónima, ej. `function ($event) { ... }`.

La firma de un gestor de eventos es:

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

Un gestor de eventos puede obtener la siguiente información acerca de un evento ya sucedido mediante el parámetro `$event`:

- `nombre del evento`
- `evento enviando`: el objeto desde el que se ha ejecutado `trigger()`
- `custom data`: los datos que se proporcionan al adjuntar el gestor de eventos (se explicará más adelante)

5.3.2. Añadir Gestores de Eventos

Se puede añadir un gestor a un evento llamando al método `yii\base\Component::on()`. Por ejemplo:

```
$foo = new Foo;

// este gestor es una función global
$foo->on(Foo::EVENT_HELLO, 'function_name');

// este gestor es un método de objeto
$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);

// este gestor es un método de clase estática
$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// este gestor es una función anónima
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // event handling logic
});
```

También se pueden adjuntar gestores de eventos mediante configuraciones. Se pueden encontrar más de talles en la sección Configuraciones.

Cuando se adjunta un gestor de eventos, se pueden proporcionar datos adicionales como tercer parámetro de `yii\base\Component::on()`. El gestor podrá acceder a los datos cuando se lance el evento y se ejecute el gestor. Por ejemplo:

```php
// El siguiente código muestra "abc" cuando se lanza el evento
// ya que $event->data contiene los datos enviados en el tercer parámetro de
"on"
$foo->on(Foo::EVENT_HELLO, 'function_name', 'abc');

function function_name($event) {
    echo $event->data;
}
```

5.3.3. Ordenación de Gestores de Eventos

Se puede adjuntar uno o más gestores a un único evento. Cuando se lanza un evento, se ejecutarán los gestores adjuntos en el orden que se hayan añadido al evento. Si un gestor necesita parar la invocación de los gestores que le siguen, se puede establecer la propiedad `yii\base\Event::$handled` del parámetro `$event` para que sea `true`:

```php
$foo->on(Foo::EVENT_HELLO, function ($event) {
    $event->handled = true;
});
```

De forma predeterminada, cada nuevo gestor añadido se pone a la cola de la lista de gestores del evento. Por lo tanto, el gestor se ejecutará en el último lugar cuando se lance el evento. Para insertar un nuevo gestor al principio de la cola de gestores para que sea ejecutado primero, se debe llamar a `yii\base\Component::on()`, pasando al cuarto parámetro `$append` el valor `false`:

```php
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // ...
}, $data, false);
```

5.3.4. Lanzamiento de Eventos

Los eventos se lanzan llamando al método `yii\base\Component::trigger()`. El método requiere un *nombre de evento*, y de forma opcional un objeto de evento que describa los parámetros que se enviarán a los gestores de eventos. Por ejemplo:

```php
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class Foo extends Component
{
    const EVENT_HELLO = 'hello';

    public function bar()
    {
        $this->trigger(self::EVENT_HELLO);
```

```
    }
}
```

Con el código anterior, cada llamada a `bar()` lanzará un evento llamado `hello`

> **Consejo:** Se recomienda usar las constantes de clase para representar nombres de eventos. En el anterior ejemplo, la constante `EVENT_HELLO` representa el evento `hello`. Este enfoque proporciona tres beneficios. Primero, previene errores tipográficos. Segundo, puede hacer que los IDEs reconozcan los eventos en las funciones de auto-completado. Tercero, se puede ver que eventos soporta una clase simplemente revisando la declaración de constantes.

A veces cuando se lanza un evento se puede querer pasar información adicional al gestor de eventos. Por ejemplo, un mailer puede querer enviar la información del mensaje para que los gestores del evento `messageSent` para que los gestores puedan saber las particularidades del mensaje enviado. Para hacerlo, se puede proporcionar un objeto de tipo evento como segundo parámetro al método `yii\base\Component::trigger()`. El objeto de tipo evento debe ser una instancia de la clase `yii\base\Event` o de su clase hija. Por ejemplo:

```
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class MessageEvent extends Event
{
    public $message;
}

class Mailer extends Component
{
    const EVENT_MESSAGE_SENT = 'messageSent';

    public function send($message)
    {
        // ...enviando $message...

        $event = new MessageEvent;
        $event->message = $message;
        $this->trigger(self::EVENT_MESSAGE_SENT, $event);
    }
}
```

Cuando se lanza el método `yii\base\Component::trigger()`, se ejecutarán todos los gestores adjuntos al evento.

5.3.5. Desadjuntar Gestores de Evento

Para desadjuntar un gestor de un evento, se puede ejecutar el método `yii\base\Component::off()`. Por ejemplo:

```
// el gestor es una función global
$foo->off(Foo::EVENT_HELLO, 'function_name');

// el gestor es un método de objeto
$foo->off(Foo::EVENT_HELLO, [$object, 'methodName']);

// el gestor es un método estático de clase
$foo->off(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// el gestor es una función anónima
$foo->off(Foo::EVENT_HELLO, $anonymousFunction);
```

Tenga en cuenta que en general no se debe intentar desadjuntar las funciones anónimas a no ser que se almacene donde se ha adjuntado al evento. En el anterior ejemplo, se asume que la función anónima se almacena como variable `$anonymousFunction`.

Para desadjuntar TODOS los gestores de un evento, se puede llamar `yii\base\Component::off()` sin el segundo parámetro:

```
$foo->off(Foo::EVENT_HELLO);
```

5.3.6. Nivel de Clase (Class-Level) Gestores de Eventos

En las subsecciones anteriores se ha descrito como adjuntar un gestor a un evento a *nivel de instancia*. A veces, se puede querer que un gestor responda todos los eventos de *todos* las instancias de una clase en lugar de una instancia especifica. En lugar de adjuntar un gestor de eventos a una instancia, se puede adjuntar un gestor a *nivel de clase* llamando al método estático `yii\base\Event::on()`.

Por ejemplo, un objeto de tipo Active Record lanzará un evento `EVENT_AFTER_INSERT` cada vez que inserte un nuevo registro en la base de datos. Para poder registrar las inserciones efectuadas por *todos* los objetos Active Record, se puede usar el siguiente código:

```
use Yii;
use yii\base\Event;
use yii\db\ActiveRecord;

Event::on(ActiveRecord::class, ActiveRecord::EVENT_AFTER_INSERT, function
($event) {
    Yii::debug(get_class($event->sender) . ' is inserted');
});
```

Se invocará al gestor de eventos cada vez que una instancia de `ActiveRecord`, o de uno de sus clases hijas, lance un evento de tipo `EVENT_AFTER_INSERT`.

Se puede obtener el objeto que ha lanzado el evento mediante `$event->sender` en el gestor.

Cuando un objeto lanza un evento, primero llamará los gestores a nivel de instancia, y a continuación los gestores a nivel de clase.

Se puede lanzar un evento de tipo *nivel de clase* llamando al método estático `yii\base\Event::trigger()`. Un evento de nivel de clase no se asocia a un objeto en particular. Como resultado, esto provocará solamente la invocación de los gestores de eventos a nivel de clase.

```
use yii\base\Event;

Event::on(Foo::class, Foo::EVENT_HELLO, function ($event) {
    var_dump($event->sender);  // displays "null"
});

Event::trigger(Foo::class, Foo::EVENT_HELLO);
```

Tenga en cuenta que en este caso, el `$event->sender` hace referencia al nombre de la clase que lanza el evento en lugar de a la instancia del objeto.

> **Nota:** Debido a que los gestores a nivel de clase responderán a los eventos lanzados por cualquier instancia de la clase, o cualquier clase hija, se debe usar con cuidado, especialmente en las clases de bajo nivel (low-level), tales como `yii\base\BaseObject`.

Para desadjuntar un gestor de eventos a nivel de clase, se tiene que llamar a `yii\base\Event::off()`. Por ejemplo:

```
// desadjunta $handler
Event::off(Foo::class, Foo::EVENT_HELLO, $handler);

// desadjunta todos los gestores de Foo::EVENT_HELLO
Event::off(Foo::class, Foo::EVENT_HELLO);
```

5.3.7. Eventos Globales

Yii soporta los llamados *eventos globales*, que en realidad es un truco basado en el gestor de eventos descrito anteriormente. El evento global requiere un Singleton globalmente accesible, tal como la instancia de aplicación en si misma.

Para crear un evento global, un evento remitente (event sender) llama al método `trigger()` del Singleton para lanzar el evento, en lugar de llamar al propio método `trigger()` del remitente. De forma similar, los gestores de eventos se adjuntan al evento del Singleton. Por ejemplo:

```
use Yii;
use yii\base\Event;
use app\components\Foo;
```

```
Yii::$app->on('bar', function ($event) {
    echo get_class($event->sender);  // muestra "app\components\Foo"
});

Yii::$app->trigger('bar', new Event(['sender' => new Foo]));
```

Un beneficio de usar eventos globales es que no se necesita un objeto cuando se adjuntan gestores a un evento para que sean lanzados por el objeto. En su lugar, los gestores adjuntos y el lanzamiento de eventos se efectúan en el Singleton (ej. la instancia de la aplicación).

Sin embargo, debido a que los `namespaces` de los eventos globales son compartidos por todas partes, se les deben asignar nombres bien pensados, como puede ser la introducción de algún `namespace` (ej. "frontend.mail.sent", "backend.mail.sent").

5.4. Comportamientos

Comportamientos son instancias de `yii\base\Behavior` o sus clases "hija". Comportamientos, también conocido como mixins[2], te permiten mejorar la funcionalidad de un `componente` existente sin necesidad de modificar su herencia de clases. Cuando un comportamiento se une a un componente, "inyectará" sus métodos y propiedades dentro del componente, y podrás acceder a esos métodos y propiedades como si hubieran estado definidos por la clase de componente. Además, un comportamiento puede responder a eventos disparados por el componente de modo que se pueda personalizar o adaptar a la ejecución normal del código del componente.

5.4.1. Definiendo comportamientos

Para definir un comportamiento, se debe crear una clase que exiende `yii\base\Behavior`, o se extiende una clase hija. Por ejemplo:

```
namespace app\components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;

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

[2] https://es.wikipedia.org/wiki/Mixin

```
    }

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

    public function foo()
    {
        // ...
    }
}
```

El código anterior define la clase de comportamiento (behavior) app\components\MyBehavior, con dos propiedades --prop1 yprop2--y un métodofoo(). Tenga en cuenta que la propiedadprop2 se define a través de la gettergetProp2() y el settersetProp2()‘. Este caso es porque `yii\base\Behavior` extiende `yii\base\BaseObject` y por lo tanto se apoya en la definición de propiedades via getters y setters.

Debido a que esta clase es un comportamiento, cuando está unido a un componente, el componente también tienen la propiedad `prop1` y `prop2` y el método `foo()`.

> **Consejo:** Dentro de un comportamiento, puede acceder al componente que el comportamiento está unido a través de la propiedad `yii\base\Behavior::$owner`.

5.4.2. Gestión de eventos de componentes

Si un comportamiento necesita responder a los acontecimientos desencadenados por el componente al que está unido, se debe reemplazar el método `yii\base\Behavior::events()`. Por ejemplo:

```
namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
```

```
        // ...
    }
}
```

El método `events()` debe devolver una lista de eventos y sus correspondientes controladores. El ejemplo anterior declara que el evento `EVENT_BEFORE_VALIDATE` existe y esta exists y define su controlador, `beforeValidate()`. Al especificar un controlador de eventos, puede utilizar uno de los siguientes formatos:

- una cadena que se refiere al nombre de un método de la clase del comportamiento, como el ejemplo anterior
- un arreglo de objeto o nombre de clase, y un nombre de método como una cadena (sin paréntesis), ej., `[$object, 'methodName']`;
- una función anónima

La firma de un controlador de eventos debe ser la siguiente, donde `$ event` refiere al parámetro de evento. Por favor, consulte la sección Eventos para más detalles sobre los eventos.

```
function ($event) {
}
```

5.4.3. Vinculando Comportamientos

Puedes vincular un comportamiento a un `componente` ya sea estática o dinámicamente. La primera forma es la más comúnmente utilizada en la práctica.

Para unir un comportamiento estáticamente, reemplaza el método `behaviors()` dde la clase de componente a la que se une el comportamiento. El método `behaviors()` debe devolver una lista de comportamiento configuraciones. Cada configuración de comportamiento puede ser un nombre de clase de comportamiento o un arreglo de configuración:

```
namespace app\models;

use yii\db\ActiveRecord;
use app\components\MyBehavior;

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // anonymous behavior, behavior class name only
            MyBehavior::class,

            // named behavior, behavior class name only
            'myBehavior2' => MyBehavior::class,

            // anonymous behavior, configuration array
            [
```

```
                'class' => MyBehavior::class,
                'prop1' => 'value1',
                'prop2' => 'value2',
            ],

            // named behavior, configuration array
            'myBehavior4' => [
                'class' => MyBehavior::class,
                'prop1' => 'value1',
                'prop2' => 'value2',
            ]
        ];
    }
}
```

Puedes asociciar un nombre a un comportamiento especificándolo en la clave de la matriz correspondiente a la configuración del comportamiento. En este caso, el comportamiento puede ser llamado un *comportamiento nombrado* (named behavior). En el ejemplo anterior, hay dos tipos de comportamientos nombrados: `myBehavior2` y `myBehavior4`. Si un comportamiento no está asociado con un nombre, se le llama *comportamiento anónimo* (anonymous behavior).

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

```
use app\components\MyBehavior;

// vincular un objeto comportamiento "behavior"
$component->attachBehavior('myBehavior1', new MyBehavior());

// vincular una clase comportamiento
$component->attachBehavior('myBehavior2', MyBehavior::class);

// asociar una matriz de configuración
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::class,
    'prop1' => 'value1',
    'prop2' => 'value2',
]);
```

Puede vincular múltiples comportamientos a la vez mediante el uso del método `yii\base\Component::attachBehaviors()`. Por ejemplo,

```
$component->attachBehaviors([
    'myBehavior1' => new MyBehavior(), // un comportamiento nombrado
    MyBehavior::class,                 // un comportamiento anónimo
]);
```

También puedes asociar comportamientos a traves de configuraciones como el siguiente:

```
[
    'as myBehavior2' => MyBehavior::class,

    'as myBehavior3' => [
        'class' => MyBehavior::class,
        'prop1' => 'value1',
        'prop2' => 'value2',
    ],
]
```

Para más detalles, por favor visita la sección Configuraciones.

5.4.4. Usando comportamientos

Para poder utilizar un comportamiento, primero tienes que unirlo a un `componente` según las instrucciones anteriores. Una vez que un comportamiento ha sido vinculado a un componente, su uso es sencillo.

Puedes usar a una variable *pública* o a una propiedad definida por un `getter` y/o un `setter` del comportamiento a través del componente con el que se ha vinculado:

```
// "prop1" es una propiedad definida en la clase comportamiento
echo $component->prop1;
$component->prop1 = $value;
```

También puedes llamar métodos *públicos* del comportamiento de una forma similar:

```
// foo() es un método público definido dentro de la clase comportamiento
$component->foo();
```

Como puedes ver, aunque `$component` no tiene definida `prop1` y `bar()`, que se pueden utilizar como si son parte de la definición de componentes debido al comportamiento vinculado.

Si dos comportamientos definen la misma propiedad o método y ambos están vinculados con el mismo componente, el comportamiento que ha sido vinculado *primero* tendrá preferencia cuando se esté accediendo a la propiedad o método.

Un comportamiento puede estar asociado con un nombre cuando se une a un componente. Si este es el caso, es posible acceder al objeto de comportamiento mediante el nombre, como se muestra a continuación,

```
$behavior = $component->getBehavior('myBehavior');
```

También puedes acceder a todos los comportamientos vinculados al componente:

```
$behaviors = $component->getBehaviors();
```

5.4.5. Desasociar Comportamientos

Para desasociar un comportamiento, puedes llamar el método `yii\base\Component::detachBehavior()` con el nombre con el que se le asoció:

```
$component->detachBehavior('myBehavior1');
```

También puedes desvincular *todos* los comportamientos:

```
$component->detachBehaviors();
```

5.4.6. Utilizando

Para terminar, vamos a echar un vistazo a `yii\behaviors\TimestampBehavior`. Este comportamiento soporta de forma automática la actualización de atributos timestamp de un modelo `Registro Activo` (Active Record) en cualquier momento donde se guarda el modelo (ej., en la inserción o actualización).

Primero, vincula este comportamiento a la clase `Active Record` que desees utilizar.

```
namespace app\models\User;

use yii\db\ActiveRecord;
use yii\behaviors\TimestampBehavior;

class User extends ActiveRecord
{
    // ...

    public function behaviors()
    {
        return [
            [
                'class' => TimestampBehavior::class,
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at',
                    'updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
            ],
        ];
    }
}
```

La configuración del comportamiento anterior especifica que

- cuando el registro está siendo insertado, el comportamiento debe asignar el sello de tiempo actual a los atributos `created_at` y `updated_at`;
- cuando el registro está siendo actualizado, el comportamiento debe asignar el sello de tiempo actual al atributo `updated_at`.

Ahora si tienes un objeto `User` e intentas guardarlo, descubrirás que sus campos `created_at` y `updated_at` están automáticamente actualizados con el sello de tiempo actual:

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
echo $user->created_at;  // muestra el sello tiempo actual (timestamp)
```

El comportamiento `TimestampBehavior` también ofrece un método muy útil llamado `touch()`, que asigna el sello de tiempo actual a un atributo especificado y lo guarda automáticamente en la base de datos:

```
$user->touch('login_time');
```

5.4.7. Comparación con Traits

Mientras que los comportamientos son similares a traits[3] en cuanto que ambos "inyectan" sus métodos y propiedades a la clase primaria, son diferentes en muchos aspectos. Tal y como se describe abajo, los dos tienen sus ventajas y desventajas. Son más como complementos el uno al otro en lugar de alternativas.

Razones para utilizar comportamientos

Las clases de comportamientos, como todas las clases, soportan herencias. Traits, por otro lado, pueden ser considerados como un copia-y-pega de PHP. Ellos no soportan la herencia de clases.

Los comportamientos pueden ser asociados y desasociados a un componente dinámicamente sin necesidad de que la clase del componente sea modificada. Para usar un trait, debes modificar la clase que la usa.

Los comportamientos son configurables mientras que los traits no.

Los comportamientos pueden personalizar la ejecución de un componente al responder a sus eventos.

Cuando hay un conflicto de nombre entre los diferentes comportamientos vinculados a un mismo componente, el conflicto es automáticamente resuelto respetando al que ha sido asociado primero. El conflicto de nombres en traits requiere que manualmente sean resueltos cambiando el nombre de las propiedades o métodos afectados.

Razones para utilizar los Traits

Los Traits son mucho más eficientes que los comportamientos debido a que los últimos son objetos que consumen tiempo y memoria.

Los IDEs (Programas de desarrollo) son más amigables con traits ya que son una construcción del lenguaje nativo.

[3] https://www.php.net/manual/es/language.oop5.traits.php

5.5. Configuración

Las configuraciones se utilizan ampliamente en Yii al crear nuevos objetos o inicializar los objetos existentes. Las configuraciones por lo general incluyen el nombre de la clase del objeto que se está creando, y una lista de los valores iniciales que deberían ser asignadas a las del propiedades objeto. Las configuraciones también pueden incluir una lista de manipuladores que deban imponerse a del objeto eventos y/o una lista de comportamientos que también ha de atribuirse al objeto.

A continuación, una configuración que se utiliza para crear e inicializar una conexión a base de datos:

```
$config = [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];

$db = Yii::createObject($config);
```

El método `Yii::createObject()` toma una matriz de configuración como su argumento, y crea un objeto intanciando la clase llamada en la configuración. Cuando se crea una instancia del objeto, el resto de la configuración se utilizará para inicializar las propiedades del objeto, controladores de eventos y comportamientos.

Si usted ya tiene un objeto, puede usar `Yii::configure()` para inicializar las propiedades del objeto con una matriz de configuración:

```
Yii::configure($object, $config);
```

Tenga en cuenta que en este caso, la matriz de configuración no debe contener un elemento `class`.

5.5.1. Formato de Configuración

El formato de una configuración se puede describir formalmente como:

```
[
    'class' => 'ClassName',
    'propertyName' => 'propertyValue',
    'on eventName' => $eventHandler,
    'as behaviorName' => $behaviorConfig,
]
```

donde

- El elemento `class` especifica un nombre de clase completo para el objeto que se está creando.

- Los elementos `propertyName` especifica los valores iniciales de la propiedad con nombre. Las claves son los nombres de las propiedades y los valores son los valores iniciales correspondientes. Sólo los miembros de variables públicas y propiedades definidas por getters/setters se pueden configurar.
- Los elementos `on eventName` especifican qué manipuladores deberán adjuntarse al del objeto eventos. Observe que las claves de matriz se forman prefijando nombres de eventos con `on`. Por favor, consulte la sección Eventos para los formatos de controlador de eventos compatibles.
- Los elementos `as behaviorName` especifican qué comportamientos deben adjuntarse al objeto. Observe que las claves de matriz se forman prefijando nombres de comportamiento con `as`; el valor, `$behaviorConfig`, representa la configuración para la creación de un comportamiento, como una configuración normal descrita aquí.

A continuación se muestra un ejemplo de una configuración con los valores de propiedad iniciales, controladores de eventos y comportamientos:

```
[
    'class' => 'app\components\SearchEngine',
    'apiKey' => 'xxxxxxxx',
    'on search' => function ($event) {
        Yii::info("Keyword searched: " . $event->keyword);
    },
    'as indexer' => [
        'class' => 'app\components\IndexerBehavior',
        // ... property init values ...
    ],
]
```

5.5.2. Usando Configuraciones

Las configuraciones se utilizan en muchos lugares en Yii. Al comienzo de esta sección, hemos demostrado cómo crear un objeto según una configuración mediante el uso de `Yii::createObject()`. En este apartado, vamos a describir configuraciones de aplicaciones y configuraciones widget - dos principales usos de configuraciones.

Configuraciones de aplicación

Configuración para una aplicación es probablemente una de las configuraciones más complejas. Esto se debe a que la clase `aplicación` tiene un montón de propiedades y eventos configurables. Más importante aún, su propiedad `componentes` que puede recibir una gran variedad de configuraciones para crear componentes que se registran a través de la aplicación. Lo siguiente es un resumen del archivo de configuración de la aplicación para la plantilla básica de la aplicación.

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
        'log' => [
            'class' => 'yii\log\Dispatcher',
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                ],
            ],
        ],
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=stay2',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
];
```

La configuración no tiene una clave `class`. Esto es porque se utiliza como sigue en un script de entrada, donde ya se le da el nombre de la clase,

```
(new yii\web\Application($config))->run();
```

Para más detalles sobre la configuración de la propiedad `components` de una aplicación se puede encontrar en la sección Aplicación y la sección Localizador de Servicio.

Configuración Widget

Cuando se utiliza widgets, a menudo es necesario utilizar las configuraciones para personalizar las propiedades de widgets. Tanto los metodos `yii\base\Widget::widget()` y `yii\base\Widget::begin()` pueden usarse para crear un widget. Toman un arreglo de configuración, como el siguiente,

```
use yii\widgets\Menu;

echo Menu::widget([
    'activateItems' => false,
    'items' => [
```

```
        ['label' => 'Home', 'url' => ['site/index']],
        ['label' => 'Products', 'url' => ['product/index']],
        ['label' => 'Login', 'url' => ['site/login'], 'visible' =>
        Yii::$app->user->isGuest],
    ],
]);
```

El código anterior crea un widget `Menu` e inicializa su propiedad `activeItems` en falsa. La propiedad `items` también se configura con elementos de menú que se muestran.

Tenga en cuenta que debido a que el nombre de la clase ya está dado, la matriz de configuración no deben tener la clave `class`.

5.5.3. Archivos de Configuración

Cuando una configuración es muy compleja, una práctica común es almacenarla en uno o múltiples archivos PHP, conocidos como *archivos de configuración*. Un archivo de configuración devuelve un array de PHP que representa la configuración. Por ejemplo, es posible mantener una configuración de la aplicación en un archivo llamado `web.php`, como el siguiente,

```
return [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require __DIR__ . '/../vendor/yiisoft/extensions.php',
    'components' => require __DIR__ . '/components.php',
];
```

Debido a que la configuración `componentes` es compleja también, se guarda en un archivo separado llamado `components.php` y "requerir" este archivo en `web.php` como se muestra arriba. El contenido de `components.php` es el siguiente,

```
return [
    'cache' => [
        'class' => 'yii\caching\FileCache',
    ],
    'mailer' => [
        'class' => 'yii\swiftmailer\Mailer',
    ],
    'log' => [
        'class' => 'yii\log\Dispatcher',
        'traceLevel' => YII_DEBUG ? 3 : 0,
        'targets' => [
            [
                'class' => 'yii\log\FileTarget',
            ],
        ],
    ],
    'db' => [
        'class' => 'yii\db\Connection',
```

```
        'dsn' => 'mysql:host=localhost;dbname=stay2',
        'username' => 'root',
        'password' => '',
        'charset' => 'utf8',
    ],
];
```

Para obtener una configuración almacenada en un archivo de configuración, simplemente "requerir" este, como el siguiente:

```
$config = require 'path/to/web.php';
(new yii\web\Application($config))->run();
```

5.5.4. Configuraciones por Defecto

El método `Yii::createObject()` es implementado en base a contenedor de inyección de dependencia. Le permite especificar un conjunto de los llamados *configuraciones predeterminadas* que se aplicarán a todos los casos de las clases especificadas cuando se crean utilizando `Yii::createObject()`. Las configuraciones por defecto se puede especificar llamando `Yii::$container->set()` en el código bootstrapping.

Por ejemplo, si desea personalizar `yii\widgets\LinkPager` para que TODO enlace de búsqueda muestre como máximo 5 botones de página (el valor por defecto es 10), puede utilizar el siguiente código para lograr este objetivo,

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

Sin utilizar las configuraciones predeterminadas, usted tendría que configurar `maxButtonCount` en cada lugar en el que utiliza enlace paginador.

5.5.5. Constantes de Entorno

Las configuraciones a menudo varían de acuerdo al entorno en que se ejecuta una aplicación. Por ejemplo, en el entorno de desarrollo, es posible que desee utilizar una base de datos llamada `mydb_dev`, mientras que en servidor de producción es posible que desee utilizar la base de datos `mydb_prod`. Para facilitar la conmutación de entornos, Yii proporciona una constante llamado `YII_ENV` que se puede definir en el script de entrada de su aplicación. Por ejemplo,

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

Usted puede definir `YII_ENV` como uno de los valores siguientes:

- `prod`: entorno de producción. La constante `YII_ENV_PROD` evaluará como verdadero. Este es el valor por defecto de `YII_ENV` si no esta definida.

- `dev`: entorno de desarrollo. La constante `YII_ENV_DEV` evaluará como verdadero.
- `test`: entorno de pruebas. La constante `YII_ENV_TEST` evaluará como verdadero.

Con estas constantes de entorno, puede especificar sus configuraciones condicionales basado en el entorno actual. Por ejemplo, la configuración de la aplicación puede contener el siguiente código para permitir que el depurador y barra de herramientas de depuración en el entorno de desarrollo.

```
$config = [...];

if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
}

return $config;
```

5.6. Alias

Loa alias son utilizados para representar rutas o URLs de manera que no tengas que escribir explícitamente rutas absolutas o URLs en tu proyecto. Un alias debe comenzar con el signo `@` para ser diferenciado de una ruta normal de archivo y de URLs. Los alias definidos sin el `@` del principio, serán prefijados con el signo `@`.

Yii trae disponibles varios alias predefinidos. Por ejemplo, el alias `@yii` representa la ruta de instalación del framework Yii; `@web` representa la URL base para la aplicación Web ejecutándose.

5.6.1. Definir Alias

Para definir un alias puedes llamar a `Yii::setAlias()` para una determinada ruta de archivo o URL. Por ejemplo,

```
// un alias de una ruta de archivos
Yii::setAlias('@foo', '/path/to/foo');

// una alias de un URL
Yii::setAlias('@bar', 'https://www.example.com');
```

> **Nota:** Una ruta de archivo o URL en alias NO debe necesariamente referirse a un archivo o recurso existente.

Dado un alias, puedes derivar un nuevo alias (sin necesidad de llamar `Yii::setAlias()`) anexando una barra diagonal `/` seguida por uno o varios segmentos de la ruta. Llamamos los alias definidos a través de `Yii::setAlias()`

alias de raíz (root alias), mientras que los alias derivados de ellos *alias derivados* (derived aliases). Por ejemplo, `@foo` es un alias de raíz, mientras que `@foo/bar/file.php` es un alias derivado.

Puedes definir un alias usando otro alias (ya sea un alias de raíz o derivado):

```
Yii::setAlias('@foobar', '@foo/bar');
```

Los alias de raíz están usualmente definidos durante la etapa bootstrapping de la aplicación. Por ejemplo, puedes llamar a `Yii::setAlias()` en el script de entrada. Por conveniencia, Application provee una propiedad modificable llamada `aliases` que puedes configurar en la configuración de la aplicación, como por ejemplo,

```
return [
    // ...
    'aliases' => [
        '@foo' => '/path/to/foo',
        '@bar' => 'https://www.example.com',
    ],
];
```

5.6.2. Resolución de Alias

Puedes llamar `Yii::getAlias()` para resolver un alias de raíz en la ruta o URL que representa. El mismo método puede además resolver un alias derivado en su correspondiente ruta de archivo o URL. Por ejemplo,

```
echo Yii::getAlias('@foo');               // muestra: /path/to/foo
echo Yii::getAlias('@bar');               // muestra:
https://www.example.com
echo Yii::getAlias('@foo/bar/file.php'); // muestra:
/path/to/foo/bar/file.php
```

La ruta de archivo/URL representado por un alias derivado está determinado por la sustitución de la parte de su alias raíz con su correspondiente ruta/Url en el alias derivado.

> **Nota:** El método `Yii::getAlias()` no comprueba si la ruta/URL resultante hacer referencia a un archivo o recurso existente.

Un alias de raíz puede contener carácteres `/`. El método `Yii::getAlias()` es lo suficientemente inteligente para saber qué parte de un alias es un alias de raíz y por lo tanto determinar correctamente la correspondiente ruta de archivo o URL. Por ejemplo,

```
Yii::setAlias('@foo', '/path/to/foo');
Yii::setAlias('@foo/bar', '/path2/bar');
Yii::getAlias('@foo/test/file.php');  // muestra:
/path/to/foo/test/file.php
Yii::getAlias('@foo/bar/file.php');   // muestra: /path2/bar/file.php
```

Si `@foo/bar` no está definido como un alias de raíz, la última declaración mostraría `/path/to/foo/bar/file.php`.

5.6.3. Usando Alias

Los alias son utilizados en muchos lugares en Yii sin necesidad de llamar `Yii::getAlias()` para convertirlos en rutas/URLs. Por ejemplo, `yii\caching\FileCache::$cachePath` puede aceptar tanto una ruta de archivo como un alias que represente la ruta de archivo, gracias al prefijo `@` el cual permite diferenciar una ruta de archivo de un alias.

```
use yii\caching\FileCache;

$cache = new FileCache([
    'cachePath' => '@runtime/cache',
]);
```

Por favor, presta atención a la documentación API para ver si una propiedad o el parámetro de un método soporta alias.

5.6.4. Alias Predefinidos

Yii predefine un conjunto de alias para aliviar la necesidad de hacer referencia a rutas de archivo o URLs que son utilizadas regularmente. La siguiente es la lista de alias predefinidos por Yii:

- `@yii`: el directorio donde el archivo `BaseYii.php` se encuentra (también llamado el directorio del framework).
- `@app`: la `ruta base` de la aplicación que se está ejecutando actualmente.
- `@runtime`: la `ruta de ejecución` de la aplicación en ejecución. Por defecto `@app/runtime`.
- `@webroot`: el directorio raíz Web de la aplicación Web se está ejecutando actualmente.
- `@web`: la URL base de la aplicación web se ejecuta actualmente. Tiene el mismo valor que `yii\web\Request::$baseUrl`.
- `@vendor`: el `directorio vendor de Composer`. Por defecto `@app/vendor`.
- `@bower`, el directorio raíz que contiene paquetes bower[4]. Por defecto `@vendor/bower`.
- `@npm`, el directorio raíz que contiene paquetes npm[5]. Por defecto `@vendor/npm`.

El alias `@yii` se define cuando incluyes el archivo `Yii.php` en tu script de entrada, mientras que el resto de los alias están definidos en el constructor de la aplicación cuando se aplica la configuración de la aplicación.

[4] https://bower.io/
[5] https://www.npmjs.com/

5.6.5. Alias en Extensiones

Un alias se define automáticamente por cada extensión que ha sido instalada a través de Composer. El alias es nombrado tras el `namespace` de raíz de la extensión instalada tal y como está declarada en su archivo `composer.json`, y representa el directorio raíz de la extensión. Por ejemplo, si instalas la extensión `yiisoft/yii2-jui`, tendrás automáticamente definido el alias `@yii/jui` durante la etapa bootstrapping de la aplicación:

```
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

5.7. Autocarga de clases

Yii depende del mecanismo de autocarga de clases[6] para localizar e incluir los archivos de las clases requiridas. Proporciona un cargador de clases de alto rendimiento que cumple con el estandard PSR-4[7]. El cargador se instala cuando incluyes el archivo `Yii.php`.

> **Nota:** Para simplificar la descripción, en esta sección sólo hablaremos de la carga automática de clases. Sin embargo, ten en cuenta que el contenido que describimos aquí también se aplica a la autocarga de interfaces y rasgos (Traits).

5.7.1. Usando el Autocargador de Yii

Para utilizar el cargador automático de clases de Yii, deberías seguir dos reglas básicas cuando desarrolles y nombres tus clases:

- Cada clase debe estar bajo un espacio de nombre (namespace). Por ejemplo `foo\bar\MyClass`.
- Cada clase debe estar guardada en un archivo individual cuya ruta está determinada por el siguiente algoritmo:

```
// $className es un nombre completo de clase con las iniciales barras
invertidas.
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) .
'.php');
```

Por ejemplo, si el nombre de una clase es `foo\bar\MyClass`, el alias la correspondiente ruta de archivo de la clase sería `@foo/bar/MyClass.php`. Para que este sea capaz de ser resuelto como una ruta de archivo, ya sea `@foo` o `@foo/bar` debe ser un alias de raíz (root alias).

[6] https://www.php.net/manual/es/language.oop5.autoload.php

[7] https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md

Cuando utilizas la Plantilla de Aplicación Básica, puede que pongas tus clases bajo el nivel superior de espacio de nombres `app` para que de esta manera pueda ser automáticamente cargado por Yii sin tener la necesidad de definir un nuevo alias. Esto es porque `@app` es un alias predefinido, y el nombre de una clase tal como `app\components\MyClass` puede ser resuelto en el archivo de la clase `AppBasePath/components/MyClass.php`, de acuerdo con el algoritmo previamente descrito.

En la Plantilla de Aplicación Avanzada, cada nivel tiene su propio alias. Por ejemplo, el nivel `front-end` tiene un alias de raíz `@frontend` mientras que el nivel `back-end` tiene `@backend`. Como resultado, es posible poner las clases `front-end` bajo el espacio de nombres `frontend` mientras que las clases `back-end` pueden hacerlo bajo `backend`. Esto permitirá que estas clases sean automaticamente cargadas por el autocargador de Yii.

5.7.2. Mapa de Clases

El autocargador de clases de Yii soporta el *mapa de clases*, que mapea nombres de clases to sus correpondientes rutas de archvios. Cuando el autocargador esta cargando una clase, primero chequeará si la clase se encuentra en el mapa. Si es así, el correspondiente archivo será incluido directamente sin más comprobación. Esto hace que la clase se cargue muy rápidamente. De hecho, todas las clases de Yii son autocargadas de esta manera.

Puedes añadir una clase al mapa de clases `Yii::$classMap` de la siguiente forma,

```
Yii::$classMap['foo\bar\MyClass'] = 'path/to/MyClass.php';
```

Alias puede ser usado para especificar la ruta de archivos de clases. Deberías iniciar el mapeo de clases en el proceso bootstrapping de la aplicación para que de esta manera el mapa esté listo antes de que tus clases sean usadas.

5.7.3. Usando otros Autocargadores

Debido a que Yii incluye Composer como un gestor de dependencias y extensions, es recomendado que también instales el autocargador de Composer. Si estás usando alguna librería externa que requiere sus autocargadores, también deberías instalarlos.

Cuando se utiliza el cargador de clases automático de Yii conjuntamente con otros autocargadores, deberías incluir el archivo `Yii.php` *después* de que todos los demás autocargadores se hayan instalado. Esto hará que el autocargador de Yii sea el primero en responder a cualquier petición de carga automática de clases. Por ejemplo, el siguiente código ha sido extraido del script de entrada de la Plantilla de Aplicación Básica. La primera línea instala el autocargador de Composer, mientras que la segunda línea instala el autocargador de Yii.

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

Puedes usar el autocargador de Composer sin el autocargador de Yii. Sin embargo, al hacerlo, la eficacia de la carga de tus clases puede que se degrade, y además deberías seguir las reglas establecidas por Composer para que tus clases pudieran ser autocargables.

> **Nota:** Si no deseas utilizar el autocargador de Yii, tendrás que crear tu propia versión del archivo `Yii.php` e incluirlo en tu script de entrada.

5.7.4. Carga Automática de Clases de Extensiones

El autocargador de Yii es capaz de autocargar clases de extensiones. El único requirimiento es que la extensión especifique correctamente la sección de `autoload` (autocarga) en su archivo `composer.json`. Por favor, consulta la documentación de Composer[8] para más detalles acerca de la especificación `autoload`.

En el caso de que no quieras usar el autocargador de Yii, el autocargador de Composer podría cargar las clases de extensiones por tí.

5.8. Localizador de Servicios

Un localizador de servicios es un objeto que sabe cómo proporcionar todo tipo de servicios (o componentes) que puede necesitar una aplicación. Dentro de un localizador de servicios, existe en cada componente como una única instancia, únicamente identificado por un ID. Se utiliza el ID para recuperar un componente desde el localizador de servicios.

En Yii, un localizador de servicio es simplemente una instancia de `yii\di\ServiceLocator`, o de una clase hija.

El localizador de servicio más utilizado en Yii es el objeto *aplicación*, que se puede acceder a través de `\Yii::$app`. Los servicios que prestá son llamadas *componentes de la aplicación*, como los componentes `request`, `response`, and `urlManager`. Usted puede configurar estos componentes, o incluso cambiarlos por sus propias implementaciones fácilmente a través de la funcionalidad proporcionada por el localizador de servicios.

Además del objeto de aplicación, cada objeto módulo es también un localizador de servicios.

Para utilizar un localizador de servicios, el primer paso es registrar los componentes de la misma. Un componente se puede registrar a través de `yii\di\ServiceLocator::set()`. El código siguiente muestra diferentes maneras de registrarse componentes:

[8] https://getcomposer.org/doc/04-schema.md#autoload

```
use yii\di\ServiceLocator;
use yii\caching\FileCache;

$locator = new ServiceLocator;

// register "cache" using a class name that can be used to create a
component
$locator->set('cache', 'yii\caching\ApcCache');

// register "db" using a configuration array that can be used to create a
component
$locator->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=demo',
    'username' => 'root',
    'password' => '',
]);

// register "search" using an anonymous function that builds a component
$locator->set('search', function () {
    return new app\components\SolrService;
});

// register "pageCache" using a component
$locator->set('pageCache', new FileCache);
```

Una vez que el componente se ha registrado, usted puede acceder a él utilizando su ID, en una de las dos formas siguientes:

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

Como puede observarse, `yii\di\ServiceLocator` le permite acceder a un componente como una propiedad utilizando el ID de componente. Cuando acceda a un componente, por primera vez, `yii\di\ServiceLocator` utilizará la información de registro de componente para crear una nueva instancia del componente y devolverlo. Más tarde, si se accede de nuevo al componente, el localizador de servicio devolverá la misma instancia.

Usted puede utilizar `yii\di\ServiceLocator::has()` para comprobar si un ID de componente ya ha sido registrada. Si llama `yii\di\ServiceLocator::get()` con una identificación válida, se produce una excepción.

Debido a que los localizadores de servicios a menudo se crean con configuraciones, se proporciona una propiedad que puede escribir el nombre `components`. Esto le permite configurar y registrar varios componentes a la vez. El siguiente código muestra un arreglo de configuración que se puede utilizar para configurar una aplicación, al mismo tiempo que el registro de la “db”, “cache” y “buscar” componentes:

```
return [
    // ...
```

```
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],
        'cache' => 'yii\caching\ApcCache',
        'search' => function () {
            return new app\components\SolrService;
        },
    ],
];
```

5.9. Contenedor de Inyección de Dependencias

Un contenedor de Inyección de Dependencias (ID), es un objeto que sabe como instancias y configurar objetos y sus objetos dependientes. El articulo de Martin[9] contiene una buena explicación de porque son útiles los contenedores de ID. A continuación explicaremos como usar el contenedor de ID que proporciona Yii.

5.9.1. Inyección de Dependencias

Yii proporciona la función de contenedor de ID mediante la clase `yii\di\Container`. Soporta los siguientes tipos de ID:

- Inyección de constructores;
- Inyección de setters y propiedades;
- Inyección de llamadas de retorno PHP[10];

Inyección de Constructores

El contenedor de ID soporta inyección de constructores con la ayuda de los indicios (hint) de tipo para los parámetros del constructor. Los indicios de tipo le proporcionan información al contenedor para saber cuáles son las clases o interfaces dependientes al usarse para crear un nuevo objeto. El contenedor intentara obtener las instancias de las clases o interfaces dependientes y las inyectará dentro del nuevo objeto mediante el constructor. Por ejemplo,

```
class Foo
{
    public function __construct(Bar $bar)
    {
```

[9] https://martinfowler.com/articles/injection.html
[10] https://www.php.net/manual/es/language.types.callable.php

```
    }
}

$foo = $container->get('Foo');
// que es equivalente a:
$bar = new Bar;
$foo = new Foo($bar);
```

Inyección de Setters y Propiedades

La inyección de setters y propiedades se admite a través de configuraciones. Cuando se registra una dependencia o se crea un nuevo objeto, se puede proporcionar una configuración que usará el contenedor para inyectar las dependencias a través de sus correspondientes setters y propiedades. Por ejemplo,

```
use yii\base\BaseObject;

class Foo extends BaseObject
{
    public $bar;

    private $_qux;

    public function getQux()
    {
        return $this->_qux;
    }

    public function setQux(Qux $qux)
    {
        $this->_qux = $qux;
    }
}

$container->get('Foo', [], [
    'bar' => $container->get('Bar'),
    'qux' => $container->get('Qux'),
]);
```

Inyección de Llamadas de retorno PHP

En este caso, el contenedor usará una llamada de retorno PHP registrada para construir una nueva instancia de una clase. La llamada de retorno se responsabiliza de que dependencias debe inyectar al nuevo objeto creado. Por ejemplo,

```
$container->set('Foo', function ($container, $params, $config) {
    return new Foo(new Bar);
});

$foo = $container->get('Foo');
```

5.9.2. Registro de dependencias

Se puede usar `yii\di\Container::set()` para registrar dependencias. El registro requiere un nombre de dependencia así como una definición de dependencia. Un nombre de dependencia puede ser un nombre de clase, un nombre de interfaz, o un nombre de alias; y una definición de dependencia puede ser un nombre de clase, un array de configuración, o una llamada de retorno PHP.

```
$container = new \yii\di\Container;

// registra un nombre de clase como tal. Puede se omitido.
$container->set('yii\db\Connection');

// registra una interfaz
// Cuando una clase depende de una interfaz, la clase correspondiente
// se instanciará como un objeto dependiente
$container->set('yii\mail\MailInterface', 'yii\swiftmailer\Mailer');

// registra un nombre de alias. Se puede usar $container->get('foo')
// para crear una instancia de Connection
$container->set('foo', 'yii\db\Connection');

// registrar una clase con configuración. La configuración
// se aplicara cuando la clase se instancie por get()
$container->set('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// registra un nombre de alias con configuración de clase
// En este caso, se requiere un elemento "clase" para especificar la clase
$container->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// registra una llamada de retorno de PHP
// La llamada de retorno sera ejecutada cada vez que se ejecute
$container->get('db')
$container->set('db', function ($container, $params, $config) {
    return new \yii\db\Connection($config);
});

// registra un componente instancia
// $container->get('pageCache') devolverá la misma instancia cada vez que se
ejecute
$container->set('pageCache', new FileCache);
```

> **Consejo:** Si un nombre de dependencia es el mismo que la definición de dependencia, no es necesario registrarlo con el contenedor de ID.

Una dependencia registrada mediante `set()` generará una instancia cada vez que se necesite la dependencia. Se puede usar `yii\di\Container::setSingleton()` para registrar una dependencia que genere una única instancia:

```
$container->setSingleton('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

5.9.3. Resolución de Dependencias

Una ves se hayan registrado las dependencias, se puede usar el contenedor de ID para crear nuevos objetos, y el contenedor resolverá automáticamente las dependencias instanciándolas e inyectándolas dentro de los nuevos objetos creados. La resolución de dependencias es recursiva, esto significa que si una dependencia tiene otras dependencias, estas dependencias también se resolverán automáticamente.

Se puede usar `yii\di\Container::get()` para crear nuevos objetos. El método obtiene el nombre de dependencia, que puede ser un nombre de clase, un nombre de interfaz o un nombre de alias. El nombre de dependencia puede estar registrado o no mediante `set()` o `setSingleton()`. Se puede proporcionar opcionalmente un listado de los parámetros del constructor de clase y una configuración para configurar los nuevos objetos creados. Por ejemplo,

```
// "db" ha sido registrado anteriormente como nombre de alias
$db = $container->get('db');

// equivalente a: $engine = new \app\components\SearchEngine($apiKey,
['type' => 1]);
$engine = $container->get('app\components\SearchEngine', [$apiKey], ['type'
=> 1]);
```

Por detrás, el contenedor de ID efectúa mucho más trabajo la creación de un nuevo objeto. El contenedor primero inspeccionará la clase constructora para encontrar los nombres de clase o interfaces dependientes y después automáticamente resolverá estas dependencias recursivamente.

El siguiente código muestra un ejemplo más sofisticado. La clase `UserLister` depende del un objeto que implementa la interfaz `UserFinderInterface`; la clase `UserFinder` implementa la interfaz y depende del objeto `Connection`. Todas estas dependencias se declaran a través de insinuaciones (hinting) de

los parámetros del constructor de clase. Con el registro de dependencia de propiedades, el contenedor de ID puede resolver las dependencias automáticamente y crear una nueva instancia de `UserLister` con una simple llamada a `get('userLister')`.

```
namespace app\models;

use yii\base\BaseObject;
use yii\db\Connection;
use yii\di\Container;

interface UserFinderInterface
{
    function findUser();
}

class UserFinder extends BaseObject implements UserFinderInterface
{
    public $db;

    public function __construct(Connection $db, $config = [])
    {
        $this->db = $db;
        parent::__construct($config);
    }

    public function findUser()
    {
    }
}

class UserLister extends BaseObject
{
    public $finder;

    public function __construct(UserFinderInterface $finder, $config = [])
    {
        $this->finder = $finder;
        parent::__construct($config);
    }
}

$container = new Container;
$container->set('yii\db\Connection', [
    'dsn' => '...',
]);
$container->set('app\models\UserFinderInterface', [
    'class' => 'app\models\UserFinder',
]);
$container->set('userLister', 'app\models\UserLister');

$lister = $container->get('userLister');
```

```
// que es equivalente a:

$db = new \yii\db\Connection(['dsn' => '...']);
$finder = new UserFinder($db);
$lister = new UserLister($finder);
```

5.9.4. Uso Practico

Yii crea un contenedor de ID cuando se incluye el archivo `Yii.php` en el script de entrada de la aplicación. Cuando se llama a `Yii::createObject()` el método realmente llama al contenedor del método `get()` para crear un nuevo objeto. Como se ha comentado anteriormente, el contenedor de ID resolverá automáticamente las dependencias (si las hay) y las inyectará dentro del nuevo objeto creado. Debido a que Yii utiliza `Yii::createObject()` en la mayor parte del núcleo (core) para crear nuevo objetos, podemos personalizar los objetos globalmente para que puedan tratar con `Yii::$container`.

Por ejemplo, se puede personalizar globalmenete el numero predeterminado de números de botones de paginación de `yii\widgets\LinkPager`:

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

Ahora si se usa el widget en una vista con el siguiente código, la propiedad `maxButtonCount` será inicializada con valor 5 en lugar de 10 que es el valor predeterminado definido en la clase.

```
echo \yii\widgets\LinkPager::widget();
```

Se puede sobrescribir el valor establecido mediante el contenedor de ID, como a continuación:

```
echo \yii\widgets\LinkPager::widget(['maxButtonCount' => 20]);
```

Otro ejemplo es aprovechar la ventaja de la inyección automática de constructores de contenedores de ID. Asumiendo que la clase controlador depende de otros objetos, tales como un servicio de reservas de hotel. Se puede declarar una dependencia a través de un parámetro del constructor y permitir al contenedor de ID resolverla por nosotros.

```
namespace app\controllers;

use yii\web\Controller;
use app\components\BookingInterface;

class HotelController extends Controller
{
    protected $bookingService;

    public function __construct($id, $module, BookingInterface
    $bookingService, $config = [])
```

```
    {
        $this->bookingService = $bookingService;
        parent::__construct($id, $module, $config);
    }
}
```

Si se accede al controlador desde el navegador, veremos un error advirtiendo que `BookingInterface` no puede ser instanciada. Esto se debe a que necesitamos indicar al contenedor de ID como tratar con esta dependencia:

```
\Yii::$container->set('app\components\BookingInterface',
'app\components\BookingService');
```

Ahora si se accede al contenedor nuevamente, se creará una instancia de `app\components\BookingService` y se inyectará a como tercer parámetro al constructor del controlador.

5.9.5. Cuando Registrar Dependencias

El registro de dependencias debe hacerse lo antes posible debido a que las dependencias se necesitan cuando se crean nuevos objetos. A continuación se listan practicas recomendadas:

- Siendo desarrolladores de una aplicación, podemos registrar dependencias en el script de entrada o en un script incluido en el script de entrada.
- Siendo desarrolladores de una extension redistribuible, podemos registrar dependencias en la clase de boostraping de la extensión.

5.9.6. Resumen

Tanto la inyección de dependencias como el localizador de servicios son patrones de diseño populares que permiten construir software con acoplamiento flexible y más fácil de testear. Se recomienda encarecida la lectura del articulo de Martin[11] para obtener una mejor comprensión de la inyección de dependencias y de la localización de servicios.

Yii implementa su propio localizador de servicios por encima del contenedor de ID. Cuando un localizador de servicios intenta crear una nueva instancia de objeto, se desviará la llamada al contenedor de ID. Este último resolverá las dependencias automáticamente como se ha descrito anteriormente.

[11] https://martinfowler.com/articles/injection.html

Capítulo 6

Trabajar con bases de datos

6.1. Objetos de Acceso a Bases de Datos

Construido sobre PDO[1], Yii DAO (Objetos de Acceso a Bases de Datos) proporciona una API orientada a objetos para el acceso a bases de datos relacionales. Es el fundamento para otros métodos de acceso a bases de datos más avanzados, incluyendo el constructor de consultas y active record.

Al utilizar Yii DAO, principalmente vas a tratar con SQLs planos y arrays PHP. Como resultado, esta es la manera más eficiente de acceder a las bases de datos. Sin embargo, como la sintaxis puede variar para las diferentes bases de datos, utilizando Yii DAO también significa que tienes que tienes que tomar un esfuerzo adicional para crear una aplicación de database-agnostic.

Yii DAO soporta las siguientes bases de datos:

- MySQL[2]
- MariaDB[3]
- SQLite[4]
- PostgreSQL[5]: versión 8.4 o superior.
- CUBRID[6]: versión 9.3 o superior.
- Oracle[7]
- MSSQL[8]: versión 2008 o superior.

6.1.1. Creando Conexiones DB

Para acceder a una base de datos, primero necesitas conectarte a tu bases de datos mediante la creación de una instancia de `yii\db\Connection`:

[1] https://www.php.net/manual/es/book.pdo.php
[2] https://www.mysql.com/
[3] https://mariadb.com/
[4] https://sqlite.org/
[5] https://www.postgresql.org/
[6] https://www.cubrid.org/
[7] https://www.oracle.com/database/
[8] https://www.microsoft.com/en-us/sqlserver/default.aspx

```
$db = new yii\db\Connection([
    'dsn' => 'mysql:host=localhost;dbname=example',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

Debido a una conexión DB a menudo necesita ser accedido en diferentes lugares, una práctica común es configurarlo en términos de un componente de aplicación como se muestra a continuación:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=example',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
    // ...
];
```

Puedes acceder a la conexión DB mediante la expresión `Yii::$app->db`.

> **Consejo:** Puedes configurar múltiples componentes de aplicación DB si tu aplicación necesita acceder a múltiples bases de datos.

Cuando configuras una conexión DB, deberías siempre especificar el Nombre de Origen de Datos (DSN) mediante la propiedad `dsn`. El formato del DSN varia para cada diferente base de datos. Por favor consulte el manual de PHP[9] para más detalles. Abajo están algunos ejemplos:

- MySQL, MariaDB: `mysql:host=localhost;dbname=mydatabase`
- SQLite: `sqlite:/path/to/database/file`
- PostgreSQL: `pgsql:host=localhost;port=5432;dbname=mydatabase`
- CUBRID: `cubrid:dbname=demodb;host=localhost;port=33000`
- MS SQL Server (mediante sqlsrv driver): `sqlsrv:Server=localhost;Database=mydatabase`
- MS SQL Server (mediante dblib driver): `dblib:host=localhost;dbname=mydatabase`
- MS SQL Server (mediante mssql driver): `mssql:host=localhost;dbname=mydatabase`
- Oracle: `oci:dbname=//localhost:1521/mydatabase`

Nota que si estás conectándote con una base de datos mediante ODBC, deberías configurar la propiedad `yii\db\Connection::$driverName` para que Yii pueda conocer el tipo de base de datos actual. Por ejemplo,

[9] https://www.php.net/manual/es/function.PDO-construct.php

```
'db' => [
    'class' => 'yii\db\Connection',
    'driverName' => 'mysql',
    'dsn' => 'odbc:Driver={MySQL};Server=localhost;Database=test',
    'username' => 'root',
    'password' => '',
],
```

Además de la propiedad `dsn`, a menudo es necesario configurar el `username` y `password`. Por favor consulta `yii\db\Connection` para ver la lista completa de propiedades configurables.

> **Información:** Cuando se crea una instancia de conexión DB, la conexión actual a la base de datos no se establece hasta que ejecutes el primer SQL o llames explícitamente al método `open()`.

6.1.2. Ejecutando Consultas SQL

Una vez tienes instanciada una conexión a la base de datos, se pueden ejecutar consultas SQL tomando los siguientes pasos:

1. Crea un `yii\db\Command` con SQL plano;
2. Vincula parámetros (opcional);
3. Llama a uno de los métodos de ejecución SQL con `yii\db\Command`.

El siguiente ejemplo muestra varias maneras de obtener datos de una base de datos:

```
$db = new yii\db\Connection(...);

// retorna un conjunto de filas. Cada fila es un array asociativo de
columnas de nombres y valores.
// un array vacío es retornado si no hay resultados
$posts = $db->createCommand('SELECT * FROM post')
            ->queryAll();

// retorna una sola fila (la primera fila)
// `false` es retornado si no hay resultados
$post = $db->createCommand('SELECT * FROM post WHERE id=1')
           ->queryOne();

// retorna una sola columna (la primera columna)
// un array vacío es retornado si no hay resultados
$titles = $db->createCommand('SELECT title FROM post')
             ->queryColumn();

// retorna un escalar
// `false` es retornado si no hay resultados
$count = $db->createCommand('SELECT COUNT(*) FROM post')
            ->queryScalar();
```

Nota: Para preservar la precisión, los datos obtenidos de las bases de datos son todos representados como cadenas, incluso si el tipo de columna correspondiente a la base de datos es numérico.

Consejo: Si necesitas ejecutar una consulta SQL inmediatamente después de establecer una conexión (ej., para establecer una zona horaria o un conjunto de caracteres), puedes hacerlo con el evento `yii\db\Connection::EVENT_AFTER_OPEN`. Por ejemplo,

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            // ...
            'on afterOpen' => function($event) {
                // $event->sender se refiere a la conexión DB
                $event->sender->createCommand("SET time_zone =
                'UTC'")->execute();
            }
        ],
    ],
    // ...
];
```

Parámetros Vinculados (Binding Parameters)

Cuando creamos un comando DB para un SQL con parámetros, nosotros deberíamos casi siempre aprovechar el uso de los parámetros vinculados para prevenir los ataques de inyección de SQL. Por ejemplo,

```
$post = $db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status')
           ->bindValue(':id', $_GET['id'])
           ->bindValue(':status', 1)
           ->queryOne();
```

En la sentencia SQL, puedes incrustar uno o múltiples parámetros placeholders (ej. `:id` en el ejemplo anterior). Un parámetro placeholder debería ser una cadena que empiece con dos puntos. A continuación puedes llamar a uno de los siguientes métodos para unir los valores de los parámetros vinculados:

- `bindValue()`: une un solo parámetro
- `bindValues()`: une múltiples parámetros en una sola llamada
- `bindParam()`: similar a `bindValue()` pero también soporta las referencias de parámetros vinculados.

El siguiente ejemplo muestra formas alternativas de vincular parámetros:

```
$params = [':id' => $_GET['id'], ':status' => 1];
```

```
$post = $db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status')
           ->bindValues($params)
           ->queryOne();

$post = $db->createCommand('SELECT * FROM post WHERE id=:id AND
status=:status', $params)
           ->queryOne();
```

La vinculación parámetros es implementada mediante sentencias preparadas (prepared statements)[10]. Además de prevenir ataques de inyección de SQL, también puede mejorar el rendimiento preparando una sola vez una sentencia SQL y ejecutándola múltiples veces con diferentes parámetros. Por ejemplo,

```
$command = $db->createCommand('SELECT * FROM post WHERE id=:id');

$post1 = $command->bindValue(':id', 1)->queryOne();
$post2 = $command->bindValue(':id', 2)->queryOne();
```

Porque `bindParam()` soporta parámetros vinculados por referencias, el código de arriba también puede ser escrito como lo siguiente:

```
$command = $db->createCommand('SELECT * FROM post WHERE id=:id')
              ->bindParam(':id', $id);

$id = 1;
$post1 = $command->queryOne();

$id = 2;
$post2 = $command->queryOne();
```

Observe que vincula el placeholder a la variable `$id` antes de la ejecución, y entonces cambia el valor de esa variable antes de cada subsiguiente ejecución (esto se hace a menudo con bucles). Ejecutando consultas de esta manera puede ser bastante más eficiente que ejecutar una nueva consulta para cada valor diferente del parámetro.

Ejecutando Consultas Non-SELECT

El método `queryXyz()` introducidos en las secciones previas todos tratan con consultas SELECT los cuales recogen los datos de la base de datos. Para las consultas que no devuelven datos, deberías llamar a el método `yii\db\Command::execute()` en su lugar. Por ejemplo,

```
$db->createCommand('UPDATE post SET status=1 WHERE id=1')
   ->execute();
```

[10] https://www.php.net/manual/es/mysqli.quickstart.prepared-statements.php

El método `yii\db\Command::execute()` retorna el número de filas afectadas por la ejecución SQL.

Para consultas INSERT, UPDATE y DELETE, en vez de escribir SQLs planos, puedes llamar a `insert()`, `update()`, `delete()`, respectivamente, construyen los correspondientes SQLs. Estos métodos entrecomillan adecuadamente las tablas y los nombres de columnas y los valores de los parámetros vinculados. Por ejemplo,

```
// INSERT (table name, column values)
$db->createCommand()->insert('user', [
    'name' => 'Sam',
    'age' => 30,
])->execute();

// UPDATE (table name, column values, condition)
$db->createCommand()->update('user', ['status' => 1], 'age >
30')->execute();

// DELETE (table name, condition)
$db->createCommand()->delete('user', 'status = 0')->execute();
```

Puedes también llamar a `batchInsert()` para insertar múltiples filas de una sola vez, que es mucho más eficiente que insertar una fila de cada vez:

```
// table name, column names, column values
$db->createCommand()->batchInsert('user', ['name', 'age'], [
    ['Tom', 30],
    ['Jane', 20],
    ['Linda', 25],
])->execute();
```

6.1.3. Entrecomillado de Tablas y Nombres de Columna

Al escribir código de database-agnostic, entrecomillar correctamente los nombres de las tablas y las columnas es a menudo un dolor de cabeza porque las diferentes bases de datos tienen diferentes reglas para entrecomillar los nombres. Para solventar este problema, puedes usar la siguiente sintaxis de entrecomillado introducido por Yii:

- `[[column name]]`: encierra con dobles corchetes el nombre de una columna que debe ser entrecomillado;
- `{{table name}}`: encierra con dobles llaves el nombre de una tabla que debe ser entrecomillado.

Yii DAO automáticamente convertirá tales construcciones en un SQL con los correspondientes entrecomillados de los nombres de las columnas o tablas. Por ejemplo,

```
// ejecuta esta SQL para MySQL: SELECT COUNT(`id`) FROM `employee`
$count = $db->createCommand("SELECT COUNT([[id]]) FROM {{employee}}")
            ->queryScalar();
```

Usando Prefijos de Tabla

Si la mayoría de tus tablas de BD utilizan algún prefijo común en sus tablas, puedes usar la función de prefijo de tabla soportado por Yii DAO.

Primero, especifica el prefijo de tabla mediante la propiedad `yii\db\Connection::$tablePrefix`:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            // ...
            'tablePrefix' => 'tbl_',
        ],
    ],
];
```

Luego en tu código, siempre que lo necesites para hacer referencia a una tabla cuyo nombre tiene un prefijo, utiliza la sintaxis `{{%table name}}`. El carácter porcentaje se sustituye con el prefijo de la tabla que has especificado en la configuración de la conexión DB. Por ejemplo,

```
// ejecuta esta SQL para MySQL: SELECT COUNT(`id`) FROM `tbl_employee`
$count = $db->createCommand("SELECT COUNT([[id]]) FROM {{%employee}}")
            ->queryScalar();
```

6.1.4. Realización de Transacciones

Cuando se ejecutan múltiples consultas relacionadas en una secuencia, puede que se tengan que envolver en una transacción para asegurar la integridad de los datos y la consistencia de tu base de datos. Si cualquiera de las consultas falla, la base de datos debe ser revertida al estado anterior como si ninguna de estas consultas se haya ejecutado.

El siguiente código muestra una manera típica de usar transacciones:

```
$db->transaction(function($db) {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... ejecutando otras sentencias SQL
});
```

El código de arriba es equivalente a lo siguiente:

```
$transaction = $db->beginTransaction();

try {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... ejecutando otras sentencias SQL
```

```
    $transaction->commit();

} catch(\Exception $e) {

    $transaction->rollBack();

    throw $e;
}
```

Al llamar al método `beginTransaction()`, se inicia una nueva transacción. La transacción se representa como un objeto `yii\db\Transaction` almacenado en la variable `$transaction`. Luego, las consultas que se ejecutan están encerrados en un bloque `try...catch...`. Si todas las consultas son ejecutadas satisfactoriamente, el método `commit()` es llamado para confirmar la transacción. De lo contrario, una excepción se disparará y se capturará, y el método `rollBack()` es llamado para revertir los cambios hechos por las consultas antes de que fallara la consulta en la transacción.

Especificando los Niveles de Aislamiento

Yii también soporta la configuración de [niveles de aislamiento] para tus transacciones. Por defecto, cuando comienza una nueva transacción, utilizará el nivel de aislamiento definido por tu sistema de base de datos. Se puede sobrescribir el nivel de aislamiento por defecto de la siguiente manera,

```
$isolationLevel = \yii\db\Transaction::REPEATABLE_READ;

$db->transaction(function ($db) {
    ....
}, $isolationLevel);

// or alternatively

$transaction = $db->beginTransaction($isolationLevel);
```

Yii proporciona cuatro constantes para los niveles de aislamiento más comunes:

- `yii\db\Transaction::READ_UNCOMMITTED` - el nivel más bajo, pueden ocurrir lecturas Dirty, lecturas Non-repeatable y Phantoms.
- `yii\db\Transaction::READ_COMMITTED` - evita lecturas Dirty.
- `yii\db\Transaction::REPEATABLE_READ` - evita lecturas Dirty y lecturas Non-repeatable.
- `yii\db\Transaction::SERIALIZABLE` - el nivel más fuerte, evita todos los problemas nombrados anteriormente.

Además de usar las constantes de arriba para especificar los niveles de aislamiento, puedes también usar cadenas con una sintaxis valida soportada

por el DBMS que estés usando. Por ejemplo, en PostgreSQL, puedes utilizar `SERIALIZABLE READ ONLY DEFERRABLE`.

Tenga en cuenta que algunos DBMS permiten configuraciones de niveles de aislamiento solo a nivel de conexión. Las transacciones subsiguientes recibirá el mismo nivel de aislamiento , incluso si no se especifica ninguna. Al utilizar esta característica es posible que necesites ajustar el nivel de aislamiento para todas las transacciones de forma explícitamente para evitar conflictos en las configuraciones. En el momento de escribir esto, solo MSSQL y SQLite serán afectadas.

> **Nota:** SQLite solo soporta dos niveles de aislamiento, por lo que solo se puede usar `READ UNCOMMITTED` y `SERIALIZABLE`. El uso de otros niveles causará el lanzamiento de una excepción.

> **Nota:** PostgreSQL no permite configurar el nivel de aislamiento antes que la transacción empiece por lo que no se puede especificar el nivel de aislamiento directamente cuando empieza la transacción. Se tiene que llamar a `yii\db\Transaction::setIsolationLevel()` después de que la transacción haya empezado.

Transacciones Anidadas

Si tu DBMS soporta Savepoint, puedes anidar múltiples transacciones como a continuación:

```
$db->transaction(function ($db) {
    // outer transaction

    $db->transaction(function ($db) {
        // inner transaction
    });
});
```

O alternativamente,

```
$outerTransaction = $db->beginTransaction();
try {
    $db->createCommand($sql1)->execute();

    $innerTransaction = $db->beginTransaction();
    try {
        $db->createCommand($sql2)->execute();
        $innerTransaction->commit();
    } catch (Exception $e) {
        $innerTransaction->rollBack();
    }

    $outerTransaction->commit();
```

```
} catch (Exception $e) {
    $outerTransaction->rollBack();
}
```

6.1.5. Replicación y División Lectura-Escritura

Muchos DBMS soportan replicación de bases de datos[11] para tener una mejor disponibilidad de la base de datos y un mejor tiempo de respuesta del servidor. Con la replicación de bases de datos, los datos están replicados en los llamados *servidores maestros* (master servers) y *servidores esclavos* (slave servers). Todas las escrituras y actualizaciones deben hacerse en el servidor maestro, mientras que las lecturas se efectuarán en los servidores esclavos.

Para aprovechar las ventajas de la replicación de la base de datos y lograr una división de lecuta-escritura, se puede configurar el componente `yii\db\Connection` como se muestra a continuación:

```
[
    'class' => 'yii\db\Connection',

    // configuración para el maestro
    'dsn' => 'dsn for master server',
    'username' => 'master',
    'password' => '',

    // configuración para los esclavos
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // utiliza un tiempo de espera de conexión más pequeña
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // listado de configuraciones de esclavos
    'slaves' => [
        ['dsn' => 'dsn for slave server 1'],
        ['dsn' => 'dsn for slave server 2'],
        ['dsn' => 'dsn for slave server 3'],
        ['dsn' => 'dsn for slave server 4'],
    ],
]
```

La configuración anterior especifica una configuración con un único maestro y múltiples esclavos. Uno de los esclavos se conectará y se usará para ejecutar consultas de lectura, mientras que el maestro se usará para realizar consultas de escritura. De este modo la división de lectura-escritura se logra automáticamente con esta configuración, Por ejemplo,

[11] https://es.wikipedia.org/wiki/Replicaci%C3%B3n_(inform%C3%A1tica)

```php
// crea una instancia de Connection usando la configuración anterior
$db = Yii::createObject($config);

// consulta contra uno de los esclavos
$rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();

// consulta contra el maestro
$db->createCommand("UPDATE user SET username='demo' WHERE id=1")->execute();
```

> **Información:** Las consultas realizadas llamando a `yii\db\Command::execute()` se consideran consultas de escritura, mientras que todas las demás se ejecutan mediante alguno de los métodos "query" de `yii\db\Command` son consultas de lectura. Se puede obtener la conexión de esclavo activa mediante `$db->slave`.

El componente `Connection` soporta el balanceo de carga y la conmutación de errores entre esclavos. Cuando se realiza una consulta de lectura por primera vez, el componente `Connection` elegirá un esclavo aleatorio e intentará realizar una conexión a este. Si el esclavo se encuentra "muerto", se intentará con otro. Si no está disponible ningún esclavo, se conectará al maestro. Configurando una `server status cache`, se recordarán los servidores "muertos" por lo que no se intentará volver a conectar a ellos durante `certain period of time`.

> **Información:** En la configuración anterior, se especifica un tiempo de espera (timeout) de conexión de 10 segundos para cada esclavo. Esto significa que si no se puede conectar a un esclavo en 10 segundos, este será considerado como "muerto". Se puede ajustar el parámetro basado en el entorno actual.

También se pueden configurar múltiples maestros con múltiples esclavos. Por ejemplo,

```php
[
    'class' => 'yii\db\Connection',

    // configuracion habitual para los maestros
    'masterConfig' => [
        'username' => 'master',
        'password' => '',
        'attributes' => [
            // utilizar un tiempo de espera de conexión más pequeña
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // listado de configuraciones de maestros
    'masters' => [
        ['dsn' => 'dsn for master server 1'],
        ['dsn' => 'dsn for master server 2'],
```

```
    ],

    // configuración habitual para esclavos
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // utilizar un tiempo de espera de conexión más pequeña
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // listado de configuración de esclavos
    'slaves' => [
        ['dsn' => 'dsn for slave server 1'],
        ['dsn' => 'dsn for slave server 2'],
        ['dsn' => 'dsn for slave server 3'],
        ['dsn' => 'dsn for slave server 4'],
    ],
]
```

La configuración anterior especifica dos maestros y cuatro esclavos. El componente `Connection` también da soporte al balanceo de carga y la conmutación de errores entre maestros igual que hace con los esclavos. La diferencia es que cuando no se encuentra ningún maestro disponible se lanza una excepción.

> **Nota:** cuando se usa la propiedad `masters` para configurar uno o múltiples maestros, se ignorarán todas las otras propiedades que especifiquen una conexión de base de datos (ej. `dsn`, `username`, `password`), junto con el mismo objeto `Connection`.

Por defecto. las transacciones usan la conexión del maestro. Y dentro de una transacción, todas las operaciones de DB usarán la conexión del maestro. Por ejemplo,

```
// la transacción empieza con la conexión al maestro
$transaction = $db->beginTransaction();

try {
    // las dos consultas se ejecutan contra el maestro
    $rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
    $db->createCommand("UPDATE user SET username='demo' WHERE
    id=1")->execute();

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
}
```

Si se quiere empezar la transacción con una conexión a un esclavo, se debe hacer explícitamente como se muestra a continuación:

```
$transaction = $db->slave->beginTransaction();
```

A veces, se puede querer forzar el uso de una conexión maestra para realizar una consulta de lectura. Se puede lograr usando el método `useMaster()`:

```
$rows = $db->useMaster(function ($db) {
    return $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
});
```

También se puede utilizar directamente estableciendo `$db->enableSlaves` a `false` para que se redirijan todas las consultas a la conexión del maestro.

6.1.6. Trabajando con Esquemas de Bases de Datos

Yii DAO proporciona todo un conjunto de métodos que permites manipular el esquema de tu base de datos, tal como crear nuevas tablas, borrar una columna de una tabla, etc. Estos métodos son listados a continuación:

- `createTable()`: crea una tabla
- `renameTable()`: renombra una tabla
- `dropTable()`: remueve una tabla
- `truncateTable()`: remueve todas las filas de una tabla
- `addColumn()`: añade una columna
- `renameColumn()`: renombra una columna
- `dropColumn()`: remueve una columna
- `alterColumn()`: altera una columna
- `addPrimaryKey()`: añade una clave primaria
- `dropPrimaryKey()`: remueve una clave primaria
- `addForeignKey()`: añade una clave ajena
- `dropForeignKey()`: remueve una clave ajena
- `createIndex()`: crea un indice
- `dropIndex()`: remueve un indice

Estos métodos puedes ser usados como se muestra a continuación:

```
// CREATE TABLE
$db->createCommand()->createTable('post', [
    'id' => 'pk',
    'title' => 'string',
    'text' => 'text',
]);
```

También puedes recuperar la información de definición de una tabla a través del método `getTableSchema()` de una conexión DB. Por ejemplo,

```
$table = $db->getTableSchema('post');
```

El método retorna un objeto `yii\db\TableSchema` que contiene la información sobre las columnas de las tablas, claves primarias, claves ajenas, etc. Toda esta información principalmente es utilizada por el constructor de consultas y active record para ayudar a escribir código database-agnostic.

6.2. Constructor de Consultas

Nota: Esta sección está en desarrollo.

Yii proporciona una capa de acceso básico a bases de datos como se describe en la sección Objetos de Acceso a Bases de Datos. La capa de acceso a bases de datos proporciona un método de bajo nivel (low-level) para interaccionar con la base de datos. Aunque a veces puede ser útil la escritura de sentencias SQLs puras, en otras situaciones puede ser pesado y propenso a errores. Otra manera de tratar con bases de datos puede ser el uso de Constructores de Consultas (Query Builder). El Constructor de Consultas proporciona un medio orientado a objetos para generar las consultas que se ejecutarán.

Un uso típico de Constructor de Consultas puede ser el siguiente:

```
$rows = (new \yii\db\Query())
    ->select('id, name')
    ->from('user')
    ->limit(10)
    ->all();

// que es equivalente al siguiente código:

$query = (new \yii\db\Query())
    ->select('id, name')
    ->from('user')
    ->limit(10);

//  Crear un comando. Se puede obtener la consulta SQL actual utilizando
$command->sql
$command = $query->createCommand();

// Ejecutar el comando:
$rows = $command->queryAll();
```

6.2.1. Métodos de Consulta

Como se puede observar, primero se debe tratar con `yii\db\Query`. En realidad, `Query` sólo se encarga de representar diversa información de la consulta. La lógica para generar la consulta se efectúa mediante `yii\db\QueryBuilder` cuando se llama al método `createCommand()`, y la ejecución de la consulta la efectúa `yii\db\Command`.

Se ha establecido, por convenio, que `yii\db\Query` proporcione un conjunto de métodos de consulta comunes que construirán la consulta, la ejecutarán, y devolverán el resultado. Por ejemplo:

- `all()`: construye la consulta, la ejecuta y devuelve todos los resultados en formato de array.
- `one()`: devuelve la primera fila del resultado.
- `column()`: devuelve la primera columna del resultado.

- `scalar()`: devuelve la primera columna en la primera fila del resultado.
- `exists()`: devuelve un valor indicando si la el resultado devuelve algo.
- `count()`: devuelve el resultado de la consulta `COUNT`. Otros métodos similares incluidos son `sum($q)`, `average($q)`, `max($q)`, `min($q)`, que soportan las llamadas funciones de agregación. El parámetro `$q` es obligatorio en estos métodos y puede ser el nombre de la columna o expresión.

6.2.2. Construcción de Consultas

A continuación se explicará como construir una sentencia SQL que incluya varias clausulas. Para simplificarlo, usamos `$query` para representar el objeto `yii\db\Query`:

Para formar una consulta `SELECT` básica, se necesita especificar que columnas y de que tablas se seleccionarán:

```
$query->select('id, name')
    ->from('user');
```

Las opciones de select se pueden especificar como una cadena de texto (string) separada por comas o como un array. La sintaxis del array es especialmente útil cuando se forma la selección dinámicamente.

```
$query->select(['id', 'name'])
    ->from('user');
```

> **Información:** Se debe usar siempre el formato array si la clausula `SELECT` contiene expresiones SQL. Esto se debe a que una expresión SQL como `CONCAT(first_name, last_name) AS full_name` puede contener comas. Si se junta con otra cadena de texto de otra columna, puede ser que la expresión se divida en varias partes por comas, esto puede conllevar a errores.

Cuando se especifican columnas, se pueden incluir los prefijos de las tablas o alias de columnas, ej. `user.id`, `user.id AS user_id`. Si se usa un array para especificar las columnas, también se pueden usar las claves del array para especificar los alias de columna, ej. `['user_id' => 'user.id', 'user_name' => 'user.name']`.

A partir de la versión 2.0.1, también se pueden seleccionar subconsultas como columnas. Por ejemplo:

```
$subQuery = (new Query)->select('COUNT(*)')->from('user');
$query = (new Query)->select(['id', 'count' => $subQuery])->from('post');
// $query representa la siguiente sentencia SQL:
// SELECT `id`, (SELECT COUNT(*) FROM `user`) AS `count` FROM `post`
```

Para seleccionar filas distintas, se puede llamar a `distinct()`, como se muestra a continuación:

```
$query->select('user_id')->distinct()->from('post');
```

Para especificar de que tabla(s) se quieren seleccionar los datos, se llama a `from()`:

```
$query->select('*')->from('user');
```

Se pueden especificar múltiples tablas usando una cadena de texto separado por comas o un array. Los nombres de tablas pueden contener prefijos de esquema (ej. `'public.user'`) y/o alias de tablas (ej. "user u'). El método entrecomillara automáticamente los nombres de tablas a menos que contengan algún paréntesis (que significa que se proporciona la tabla como una subconsulta o una expresión de BD). Por ejemplo:

```
$query->select('u.*, p.*')->from(['user u', 'post p']);
```

Cuando se especifican las tablas como un array, también se pueden usar las claves de los arrays como alias de tablas (si una tabla no necesita alias, no se usa una clave en formato texto). Por ejemplo:

```
$query->select('u.*, p.*')->from(['u' => 'user', 'p' => 'post']);
```

Se puede especificar una subconsulta usando un objeto `Query`. En este caso, la clave del array correspondiente se usará como alias para la subconsulta.

```
$subQuery = (new Query())->select('id')->from('user')->where('status=1');
$query->select('*')->from(['u' => $subQuery]);
```

Habitualmente se seleccionan los datos basándose en ciertos criterios. El Constructor de Consultas tiene algunos métodos útiles para especificarlos, el más poderoso de estos es `where`, y se puede usar de múltiples formas.

La manera más simple para aplicar una condición es usar una cadena de texto:

```
$query->where('status=:status', [':status' => $status]);
```

Cuando se usan cadenas de texto, hay que asegurarse que se unen los parámetros de la consulta, no crear una consulta mediante concatenación de cadenas de texto. El enfoque anterior es seguro, el que se muestra a continuación, no lo es:

```
$query->where("status=$status"); // Peligroso!
```

En lugar de enlazar los valores de estado inmediatamente, se puede hacer usando `params` o `addParams`:

```
$query->where('status=:status');
$query->addParams([':status' => $status]);
```

Se pueden establecer múltiples condiciones en `where` usando el *formato hash*.

```
$query->where([
    'status' => 10,
    'type' => 2,
    'id' => [4, 8, 15, 16, 23, 42],
]);
```

El código generará la el siguiente SQL:

```
WHERE (`status` = 10) AND (`type` = 2) AND (`id` IN (4, 8, 15, 16, 23, 42))
```

El valor NULO es un valor especial en las bases de datos, y el Constructor de Consultas lo gestiona inteligentemente. Este código:

```
$query->where(['status' => null]);
```

da como resultado la siguiente cláusula WHERE:

```
WHERE (`status` IS NULL)
```

También se pueden crear subconsultas con objetos de tipo `Query` como en el siguiente ejemplo:

```
$userQuery = (new Query)->select('id')->from('user');
$query->where(['id' => $userQuery]);
```

que generará el siguiente código SQL:

```
WHERE `id` IN (SELECT `id` FROM `user`)
```

Otra manera de usar el método es el formato de operando que es `[operator, operand1, operand2, ...]`.

El operando puede ser uno de los siguientes (ver también `yii\db\QueryInterface::where()`):

- `and`: los operandos deben concatenerase usando `AND`. por ejemplo, `['and', 'id=1', 'id=2']` generará `id=1 AND id=2`. Si el operando es un array, se convertirá en una cadena de texto usando las reglas aquí descritas. Por ejemplo, `['and', 'type=1', ['or', 'id=1', 'id=2']]` generará `type=1 AND (id=1 OR id=2)`. El método no ejecutará ningún filtrado ni entrecomillado.

- `or`: similar al operando `and` exceptuando que los operando son concatenados usando `OR`.
- `between`: el operando 1 debe ser el nombre de columna, y los operandos 2 y 3 deben ser los valores iniciales y finales del rango en el que se encuentra la columna. Por ejemplo, `['between', 'id', 1, 10]` generará `id BETWEEN 1 AND 10`.
- `not between`: similar a `between` exceptuando que `BETWEEN` se reemplaza por `NOT BETWEEN` en la condición generada.
- `in`: el operando 1 debe ser una columna o una expresión de BD. El operando 2 puede ser un array o un objeto de tipo `Query`. Generará una condición `IN`. Si el operando 2 es un array, representará el rango de valores que puede albergar la columna o la expresión de BD; Si el operando 2 es un objeto de tipo `Query`, se generará una subconsulta y se usará como rango de la columna o de la expresión de BD. Por ejemplo, `['in', 'id', [1, 2, 3]]` generará `id IN (1, 2, 3)`. El método entrecomillará adecuadamente el nombre de columna y filtrará los valores del rango. El operando `in` también soporta columnas compuestas. En este caso, el operando 1 debe se un array de columnas, mientras que el operando 2 debe ser un array de arrays o un objeto de tipo `Query` que represente el rango de las columnas.
- `not in`: similar que el operando `in` exceptuando que `IN` se reemplaza por `NOT IN` en la condición generada.
- `like`: el operando 1 debe ser una columna o una expresión de BD, y el operando 2 debe ser una cadena de texto o un array que represente los valores a los que tienen que asemejarse la columna o la expresión de BD.Por ejemplo, `['like', 'name', 'tester']` generará `name LIKE '%tester%'`. Cuando se da el valor rango como un array, se generarán múltiples predicados `LIKE` y se concatenaran usando `AND`. Por ejemplo, `['like', 'name', ['test', 'sample']]` generará `name LIKE '%test%' AND name LIKE '%sample%'`. También se puede proporcionar un tercer operando opcional para especificar como deben filtrarse los caracteres especiales en los valores. El operando debe se un array que mapeen los caracteres especiales a sus caracteres filtrados asociados. Si no se proporciona este operando, se aplicará el mapeo de filtrado predeterminado. Se puede usar `false` o un array vacío para indicar que los valores ya están filtrados y no se necesita aplicar ningún filtro. Hay que tener en cuenta que cuando se usa un el mapeo de filtrado (o no se especifica el tercer operando), los valores se encerraran automáticamente entre un par de caracteres de porcentaje.

  Nota: Cuando se usa PostgreSQL también se puede usar `ilike`[12] en lugar de `like` para filtrar resultados insensibles a mayúsculas

[12] https://www.postgresql.org/docs/8.3/static/functions-matching.html#FUNCTIONS-LIKE

(case-insensitive).

- `or like`: similar al operando `like` exceptuando que se usa `OR` para concatenar los predicados `LIKE` cuando haya un segundo operando en un array.
- `not like`: similar al operando `like` exceptuando que se usa `LIKE` en lugar de `NOT LIKE` en las condiciones generadas.
- `or not like`: similar al operando `not like` exceptuando que se usa `OR` para concatenar los predicados `NOT LIKE`.
- `exists`: requiere un operando que debe ser una instancia de `yii\db\Query` que represente la subconsulta. Esto generará una expresión `EXISTS (sub-query)`.
- `not exists`: similar al operando `exists` y genera una expresión `NOT EXISTS (sub-query)`.

Adicionalmente se puede especificar cualquier cosa como operando:

```
$query->select('id')
    ->from('user')
    ->where(['>=', 'id', 10]);
```

Cuyo resultado será:

```
SELECT id FROM user WHERE id >= 10;
```

Si se construyen partes de una condición dinámicamente, es muy convenientes usar `andWhere()` y `orWhere()`:

```
$status = 10;
$search = 'yii';

$query->where(['status' => $status]);
if (!empty($search)) {
    $query->andWhere(['like', 'title', $search]);
}
```

En el caso que `$search` no este vacío, se generará el siguiente código SQL:

```
WHERE (`status` = 10) AND (`title` LIKE '%yii%')
```

Construcción de Condiciones de Filtro Cuando se generan condiciones de filtro basadas en datos recibidos de usuarios (inputs), a menudo se quieren gestionar de forma especial las "datos vacíos" para ignorarlos en los filtros. Por ejemplo, teniendo un formulario HTML que obtiene el nombre de usuario y la dirección de correo electrónico. Si el usuario solo rellena el campo de nombre de usuario, se puede querer generar una consulta para saber si el nombre de usuario recibido es valido. Se puede usar `filterWhere()` para conseguirlo:

```php
// $username y $email son campos de formulario rellenados por usuarios
$query->filterWhere([
    'username' => $username,
    'email' => $email,
]);
```

El método `filterWhere()` es muy similar al método `where()`. La principal diferencia es que el `filterWhere()` eliminará los valores vacíos de las condiciones proporcionadas. Por lo tanto si `$email` es "vació", la consulta resultante será `...WHERE username=:username`; y si tanto `$username` como `$email` son "vacías", la consulta no tendrá `WHERE`.

Decimos que un valor es *vacío* si es nulo, una cadena de texto vacía, una cadena de texto que consista en espacios en blanco o un array vacío.

También se pueden usar `andFilterWhere()` y `orFilterWhere()` para añadir más condiciones de filtro.

Se pueden usar `orderBy` y `addOrderBy` para ordenar resultados:

```php
$query->orderBy([
    'id' => SORT_ASC,
    'name' => SORT_DESC,
]);
```

Aquí estamos ordenando por `id` ascendente y después por `name` descendente.

and

Para añadir `GROUP BY` al SQL generado se puede usar el siguiente código:

```php
$query->groupBy('id, status');
```

Si se quieren añadir otro campo después de usar `groupBy`:

```php
$query->addGroupBy(['created_at', 'updated_at']);
```

Para añadir la condición `HAVING` se pueden usar los métodos `having` y `andHaving` y `orHaving`. Los parámetros para ellos son similares a los del grupo de métodos `where`:

```php
$query->having(['status' => $status]);
```

and

Para limitar el resultado a 10 filas se puede usar `limit`:

```php
$query->limit(10);
```

Para saltarse las 100 primeras filas, se puede usar:

```php
$query->offset(100);
```

Las clausulas `JOIN` se generan en el Constructor de Consultas usando el método join aplicable:

- `innerJoin()`
- `leftJoin()`
- `rightJoin()`

Este left join selecciona los datos desde dos tablas relacionadas en una consulta:

```
$query->select(['user.name AS author', 'post.title as title'])
    ->from('user')
    ->leftJoin('post', 'post.user_id = user.id');
```

En el código, el primer parámetro del método `leftjoin` especifica la tabla a la que aplicar el join. El segundo parámetro, define la condición del join.

Si la aplicación de bases de datos soporta otros tipos de joins, se pueden usar mediante el método `join` genérico:

```
$query->join('FULL OUTER JOIN', 'post', 'post.user_id = user.id');
```

El primer argumento es el tipo de join a realizar. El segundo es la tabla a la que aplicar el join, y el tercero es la condición:

Como en `FROM`, también se pueden efectuar joins con subconsultas. Para hacerlo, se debe especificar la subconsulta como un array que tiene que contener un elemento. El valor del array tiene que ser un objeto de tipo `Query` que represente la subconsulta, mientras que la clave del array es el alias de la subconsulta. Por ejemplo:

```
$query->leftJoin(['u' => $subQuery], 'u.id=author_id');
```

En SQL `UNION` agrega resultados de una consulta a otra consulta. Las columnas devueltas por ambas consultas deben coincidir. En Yii para construirla, primero se pueden formar dos objetos de tipo query y después usar el método `union`:

```
$query = new Query();
$query->select("id, category_id as type, name")->from('post')->limit(10);

$anotherQuery = new Query();
$anotherQuery->select('id, type, name')->from('user')->limit(10);

$query->union($anotherQuery);
```

6.2.3. Consulta por Lotes

Cuando se trabaja con grandes cantidades de datos, los métodos como `yii\db\Query::all()` no son adecuados ya que requieren la carga de todos los datos en memoria. Para mantener los requerimientos de memoria reducidos, Yii proporciona soporte a las llamadas consultas por lotes (batch query). Una consulta por lotes usa un cursor de datos y recupera los datos en bloques.

Las consultas por lotes se pueden usar del siguiente modo:

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
    ->orderBy('id');

foreach ($query->batch() as $users) {
    // $users is an array of 100 or fewer rows from the user table
}

// o si se quieren iterar las filas una a una
foreach ($query->each() as $user) {
    // $user representa uno fila de datos de la tabla user
}
```

Los métodos `yii\db\Query::batch()` y `yii\db\Query::each()` devuelven un objeto `yii\db\BatchQueryResult` que implementa una interfaz `Iterator` y así se puede usar en el constructor `foreach`. Durante la primera iteración, se efectúa una consulta SQL a la base de datos. Desde entonces, los datos se recuperan por lotes en las iteraciones. El tamaño predeterminado de los lotes es 100, que significa que se recuperan 100 filas de datos en cada lote. Se puede modificar el tamaño de los lotes pasando pasando un primer parámetro a los métodos `batch()` o `each()`.

En comparación con `yii\db\Query::all()`, las consultas por lotes sólo cargan 100 filas de datos en memoria cada vez. Si el procesan los datos y después se descartan inmediatamente, las consultas por lotes, pueden ayudar a mantener el uso de memora bajo un limite.

Si se especifica que el resultado de la consulta tiene que ser indexado por alguna columna mediante `yii\db\Query::indexBy()`, las consultas por lotes seguirán manteniendo el indice adecuado. Por ejemplo,

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
    ->indexBy('username');

foreach ($query->batch() as $users) {
    // $users esta indexado en la columna "username"
```

```
}

foreach ($query->each() as $username => $user) {
}
```

Error: not existing file: db-active-record.md

6.3. Migración de Base de Datos

Durante el curso de desarrollo y mantenimiento de una aplicación con base de datos, la estructura de dicha base de datos evoluciona tanto como el código fuente. Por ejemplo, durante el desarrollo de una aplicación, una nueva tabla podría ser necesaria; una vez que la aplicación se encuentra en producción, podría descrubrirse que debería crearse un índice para mejorar el tiempo de ejecución de una consulta; y así sucesivamente. Debido a los cambios en la estructura de la base de datos a menudo se requieren cambios en el código, Yii soporta la característica llamada *migración de base de datos*, la cual permite tener un seguimiento de esos cambios en término de *migración de base de datos*, cuyo versionado es controlado junto al del código fuente.

Los siguientes pasos muestran cómo una migración puede ser utilizada por un equipo durante el desarrollo:

1. Tim crea una nueva migración (por ej. crea una nueva table, cambia la definición de una columna, etc.).
2. Tim hace un commit con la nueva migración al sistema de control de versiones (por ej. Git, Mercurial).
3. Doug actualiza su repositorio desde el sistema de control de versiones y recibe la nueva migración.
4. Doug aplica dicha migración a su base de datos local de desarrollo, de ese modo sincronizando su base de datos y reflejando los cambios que hizo Tim.

Los siguientes pasos muestran cómo hacer una puesta en producción con una migración de base de datos:

1. Scott crea un tag de lanzamiento en el repositorio del proyecto que contiene algunas migraciones de base de datos.
2. Scott actualiza el código fuente en el servidor de producción con el tag de lanzamiento.
3. Scott aplica cualquier migración de base de datos acumulada a la base de datos de producción.

Yii provee un grupo de herramientas de línea de comandos que te permite:

- crear nuevas migraciones;
- aplicar migraciones;
- revertir migraciones;
- re-aplicar migraciones;
- mostrar el historial y estado de migraciones.

Todas esas herramientas son accesibles a través del comando `yii migrate`. En esta sección describiremos en detalle cómo lograr varias tareas utilizando dichas herramientas. Puedes a su vez ver el uso de cada herramienta a través del comando de ayuda `yii help migrate`.

> **Consejo:** las migraciones pueden no sólo afectar un esquema de base de datos sino también ajustar datos existentes para que encajen en el nuevo esquema, crear herencia RBAC o también limpiar el cache.

6.3.1. Creando Migraciones

Para crear una nueva migración, ejecuta el siguiente comando:

```
yii migrate/create <name>
```

El argumento requerido `name` da una pequeña descripción de la nueva migración. Por ejemplo, si la migración se trata acerca de crear una nueva tabla llamada *news*, podrías utilizar el nombre `create_news_table` y ejecutar el siguiente comando:

```
yii migrate/create create_news_table
```

> **Nota:** Debido a que el argumento `name` será utilizado como parte del nombre de clase de la migración generada, sólo debería contener letras, dígitos, y/o guines bajos.

El comando anterior un nuevo archivo de clase PHP llamado `m150101_185401_create_news_table.php` en el directorio `@app/migrations`. El archivo contendrá el siguiente código, que principalmente declara una clase de tipo migración `m150101_185401_create_news_table` con el siguiente esqueleto de código:

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {

    }

    public function down()
    {
        echo "m101129_185401_create_news_table cannot be reverted.\n";

        return false;
    }
```

```
    /*
    // Use safeUp/safeDown to run migration code within a transaction
    public function safeUp()
    {
    }

    public function safeDown()
    {
    }
    */
}
```

Cada migración de base de datos es definida como una clase PHP que extiende de `yii\db\Migration`. La nombre de clase de la migración es generado automáticamente en el formato `m<YYMMDD_HHMMSS>_<Name>`, donde

- `<YYMMDD_HHMMSS>` se refiere a la marca de tiempo UTC en la cual el comando de migración fue ejecutado.
- `<Name>` es el mismo valor del argumento `name` provisto al ejecutar el comando.

En la clase de la migración, se espera que tu escribas código en el método `up()`, que realiza los cambios en la base de datos. Podrías también querer introducir código en el método `down()`, que debería revertir los cambios realizados por `up()`. El método `up()` es llamado cuando actualizas la base de datos con esta migración, mientras que el método `down()` es llamado cuando reviertes dicha migración. El siguiente código muestra cómo podrías implementar la clase de migración para crear la tabla `news`:

```
<?php

use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => Schema::TYPE_PK,
            'title' => Schema::TYPE_STRING . ' NOT NULL',
            'content' => Schema::TYPE_TEXT,
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

Información: No todas las migraciones son reversibles. Por ejem-

> plo, si el método `up()` elimina un registro en una tabla, podrías no ser capáz de recuperarla en el método `down()`. A veces, podrías ser simplemente demasiado perezoso para implementar el método `down()`, debido a que no es muy común revertir migraciones de base de datos. En este caso, deberías devolver `false` en el método `down()` para indicar que dicha migración no es reversible.

La clase de migración de base de datos `yii\db\Migration` expone una conexión a la base de datos mediante la propiedad `db`. Puedes utilizar esto para manipular el esquema de la base de datos utilizando métodos como se describen en Trabajando con Esquemas de Base de Datos.

En vez de utilizar tipos físicos, al crear tablas o columnas deberías utilizar los *tipos abstractos* así las migraciones son independientes de algún DBMS específico. La clase `yii\db\Schema` define un grupo de constantes que representan los tipos abstractos soportados. Dichas constantes son llamadas utilizando el formato de `TYPE_<Name>`. Por ejemplo, `TYPE_PK` se refiere al tipo clave primaria auto-incremental; `TYPE_STRING` se refiere al tipo string. Cuando se aplica una migración a una base de datos en particular, los tipos abstractos serán traducidos a los tipos físicos correspondientes. En el caso de MySQL, `TYPE_PK` será transformado en `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY`, mientras `TYPE_STRING` se vuelve `varchar(255)`.

Puedes agregar restricciones adicionales al utilizar tipos abstractos. En el ejemplo anterior, `NOT NULL` es agregado a `Schema::TYPE_STRING` para especificar que la columna no puede ser `null`.

> **Información:** El mapeo entre tipos abstractos y tipos físicos es especificado en la propiedad `$typeMap` en cada clase concreta `QueryBuilder`.

Desde la versión 2.0.6, puedes hacer uso del recientemente introducido generador de esquemas, el cual provee una forma más conveniente de definir las columnas. De esta manera, la migración anterior podría ser escrita así:

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);
    }
```

```
    public function down()
    {
        $this->dropTable('news');
    }
}
```

Existe una lista de todos los métodos disponibles para la definición de tipos de columna en la API de la documentación de `yii\db\SchemaBuilderTrait`.

6.3.2. Generar Migraciones

Desde la versión 2.0.7 la consola provee una manera muy conveniente de generar migraciones.

Si el nombre de la migración tiene una forma especial, por ejemplo `create_xxx_table` o `drop_xxx_table` entonces el archivo de la migración generada contendrá código extra, en este caso para crear/eliminar tablas. A continuación se describen todas estas variantes.

Crear Tabla

```
yii migrate/create create_post_table
```

esto genera

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey()
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

Para crear las columnas en ese momento, las puedes especificar vía la opción `--fields`.

```
yii migrate/create create_post_table --fields="title:string,body:text"
```

genera

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

Puedes especificar más parámetros para las columnas.

```
yii migrate/create create_post_table
--fields="title:string(12):notNull:unique,body:text"
```

genera

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }
```

```
    /**
     * {@inheritdoc}
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

> **Nota:** la clave primaria es automáticamente agragada y llamada `id` por defecto. Si quieres utilizar otro nombre puedes especificarlo así `--fields="name:primaryKey"`.

Claves Foráneas Desde 2.0.8 el generador soporta claves foráneas utilizando la palabra clave `foreignKey`.

```
yii migrate/create create_post_table
--fields="author_id:integer:notNull:foreignKey(user),category_id:integer:defaultValue(1):foreignKey,title:s
```

genera

```
/**
 * Handles the creation for table `post`.
 * Has foreign keys to the tables:
 *
 * - `user`
 * - `category`
 */
class m160328_040430_create_post_table extends Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'author_id' => $this->integer()->notNull(),
            'category_id' => $this->integer()->defaultValue(1),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);

        // creates index for column `author_id`
        $this->createIndex(
            'idx-post-author_id',
            'post',
            'author_id'
        );

        // add foreign key for table `user`
```

```
        $this->addForeignKey(
            'fk-post-author_id',
            'post',
            'author_id',
            'user',
            'id',
            'CASCADE'
        );

        // creates index for column `category_id`
        $this->createIndex(
            'idx-post-category_id',
            'post',
            'category_id'
        );

        // add foreign key for table `category`
        $this->addForeignKey(
            'fk-post-category_id',
            'post',
            'category_id',
            'category',
            'id',
            'CASCADE'
        );
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        // drops foreign key for table `user`
        $this->dropForeignKey(
            'fk-post-author_id',
            'post'
        );

        // drops index for column `author_id`
        $this->dropIndex(
            'idx-post-author_id',
            'post'
        );

        // drops foreign key for table `category`
        $this->dropForeignKey(
            'fk-post-category_id',
            'post'
        );

        // drops index for column `category_id`
        $this->dropIndex(
            'idx-post-category_id',
            'post'
```

```
        );

        $this->dropTable('post');
    }
}
```

La posición de la palabra clave `foreignKey` en la descripción de la columna no cambia el código generado. Esto significa:

- `author_id:integer:notNull:foreignKey(user)`
- `author_id:integer:foreignKey(user):notNull`
- `author_id:foreignKey(user):integer:notNull`

Todas generan el mismo código.

La palabra clave `foreignKey` puede tomar un parámetro entre paréntesis el cual será el nombre de la tabla relacionada por la clave foránea generada. Si no se pasa ningún parámetro el nombre de la tabla será deducido en base al nombre de la columna.

En el ejemplo anterior `author_id:integer:notNull:foreignKey(user)` generará una columna llamada `author_id` con una clave foránea a la tabla `user` mientras `category_id:integer:defaultValue(1):foreignKey` generará `category_id` con una clave foránea a la tabla `category`.

Eliminar Tabla

```
yii migrate/create drop_post_table
--fields="title:string(12):notNull:unique,body:text"
```

genera

```
class m150811_220037_drop_post_table extends Migration
{
    public function up()
    {
        $this->dropTable('post');
    }

    public function down()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }
}
```

Agregar Columna

Si el nombre de la migración está en la forma `add_xxx_column_to_yyy_table` entonces el archivo generado contendrá las declaraciones `addColumn` y `dropColumn` necesarias.

Para agregar una columna:

```
yii migrate/create add_position_column_to_post_table
--fields="position:integer"
```

genera

```
class m150811_220037_add_position_column_to_post_table extends Migration
{
    public function up()
    {
        $this->addColumn('post', 'position', $this->integer());
    }

    public function down()
    {
        $this->dropColumn('post', 'position');
    }
}
```

Eliminar Columna

Si el nombre de la migración está en la forma `drop_xxx_column_from_yyy_table` entonces el archivo generado contendrá las declaraciones `addColumn` y `dropColumn` necesarias.

```
yii migrate/create drop_position_column_from_post_table
--fields="position:integer"
```

genera

```
class m150811_220037_drop_position_column_from_post_table extends Migration
{
    public function up()
    {
        $this->dropColumn('post', 'position');
    }

    public function down()
    {
        $this->addColumn('post', 'position', $this->integer());
    }
}
```

Agregar Tabla de Unión

Si el nombre de la migración está en la forma `create_junction_table_for_xxx_and_yyy_tables` entonces se generará el código necesario para una tabla de unión.

```
yii migrate/create create_junction_table_for_post_and_tag_tables
--fields="created_at:dateTime"
```

genera

```php
/**
 * Handles the creation for table `post_tag`.
 * Has foreign keys to the tables:
 *
 * - `post`
 * - `tag`
 */
class m160328_041642_create_junction_table_for_post_and_tag_tables extends
Migration
{
    /**
     * {@inheritdoc}
     */
    public function up()
    {
        $this->createTable('post_tag', [
            'post_id' => $this->integer(),
            'tag_id' => $this->integer(),
            'created_at' => $this->dateTime(),
            'PRIMARY KEY(post_id, tag_id)',
        ]);

        // creates index for column `post_id`
        $this->createIndex(
            'idx-post_tag-post_id',
            'post_tag',
            'post_id'
        );

        // add foreign key for table `post`
        $this->addForeignKey(
            'fk-post_tag-post_id',
            'post_tag',
            'post_id',
            'post',
            'id',
            'CASCADE'
        );

        // creates index for column `tag_id`
        $this->createIndex(
            'idx-post_tag-tag_id',
            'post_tag',
            'tag_id'
        );

        // add foreign key for table `tag`
        $this->addForeignKey(
            'fk-post_tag-tag_id',
            'post_tag',
            'tag_id',
            'tag',
            'id',
            'CASCADE'
```

```
        );
    }

    /**
     * {@inheritdoc}
     */
    public function down()
    {
        // drops foreign key for table `post`
        $this->dropForeignKey(
            'fk-post_tag-post_id',
            'post_tag'
        );

        // drops index for column `post_id`
        $this->dropIndex(
            'idx-post_tag-post_id',
            'post_tag'
        );

        // drops foreign key for table `tag`
        $this->dropForeignKey(
            'fk-post_tag-tag_id',
            'post_tag'
        );

        // drops index for column `tag_id`
        $this->dropIndex(
            'idx-post_tag-tag_id',
            'post_tag'
        );

        $this->dropTable('post_tag');
    }
}
```

Migraciones Transaccionales

Al ejecutar migraciones complejas de BD, es importante asegurarse que todas las migraciones funcionen o fallen como una unidad así la base de datos puede mantener integridad y consistencia. Para alcanzar este objetivo, se recomienda que encierres las operación de la BD de cada migración en una transacción.

Una manera simple de implementar migraciones transaccionales es poniendo el código de las migraciones en los métodos `safeUp()` y `safeDown()`. Estos métodos se diferencias con `up()` y `down()` en que son encerrados implícitamente en una transacción. Como resultado, si alguna de las operaciones dentro de estos métodos falla, todas las operaciones previas son automáticamente revertidas.

En el siguiente ejemplo, además de crear la tabla `news` también insertamos

un registro inicial dentro de la dicha tabla.

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function safeUp()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);

        $this->insert('news', [
            'title' => 'test 1',
            'content' => 'content 1',
        ]);
    }

    public function safeDown()
    {
        $this->delete('news', ['id' => 1]);
        $this->dropTable('news');
    }
}
```

Ten en cuenta que usualmente cuando ejecutas múltiples operaciones en la BD en `safeUp()`, deberías revertir su orden de ejecución en `safeDown()`. En el ejemplo anterior primero creamos la tabla y luego insertamos la finla en `safeUp()`; mientras que en `safeDown()` primero eliminamos el registro y posteriormente eliminamos la tabla.

> **Nota:** No todos los DBMS soportan transacciones. Y algunas consultas a la BD no pueden ser puestas en transacciones. Para algunos ejemplos, por favor lee acerca de commits implícitos[13]. En estos casos, deberías igualmente implementar `up()` y `down()`.

Métodos de Acceso a la Base de Datos

La clase base `yii\db\Migration` provee un grupo de métodos que te permiten acceder y manipular bases de datos. Podrías encontrar que estos métodos son nombrados de forma similar a los métodos DAO provistos por la clase `yii\db\Command`. Por ejemplo, el método `yii\db\Migration::createTable()` te permite crear una nueva tabla, tal como lo hace `yii\db\Command::createTable()`.

[13] https://dev.mysql.com/doc/refman/5.7/en/implicit-commit.html

El beneficio de utilizar lo métodos provistos por `yii\db\Migration` es que no necesitas explícitamente crear instancias de `yii\db\Command`, y la ejecución de cada método mostrará automáticamente mensajes útiles diciéndote qué operaciones de la base de datos se realizaron y cuánto tiempo tomaron.

Debajo hay una lista de todos los métodos de acceso a la base de datos:

- `execute()`: ejecuta una declaración SQL
- `insert()`: inserta un único registro
- `batchInsert()`: inserta múltiples registros
- `update()`: actualiza registros
- `delete()`: elimina registros
- `createTable()`: crea una nueva tabla
- `renameTable()`: renombra una tabla
- `dropTable()`: elimina una tabla
- `truncateTable()`: elimina todos los registros de una tabla
- `addColumn()`: agrega una columna
- `renameColumn()`: renombra una columna
- `dropColumn()`: elimina una columna
- `alterColumn()`: modifica una columna
- `addPrimaryKey()`: agrega una clave primaria
- `dropPrimaryKey()`: elimina una clave primaria
- `addForeignKey()`: agrega una clave foránea
- `dropForeignKey()`: elimina una clave foránea
- `createIndex()`: crea un índice
- `dropIndex()`: elimina un índice
- `addCommentOnColumn()`: agrega un comentario a una columna
- `dropCommentFromColumn()`: elimina un comentario de una columna
- `addCommentOnTable()`: agrega un comentario a una tabla
- `dropCommentFromTable()`: elimina un comentario de una tabla

Información: `yii\db\Migration` no provee un método de consulta a la base de datos. Esto es porque normalmente no necesitas mostrar mensajes detallados al traer datos de una base de datos. También se debe a que puedes utilizar el poderoso Query Builder para generar y ejecutar consultas complejas.

Nota: Al manipular datos utilizando una migración podrías encontrar que utilizando tus clases Active Record para esto podría ser útil ya que algo de la lógica ya está implementada ahí. Ten en cuenta de todos modos, que en contraste con el código escrito en las migraciones, cuya naturaleza es permanecer constante por siempre, la lógica de la aplicación está sujeta a cambios. Entonces al utilizar Active Record en migraciones, los cambios en la lógica en la capa Active Record podrían accidentalmente romper migraciones existentes. Por esta razón, el código de las migracio-

nes debería permanecer independiente de determinada lógica de la aplicación tal como clases Active Record.

6.3.3. Aplicar Migraciones

To upgrade a database to its latest structure, you should apply all available new migrations using the following command: Para actualizar una base de datos a su última estructura, deberías aplicar todas las nuevas migraciones utilizando el siguiente comando:

```
yii migrate
```

Este comando listará todas las migraciones que no han sido aplicadas hasta el momento. Si confirmas que quieres aplicar dichas migraciones, se correrá el método `up()` o `safeUp()` en cada clase de migración nueva, una tras otra, en el orden de su valor de marca temporal. Si alguna de las migraciones falla, el comando terminará su ejecución sin aplicar el resto de las migraciones.

> **Consejo:** En caso de no disponer de la línea de comandos en el servidor, podrías intentar utilizar la extensión web shell[14].

Por cada migración aplicada correctamente, el comando insertará un registro en la base de datos, en la tabla llamada `migration` para registrar la correcta aplicación de la migración. Esto permitirá a la herramienta de migración identificar cuáles migraciones han sido aplicadas y cuáles no.

> **Información:** La herramienta de migración creará automáticamente la tabla `migration` en la base de datos especificada en la opción `db` del comando. Por defecto, la base de datos es especificada en el componente de aplicación `db`.

A veces, podrías sólo querer aplicar una o algunas pocas migraciones, en vez de todas las migraciones disponibles. Puedes hacer esto el número de migraciones que quieres aplicar al ejecutar el comando. Por ejemplo, el siguiente comando intentará aplicar las tres siguientes migraciones disponibles:

```
yii migrate 3
```

Puedes además explícitamente especificar una migración en particular a la cual la base de datos debería migrar utilizando el comando `migrate/to` de acuerdo a uno de los siguientes formatos:

```
yii migrate/to 150101_185401                    # utiliza la marca
temporal para especificar la migración
yii migrate/to "2015-01-01 18:54:01"            # utiliza un string que
puede ser analizado por strtotime()
```

[14] https://github.com/samdark/yii2-webshell

```
yii migrate/to m150101_185401_create_news_table   # utiliza el nombre
completo
yii migrate/to 1392853618                         # utiliza el tiempo UNIX
```

Si hubiera migraciones previas a la especificada sin aplicar, estas serán aplicadas antes de que la migración especificada sea aplicada.

Si la migración especificada ha sido aplicada previamente, cualquier migración aplicada posteriormente será revertida.

6.3.4. Revertir Migraciones

Para revertir (deshacer) una o varias migraciones ya aplicadas, puedes ejecutar el siguiente comando:

```
yii migrate/down     # revierte la más reciente migración aplicada
yii migrate/down 3   # revierte las 3 últimas migraciones aplicadas
```

> **Nota:** No todas las migraciones son reversibles. Intentar revertir tales migraciones producirá un error y detendrá completamente el proceso de reversión.

6.3.5. Rehacer Migraciones

Rehacer (re-ejecutar) migraciones significa primero revertir las migraciones especificadas y luego aplicarlas nuevamente. Esto puede hacerse de esta manera:

```
yii migrate/redo        # rehace la más reciente migración aplicada
yii migrate/redo 3      # rehace las 3 últimas migraciones aplicadas
```

> **Nota:** Si una migración no es reversible, no tendrás posibilidades de rehacerla.

6.3.6. Listar Migraciones

Para listar cuáles migraciones han sido aplicadas y cuáles no, puedes utilizar los siguientes comandos:

```
yii migrate/history     # muestra las últimas 10 migraciones aplicadas
yii migrate/history 5   # muestra las últimas 5 migraciones aplicadas
yii migrate/history all # muestra todas las migraciones aplicadas

yii migrate/new         # muestra las primeras 10 nuevas migraciones
yii migrate/new 5       # muestra las primeras 5 nuevas migraciones
yii migrate/new all     # muestra todas las nuevas migraciones
```

6.3.7. Modificar el Historial de Migraciones

En vez de aplicar o revertir migraciones, a veces simplemente quieres marcar que tu base de datos ha sido actualizada a una migración en particular. Esto sucede normalmente cuando cambias manualmente la base de datos a un estado particular y no quieres que la/s migración/es de ese cambio sean re-aplicadas posteriormente. Puedes alcanzar este objetivo con el siguiente comando:

```
yii migrate/mark 150101_185401                      # utiliza la marca
temporal para especificar la migración
yii migrate/mark "2015-01-01 18:54:01"              # utiliza un string que
puede ser analizado por strtotime()
yii migrate/mark m150101_185401_create_news_table   # utiliza el nombre
completo
yii migrate/mark 1392853618                         # utiliza el tiempo UNIX
```

El comando modificará la tabla `migration` agregando o eliminado ciertos registros para indicar que en la base de datos han sido aplicadas las migraciones hasta la especificada. Ninguna migración será aplicada ni revertida por este comando.

6.3.8. Personalizar Migraciones

Hay varias maneras de personalizar el comando de migración.

Utilizar Opciones de la Línea de Comandos

El comando de migración trae algunas opciones de línea de comandos que pueden ser utilizadas para personalizar su comportamiento:

- `interactive`: boolean (por defecto `true`), especificar si se debe ejecutar la migración en modo interactivo. Cuando se indica `true`, se le pedirá confirmación al usuario antes de ejecutar ciertas acciones. Puedes querer definirlo como `false` si el comando está siendo utilizado como un proceso de fondo.
- `migrationPath`: string (por defecto `@app/migrations`), especifica el directorio que contiene todos los archivos de clase de las migraciones. Este puede ser especificado tanto como una ruta a un directorio un alias de ruta. Ten en cuenta que el directorio debe existir, o el comando disparará un error.
- `migrationTable`: string (por defecto `migration`), especifica el nombre de la tabla de la base de datos que almacena información del historial de migraciones. Dicha tabla será creada por el comando en caso de que no exista. Puedes también crearla manualmente utilizando la estructura `version varchar(255) primary key, apply_time integer`.

- `db`: string (por defecto `db`), especifica el ID del componente de aplicación de la base de datos. Esto representa la base de datos que será migrada en este comando.
- `templateFile`: string (por defecto `@yii/views/migration.php`), especifica la ruta al template utilizado para generar el esqueleto de los archivos de clases de migración. Puede ser especificado tanto como una ruta a un archivo como una alias de una ruta. El template es un archivo PHP en el cual puedes utilizar una variable predefinida llamada `$className` para obtener el nombre de clase de la migración.
- `generatorTemplateFiles`: array (por defecto '[

  ```
  'create_table' => '@yii/views/createTableMigration.php',
  'drop_table' => '@yii/views/dropTableMigration.php',
  'add_column' => '@yii/views/addColumnMigration.php',
  'drop_column' => '@yii/views/dropColumnMigration.php',
  'create_junction' => '@yii/views/createTableMigration.php'
  ```

  ]'), especifica los templates utilizados para generar las migraciones. Ver "Generar Migraciones" para más detalles.
- `fields`: array de strings de definiciones de columna utilizado por el código de migración. Por defecto `[]`. El formato de cada definición es `COLUMN_NAME:COLUMN_TYPE:COLUMN_DECORATOR`. Por ejemplo, `--fields=name:string(12):notNull` produce una columna string de tamaño 12 que es not `null`.

El siguiente ejemplo muestra cómo se pueden utilizar estas opciones.

Por ejemplo, si queremos migrar un módulo `forum` cuyos arhivos de migración están ubicados dentro del directorio `migrations` del módulo, podemos utilizar el siguientedocs/guide-es/db-migrations.md comando:

```
# realiza las migraciones de un módulo forum sin interacción del usuario
yii migrate --migrationPath=@app/modules/forum/migrations --interactive=0
```

Configurar el Comando Globalmente

En vez de introducir los valores de las opciones cada vez que ejecutas un comandod e migración, podrías configurarlos de una vez por todas en la configuración de la aplicación como se muestra a continuación:

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationTable' => 'backend_migration',
        ],
    ],
];
```

Con esta configuración, cada vez que ejecutes un comando de migración, la tabla `backend_migration` será utilizada para registrar el historial de migraciones. No necesitarás volver a especificarla con la opción `migrationTable` de la línea de comandos.

6.3.9. Migrar Múltiples Bases de Datos

Por defecto, las migraciones son aplicadas en la misma base de datos especificada en el componente de aplicación `db`. Si quieres que sean aplicadas en una base de datos diferente, puedes especificar la opción `db` como se muestra a continuación,

```
yii migrate --db=db2
```

El comando anterior aplicará las migraciones en la base de datos `db2`.

A veces puede suceder que quieras aplicar *algunas* de las migraciones a una base de datos, mientras algunas otras a una base de datos distinta. Para lograr esto, al implementar una clase de migración debes especificar explícitamente el ID del componente DB que la migración debe utilizar, como a continuación:

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function init()
    {
        $this->db = 'db2';
        parent::init();
    }
}
```

La migración anterior se aplicará a `db2`, incluso si especificas una base de datos diferente en la opción `db` de la línea de comandos. Ten en cuenta que el historial aún será registrado in la base de datos especificada en la opción `db` de la línea de comandos.

Si tienes múltiples migraciones que utilizan la misma base de datos, es recomandable que crees una clase base de migración con el código `init()` mostrado. Entonces cada clase de migración puede extender de esa clase base.

> **Consejo:** Aparte de definir la propiedad `db`, puedes también operar en diferentes bases de datos creando nuevas conexiones de base de datos en tus clases de migración. También puedes utilizar métodos DAO con esas conexiones para manipular diferentes bases de datos.

Another strategy that you can take to migrate multiple databases is to keep migrations for different databases in different migration paths. Then you can migrate these databases in separate commands like the following: Otra estrategia que puedes seguir para migrar múltiples bases de datos es mantener las

migraciones para diferentes bases de datos en distintas rutas de migración. Entonces podrías migrar esas bases de datos en comandos separados como a continuación:

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1
yii migrate --migrationPath=@app/migrations/db2 --db=db2
...
```

El primer comando aplicará las migraciones que se encuentran en `@app/migrations/db1` en la base de datos `db1`, el segundo comando aplicará las migraciones que se encuentran en `@app/migrations/db2` en `db2`, y así sucesivamente.

Error: not existing file: db-sphinx.md

Error: not existing file: db-redis.md

Error: not existing file: db-mongodb.md

Error: not existing file: db-elastic-search.md

Capítulo 7

Obtener datos de los usuarios

Error: not existing file: input-forms.md

7.1. Validación de Entrada

Como regla básica, nunca debes confiar en los datos recibidos de un usuario final y deberías validarlo siempre antes de ponerlo en uso.

Dado un modelo poblado con entradas de usuarios, puedes validar esas entradas llamando al método `yii\base\Model::validate()`. Dicho método devolverá un valor booleano indicando si la validación tuvo éxito o no. En caso de que no, puedes obtener los mensajes de error de la propiedad `yii\base\Model::$errors`. Por ejemplo,

```
$model = new \app\models\ContactForm();

// poblar los atributos del modelo desde la entrada del usuario
$model->load(\Yii::$app->request->post());
// lo que es equivalente a:
// $model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // toda la entrada es válida
} else {
    // la validación falló: $errors es un array que contienen los mensajes
    de error
    $errors = $model->errors;
}
```

7.1.1. Declarar Reglas

Para hacer que `validate()` realmente funcione, debes declarar reglas de validación para los atributos que planeas validar. Esto debería hacerse sobrescribiendo el método `yii\base\Model::rules()`. El siguiente ejemplo muestra cómo son declaradas las reglas de validación para el modelo `ContactForm`:

```
public function rules()
{
    return [
        // los atributos name, email, subject y body son obligatorios
        [['name', 'email', 'subject', 'body'], 'required'],

        // el atributo email debe ser una dirección de email válida
        ['email', 'email'],
    ];
}
```

El método `rules()` debe devolver un array de reglas, la cual cada una tiene el siguiente formato:

```
[
    // requerido, especifica qué atributos deben ser validados por esta
    regla.
    // Para un sólo atributo, puedes utilizar su nombre directamente
```

```
    // sin tenerlo dentro de un array
    ['attribute1', 'attribute2', ...],

    // requerido, especifica de qué tipo es la regla.
    // Puede ser un nombre de clase, un alias de validador, o el nombre de
    un método de validación
    'validator',

    // opcional, especifica en qué escenario/s esta regla debe aplicarse
    // si no se especifica, significa que la regla se aplica en todos los
    escenarios
    // Puedes también configurar la opción "except" en caso de que quieras
    aplicar la regla
    // en todos los escenarios salvo los listados
    'on' => ['scenario1', 'scenario2', ...],

    // opcional, especifica atributos adicionales para el objeto validador
    'property1' => 'value1', 'property2' => 'value2', ...
]
```

Por cada regla debes especificar al menos a cuáles atributos aplica la regla y cuál es el tipo de la regla. Puedes especificar el tipo de regla de las siguientes maneras:

- el alias de un validador propio del framework, tal como `required`, `in`, `date`, etc. Por favor consulta Validadores del núcleo para la lista completa de todos los validadores incluidos.
- el nombre de un método de validación en la clase del modelo, o una función anónima. Consulta la subsección Validadores en Línea para más detalles.
- el nombre completo de una clase de validador. Por favor consulta la subsección Validadores Independientes para más detalles.

Una regla puede ser utilizada para validar uno o varios atributos, y un atributo puede ser validado por una o varias reglas. Una regla puede ser aplicada en ciertos escenarios con tan sólo especificando la opción `on`. Si no especificas una opción `on`, significa que la regla se aplicará en todos los escenarios.

Cuando el método `validate()` es llamado, este sigue los siguientes pasos para realiza la validación:

1. Determina cuáles atributos deberían ser validados obteniendo la lista de atributos de `yii\base\Model::scenarios()` utilizando el `scenario` actual. Estos atributos son llamados *atributos activos*.

2. Determina cuáles reglas de validación deberían ser validados obteniendo la lista de reglas de `yii\base\Model::rules()` utilizando el `scenario` actual. Estas reglas son llamadas *reglas activas*.

3. Utiliza cada regla activa para validar cada atributo activo que esté asociado a la regla. Las reglas de validación son evaluadas en el orden en que están listadas.

De acuerdo a los pasos de validación mostrados arriba, un atributo será validado si y sólo si es un atributo activo declarado en `scenarios()` y está asociado a una o varias reglas activas declaradas en `rules()`.

> **Nota:** Es práctico darle nombre a las reglas, por ej:

```
public function rules()
{
    return [
        // ...
        'password' => [['password'], 'string', 'max' => 60],
    ];
}
```

Puedes utilizarlas en una subclase del modelo:

```
public function rules()
{
    $rules = parent::rules();
    unset($rules['password']);
    return $rules;
}
```

Personalizar Mensajes de Error

La mayoría de los validadores tienen mensajes de error por defecto que serán agregados al modelo siendo validado cuando sus atributos fallan la validación. Por ejemplo, el validador `required` agregará el mensaje "Username no puede estar vacío." a un modelo cuando falla la validación del atributo `username` al utilizar esta regla.

Puedes especificar el mensaje de error de una regla especificado la propiedad `message` al declarar la regla, como a continuación,

```
public function rules()
{
    return [
        ['username', 'required', 'message' => 'Por favor escoge un nombre de
        usuario.'],
    ];
}
```

Algunos validadores pueden soportar mensajes de error adicionales para describir más precisamente las causas del fallo de validación. Por ejemplo, el validador `number` soporta `tooBig` y `tooSmall` para describir si el fallo de validación es porque el valor siendo validado es demasiado grande o demasiado pequeño, respectivamente. Puedes configurar estos mensajes de error tal como cualquier otroa propiedad del validador en una regla de validación.

Eventos de Validación

Cuando el método `yii\base\Model::validate()` es llamado, este llamará a dos métodos que puedes sobrescribir para personalizar el proceso de validación:

- `yii\base\Model::beforeValidate()`: la implementación por defecto lanzará un evento `yii\base\Model::EVENT_BEFORE_VALIDATE`. Puedes tanto sobrescribir este método o responder a este evento para realizar algún trabajo de pre procesamiento (por ej. normalizar datos de entrada) antes de que ocurra la validación en sí. El método debe devolver un booleano que indique si la validación debe continuar o no.
- `yii\base\Model::afterValidate()`: la implementación por defecto lanzará un evento `yii\base\Model::EVENT_AFTER_VALIDATE`. uedes tanto sobrescribir este método o responder a este evento para realizar algún trabajo de post procesamiento después de completada la validación.

Validación Condicional

Para validar atributos sólo en determinadas condiciones, por ej. la validación de un atributo depende del valor de otro atributo puedes utilizar la propiedad `when` para definir la condición. Por ejemplo,

```
['state', 'required', 'when' => function($model) {
    return $model->country == 'USA';
}]
```

La propiedad `when` toma un método invocable PHP con la siguiente firma:

```
/**
 * @param Model $model el modelo siendo validado
 * @param string $attribute al atributo siendo validado
 * @return bool si la regla debe ser aplicada o no
 */
function ($model, $attribute)
```

Si también necesitas soportar validación condicional del lado del cliente, debes configurar la propiedad `whenClient`, que toma un string que representa una función JavaScript cuyo valor de retorno determina si debe aplicarse la regla o no. Por ejemplo,

```
['state', 'required', 'when' => function ($model) {
    return $model->country == 'USA';
}, 'whenClient' => "function (attribute, value) {
    return $('#country').val() == 'USA';
}"]
```

Filtro de Datos

La entrada del usuario a menudo debe ser filtrada o pre procesada. Por ejemplo, podrías querer eliminar los espacions alrededor de la entrada `username`. Puedes utilizar reglas de validación para lograrlo.

Los siguientes ejemplos muestran cómo eliminar esos espacios en la entrada y cómo transformar entradas vacías en `null` utilizando los validadores del framework trim y default:

```
return [
    [['username', 'email'], 'trim'],
    [['username', 'email'], 'default'],
];
```

También puedes utilizar el validador más general filter para realizar filtros de datos más complejos.

Como puedes ver, estas reglas de validación no validan la entrada realmente. En cambio, procesan los valores y los guardan en el atributo siendo validado.

Manejando Entradas Vacías

Cuando los datos de entrada son enviados desde formularios HTML, a menudo necesitas asignar algunos valores por defecto a las entradas si estas están vacías. Puedes hacerlo utilizando el validador default. Por ejemplo,

```
return [
    // convierte "username" y "email" en `null` si estos están vacíos
    [['username', 'email'], 'default'],

    // convierte "level" a 1 si está vacío
    ['level', 'default', 'value' => 1],
];
```

Por defecto, una entrada se considera vacía si su valor es un string vacío, un array vacío o `null`. Puedes personalizar la lógica de detección de valores vacíos configurando la propiedad `yii\validators\Validator::isEmpty()` con una función PHP invocable. Por ejemplo,

```
['agree', 'required', 'isEmpty' => function ($value) {
    return empty($value);
}]
```

> **Nota:** La mayoría de los validadores no manejan entradas vacías si su propiedad `yii\validators\Validator::$skipOnEmpty` toma el valor por defecto `true`. Estas serán simplemente salteadas durante la validación si sus atributos asociados reciben una entrada vacía. Entre los validadores del framework, sólo `captcha`, `default`, `filter`, `required`, y `trim` manejarán entradas vacías.

7.1.2. Validación Ad Hoc

A veces necesitas realizar *validación ad hoc* para valores que no están ligados a ningún modelo.

Si sólo necesitas realizar un tipo de validación (por ej: validar direcciones de email), podrías llamar al método `validate()` de los validadores deseados, como a continuación:

```
$email = 'test@example.com';
$validator = new yii\validators\EmailValidator();

if ($validator->validate($email, $error)) {
    echo 'Email válido.';
} else {
    echo $error;
}
```

> **Nota:** No todos los validadores soportan este tipo de validación. Un ejemplo es el validador del framework unique, que está diseñado para trabajar sólo con un modelo.

Si necesitas realizar varias validaciones contro varios valores, puedes utilizar `yii\base\DynamicModel`, que soporta declarar tanto los atributos como las reglas sobre la marcha. Su uso es como a continuación:

```
public function actionSearch($name, $email)
{
    $model = DynamicModel::validateData(compact('name', 'email'), [
        [['name', 'email'], 'string', 'max' => 128],
        ['email', 'email'],
    ]);

    if ($model->hasErrors()) {
        // validación fallida
    } else {
        // validación exitosa
    }
}
```

El método `yii\base\DynamicModel::validateData()` crea una instancia de `DynamicModel`, define los atributos utilizando los datos provistos (`name` e `email` en este ejemplo), y entonces llama a `yii\base\Model::validate()` con las reglas provistas.

Alternativamente, puedes utilizar la sintaxis más "clásica" para realizar la validación ad hoc:

```
public function actionSearch($name, $email)
{
    $model = new DynamicModel(compact('name', 'email'));
    $model->addRule(['name', 'email'], 'string', ['max' => 128])
```

```
        ->addRule('email', 'email')
        ->validate();

    if ($model->hasErrors()) {
        // validación fallida
    } else {
        // validación exitosa
    }
}
```

Después de la validación, puedes verificar si la validación tuvo éxito o no llamando al método `hasErrors()`, obteniendo así los errores de validación de la propiedad `errors`, como haces con un modelo normal. Puedes también acceder a los atributos dinámicos definidos a través de la instancia del modelo, por ej., `$model->name` y `$model->email`.

7.1.3. Crear Validadores

Además de los validadores del framework incluidos en los lanzamientos de Yii, puedes también crear tus propios validadores. Puedes crear validadores en línea o validadores independientes.

Validadores en Línea

Un validador en línea es uno definido en términos del método de un modelo o una función anónima. La firma del método/función es:

```
/**
 * @param string $attribute el atributo siendo validado actualmente
 * @param mixed $params el valor de los "parámetros" dados en la regla
 */
function ($attribute, $params)
```

Si falla la validación de un atributo, el método/función debería llamar a `yii\base\Model::addError()` para guardar el mensaje de error en el modelo de manera que pueda ser recuperado más tarde y presentado a los usuarios finales.

Debajo hay algunos ejemplos:

```
use yii\base\Model;

class MyForm extends Model
{
    public $country;
    public $token;

    public function rules()
    {
        return [
```

```
            // un validador en línea definido como el método del modelo
            validateCountry()
            ['country', 'validateCountry'],

            // un validador en línea definido como una función anónima
            ['token', function ($attribute, $params) {
                if (!ctype_alnum($this->$attribute)) {
                    $this->addError($attribute, 'El token debe contener
                    letras y dígitos.');
                }
            }],
        ];
    }

    public function validateCountry($attribute, $params)
    {
        if (!in_array($this->$attribute, ['USA', 'Web'])) {
            $this->addError($attribute, 'El país debe ser "USA" o "Web".');
        }
    }
}
```

> **Nota:** Por defecto, los validadores en línea no serán aplicados si sus atributos asociados reciben entradas vacías o si alguna de sus reglas de validación ya falló. Si quieres asegurarte de que una regla siempre sea aplicada, puedes configurar las reglas `skipOnEmpty` y/o `skipOnError` como `false` en las declaraciones de las reglas. Por ejemplo:
>
> ```
> [
>     ['country', 'validateCountry', 'skipOnEmpty' => false,
>     'skipOnError' => false],
> ]
> ```

Validadores Independientes

Un validador independiente es una clase que extiende de `yii\validators\Validator` o sus sub clases. Puedes implementar su lógica de validación sobrescribiendo el método `yii\validators\Validator::validateAttribute()`. Si falla la validación de un atributo, llama a `yii\base\Model::addError()` para guardar el mensaje de error en el modelo, tal como haces con los validadores en línea.

Por ejemplo, el validador en línea de arriba podría ser movida a una nueva clase [[components/validators/CountryValidator]].

```
namespace app\components;

use yii\validators\Validator;

class CountryValidator extends Validator
```

```
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['USA', 'Web'])) {
            $this->addError($model, $attribute, 'El país debe ser "USA" o
            "Web".');
        }
    }
}
```

Si quieres que tu validador soporte la validación de un valor sin modelo, deberías también sobrescribir el métodoyii\validators\Validator::validate(). Puedes también sobrescribir yii\validators\Validator::validateValue() en vez de validateAttribute() y validate() porque por defecto los últimos dos métodos son implementados llamando a validateValue().

Debajo hay un ejemplo de cómo podrías utilizar la clase del validador de arriba dentro de tu modelo.

```
namespace app\models;

use Yii;
use yii\base\Model;
use app\components\validators\CountryValidator;

class EntryForm extends Model
{
    public $name;
    public $email;
    public $country;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['country', CountryValidator::class],
            ['email', 'email'],
        ];
    }
}
```

7.1.4. Validación del Lado del Cliente

La validación del lado del cliente basada en JavaScript es deseable cuando la entrada del usuario proviene de formularios HTML, dado que permite a los usuarios encontrar errores más rápido y por lo tanto provee una mejor experiencia. Puedes utilizar o implementar un validador que soporte validación del lado del cliente *en adición a* validación del lado del servidor.

> **Información:** Si bien la validación del lado del cliente es deseable, no es una necesidad. Su principal propósito es proveer al

usuario una mejor experiencia. Al igual que datos de entrada que vienen del los usuarios finales, nunca deberías confiar en la validación del lado del cliente. Por esta razón, deberías realizar siempre la validación del lado del servidor llamando a `yii\base\Model::validate()`, como se describió en las subsecciones previas.

Utilizar Validación del Lado del Cliente

Varios validadores del framework incluyen validación del lado del cliente. Todo lo que necesitas hacer es solamente utilizar `yii\widgets\ActiveForm` para construir tus formularios HTML. Por ejemplo, `LoginForm` mostrado abajo declara dos reglas: una utiliza el validador del framework required, el cual es soportado tanto en lado del cliente como del servidor; y el otro usa el validador en línea `validatePassword`, que es sólo soportado de lado del servidor.

```
namespace app\models;

use yii\base\Model;
use app\models\User;

class LoginForm extends Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // username y password son ambos requeridos
            [['username', 'password'], 'required'],

            // password es validado por validatePassword()
            ['password', 'validatePassword'],
        ];
    }

    public function validatePassword()
    {
        $user = User::findByUsername($this->username);

        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError('password', 'Username o password incorrecto.');
        }
    }
}
```

El formulario HTML creado en el siguiente código contiene dos campos de entrada: `username` y `password`. Si envias el formulario sin escribir nada, encontrarás que los mensajes de error requiriendo que escribas algo aparecen sin que haya comunicación alguna con el servidor.

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('Login') ?>
<?php yii\widgets\ActiveForm::end(); ?>
```

Detrás de escena, `yii\widgets\ActiveForm` leerá las reglas de validación declaradas en el modelo y generará el código JavaScript apropiado para los validadores que soportan validación del lado del cliente. Cuando un usuario cambia el valor de un campo o envia el formulario, se lanzará la validación JavaScript del lado del cliente.

Si quieres deshabilitar la validación del lado del cliente completamente, puedes configurar la propiedad `yii\widgets\ActiveForm::$enableClientValidation` como `false`. También puedes deshabilitar la validación del lado del cliente de campos individuales configurando su propiedad `yii\widgets\ActiveField::$enableClientValidation` como `false`. Cuando `enableClientValidation` es configurado tanto a nivel de campo como a nivel de formulario, tendrá prioridad la primera.

Implementar Validación del Lado del Cliente

Para crear validadores que soportan validación del lado del cliente, debes implementar el método `yii\validators\Validator::clientValidateAttribute()`, que devuelve una pieza de código JavaScript que realiza dicha validación. Dentro del código JavaScript, puedes utilizar las siguientes variables predefinidas:

- `attribute`: el nombre del atributo siendo validado.
- `value`: el valor siendo validado.
- `messages`: un array utilizado para contener los mensajes de error de validación para el atributo.
- `deferred`: un array con objetos diferidos puede ser insertado (explicado en la subsección siguiente).

En el siguiente ejemplo, creamos un `StatusValidator` que valida si la entrada es un status válido contra datos de status existentes. El validador soporta tato tanto validación del lado del servidor como del lado del cliente.

```
namespace app\components;

use yii\validators\Validator;
use app\models\Status;

class StatusValidator extends Validator
{
    public function init()
    {
        parent::init();
        $this->message = 'Entrada de Status Inválida.';
```

```
    }

    public function validateAttribute($model, $attribute)
    {
        $value = $model->$attribute;
        if (!Status::find()->where(['id' => $value])->exists()) {
            $model->addError($attribute, $this->message);
        }
    }

    public function clientValidateAttribute($model, $attribute, $view)
    {
        $statuses =
        json_encode(Status::find()->select('id')->asArray()->column());
        $message = json_encode($this->message, JSON_UNESCAPED_SLASHES |
        JSON_UNESCAPED_UNICODE);
        return <<<JS
if ($.inArray(value, $statuses) === -1) {
    messages.push($message);
}
JS;
    }
}
```

Consejo: El código de arriba muestra principalmente cómo soportar validación del lado del cliente. En la práctica, puedes utilizar el validador del framework in para alcanzar el mismo objetivo. Puedes escribir la regla de validación como a como a continuación:

```
[
    ['status', 'in', 'range' =>
    Status::find()->select('id')->asArray()->column()],
]
```

Consejo: Si necesitas trabajar con validación del lado del cliente manualmente, por ejemplo, agregar campos dinámicamente o realizar alguna lógica de UI, consulta Trabajar con ActiveForm vía JavaScript[1] en el Yii 2.0 Cookbook.

Validación Diferida

Si necesitas realizar validación del lado del cliente asincrónica, puedes crear Objetos Diferidos[2]. Por ejemplo, para realizar validación AJAX personalizada, puedes utilizar el siguiente código:

[1] https://github.com/samdark/yii2-cookbook/blob/master/book/forms-activeform-js.md

[2] https://api.jquery.com/category/deferred-object/

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.push($.get("/check", {value: value}).done(function(data) {
            if ('' !== data) {
                messages.push(data);
            }
        }));
JS;
}
```

Arriba, la variable `deferred` es provista por Yii, y es un array de Objetos Diferidos. El método `$.get()` de jQuery crea un Objeto Diferido, el cual es insertado en el array `deferred`.

Puedes también crear un Objeto Diferito explícitamente y llamar a su método `resolve()` cuando la llamada asincrónica tiene lugar. El siguiente ejemplo muestra cómo validar las dimensiones de un archivo de imagen del lado del cliente.

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        var def = $.Deferred();
        var img = new Image();
        img.onload = function() {
            if (this.width > 150) {
                messages.push('Imagen demasiado ancha!!');
            }
            def.resolve();
        }
        var reader = new FileReader();
        reader.onloadend = function() {
            img.src = reader.result;
        }
        reader.readAsDataURL(file);

        deferred.push(def);
JS;
}
```

> **Nota:** El método `resolve()` debe ser llamado después de que el atributo ha sido validado. De otra manera la validación principal del formulario no será completada.

Por simplicidad, el array `deferred` está equipado con un método de atajo, `add()`, que automáticamente crea un Objeto Diferido y lo agrega al array `deferred`. Utilizando este método, puedes simplificar el ejemplo de arriba de esta manera,

```
public function clientValidateAttribute($model, $attribute, $view)
{
```

```
    return <<<JS
        deferred.add(function(def) {
            var img = new Image();
            img.onload = function() {
                if (this.width > 150) {
                    messages.push('Imagen demasiado ancha!!');
                }
                def.resolve();
            }
            var reader = new FileReader();
            reader.onloadend = function() {
                img.src = reader.result;
            }
            reader.readAsDataURL(file);
        });
JS;
}
```

Validación AJAX

Algunas validaciones sólo pueden realizarse del lado del servidor, debido a que sólo el servidor tiene la información necesaria. Por ejemplo, para validar si un nombre de usuario es único o no, es necesario revisar la tabla de usuarios del lado del servidor. Puedes utilizar validación basada en AJAX en este caso. Esta lanzará una petición AJAX de fondo para validar la entrada mientras se mantiene la misma experiencia de usuario como en una validación del lado del cliente regular.

Para habilitar la validación AJAX individualmente un campo de entrada, configura la propiedad `enableAjaxValidation` de ese campo como `true` y especifica un único `id` de formulario:

```
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'registration-form',
]);

echo $form->field($model, 'username', ['enableAjaxValidation' => true]);

// ...

ActiveForm::end();
```

Para habiliar la validación AJAX en el formulario entero, configura `enableAjaxValidation` como `true` a nivel del formulario:

```
$form = ActiveForm::begin([
    'id' => 'contact-form',
    'enableAjaxValidation' => true,
]);
```

> **Nota:** Cuando la propiedad `enableAjaxValidation` es configurada tanto a nivel de campo como a nivel de formulario, la primera tendrá prioridad.

Necesitas también preparar el servidor para que pueda manejar las peticiones AJAX. Esto puede alcanzarse con una porción de código como la siguiente en las acciones del controlador:

```
if (Yii::$app->request->isAjax && $model->load(Yii::$app->request->post()))
{
    Yii::$app->response->format = Response::FORMAT_JSON;
    return ActiveForm::validate($model);
}
```

El código de arriba chequeará si la petición actual es AJAX o no. Si lo es, responderá esta petición ejecutando la validación y devolviendo los errores en formato JSON.

> **Información:** Puedes también utilizar Validación Diferida para realizar validación AJAX. De todos modos, la característica de validación AJAX descrita aquí es más sistemática y requiere menos esfuerzo de escritura de código.

Cuando tanto `enableClientValidation` como `enableAjaxValidation` son definidas como `true`, la petición de validación AJAX será lanzada sólo después de una validación del lado del cliente exitosa.

7.2. Subir Archivos

Subir archivos en Yii es normalmente realizado con la ayuda de `yii\web\UploadedFile`, que encapsula cada archivo subido en un objeto `UploadedFile`. Combinado con `yii\widgets\ActiveForm` y modelos, puedes fácilmente implementar un mecanismo seguro de subida de archivos.

7.2.1. Crear Modelos

Al igual que al trabajar con entradas de texto plano, para subir un archivo debes crear una clase de modelo y utilizar un atributo de dicho modelo para mantener la instancia del archivo subido. Debes también declarar una regla para validar la subida del archivo. Por ejemplo,

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
```

```
{
    /**
     * @var UploadedFile
     */
    public $imageFile;

    public function rules()
    {
        return [
            [['imageFile'], 'file', 'skipOnEmpty' => false, 'extensions' =>
            'png, jpg'],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            $this->imageFile->saveAs('uploads/' . $this->imageFile->baseName
            . '.' . $this->imageFile->extension);
            return true;
        } else {
            return false;
        }
    }
}
```

En el código anterior, el atributo `imageFile` es utilizado para mantener una instancia del archivo subido. Este está asociado con una regla de validación `file`, que utiliza `yii\validators\FileValidator` para asegurarse que el archivo a subir tenga extensión `png` o `jpg`. El método `upload()` realizará la validación y guardará el archivo subido en el servidor.

El validador `file` te permite chequear las extensiones, el tamaño, el tipo MIME, etc. Por favor consulta la sección Validadores del Framework para más detalles.

> **Consejo:** Si estás subiendo una imagen, podrías considerar el utilizar el validador `image`. El validador `image` es implementado a través de `yii\validators\ImageValidator`, que verifica que un atributo haya recibido una imagen válida que pueda ser tanto guardada como procesada utilizando la Extensión Imagine[3].

7.2.2. Renderizar Campos de Subida de Archivos

A continuación, crea un campo de subida de archivo en la vista:

```
<?php
use yii\widgets\ActiveForm;
?>
```

[3] https://github.com/yiisoft/yii2-imagine

```
<?php $form = ActiveForm::begin(['options' => ['enctype' =>
'multipart/form-data']]) ?>

    <?= $form->field($model, 'imageFile')->fileInput() ?>

    <button>Enviar</button>

<?php ActiveForm::end() ?>
```

Es importante recordad que agregues la opción `enctype` al formulario para que el archivo pueda ser subido apropiadamente. La llamada a `fileInput()` renderizará un tag `<input type="file">` que le permitirá al usuario seleccionar el archivo a subir.

> **Consejo:** desde la versión 2.0.8, `fileInput` agrega la opción `enctype` al formulario automáticamente cuando se utiliza una campo de subida de archivo.

7.2.3. Uniendo Todo

Ahora, en una acción del controlador, escribe el código que una el modelo y la vista para implementar la subida de archivos:

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFile = UploadedFile::getInstance($model,
            'imageFile');
            if ($model->upload()) {
                // el archivo se subió exitosamente
                return;
            }
        }

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

En el código anterior, cuando se envía el formulario, el método `yii\web\UploadedFile::getInstance()` es llamado para representar el archivo su-

bido como una instancia de `UploadedFile`. Entonces dependemos de la validación del modelo para asegurarnos que el archivo subido es válido y entonces subirlo al servidor.

7.2.4. Uploading Multiple Files

También puedes subir varios archivos a la vez, con algunos ajustes en el código de las subsecciones previas.

Primero debes ajustar la clase del modelo, agregando la opción `maxFiles` en la regla de validación `file` para limitar el número máximo de archivos a subir. Definir `maxFiles` como `0` significa que no hay límite en el número de archivos a subir simultáneamente. El número máximo de archivos permitidos para subir simultáneamente está también limitado por la directiva PHP `max_file_uploads`[4], cuyo valor por defecto es 20. El método `upload()` debería también ser modificado para guardar los archivos uno a uno.

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile[]
     */
    public $imageFiles;

    public function rules()
    {
        return [
            [['imageFiles'], 'file', 'skipOnEmpty' => false, 'extensions' =>
            'png, jpg', 'maxFiles' => 4],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            foreach ($this->imageFiles as $file) {
                $file->saveAs('uploads/' . $file->baseName . '.' .
                $file->extension);
            }
            return true;
        } else {
            return false;
        }
    }
}
```

[4] https://www.php.net/manual/es/ini.core.php#ini.max-file-uploads

En el archivo de la vista, debes agregar la opción `multiple` en la llamada a `fileInput()` de manera que el campo pueda recibir varios archivos:

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' =>
'multipart/form-data']]) ?>

    <?= $form->field($model, 'imageFiles[]')->fileInput(['multiple' =>
true, 'accept' => 'image/*']) ?>

    <button>Enviar</button>

<?php ActiveForm::end() ?>
```

Y finalmente en la acción del controlador, debes llamar `UploadedFile::getInstances()` en vez de `UploadedFile::getInstance()` para asignar un array de instancias `UploadedFile` a `UploadForm::imageFiles`.

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFiles = UploadedFile::getInstances($model,
            'imageFiles');
            if ($model->upload()) {
                // el archivo fue subido exitosamente
                return;
            }
        }

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

Error: not existing file: input-tabular-input.md

7.3. Obtención de datos para los modelos de múltiples

Cuando se trata de algunos datos complejos, es posible que puede que tenga que utilizar varios modelos diferentes para recopilar la entrada del usuario. Por ejemplo, suponiendo que la información de inicio de sesión del usuario se almacena en la tabla `user`, mientras que el perfil de usuario la información se almacena en la tabla `Profile`, es posible que desee para recoger los datos de entrada sobre un usuario a través de un modelo `User` y un modelo `Profile`. Con el modelo de Yii y apoyo formulario, puede solucionar este problema de una manera que no es mucho diferente de la manipulación de un solo modelo.

En lo que sigue, vamos a mostrar cómo se puede crear un formulario que permitirá recoger datos tanto para los modelos `User` y `Profile`.

En primer lugar, la acción del controlador para la recogida de los datos del usuario y del perfil se puede escribir de la siguiente manera,

```
namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use app\models\User;
use app\models\Profile;

class UserController extends Controller
{
    public function actionUpdate($id)
    {
        $user = User::findOne($id);
        if (!$user) {
            throw new NotFoundHttpException("The user was not found.");
        }

        $profile = Profile::findOne($user->profile_id);

        if (!$profile) {
            throw new NotFoundHttpException("The user has no profile.");
        }

        $user->scenario = 'update';
        $profile->scenario = 'update';

        if ($user->load(Yii::$app->request->post()) &&
        $profile->load(Yii::$app->request->post())) {
            $isValid = $user->validate();
            $isValid = $profile->validate() && $isValid;
            if ($isValid) {
                $user->save(false);
                $profile->save(false);
                return $this->redirect(['user/view', 'id' => $id]);
```

```
            }
        }

        return $this->render('update', [
            'user' => $user,
            'profile' => $profile,
        ]);
    }
}
```

En la acción `update`, primero cargamos los modelos `User` y `Profile` que se actualicen desde la base de datos. Luego llamamos `yii\base\Model::load()` para llenar estos dos modelos con la entrada del usuario. Si tiene éxito, se validará los dos modelos y guardarlos. De lo contrario vamos a renderizar la vista `update` que tiene el siguiente contenido:

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'user-update-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($user, 'username') ?>

    ...other input fields...

    <?= $form->field($profile, 'website') ?>

    <?= Html::submitButton('Update', ['class' => 'btn btn-primary']) ?>
<?php ActiveForm::end() ?>
```

Como se puede ver, en el `update` vista que haría que los campos de entrada utilizando dos modelos `User` y `Profile`.

Capítulo 8

Visualizar datos

Error: not existing file: output-formatting.md

8.1. Paginación

Cuando hay muchos datos a mostrar en una sola página, una estrategia común es mostrarlos en varias páginas y en cada una de ellas mostrar sólo una pequeña porción de datos. Esta estrategia es conocida como *paginación*.

Yii utiliza el objeto `yii\data\Pagination` para representar la información acerca del esquema de paginación. En particular,

- `cuenta total` especifica el número total de ítems de datos. Ten en cuenta que este es normalmente un número mucho mayor que el número de ítems necesarios a mostrar en una simple página.
- `tamaño de página` especifica cuántos ítems de datos contiene cada página. El valor por defecto es 20.
- `página actual` da el número de la página actual (comenzando desde 0). El valor por defecto es 0, lo que sería la primera página.

Con un objeto `yii\data\Pagination` totalmente especificado, puedes obtener y mostrar datos en partes. Por ejemplo, si estás recuperando datos de una base de datos, puedes especificar las cláusulas `OFFSET` y `LIMIT` de la consulta a la BD correspondientes a los valores provistos por la paginación. A continuación hay un ejemplo,

```
use yii\data\Pagination;

// construye una consulta a la BD para obtener todos los artículos con
status = 1
$query = Article::find()->where(['status' => 1]);

// obtiene el número total de artículos (pero no recupera los datos de los
artículos todavía)
$count = $query->count();

// crea un objeto paginación con dicho total
$pagination = new Pagination(['totalCount' => $count]);

// limita la consulta utilizando la paginación y recupera los artículos
$articles = $query->offset($pagination->offset)
    ->limit($pagination->limit)
    ->all();
```

¿Qué página de artículos devolverá el ejemplo de arriba? Depende de si se le es pasado un parámetro llamado `page`. Por defecto, la paginación intentará definir la `página actual` con el valor del parámetro `page`. Si el parámetro no es provisto, entonces tomará por defecto el valor 0.

Para facilitar la construcción de elementos UI que soporten paginación, Yii provee el widget `yii\widgets\LinkPager`, que muestra una lista de botones de navegación que el usuario puede presionar para indicar qué página de datos debería mostrarse. El widget toma un objeto de paginación y tal manera conoce cuál es la página actual y cuántos botones debe mostrar. Por ejemplo,

```php
use yii\widgets\LinkPager;

echo LinkPager::widget([
    'pagination' => $pagination,
]);
```

Si quieres construir los elementos de UI manualmente, puedes utilizar `yii\data\Pagination::createUrl()` para generar URLs que dirigirán a las distintas páginas. El método requiere un parámetro de página y generará una URL apropiadamente formada contieniendo el parámetro de página. Por ejemplo,

```php
// especifica la ruta que la URL generada debería utilizar
// Si no lo especificas, se utilizará la ruta de la petición actual
$pagination->route = 'article/index';

// muestra: /index.php?r=article%2Findex&page=100
echo $pagination->createUrl(100);

// muestra: /index.php?r=article%2Findex&page=101
echo $pagination->createUrl(101);
```

> **Consejo:** puedes personalizar el parámetro `page` de la consulta configurando la propiedad `pageParam` al crear el objeto de la paginación.

Error: not existing file: output-sorting.md

8.2. Proveedores de datos

En las secciones sobre paginación y ordenación se describe como permitir a los usuarios finales elegir que se muestre una página de datos en particular, y ordenar los datos por algunas columnas. Como la tarea de paginar y ordenar datos es muy común, Yii proporciona un conjunto de clases *proveedoras de datos* para encapsularla.

Un proveedor de datos es una clase que implementa la interfaz `yii\data\DataProviderInterface`. Básicamente se encarga de obtener datos paginados y ordenados. Normalmente se usa junto con *widgets* de datos para que los usuarios finales puedan paginar y ordenar datos de forma interactiva.

Yii incluye las siguientes clases proveedoras de datos:

- `yii\data\ActiveDataProvider`: usa `yii\db\Query` o `yii\db\ActiveQuery` para consultar datos de bases de datos y devolverlos como *arrays* o instancias Active Record.
- `yii\data\SqlDataProvider`: ejecuta una sentencia SQL y devuelve los datos de la base de datos como *arrays*.
- `yii\data\ArrayDataProvider`: toma un *array* grande y devuelve una rodaja de él basándose en las especificaciones de paginación y ordenación.

El uso de todos estos proveedores de datos comparte el siguiente patrón común:

```
// Crear el proveedor de datos configurando sus propiedades de paginación y
ordenación
$provider = new XyzDataProvider([
    'pagination' => [...],
    'sort' => [...],
]);

// Obtener los datos paginados y ordenados
$models = $provider->getModels();

// Obtener el número de elementos de la página actual
$count = $provider->getCount();

// Obtener el número total de elementos entre todas las páginas
$totalCount = $provider->getTotalCount();
```

Se puede especificar los comportamientos de paginación y ordenación de un proveedor de datos configurando sus propiedades `pagination` y `sort`, que corresponden a las configuraciones para `yii\data\Pagination` y `yii\data\Sort` respectivamente. También se pueden configurar a `false` para inhabilitar las funciones de paginación y/u ordenación.

Los *widgets* de datos, como `yii\grid\GridView`, tienen una propiedad llamada `dataProvider` que puede tomar una instancia de un proveedor de datos y mostrar los datos que proporciona. Por ejemplo,

```
echo yii\grid\GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

Estos proveedores de datos varían principalmente en la manera en que se especifica la fuente de datos. En las siguientes secciones se explica el uso detallado de cada uno de estos proveedores de datos.

8.2.1. Proveedor de datos activo

Para usar `yii\data\ActiveDataProvider`, hay que configurar su propiedad `query`. Puede tomar un objeto [[yii\db\Query] o `yii\db\ActiveQuery`. En el primer caso, los datos devueltos serán *arrays*. En el segundo, los datos devueltos pueden ser *arrays* o instancias de Active Record. Por ejemplo:

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['state_id' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'defaultOrder' => [
            'created_at' => SORT_DESC,
            'title' => SORT_ASC,
        ]
    ],
]);

// Devuelve un array de objetos Post
$posts = $provider->getModels();
```

En el ejemplo anterior, si `$query` se crea el siguiente código, el proveedor de datos devolverá *arrays* en bruto.

```
use yii\db\Query;

$query = (new Query())->from('post')->where(['state' => 1]);
```

> **Nota:** Si una consulta ya tiene la cláusula `orderBy`, las nuevas instrucciones de ordenación dadas por los usuarios finales (mediante la configuración de `sort`) se añadirán a la cláusula `orderBy` previa. Las cláusulas `limit` y `offset` que pueda haber se sobrescribirán por la petición de paginación de los usuarios finales (mediante la configuración de `pagination`).

Por omisión, `yii\data\ActiveDataProvider` usa el componente `db` de la aplicación como conexión con la base de datos. Se puede indicar una conexión con base de datos diferente configurando la propiedad `yii\data\ActiveDataProvider::$db`.

8.2.2. Proveedor de datos SQL

`yii\data\SqlDataProvider` funciona con una sentencia SQL en bruto, que se usa para obtener los datos requeridos. Basándose en las especificaciones de `sort` y `pagination`, el proveedor ajustará las cláusulas `ORDER BY` y `LIMIT` de la sentencia SQL acordemente para obtener sólo la página de datos solicitados en el orden deseado.

Para usar `yii\data\SqlDataProvider`, hay que especificar las propiedades `sql` y `totalCount`. Por ejemplo:

```
use yii\data\SqlDataProvider;

$count = Yii::$app->db->createCommand('
    SELECT COUNT(*) FROM post WHERE status=:status
', [':status' => 1])->queryScalar();

$provider = new SqlDataProvider([
    'sql' => 'SELECT * FROM post WHERE status=:status',
    'params' => [':status' => 1],
    'totalCount' => $count,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => [
            'title',
            'view_count',
            'created_at',
        ],
    ],
]);

// Devuelve un array de filas de datos
$models = $provider->getModels();
```

> **Información:** La propiedad `totalCount` se requiere sólo si se necesita paginar los datos. Esto es porque el proveedor modificará la sentencia SQL especificada vía `sql` para que devuelva sólo la pagina de datos solicitada. El proveedor sigue necesitando saber el número total de elementos de datos para calcular correctamente el número de páginas.

8.2.3. Proveedor de datos de *arrays*

Se recomienda usar `yii\data\ArrayDataProvider` cuando se trabaja con un *array* grande. El proveedor permite devolver una página de los datos del *array* ordenados por una o varias columnas. Para usar `yii\data\ArrayDataProvider`, hay que especificar la propiedad `allModels` como el *array* grande. Los elementos del *array* grande pueden ser *arrays* asociativos (por ejemplo resultados de consultas de DAO u objetos (por ejemplo instancias de Active Record. Por ejemplo:

```
use yii\data\ArrayDataProvider;

$data = [
    ['id' => 1, 'name' => 'name 1', ...],
    ['id' => 2, 'name' => 'name 2', ...],
    ...
    ['id' => 100, 'name' => 'name 100', ...],
];

$provider = new ArrayDataProvider([
    'allModels' => $data,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => ['id', 'name'],
    ],
]);

// Obtener las filas de la página solicitada
$rows = $provider->getModels();
```

> **Nota:** En comparación con Active Data Provider y SQL Data Provider, Array Data Provider es menos eficiente porque requiere cargar *todos* los datos en memoria.

8.2.4. Trabajar con las claves de los datos

Al utilizar los elementos de datos devueltos por un proveedor de datos, con frecuencia necesita identificar cada elemento de datos con una clave única. Por ejemplo, si los elementos de datos representan información de los clientes, puede querer usar el ID de cliente como la clave de cada conjunto de datos de un cliente. Los proveedores de datos pueden devolver una lista de estas claves correspondientes a los elementos de datos devueltos por `yii\data\DataProviderInterface::getModels()`. Por ejemplo:

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);
```

```
$provider = new ActiveDataProvider([
    'query' => $query,
]);

// Devuelve un array de objetos Post
$posts = $provider->getModels();

// Devuelve los valores de las claves primarias correspondientes a $posts
$ids = $provider->getKeys();
```

En el ejemplo superior, como se le proporciona a `yii\data\ActiveDataProvider` un objeto `yii\db\ActiveQuery`, es lo suficientemente inteligente como para devolver los valores de las claves primarias como las claves. También puede indicar explícitamente cómo se deben calcular los valores de la clave configurando `yii\data\ActiveDataProvider::$key` con un nombre de columna o un invocable que calcule los valores de la clave. Por ejemplo:

```
// Utiliza la columna «slug» como valores de la clave
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => 'slug',
]);

// Utiliza el resultado de md5(id) como valores de la clave
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => function ($model) {
        return md5($model->id);
    }
]);
```

8.2.5. Creación de un proveedor de datos personalizado

Para crear su propio proveedor de datos personalizado, debe implementar `yii\data\DataProviderInterface`. Una manera más fácil es extender `yii\data\BaseDataProvider`, que le permite centrarse en la lógica central del proveedor de datos. En particular, esencialmente necesita implementar los siguientes métodos:

- `prepareModels()`: prepara los modelos de datos que estarán disponibles en la página actual y los devuelve como un *array*.
- `prepareKeys()`: acepta un *array* de modelos de datos disponibles actualmente y devuelve las claves asociadas a ellos.
- `prepareTotalCount`: devuelve un valor que indica el número total de modelos de datos en el proveedor de datos.

Debajo se muestra un ejemplo de un proveedor de datos que lee datos CSV eficientemente:

```
<?php
use yii\data\BaseDataProvider;
```

```php
class CsvDataProvider extends BaseDataProvider
{
    /**
     * @var string nombre del fichero CSV a leer
     */
    public $filename;

    /**
     * @var string|callable nombre de la columna clave o un invocable que la
devuelva
     */
    public $key;

    /**
     * @var SplFileObject
     */
    protected $fileObject;  // SplFileObject es muy práctico para buscar una
    línea concreta en un fichero


    /**
     * {@inheritdoc}
     */
    public function init()
    {
        parent::init();

        // Abrir el fichero
        $this->fileObject = new SplFileObject($this->filename);
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareModels()
    {
        $models = [];
        $pagination = $this->getPagination();

        if ($pagination === false) {
            // En caso de que no haya paginación, leer todas las líneas
            while (!$this->fileObject->eof()) {
                $models[] = $this->fileObject->fgetcsv();
                $this->fileObject->next();
            }
        } else {
            // En caso de que haya paginación, leer sólo una única página
            $pagination->totalCount = $this->getTotalCount();
            $this->fileObject->seek($pagination->getOffset());
            $limit = $pagination->getLimit();

            for ($count = 0; $count < $limit; ++$count) {
                $models[] = $this->fileObject->fgetcsv();
```

```
                $this->fileObject->next();
            }
        }

        return $models;
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareKeys($models)
    {
        if ($this->key !== null) {
            $keys = [];

            foreach ($models as $model) {
                if (is_string($this->key)) {
                    $keys[] = $model[$this->key];
                } else {
                    $keys[] = call_user_func($this->key, $model);
                }
            }

            return $keys;
        }

        return array_keys($models);
    }

    /**
     * {@inheritdoc}
     */
    protected function prepareTotalCount()
    {
        $count = 0;

        while (!$this->fileObject->eof()) {
            $this->fileObject->next();
            ++$count;
        }

        return $count;
    }
}
```

8.2.6. Filtrar proveedores de datos usando filtros de datos

Si bien puede construir condiciones para un proveedor de datos activo manualmente tal y como se describe en las secciones Filtering Data y Separate Filter Form de la guía de *widgets* de datos, Yii tiene filtros de datos que son muy útiles si necesita condiciones de filtro flexibles. Los filtros de datos se pueden usar así:

```php
$filter = new ActiveDataFilter([
    'searchModel' => 'app\models\PostSearch'
]);

$filterCondition = null;

// Puede cargar los filtros de datos de cualquier fuente.
// Por ejemplo, si prefiere JSON en el cuerpo de la petición,
// use Yii::$app->request->getBodyParams() aquí abajo:
if ($filter->load(\Yii::$app->request->get())) {
    $filterCondition = $filter->build();
    if ($filterCondition === false) {
        // Serializer recibiría errores
        return $filter;
    }
}

$query = Post::find();
if ($filterCondition !== null) {
    $query->andWhere($filterCondition);
}

return new ActiveDataProvider([
    'query' => $query,
]);
```

El propósito del modelo `PostSearch` es definir por qué propiedades y valores se permite filtrar:

```php
use yii\base\Model;

class PostSearch extends Model
{
    public $id;
    public $title;

    public function rules()
    {
        return [
            ['id', 'integer'],
            ['title', 'string', 'min' => 2, 'max' => 200],
        ];
    }
}
```

Los filtros de datos son bastante flexibles. Puede personalizar cómo se construyen las condiciones y qué operadores se permiten. Para más detalles consulte la documentación de la API en `yii\data\DataFilter`.

8.3. Widgets de datos

Yii proporciona un conjunto de widgets que se pueden usar para mostrar datos. Mientras que el *widget* DetailView se puede usar para mostrar los datos de un único registro, ListView y GridView se pueden usar para mostrar una lista o tabla de registros de datos proporcionando funcionalidades como paginación, ordenación y filtro.

8.3.1. DetailView

El *widget* `DetailView` muestra los detalles de un único `modelo` de datos.

Se recomienda su uso para mostrar un modelo en un formato estándar (por ejemplo, cada atributo del modelo se muestra como una fila en una tabla). El modelo puede ser tanto una instancia o subclase de `yii\base\Model` como un active record o un *array* asociativo.

DetailView usa la propiedad `$attributes` para determinar qué atributos del modelo se deben mostrar y cómo se deben formatear. En la sección sobre formateadores se pueden ver las opciones de formato disponibles.

Un uso típico de DetailView sería así:

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        'title',                                          // atributo title
        (en texto plano)
        'description:html',                               // atributo
        description formateado como HTML
        [                                                 // nombre del
        propietario del modelo
            'label' => 'Owner',
            'value' => $model->owner->name,
            'contentOptions' => ['class' => 'bg-red'],    // atributos HTML
            para personalizar el valor
            'captionOptions' => ['tooltip' => 'Tooltip'], // atributos HTML
            para personalizar la etiqueta
        ],
        'created_at:datetime',                            // fecha de
        creación formateada como datetime
    ],
]);
```

Recuerde que a diferencia de `yii\widgets\GridView`, que procesa un conjunto de modelos, `DetailView` sólo procesa uno. Así que la mayoría de las veces no hay necesidad de usar funciones anónimas ya que `$model` es el único modelo a mostrar y está disponible en la vista como una variable.

Sin embargo, en algunos casos el uso de una función anónima puede ser útil. Por ejemplo cuando `visible` está especificado y se desea impedir el cálculo de `value` en case de que evalúe a `false`:

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        [
            'attribute' => 'owner',
            'value' => function ($model) {
                return $model->owner->name;
            },
            'visible' => \Yii::$app->user->can('posts.owner.view'),
        ],
    ],
]);
```

8.3.2. ListView

El *widget* `ListView` se usa para mostrar datos de un proveedor de datos. Cada modelo de datos se representa usando el `fichero de vista` indicado. Como proporciona de serie funcionalidades tales como paginación, ordenación y filtro, es útil tanto para mostrar información al usuario final como para crear una interfaz de usuario de gestión de datos.

Un uso típico es el siguiente:

```
use yii\widgets\ListView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);

echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
]);
```

El fichero de vista `_post` podría contener lo siguiente:

```
<?php
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
?>
<div class="tarea">
    <h2><?= Html::encode($model->title) ?></h2>

    <?= HtmlPurifier::process($model->text) ?>
</div>
```

En el fichero de vista anterior, el modelo de datos actual está disponible como `$model`. Además están disponibles las siguientes variables:

- `$key`: mixto, el valor de la clave asociada a este elemento de datos.
- `$index`: entero, el índice empezando por cero del elemento de datos en el array de elementos devuelto por el proveedor de datos.
- `$widget`: ListView, esta instancia del *widget*.

Si se necesita pasar datos adicionales a cada vista, se puede usar la propiedad `$viewParams` para pasar parejas clave-valor como las siguientes:

```
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
    'viewParams' => [
        'fullView' => true,
        'context' => 'main-page',
        // ...
    ],
]);
```

Entonces éstas también estarán disponibles en la vista como variables.

8.3.3. GridView

La cuadrícula de datos o `GridView` es uno de los *widgets* de Yii más potentes. Es extremadamente útil si necesita construir rápidamente la sección de administración del sistema. Recibe los datos de un proveedor de datos y representa cada fila usando un conjunto de `columnas` que presentan los datos en forma de tabla.

Cada fila de la tabla representa los datos de un único elemento de datos, y una columna normalmente representa un atributo del elemento (algunas columnas pueden corresponder a expresiones complejas de los atributos o a un texto estático).

El mínimo código necesario para usar GridView es como sigue:

```
use yii\grid\GridView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

El código anterior primero crea un proveedor de datos y a continuación usa GridView para mostrar cada atributo en cada fila tomados del proveedor de datos. La tabla mostrada está equipada de serie con las funcionalidades de ordenación y paginación.

Columnas de la cuadrícula

Las columnas de la tabla se configuran en términos de clase `yii\grid\Column`, que se configuran en la propiedad `columns` de la configuración del GridView. Dependiendo del tipo y ajustes de las columnas éstas pueden presentar los datos de diferentes maneras. La clase predefinida es `yii\grid\DataColumn`, que representa un atributo del modelo por el que se puede ordenar y filtrar.

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        // Columnas sencillas definidas por los datos contenidos en
        $dataProvider.
        // Se usarán los datos de la columna del modelo.
        'id',
        'username',
        // Un ejemplo más complejo.
        [
            'class' => 'yii\grid\DataColumn',  // Se puede omitir, ya que es
            la predefinida.
            'value' => function ($data) {
                return $data->name;  // $data['name'] para datos de un
                array, por ejemplo al usar SqlDataProvider.
            },
        ],
    ],
]);
```

Observe que si no se especifica la parte `columns` de la configuración, Yii intenta mostrar todas las columnas posibles del modelo del proveedor de datos.

Clases de columna

Las columnas de la cuadrícula se pueden personalizar usando diferentes clases de columna:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\SerialColumn', // <-- aquí
            // puede configurar propiedades adicionales aquí
        ],
```

Además de las clases de columna proporcionadas por Yii que se revisarán más abajo, puede crear sus propias clases de columna.

Cada clase de columna extiende `yii\grid\Column` de modo que hay algunas opciones comunes que puede establecer al configurar las columnas de una cuadrícula.

- `header` permite establecer el contenida para la fila cabecera
- `footer` permite establece el contenido de la fila al pie
- `visible` define si la columna debería ser visible.
- `content` le permite pasar una función PHP válida que devuelva datos para una fila. El formato es el siguiente:

```
function ($model, $key, $index, $column) {
    return 'una cadena';
}
```

Puede indicar varias opciones HTML del contenedor pasando *arrays* a:

- `headerOptions`
- `footerOptions`
- `filterOptions`
- `contentOptions`

8.4. Trabajar con Scripts del Cliente

Nota: Esta sección se encuentra en desarrollo.

Registrar scripts

Con el objeto `yii\web\View` puedes registrar scripts. Hay dos métodos dedicados a esto: `registerJs()` para scripts en línea `registerJsFile()` para scripts externos. Los scripts en línea son útiles para configuración y código generado dinámicamente. El método para agregarlos puede ser utilizado así:

```
$this->registerJs("var options = ".json_encode($options).";", View::POS_END,
'my-options');
```

El primer argumento es el código JS real que queremos insertar en la página. El segundo argumento determina en qué parte de la página debería ser insertado el script. Los valores posibles son:

- `View::POS_HEAD` para la sección head.
- `View::POS_BEGIN` justo después de la etiqueta `<body>`.
- `View::POS_END` justo antes de cerrar la etiqueta `</body>`.
- `View::POS_READY` para ejecutar código en el evento `ready` del documento. Esto registrará `jQuery` automáticamente.
- `View::POS_LOAD` para ejecutar código en el evento `load` del documento. Esto registrará `jQuery` automáticamente.

El último argumento es un ID único del script, utilizado para identificar el bloque de código y reemplazar otro con el mismo ID en vez de agregar uno nuevo. En caso de no proveerlo, el código JS en sí será utilizado como ID.

Un script externo puede ser agregado de esta manera:

```
$this->registerJsFile('https://example.com/js/main.js', ['depends' =>
[\yii\web\JqueryAsset::class]]);
```

Los argumentos para `registerJsFile()` son similares a los de `registerCssFile()`. En el ejemplo anterior, registramos el archivo `main.js` con dependencia de `JqueryAsset`. Esto quiere decir que el archivo `main.js` será agregado DESPUÉS de `jquery.js`. Si esta especificación de dependencia, el orden relativo entre `main.js` y `jquery.js` sería indefinido.

Como para `registerCssFile()`, es altamente recomendable que utilices asset bundles para registrar archivos JS externos más que utilizar `registerJsFile()`.

Registrar asset bundles

Como mencionamos anteriormente, es preferible utilizar asset bundles en vez de usar CSS y JavaScript directamente. Puedes obtener detalles de cómo definir asset bundles en la sección gestor de assets de esta guía. Utilizar asset bundles ya definidos es muy sencillo:

```
\frontend\assets\AppAsset::register($this);
```

Registrar CSS

Puedes registrar CSS utilizando `registerCss()` o `registerCssFile()`. El primero registra un bloque de código CSS mientras que el segundo registra un archivo CSS externo. Por ejemplo,

```
$this->registerCss("body { background: #f00; }");
```

El código anterior dará como resultado que se agregue lo siguiente a la sección head de la página:

```
<style>
body { background: #f00; }
</style>
```

Si quieres especificar propiedades adicionales a la etiqueta style, pasa un array de claves-valores como tercer argumento. Si necesitas asegurarte que haya sólo una etiqueta style utiliza el cuarto argumento como fue mencionado en las descripciones de meta etiquetas.

```
$this->registerCssFile("https://example.com/css/themes/black-and-white.css",
[
    'depends' => [BootstrapAsset::class],
    'media' => 'print',
], 'css-print-theme');
```

El código de arriba agregará un link al archivo CSS en la sección head de la página.

- El primer argumento especifica el archivo CSS a ser registrado.

- El segundo argumento especifica los atributos HTML de la etiqueta `<link>` resultante. La opción `depends` es especialmente tratada. Esta especifica de qué asset bundles depende este archivo CSS. En este caso, depende del asset bundle `yii\bootstrap\BootstrapAsset`. Esto significa que el archivo CSS será agregado *después* de los archivos CSS de `yii\bootstrap\BootstrapAsset`.
- El último argumento especifica un ID que identifica al archivo CSS. Si no es provisto, se utilizará la URL del archivo.

Es altamente recomendable que ustilices asset bundles para registrar archivos CSS en vez de utilizar `registerCssFile()`. Utilizar asset bundles te permite combinar y comprimir varios archivos CSS, deseable en sitios web de tráfico alto.

8.5. Temas

> **Nota:** Esta sección está en desarrollo.

Un tema (theme) es un directorio de archivos y de vistas (views) y layouts. Cada archivo de este directorio sobrescribe el archivo correspondiente de una aplicación cuando se renderiza. Una única aplicación puede usar múltiples temas para que pueden proporcionar experiencias totalmente diferentes. Solo se puede haber un único tema activo.

> **Nota:** Los temas no están destinados a ser redistribuidos ya que están demasiado ligados a la aplicación. Si se quiere redistribuir una apariencia personalizada, se puede considerar la opción de asset bundles de archivos CSS y Javascript.

8.5.1. Configuración de un Tema

La configuración de un tema se especifica a través del componente `view` de la aplicación. Para establecer que un tema trabaje con vistas de aplicación básicas, la configuración de la aplicación debe contener lo siguiente:

```
'components' => [
    'view' => [
        'theme' => [
            'pathMap' => ['@app/views' => '@app/themes/basic'],
            'baseUrl' => '@web/themes/basic',
        ],
    ],
],
```

En el ejemplo anterior, el `pathMap` define un mapa (map) de las rutas a las que se aplicará el tema mientras que `baseUrl` define la URL base para los recursos a los que hacen referencia los archivos del tema.

En nuestro caso `pathMap` es `['@app/views' => '@app/themes/basic']`. Esto significa que cada vista de `@app/views` primero se buscará en `@app/themes/basic` y si existe, se usará la vista del directorio del tema en lugar de la vista original.

Por ejemplo, con la configuración anterior, la versión del tema para la vista `@app/views/site/index.php` será `@app/themes/basic/site/index.php`. Básicamente se reemplaza `@app/views` en `@app/views/site/index.php` por `@app/themes/basic`.

Temas para Módulos

Para utilizar temas en los módulos, el `pathMap` debe ser similar al siguiente:

```
'components' => [
    'view' => [
        'theme' => [
            'pathMap' => [
                '@app/views' => '@app/themes/basic',
                '@app/modules' => '@app/themes/basic/modules', // <-- !!!
            ],
        ],
    ],
],
```

Esto permite aplicar el tema a `@app/modules/blog/views/comment/index.php` con la vista `@app/themes/basic/modules/blog/views/comment/index.php`.

Temas para Widgets

Para utilizar un tema en una vista que se encuentre en `@app/widgets/currency/views/index.php`, se debe aplicar la siguiente configuración para el componente vista, tema:

```
'components' => [
    'view' => [
        'theme' => [
            'pathMap' => ['@app/widgets' => '@app/themes/basic/widgets'],
        ],
    ],
],
```

Con la configuración anterior, se puede crear una versión de la vista `@app/widgets/currency/index.php` para que se aplique el tema en `@app/themes/basic/widgets/currency/views/index.php`.

8.5.2. Uso de Multiples Rutas

Es posible mapear una única ruta a múltiples rutas de temas. Por ejemplo:

```
'pathMap' => [
    '@app/views' => [
```

```
        '@app/themes/christmas',
        '@app/themes/basic',
    ],
]
```

En este caso, primero se buscara la vista en `@app/themes/christmas/site/index.php`, si no se encuentra, se intentará en `@app/themes/basic/site/index.php`. Si la vista no se encuentra en ninguna de rutas especificadas, se usará la vista de aplicación.

Esta capacidad es especialmente útil si se quieren sobrescribir algunas rutas temporal o condicionalmente.

Capítulo 9

Seguridad

9.1. Authentication

La autenticación es el proceso de verificar la identidad de un usuario. Usualmente se usa un identificador (ej. un `username` o una dirección de correo electrónico) y una token secreto (ej. una contraseña o un token de acceso) para juzgar si el usuario es quien dice ser. La autenticación es la base de la función de inicio de sesión.

Yii proporciona un marco de autenticación que conecta varios componentes para soportar el inicio de sesión. Para utilizar este marco, usted necesita principalmente hacer el siguiente trabajo:

- Configurar el componente de la aplicación `user`;
- Crear una clase que implemente la interfaz `yii\web\IdentityInterface`.

9.1.1. Configurando `yii\web\User`

El componente `user` gestiona el estado de autenticación del usuario. Requiere que especifiques una `clase de identidad` la cual contiene la lógica de autenticación. En la siguiente configuración de la aplicación, la `clase identity` para `user` está configurada para ser `app\models\User` cuya implementación se explica en la siguiente subsección:

```
return [
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
        ],
    ],
];
```

9.1.2. Implementando `yii\web\IdentityInterface`

La `clase identity` debe implementar la `yii\web\IdentityInterface` que contiene los siguientes métodos:

- `findIdentity()`: busca una instancia de la clase identidad usando el ID de usuario especificado. Este método se utiliza cuando se necesita mantener el estado de inicio de sesión (login) a través de la sesión.
- `findIdentityByAccessToken()`: busca una instancia de la clase de identidad usando el token de acceso especificado. Este método se utiliza cuando se necesita autenticar a un usuario mediante un único token secreto (ej. en una aplicación RESTful sin estado).
- `getId()`: devuelve el ID del usuario representado por esta instancia de identidad.
- `getAuthKey()`: devuelve una clave utilizada para validar la sesión y el auto-login en caso de que esté habilitado.
- `validateAuthKey()`: implementa la lógica para verificar la clave de autenticación.

Si no se necesita un método en particular, se podría implementar con un cuerpo vacío, Por ejemplo, Si un método en particular no es necesario, puedes implementarlo con un cuerpo vacío. Por ejemplo, si tu aplicación es una aplicación RESTful pura sin estado, sólo necesitarás implementar `findIdentityByAccessToken()` y `getId()` dejando el resto de métodos con un cuerpo vacío. O si tu aplicación utiliza autenticación sólo de sesión, necesitarías implementar todos los métodos excepto findIdentityByAccessToken().

En el siguiente ejemplo, una clase `identity` es implementada como una clase Active Record asociada con la tabla de base de datos `user`.

```
<?php

use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function tableName()
    {
        return 'user';
    }

    /**
     * Buscar una identidad por el ID dado.
     *
     * @param string|int $id ID que debe buscarse
     * @return IdentityInterface|null objeto de identidad que coincide con
el ID dado.
     */
    public static function findIdentity($id)
    {
        return static::findOne($id);
    }

    /**
     * Buscar una identidad por el token dado..
```

```
     *
     * @param string $token token que debe buscarse
     * @return IdentityInterface|null objeto de identidad que coincide con
el token dado.
     */
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }

    /**
     * @return int|string ID del usuario actual
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @return string|null llave de autenticación del usuario actual
     */
    public function getAuthKey()
    {
        return $this->auth_key;
    }

    /**
     * @param string $authKey
     * @return bool|null si la llave de autenticación es válida para el
usuario actual
     */
    public function validateAuthKey($authKey)
    {
        return $this->getAuthKey() === $authKey;
    }
}
```

Puede utilizar el siguiente código para generar una clave de autenticación para cada usuario y almacenarla en la tabla `user`:

```
class User extends ActiveRecord implements IdentityInterface
{
    ......

    public function beforeSave($insert)
    {
        if (parent::beforeSave($insert)) {
            if ($this->isNewRecord) {
                $this->auth_key =
                \Yii::$app->security->generateRandomString();
            }
            return true;
        }
        return false;
```

```
    }
}
```

> Nota: No confundas la clase de identidad `User` con `yii\web\User`. La primera es la clase que implementa la lógica de autenticación. Suele implementarse como una clase Active Record asociada a algún almacenamiento persistente para guardar la información de las credenciales del usuario. Esta última es una clase de componente de aplicación responsable de gestionar el estado de autenticación del usuario.

9.1.3. Usando `yii\web\User`

Principalmente se usa `yii\web\User` en términos del componente de aplicación `user`.

Puede detectar la identidad del usuario actual usando la expresión `Yii::$app->user->identity`. Devuelve una instancia de la clase `identity` que representa al usuario actualmente conectado, o `null` si el usuario actual no está autenticado (es decir, es un invitado). El siguiente código muestra como recuperar otra información relacionada con la autenticación desde `yii\web\User`:

```
// El usuario actual identificado. `null` si el usuario no esta
autenticado.
$identity = Yii::$app->user->identity;

// El ID del usuario actual. `null` si el usuario no esta autenticado.
$id = Yii::$app->user->id;

// si el usuario actual es un invitado (No autenticado)
$isGuest = Yii::$app->user->isGuest;
```

Para acceder a un usuario, puede utilizar el siguiente código:

```
// encontrar una identidad de usuario con el nombre de usuario
especificado.
// tenga en cuenta que es posible que desee comprobar la contraseña si es
necesario
$identity = User::findOne(['username' => $username]);

// inicia la sesión del usuario.
Yii::$app->user->login($identity);
```

El método `yii\web\User::login()` establece la identidad del usuario actual a `yii\web\User`. Si la sesión es `habilitada`, mantendrá la identidad en la sesión para que el estado de autenticación del usuario se mantenga durante toda la sesión. Si el login basado en cookies (es decir, inicio de sesión "recordarme") está `habilitado`, también guardará la identidad en una cookie para que el estado de autenticación del usuario pueda ser recuperado desde la cookie mientras la cookie permanezca válida.

Para habilitar el login basado en cookies, necesita configurar `yii\web\User::$enableAutoLogin` como `true` en la configuración de la aplicación. También necesita proporcionar un parámetro de tiempo de duración cuando llame al método `yii\web\User::login()`.

Para cerrar la sesión de un usuario, basta con llamar a:

```
Yii::$app->user->logout();
```

Tenga en cuenta que cerrar la sesión de un usuario sólo tiene sentido cuando la sesión está activada. El método limpiará el estado de autenticación del usuario tanto de la memoria como de la sesión. Y por defecto, también destruirá *todos* los datos de sesión del usuario. Si desea mantener los datos de sesión, debe llamar a `Yii::$app->user->logout(false)`, en su lugar.

9.1.4. Eventos de Autenticación

La clase `yii\web\User` genera algunos eventos durante los procesos de inicio y cierre de sesión.

- `EVENT_BEFORE_LOGIN`: levantado al comienzo de `yii\web\User::login()`. Si el manejador del evento establece la propiedad `isValid` del objeto evento a `false`, el proceso de inicio de sesión será cancelado.
- `EVENT_AFTER_LOGIN`: se produce después de un inicio de sesión exitoso.
- `EVENT_BEFORE_LOGOUT`: levantado al comienzo de `yii\web\User::logout()`. Si el manejador del evento establece la propiedad `isValid` del objeto evento a `false`, el proceso de cierre de sesión será cancelado.
- `EVENT_AFTER_LOGOUT`: se produce después de un cierre de sesión exitoso. Usted puede responder a estos eventos para implementar características como auditoria de inicio de sesión, estadísticas de usuarios en línea. Por ejemplo, en el manejador para `EVENT_AFTER_LOGIN`, puede registrar la hora de inicio de sesión y la dirección IP en la tabla `user`.

9.2. Autorización

Autorización esl el proceso de verificación de que un usuario tenga sugifientes permisos para realizar algo. Yii provee dos métodos de autorización: Filtro de Control de Acceso y Control Basado en Roles (ACF y RBAC por sus siglas en inglés).

9.2.1. Filtro de Control de Acceso

Filtro de Control de Acceso (ACF) es un único método de autorización implementado como `yii\filters\AccessControl`, el cual es mejor utilizado por aplicaciones que sólo requieran un control de acceso simple. Como su nombre lo indica, ACF es un filtro de acción que puede ser utilizado en un controlador o en un módulo. Cuando un usuario solicita la ejecución de una

acción, ACF comprobará una lista de `reglas de acceso` para determinar si el usuario tiene permitido acceder a dicha acción.

El siguiente código muestra cómo utilizar ACF en el controlador `site`:

```
use yii\web\Controller;
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::class,
                'only' => ['login', 'logout', 'signup'],
                'rules' => [
                    [
                        'allow' => true,
                        'actions' => ['login', 'signup'],
                        'roles' => ['?'],
                    ],
                    [
                        'allow' => true,
                        'actions' => ['logout'],
                        'roles' => ['@'],
                    ],
                ],
            ],
        ];
    }
    // ...
}
```

En el código anterior, ACF es adjuntado al controlador `site` en forma de behavior (comportamiento). Esta es la forma típica de utilizar un filtro de acción. La opción `only` especifica que el ACF debe ser aplicado solamente a las acciones `login`, `logout` y `signup`. Las acciones restantes en el controlador `site` no están sujetas al control de acceso. La opción `rules` lista las `reglas de acceso`, y se lee como a continuación:

- Permite a todos los usuarios invitados (sin autenticar) acceder a las acciones `login` y `signup`. La opción `roles` contiene el signo de interrogación `?`, que es un código especial para representar a los "invitados".
- Permite a los usuarios autenticados acceder a la acción `logout`. El signo `@` es otro código especial que representa a los "usuarios autenticados".

ACF ejecuta la comprobación de autorización examinando las reglas de acceso una a una desde arriba hacia abajo hasta que encuentra una regla que aplique al contexto de ejecución actual. El valor `allow` de la regla que coincida será entonces utilizado para juzgar si el usuario está autorizado o no. Si ninguna de las reglas coincide, significa que el usuario NO está autorizado, y el ACF detendrá la ejecución de la acción.

Cuando el ACF determina que un usuario no está autorizado a acceder a la acción actual, toma las siguientes medidas por defecto:

- Si el usuario es un invitado, llamará a `yii\web\User::loginRequired()` para redireccionar el navegador a la pantalla de login.
- Si el usuario está autenticado, lanzará una excepción `yii\web\ForbiddenHttpException`.

Puedes personalizar este comportamiento configurando la propiedad `yii\filters\AccessControl::$denyCallback` como a continuación:

```
[
    'class' => AccessControl::class,
    ...
    'denyCallback' => function ($rule, $action) {
        throw new \Exception('No tienes los suficientes permisos para
        acceder a esta página');
    }
]
```

Las `Reglas de Acceso` soportan varias opciones. Abajo hay un resumen de las mismas. También puedes extender de `yii\filters\AccessRule` para crear tus propias clases de reglas de acceso personalizadas.

- `allow`: especifica si la regla es de tipo "allow" (permitir) o "deny" (denegar).
- `actions`: especifica con qué acciones coinciden con esta regla. Esta debería ser un array de IDs de acciones. La comparación es sensible a mayúsculas. Si la opción está vacía o no definida, significa que la regla se aplica a todas las acciones.
- `controllers`: especifica con qué controladores coincide esta regla. Esta debería ser un array de IDs de controladores. Cada ID de controlador es prefijado con el ID del módulo (si existe). La comparación es sensible a mayúsculas. Si la opción está vacía o no definida, significa que la regla se aplica a todos los controladores.
- `roles`: especifica con qué roles de usuarios coincide esta regla. Son reconocidos dos roles especiales, y son comprobados vía `yii\web\User::$isGuest`:
  - `?`: coincide con el usuario invitado (sin autenticar)
  - `@`: coincide con el usuario autenticado

  El utilizar otro nombre de rol invocará una llamada a `yii\web\User::can()`, que requiere habilitar RBAC (a ser descrito en la próxima subsección). Si la opción está vacía o no definida, significa que la regla se aplica a todos los roles.
- `ips`: especifica con qué `dirección IP del cliente` coincide esta regla. Una dirección IP puede contener el caracter especial `*` al final de manera que coincidan todas las IPs que comiencen igual. Por ejemplo, '192.168.*' coincide con las direcciones IP en el segmento '192.168.'. Si la opción está vacía o no definida, significa que la regla se aplica a todas las direcciones IP.

- `verbs`: especifica con qué método de la solicitud (por ej. `GET`, `POST`) coincide esta regla. La comparación no distingue minúsculas de mayúsculas.
- `matchCallback`: especifica una función PHP invocable que debe ser llamada para determinar si la regla debe ser aplicada.
- `denyCallback`: especifica una función PHP invocable que debe ser llamada cuando esta regla deniegue el acceso.

Debajo hay un ejemplo que muestra cómo utilizar la opción `matchCallback`, que te permite escribir lógica de comprabación de acceso arbitraria:

```
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::class,
                'only' => ['special-callback'],
                'rules' => [
                    [
                        'actions' => ['special-callback'],
                        'allow' => true,
                        'matchCallback' => function ($rule, $action) {
                            return date('d-m') === '31-10';
                        }
                    ],
                ],
            ],
        ];
    }

    // Callback coincidente llamado! Esta página sólo puede ser accedida
    cada 31 de Octubre
    public function actionSpecialCallback()
    {
        return $this->render('happy-halloween');
    }
}
```

9.2.2. Control de Acceso Basado en Roles (RBAC)

El Control de Acceso Basado en Roles (RBAC) provee una simple pero poderosa manera centralizada de control de acceso. Por favos consulta la Wikipedia[1] para más detalles sobre comparar RBAC con otros mecanismos de control de acceso más tradicionales.

Yii implementa una Jerarquía General RBAC, siguiendo el modelo NIST

[1] https://en.wikipedia.org/wiki/Role-based_access_control

RBAC[2]. Esto provee la funcionalidad RBAC a través de componente de la aplicación `authManager`.

Utilizar RBAC envuelve dos cosas. La primera es construir los datos de autorización RBAC, y la segunda es utilizar esos datos de autorización para comprobar el acceso en los lugares donde se necesite.

Para facilitar la próxima descripción, necesitamos primero instroducir algunos conceptos RBAC básicos.

Conceptos Básicos

Un rol representa una colección de *permisos* (por ej. crear posts, actualizar posts). Un rol puede ser asignado a uno o varios usuarios. Para comprobar que un usuario cuenta con determinado permiso, podemos comprobar si el usuario tiene asignado un rol que cuente con dicho permiso.

Asociado a cada rol o permiso, puede puede haber una *regla*. Una regla representa una porción de código que será ejecutada durante la comprobación de acceso para determinar si el rol o permiso correspondiente aplica al usuario actual. Por ejemplo, el permiso "actualizar post" puede tener una regla que compruebe que el usuario actual es el autor del post. Durante la comprobación de acceso, si el usuario NO es el autor del post, se considerará que el/ella no cuenta con el permiso "actualizar post".

Tanto los roles como los permisos pueden ser organizados en una jerarquía. En particular, un rol puede consistir en otros roles o permisos; y un permiso puede consistir en otros permisos. Yii implementa una jerarquía de *orden parcial*, que incluye una jerarquía de *árbol* especial. Mientras que un rol puede contener un permiso, esto no sucede al revés.

Configurar RBAC

Antes de definir todos los datos de autorización y ejecutar la comprobación de acceso, necesitamos configurar el componente de la aplicación `authManager`. Yii provee dos tipos de administradores de autorización: `yii\rbac\PhpManager` y `yii\rbac\DbManager`. El primero utiliza un archivo PHP para almacenar los datos de autorización, mientras que el segundo almacena dichos datos en una base de datos. Puedes considerar utilizar el primero si tu aplicación no requiere una administración de permisos y roles muy dinámica.

Utilizar El siguiente código muestra cómo configurar `authManager` en la configuración de nuestra aplicación utilizando la clase `yii\rbac\PhpManager`:

```
return [
    // ...
```

[2] https://csrc.nist.gov/CSRC/media/Publications/conference-paper/1992/10/13/role-based-access-controls/documents/ferraiolo-kuhn-92.pdf

```
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
        // ...
    ],
];
```

El `authManager` ahora puede ser accedido vía `\Yii::$app->authManager`.

Por defecto, `yii\rbac\PhpManager` almacena datos RBAC en archivos bajo el directorio `@app/rbac`. Asegúrate de que el directorio y todos sus archivos son tienen permiso de escritura para el proceso del servidor Web si la jerarquía de permisos necesita ser modoficada en línea.

Utilizar El sigiente código muestra cómo configurar `authManager` en la configuración de la aplicación utilizando la clase `yii\rbac\DbManager`:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\DbManager',
        ],
        // ...
    ],
];
```

> **Nota:** si estás utilizando el template yii2-basic-app, existe el archivo de configuración `config/console.php` donde necesita declararse `authManager` adicionalmente a `config/web.php`. En el caso de yii2-advanced-app, `authManager` sólo debe declararse en `common/config/main.php`.

`DbManager` utiliza cuatro tablas de la BD para almacenar los datos:

- `itemTable`: la tabla para almacenar los ítems de autorización. Por defecto "auth_item".
- `itemChildTable`: la tabla para almacentar la jerarquía de los ítems de autorización. Por defecto "auth_item_child".
- `assignmentTable`: la tabla para almacenar las asignaciones de los ítems de autorización. Por defecto "auth_assignment".
- `ruleTable`: la tabla para almacenar las reglas. Por defecto "auth_rule".

Antes de continuar, necesitas crear las tablas respectivas en la base de datos. Para hacerlo, puedes utilizar las migraciones contenidas en `@yii/rbac/migrations`:

```
yii migrate --migrationPath=@yii/rbac/migrations
```

El `authManager` puede ahora ser accedido vía `\Yii::$app->authManager`.

Construir los Datos de Autorización

Construir los datos de autorización implica las siguientes tareas:

- definir roles y permisos;
- establecer relaciones entre roles y permisos;
- definir reglas;
- asociar reglas con roles y permisos;
- asignar roles a usuarios.

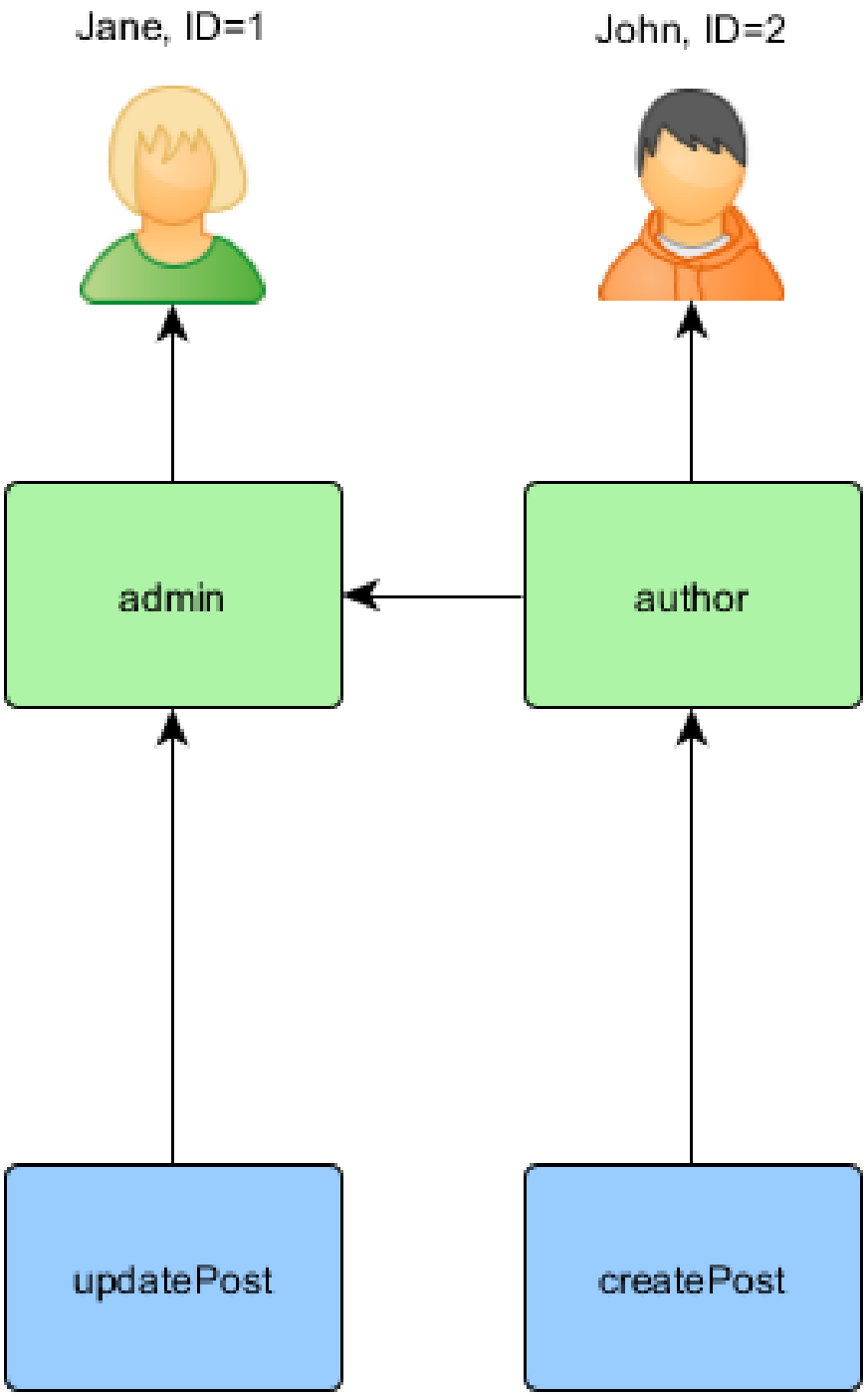
Dependiendo de los requerimientos de flexibilidad en la autorización, las tareas se pueden lograr de diferentes maneras.

Si la jerarquía de permisos no cambia en absoluto y tienes un número fijo de usuarios puede crear un comando de consola que va a inicializar los datos de autorización una vez a través de las API que ofrece por `authManager`:

```
<?php
namespace app\commands;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
{
    public function actionInit()
    {
        $auth = Yii::$app->authManager;

        // agrega el permiso "createPost"
        $createPost = $auth->createPermission('createPost');
        $createPost->description = 'Create a post';
        $auth->add($createPost);

        // agrega el permiso "updatePost"
        $updatePost = $auth->createPermission('updatePost');
        $updatePost->description = 'Update post';
        $auth->add($updatePost);

        // agrega el rol "author" y le asigna el permiso "createPost"
        $author = $auth->createRole('author');
        $auth->add($author);
        $auth->addChild($author, $createPost);

        // agrega el rol "admin" y le asigna el permiso "updatePost"
        // más los permisos del rol "author"
        $admin = $auth->createRole('admin');
        $auth->add($admin);
        $auth->addChild($admin, $updatePost);
        $auth->addChild($admin, $author);

        // asigna roles a usuarios. 1 y 2 son IDs devueltos por
        IdentityInterface::getId()
        // usualmente implementado en tu modelo User.
        $auth->assign($author, 2);
        $auth->assign($admin, 1);
    }
}
```

Nota: Si estas utilizando el template avanzado, necesitas poner tu `RbacController` dentro del directorio `console/controllers` y cambiar el espacio de nombres a `console\controllers`.

Después de ejecutar el comando `yii rbac/init`, obtendremos la siguiente jerarquía:



"Author" puede crear un post, "admin" puede actualizar posts y hacer todo lo que puede hacer "author".

Si tu aplicación permite el registro de usuarios, necesitas asignar los roles necesarios para cada usuario nuevo. Por ejemplo, para que todos los usuarios registrados tengan el rol "author", en el template de aplicación avanzada debes modificar `frontend\models\SignupForm::signup()` como a continuación:

```
public function signup()
{
    if ($this->validate()) {
        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
        $user->setPassword($this->password);
        $user->generateAuthKey();
        $user->save(false);

        // las siguientes tres líneas fueron agregadas
        $auth = Yii::$app->authManager;
        $authorRole = $auth->getRole('author');
        $auth->assign($authorRole, $user->getId());

        return $user;
    }

    return null;
}
```

Para aplicaciones que requieren un control de acceso complejo con una actualización constante en los datos de autorización, puede ser necesario desarrollar una interfaz especial (por ej. un panel de administración) utilizando las APIs ofrecidas por `authManager`.

Utilizar Reglas

Como se había mencionado, las reglas agregan restricciones adicionales a los roles y permisos. Una regla es una clase extendida de `yii\rbac\Rule`. Debe implementar al método `execute()`. En la jerarquía que creamos previamente, "author" no puede editar su propio post. Vamos a arreglarlo. Primero necesitamos una regla para comprobar que el usuario actual es el autor del post:

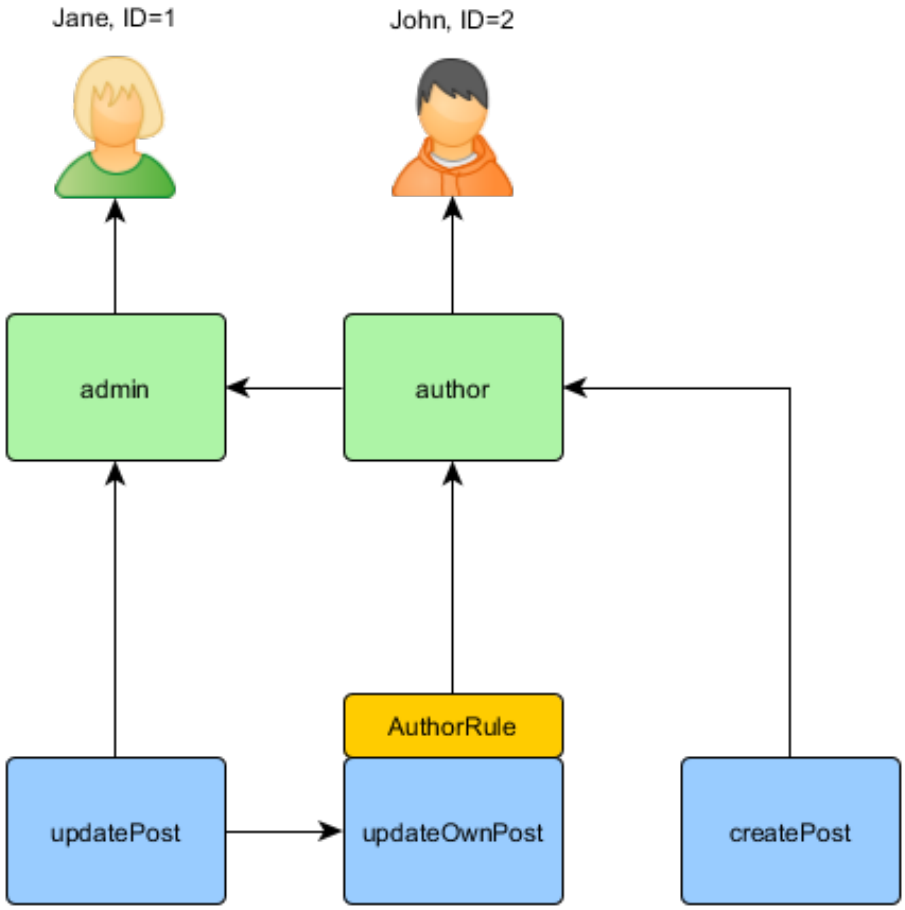
```
namespace app\rbac;

use yii\rbac\Rule;

/**
 * Comprueba si authorID coincide con el usuario pasado como parámetro
 */
class AuthorRule extends Rule
{
    public $name = 'isAuthor';
```

```
    /**
     * @param string|int $user el ID de usuario.
     * @param Item $item el rol o permiso asociado a la regla
     * @param array $params parámetros pasados a
ManagerInterface::checkAccess().
     * @return bool un valor indicando si la regla permite al rol o permiso
con el que está asociado.
     */
    public function execute($user, $item, $params)
    {
        return isset($params['post']) ? $params['post']->createdBy == $user
        : false;
    }
}
```

La regla anterior comprueba si el `post` fue creado por `$user`. Crearemos un permiso especial, `updateOwnPost`, en el comando que hemos utilizado anteriormente:

```
$auth = Yii::$app->authManager;

// agrega la regla
$rule = new \app\rbac\AuthorRule;
$auth->add($rule);

// agrega el permiso "updateOwnPost" y le asocia la regla.
$updateOwnPost = $auth->createPermission('updateOwnPost');
$updateOwnPost->description = 'Update own post';
$updateOwnPost->ruleName = $rule->name;
$auth->add($updateOwnPost);

// "updateOwnPost" será utilizado desde "updatePost"
$auth->addChild($updateOwnPost, $updatePost);

// permite a "author" editar sus propios posts
$auth->addChild($author, $updateOwnPost);
```

Ahora tenemos la siguiente jerarquía:

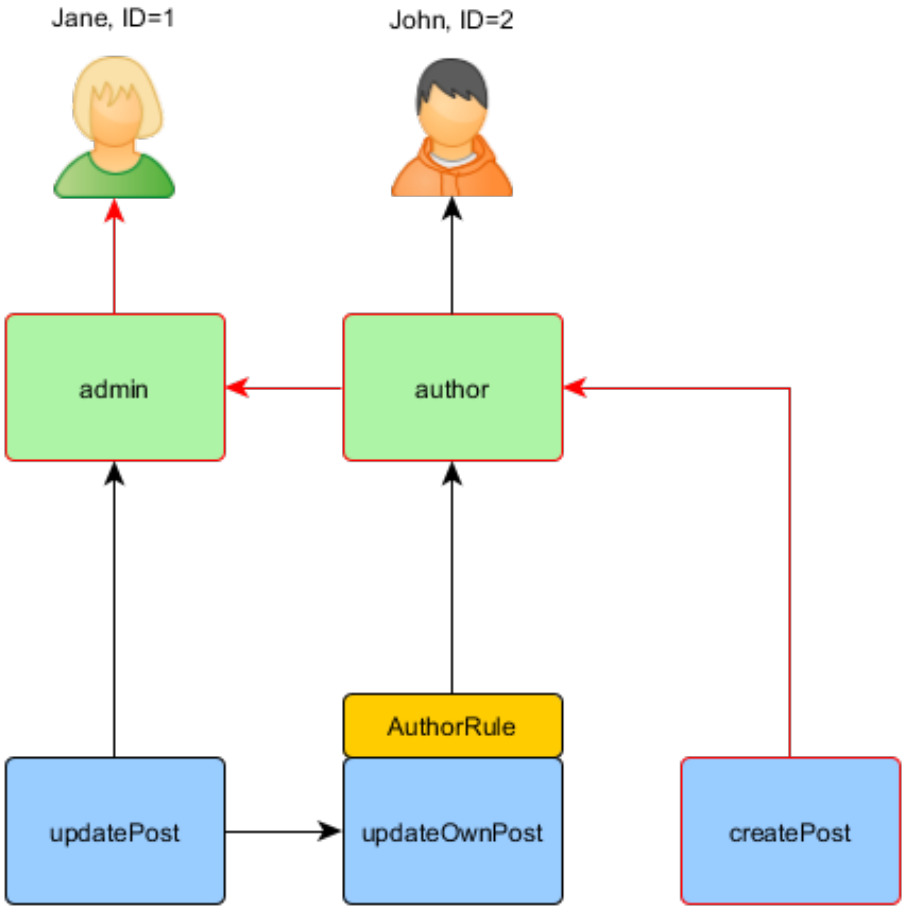


Comprobación de Acceso

Con los datos de autorización listos, la comprobación de acceso se hace con una simple llamada al método `yii\rbac\ManagerInterface::checkAccess()`. Dado que la mayoría de la comprobación de acceso se hace sobre el usuario actual, para mayor comodidad Yii proporciona el atajo `yii\web\User::can()`, que puede ser utilizado como a continuación:

```
if (\Yii::$app->user->can('createPost')) {
    // crear el post
}
```

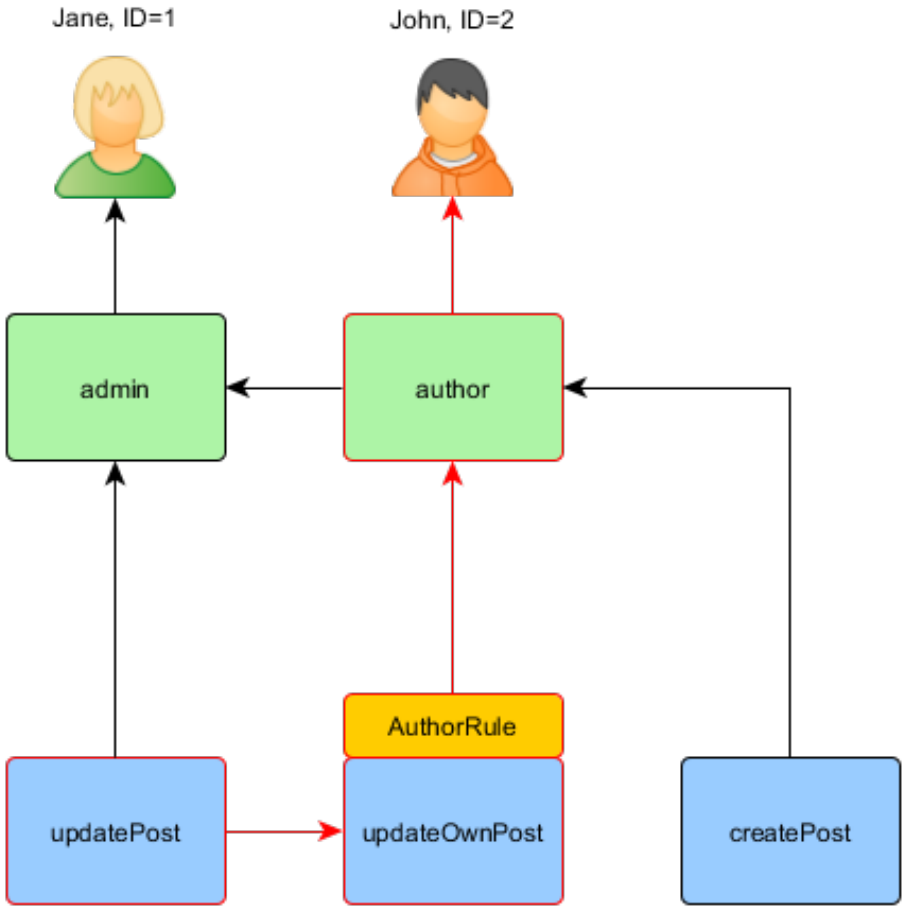
Si el usuario actual es Jane con `ID=1`, comenzamos desde `createPost` y tratamos de alcanzar a `Jane`:



Con el fin de comprobar si un usuario puede actualizar un post, necesitamos pasarle un parámetro adicional requerido por `AuthorRule`, descrito antes:

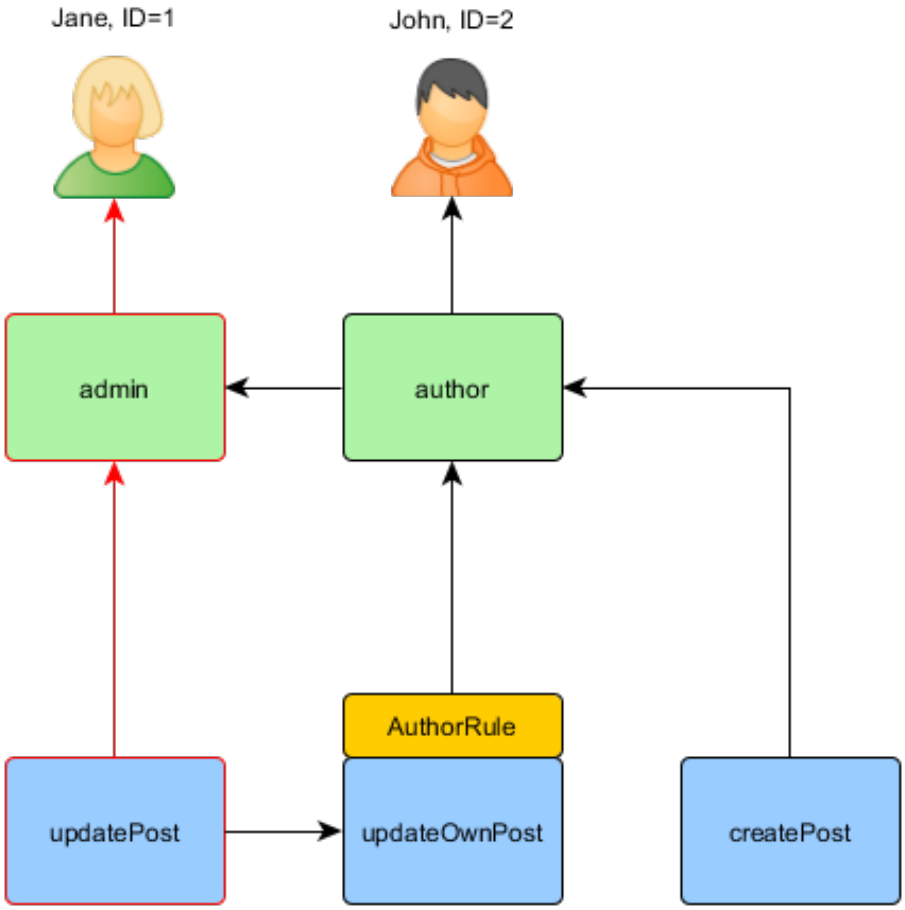
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {
    // actualizar post
}
```

Aquí es lo que sucede si el usuario actual es John:



Comenzamos desde `updatePost` y pasamos por `updateOwnPost`. Con el fin de pasar la comprobación de acceso, `AuthorRule` debe devolver `true` desde su método `execute()`. El método recive `$params` desde la llamada al método `can()`, cuyo valor es `['post' => $post]`. Si todo está bien, vamos a obtener `author`, el cual es asignado a John.

En caso de Jane es un poco más simple, ya que ella es un "admin":



Utilizar Roles por Defecto

Un rol por defecto es un rol que esta asignado *implícitamente* a *todos* los usuarios. La llamada a `yii\rbac\ManagerInterface::assign()` no es necesaria, y los datos de autorización no contienen su información de asignación.

Un rol por defecto es usualmente asociado con una regla que determina si el rol aplica al usuario siendo verificado.

Los roles por defecto se utilizan a menudo en aplicaciones que ya tienen algún tipo de asignación de roles. Por ejemplo, una aplicación puede tener una columna "grupo" en su tabla de usuario para representar a qué grupo de privilegio pertenece cada usuario. Si cada grupo privilegio puede ser conectado a un rol de RBAC, se puede utilizar la función de rol por defecto para asignar cada usuario a un rol RBAC automáticamente. Usemos un ejemplo para mostrar cómo se puede hacer esto.

Suponga que en la tabla de usuario, usted tiene una columna `group` que

utiliza 1 para representar el grupo administrador y 2 al grupo autor. Planeas tener dos roles RBAC, `admin` y `author`, para representar los permisos de estos dos grupos, respectivamente. Puede configurar los datos RBAC de la siguiente manera,

```
namespace app\rbac;

use Yii;
use yii\rbac\Rule;

/**
 * Comprueba si el grupo coincide
 */
class UserGroupRule extends Rule
{
    public $name = 'userGroup';

    public function execute($user, $item, $params)
    {
        if (!Yii::$app->user->isGuest) {
            $group = Yii::$app->user->identity->group;
            if ($item->name === 'admin') {
                return $group == 1;
            } elseif ($item->name === 'author') {
                return $group == 1 || $group == 2;
            }
        }
        return false;
    }
}

$auth = Yii::$app->authManager;

$rule = new \app\rbac\UserGroupRule;
$auth->add($rule);

$author = $auth->createRole('author');
$author->ruleName = $rule->name;
$auth->add($author);
// ... agrega permisos hijos a $author ...

$admin = $auth->createRole('admin');
$admin->ruleName = $rule->name;
$auth->add($admin);
$auth->addChild($admin, $author);
// ... agrega permisos hijos a $admin ...
```

Tenga en cuenta que en el ejemplo anterior, dado que "author" es agregado como hijo de "admin", cuando implementes el método `execute()` de la clase de la regla, necesitas respetar esta jerarquía. Esto se debe a que cuando el nombre del rol es "author", el método `execute()` devolverá `true` si el grupo de usuario es tanto 1 como 2 (lo que significa que el usuario se encuentra en

cualquiera de los dos grupos, “admin” o “author”).

Luego, configura `authManager` enumerando los dos roles en `yii\rbac\BaseManager::$defaultRoles`:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
            'defaultRoles' => ['admin', 'author'],
        ],
        // ...
    ],
];
```

Ahora si realizas una comprobación de acceso, tanto el rol `admin` y como el rol `author` serán comprobados evaluando las reglas asociadas con ellos. Si la regla devuelve `true`, significa que la regla aplica al usuario actual. Basado en la implementación de la regla anterior, esto significa que si el valor `group` en un usuario es 1, el rol `admin` se aplicaría al usuario; y si el valor de `group` es 2, se le aplicaría el rol `author`.

9.3. Trabajar con Passwords

La mayoría de los desarrolladores saben que los passwords no deben ser guardados en texto plano, pero muchos desarrolladores aún creen que es seguro aplicar a los passowrds hash `md5` o `sha1`. Hubo un tiempo cuando utilizar esos algoritmos de hash mencionados era suficiente, pero el hardware moderno hace posible que ese tipo de hash e incluso más fuertes, puedan revertirse rápidamente utilizando ataques de fuerza bruta.

Para poder proveer de una seguridad mayor para los passwords de los usuarios, incluso en el peor de los escenarios (tu aplicación sufre una brecha de seguridad), necesitas utilizar un algoritmo que resista los ataques de fuerza bruta. La mejor elección actualmente es `bcrypt`. En PHP, puedes generar un hash `bcrypt` utilizando la función crypt[3]. Yii provee dos funciones auxiliares que hacen que `crypt` genere y verifique los hash más fácilmente.

Cuando un usuario provee un password por primera vez (por ej., en la registración), dicho password necesita ser pasado por un hash:

```
$hash = Yii::$app->getSecurity()->generatePasswordHash($password);
```

El hash puede estar asociado con el atributo del model correspondiente, de manera que pueda ser almacenado en la base de datos para uso posterior.

Cuando un usuario intenta ingresar al sistema, el password enviado debe ser verificado con el password con hash almacenado previamente:

[3] https://www.php.net/manual/en/function.crypt.php

```php
if (Yii::$app->getSecurity()->validatePassword($password, $hash)) {
    // todo en orden, dejar ingresar al usuario
} else {
    // password erróneo
}
```

Error: not existing file: security-best-practices.md

Capítulo 10

Caché

10.1. El Almacenamiento en Caché

El almacenamiento en caché es una forma económica y eficaz para mejorar el rendimiento de una aplicación web. Mediante el almacenamiento de datos relativamente estáticos en la memoria caché y su correspondiente recuperación cuando éstos sean solicidatos, la aplicación salvaría todo ese tiempo y recursos necesarios para volver a generarlos cada vez desde cero.

El almacenamiento en caché se puede usar en diferentes niveles y lugares en una aplicación web. En el lado del servidor, al más bajo nivel, la caché puede ser usada para almacenar datos básicos, tales como una una lista de los artículos más recientes obtenidos de una base de datos; y en el más alto nivel, la caché puede ser usada para almacenar fragmentos o la totalidad de las páginas web, tales como el resultado del renderizado de los artículos más recientes. En el lado del cliente, el almacenamiento en caché HTTP puede ser utilizado para mantener el contenido de la página que ha sido visitada más recientemente en el caché del navegador.

Yii soporta los siguientes mecanismos de almacenamiento de caché:

- Caché de datos
- Caché de fragmentos
- Caché de páginas
- Caché HTTP

10.2. Almacenamiento de Datos en Caché

El almacenamiento de datos en caché trata del almacenamiento de alguna variable PHP en caché y recuperarla más tarde del mismo. También es la base de algunas de las características avanzadas de almacenamiento en caché, tales como el almacenamiento en caché de consultas a la base de datos y el almacenamiento en caché de contenido.

El siguiente código muestra el típico patrón de uso para el almacenamiento en caché, donde la variable `$cache` se refiere al componente caché:

```
// intenta recuperar $data de la caché
$data = $cache->get($key);

if ($data === false) {

    // $data no ha sido encontrada en la caché, calcularla desde cero

    // guardar $data en caché para así recuperarla la próxima vez
    $cache->set($key, $data);
}

// $data está disponible aquí
```

10.2.1. Componentes de Caché

El almacenamiento de datos en caché depende de los llamados *cache components* (componentes de caché) los cuales representan diferentes tipos de almacenamiento en caché, como por ejemplo en memoria, en archivos o en base de datos.

Los Componentes de Caché están normalmente registrados como componentes de la aplicación para que de esta forma puedan ser configurados y accesibles globalmente. El siguiente código muestra cómo configurar el componente de aplicación `cache` para usar memcached[1] con dos servidores caché:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
        'servers' => [
            [
                'host' => 'server1',
                'port' => 11211,
                'weight' => 100,
            ],
            [
                'host' => 'server2',
                'port' => 11211,
                'weight' => 50,
            ],
        ],
    ],
],
```

Puedes acceder al componente de caché usando la expresión `Yii::$app->cache`.

Debido a que todos los componentes de caché soportan el mismo conjunto de APIs, podrías cambiar el componente de caché subyacente por otro diferente mediante su reconfiguración en la configuración de la aplicación

[1] https://memcached.org/

sin tener que modificar el código que utiliza la caché. Por ejemplo, podrías modificar la configuración anterior para usar `APC cache`:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
],
```

> **Consejo:** Puedes registrar múltiples componentes de aplicación de caché. El componente llamado `cache` es usado por defecto por muchas clases caché-dependiente (ej. `yii\web\UrlManager`).

Almacenamientos de Caché Soportados

Yii proporciona varios componentes de caché que pueden almacenar datos en diferentes medios. A continuación se muestra un listado con los componentes de caché disponibles:

- `yii\caching\ApcCache`: utiliza la extensión de PHP APC[2]. Esta opción puede ser considerada como la más rápida de entre todas las disponibles para una aplicación centralizada. (ej. un servidor, no dedicado al balance de carga, etc).
- `yii\caching\DbCache`: utiliza una tabla de base de datos para almacenar los datos. Por defecto, se creará y usará como base de datos SQLite3[3] en el directorio runtime. Se puede especificar explícitamente que base de datos va a ser utilizada configurando la propiedad `db`.
- `yii\caching\DummyCache`: dummy cache (caché tonta) que no almacena en caché nada. El propósito de este componente es simplificar el código necesario para chequear la disponibilidad de caché. Por ejemplo, durante el desarrollo o si el servidor no tiene soporte de caché actualmente, puede utilizarse este componente de caché. Cuando este disponible un soporte en caché, puede cambiarse el componente correspondiente. En ambos casos, puede utilizarse el mismo código `Yii::$app->cache->get($key)` para recuperar un dato sin la preocupación de que `Yii::$app->cache` pueda ser `null`.
- `yii\caching\FileCache`: utiliza un fichero estándar para almacenar los datos. Esto es adecuado para almacenar grandes bloques de datos (como páginas).
- `yii\caching\MemCache`: utiliza las extensiones de PHP memcache[4] y memcached[5]. Esta opción puede ser considerada como la más rápida cuando la caché es manejada en una aplicación distribuida (ej. con varios servidores, con balance de carga, etc..)

[2] https://www.php.net/manual/es/book.apc.php
[3] https://sqlite.org/
[4] https://www.php.net/manual/es/book.memcache.php
[5] https://www.php.net/manual/es/book.memcached.php

- `yii\redis\Cache`: implementa un componente de caché basado en Redis[6] que almacenan pares clave-valor (requiere la versión 2.6.12 de redis).
- `yii\caching\WinCache`: utiliza la extensión de PHP WinCache[7] (ver también[8]).
- `yii\caching\XCache` *(deprecated)*: utiliza la extensión de PHP XCache[9].
- `yii\caching\ZendDataCache` *(deprecated)*: utiliza Zend Data Cache[10] como el medio fundamental de caché.

> **Consejo:** Puedes utilizar diferentes tipos de almacenamiento de caché en la misma aplicación. Una estrategia común es la de usar almacenamiento de caché en memoria para almacenar datos que son pequeños pero que son utilizados constantemente (ej. datos estadísticos), y utilizar el almacenamiento de caché en archivos o en base de datos para guardar datos que son grandes y utilizados con menor frecuencia (ej. contenido de página).

10.2.2. API de Caché

Todos los componentes de almacenamiento de caché provienen de la misma clase "padre" `yii\caching\Cache` y por lo tanto soportan la siguiente API:

- `get()`: recupera un elemento de datos de la memoria caché con una clave especificada. Un valor nulo será devuelto si el elemento de datos no ha sido encontrado en la memoria caché o si ha expirado o ha sido invalidado.
- `set()`: almacena un elemento de datos identificado por una clave en la memoria caché.
- `add()`: almacena un elemento de datos identificado por una clave en la memoria caché si la clave no se encuentra en la memoria caché.
- `mget()`: recupera varios elementos de datos de la memoria caché con las claves especificadas.
- `mset()`: almacena múltiples elementos de datos en la memoria caché. Cada elemento se identifica por una clave.
- `madd()`: almacena múltiples elementos de datos en la memoria caché. Cada elemento se identifica con una clave. Si una clave ya existe en la caché, el elemento será omitido.

[6] https://redis.io/
[7] https://iis.net/downloads/microsoft/wincache-extension
[8] https://www.php.net/manual/es/book.wincache.php
[9] https://en.wikipedia.org/wiki/List_of_PHP_accelerators#XCache
[10] https://files.zend.com/help/Zend-Server-6/zend-server.htm#data_cache_component.htm

- `exists()`: devuelve un valor que indica si la clave especificada se encuentra en la memoria caché.
- `delete()`: elimina un elemento de datos identificado por una clave de la caché.
- `flush()`: elimina todos los elementos de datos de la cache.

> **Nota:** No Almacenes el valor boolean `false` en caché directamente porque el método `get()` devuelve el valor `false` para indicar que el dato no ha sido encontrado en la caché. Puedes poner `false` dentro de un array y cachear este array para evitar este problema.

Algunos sistemas de almacenamiento de caché, como por ejemplo MemCache, APC, pueden recuperar múltiples valores almacenados en modo de lote (batch), lo que puede reducir considerablemente la sobrecarga que implica la recuperación de datos almacenados en la caché. Las API `mget()` y `madd()` se proporcionan para utilizar esta característica. En el caso de que el sistema de memoria caché no lo soportara, ésta sería simulada.

Puesto que `yii\caching\Cache` implementa `ArrayAccess`, un componente de caché puede ser usado como un array. El siguiente código muestra unos ejemplos:

```
$cache['var1'] = $value1;  // equivalente a: $cache->set('var1', $value1);
$value2 = $cache['var2'];  // equivalente a: $value2 = $cache->get('var2');
```

Claves de Caché

Cada elemento de datos almacenado en caché se identifica por una clave. Cuando se almacena un elemento de datos en la memoria caché, se debe especificar una clave. Más tarde, cuando se recupera el elemento de datos de la memoria caché, se debe proporcionar la clave correspondiente.

Puedes utilizar una cadena o un valor arbitrario como una clave de caché. Cuando una clave no es una cadena de texto, ésta será automáticamente serializada en una cadena.

Una estrategia común para definir una clave de caché es incluir en ella todos los factores determinantes en términos de un array. Por ejemplo, `yii\db\Schema` utiliza la siguiente clave para almacenar en caché la información del esquema de una tabla de base de datos:

```
[
    __CLASS__,              // nombre de la clase del esquema
    $this->db->dsn,         // nombre del origen de datos de la conexión BD
    $this->db->username,    // usuario para la conexión BD
    $name,                  // nombre de la tabla
];
```

Como puedes ver, la clave incluye toda la información necesaria para especificar de una forma exclusiva una tabla de base de datos.

Cuando en un mismo almacenamiento en caché es utilizado por diferentes aplicaciones, se debería especificar un prefijo único para las claves de la caché por cada una de las aplicaciones para así evitar conflictos. Esto puede hacerse mediante la configuración de la propiedad `yii\caching\Cache::$keyPrefix`. Por ejemplo, en la configuración de la aplicación podrías escribir el siguiente código:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
        'keyPrefix' => 'myapp',       // un prefijo de clave de caché único
    ],
],
```

Para garantizar la interoperabilidad, deberían utilizarse sólo caracteres alfanuméricos.

Caducidad de Caché

Un elemento de datos almacenado en la memoria caché permanecerá en ella para siempre, a menos que sea removida de alguna manera debido a alguna directiva de caché (ej. el espacio de almacenamiento en caché está lleno y los datos más antiguos se eliminan). Para cambiar este comportamiento, podrías proporcionar un parámetro de caducidad al llamar `set()` para guardar el elemento de datos. El parámetro nos indica por cuántos segundos el elemento se mantendrá válido en memoria caché. Cuando llames `get()` para recuperar el elemento, si el tiempo de caducidad ha pasado, el método devolverá `false`, indicando que el elemento de datos no ha sido encontrado en la memoria caché. Por ejemplo,

```
// guardar los datos en memoria caché al menos 45 segundos
$cache->set($key, $data, 45);

sleep(50);

$data = $cache->get($key);
if ($data === false) {
    // $data ha caducado o no ha sido encontrado en la memoria caché
}
```

Dependencias de Caché

Además de configurar el tiempo de caducidad, los datos almacenados en caché pueden también ser invalidados conforme a algunos cambios en la caché de dependencias. Por ejemplo, `yii\caching\FileDependency` representa la dependencia del tiempo de modificación del archivo. Cuando esta

dependencia cambia, significa que el archivo correspondiente ha cambiado. Como resultado, cualquier contenido anticuado que sea encontrado en la caché debería ser invalidado y la llamada a `get()` debería retornar falso.

Una dependencia es representada como una instancia de `yii\caching\Dependency` o su clase hija. Cuando llamas `set()` para almacenar un elemento de datos en la caché, puedes pasar el objeto de dependencia asociado. Por ejemplo,

```
// Crear una dependencia sobre el tiempo de modificación del archivo
example.txt.
$dependency = new \yii\caching\FileDependency(['fileName' =>
'example.txt']);

// Los datos expirarán en 30 segundos.
// También podría ser invalidada antes si example.txt es modificado.
$cache->set($key, $data, 30, $dependency);

// La caché comprobará si los datos han expirado.
// También comprobará si la dependencia ha cambiado.
// Devolverá `false` si se encuentran algunas de esas condiciones.
$data = $cache->get($key);
```

Aquí abajo se muestra un sumario de las dependencias disponibles:

- `yii\caching\ChainedDependency`: la dependencia cambia si cualquiera de las dependencias en la cadena cambia.
- `yii\caching\DbDependency`: la dependencia cambia si el resultado de la consulta de la sentencia SQL especificada cambia.
- `yii\caching\ExpressionDependency`: la dependencia cambia si el resultado de la expresión de PHP especificada cambia.
- `yii\caching\CallbackDependency`: la dipendenza viene modificata se il risultato della callback PHP specificata cambia.
- `yii\caching\FileDependency`: la dependencia cambia si se modifica la última fecha de modificación del archivo.
- `yii\caching\TagDependency`: marca un elemento de datos en caché con un nombre de grupo. Puedes invalidar los elementos de datos almacenados en caché con el mismo nombre del grupo a la vez llamando a `yii\caching\TagDependency::invalidate()`.

10.2.3. Consultas en Caché

Las consultas en caché es una característica especial de caché construido sobre el almacenamiento de caché de datos. Se proporciona para almacenar en caché el resultado de consultas a la base de datos.

Las consultas en caché requieren una `DB connection` y un componente de aplicación caché válido. El uso básico de las consultas en memoria caché es el siguiente, asumiendo que `db` es una instancia de `yii\db\Connection`:

```
$result = $db->cache(function ($db) {

    // el resultado de la consulta SQL será servida de la caché
    // si el cacheo de consultas está habilitado y el resultado de la
    consulta se encuentra en la caché
    return $db->createCommand('SELECT * FROM customer WHERE
    id=1')->queryOne();

});
```

El cacheo de consultas puede ser usado tanto para DAO como para Active-Record:

```
$result = Customer::getDb()->cache(function ($db) {
    return Customer::find()->where(['id' => 1])->one();
});
```

> **Nota:** Algunos DBMS (ej. MySQL[11]) también soporta el almacenamiento en caché desde el mismo servidor de la BD. Puedes optar por utilizar cualquiera de los mecanismos de memoria caché. El almacenamiento en caché de consultas previamente descrito tiene la ventaja que de que se puede especificar dependencias de caché de una forma flexible y son potencialmente mucho más eficientes.

Configuraciones

Las consultas en caché tienen tres opciones configurables globales a través de `yii\db\Connection`:

- `enableQueryCache`: activa o desactiva el cacheo de consultas. Por defecto es `true`. Tenga en cuenta que para activar el cacheo de consultas, también necesitas tener una caché válida, especificada por `queryCache`.
- `queryCacheDuration`: representa el número de segundos que un resultado de la consulta permanecerá válida en la memoria caché. Puedes usar 0 para indicar que el resultado de la consulta debe permanecer en la caché para siempre. Esta propiedad es el valor usado por defecto cuando `yii\db\Connection::cache()` es llamada sin especificar una duración.
- `queryCache`: representa el ID del componente de aplicación de caché. Por defecto es `'cache'`. El almacenamiento en caché de consultas se habilita sólo si hay un componente de la aplicación de caché válida.

Usos

Puedes usar `yii\db\Connection::cache()` si tienes multiples consultas SQL que necesitas a aprovechar el cacheo de consultas. El uso es de la siguiente manera,

[11] https://dev.mysql.com/doc/refman/5.6/en/query-cache.html

```
$duration = 60;     // resultado de la consulta de caché durante 60
segundos.
$dependency = ...;  // dependencia opcional

$result = $db->cache(function ($db) {

    // ... realiza consultas SQL aquí ...

    return $result;

}, $duration, $dependency);
```

Cualquier consulta SQL en una función anónima será cacheada durante el tiempo indicado con la dependencia especificada. Si el resultado de la consulta se encuentra válida en la caché, la consulta se omitirá y el resultado se servirá de la caché en su lugar. Si no especificar el parámetro `$duration`, el valor `queryCacheDuration` será usado en su lugar.

A veces dentro de `cache()`, puedes querer desactivar el cacheo de consultas para algunas consultas especificas. Puedes usar `yii\db\Connection::noCache()` en este caso.

```
$result = $db->cache(function ($db) {

    // consultas SQL que usan el cacheo de consultas

    $db->noCache(function ($db) {

        // consultas SQL que no usan el cacheo de consultas

    });

    // ...

    return $result;
});
```

Si lo deseas puedes usar el cacheo de consultas para una simple consulta, puedes llamar a `yii\db\Command::cache()` cuando construyas el comando. Por ejemplo,

```
// usa el cacheo de consultas y asigna la duración de la consulta de caché
por 60 segundos
$customer = $db->createCommand('SELECT * FROM customer WHERE
id=1')->cache(60)->queryOne();
```

También puedes usar `yii\db\Command::noCache()` para desactivar el cacheo de consultas de un simple comando. Por ejemplo,

```
$result = $db->cache(function ($db) {

    // consultas SQL que usan cacheo de consultas
```

```
    // no usa cacheo de consultas para este comando
    $customer = $db->createCommand('SELECT * FROM customer WHERE
    id=1')->noCache()->queryOne();

    // ...

    return $result;
});
```

Limitaciones

El almacenamiento en caché de consultas no funciona con los resultados de consulta que contienen controladores de recursos. Por ejemplo, cuando se utiliza el tipo de columna `BLOB` en algunos DBMS, el resultado de la consulta devolverá un recurso para manejar los datos de la columna.

Algunos sistemas de almacenamiento caché tienen limitación de tamaño. Por ejemplo, memcache limita el tamaño máximo de cada entrada a 1MB. Por lo tanto, si el tamaño de un resultado de la consulta excede ese límite, el almacenamiento en caché fallará.

10.3. Caché de Fragmentos

La Caché de Fragmentos se refiere al almacenamiento en caché de un fragmento, o sección, de una página Web. Por ejemplo, si una página muestra un sumario de las ventas anuales en una tabla, podrías guardar esta tabla en memoria caché para eliminar el tiempo necesario para generar esta tabla en cada petición (request). La caché de fragmentos está construido sobre la caché de datos.

Para usar la caché de fragmentos, utiliza el siguiente código en tu vista (view):

```
if ($this->beginCache($id)) {

    // ... generar contenido aquí ...

    $this->endCache();
}
```

Es decir, encierra la lógica de la generación del contenido entre las llamadas `beginCache()` y `endCache()`. Si el contenido se encuentra en la memoria caché, `beginCache()` mostrará el contenido y devolverá `false`, saltandose así la lógica de generación del contenido. De lo contrario, el código de generación se ejecutaría y al alcanzar la llamada `endCache()`, el contenido generado será capturado y almacenado en la memoria caché.

Como en la caché de datos, un `$id` (clave) único es necesario para identificar un contenido guardado en caché.

10.3.1. Opciones de Caché

Puedes especificar opciones adicionales para la caché de fragmentos pasando el array de opciones como segundo parametro del método `beginCache()`. Entre bastidores, este array de opciones se utiliza para configurar el widget `yii\widgets\FragmentCache` que es en realidad el que implementa la funcionalidad de la caché de fragmentos.

Duración

Quizás la opción más utilizada en la caché de fragmentos es `duración`. Ésta especifica cuántos segundos el contenido puede permanecer como válido en la memoria caché. El siguiente código almacena en la caché el fragmento de contenido para una hora a lo sumo:

```
if ($this->beginCache($id, ['duration' => 3600])) {

    // ... generar contenido aquí ...

    $this->endCache();
}
```

Si la opción no está activada, se tomará el valor por defecto 60, lo que significa que el contenido almacenado en caché expirará en 60 segundos.

Dependencias

Como en la caché de datos, el fragmento de contenido que está siendo almacenado en caché también puede tener dependencias. Por ejemplo, el contenido de un artículo que se muestre depende de si el mensaje se modifica o no.

Para especificar una dependencia, activa la opción `dependencia` (dependency), que puede ser un objecto `yii\caching\Dependency` o un array de configuración para crear un objecto `Dependency`. El siguiente código especifica que la caché de fragmento depende del cambio del valor de la columna `updated_at`:

```
$dependency = [
    'class' => 'yii\caching\DbDependency',
    'sql' => 'SELECT MAX(updated_at) FROM post',
];

if ($this->beginCache($id, ['dependency' => $dependency])) {

    // ... generar contenido aquí ...

    $this->endCache();
}
```

Variaciones

El contenido almacenado en caché puede variar de acuerdo a ciertos parámetros. Por ejemplo, para una aplicación Web que soporte multiples idiomas, la misma pieza del código de la vista puede generar el contenido almacenado en caché en diferentes idiomas. Por lo tanto, es posible que desees hacer variaciones del mismo contenido almacenado en caché de acuerdo con la actual selección del idioma en la aplicación.

Para especificar variaciones en la memoria caché, configura la opción `variaciones` (variations), la cual deberá ser un array de valores escalares, cada uno de ellos representando un factor de variación. Por ejemplo, para hacer que el contenido almacenado en la caché varíe por lenguaje, podrías usar el siguiente código:

```
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {

    // ... generar código aquí ...

    $this->endCache();
}
```

Alternando el Almacenamiento en Caché

Puede que a veces quieras habilitar la caché de fragmentos únicamente cuando ciertas condiciones se cumplan. Por ejemplo, para una página que muestra un formulario, tal vez quieras guardarlo en la caché cuando es inicialmente solicitado (a través de una petición GET). Cualquier muestra posterior (a través de una petición POST) del formulario no debería ser almacenada en caché ya que el formulario puede que contenga entradas del usuario. Para hacerlo, podrías configurar la opción de `activado` (enabled), de la siguiente manera:

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {

    // ... generar contenido aquí ...

    $this->endCache();
}
```

10.3.2. Almacenamiento en Caché Anidada

El almacenamiento en caché de fragmentos se puede anidar. Es decir, un fragmento de caché puede ser encerrado dentro de otro fragmento que también se almacena en caché. Por ejemplo, los comentarios se almacenan en una caché de fragmento interno, y se almacenan conjuntamente con el contenido del artículo en un fragmento de caché exterior. El siguiente código muestra cómo dos fragmentos de caché pueden ser anidados:

```php
if ($this->beginCache($id1)) {

    // ... lógica de generación de contenido externa ...

    if ($this->beginCache($id2, $options2)) {

        // ... lógica de generación de contenido anidada ...

        $this->endCache();
    }

    // ... lógica de generación de contenido externa ...

    $this->endCache();
}
```

Existen diferentes opciones de configuración para las cachés anidadas. Por ejemplo, las cachés internas y las cachés externas pueden usar diferentes valores de duración. Aún cuando los datos almacenados en la caché externa sean invalidados, la caché interna puede todavía proporcionar un fragmento válido. Sin embargo, al revés no es cierto. Si la caché externa es evaluada como válida, seguiría proporcionando la misma copia en caché incluso después de que el contenido en la caché interna haya sido invalidada. Por lo tanto, hay que tener mucho cuidado al configurar el tiempo de duración o las dependencias de las cachés anidadas, de lo contrario los fragmentos internos que ya estén obsoletos se pueden seguir manteniendo en el fragmento externo.

10.3.3. Contenido Dinámico

Cuando se usa la caché de fragmentos, podrías encontrarte en la situación que un fragmento grande de contenido es relavitamente estático excepto en uno u otro lugar. Por ejemplo, la cabeza de una página (header) puede que muestre el menú principal junto al nombre del usuario actual. Otro problema es que el contenido que está siendo almacenado en caché puede que contenga código PHP que debe ser ejecutado en cada petición (por ejemplo, el código para registrar un paquete de recursos (asset bundle)). En ambos casos, podríamos resolver el problema con lo que llamamos la característica de *contenido dinámico*.

Entendemos *contenido dinámico* como un fragmento de salida que no debería ser guardado en caché incluso si está encerrado dentro de un fragmento de caché. Para hacer el contenido dinámico todo el tiempo, éste ha de ser generado ejecutando cierto código PHP en cada petición, incluso si el contenido está siendo mostrado desde la caché.

Puedes llamar a `yii\base\View::renderDynamic()` dentro de un fragmento almacenado en caché para insertar código dinámico en el lugar deseado como, por ejemplo, de la siguiente manera,

```
if ($this->beginCache($id1)) {

    // ... lógica de generación de contenido ...

    echo $this->renderDynamic('return Yii::$app->user->identity->name;');

    // ... lógica de generación de contenido ...

    $this->endCache();
}
```

El método `renderDynamic()` toma una pieza de código PHP como su parámetro. El valor devuelto del código PHP se trata como contenido dinámico. El mismo código PHP será ejecutado en cada petición, sin importar que esté dentro de un fragmento que está siendo servido desde la caché o no.

10.4. Caché de Páginas

La caché de páginas se refiere a guardar el contenido de toda una página en el almacenamiento de caché del servidor. Posteriormente, cuando la misma página sea requerida de nuevo, su contenido será devuelto desde la caché en vez de volver a generarlo desde cero.

El almacenamiento en caché de páginas está soportado por `yii\filters\PageCache`, un filtro de acción. Puede ser utilizado de la siguiente forma en un controlador:

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\PageCache',
            'only' => ['index'],
            'duration' => 60,
            'variations' => [
                \Yii::$app->language,
            ],
            'dependency' => [
                'class' => 'yii\caching\DbDependency',
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
        ],
    ];
}
```

El código anterior establece que el almacenamiento de páginas en caché debe ser utilizado sólo en la acción `index`; el contenido de la página debería almacenarse durante un máximo de 60 segundos y ser variado por el idioma actual de la aplicación; además, el almacenamiento de la página en caché debería ser invalidado si el número total de artículos ha cambiado.

Como puedes ver, la caché de páginas es muy similar a la caché de fragmentos. Ambos soportan opciones tales como `duration`, `dependencies`, `variations`, y `enabled`. Su principal diferencia es que la caché de páginas está implementado como un filtro de acción mientras que la caché de fragmentos se hace en un widget.

Puedes usar la caché de fragmentos así como el contenido dinámico junto con la caché de páginas.

10.5. Caché HTTP

Además del almacenamiento de caché en el servidor que hemos descrito en secciones anteriores, las aplicaciones Web pueden hacer uso de la caché en el lado del cliente para así ahorrar tiempo y recursos para generar y transmitir el mismo contenido una y otra vez.

Para usar la caché del lado del cliente, puedes configurar `yii\filters\HttpCache` como un filtro en el controlador para aquellas acciones cuyo resultado deba estar almacenado en la caché en el lado del cliente. `HttpCache` solo funciona en peticiones `GET` y `HEAD`. Puede manejar tres tipos de cabeceras (headers) HTTP relacionadas en este tipo de consultas:

- `Last-Modified`
- `Etag`
- `Cache-Control`

10.5.1. La Cabecera

La cabecera `Last-Modified` usa un sello de tiempo para indicar si la página ha sido modificada desde que el cliente la almacena en la caché.

Puedes configurar la propiedad `yii\filters\HttpCache::$lastModified` para activar el envío de la cabecera `Last-Modified`. La propiedad debe ser una llamada de retorno (callable) PHP que devuelva un timestamp UNIX sobre el tiempo de modificación de la página. El formato de la función de llamada de retorno debe ser el siguiente,

```
/**
 * @param Action $action el objeto acción que se está controlando
actualmente
 * @param array $params el valor de la propiedad "params"
 * @return int un sello de tiempo UNIX que representa el tiempo de
modificación de la página
 */
function ($action, $params)
```

El siguiente es un ejemplo haciendo uso de la cabecera `Last-Modified`:

```
public function behaviors()
{
```

```
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('post')->max('updated_at');
            },
        ],
    ];
}
```

El código anterior establece que la memoria caché HTTP debe ser habilitada únicamente por la acción `index`. Se debe generar una cabecera HTTP `Last-Modified` basado en el último tiempo de actualización de los artículos. Cuando un navegador visita la página `index` la primera vez, la página será generada en el servidor y enviada al navegador; Si el navegador visita la misma página de nuevo y no ningún artículo modificado durante el período, el servidor no volverá a regenerar la página, y el navegador usará la versión caché del lado del cliente. Como resultado, la representación del lado del servidor y la transmisión del contenido de la página son ambos omitidos.

10.5.2. La Cabecera

La cabecera "Entity Tag" (o para abreviar `ETag`) usa un hash para representar el contenido de una página. Si la página ha sido cambiada, el hash también cambiará. Al comparar el hash guardado en el lado del cliente con el hash generado en el servidor, la caché puede determinar si la página ha cambiado y deber ser retransmitida.

Puedes configurar la propiedad `yii\filters\HttpCache::$etagSeed` para activar el envío de la cabecera `ETag`. La propiedad debe ser una función de retorno (callable) PHP que devuelva una semilla para la generación del hash de `ETag`. El formato de la función de retorno es el siguiente:

```
/**
 * @param Action $action el objeto acción que se está controlando
actualmente
 * @param array $params el valor de la propiedad "params"
 * @return string una cadena usada como semilla para la generación del hash
de ETag
 */
function ($action, $params)
```

El siguiente es un ejemplo de cómo usar la cabecera `ETag`:

```
public function behaviors()
{
    return [
        [
```

```
            'class' => 'yii\filters\HttpCache',
            'only' => ['view'],
            'etagSeed' => function ($action, $params) {
                $post = $this->findModel(\Yii::$app->request->get('id'));
                return serialize([$post->title, $post->content]);
            },
        ],
    ];
}
```

El código anterior establece que la caché HTTP debe ser activada únicamente para la acción `view`. Debería generar una cabecera HTTP `ETag` basándose en el título y contenido del artículo consultado. Cuando un navegador visita la página `view` por primera vez, la página se generará en el servidor y será enviada al navegador; Si el navegador visita la misma página de nuevo y no ha ocurrido un cambio en el título o contenido del artículo, el servidor no volverá a generar la página, y el navegador usará la versión guardada en la caché del lado del cliente. Como resultado, la representación del lado del servidor y la transmisión del contenido de la página son ambos omitidos.

ETags permiten estrategias de almacenamiento de caché más complejas y/o mucho más precisas que las cabeceras `Last-Modified`. Por ejemplo, un ETag puede ser invalidado si el sitio Web ha cambiado de tema (theme).

La generación de un ETag que requiera muchos recursos puede echar por tierra el propósito de estar usando `HttpCache` e introducir una sobrecarga innecesaria, ya que debe ser re-evaluada en cada solicitud (request). Trata de encontrar una expresión sencilla para invalidar la caché si la página ha sido modificada.

> **Nota:** En cumplimiento con RFC 7232[12], `HttpCache` enviará ambas cabeceras `ETag` y `Last-Modified` si ambas están configuradas. Y si el clientes envía tanto la cabecera `If-None-Match` como la cabecera `If-Modified-Since`, solo la primera será respetada.

10.5.3. La Cabecera

La cabecera `Cache-Control` especifica la directiva general de la caché para páginas. Puedes enviarla configurando la propiedad `yii\filters\HttpCache::$cacheControlHeader` con el valor de la cabecera. Por defecto, la siguiente cabecera será enviada:

```
Cache-Control: public, max-age=3600
```

10.5.4. Limitador de la Sesión de Caché

Cuando una página utiliza la sesión, PHP enviará automáticamente cabeceras HTTP relacionadas con la caché tal y como se especifican en `session.cache_limiter`

[12] https://datatracker.ietf.org/doc/html/rfc7232#section-2.4

de la configuración INI de PHP. Estas cabeceras pueden interferir o deshabilitar el almacenamiento de caché que desees de `HttpCache`. Para evitar este problema, por defecto `HttpCache` deshabilitará automáticamente el envío de estas cabeceras. Si deseas modificar este comportamiento, tienes que configurar la propiedad `yii\filters\HttpCache::$sessionCacheLimiter`. La propiedad puede tomar un valor de cadena, incluyendo `public`, `private`, `private_no_expire`, and `nocache`. Por favor, consulta el manual PHP acerca de session_cache_limiter()[13] para una mejor explicación sobre esos valores.

10.5.5. Implicaciones SEO

Los robots de motores de búsqueda tienden a respetar las cabeceras de caché. Dado que algunos `crawlers` tienen limitado el número de páginas que pueden rastrear por dominios dentro de un cierto período de tiempo, la introducción de cabeceras de caché pueden ayudar a la indexación del sitio Web y reducir el número de páginas que deben ser procesadas.

[13] https://www.php.net/manual/es/function.session-cache-limiter.php

Capítulo 11

Servicios Web RESTful

11.1. Guía Breve

Yii ofrece todo un conjunto de herramientas para simplificar la tarea de implementar un servicio web APIs RESTful. En particular, Yii soporta las siguientes características sobre APIs RESTful;

- Prototipado rápido con soporte para APIs comunes para Active Record;
- Formato de respuesta de negocio (soporta JSON y XML por defecto);
- Personalización de objetos serializados con soporte para campos de salida seleccionables;
- Formateo apropiado de colecciones de datos y validación de errores;
- Soporte para HATEOAS[1];
- Eficiente enrutamiento con una adecuada comprobación del verbo(verb) HTTP;
- Incorporado soporte para las `OPTIONS` y `HEAD` verbos;
- Autenticación y autorización;
- Cacheo de datos y cacheo HTTP;
- Limitación de rango;

A continuación, utilizamos un ejemplo para ilustrar como se puede construir un conjunto de APIs RESTful con un esfuerzo mínimo de codificación.

Supongamos que deseas exponer los datos de los usuarios vía APIs RESTful. Los datos de usuario son almacenados en la tabla DB `user`, y ya tienes creado la clase `ActiveRecord` `app\models\User` para acceder a los datos del usuario.

11.1.1. Creando un controlador

Primero, crea una clase controladora `app\controllers\UserController` como la siguiente,

[1] https://es.wikipedia.org/wiki/HATEOAS

```
namespace app\controllers;

use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
}
```

La clase controladora extiende de `yii\rest\ActiveController`. Especificado por `modelClass` como `app\models\User`, el controlador sabe que modelo puede ser usado para recoger y manipular sus datos.

11.1.2. Configurando las reglas de las URL

A continuación, modifica la configuración del componente `urlManager` en la configuración de tu aplicación:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

La configuración anterior principalmente añade una regla URL para el controlador `user` de manera que los datos de user pueden ser accedidos y manipulados con URLs amigables y verbos HTTP significativos.

11.1.3. Habilitando entradas JSON

Para permitir que la API acepte datos de entrada con formato JSON, configura la propiedad `parsers` del componente de aplicación `request` para usar `yii\web\JsonParser` para entradas JSON:

```
'request' => [
    'parsers' => [
        'application/json' => 'yii\web\JsonParser',
    ]
]
```

> **Consejo:** La configuración anterior es opcional. Sin la configuración anterior, la API sólo reconocería `application/x-www-form-urlencoded` y `multipart/form-data` como formatos de entrada.

11.1.4. Probándolo

Con la mínima cantidad de esfuerzo, tienes ya finalizado tu tarea de crear las APIs RESTful para acceder a los datos de user. Las APIs que tienes creado incluyen:

- `GET /users`: una lista de todos los usuarios página por página;
- `HEAD /users`: muestra la información general de la lista de usuarios;
- `POST /users`: crea un nuevo usuario;
- `GET /users/123`: devuelve los detalles del usuario 123;
- `HEAD /users/123`: muestra la información general del usuario 123;
- `PATCH /users/123` y `PUT /users/123`: actualiza el usuario 123;
- `DELETE /users/123`: elimina el usuario 123;
- `OPTIONS /users`: muestra los verbos compatibles respecto al punto final `/users`;
- `OPTIONS /users/123`: muestra los verbos compatibles respecto al punto final `/users/123`.

> **Información:** Yii automáticamente pluraliza los nombres de los controladores para usarlo en los puntos finales. Puedes configurar esto usando la propiedad `yii\rest\UrlRule::$pluralize`.

Puedes acceder a tus APIs con el comando `curl` de la siguiente manera,

```
$ curl -i -H "Accept:application/json" "http://localhost/users"

HTTP/1.1 200 OK
...
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

[
    {
        "id": 1,
        ...
    },
    {
        "id": 2,
        ...
    },
    ...
]
```

Intenta cambiar el tipo de contenido aceptado para ser `application/xml`, y verá que el resultado se devuelve en formato XML:

```
$ curl -i -H "Accept:application/xml" "http://localhost/users"

HTTP/1.1 200 OK
...
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<response>
    <item>
        <id>1</id>
        ...
    </item>
    <item>
        <id>2</id>
        ...
    </item>
    ...
</response>
```

El siguiente comando creará un nuevo usuario mediante el envío de una petición POST con los datos del usuario en formato JSON:

```
$ curl -i -H "Accept:application/json" -H "Content-Type:application/json"
-XPOST "http://localhost/users" -d '{"username": "example", "email":
"user@example.com"}'

HTTP/1.1 201 Created
...
Location: http://localhost/users/1
Content-Length: 99
Content-Type: application/json; charset=UTF-8

{"id":1,"username":"example","email":"user@example.com","created_at":1414674789,"updated_at":1414
```

> **Consejo:** También puedes acceder a tus APIs a través del navegador web introduciendo la URL `http://localhost/users`. Sin embargo, es posible que necesites algunos plugins para el navegador para enviar cabeceras especificas en la petición.

Como se puede ver, en las cabeceras de la respuesta, hay información sobre la cuenta total, número de páginas, etc. También hay enlaces que permiten navegar por otras páginas de datos. Por ejemplo, `http://localhost/users?page=2` le daría la página siguiente de los datos de usuario.

Utilizando los parámetros `fields` y `expand`, puedes también especificar que campos deberían ser incluidos en el resultado. Por ejemplo, la URL `http://localhost/users?fields=id,email` sólo devolverá los campos `id` y `email`.

> **Información:** Puedes haber notado que el resultado de `http://localhost/users` incluye algunos campos sensibles, tal como `password_hash`, `auth_key`. Seguramente no quieras que éstos aparecieran en el resultado de tu API. Puedes y deberías filtrar estos campos como se describe en la sección Response Formatting.

11.1.5. Resumen

Utilizando el framework Yii API RESTful, implementa un punto final API en términos de una acción de un controlador, y utiliza un controlador para organizar las acciones que implementan los puntos finales para un sólo tipo de recurso.

Los recursos son representados como modelos de datos que extienden de la clase `yii\base\Model`. Si estás trabajando con bases de datos (relacionales o NoSQL), es recomendable utilizar `ActiveRecord` para representar los recursos.

Puedes utilizar `yii\rest\UrlRule` para simplificar el enrutamiento de los puntos finales de tu API.

Aunque no es obligatorio, es recomendable que desarrolles tus APIs RESTful como una aplicación separada, diferente de tu WEB front end y tu back end para facilitar el mantenimiento.

11.2. Recursos

Las APIs RESTful lo son todos para acceder y manipular *recursos (resources)*. Puedes observar los recursos en el paradigma MVC en Modelos (models) .

Mientras que no hay restricción a cómo representar un recurso, en YII usualmente, puedes representar recursos como objetos de la clase `yii\base\Model` o sus clases hijas (p.e. `yii\db\ActiveRecord`), por las siguientes razones:

- `yii\base\Model` implementa el interface `yii\base\Arrayable` , el cual te permite personalizar como exponer los datos de los recursos a través de las APIs RESTful.
- `yii\base\Model` soporta Validación de entrada (input validation), lo cual es muy usado en las APIs RESTful para soportar la entrada de datos.
- `yii\db\ActiveRecord` provee un poderoso soporte para el acceso a datos en bases de datos y su manipulación, lo que lo le hace servir perfectamente si sus recursos de datos están en bases de datos.

En esta sección, vamos principalmente a describir como la clase con recursos que extiende de `yii\base\Model` (o sus clases hijas) puede especificar qué datos puede ser devueltos vía las APIs RESTful. Si la clase de los recursos no extiende de `yii\base\Model`, entonces todas las variables públicas miembro serán devueltas.

11.2.1. Campos (fields)

Cuando incluimos un recurso en una respuesta de la API RESTful, el recurso necesita ser serializado en una cadena. Yii divide este proceso en dos pasos. Primero, el recurso es convertido en un array por `yii\rest\Serializer`. Segundo, el array es serializado en una cadena en el formato requerido (p.e. JSON, XML) por `response formatters`. El primer paso es en el que debes de concentrarte principalmente cuando desarrolles una clase de un recurso.

Sobreescribiendo `fields()` y/o `extraFields()`, puedes especificar qué datos, llamados *fields*, en el recursos, pueden ser colocados en el array que le representa. La diferencia entre estos dos métodos es que el primero especifica el conjunto de campos por defecto que deben ser incluidos en el array que los representa, mientras que el último especifica campos adicionales que deben de ser incluidos en el array si una petición del usuario final para ellos vía el parámetro de consulta `expand`. Por ejemplo,

```
// devuelve todos los campos declarados en fields()
http://localhost/users

// sólo devuelve los campos id y email, provistos por su declaración en
fields()
http://localhost/users?fields=id,email

// devuelve todos los campos en fields() y el campo profile siempre y cuando
esté declarado en extraFields()
http://localhost/users?expand=profile

// sólo devuelve los campos id, email y profile, siempre y cuando ellos
estén declarados en fields() y extraFields()
http://localhost/users?fields=id,email&expand=profile
```

Sobreescribiendo

Por defecto, `yii\base\Model::fields()` devuelve todos los atributos de los modelos como si fueran campos, mientras `yii\db\ActiveRecord::fields()` sólo devuelve los atributos que tengan datos en la base de datos.

Puedes sobreescribir `fields()` para añadir, quitar, renombrar o redefinir campos. El valor de retorno de `fields()` ha de estar en un array. Las claves del array son los nombres de los campos y los valores del array son las correspondientes definiciones de los campos que pueden ser tanto nombres

de propiedades/atributos o funciones anónimas que devuelven los correspondientes valores del los campos. En el caso especial de que el nombre de un campo sea el mismo que su definición puedes omitir la clave en el array. Por ejemplo,

```
// explícitamente lista cada campo, siendo mejor usarlo cuando quieras
asegurarte que los cambios
// en una tabla de la base de datos o en un atributo del modelo no provoque
el cambio de tu campo (para mantener la compatibilidad anterior).
public function fields()
{
    return [
        // el nombre de campo es el mismo nombre del atributo
        'id',
        // el nombre del campo es "email", su atributo se denomina
        "email_address"
        'email' => 'email_address',
        // el nombre del campo es "name", su valor es definido está definido
        por una función anónima de retrollamada (callback)
        'name' => function () {
            return $this->first_name . ' ' . $this->last_name;
        },
    ];
}

// el ignorar algunos campos, es mejor usarlo cuando heredas de una
implementación padre
// y pones en la lista negra (blacklist) algunos campos sensibles
public function fields()
{
    $fields = parent::fields();

    // quita los campos con información sensible
    unset($fields['auth_key'], $fields['password_hash'],
    $fields['password_reset_token']);

    return $fields;
}
```

> **Aviso:** Dado que, por defecto, todos los atributos de un modelo pueden ser incluidos en la devolución del API, debes examinar tus datos para estar seguro de que no contiene información sensible. Si se da este tipo de información, debes sobreescribir `fields()` para filtrarlos. En el ejemplo anterior, escogemos quitar `auth_key`, `password_hash` y `password_reset_token`.

Sobreescribiendo

Por defecto, `yii\base\Model::extraFields()` no devuelve nada, mientras que `yii\db\ActiveRecord::extraFields()` devuelve los nombres de las relaciones que tienen datos (populated) obtenidos de la base de datos.

El formato de devolución de los datos de `extraFields()` es el mismo que el de `fields()`. Usualmente, `extraFields()` es principalmente usado para especificar campos cuyos valores sean objetos. Por ejemplo, dado la siguiente declaración de campo,

```
public function fields()
{
    return ['id', 'email'];
}

public function extraFields()
{
    return ['profile'];
}
```

la petición `http://localhost/users?fields=id,email&expand=profile` puede devolver los siguientes datos en formato JSON :

```
[
    {
        "id": 100,
        "email": "100@example.com",
        "profile": {
            "id": 100,
            "age": 30,
        }
    },
    ...
]
```

11.2.2. Enlaces (Links)

HATEOAS[2], es una abreviación de Hipermedia es el Motor del Estado de la Aplicación (Hypermedia as the Engine of Application State), promueve que las APIs RESTfull devuelvan información que permita a los clientes descubrir las acciones que soportan los recursos devueltos. El sentido de HATEOAS es devolver un conjunto de hiperenlaces con relación a la información, cuando los datos de los recursos son servidos por las APIs.

Las clases con recursos pueden soportar HATEOAS implementando el interfaz `yii\web\Linkable` . El interfaz contiene sólo un método `getLinks()` el cual debe de de devolver una lista de `links`. Típicamente, debes devolver al menos un enlace `self` representando la URL al mismo recurso objeto. Por ejemplo,

```
use yii\db\ActiveRecord;
use yii\web\Link;
use yii\web\Linkable;
use yii\helpers\Url;
```

[2] https://es.wikipedia.org/wiki/HATEOAS

```
class User extends ActiveRecord implements Linkable
{
    public function getLinks()
    {
        return [
            Link::REL_SELF => Url::to(['user/view', 'id' => $this->id],
            true),
        ];
    }
}
```

Cuando un objeto `User` es devuelto en una respuesta, puede contener un elemento `_links` representando los enlaces relacionados con el usuario, por ejemplo,

```
{
    "id": 100,
    "email": "user@example.com",
    // ...
    "_links" => {
        "self": {
            "href": "https://example.com/users/100"
        }
    }
}
```

11.2.3. Colecciones

Los objetos de los recursos pueden ser agrupados en *collections*. Cada colección contiene una lista de recursos objeto del mismo tipo.

Las colecciones pueden ser representadas como arrays pero, es usualmente más deseable representarlas como proveedores de datos (data providers). Esto es así porque los proveedores de datos soportan paginación y ordenación de los recursos, lo cual es comunmente necesario en las colecciones devueltas con las APIs RESTful. Por ejemplo, la siguiente acción devuelve un proveedor de datos sobre los recursos post:

```
namespace app\controllers;

use yii\rest\Controller;
use yii\data\ActiveDataProvider;
use app\models\Post;

class PostController extends Controller
{
    public function actionIndex()
    {
        return new ActiveDataProvider([
            'query' => Post::find(),
        ]);
```

```
    }
}
```

Cuando un proveedor de datos está enviando una respuesta con el API RESTful, `yii\rest\Serializer` llevará la actual página de los recursos y los serializa como un array de recursos objeto. Adicionalmente, `yii\rest\Serializer` puede incluir también la información de paginación a través de las cabeceras HTTP siguientes:

- `X-Pagination-Total-Count`: Número total de recursos;
- `X-Pagination-Page-Count`: Número de páginas;
- `X-Pagination-Current-Page`: Página actual (iniciando en 1);
- `X-Pagination-Per-Page`: Número de recursos por página;
- `Link`: Un conjunto de enlaces de navegación permitiendo al cliente recorrer los recursos página a página.

Un ejemplo se puede ver en la sección Inicio rápido (Quick Start).

11.3. Controladores

Después de crear las clases de recursos y especificar cómo debe ser el formato de datos de recursos, el siguiente paso es crear acciones del controlador para exponer los recursos a los usuarios finales a través de las APIs RESTful.

Yii ofrece dos clases controlador base para simplificar tu trabajo de crear acciones REST: `yii\rest\Controller` y `yii\rest\ActiveController`. La diferencia entre estos dos controladores es que este último proporciona un conjunto predeterminado de acciones que están específicamente diseñado para trabajar con los recursos representados como Active Record. Así que si estás utilizando Active Record y te sientes cómodo con las acciones integradas provistas, podrías considerar extender tus controladores de `yii\rest\ActiveController`, lo que te permitirá crear potentes APIs RESTful con un mínimo de código.

Ambos `yii\rest\Controller` y `yii\rest\ActiveController` proporcionan las siguientes características, algunas de las cuales se describen en detalle en las siguientes secciones:

- Método de Validación HTTP;
- Negociación de contenido y formato de datos;
- Autenticación;
- Límite de Rango.

`yii\rest\ActiveController` además provee de las siguientes características:

- Un conjunto de acciones comunes necesarias: `index`, `view`, `create`, `update`, `delete`, `options`;
- La autorización del usuario de acuerdo a la acción y recurso solicitado.

11.3.1. Creando Clases de Controlador

Al crear una nueva clase de controlador, una convención para nombrar la clase del controlador es utilizar el nombre del tipo de recurso en singular. Por ejemplo, para servir información de usuario, el controlador puede ser nombrado como `UserController`.

Crear una nueva acción es similar a crear una acción para una aplicación Web. La única diferencia es que en lugar de renderizar el resultado utilizando una vista llamando al método `render()`, para las acciones REST regresas directamente los datos. El `serializer` y el `response object` se encargarán de la conversión de los datos originales al formato solicitado. Por ejemplo,

```
public function actionView($id)
{
    return User::findOne($id);
}
```

11.3.2. Filtros

La mayoría de las características API REST son proporcionadas por `yii\rest\Controller` son implementadas en los términos de filtros. En particular, los siguientes filtros se ejecutarán en el orden en que aparecen:

- `contentNegotiator`: soporta la negociación de contenido, que se explica en la sección Formateo de respuestas;
- `verbFilter`: soporta métodos de validación HTTP;
- `authenticator`: soporta la autenticación de usuarios, que se explica en la sección Autenticación;
- `rateLimiter`: soporta la limitación de rango, que se explica en la sección Límite de Rango.

Estos filtros se declaran nombrándolos en el método `behaviors()`. Puede sobrescribir este método para configurar filtros individuales, desactivar algunos de ellos, o añadir los tuyos. Por ejemplo, si sólo deseas utilizar la autenticación básica HTTP, puede escribir el siguiente código:

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::class,
    ];
    return $behaviors;
}
```

11.3.3. Extendiendo

Si tu clase controlador extiende de `yii\rest\ActiveController`, debe establecer su propiedad `modelClass` con el nombre de la clase del recurso

que planeas servir a través de este controlador. La clase debe extender de `yii\db\ActiveRecord`.

Personalizando Acciones

Por defecto, `yii\rest\ActiveController` provee de las siguientes acciones:

- `index`: listar recursos página por página;
- `view`: devolver el detalle de un recurso específico;
- `create`: crear un nuevo recurso;
- `update`: actualizar un recurso existente;
- `delete`: eliminar un recurso específico;
- `options`: devolver los métodos HTTP soportados.

Todas esta acciones se declaran a través de método `actions()`. Puedes configurar estas acciones o desactivar alguna de ellas sobrescribiendo el método `actions()`, como se muestra a continuación,

```
public function actions()
{
    $actions = parent::actions();

    // disable the "delete" and "create" actions
    unset($actions['delete'], $actions['create']);

    // customize the data provider preparation with the
    "prepareDataProvider()" method
    $actions['index']['prepareDataProvider'] = [$this,
    'prepareDataProvider'];

    return $actions;
}

public function prepareDataProvider()
{
    // prepare and return a data provider for the "index" action
}
```

Por favor, consulta las referencias de clases de acciones individuales para aprender las opciones de configuración disponibles para cada una.

Realizando Comprobación de Acceso

Al exponer los recursos a través de RESTful APIs, a menudo es necesario comprobar si el usuario actual tiene permiso para acceder y manipular el/los recurso solicitado/s. Con `yii\rest\ActiveController`, esto puede lograrse sobrescribiendo el método `checkAccess()` como a continuación,

```
/**
 * Checks the privilege of the current user.
```

```
     *
     * This method should be overridden to check whether the current user has
    the privilege
     * to run the specified action against the specified data model.
     * If the user does not have access, a [[ForbiddenHttpException]] should be
    thrown.
     *
     * @param string $action the ID of the action to be executed
     * @param \yii\base\Model $model the model to be accessed. If `null`, it
    means no specific model is being accessed.
     * @param array $params additional parameters
     * @throws ForbiddenHttpException if the user does not have access
     */
    public function checkAccess($action, $model = null, $params = [])
    {
        // check if the user can access $action and $model
        // throw ForbiddenHttpException if access should be denied
        if ($action === 'update' || $action === 'delete') {
            if ($model->author_id !== \Yii::$app->user->id)
                throw new \yii\web\ForbiddenHttpException(sprintf('You can only
                %s articles that you\'ve created.', $action));
        }
    }
```

El método `checkAccess()` será llamado por defecto en las acciones predeterminadas de `yii\rest\ActiveController`. Si creas nuevas acciones y también deseas llevar a cabo la comprobación de acceso, debe llamar a este método de forma explícita en las nuevas acciones.

> **Consejo:** Puedes implementar `checkAccess()` mediante el uso del Componente Role-Based Access Control (RBAC).

11.4. Enrutamiento

Con las clases de controlador y recurso preparadas, puedes acceder a los recursos usando una URL como `http://localhost/index.php?r=user/create`, parecida a la que usas con aplicaciones Web normales.

En la práctica, querrás usualmente usar URLs limpias y obtener ventajas de los verbos HTTP. Por ejemplo, una petición `POST /users` significaría acceder a la acción `user/create`. Esto puede realizarse fácilmente configurando el componente de la aplicación `urlManager` como sigue:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

En comparación con la gestión de URL en las aplicaciones Web, lo principalmente nuevo de lo anterior es el uso de `yii\rest\UrlRule` para el enrutamiento de las peticiones con el API RESTful. Esta clase especial de regla URL creará un conjunto completo de reglas URL hijas para soportar el enrutamiento y creación de URL para el/los controlador/es especificados. Por ejemplo, el código anterior es equivalente a las siguientes reglas:

```
[
    'PUT,PATCH users/<id>' => 'user/update',
    'DELETE users/<id>' => 'user/delete',
    'GET,HEAD users/<id>' => 'user/view',
    'POST users' => 'user/create',
    'GET,HEAD users' => 'user/index',
    'users/<id>' => 'user/options',
    'users' => 'user/options',
]
```

Y los siguientes puntos finales del API son mantenidos por esta regla:

- `GET /users`: lista de todos los usuarios página a página;
- `HEAD /users`: muestra Íla información resumen del usuario listado;
- `POST /users`: crea un nuevo usuario;
- `GET /users/123`: devuelve los detalles del usuario 123;
- `HEAD /users/123`: muestra la información resumen del usuario 123;
- `PATCH /users/123` y `PUT /users/123`: actualizan al usuario 123;
- `DELETE /users/123`: borra el usuario 123;
- `OPTIONS /users`: muestra los verbos soportados de acuerdo al punto final `/users`;
- `OPTIONS /users/123`: muestra los verbos soportados de acuerdo al punto final `/users/123`.

Puedes configurar las opciones `only` y `except` para explícitamente listar cuáles acciones soportar o cuáles deshabilitar, respectivamente. Por ejemplo,

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'except' => ['delete', 'create', 'update'],
],
```

También puedes configurar las propiedades `patterns` o `extraPatterns` para redefinir patrones existentes o añadir nuevos soportados por esta regla. Por ejemplo, para soportar una nueva acción `search` para el punto final `GET /users/search`, configura la opción `extraPatterns` como sigue,

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'extraPatterns' => [
        'GET search' => 'search',
    ],
]
```

Puedes haber notado que el ID del controlador `user` aparece en formato plural `users` en los puntos finales de las URLs. Esto se debe a que `yii\rest\UrlRule` automáticamente pluraliza los IDs de los controladores al crear reglas URL hijas. Puedes desactivar este comportamiento definiendo la propiedad `yii\rest\UrlRule::$pluralize` como `false`.

> **Información:** La pluralización de los IDs de los controladores es realizada por `yii\helpers\Inflector::pluralize()`. Este método respeta reglas especiales de pluralización. Por ejemplo, la palabra `box` (caja) será pluralizada como `boxes` en vez de `boxs`.

En caso de que la pluralización automática no encaje en tus requerimientos, puedes además configurar la propiedad `yii\rest\UrlRule::$controller` para especificar explícitamente cómo mapear un nombre utilizado en un punto final URL a un ID de controlador. Por ejemplo, el siguiente código mapea el nombre `u` al ID del controlador `user`.

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['u' => 'user'],
]
```

11.5. Formato de Respuesta

Cuando se maneja una petición al API RESTful, una aplicación realiza usualmente los siguientes pasos que están relacionados con el formato de la respuesta:

1. Determinar varios factores que pueden afectar al formato de la respuesta, como son el tipo de medio, lenguaje, versión, etc. Este proceso es también conocido como negociación de contenido (content negotiation)[3].

2. La conversión de objetos recurso en arrays, como está descrito en la sección Recursos (Resources). Esto es realizado por la clase `yii\rest\Serializer`.

3. La conversión de arrays en cadenas con el formato determinado por el paso de negociación de contenido. Esto es realizado por los `formatos de respuesta` registrados con la propiedad `formatters` del componente de la aplicación `response`.

[3] https://es.wikipedia.org/wiki/Negociaci%C3%B3n_de_contenido

11.5.1. Negociación de contenido (Content Negotiation)

Yii soporta la negociación de contenido a través del filtro `yii\filters\ContentNegotiator`. La clase de controlador base del API RESTful `yii\rest\Controller` está equipada con este filtro bajo el nombre `contentNegotiator`. El filtro provee tanto un formato de respuesta de negociación como una negociación de lenguaje. Por ejemplo, si la petición API RESTful contiene la siguiente cabecera,

```
Accept: application/json; q=1.0, */*; q=0.1
```

puede obtener una respuesta en formato JSON, como a continuación:

```
$ curl -i -H "Accept: application/json; q=1.0, */*; q=0.1"
"http://localhost/users"

HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
X-Powered-By: PHP/5.4.20
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

[
    {
        "id": 1,
        ...
    },
    {
        "id": 2,
        ...
    },
    ...
]
```

Detrás de escena, antes de que sea ejecutada una acción del controlador del API RESTful, el filtro `yii\filters\ContentNegotiator` comprobará la cabecera HTTP `Accept` de la petición y definirá el `response format` como `'json'`. Después de que la acción sea ejecutada y devuelva el objeto recurso o la colección resultante, `yii\rest\Serializer` convertirá el resultado en un array. Y finalmente, `yii\web\JsonResponseFormatter` serializará el array en una cadena JSON incluyéndola en el cuerpo de la respuesta.

Por defecto, el API RESTful soporta tanto el formato JSON como el XML. Para soportar un nuevo formato, debes configurar la propiedad `formats`

del filtro `contentNegotiator` tal y como sigue, en las clases del controlador del API:

```
use yii\web\Response;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['contentNegotiator']['formats']['text/html'] =
    Response::FORMAT_HTML;
    return $behaviors;
}
```

Las claves de la propiedad `formats` son los tipos MIME soportados, mientras que los valores son los nombres de formato de respuesta correspondientes, los cuales deben ser soportados en `yii\web\Response::$formatters`.

11.5.2. Serialización de Datos

Como hemos descrito antes, `yii\rest\Serializer` es la pieza central responsable de convertir objetos recurso o colecciones en arrays. Reconoce objetos tanto implementando `yii\base\ArrayableInterface` como `yii\data\DataProviderInterface`. El primer formateador es implementado principalmente para objetos recursos, mientras que el segundo para recursos colección.

Puedes configurar el serializador definiendo la propiedad `yii\rest\Controller::$serializer` con un array de configuración. Por ejemplo, a veces puedes querer ayudar a simplificar el trabajo de desarrollo del cliente incluyendo información de la paginación directamente en el cuerpo de la respuesta. Para hacer esto, configura la propiedad `yii\rest\Serializer::$collectionEnvelope` como sigue:

```
use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
    public $serializer = [
        'class' => 'yii\rest\Serializer',
        'collectionEnvelope' => 'items',
    ];
}
```

Puedes obtener la respuesta que sigue para la petición `http://localhost/users`:

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
X-Powered-By: PHP/5.4.20
```

```
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "items": [
        {
            "id": 1,
            ...
        },
        {
            "id": 2,
            ...
        },
        ...
    ],
    "_links": {
        "self": {
            "href": "http://localhost/users?page=1"
        },
        "next": {
            "href": "http://localhost/users?page=2"
        },
        "last": {
            "href": "http://localhost/users?page=50"
        }
    },
    "_meta": {
        "totalCount": 1000,
        "pageCount": 50,
        "currentPage": 1,
        "perPage": 20
    }
}
```

11.6. Autenticación

A diferencia de las aplicaciones Web, las API RESTful son usualmente sin estado (stateless), lo que permite que las sesiones o las cookies no sean usadas. Por lo tanto, cada petición debe llevar alguna suerte de credenciales de autenticación, porque la autenticación del usuario no puede ser mantenida por las sesiones o las cookies. Una práctica común es enviar una pieza (token) secreta de acceso con cada petición para autenticar al usuario. Dado que una pieza de autenticación puede ser usada para identificar y autenticar

solamente a un usuario, **la API de peticiones tiene que ser siempre enviado vía HTTPS para prevenir ataques tipo "man-in-the-middle" (MitM)** .

Hay muchas maneras de enviar una token (pieza) de acceso:

- Autenticación Básica HTTP[4]: la pieza de acceso es enviada como nombre de usuario. Esto sólo debe de ser usado cuando la pieza de acceso puede ser guardada de forma segura en la parte del API del consumidor. Por ejemplo, el API del consumidor es un programa ejecutándose en un servidor.
- Parámetro de la consulta: la pieza de acceso es enviada como un parámetro de la consulta en la URL de la API, p.e., `https://example.com/users?access-token=xxxxxxxx`. Debido que muchos servidores dejan los parámetros de consulta en los logs del servidor, esta aproximación suele ser usada principalmente para servir peticiones `JSONP` que no usen las cabeceras HTTP para enviar piezas de acceso.
- OAuth 2[5]: la pieza de acceso es obtenida por el consumidor por medio de una autorización del servidor y enviada al API del servidor según el protocolo OAuth 2 tokens HTTP del portador[6].

Yii soporta todos los métodos anteriores de autenticación. Puedes crear nuevos métodos de autenticación de una forma fácil.

Para activar la autenticación para tus APIs, sigue los pasos siguientes:

1. Configura el componente `user` de la aplicación:
   - Define la propiedad `enableSession` como `false`.
   - Define la propiedad `loginUrl` como `null` para mostrar un error HTTP 403 en vez de redireccionar a la pantalla de login.
2. Especifica cuál método de autenticación planeas usar configurando el comportamiento (behavior) `authenticator` en tus clases de controladores REST.
3. Implementa `yii\web\IdentityInterface::findIdentityByAccessToken()` en tu `clase de identidad de usuarios`.

El paso 1 no es necesario pero sí recomendable para las APIs RESTful, pues son sin estado (stateless). Cuando `enableSession` es `false`, el estado de autenticación del usuario puede NO persistir entre peticiones usando sesiones. Si embargo, la autenticación será realizada para cada petición, lo que se consigue en los pasos 2 y 3.

> Tip:Puedes configurar `enableSession` del componente de la aplicación `user` en la configuración de las aplicaciones si estás desarrollando APIs RESTful en términos de un aplicación. Si desarrollas

[4] https://es.wikipedia.org/wiki/Autenticaci%C3%B3n_de_acceso_b%C3%A1sica
[5] https://oauth.net/2/
[6] https://datatracker.ietf.org/doc/html/rfc6750

un módulo de las APIs RESTful, puedes poner la siguiente línea en el método del módulo `init()`, tal y como sigue:

```
public function init()
{
    parent::init();
    \Yii::$app->user->enableSession = false;
}
```

Por ejemplo, para usar HTTP Basic Auth, puedes configurar el comportamiento (behavior) `authenticator` como sigue,

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::class,
    ];
    return $behaviors;
}
```

Si quieres implementar los tres métodos de autenticación explicados antes, puedes usar `CompositeAuth` de la siguiente manera,

```
use yii\filters\auth\CompositeAuth;
use yii\filters\auth\HttpBasicAuth;
use yii\filters\auth\HttpBearerAuth;
use yii\filters\auth\QueryParamAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => CompositeAuth::class,
        'authMethods' => [
            HttpBasicAuth::class,
            HttpBearerAuth::class,
            QueryParamAuth::class,
        ],
    ];
    return $behaviors;
}
```

Cada elemento en `authMethods` debe de ser el nombre de una clase de método de autenticación o un array de configuración.

La implementación de `findIdentityByAccessToken()` es específico de la aplicación. Por ejemplo, en escenarios simples cuando cada usuario sólo puede tener un token de acceso, puedes almacenar este token en la columna `access_token` en la tabla de usuario. El método debe de ser inmediatamente implementado en la clase `User` como sigue,

```php
use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }
}
```

Después que la autenticación es activada, tal y como se describe arriba, para cada petición de la API, el controlador solicitado puede intentar autenticar al usuario en su evento `beforeAction()`.

Si la autenticación tiene éxito, el controlador realizará otras comprobaciones (como son límite del ratio, autorización) y entonces ejecutar la acción. La identidad del usuario autenticado puede ser recuperada via `Yii::$app->user->identity`.

Si la autenticación falla, una respuesta con estado HTTP 401 será devuelta junto con otras cabeceras apropiadas (tal como la cabecera para autenticación básica HTTP `WWW-Authenticate`).

11.6.1. Autorización

Después de que un usuario se ha autenticado, probablementer querrás comprobar si él o ella tiene los permisos para realizar la acción solicitada. Este proceso es llamado *autorización (authorization)* y está cubierto en detalle en la Sección de Autorización.

Si tus controladores extienden de `yii\rest\ActiveController`, puedes sobreescribir el método `checkAccess()` para realizar la comprobación de la autorización. El método será llamado por las acciones contenidas en `yii\rest\ActiveController`.

11.7. Limitando el rango (rate)

Para prevenir el abuso, puedes considerar añadir un *límitación del rango (rate limiting)* para tus APIs. Por ejemplo, puedes querer limitar el uso del API de cada usuario a un máximo de 100 llamadas al API dentro de un periodo de 10 minutos. Si se reciben demasiadas peticiones de un usuario dentro del periodo de tiempo declarado, deveríá devolverse una respuesta con código de estado 429 (que significa "Demasiadas peticiones").

Para activar la limitación de rango, la clase `user identity class` debe implementar `yii\filters\RateLimitInterface`. Este interface requiere la implementación de tres métodos:

- `getRateLimit()`: devuelve el número máximo de peticiones permitidas y el periodo de tiempo (p.e., `[100, 600]` significa que como mucho puede haber 100 llamadas al API dentro de 600 segundos).

- `loadAllowance()`: devuelve el número de peticiones que quedan permitidas y el tiempo (fecha/hora) UNIX con el último límite del rango que ha sido comprobado.
- `saveAllowance()`: guarda ambos, el número restante de peticiones permitidas y el tiempo actual (fecha/hora) UNIX .

Puedes usar dos columnas en la tabla de usuario para guardar la información de lo permitido y la fecha/hora (timestamp). Con ambas definidas, entonces `loadAllowance()` y `saveAllowance()` pueden ser utilizados para leer y guardar los valores de las dos columnas correspondientes al actual usuario autenticado. Para mejorar el desempeño, también puedes considerar almacenar esas piezas de información en caché o almacenamiento NoSQL.

Una vez que la clase de identidad implementa la interfaz requerida, Yii utilizará automáticamente `yii\filters\RateLimiter` configurado como un filtro de acción para que `yii\rest\Controller` compruebe el límite de rango. El limitador de rango lanzará una excepeción `yii\web\TooManyRequestsHttpException` cuando el límite del rango sea excedido.

Puedes configurar el limitador de rango en tu clase controlador REST como sigue:

```
public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['rateLimiter']['enableRateLimitHeaders'] = false;
    return $behaviors;
}
```

Cuando se activa el límite de rango, por defecto todas las respuestas serán enviadas con la siguiente cabecera HTTP conteniendo información sobre el límite actual de rango:

- `X-Rate-Limit-Limit`, el máximo número de peticiones permitidas en un periodo de tiempo
- `X-Rate-Limit-Remaining`, el número de peticiones restantes en el periodo de tiempo actual
- `X-Rate-Limit-Reset`, el número de segundos a esperar para pedir el máximo número de peticiones permitidas

Puedes desactivar estas cabeceras configurando `yii\filters\RateLimiter::$enableRateLimitHeaders` a `false`, tal y como en el anterior ejemplo.

11.8. Versionado

Una buena API ha de ser *versionada*: los cambios y las nuevas características son implementadas en las nuevas versiones del API, en vez de estar continuamente modificando sólo una versión. Al contrario que en las aplicaciones Web, en las cuales tienes total control del código de ambas partes lado del cliente y lado del servidor, las APIs están destinadas a ser usadas por

los clientes fuera de tu control. Por esta razón, compatibilidad hacia atrás (BC Backward compatibility) de las APIs ha de ser mantenida siempre que sea posible. Si es necesario un cambio que puede romper la BC, debes de introducirla en la nueva versión del API, e incrementar el número de versión. Los clientes que la usan pueden continuar usando la antigua versión de trabajo del API; los nuevos y actualizados clientes pueden obtener la nueva funcionalidad de la nueva versión del API.

> **Consejo:** referirse a Semántica del versionado[7] para más información en el diseño del número de versión del API.

Una manera común de implementar el versionado de la API es embeber el número de versión en las URLs de la API. Por ejemplo, `https://example.com/v1/users` se refiere al punto final `/users` de la versión 1 de la API.

Otro método de versionado de la API, la cual está ganando predominancia recientemente, es poner el número de versión en las cabeceras de la petición HTTP. Esto se suele hacer típicamente a través la cabecera `Accept`:

```
// vía parámetros
Accept: application/json; version=v1
// vía de el tipo de contenido del proveedor
Accept: application/vnd.company.myapp-v1+json
```

Ambos métodos tienen sus pros y sus contras, y hay gran cantidad de debates sobre cada uno. Debajo puedes ver una estrategia práctica para el versionado de la API que es una mezcla de estos dos métodos:

- Pon cada versión superior de la implementación de la API en un módulo separado cuyo ID es el número de la versión mayor (p.e. `v1`, `v2`). Naturalmente, las URLs de la API contendrán números de versión mayores.
- Dentro de cada versión mayor (y por lo tanto, dentro del correspondiente módulo), usa la cabecera de HTTP `Accept` para determinar el número de la versión menor y escribe código condicional para responder a la menor versión como corresponde.

Para cada módulo sirviendo una versión mayor, el módulo debe incluir las clases de recursos y y controladores que especifican la versión. Para separar mejor la responsabilidad del código, puedes conservar un conjunto común de clases base de recursos y controladores, y hacer subclases de ellas en cada versión individual del módulo. Dentro de las subclases, impementa el código concreto como por ejemplo `Model::fields()`.

Tu código puede estar organizado como lo que sigue:

```
api/
    common/
        controllers/
```

[7] https://semver.org/

```
            UserController.php
            PostController.php
        models/
            User.php
            Post.php
    modules/
        v1/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
                Post.php
        v2/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
                Post.php
```

La configuración de tu aplicación puede tener este aspecto:

```
return [
    'modules' => [
        'v1' => [
            'basePath' => '@app/modules/v1',
            'controllerNamespace' => 'app\modules\v1\controllers',
        ],
        'v2' => [
            'basePath' => '@app/modules/v2',
            'controllerNamespace' => 'app\modules\v2\controllers',
        ],
    ],
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'enableStrictParsing' => true,
            'showScriptName' => false,
            'rules' => [
                ['class' => 'yii\rest\UrlRule', 'controller' => ['v1/user',
                'v1/post']],
                ['class' => 'yii\rest\UrlRule', 'controller' => ['v2/user',
                'v2/post']],
            ],
        ],
    ],
];
```

Como consecuencia del código anterior, `https://example.com/v1/users` devolverá la lista de usuarios en la versión 1, mientras `https://example.com/v2/users` devolverá la versión 2 de los usuarios.

Gracias a los módulos, el código de las diferentes principales versiones puede ser aislado. Pero los módulos hacen posible reutilizar el código a través de los módulos vía clases base comunes y otros recursos compartidos.

Para tratar con versiones menores, puedes tomar ventaja de la característica de negociación de contenido provista por el comportamiento (behavior) `contentNegotiator`. El comportamiento `contentNegotiator` definirá la propiedad `yii\web\Response::$acceptParams` cuando determina qué tipo de contenido soportar.

Por ejemplo, si una petición es enviada con la cabecera HTTP `Accept: application/json; version=v1`, después de la negociación de contenido, `yii\web\Response::$acceptParams` contendrá el valor `['version' => 'v1']`.

Basado en la información de versión contenida en `acceptParams`, puedes escribir código condicional en lugares como acciones, clases de recursos, serializadores, etc. para proveer la funcionalidad apropiada.

Dado que por definición las versiones menores requireren mantener la compatibilidad hacia atrás, con suerte no tendrás demasiadas comprobaciones de versión en tu código. De otra manera, probablemente puede ocurrir que necesites crear una versión mayor.

11.9. Manejo de errores

Cuando se maneja una petición de API RESTful, si ocurre un error en la petición del usuario o si algo inesperado ocurre en el servidor, simplemente puedes lanzar una excepción para notificar al usuario que algo erróneo ocurrió. Si puedes identificar la causa del error (p.e., el recurso solicitado no existe), debes considerar lanzar una excepción con el código HTTP de estado apropiado (p.e., `yii\web\NotFoundHttpException` representa un código de estado 404). Yii enviará la respuesta a continuación con el correspondiente código de estado HTTP y el texto. Yii puede incluir también la representación serializada de la excepción en el cuerpo de la respuesta. Por ejemplo:

```
HTTP/1.1 404 Not Found
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
    "code": 0,
    "status": 404
}
```

La siguiente lista sumariza los códigos de estado HTTP que son usados por el framework REST:

- `200`: OK. Todo ha funcionado como se esperaba.
- `201`: El recurso ha creado con éxito en respuesta a la petición `POST`. La cabecera de situación `Location` contiene la URL apuntando al nuevo recurso creado.
- `204`: La petición ha sido manejada con éxito y el cuerpo de la respuesta no tiene contenido (como una petición `DELETE`).
- `304`: El recurso no ha sido modificado. Puede usar la versión en caché.
- `400`: Petición errónea. Esto puede estar causado por varias acciones de el usuario, como proveer un JSON no válido en el cuerpo de la petición, proveyendo parámetros de acción no válidos, etc.
- `401`: Autenticación fallida.
- `403`: El usuario autenticado no tiene permitido acceder a la API final.
- `404`: El recurso pedido no existe.
- `405`: Método no permitido. Por favor comprueba la cabecera `Allow` por los métodos HTTP permitidos.
- `415`: Tipo de medio no soportado. El tipo de contenido pedido o el número de versión no es válido.
- `422`: La validación de datos ha fallado (en respuesta a una petición `POST` , por ejemplo). Por favor, comprueba en el cuerpo de la respuesta el mensaje detallado.
- `429`: Demasiadas peticiones. La petición ha sido rechazada debido a un limitación de rango.
- `500`: Error interno del servidor. Esto puede estar causado por errores internos del programa.

11.9.1. Personalizar la Respuesta al Error

A veces puedes querer personalizar el formato de la respuesta del error por defecto . Por ejemplo, en lugar de depender del uso de diferentes estados HTTP para indicar los diferentes errores, puedes querer usar siempre el estado HTTP 200 y encapsular el código de estado HTTP real como parte de una estructura JSON en la respuesta, como se muestra a continuación,

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "success": false,
    "data": {
        "name": "Not Found Exception",
        "message": "The requested resource was not found.",
        "code": 0,
        "status": 404
```

```
    }
}
```

Para lograrlo, puedes responder al evento `beforeSend` del componente `response` en la configuración de la aplicación:

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null &&
                Yii::$app->request->get('suppress_response_code')) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
    ],
];
```

El anterior código reformateará la respuesta (sea exitosa o fallida) como se explicó cuando `suppress_response_code` es pasado como un parámetro `GET`.

Capítulo 12

Herramientas de Desarrollo

Error: not existing file: tool-gii.md

Error: not existing file: tool-api-doc.md

Capítulo 13

Pruebas

13.1. Tests

Las pruebas son una parte importante del desarrollo de software. Seamos conscientes de ello o no, ralizamos pruebas contínuamente. Por ejemplo, cuando escribimos una clase en PHP, podemos depurarla paso a paso o simplemente usar declaraciones `echo` o `die` para verificar que la implementación funciona conforme a nuestro plan inicial. En el caso de una aplicación web, introducimos algunos datos de prueba en los formularios para asegurarnos de que la página interactúa con nosotros como esperábamos.

El proceso de testeo se puede automatizar para que cada vez que necesitemos verificar algo, solamente necesitemos invocar el código que lo hace por nosotros. El código que verifica que el restulado coincide con lo que habíamos planeado se llama *test* y el proceso de su creación y posterior ejecución es conocido como *testeo automatizado*, que es el principal tema de estos capítulos sobre testeo.

13.1.1. Desarrollo con tests

El Desarrollo Dirigido por Pruebas (*Test-Driven Development* o TDD) y el Desarrollo Dirigido por Corpotamientos (*Behavior-Driven Development* o BDD) son enfoques para desarrollar software, en los que se describe el comportamiento de un trozo de código o de toda la funcionalidad como un conjunto de escenarios o pruebas antes de escribir el código real y sólo entonces crear la implementación que permite pasar esos tests verificando que se ha logrado el comportamiento pretendido.

El proceso de desarrollo de una funcionalidad es el siguiente:

- Crear un nuevo test que describe una funcionalidad a implementar.
- Ejecutar el nuevo test y asegurarse de que falla. Esto es lo esperado, dado que todavía no hay ninguna implementación.
- Escribir un código sencillo para superar el nuevo test.
- Ejecutar todos los tests y asegurarse de que se pasan todos.

- Mejorar el código y asegurarse de que los tests siguen superándose.

Una vez hecho, se repite el proceso de neuvo para otra funcionalidad o mejora. Si se va a cambiar la funcionalidad existente, también hay que cambiar los tests.

> **Consejo:** Si siente que está perdiendo tiempo haciendo un montón de iteraciones pequeñas y simples, intente cubrir más por cada escenario de test, de modo que haga más cosas antes de ejecutar los tests de nuevo. Si está depurando demasiado, intente hacer lo contrario.

La razón para crear los tests antes de hacer ninguna implementación es que eso nos permite centrarnos en lo que queremos alcanzar y sumergirnos totalmente en «cómo hacerlo» después. Normalmente conduce a mejores abstracciones y a un más fácil mantenimiento de los tests cuando toque hacer ajustes a las funcionalidades o componentes menos acoplados.

Para resumir, las ventajas de este enfoque son las siguientes:

- Le mantiene centrado en una sola cosa en cada momento, lo que resulta en una mejor planificación e implementación.
- Resulta en más funcionalidades cubiertas por tests, y en mayor detalle. Es decir, si se superan los tests, lo más problable es que no haya nada roto.

A largo plazo normalmente tiene como efecto un buen ahorro de tiempo.

13.1.2. Qué y cómo probar

Aunque el enfoque de primero los tests descrito arriba tiene sentido para el largo plazo y proyectos relativamente complejos, sería excesivo para proyectos más simples. Hay algunas indicaciones de cuándo es apropiado:

- El proyecto ya es grande y complejo.
- Los requisitos del proyecto están empezando a hacerse complejos. El proyecto crece constantemente.
- El proyecto pretende a ser a largo plazo.
- El coste de fallar es demasiado alto.

No hay nada malo en crear tests que cubran el comportamiento de una implementación existente.

- Es un proyecto legado que se va a renovar gradualmente.
- Le han dado un proyecto sobre el que trabajar y no tiene tests.

En algunos casos cualquier forma de testo automatizado sería exagerada:

- El proyecto es sencillo y no se va a volver más complejo.
- Es un proyecto puntual en el que no se seguirá trabajando.

De todas formas, si dispone de tiempo, es bueno automatizar las pruebas también en esos casos.

13.1.3. Más lecturas

- Test Driven Development: By Example / Kent Beck. ISBN: 0321146530.

13.2. Preparación del entorno de pruebas

Yii 2 ha mantenido oficialmente integración con el *framework* de testeo `Codeception`[1], que le permite crear los siguientes tipos de tests:

- Unitarias - verifica que una unidad simple de código funciona como se espera;
- Funcional - verifica escenarios desde la perspectiva de un usuario a través de la emulación de un navegador;
- De aceptación - verifica escenarios desde la perspectiva de un usuario en un navegador.

Yii provee grupos de pruebas listos para utilizar para los tres tipos de test, tanto en la plantilla de proyecto `yii2-basic`[2] como en `yii2-advanced`[3].

Codeception viene preinstalado tanto en la plantilla de proyecto básica como en la avanzada. En caso de que no use una de estas plantillas, puede instalar Codeception ejecutando las siguientes órdenes de consola:

```
composer require codeception/codeception
composer require codeception/specify
composer require codeception/verify
```

13.3. Pruebas unitarias

Un test unitario se encarga de verificar que una unidad simple de código funcione como se espera. Esto decir, dados diferentes parámetros de entrada, el test verifica que el método de la clase devuelve el resultado esperado. Normalmente los tests unitarios son desarrollados por la persona que escribe las clases testeadas.

Los tests unitarios en Yii están construidos en base a PHPUnit y, opcionalmente, Codeception, por lo que se recomienda consultar su respectiva documentación:

- Codeception para el *framework* Yii[4]
- Tests unitarios con Codeception[5]
- Documentación de PHPUnit, empezando por el capítulo 2[6]

[1] https://github.com/Codeception/Codeception
[2] https://github.com/yiisoft/yii2-app-basic
[3] https://github.com/yiisoft/yii2-app-advanced
[4] https://codeception.com/for/yii
[5] https://codeception.com/docs/05-UnitTests
[6] https://phpunit.de/manual/current/en/writing-tests-for-phpunit.html

13.3.1. Ejecución de tests en las plantillas básica y avanzada

Si ha empezado con la plantilla avanzada, consulte la guía de testeo[7] para más detalles sobre la ejecución de tests.

Si ha empezado con la plantilla básica, consulte la sección sobre testeo de su README[8].

13.3.2. Tests unitarios del framework

Si desea ejecutar tests unitarios para el framework Yii en sí, consulte «Comenzando con el desarrollo de Yii 2[9]».

13.4. Tests funcionales

Los tests funcionales verifican escenarios desde la perspectiva de un usuario. Son similares a los tests de aceptación pero en lugar de comunicarse vía HTTP rellena el entorno como parámetros POST y GET y después ejecuta una instancia de la aplicación directamente desde el código.

Los tests funcionales son generalmente más rápidos que los tests de aceptación y proporcionan *stack traces* detalladas en los fallos. Como regla general, debería preferirlos salvo que tenga una configuración de servidor web especial o una interfaz de usuario compleja en Javascript.

Las pruebas funcionales se implementan con ayuda del *framework* Codeception, que tiene una buena documentación:

- Codeception para el *framework* Yii[10]
- Tests funcionales de Codeception[11]

13.4.1. Ejecución de tests en las plantillas básica y avanzada

Si ha empezado con la plantilla avanzada, consulte la guía de testeo[12] para más detalles sobre la ejecución de tests.

Si ha empezado con la plantilla básica, consulte la sección sobre testeo de su README[13].

---

[7] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/start-testing.md

[8] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing

[9] https://github.com/yiisoft/yii2/blob/master/docs/internals/getting-started.md

[10] https://codeception.com/for/yii

[11] https://codeception.com/docs/04-FunctionalTests

[12] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/start-testing.md

[13] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing

13.5. Tests de aceptación

Un test de aceptación verifica escenarios desde la perspectiva de un usuario. Se accede a la aplicación testeada por medio de PhpBrowser o de un navegador de verdad. En ambos casos los navegadores se comunican vía HTTP así que la aplicación debe ser servida por un servidor web.

Los tests de aceptación se implementan con ayuda del *framework* Codeception, que tiene una buena documentación:

- Codeception para el *framework* Yii[14]
- Tests funcionales de Codeception[15]

13.5.1. Ejecución de tests en las plantillas básica y avanzada

Si ha empezado con la plantilla avanzada, consulte la guía de testeo[16] para más detalles sobre la ejecución de tests.

Si ha empezado con la plantilla básica, consulte la sección sobre testeo de su README[17].

13.6. Fixtures

Los fixtures son una parte importante de los tests. Su propósito principal es el de preparar el entorno en una estado fijado/conocido de manera que los tests sean repetibles y corran de la manera esperada. Yii provee un framework de fixtures que te permite dichos fixtures de manera precisa y usarlo de forma simple.

Un concepto clave en el framework de fixtures de Yii es el llamado *objeto fixture*. Un objeto fixture representa un aspecto particular de un entorno de pruebas y es una instancia de `yii\test\Fixture` o heredada de esta. Por ejemplo, puedes utilizar `UserFixture` para asegurarte de que la tabla de usuarios de la BD contiene un grupo de datos fijos. Entonces cargas uno o varios objetos fixture antes de correr un test y lo descargas cuando el test ha concluido.

Un fixture puede depender de otros fixtures, especificándolo en su propiedad `yii\test\Fixture::$depends`. Cuando un fixture está siendo cargado, los fixtures de los que depende serán cargados automáticamente ANTES que él; y cuando el fixture está siendo descargado, los fixtures dependientes serán descargados DESPUÉS de él.

[14] https://codeception.com/for/yii

[15] https://codeception.com/docs/04-FunctionalTests

[16] https://github.com/yiisoft/yii2-app-advanced/blob/master/docs/guide/start-testing.md

[17] https://github.com/yiisoft/yii2-app-basic/blob/master/README.md#testing

13.6.1. Definir un Fixture

Para definir un fixture, crea una nueva clase que extienda de `yii\test\Fixture` o `yii\test\ActiveFixture`. El primero es más adecuado para fixtures de propósito general, mientras que el último tiene características mejoradas específicamente diseñadas para trabajar con base de datos y ActiveRecord.

El siguiente código define un fixture acerca del ActiveRecord `User` y su correspondiente tabla user.

```
<?php
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserFixture extends ActiveFixture
{
    public $modelClass = 'app\models\User';
}
```

> **Consejo:** Cada `ActiveFixture` se encarga de preparar la tabla de la DB para los tests. Puedes especificar la tabla definiendo tanto la propiedad `yii\test\ActiveFixture::$tableName` o la propiedad `yii\test\ActiveFixture::$modelClass`. Haciéndolo como el último, el nombre de la tabla será tomado de la clase `ActiveRecord` especificada en `modelClass`.
>
> **Nota:** `yii\test\ActiveFixture` es sólo adecualdo para bases de datos SQL. Para bases de datos NoSQL, Yii provee las siguientes clases `ActiveFixture`:
>
> - Mongo DB: `yii\mongodb\ActiveFixture`
> - Elasticsearch: `yii\elasticsearch\ActiveFixture` (desde la versión 2.0.2)

Los datos para un fixture `ActiveFixture` son usualmente provistos en un archivo ubicado en `FixturePath/data/TableName.php`, donde `FixturePath` corresponde al directorio conteniendo el archivo de clase del fixture, y `TableName` es el nombre de la tabla asociada al fixture. En el ejemplo anterior, el archivo debería ser `@app/tests/fixtures/data/user.php`. El archivo de datos debe devolver un array de registros a ser insertados en la tabla user. Por ejemplo,

```
<?php
return [
    'user1' => [
        'username' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
```

```
        'password' =>
        '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/iK0r3jRuwQEs2ldRu.a2',
    ],
    'user2' => [
        'username' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphhOh9q',
        'password' =>
        '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6viYG5xJExU6',
    ],
];
```

Puedes dar un alias al registro tal que más tarde en tu test, puedas referirte a ese registra a través de dicho alias. En el ejemplo anterior, los dos registros tienen como alias `user1` y `user2`, respectivamente.

Además, no necesitas especificar los datos de columnas auto-incrementales. Yii automáticamente llenará esos valores dentro de los registros cuando el fixture está siendo cargado.

> **Consejo:** Puedes personalizar la ubicación del archivo de datos definiendo la propiedad `yii\test\ActiveFixture::$dataFile`. Puedes también sobrescribir `yii\test\ActiveFixture::getData()` para obtener los datos.

Como se describió anteriormente, un fixture puede depender de otros fixtures. Por ejemplo, un `UserProfileFixture` puede necesitar depender de `UserFixture` porque la table de perfiles de usuarios contiene una clave foránea a la tabla user. La dependencia es especificada vía la propiedad `yii\test\Fixture::$depends`, como a continuación,

```
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserProfileFixture extends ActiveFixture
{
    public $modelClass = 'app\models\UserProfile';
    public $depends = ['app\tests\fixtures\UserFixture'];
}
```

La dependencia también asegura que los fixtures son cargados y descargados en un orden bien definido. En el ejemplo `UserFixture` será siempre cargado antes de `UserProfileFixture` para asegurar que todas las referencias de las claves foráneas existan y será siempre descargado después de `UserProfileFixture` por la misma razón.

Arriba te mostramos cómo definir un fixture de BD. Para definir un fixture no relacionado a BD (por ej. un fixture acerca de archivos y directorios), puedes extender de la clase base más general `yii\test\Fixture` y sobrescribir los métodos `load()` y `unload()`.

13.6.2. Utilizar Fixtures

Si estás utilizando Codeception[18] para hacer tests de tu código, deberías considerar el utilizar la extensión `yii2-codeception`, que tiene soporte incorporado para la carga y acceso a fixtures. En caso de que utilices otros frameworks de testing, puedes usar `yii\test\FixtureTrait` en tus casos de tests para alcanzar el mismo objetivo.

A continuación describiremos cómo escribir una clase de test de unidad `UserProfile` utilizando `yii2-codeception`.

En tu clase de test de unidad que extiende de `yii\codeception\DbTestCase` o `yii\codeception\TestCase`, indica cuáles fixtures quieres utilizar en el método `fixtures()`. Por ejemplo,

```
namespace app\tests\unit\models;

use yii\codeception\DbTestCase;
use app\tests\fixtures\UserProfileFixture;

class UserProfileTest extends DbTestCase
{
    public function fixtures()
    {
        return [
            'profiles' => UserProfileFixture::class,
        ];
    }

    // ...métodos de test...
}
```

Los fixtures listados en el método `fixtures()` serán automáticamente cargados antes de correr cada método de test en el caso de test y descargado al finalizar cada uno. También, como describimos antes, cuando un fixture está siendo cargado, todos sus fixtures dependientes serán cargados primero. En el ejemplo de arriba, debido a que `UserProfileFixture` depende de `UserFixture`, cuando ejecutas cualquier método de test en la clase, dos fixtures serán cargados secuencialmente: `UserFixture` y `UserProfileFixture`.

Al especificar fixtures en `fixtures()`, puedes utilizar tanto un nombre de clase o un array de configuración para referirte a un fixture. El array de configuración te permitirá personalizar las propiedades del fixture cuando este es cargado.

Puedes también asignarles alias a los fixtures. En el ejemplo anterior, el `UserProfileFixture` tiene como alias `profiles`. En los métodos de test, puedes acceder a un objeto fixture utilizando su alias. Por ejemplo, `$this->profiles` devolverá el objeto `UserProfileFixture`.

Dado que `UserProfileFixture` extiende de `ActiveFixture`, puedes por lo tanto usar la siguiente sintáxis para acceder a los datos provistos por el fixture:

[18] https://codeception.com/

```
// devuelve el registro del fixture cuyo alias es 'user1'
$row = $this->profiles['user1'];
// devuelve el modelo UserProfile correspondiente al registro cuyo alias es
'user1'
$profile = $this->profiles('user1');
// recorre cada registro en el fixture
foreach ($this->profiles as $row) ...
```

> **Información:** `$this->profiles` es todavía del tipo `UserProfileFixture`. Las características de acceso mostradas arriba son implementadas a través de métodos mágicos de PHP.

13.6.3. Definir y Utilizar Fixtures Globales

Los fixtures descritos arriba son principalmente utilizados para casos de tests individuales. En la mayoría de los casos, puedes necesitar algunos fixtures globales que sean aplicados a TODOS o muchos casos de test. Un ejemplo sería `yii\test\InitDbFixture`, que hace dos cosas:

- Realiza alguna tarea de inicialización común al ejectutar un script ubicado en `@app/tests/fixtures/initdb.php`;
- Deshabilita la comprobación de integridad antes de cargar otros fixtures de BD, y la rehabilita después de que todos los fixtures son descargados.

Utilizar fixtures globales es similar a utilizar los no-globales. La única diferencia es que declaras estos fixtures en `yii\codeception\TestCase::globalFixtures()` en vez de en `fixtures()`. Cuando un caso de test carga fixtures, primero carga los globales y luego los no-globales.

Por defecto, `yii\codeception\DbTestCase` ya declara `InitDbFixture` en su método `globalFixtures()`. Esto significa que sólo necesitas trabajar con `@app/tests/fixtures/initdb.php` si quieres realizar algún trabajo de inicialización antes de cada test. Sino puedes simplemente enfocarte en desarrollar cada caso de test individual y sus fixtures correspondientes.

13.6.4. Organizar Clases de Fixtures y Archivos de Datos

Por defecto, las clases de fixtures busca los archivos de datos correspondientes dentro de la carpeta `data`, que es una subcarpeta de la carpeta conteniendo los archivos de clases de fixtures. Puedes seguir esta convención al trabajar en proyectos simples. Para proyectos más grandes, es probable que a menudo necesites intercambiar entre diferentes archivos de datos para la misma clase de fixture en diferentes tests. Recomendamos que organices los archivos de datos en forma jerárquica similar a tus espacios de nombre de clases. Por ejemplo,

```
# bajo la carpeta tests\unit\fixtures
```

```
data\
    components\
        fixture_data_file1.php
        fixture_data_file2.php
        ...
        fixture_data_fileN.php
    models\
        fixture_data_file1.php
        fixture_data_file2.php
        ...
        fixture_data_fileN.php
# y así sucesivamente
```

De esta manera evitarás la colisión de archivos de datos de fixtures entre tests y podrás utlilizarlos como necesites.

> **Nota:** En el ejemplo de arriba los archivos de fixtures son nombrados así sólo como ejemplo. En la vida real deberías nombrarlos de acuerdo a qué clase de fixture extienden tus clases de fixtures. Por ejemplo, si estás extendiendo de `yii\test\ActiveFixture` para fixtures de BD, deberías utilizar nombres de tabla de la BD como nombres de los archivos de fixtures; Si estás extendiendo de `yii\mongodb\ActiveFixture` para fixtures de MongoDB, deberías utilizar nombres de colecciones para los nombres de archivo.

Se puede utilizar una jerarquía similar para organizar archivos de clases de fixtures. En vez de utilizar `data` como directorio raíz, podrías querer utilizar `fixtures` como directorio raíz para evitar conflictos con los archivos de datos.

13.6.5. Resumen

> **Nota:** Esta sección se encuentra en desarrollo.

Arriba, definimos cómo definir y utilizar fixtures. Abajo resumiremos el típico flujo de trabajo de correr tests de unidad relacionados a BD:

1. Usa la herramienta `yii migrate` para actualizar tu base de datos de prueba a la última versión;
2. Corre el caso de test:
   - Carga los fixtures: limpia las tablas de la BD relevantes y cargala con los datos de los fixtures;
   - Realiza el test en sí;
   - Descarga los fixtures.
3. Repite el Paso 2 hasta que todos los tests terminen.

Lo siguiente, a ser limpiado

13.7. Administrar Fixtures

> **Nota:** Esta sección está en desarrollo.
>
> todo: este tutorial podría ser unificado con la parte de arriba en test-fixtures.md

Los fixtures son una parte importante del testing. Su principal propósito es el de poblarte con datos necesarios para el test de diferentes casos. Con estos datos. utilizar tests se vuelve más eficiente y útil.

Yii soporta fixtures a través de la herramienta de línea de comandos `yii fixture`. Esta herramienta soporta:

- Cargar fixtures a diferentes almacenamientos: RDBMS, NoSQL, etc;
- Descargar fixtures de diferentes maneras (usualmente limpiando el almacenamiento);
- Auto-generar fixtures y poblarlos con datos al azar.

13.7.1. Formato de Fixtures

Los fixtures son objetos con diferentes métodos y configuraciones, inspecciónalos en la documentación oficial[19]. Asumamos que tenemos datos de fixtures a cargar:

```
#archivo users.php bajo la ruta de los fixtures, por defecto
@tests\unit\fixtures\data

return [
    [
        'name' => 'Chase',
        'login' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
        'password' =>
        '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/iK0r3jRuwQEs2ldRu.a2',
    ],
    [
        'name' => 'Celestine',
        'login' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphhOh9q',
        'password' =>
        '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6viYG5xJExU6',
    ],
];
```

Si estamos utilizando un fixture que carga datos en la base de datos, entonces esos registros serán insertados en la tabla `users`. Si estamos utilizando fixtures

[19] https://github.com/yiisoft/yii2/blob/master/docs/guide-es/test-fixtures.md

no sql, por ejemplo de `mongodb`, entonces estos datos serán aplicados a la colección mongodb `users`. Para aprender cómo implementar varias estrategias de carga y más, visita la documentación oficial[20]. El fixture de ejemplo de arriba fue autogenerado por la extensión `yii2-faker`, lee más acerca de esto en su sección. Los nombres de clase de fixtures no deberían ser en plural.

13.7.2. Cargar fixtures

Las clases de fixture deberían tener el prefijo `Fixture`. Por defecto los fixtures serán buscados bajo el espacio de nombre `tests\unit\fixtures`, puedes modificar este comportamiento con opciones de comando o configuración. Puedes excluir algunos fixtures para carga o descarga especificando - antes de su nombre, por ejemplo `-User`.

Para cargar un fixture, ejecuta el siguiente comando:

```
yii fixture/load <fixture_name>
```

El parámetro requerido `fixture_name` especifica un nombre de fixture cuyos datos serán cargados. Puedes cargar varios fixtures de una sola vez. Abajo se muestran formatos correctos de este comando:

```
// carga el fixture `User`
yii fixture/load User

// lo mismo que arriba, dado que la acción por defecto del comando "fixture"
es "load"
yii fixture User

// carga varios fixtures
yii fixture "User, UserProfile"

// carga todos los fixtures
yii fixture/load "*"

// lo mismo que arriba
yii fixture "*"

// carga todos los fixtures excepto uno
yii fixture "*, -DoNotLoadThisOne"

// carga fixtures, pero los busca en diferente espacio de nombre. El espacio
de nombre por defecto es: tests\unit\fixtures.
yii fixture User --namespace='alias\my\custom\namespace'

// carga el fixture global `some\name\space\CustomFixture` antes de que
otros fixtures sean cargados.
// Por defecto está opción se define como `InitDbFixture` para
habilitar/deshabilitar la comprobación de integridad. Puedes especificar
varios
```

[20] https://github.com/yiisoft/yii2/blob/master/docs/guide-es/test-fixtures.md

```
// fixtures globales separados por coma.
yii fixture User --globalFixtures='some\name\space\Custom'
```

13.7.3. Descargar fixtures

Para descargar un fixture, ejecuta el siguiente comando:

```
// descarga el fixture Users, por defecto limpiará el almacenamiento del
fixture (por ejemplo la tabla "users", o la colección "users" si es un
fixture mongodb).
yii fixture/unload User

// descarga varios fixtures
yii fixture/unload "User, UserProfile"

// descarga todos los fixtures
yii fixture/unload "*"

// descarga todos los fixtures excepto uno
yii fixture/unload "*, -DoNotUnloadThisOne"
```

Opciones de comando similares como: `namespace`, `globalFixtures` también pueden ser aplicadas a este comando.

13.7.4. Configurar el Comando Globalmente

Mientras que las opciones de línea de comandos nos permiten configurar el comando de migración en el momento, a veces queremos configurar el comando de una vez y para siempre. Por ejemplo puedes configurar diferentes rutas de migración como a continuación:

```
'controllerMap' => [
    'fixture' => [
        'class' => 'yii\console\controllers\FixtureController',
        'namespace' => 'myalias\some\custom\namespace',
        'globalFixtures' => [
            'some\name\space\Foo',
            'other\name\space\Bar'
        ],
    ],
]
```

13.7.5. Autogenerando fixtures

Yii puede también autogenerar fixtures por tí basándose en algún template. Puedes generar tus fixtures con distintos datos en diferentes lenguajes y formatos. Esta característica es realizada por la librería Faker[21] y la extensión `yii2-faker`. Visita la guía de la extensión[22] para mayor documentación.

[21] https://github.com/fzaninotto/Faker
[22] https://github.com/yiisoft/yii2-faker

Capítulo 14

Temas especiales

Error: not existing file: tutorial-advanced-app.md

14.1. Crear tu propia estructura de Aplicación

Nota: Esta sección se encuentra en desarrollo.

Mientras que los templates de proyectos basic[1] y advanced[2] son grandiosos para la mayoría de tus necesidades, podrías querer crear tu propio template de proyecto del cual partir todos tus proyectos.

Los templates de proyectos en Yii son simplemente repositorios conteniendo un archivo `composer.json`, y registrado como un paquete de Composer. Cualquier repositorio puede ser identificado como paquete Composer, haciéndolo instalable a través del comando de Composer `create-project`.

Dado que es un poco demasiado comenzar tu template de proyecto desde cero, es mejor utilizar uno de los templates incorporados como una base. Utilicemos el template básico aquí.

14.1.1. Clonar el Template Básico

El primer paso es clonar el template básico de Yii desde su repositorio Git:

```
git clone git@github.com:yiisoft/yii2-app-basic.git
```

Entonces espera que el repositorio sea descargado a tu computadora. Dado que los cambios realizados al template no serán enviados al repositorio, puedes eliminar el directorio `.git` y todo su contenido de la descarga.

14.1.2. Modificar los Archivos

A continuación, querrás modificar el archivo `composer.json` para que refleje tu template. Cambia los valores de `name`, `description`, `keywords`, `homepage`, `license`, y `support` de forma que describa tu nuevo template. También ajusta las opciones `require`, `require-dev`, `suggest`, y demás para que encajen con los requerimientos de tu template.

> **Nota:** En el archivo `composer.json`, utiliza el parámetro `writable` (bajo `extra`) para especificar permisos-por-archivo a ser definidos después de que la aplicación es creada a partir del template.

Luego, pasa a modificar la estructura y contenido de la aplicación como te gustaría que sea por defecto. Finalmente, actualiza el archivo README para que sea aplicable a tu template.

[1] https://github.com/yiisoft/yii2-app-basic
[2] https://github.com/yiisoft/yii2-app-advanced

14.1.3. Hacer un Paquete

Con el template definido, crea un repositorio Git a partir de él, y sube tus archivos ahí. Si tu template va a ser de código abierto, Github[3] es el mejor lugar para alojarlo. Si tu intención es que el template no sea colaborativo, cualquier sitio de repositorios Git servirá.

Ahora, necesitas registrar tu paquete para Composer. Para templates públicos, el paquete debe ser registrado en Packagist[4]. Para templates privados, es un poco más complicado registrarlo. Puedes ver instrucciones para hacerlo en la documentación de Composer[5].

14.1.4. Utilizar el Template

Eso es todo lo que se necesita para crear un nuevo template de proyecto Yii. Ahora puedes crear tus propios proyectos a partir de este template:

```
composer global require "fxp/composer-asset-plugin:^1.4.1"
composer create-project --prefer-dist --stability=dev
mysoft/yii2-app-coolone new-project
```

[3] https://github.com
[4] https://packagist.org/
[5] https://getcomposer.org/doc/05-repositories.md#hosting-your-own

Error: not existing file: tutorial-console.md

14.2. Validadores del framework

Yii provee en su núcleo un conjunto de validadores de uso común, que se pueden encontrar principalmente bajo el espacio de nombres (namespace) `yii\validators`. En vez de utilizar interminables nombres de clases para los validadores, puedes usar *alias* para especificar el uso de esos validadores del núcleo. Por ejemplo, puedes usar el alias `required` para referirte a la clase `yii\validators\RequiredValidator` :

```
public function rules()
{
    return [
        [['email', 'password'], 'required'],
    ];
}
```

La propiedad `yii\validators\Validator::$builtInValidators` declara todos los aliases de los validadores soportados.

A continuación, vamos a describir el uso principal y las propiedades de cada validador del núcleo.

14.2.1. boolean

```
[
    // comprueba si "selected" es 0 o 1, sin mirar el tipo de dato
    ['selected', 'boolean'],

    // comprueba si "deleted" es del tipo booleano, alguno entre `true` o
    `false`
    ['deleted', 'boolean', 'trueValue' => true, 'falseValue' => false,
    'strict' => true],
]
```

Este validador comprueba si el valor de la entrada (input) es booleano.

- `trueValue`: El valor representando `true`. Valor por defecto a `'1'`.
- `falseValue`: El valor representando `false`. Valor por defecto a `'0'`.
- `strict`: Si el tipo del valor de la entrada (input) debe corresponder con `trueValue` y `falseValue`. Valor por defecto a `false`.

  Nota: Ya que los datos enviados con la entrada, vía formularios HTML,son todos cadenas (strings), usted debe normalmente dejar la propiedad `strict` a `false`.

14.2.2. captcha

```
[
    ['verificationCode', 'captcha'],
]
```

Este validador es usualmente usado junto con `yii\captcha\CaptchaAction` y `yii\captcha\Captcha` para asegurarse que una entrada es la misma que lo es el código de verificación que enseña el widget `CAPTCHA`.

- `caseSensitive`: cuando la comparación del código de verificación depende de que sean mayúsculas y minúsculas (case sensitive). Por defecto a `false`.
- `captchaAction`: la ruta correspondiente a `CAPTCHA action` que representa (render) la imagen CAPTCHA. Por defecto`'site/captcha'`.
- `skipOnEmpty`: cuando la validación puede saltarse si la entrada está vacía. Por defecto a `false`, lo caul permite que la entrada sea necesaria (required).

14.2.3. `compare`

```
[
    // valida si el valor del atributo "password" es igual al
    "password_repeat"
    ['password', 'compare'],

    // valida si la edad es mayor que o igual que 30
    ['age', 'compare', 'compareValue' => 30, 'operator' => '>='],
]
```

Este validador compara el valor especificado por la entrada con otro valor y, se asegura si su relación es la especificada por la propiedad `operator`.

- `compareAttribute`: El nombre del valor del atributo con el cual debe compararse. Cuando el validador está siendo usado para validar un atributo, el valor por defecto de esta propiedad debe de ser el nombre de el atributo con el sufijo `_repeat`. Por ejemplo, si el atributo a ser validado es `password`, entonces esta propiedad contiene por defecto `password_repeat`.
- `compareValue`: un valor constante con el que el valor de entrada debe ser comparado. Cuando ambos, esta propiedad y `compareAttribute` son especificados, esta preferencia tiene precedencia.
- `operator`: el operador de comparación. Por defecto vale `==`, permitiendo comprobar si el valor de entrada es igual al de `compareAttribute` o `compareValue`. Los siguientes operadores son soportados:
  - `==`: comprueba si dos valores son iguales. La comparación se realiza en modo no estricto.
  - `===`: comprueba si dos valores son iguales. La comparación se realiza en modo estricto.
  - `!=`: comprueba si dos valores NO son iguales. La comparación se realiza en modo no estricto.
  - `!==`: comprueba si dos valores NO son iguales. La comparación se realiza en modo estricto.

- `>`: comprueba si el valor siendo validado es mayor que el valor con el que se compara.
- `>=`: comprueba si el valor siendo validado es mayor o igual que el valor con el que se compara
- `<`: comprueba si el valor siendo validado es menor que el valor con el que se compara
- `<=`: comprueba si el valor siendo validado es menor o igual que el valor con el que se compara

14.2.4. `date`

```
[
    [['from', 'to'], 'date'],
]
```

Este validador comprueba si el valor de entrada es una fecha, tiempo or fecha/tiempo y tiempo en el formato correcto. Opcionalmente, puede convertir el valor de entrada en una fecha/tiempo UNIX y almacenarla en un atributo especificado vía `timestampAttribute`.

- `format`: el formato fecha/tiempo en el que debe estar el valor a ser validado. Esto tiene que ser un patrón fecha/tiempo descrito en manual ICU[6]. Alternativamente tiene que ser una cadena con el prefijo `php:` representando un formato que ha de ser reconocido por la clase `Datetime` de PHP. Por favor, refiérase a `https://www.php.net/manual/es/datetime.createfromformat.php` sobre los formatos soportados. Si no tiene ningún valor, ha de coger el valor de `Yii::$app->formatter->dateFormat`.
- `timestampAttribute`: el nombre del atributo al cual este validador puede asignar el fecha/hora UNIX convertida desde la entrada fecha/hora.

14.2.5. `default`

```
[
    // pone el valor de "age" a null si está vacío
    ['age', 'default', 'value' => null],

    // pone el valor de "country" a "USA" si está vacío
    ['country', 'default', 'value' => 'USA'],

    // asigna "from" y "to" con una fecha 3 días y 6 días a partir de hoy,
    si está vacía
    [['from', 'to'], 'default', 'value' => function ($model, $attribute) {
        return date('Y-m-d', strtotime($attribute === 'to' ? '+3 days' : '+6
        days'));
    }],
]
```

[6] `https://unicode-org.github.io/icu/userguide/format_parse/datetime/#datetime-format-syntax`

Este validador no valida datos. En cambio, asigna un valor por defecto a los atributos siendo validados, si los atributos están vacíos.

- `value`: el valor por defecto o un elemento llamable de PHP que devuelva el valor por defecto, el cual, va a ser asignado a los atributos siendo validados, si estos están vacíos. La signatura de la función PHP tiene que ser como sigue,

```
function foo($model, $attribute) {
    // ... calcula $value ...
    return $value;
}
```

> **Información:** Cómo determinar si un valor está vacío o no, es un tópico separado cubierto en la sección Valores Vacíos .

14.2.6. `double`

```
[
    // comprueba si  "salary" es un número de tipo doble
    ['salary', 'double'],
]
```

Esta validador comprueba si el valor de entrada es un número de tipo doble. Es equivalente a el validador Número .

- `max`: el valor límite superior (incluido) de el valor. Si no tiene valor, significa que no se comprueba el valor superior.
- `min`: el valor límite inferior (incluido) de el valor. Si no tiene valor, significa que no se comprueba el valor inferior.

14.2.7. `email`

```
[
    // comprueba si "email" es una dirección válida de email
    ['email', 'email'],
]
```

Este validador comprueba si el valor de entrada es una dirección válida de email.

- `allowName`: indica cuando permitir el nombre en la dirección de email (p.e. `John Smith <john.smith@example.com>`). Por defecto a `false`.
- `checkDNS`, comprobar cuando el dominio del email existe y tiene cualquier registro A o MX. Es necesario ser consciente que esta comprobación puede fallar debido a problemas temporales de DNS, incluso si el la dirección es válida actualmente. Por defecto a `false`.
- `enableIDN`, indica cuando el proceso de validación debe tener en cuenta el informe de IDN (internationalized domain names). Por defecto a `false`. Dese cuenta que para poder usar la validación de IDN has de instalar y activar la extensión de PHP `intl`, o será lanzada una excepción.

14.2.8. exist

```
[
    // a1 necesita que exista una columna con el atributo "a1"
    ['a1', 'exist'],

    // a1 necesita existir,pero su valor puede usar a2 para comprobar la
    existencia
    ['a1', 'exist', 'targetAttribute' => 'a2'],

    // a1 y a2 necesitan existir ambos, y ambos pueden recibir un mensaje de
    error
    [['a1', 'a2'], 'exist', 'targetAttribute' => ['a1', 'a2']],

    // a1 y a2 necesitan existir ambos, sólo a1 puede recibir el mensaje de
    error
    ['a1', 'exist', 'targetAttribute' => ['a1', 'a2']],

    // a1 necesita existir comprobando la existencia ambos a2 y a3 (usando
    el valor a1)
    ['a1', 'exist', 'targetAttribute' => ['a2', 'a1' => 'a3']],

    // a1 necesita existir. Si a1 es un array, cada elemento de él tiene que
    existir.
    ['a1', 'exist', 'allowArray' => true],
]
```

Este validador comprueba si el valor de entrada puede ser encontrado en una columna de una tabla. Sólo funciona con los atributos del modelo Registro Activo (Active Record). Soporta validación tanto con una simple columna o múltiples columnas.

- `targetClass`: el nombre de la clase Registro Activo (Active Record) debe de ser usada para mirar por el valor de entrada siendo validado. Si no tiene valor, la clase del modelo actualmente siendo validado puede ser usada.
- `targetAttribute`: el nombre del atributo en `targetClass` que debe de ser usado para validar la existencia del valor de entrada. Si no tiene valor, puede usar el nombra del atributoactualmente siendo validado. Puede usar una array para validar la existencia de múltiples columnas al mismo tiempo. El array de valores son los atributos que pueden ser usados para validar la existencia, mientras que las claves del array son los atributos a ser validados. Si la clave y el valor son los mismos, solo en ese momento puedes especificar el valor.
- `filter`: filtro adicional a aplicar a la consulta de la base de datos usado para comprobar la existencia de una valor de entrada. Esto puede ser una cadena o un array representando la condición de la consulta (referirse a `yii\db\Query::where()` sobre el formato de la condición de consulta), o una función anónima con la signatura `function ($query)`, donde `$query` es el objeto `Query` que puedes modificar en la función.

- `allowArray`: indica cuando permitir que el valor de entrada sea un array. Por defecto a `false`.Si la propiedad es `true` y la entrada es un array, cada elemento del array debe existir en la columna destino. Nota que esta propiedad no puede ser `true` si estás validando, por el contrario, múltiple columnas poniendo el valor del atributo `targetAttribute` como que es un array.

14.2.9. `file`

```
[
    // comprueba si "primaryImage" es un fichero mde imagen en formato PNG,
    JPG o GIF.
    // el tamaño del fichero ha de ser menor de 1MB
    ['primaryImage', 'file', 'extensions' => ['png', 'jpg', 'gif'],
    'maxSize' => 1024*1024*1024],
]
```

Este validador comprueba que el fichero subido es el adecuado.

- `extensions`: una lista de extensiones de ficheros que pueden ser subidos. Esto puede ser tanto un array o una cadena conteniendo nombres de extensiones de ficheros separados por un espacio o coma (p.e. “gif, jpg”). Los nombres de las extensiones no diferencian mayúsculas de minúsculas (case-insensitive). Por defecto a `null`, permitiendo todas los nombres de extensiones de fichero.
- `mimeTypes`: una lista de tipos de ficheros MIME que están permitidos subir. Esto puede ser tanto un array como una cadena conteniendo tipos de fichero MIME separados por un espacio o una coma (p.e. “image/jpeg, image/png”). Los tipos Mime no diferencian mayúsculas de minúsculas (case-insensitive). Por defecto a `null`, permitiendo todos los tipos MIME.
- `minSize`: el número de bytes mínimo requerido para el fichero subido. El tamaño del fichero ha de ser superior a este valor. Por defecto a `null`, lo que significa sin límite inferior.
- `maxSize`: El número máximo de bytes del fichero a subir. El tamaño del fichero ha de ser inferior a este valor. Por defecto a `null`, significando no tener límite superior.
- `maxFiles`: el máximo número de ficheros que determinado atributo puede manejar. Por defecto a 1, lo que significa que la entrada debe de ser sólo un fichero. Si es mayor que 1, entonces la entrada tiene que ser un array conteniendo como máximo el número `maxFiles` de elementos que representan los ficheros a subir.
- `checkExtensionByMimeType`: cuando comprobar la extensión del fichero por el tipo MIME. Si la extensión producida por la comprobación del tipo MIME difiere la extensión del fichero subido, el fichero será considerado como no válido. Por defecto a `true`, significando que realiza este tipo de comprobación.

`FileValidator` es usado con `yii\web\UploadedFile`. Por favor, refiérase a la sección Subida de ficheros para una completa cobertura sobre la subida de ficheros y llevar a cabo la validación de los ficheros subidos.

14.2.10. `filter`

```
[
    // recorta (trim) las entradas "username" y "email"
    [['username', 'email'], 'filter', 'filter' => 'trim', 'skipOnArray' =>
    true],

    // normaliza la entrada de  "phone"
    ['phone', 'filter', 'filter' => function ($value) {
        // normaliza la entrada del teléfono aquí
        return $value;
    }],
]
```

Este validador no valida datos. En su lugar, aplica un filtro sobre el valor de entrada y le asigna de nuevo el atributo siendo validado.

- `filter`: una retrollamada (callback) de PHP que define un filtro. Tiene que ser un nombre de función global, una función anónima, etc. La forma de la función ha de ser `function ($value) { return $newValue; }`. Tiene que contener un valor esta propiedad.
- `skipOnArray`: cuando evitar el filtro si el valor de la entrada es un array. Por defecto a `false`. A tener en cuenta que si el filtro no puede manejar una entrada de un array, debes poner esta propiedad a `true`. En otro caso algún error PHP puede ocurrir.

  Consejo (Tip): Si quieres recortar los valores de entrada, puedes usar directamente el validador Recorte (trim).

14.2.11. `image`

```
[
    // comprueba si "primaryImage"  es una imágen valída con el tamaño
    adecuado
    ['primaryImage', 'image', 'extensions' => 'png, jpg',
        'minWidth' => 100, 'maxWidth' => 1000,
        'minHeight' => 100, 'maxHeight' => 1000,
    ],
]
```

Este validador comprueba si el valor de entrada representa un fichero de imagen válido. Extiende al validador Fichero (file) y, por lo tanto, hereda todas sus propiedades. Además, soporta las siguientes propiedades adicionales específicas para la validación de imágenes:

- `minWidth`: el mínimo ancho de la imagen. Por defecto a `null`, indicando que no hay límite inferior.

- `maxWidth`: el máximo ancho de la imagen. Por defecto a `null`, indicando que no hay límite superior.
- `minHeight`: el mínimo alto de la imagen. Por defecto a `null`, indicando que no hay límite inferior.
- `maxHeight`: el máximo alto de la imagen. Por defecto a `null`, indicando que no hay límite superior.

14.2.12. `in`

```
[
    // comprueba si "level" es 1, 2 o 3
    ['level', 'in', 'range' => [1, 2, 3]],
]
```

Este validador comprueba si el valor de entrada puede encontrarse entre determinada lista de valores.

- `range`: una lista de determinados valores dentro de los cuales el valor de entrada debe de ser mirado.
- `strict`: cuando la comparación entre el valor de entrada y los valores determinados debe de ser estricta (ambos el tipo y el valor han de ser iguales). Por defecto a `false`.
- `not`: cuando el resultado de la validación debe de ser invertido. Por defecto a `false`. Cuando esta propiedad está a `true`, el validador comprueba que el valor de entrada NO ESTÁ en la determinada lista de valores.
- `allowArray`: si se permite que el valor de entrada sea un array. Cuando es `true` y el valor de entrada es un array, cada elemento en el array debe de ser encontrado en la lista de valores determinada,o la validación fallará.

14.2.13. `integer`

```
[
    // comrpueba si "age" es un entero
    ['age', 'integer'],
]
```

Esta validador comprueba si el valor de entrada es un entero.

- `max`: el valor superior (incluido) . Si no tiene valor, significa que el validador no comprueba el límite superior.
- `min`: el valor inferior (incluido). Si no tiene valor, significa que el validador no comprueba el límite inferior.

14.2.14. `match`

```
[
    // comprueba si "username" comienza con una letra y contiene solamente
    caracteres en sus palabras
```

```
    ['username', 'match', 'pattern' => '/^[a-z]\w*$/i']
]
```

Este validador comprueba si el valor de entrada coincide con la expresión regular especificada.

- `pattern`: la expresión regular conla que el valor de entrada debe coincidir. Esta propiedad no puede estar vacía, o se lanzará una excepción.
- `not`: indica cuando invertir el resultado de la validación. Por defecto a `false`, significando que la validación es exitosa solamente si el valor de entrada coincide con el patrón. Si esta propiedad está a `true`, la validación es exitosa solamente si el valor de entrada NO coincide con el patrón.

14.2.15. number

```
[
    // comprueba si "salary" es un número
    ['salary', 'number'],
]
```

Este validador comprueba si el valor de entrada es un número. Es equivalente al validador Doble precisión (double).

- `max`: el valor superior límite (incluido) . Si no tiene valor, significa que el validador no comprueba el valor límite superior.
- `min`: el valor inferior límite (incluido) . Si no tiene valor, significa que el validador no comprueba el valor límite inferior.

14.2.16. required

```
[
    // comprueba si ambos "username" y "password" no están vacíos
    [['username', 'password'], 'required'],
]
```

El validador comprueba si el valor de entrada es provisto y no está vacío.

- `requiredValue`: el valor deseado que la entrada debería tener. Si no tiene valor, significa que la entrada no puede estar vacía.
- `strict`: indica como comprobar los tipos de los datos al validar un valor. Por defecto a `false`. Cuando `requiredValue` no tiene valor, si esta propiedad es `true`, el validador comprueba si el valor de entrada no es estrictamente `null`; si la propiedad es `false`, el validador puede usar una regla suelta para determinar si el valor está vacío o no. Cuando `requiredValue` tiene valor, la comparación entre la entrada y `requiredValue` comprobará tambien los tipos de los datos si esta propiedad es `true`.

  Información: Como determinar si un valor está vacío o no es un tópico separado cubierto en la sección Valores vacíos.

14.2.17. safe

```
[
    // marca  "description" como un atributo seguro
    ['description', 'safe'],
]
```

Este validador no realiza validación de datos. En lugar de ello, es usado para marcar un atributo como seguro atributos seguros.

14.2.18. string

```
[
    // comprueba si "username" es una cadena cuya longitud está entre 4 Y
    24
    ['username', 'string', 'length' => [4, 24]],
]
```

Este validador comprueba si el valor de entrada es una cadena válida con determinada longitud.

- `length`: especifica la longitud límite de la cadena de entrada a validar. Esto tiene que ser especificado del las siguientes formas:
  - un entero: la longitud exacta que la cadena debe de tener;
  - un array de un elemento: la longitud mínima de la cadena de entrada (p.e.`[8]`). Esto puede sobre escribir `min`.
  - un array de dos elementos: las longitudes mínima y mmáxima de la cadena de entrada (p.e. `[8, 128]`). Esto sobreescribe ambos valores de `min` y `max`.
- `min`: el mínimo valor de longitud de la cadena de entrada. Si no tiene valor, significa que no hay límite para longitud mínima.
- `max`: el máximo valor de longitud de la cadena de entrada. Si no tiene valor, significa que no hay límite para longitud máxima.
- `encoding`: la codificación de la cadena de entrada a ser validada. Si no tiene valor, usará el valor de la aplicación `charset` que por defecto es `UTF-8`.

14.2.19. trim

```
[
    // recorta (trim) los espacios en blanco que rodean a "username" y
    "email"
    [['username', 'email'], 'trim'],
]
```

Este validador no realiza validación de datos. En cambio, recorta los espacios que rodean el valor de entrada. Nota que si el valor de entrada es un array, se ignorará este validador.

14.2.20. unique

```
[
    // a1 necesita ser único en la columna representada por el atributo
    "a1"
    ['a1', 'unique'],

    // a1 necesita ser único, pero la columna a2 puede ser usado para
    comprobar la unicidad del valor a1
    ['a1', 'unique', 'targetAttribute' => 'a2'],

    // a1 y a2 necesitan ambos ser únicos, y ambospueden recibir el mensaje
    de error
    [['a1', 'a2'], 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a1 y a2 necesitan ser unicos ambos, solamente uno recibirá el mensaje
    de error
    ['a1', 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a1 necesita ser único comprobando la unicidad de ambos a2 y a3
    (usando el valor)
    ['a1', 'unique', 'targetAttribute' => ['a2', 'a1' => 'a3']],
]
```

Este validador comprueba si el valor de entrada es único en una columna de una tabla. Solo funciona con los atributos del modelo Registro Activo (Active Record). Soporta validación contra cualquiera de los casos, una columna o múltiples columnas.

- `targetClass`: el nombre de la clase Registro Activo (Active Record) que debe de ser usada para mirar por el valor de entrada que está siendo validado. Si no tiene valor, la clase del modelo actualmente validado será usada.
- `targetAttribute`: el nombre de el atributo en `targetClass`que debe de ser usado para validar la unicidad de el valor de entrada. Si no tiene valor, puede usar el nombre del atributo actualmente siendo validado. Puedes usar un array para validar la unicidad de múltiples columnas al mismo tiempo. Los valores del array son atributos que pueden ser usados para validar la unicidad, mientras que las claves del array son los atributos que cuyos valores van a ser validados. Si la clave y el valor son el mismo, entonces puedes especificar el valor.
- `filter`: filtro adicional puede ser aplicado a la consulta de la base de datos usado para comprobar la unicidad del valor de entrada. Esto puede ser una cadena o un array representando la condición adicional a la consulta (Referirse a `yii\db\Query::where()` para el formato de la condición de la consulta), o una función anónima de la forma `function ($query)`, donde `$query` es el objeto `Query` que puedes modificar en la función.

14.2.21. `url`

```
[
    // comprueba si "website" es una URL válida. Prefija con "http://" al
    atributo  "website"
    // si no tiene un esquema URI
    ['website', 'url', 'defaultScheme' => 'http'],
]
```

Este validador comprueba si el valor de entrada es una URL válida.

- `validSchemes`: un array especificando el esquema URI que debe ser considerado válido. Por defecto contiene `['http', 'https']`, significando que ambas URLS `http` y `https` son consideradas válidas.
- `defaultScheme`: el esquema de URI a poner como prefijo a la entrada si no tiene la parte del esquema. Por defecto a `null`, significando que no modifica el valor de entrada.
- `enableIDN`: Si el validador debe formar parte del registro IDN (internationalized domain names). Por defecto a `false`. Nota que para usar la validación IDN tienes que instalar y activar la extensión PHP `intl`, en otro caso una excepción será lanzada.

Error: not existing file: tutorial-i18n.md

14.3. Envío de Emails

Nota: Esta sección se encuentra en desarrollo.

Yii soporta composición y envío de emails. De cualquier modo, el núcleo del framework provee sólo la funcionalidad de composición y una interfaz básica. En mecanismo de envío en sí debería ser provisto por la extensión, dado que diferentes proyectos pueden requerir diferente implementación y esto usualmente depende de servicios y librerías externas.

Para la mayoría de los casos, puedes utilizar la extensión oficial yii2-swiftmailer[7].

14.3.1. Configuración

La configuración del componente Mail depende de la extensión que hayas elegido. En general, la configuración de tu aplicación debería verse así:

```
return [
    //....
    'components' => [
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
    ],
];
```

14.3.2. Uso Básico

Una vez configurado el componente 'mailer', puedes utilizar el siguiente código para enviar un correo electrónico:

```
Yii::$app->mailer->compose()
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Asunto del mensaje')
    ->setTextBody('Contenido en texto plano')
    ->setHtmlBody('<b>Contenido HTML</b>')
    ->send();
```

En el ejemplo anterior, el método `compose()` crea una instancia del mensaje de correo, el cual puede ser llenado y enviado. En caso de ser necesario, puedes agregar una lógica más compleja en el proceso:

```
$message = Yii::$app->mailer->compose();
if (Yii::$app->user->isGuest) {
    $message->setFrom('from@domain.com')
} else {
    $message->setFrom(Yii::$app->user->identity->email)
```

[7] https://github.com/yiisoft/yii2-swiftmailer

```
}
$message->setTo(Yii::$app->params['adminEmail'])
    ->setSubject('Asunto del mensaje')
    ->setTextBody('Contenido en texto plano')
    ->send();
```

> **Nota:** cada extensión 'mailer' viene en dos grandes clases: 'Mailer' y 'Message'. 'Mailer' siempre conoce el nombre de clase especifico de 'Message'. No intentes instanciar el objeto 'Message' directamente - siempre utiliza el método `compose()` para ello.

Puedes también enviar varios mensajes al mismo tiempo:

```
$messages = [];
foreach ($users as $user) {
    $messages[] = Yii::$app->mailer->compose()
        // ...
        ->setTo($user->email);
}
Yii::$app->mailer->sendMultiple($messages);
```

Algunas extensiones en particular pueden beneficiarse de este enfoque, utilizando mensaje simple de red, etc.

14.3.3. Componer el contenido del mensaje

Yii permite componer el contenido de los mensajes de correo a través de archivos de vista especiales. Por defecto, estos archivos deben estar ubicados en la ruta '@app/mail'.

Ejemplo de archivo de contenido de correo:

```
<?php
use yii\helpers\Html;
use yii\helpers\Url;


/* @var $this \yii\web\View instancia del componente view */
/* @var $message \yii\mail\BaseMessage instancia del mensaje de correo
recién creado */

?>
<h2>Este mensaje te permite visitar nuestro sitio con un sólo click</h2>
<?= Html::a('Ve a la página principal', Url::home('http')) ?>
```

Para componer el contenido del mensaje utilizando un archivo, simplemente pasa el nombre de la vista al método `compose()`:

```
Yii::$app->mailer->compose('home-link') // el resultado del renderizado de
la vista se transforma en el cuerpo del mensaje aquí
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Asunto del mensaje')
    ->send();
```

Puedes pasarle parámetros adicionales a la vista en el método `compose()`, los cuales estarán disponibles dentro de las vistas:

```
Yii::$app->mailer->compose('greetings', [
    'user' => Yii::$app->user->identity,
    'advertisement' => $adContent,
]);
```

Puedes especificar diferentes archivos de vista para el contenido del mensaje en HTML y texto plano:

```
Yii::$app->mailer->compose([
    'html' => 'contact-html',
    'text' => 'contact-text',
]);
```

Si especificas el nombre de la vista como un string, el resultado de su renderización será utilizado como cuerpo HTML, mientras que el cuerpo en texto plano será compuesto removiendo todas las entidades HTML del anterior.

El resultado de la renderización de la vista puede ser envuelta en el layout, que puede ser definido utiliazando `yii\mail\BaseMailer::$htmlLayout` y `yii\mail\BaseMailer::$textLayout`. Esto funciona igual a como funcionan los layouts en una aplicación web normal. El layout puede utilizar estilos CSS u otros contenidos compartidos:

```
<?php
use yii\helpers\Html;

/* @var $this \yii\web\View instancia del componente view */
/* @var $message \yii\mail\MessageInterface el mensaje siendo compuesto */
/* @var $content string el resultado de la renderización de la vista
principal */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=<?=
Yii::$app->charset ?>" />
    <style type="text/css">
        .heading {...}
        .list {...}
        .footer {...}
    </style>
    <?php $this->head() ?>
</head>
<body>
    <?php $this->beginBody() ?>
    <?= $content ?>
    <div class="footer">Saludos cordiales, el equipo de<?= Yii::$app->name
?></div>
```

```
    <?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

14.3.4. Adjuntar archivos

Puedes adjuntar archivos al mensaje utilizando los métodos `attach()` y `attachContent()`:

```
$message = Yii::$app->mailer->compose();

// Adjunta un archivo del sistema local de archivos:
$message->attach('/path/to/file.pdf');

// Crear adjuntos sobre la marcha
$message->attachContent('Contenido adjunto', ['fileName' => 'attach.txt',
'contentType' => 'text/plain']);
```

14.3.5. Incrustar imágenes

Puedes incrustar imágenes en el mensaje utilizando el método `embed()`. Este método devuelve el id del adjunto, que debería ser utilizado como tag 'img'. Este método es fácil de utilizar al componer mensajes a través de un archivo de vista:

```
Yii::$app->mailer->compose('embed-email', ['imageFileName' =>
'/path/to/image.jpg'])
    // ...
    ->send();
```

Entonces, dentro de tu archivo de vista, puedes utilizar el siguiente código:

```
<img src="<?= $message->embed($imageFileName); ?>">
```

14.3.6. Testear y depurar

Un desarrollador a menudo necesita comprobar qué emails están siendo enviados por la aplicación, cuál es su contenido y otras cosas. Yii concede dicha habilidad vía `yii\mail\BaseMailer::useFileTransport`. Si se habilita, esta opción hace que los datos del mensaje sean guardados en archivos locales en vez de enviados. Esos archivos serán guardados bajo `yii\mail\BaseMailer::fileTransportPath`, que por defecto es '@runtime/mail'.

> **Nota:** puedes o bien guardar los mensajes en archivos, o enviarlos a sus receptores correspondientes, pero no puedes hacer las dos cosas al mismo tiempo.

Un archivo de mensaje puede ser abierto por un editor de texto común, de modo que puedas ver sus cabeceras, su contenido y demás. Este mecanismo en sí puede comprobarse al depurar la aplicación o al ejecutar un test de unidad.

> **Nota:** el archivo de contenido de mensaje es compuesto vía `\yii\mail\MessageInterface::toString()`, por lo que depende de la extensión actual de correo utilizada en tu aplicación.

14.3.7. Crear tu solución personalizada de correo

Para crear tu propia solución de correo, necesitas crear 2 clases: una para 'Mailer' y otra para 'Message'. Puedes utilizar `yii\mail\BaseMailer` y `yii\mail\BaseMessage` como clases base de tu solución. Estas clases ya contienen un lógica básica, la cual se describe en esta guía. De cualquier modo, su utilización no es obligatoria, es suficiente con implementar las interfaces `yii\mail\MailerInterface` y `yii\mail\MessageInterface`. Luego necesitas implementar todos los métodos abstractos para construir tu solución.

Error: not existing file: tutorial-performance-tuning.md

Error: not existing file: tutorial-shared-hosting.md

14.4. Usar motores de plantillas

Por defecto, Yii utiliza PHP como su lenguaje de plantilla, pero puedes configurar Yii para que soporte otros motores de renderizado, tal como Twig[8] o Smarty[9], disponibles como extensiones.

El componente `view` es el responsable de renderizar las vistas. Puedes agregar un motor de plantillas personalizado reconfigurando el comportamiento (behavior) de este componente:

```
[
    'components' => [
        'view' => [
            'class' => 'yii\web\View',
            'renderers' => [
                'tpl' => [
                    'class' => 'yii\smarty\ViewRenderer',
                    //'cachePath' => '@runtime/Smarty/cache',
                ],
                'twig' => [
                    'class' => 'yii\twig\ViewRenderer',
                    'cachePath' => '@runtime/Twig/cache',
                    // Array de opciones de Twig:
                    'options' => [
                        'auto_reload' => true,
                    ],
                    'globals' => ['html' => '\yii\helpers\Html'],
                    'uses' => ['yii\bootstrap'],
                ],
                // ...
            ],
        ],
    ],
]
```

En el código de arriba, tanto Smarty como Twig son configurados para ser utilizables por los archivos de vista. Pero para tener ambas extensiones en tu proyecto, también necesitas modificar tu archivo `composer.json` para incluirlos:

```
"yiisoft/yii2-smarty": "~2.0.0",
"yiisoft/yii2-twig": "~2.0.0",
```

Ese código será agregado a la sección `require` de `composer.json`. Después de realizar ese cambio y guardar el archivo, puedes instalar estas extensiones ejecutando `composer update --prefer-dist` en la línea de comandos.

Para más detalles acerca del uso concreto de cada motor de plantillas, visita su documentación:

[8] https://twig.symfony.com/
[9] https://www.smarty.net/

- Guía de Twig[10]
- Guía de Smarty[11]

14.5. Trabajar con código de terceros

De tiempo en tiempo, puede necesitar usar algún código de terceros en sus aplicaciones Yii. O puedes querer utilizar Yii como una librería en otros sistemas de terceros. En esta sección, te enseñaremos cómo conseguir estos objetivos.

14.5.1. Utilizar librerías de terceros en Yii

Para usar una librería en una aplicación Yii, primeramente debes de asegurarte que las clases en la librería son incluidas adecuadamente o pueden ser cargadas de forma automática.

Usando Paquetes de Composer

Muchas librerías de terceros son liberadas en términos de paquetes Composer[12]. Puedes instalar este tipo de librerías siguiendo dos sencillos pasos:

1. modificar el fichero `composer.json` de tu aplicación y especificar que paquetes Composer quieres instalar.

2. ejecuta `composer install` para instalar los paquetes especificados.

Las clases en los paquetes Composer instalados pueden ser autocargados usando el cargador automatizado de Composer autoloader. Asegúrate que el fichero script de entrada de tu aplicación contiene las siguientes líneas para instalar el cargador automático de Composer:

```
// instalar el cargador automático de  Composer
require __DIR__ . '/../vendor/autoload.php';

// incluir rl fichero de la clase Yii
require __DIR__ . '/../vendor/yiisoft/yii2/Yii.php';
```

Usando librerías Descargadas

Si la librería no es liberada como un paquete de Composer, debes de seguir sus instrucciones de instalación para instalarla. En muchos casos, puedes necesitar descargar manualmente el fichero de la versión y desempaquetarlo

[10] https://github.com/yiisoft/yii2-twig/tree/master/docs/guide
[11] https://github.com/yiisoft/yii2-smarty/tree/master/docs/guide
[12] https://getcomposer.org/

en el directorio `BasePath/vendor`, donde `BasePath` representa el camino base (base path) de tu aplicación.

Si la librería lleva su propio cargador automático (autoloader), puedes instalarlo en script de entrada de tu aplicación. Es recomendable que la instalación se termine antes de incluir el fichero `Yii.php` de forma que el cargador automático tenga precedencia al cargar de forma automática las clases.

Si la librería no provee un cargador automático de clases, pero la denominación de sus clases sigue el PSR-4[13], puedes usar el cargador automático de Yii para cargar de forma automática las clases. Todo lo que necesitas es declarar un alias raíz para cada espacio de nombres (namespace) raiz usado en sus clases. Por ejemplo, asume que has instalado una librería en el directorio `vendor/foo/bar`, y que las clases de la librería están bajo el espacio de nombres raiz `xyz`. Puedes incluir el siguiente código en la configuración de tu aplicación:

```
[
    'aliases' => [
        '@xyz' => '@vendor/foo/bar',
    ],
]
```

Si ninguno de lo anterior es el caso, estaría bien que la librería dependa del camino de inclusión (include path) de configuración de PHP para localizar correctamente e incluir los ficheros de las clases. Simplemente siguiendo estas instrucciones de cómo configurar el camino de inclusión de PHP.

En el caso más grave en el que la librería necesite incluir cada uno de sus ficheros de clases, puedes usar el siguiente método para incluir las clases según se pidan:

- Identificar que clases contiene la librería.
- Listar las clases y el camino a los archivos correspondientes en `Yii::$classMap` en el script de entrada script de entrada de la aplicación. Por ejemplo,

  ```
  Yii::$classMap['Class1'] = 'path/to/Class1.php';
  Yii::$classMap['Class2'] = 'path/to/Class2.php';
  ```

14.5.2. Utilizar Yii en Sistemas de Terceros

Debido a que Yii provee muchas posibilidades excelentes, a veces puedes querer usar alguna de sus características para permitir el desarrollo o mejora de sistemas de terceros, como es WordPress, Joomla, o aplicaciones desarrolladas usando otros frameworks de PHP. Por ejemplo, puedes querer utilizar la clase `yii\helpers\ArrayHelper` o usar la característica Active Record en un sistema de terceros. Para lograr este objetivo, principalmente necesitas realizar dos pasos: instalar Yii , e iniciar Yii.

[13] https://www.php-fig.org/psr/psr-4/

Si el sistema de terceros usa Composer para manejar sus dependencias, simplemente ejecuta estos comandos para instalar Yii:

```
composer global require "fxp/composer-asset-plugin:^1.4.1"
composer require yiisoft/yii2
composer install
```

El primer comando instala el composer asset plugin[14], que permite administrar paquetes bower y npm a través de Composer. Incluso si sólo quieres utilizar la capa de base de datos u otra característica de Yii no relacionada a assets, requiere que instales el paquete composer de Yii.

Si quieres utilizar la publicación de Assets de Yii deberías agregar también la siguiente configuración a la sección `extra` de tu `composer.json`:

```
{
    ...
    "extra": {
        "asset-installer-paths": {
            "npm-asset-library": "vendor/npm",
            "bower-asset-library": "vendor/bower"
        }
    }
}
```

Visita también la sección de cómo instalar Yii para más información sobre Composer y sobre cómo solucionar posibles problemas que surjan durante la instalación.

En otro caso, puedes descargar[15] el archivo de la edición de Yii y desempaquetarla en el directorio `BasePath/vendor`.

Después, debes de modificar el script de entrada de sistema de terceros para incluir el siguiente código al principio:

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

$yiiConfig = require __DIR__ . '/../config/yii/web.php';
new yii\web\Application($yiiConfig); // No ejecutes run() aquí
```

Como puedes ver, el código anterior es muy similar al que puedes ver en script de entrada de una aplicación típica. La única diferencia es que después de que se crea la instancia de la aplicación, el método `run()` no es llamado. Esto es así porque llamando a `run()`, Yii se haría cargo del control del flujo de trabajo del manejo de las peticiones, lo cual no es necesario en este caso por estar ya es manejado por la aplicación existente.

Como en una aplicación Yii, debes configurar la instancia de la aplicación basándose en el entorno que se está ejecutando del sistema de terceros. Por

[14] https://github.com/fxpio/composer-asset-plugin
[15] https://www.yiiframework.com/download/

ejemplo, para usar la característica Active Record, necesitas configurar el componente de la aplicación `db` con los parámetros de la conexión a la BD del sistema de terceros.

Ahora puedes usar muchas características provistas por Yii. Por ejemplo, puedes crear clases Active Record y usarlas para trabajar con bases de datos.

14.5.3. Utilizar Yii 2 con Yii 1

Si estaba usando Yii 1 previamente, es como si tuvieras una aplicación Yii 1 funcionando. En vez de reescribir toda la aplicación en Yii 2, puedes solamente mejorarla usando alguna de las características sólo disponibles en Yii 2. Esto se puede lograr tal y como se describe abajo.

> **Nota:** Yii 2 requiere PHP 5.4 o superior. Debes de estar seguro que tanto tu servidor como la aplicación existente lo soportan.

Primero, instala Yii 2 en tu aplicación siguiendo las instrucciones descritas en la última subsección.

Segundo, modifica el script de entrada de la aplicación como sigue,

```
// incluir la clase Yii personalizada descrita debajo
require __DIR__ . '/../components/Yii.php';

// configuración para la aplicación Yii 2
$yii2Config = require __DIR__ . '/../config/yii2/web.php';
new yii\web\Application($yii2Config); // No llamar a run()

// configuración para la aplicación Yii 1
$yii1Config = require __DIR__ . '/../config/yii1/main.php';
Yii::createWebApplication($yii1Config)->run();
```

Debido a que ambos Yii 1 y Yii 2 tiene la clase `Yii` , debes crear una versión personalizada para combinarlas. El código anterior incluye el fichero con la clase `Yii` personalizada, que tiene que ser creada como sigue.

```
$yii2path = '/path/to/yii2';
require $yii2path . '/BaseYii.php'; // Yii 2.x

$yii1path = '/path/to/yii1';
require $yii1path . '/YiiBase.php'; // Yii 1.x

class Yii extends \yii\BaseYii
{
    // copy-paste the code from YiiBase (1.x) here
}

Yii::$classMap = include($yii2path . '/classes.php');
// registrar el autoloader de Yii 2 vía Yii 1
Yii::registerAutoloader(['Yii', 'autoload']);
// crear el contenedor de inyección de dependencia
Yii::$container = new yii\di\Container;
```

¡Esto es todo!. Ahora, en cualquier parte de tu código, puedes usar `Yii::$app` para acceder a la instancia de la aplicación de Yii 2, mientras `Yii::app()` proporciona la instancia de la aplicación de Yii 1 :

```
echo get_class(Yii::app()); // genera 'CWebApplication'
echo get_class(Yii::$app);  // genera 'yii\web\Application'
```

Capítulo 15

Widgets

Capítulo 16

Clases auxiliares

16.1. Helpers

> **Nota:** Esta sección está en desarrollo.

Yii ofrece muchas clases que ayudan a simplificar las tareas comunes de codificación, como manipulación de string o array, generación de código HTML, y más. Estas clases helper están organizadas bajo el namespace `yii\helpers` y son todo clases estáticas (lo que significa que sólo contienen propiedades y métodos estáticos y no deben ser instanciadas).

Puedes usar una clase helper directamente llamando a uno de sus métodos estáticos, como a continuación:

```
use yii\helpers\Html;

echo Html::encode('Test > test');
```

> **Nota:** Para soportar la personalización de clases helper, Yii separa cada clase helper del núcleo en dos clases: una clase base (ej. `BaseArrayHelper`) y una clase concreta (ej. `ArrayHelper`). Cuando uses un helper, deberías sólo usar la versión concreta y nunca usar la clase base.

16.1.1. Clases Helper del núcleo

Las siguientes clases helper del núcleo son proporcionadas en los releases de Yii:

- ArrayHelper
- Console
- FileHelper
- Html
- HtmlPurifier
- Image

- Inflector
- Json
- Markdown
- Security
- StringHelper
- Url
- VarDumper

16.1.2. Personalizando Las Clases Helper

Para personalizar una clase helper del núcleo (ej. `yii\helpers\ArrayHelper`), deberías crear una nueva clase extendiendo de los helpers correspondientes a la clase base (ej. `yii\helpers\BaseArrayHelper`), incluyendo su namespace. Esta clase será creada para remplazar la implementación original del framework.

El siguiente ejemplo muestra como personalizar el método `merge()` de la clase `yii\helpers\ArrayHelper`:

```
<?php

namespace yii\helpers;

class ArrayHelper extends BaseArrayHelper
{
    public static function merge($a, $b)
    {
        // tu implementación personalizada
    }
}
```

Guarda tu clase en un fichero nombrado `ArrayHelper.php`. El fichero puede estar en cualquier directorio, por ejemplo `@app/components`.

A continuación, en tu script de entrada de la aplicación, añade las siguientes lineas de código después de incluir el fichero `yii.php` para decirle a la clase autoloader de Yii que cargue tu clase personalizada en vez de la clase helper original del framework:

```
Yii::$classMap['yii\helpers\ArrayHelper'] =
'@app/components/ArrayHelper.php';
```

Nota que la personalización de clases helper sólo es útil si quieres cambiar el comportamiento de una función existente de los helpers. Si quieres añadir funciones adicionales para usar en tu aplicación puedes mejor crear un helper por separado para eso.

16.2. ArrayHelper

Adicionalmente al rico conjunto de funciones para arrays de PHP[1], el array helper de Yii proporciona métodos estáticos adicionales permitiendo trabajar con arrays de manera más eficiente.

16.2.1. Devolviendo Valores

Recuperar valores de un array, un objeto o una estructura compleja usando PHP estándar es bastante repetitivo. Tienes que comprobar primero si una clave existe con `isset`, después devolver el valor si existe, si no, devolver un valor por defecto:

```
class User
{
    public $name = 'Alex';
}

$array = [
    'foo' => [
        'bar' => new User(),
    ]
];

$value = isset($array['foo']['bar']->name) ? $array['foo']['bar']->name :
null;
```

Yii proviene de un método muy conveniente para hacerlo:

```
$value = ArrayHelper::getValue($array, 'foo.bar.name');
```

El primer argumento del método es de donde vamos a obtener el valor. El segundo argumento especifica como devolver el dato. Puede ser de la siguiente manera:

- Nombre de la clave del array o de la propiedad del objeto para recuperar el valor.
- Conjunto de puntos separados por las claves del array o los nombres de las propiedades del objeto. Esto se ha usado en el ejemplo anterior.
- Un callback que devuelve un valor.

El callback se debería usar de la siguiente manera:

```
$fullName = ArrayHelper::getValue($user, function ($user, $defaultValue) {
    return $user->firstName . ' ' . $user->lastName;
});
```

El tercer argumento opcional es el valor por defecto el cual es `null` si no se especifica. Podría ser utilizado de la siguiente manera:

[1] https://www.php.net/manual/es/book.array.php

```php
$username = ArrayHelper::getValue($comment, 'user.username', 'Unknown');
```

En caso de que quieras coger un valor y luego removerlo inmediatamente del array puedes usar el método `remove`:

```php
$array = ['type' => 'A', 'options' => [1, 2]];
$type = ArrayHelper::remove($array, 'type');
```

Después de ejecutar el código el `$array` contendrá `['options' => [1, 2]]` y `$type` debe ser `A`. Tenga en cuenta que a diferencia del método `getValue`, `remove` solo soporta nombres clave simples.

16.2.2. Comprobando la Existencia de Claves

`ArrayHelper::keyExists` funciona de la misma manera que array_key_exists[2] excepto que también soporta case-insensitive para la comparación de claves. Por ejemplo,

```php
$data1 = [
    'userName' => 'Alex',
];

$data2 = [
    'username' => 'Carsten',
];

if (!ArrayHelper::keyExists('username', $data1, false) ||
!ArrayHelper::keyExists('username', $data2, false)) {
    echo "Please provide username.";
}
```

16.2.3. Recuperando Columnas

A menudo necesitas obtener unos valores de una columna de las filas de datos u objetos de un array. Un ejemplo común es obtener una lista de IDs.

```php
$data = [
    ['id' => '123', 'data' => 'abc'],
    ['id' => '345', 'data' => 'def'],
];
$ids = ArrayHelper::getColumn($array, 'id');
```

El resultado será `['123', '345']`.

Si se requieren transformaciones adicionales o la manera de obtener el valor es complejo, se podría especificar como segundo argumento una función anónima :

```php
$result = ArrayHelper::getColumn($array, function ($element) {
    return $element['id'];
});
```

[2] https://www.php.net/manual/es/function.array-key-exists.php

16.2.4. Re-indexar Arrays

Con el fin de indexar un array según una clave especificada, se puede usar el método `index`. La entrada debería ser un array multidimensional o un array de objetos. `$key` puede ser tanto una clave del sub-array, un nombre de una propiedad del objeto, o una función anónima que debe devolver el valor que será utilizado como clave.

El atributo `$groups` es un array de claves, que será utilizado para agrupar el array de entrada en uno o más sub-arrays basado en la clave especificada.

Si el atributo `$key` o su valor por el elemento en particular es `null` y `$groups` no está definido, dicho elemento del array será descartado. De otro modo, si `$groups` es especificado, el elemento del array será agregado al array resultante sin una clave.

Por ejemplo:

```
$array = [
    ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
    ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
];
$result = ArrayHelper::index($array, 'id');');
```

El resultado será un array asociativo, donde la clave es el valor del atributo `id`

```
[
    '123' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    '345' => ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone']
    // El segundo elemento del array original es sobrescrito por el último
    elemento debido a que tiene el mismo id
]
```

Pasando una función anónima en `$key`, da el mismo resultado.

```
$result = ArrayHelper::index($array, function ($element) {
    return $element['id'];
});
```

Pasando `id` como tercer argumento, agrupará `$array` mediante `id`:

```
$result = ArrayHelper::index($array, null, 'id');
```

El resultado será un array multidimensional agrupado por `id` en su primer nivel y no indexado en su segundo nivel:

```
[
    '123' => [
        ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
    ],
```

```
    '345' => [ // todos los elementos con este índice están presentes en el
    array resultante
        ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
        ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
    ]
]
```

Una función anónima puede ser usada también en el array agrupador:

```
$result = ArrayHelper::index($array, 'data', [function ($element) {
    return $element['id'];
}, 'device']);
```

El resultado será un array multidimensional agrupado por `id` en su primer nivel, por `device` en su segundo nivel e indexado por `data` en su tercer nivel:

```
[
    '123' => [
        'laptop' => [
            'abc' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
        ]
    ],
    '345' => [
        'tablet' => [
            'def' => ['id' => '345', 'data' => 'def', 'device' => 'tablet']
        ],
        'smartphone' => [
            'hgi' => ['id' => '345', 'data' => 'hgi', 'device' =>
            'smartphone']
        ]
    ]
]
```

16.2.5. Construyendo Mapas (Maps)

Con el fin de construir un mapa (pareja clave-valor) de un array multidimensional o un array de objetos puedes usar el método `map`. Los parámetros `$from` y `$to` especifican los nombres de las claves o los nombres de las propiedades que serán configuradas en el mapa. Opcionalmente, se puede agrupar en el mapa de acuerdo al campo de agrupamiento `$group`. Por ejemplo,

```
$array = [
    ['id' => '123', 'name' => 'aaa', 'class' => 'x'],
    ['id' => '124', 'name' => 'bbb', 'class' => 'x'],
    ['id' => '345', 'name' => 'ccc', 'class' => 'y'],
];

$result = ArrayHelper::map($array, 'id', 'name');
// el resultado es:
// [
//     '123' => 'aaa',
//     '124' => 'bbb',
```

```
//     '345' => 'ccc',
// ]

$result = ArrayHelper::map($array, 'id', 'name', 'class');
// el resultado es:
// [
//     'x' => [
//         '123' => 'aaa',
//         '124' => 'bbb',
//     ],
//     'y' => [
//         '345' => 'ccc',
//     ],
// ]
```

16.2.6. Ordenamiento Multidimensional

El método `multisort` ayuda a ordenar un array de objetos o arrays anidados por una o varias claves. Por ejemplo,

```
$data = [
    ['age' => 30, 'name' => 'Alexander'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 19, 'name' => 'Barney'],
];
ArrayHelper::multisort($data, ['age', 'name'], [SORT_ASC, SORT_DESC]);
```

Después del ordenado obtendremos lo siguiente en `$data`:

```
[
    ['age' => 19, 'name' => 'Barney'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 30, 'name' => 'Alexander'],
];
```

El segundo argumento que especifica las claves para ordenar puede ser una cadena si se trata de una clave, un array en caso de que tenga múltiples claves o una función anónima como la siguiente

```
ArrayHelper::multisort($data, function($item) {
    return isset($item['age']) ? ['age', 'name'] : 'name';
});
```

El tercer argumento es la dirección. En caso de ordenar por una clave podría ser `SORT_ASC` o `SORT_DESC`. Si ordenas por múltiples valores puedes ordenar cada valor diferentemente proporcionando un array de direcciones de ordenación.

El último argumento es un PHP sort flag que toma los mismos valores que los pasados a PHP sort()[3].

[3] https://www.php.net/manual/es/function.sort.php

16.2.7. Detectando Tipos de Array

Es muy útil saber si un array es indexado o asociativo. He aquí un ejemplo:

```
// sin claves especificadas
$indexed = ['Qiang', 'Paul'];
echo ArrayHelper::isIndexed($indexed);

// todas las claves son strings
$associative = ['framework' => 'Yii', 'version' => '2.0'];
echo ArrayHelper::isAssociative($associative);
```

16.2.8. Codificación y Decodificación de Valores HTML

Con el fin de codificar o decodificar caracteres especiales en un array de strings con entidades HTML puedes usar lo siguiente:

```
$encoded = ArrayHelper::htmlEncode($data);
$decoded = ArrayHelper::htmlDecode($data);
```

Solo los valores se codifican por defecto. Pasando como segundo argumento `false` puedes codificar un array de claves también. La codificación utilizará el charset de la aplicación y podría ser cambiado pasandole un tercer argumento.

16.2.9. Fusionando Arrays

```
  /**
    * Fusiona recursivamente dos o más arrays en uno.
    * Si cada array tiene un elemento con el mismo valor string de clave, el
último
    * sobrescribirá el anterior (difiere de array_merge_recursive).
    * Se llegará a una fusión recursiva si ambos arrays tienen un elemento
tipo array
    * y comparten la misma clave.
    * Para elementos cuyas claves son enteros, los elementos del array
final
    * serán agregados al array anterior.
    * @param array $a array al que se va a fusionar
    * @param array $b array desde el cual fusionar. Puedes especificar
    * arrays adicionales mediante el tercer argumento, cuarto argumento,
etc.
    * @return array el array fusionado (los arrays originales no sufren
cambios)
    */
    public static function merge($a, $b)
```

16.2.10. Convirtiendo Objetos a Arrays

A menudo necesitas convertir un objeto o un array de objetos a un array. El caso más común es convertir los modelos de active record con el fin de

servir los arrays de datos vía API REST o utilizarlos de otra manera. El siguiente código se podría utilizar para hacerlo:

```
$posts = Post::find()->limit(10)->all();
$data = ArrayHelper::toArray($posts, [
    'app\models\Post' => [
        'id',
        'title',
        // el nombre de la clave del resultado del array => nombre de la
        propiedad
        'createTime' => 'created_at',
        // el nombre de la clave del resultado del array => función anónima
        'length' => function ($post) {
            return strlen($post->content);
        },
    ],
]);
```

El primer argumento contiene el dato que queremos convertir. En nuestro caso queremos convertir un modelo AR `Post`.

El segundo argumento es el mapeo de conversión por clase. Estamos configurando un mapeo para el modelo `Post`. Cada array de mapeo contiene un conjunto de mapeos. Cada mapeo podría ser:

- Un campo nombre para incluir como está.
- Un par clave-valor del array deseado con un nombre clave y el nombre de la columna del modelo que tomará el valor.
- Un par clave-valor del array deseado con un nombre clave y una función anónima que retorne el valor.

El resultado de la conversión anterior será:

```
[
    'id' => 123,
    'title' => 'test',
    'createTime' => '2013-01-01 12:00AM',
    'length' => 301,
]
```

Es posible proporcionar una manera predeterminada de convertir un objeto a un array para una clase especifica mediante la implementación de la interfaz `Arrayable` en esa clase.

16.2.11. Haciendo pruebas con Arrays

A menudo necesitarás comprobar está en un array o un grupo de elementos es un sub-grupo de otro. A pesar de que PHP ofrece `in_array()`, este no soporta sub-grupos u objetos de tipo `\Traversable`.

Para ayudar en este tipo de pruebas, `yii\helpers\ArrayHelper` provee `isIn()` y `isSubset()` con la misma firma del método in_array()[4].

[4] https://www.php.net/manual/es/function.in-array.php

```
// true
ArrayHelper::isIn('a', ['a']);
// true
ArrayHelper::isIn('a', new(ArrayObject['a']));

// true
ArrayHelper::isSubset(new(ArrayObject['a', 'c']), new(ArrayObject['a', 'b',
'c'])
```

16.3. Clase auxiliar Html (Html helper)

Todas las aplicaciones web generan grandes cantidades de marcado HTML (HTML markup). Si el marcado es estático, se puede realizar de forma efectiva mezclando PHP y HTML en un mismo archivo[5] pero cuando se generan dinámicamente empieza a complicarse su gestión sin ayuda extra. Yii ofrece esta ayuda en forma de una clase auxiliar Html que proporciona un conjunto de métodos estáticos para gestionar las etiquetas HTML más comúnmente usadas, sus opciones y contenidos.

> **Nota:** Si el marcado es casi estático, es preferible usar HTML directamente. No es necesario encapsularlo todo con llamadas a la clase auxiliar Html.

16.3.1. Lo fundamental

Teniendo en cuenta que la construcción de HTML dinámico mediante la concatenación de cadenas de texto se complica rápidamente, Yii proporciona un conjunto de métodos para manipular las opciones de etiquetas y la construcción de las mismas basadas en estas opciones.

Generación de etiquetas

El código de generación de etiquetas es similar al siguiente:

```
<?= Html::tag('p', Html::encode($user->name), ['class' => 'username']) ?>
```

El primer argumento es el nombre de la etiqueta. El segundo es el contenido que se ubicará entre la etiqueta de apertura y la de cierre. Hay que tener en cuenta que estamos usando `Html::encode`. Esto es debido a que el contenido no se codifica automáticamente para permitir usar HTML cuando se necesite. La tercera opción es un array de opciones HTML o, en otras palabras, los atributos de las etiquetas. En este array la clave representa el nombre del atributo como podría ser `class`, `href` o `target` y el valor es su valor.

El código anterior generará el siguiente HTML:

[5] https://www.php.net/manual/es/language.basic-syntax.phpmode.php

```
<p class="username">samdark</p>
```

Si se necesita solo la apertura o el cierre de una etiqueta, se pueden usar los métodos `Html::beginTag()` y `Html::endTag()`.

Las opciones se usan en muchos métodos de la clase auxiliar Html y en varios widgets. En todos estos casos hay cierta gestión adicional que se debe conocer:

- Si un valor es `null`, el correspondiente atributo no se renderizará.
- Los atributos cuyos valores son de tipo booleano serán tratados como atributos booleanos[6].
- Los valores de los atributos se codificarán en HTML usando `Html::encode()`.
- El atributo "data" puede recibir un array. En este caso, se "expandirá" y se renderizará una lista de atributos `data` ej. `'data' => ['id' => 1, 'name' => 'yii']` se convierte en `data-id="1" data-name="yii"`.
- El atributo "data" puede recibir un JSON. Se gestionará de la misma manera que un array ej. `'data' => ['params' => ['id' => 1, 'name' => 'yii'], 'status' => 'ok']` se convierte en `data-params='{"id":1,"name":"yii"}' data-status="ok"`.

Formación de clases y estilos dinámicamente

Cuando se construyen opciones para etiquetas HTML, a menudo nos encontramos con valores predeterminados que hay que modificar. Para añadir o eliminar clases CSS se puede usar el siguiente ejemplo:

```
$options = ['class' => 'btn btn-default'];

if ($type === 'success') {
    Html::removeCssClass($options, 'btn-default');
    Html::addCssClass($options, 'btn-success');
}

echo Html::tag('div', 'Pwede na', $options);

// cuando $type sea 'success' se renderizará
// <div class="btn btn-success">Pwede na</div>
```

Para hacer lo mismo con los estilos para el atributo `style`:

```
$options = ['style' => ['width' => '100px', 'height' => '100px']];

// devuelve style="width: 100px; height: 200px; position: absolute;"
Html::addCssStyle($options, 'height: 200px; positon: absolute;');

// devuelve style="position: absolute;"
Html::removeCssStyle($options, ['width', 'height']);
```

[6] https://html.spec.whatwg.org/multipage/common-microsyntaxes.html#boolean-attributes

Cuando se usa `addCssStyle()` se puede especificar si un array de pares clave-valor corresponde a nombres y valores de la propiedad CSS correspondiente o a una cadena de texto como por ejemplo `width: 100px; height: 200px;`. Estos formatos se pueden "hacer" y "deshacer" usando `cssStyleFromArray()` y `cssStyleToArray()`. El método `removeCssStyle()` acepta un array de propiedades que se eliminarán. Si sólo se eliminara una propiedad, se puede especificar como una cadena de texto.

16.3.2. Codificación y Decodificación del contenido

Para que el contenido se muestre correctamente y de forma segura con caracteres especiales HTML el contenido debe ser codificado. En PHP esto se hace con htmlspecialchars[7] y htmlspecialchars_decode[8]. El problema con el uso de estos métodos directamente es que se tiene que especificar la codificación y opciones extra cada vez. Ya que las opciones siempre son las mismas y la codificación debe coincidir con la de la aplicación para prevenir problemas de seguridad, Yii proporciona dos métodos simples y compactos:

```
$userName = Html::encode($user->name);
echo $userName;

$decodedUserName = Html::decode($userName);
```

16.3.3. Formularios

El trato con el marcado de formularios es una tarea repetitiva y propensa a errores. Por esto hay un grupo de métodos para ayudar a gestionarlos.

> **Nota:** hay que considerar la opción de usar `ActiveForm` en caso de que se gestionen formularios que requieran validaciones.

Creando formularios

Se puede abrir un formulario con el método `beginForm()` como se muestra a continuación:

```
<?= Html::beginForm(['order/update', 'id' => $id], 'post', ['enctype' =>
'multipart/form-data']) ?>
```

El primer argumento es la URL a la que se enviarán los datos del formulario. Se puede especificar en formato de ruta de Yii con los parámetros aceptados por `Url::to()`. El segundo es el método que se usará. `post` es el método predeterminado. El tercero es un array de opciones para la etiqueta `form`. En este caso cambiamos el método de codificación del formulario de `data` en

[7] https://www.php.net/manual/es/function.htmlspecialchars.php
[8] https://www.php.net/manual/es/function.htmlspecialchars-decode.php

una petición POST a `multipart/form-data`. Esto se requiere cuando se quieren subir archivos.

El cierre de la etiqueta `form` es simple:

```
<?= Html::endForm() ?>
```

Botones

Para generar botones se puede usar el siguiente código:

```
<?= Html::button('Press me!', ['class' => 'teaser']) ?>
<?= Html::submitButton('Submit', ['class' => 'submit']) ?>
<?= Html::resetButton('Reset', ['class' => 'reset']) ?>
```

El primer argumento para los tres métodos es el título del botón y el segundo son las opciones. El título no está codificado pero si se usan datos recibidos por el usuario, deben codificarse mediante `Html::encode()`.

Inputs

Hay dos grupos en los métodos input. Unos empiezan con `active` y se llaman inputs activos y los otros no empiezan así. Los inputs activos obtienen datos del modelo y del atributo especificado y los datos de los inputs normales se especifica directamente.

Los métodos más genéricos son:

```
type, input name, input value, options
<?= Html::input('text', 'username', $user->name, ['class' => $username]) ?>

type, model, model attribute name, options
<?= Html::activeInput('text', $user, 'name', ['class' => $username]) ?>
```

Si se conoce el tipo de input de antemano, es conveniente usar los atajos de los métodos:

- `yii\helpers\Html::buttonInput()`
- `yii\helpers\Html::submitInput()`
- `yii\helpers\Html::resetInput()`
- `yii\helpers\Html::textInput()`, `yii\helpers\Html::activeTextInput()`
- `yii\helpers\Html::hiddenInput()`, `yii\helpers\Html::activeHiddenInput()`
- `yii\helpers\Html::passwordInput()` / `yii\helpers\Html::activePasswordInput()`
- `yii\helpers\Html::fileInput()`, `yii\helpers\Html::activeFileInput()`
- `yii\helpers\Html::textarea()`, `yii\helpers\Html::activeTextarea()`

Los botones de opción (Radios) y las casillas de verificación (checkboxes) se especifican de forma un poco diferente:

```
<?= Html::radio('agree', true, ['label' => 'I agree']);
<?= Html::activeRadio($model, 'agree', ['class' => 'agreement'])

<?= Html::checkbox('agree', true, ['label' => 'I agree']);
<?= Html::activeCheckbox($model, 'agree', ['class' => 'agreement'])
```

Las listas desplegables (dropdown list) se pueden renderizar como se muestra a continuación:

```
<?= Html::dropDownList('list', $currentUserId, ArrayHelper::map($userModels,
'id', 'name')) ?>
<?= Html::activeDropDownList($users, 'id', ArrayHelper::map($userModels,
'id', 'name')) ?>

<?= Html::listBox('list', $currentUserId, ArrayHelper::map($userModels,
'id', 'name')) ?>
<?= Html::activeListBox($users, 'id', ArrayHelper::map($userModels, 'id',
'name')) ?>
```

El primer argumento es el nombre del input, el segundo es el valor seleccionado actualmente y el tercero es el array de pares clave-valor donde la clave es la lista de valores y el valor del array es la lista a mostrar.

Si se quiere habilitar la selección múltiple, se puede usar la lista seleccionable (checkbox list):

```
<?= Html::checkboxList('roles', [16, 42], ArrayHelper::map($roleModels,
'id', 'name')) ?>
<?= Html::activeCheckboxList($user, 'role', ArrayHelper::map($roleModels,
'id', 'name')) ?>
```

Si no, se puede usar la lista de opciones (radio list):

```
<?= Html::radioList('roles', [16, 42], ArrayHelper::map($roleModels, 'id',
'name')) ?>
<?= Html::activeRadioList($user, 'role', ArrayHelper::map($roleModels,
'id', 'name')) ?>
```

Etiquetas y Errores

De forma parecida que en los inputs hay dos métodos para generar etiquetas. El activo que obtiene los datos del modelo y el no-activo que acepta los datos directamente:

```
<?= Html::label('User name', 'username', ['class' => 'label username']) ?>
<?= Html::activeLabel($user, 'username', ['class' => 'label username'])
```

Para mostrar los errores del formulario de un modelo o modelos a modo de resumen puedes usar:

```
<?= Html::errorSummary($posts, ['class' => 'errors']) ?>
```

Para mostrar un error individual:

```
<?= Html::error($post, 'title', ['class' => 'error']) ?>
```

Input Names y Values

Existen métodos para obtener names, IDs y values para los campos de entrada (inputs) basados en el modelo. Estos se usan principalmente internamente pero a veces pueden resultar prácticos:

```
// Post[title]
echo Html::getInputName($post, 'title');

// post-title
echo Html::getInputId($post, 'title');

// mi primer post
echo Html::getAttributeValue($post, 'title');

// $post->authors[0]
echo Html::getAttributeValue($post, '[0]authors[0]');
```

En el ejemplo anterior, el primer argumento es el modelo y el segundo es un atributo de expresión. En su forma más simple es su nombre de atributo pero podría ser un nombre de atributo prefijado y/o añadido como sufijo con los indices de un array, esto se usa principalmente para mostrar inputs en formatos de tablas:

- `[0]content` se usa en campos de entrada de datos en formato de tablas para representar el atributo "content" para el primer modelo del input en formato de tabla;
- `dates[0]` representa el primer elemento del array del atributo "dates";
- `[0]dates[0]` representa el primer elemento del array del atributo "dates" para el primer modelo en formato de tabla.

Para obtener el nombre de atributo sin sufijos o prefijos se puede usar el siguiente código:

```
// dates
echo Html::getAttributeName('dates[0]');
```

16.3.4. Estilos y scripts

Existen dos métodos para generar etiquetas que envuelvan estilos y scripts incrustados (embebbed):

```
<?= Html::style('.danger { color: #f00; }') ?>

Genera

<style>.danger { color: #f00; }</style>

<?= Html::script('alert("Hello!");', ['defer' => true]);

Genera

<script defer>alert("Hello!");</script>
```

Si se quiere enlazar un estilo externo desde un archivo CSS:

```
<?= Html::cssFile('@web/css/ie5.css', ['condition' => 'IE 5']) ?>
```

genera

```
<!--[if IE 5]>
    <link href="https://example.com/css/ie5.css" />
<![endif]-->
```

El primer argumento es la URL. El segundo es un array de opciones. Adicionalmente, para regular las opciones se puede especificar:

- `condition` para envolver `<link` con los comentarios condicionales con condiciones especificas. Esperamos que sean necesarios los comentarios condicionales ;)
- `noscript` se puede establecer como `true` para envolver `<link` con la etiqueta `<noscript>` por lo que el sólo se incluirá si el navegador no soporta JavaScript o si lo ha deshabilitado el usuario.

Para enlazar un archivo JavaScript:

```
<?= Html::jsFile('@web/js/main.js') ?>
```

Es igual que con las CSS, el primer argumento especifica el enlace al fichero que se quiere incluir. Las opciones se pueden pasar como segundo argumento. En las opciones se puede especificar `condition` del mismo modo que se puede usar para `cssFile`.

16.3.5. Enlaces

Existe un método para generar hipervínculos a conveniencia:

```
<?= Html::a('Profile', ['user/view', 'id' => $id], ['class' =>
'profile-link']) ?>
```

El primer argumento es el título. No está codificado por lo que si se usan datos enviados por el usuario se tienen que codificar usando `Html::encode()`. El segundo argumento es el que se introducirá en `href` de la etiqueta `<a`. Se puede consultar Url::to() para obtener más detalles de los valores que acepta. El tercer argumento es un array de las propiedades de la etiqueta.

Si se requiere generar enlaces de tipo `mailto` se puede usar el siguiente código:

```
<?= Html::mailto('Contact us', 'admin@example.com') ?>
```

16.3.6. Imagenes

Para generar una etiqueta de tipo imagen se puede usar el siguiente ejemplo:

```
<?= Html::img('@web/images/logo.png', ['alt' => 'My logo']) ?>
```

genera

```
<img src="https://example.com/images/logo.png" alt="My logo" />
```

Aparte de los alias el primer argumento puede aceptar rutas, parámetros y URLs. Del mismo modo que Url::to().

16.3.7. Listas

Las listas desordenadas se puede generar como se muestra a continuación:

```
<?= Html::ul($posts, ['item' => function($item, $index) {
    return Html::tag(
        'li',
        $this->render('post', ['item' => $item]),
        ['class' => 'post']
    );
}]) ?>
```

Para generar listas ordenadas se puede usar `Html::ol()` en su lugar.

16.4. Clase Auxiliar URL (URL Helper)

La clase auxiliar URL proporciona un conjunto de métodos estáticos para gestionar URLs.

16.4.1. Obtener URLs comúnes

Se pueden usar dos métodos para obtener URLs comunes: URL de inicio (home URL) y URL base (base URL) de la petición (request) actual. Para obtener la URL de inicio se puede usar el siguiente código:

```
$relativeHomeUrl = Url::home();
$absoluteHomeUrl = Url::home(true);
$httpsAbsoluteHomeUrl = Url::home('https');
```

Si no se pasan parámetros, la URL generada es relativa. Se puede pasar `true`para obtener la URL absoluta del esquema actual o especificar el esquema explícitamente (`https`, `http`).

Para obtener la URL base de la petición actual, se puede usar el siguiente código:

```php
$relativeBaseUrl = Url::base();
$absoluteBaseUrl = Url::base(true);
$httpsAbsoluteBaseUrl = Url::base('https');
```

El único parámetro del método funciona exactamente igual que para `Url::home()`.

16.4.2. Creación de URLs

Para crear una URL para una ruta determinada se puede usar `Url::toRoute()`. El método utiliza `yii\web\UrlManager` para crear la URL:

```php
$url = Url::toRoute(['product/view', 'id' => 42]);
```

Se puede especificar la ruta como una cadena de texto, ej. `site/index`. También se puede usar un array si se quieren especificar parámetros para la URL que se esta generando. El formato del array debe ser:

```php
// genera: /index.php?r=site%2Findex&param1=value1&param2=value2
['site/index', 'param1' => 'value1', 'param2' => 'value2']
```

Si se quiere crear una URL con un enlace, se puede usar el formato de array con el parámetro `#`. Por ejemplo,

```php
// genera: /index.php?r=site/index&param1=value1#name
['site/index', 'param1' => 'value1', '#' => 'name']
```

Una ruta puede ser absoluta o relativa. Una ruta absoluta tiene una barra al principio (ej. `/site/index`), mientras que una ruta relativa no la tiene (ej. `site/index` o `index`). Una ruta relativa se convertirá en una ruta absoluta siguiendo las siguientes reglas:

- Si la ruta es una cadena vacía, se usará la `route` actual;
- Si la ruta no contiene barras (ej. `index`), se considerará que es el ID de una acción del controlador actual y se antepondrá con `yii\web\Controller::$uniqueId`;
- Si la ruta no tiene barra inicial (ej. `site/index`), se considerará que es una ruta relativa del modulo actual y se le antepondrá el `uniqueId` del modulo.

Desde la versión 2.0.2, puedes especificar una ruta en términos de alias. Si este es el caso, el alias será convertido primero en la ruta real, la cual será entonces transformada en una ruta absoluta de acuerdo a las reglas mostradas arriba.

A continuación se muestran varios ejemplos del uso de este método:

```php
// /index.php?r=site%2Findex
echo Url::toRoute('site/index');

// /index.php?r=site%2Findex&src=ref1#name
echo Url::toRoute(['site/index', 'src' => 'ref1', '#' => 'name']);
```

```
// /index.php?r=post%2Fedit&id=100     asume que el alias "@postEdit" se
definió como "post/edit"
echo Url::toRoute(['@postEdit', 'id' => 100]);

// https://www.example.com/index.php?r=site%2Findex
echo Url::toRoute('site/index', true);

// https://www.example.com/index.php?r=site%2Findex
echo Url::toRoute('site/index', 'https');
```

El otro método `Url::to()` es muy similar a `toRoute()`. La única diferencia es que este método requiere que la ruta especificada sea un array. Si se pasa una cadena de texto, se tratara como una URL.

El primer argumento puede ser:

- un array: se llamará a `toRoute()` para generar la URL. Por ejemplo: `['site/index']`, `['post/index', 'page' => 2]`. Se puede revisar `toRoute()` para obtener más detalles acerca de como especificar una ruta.
- una cadena que empiece por `@`: se tratará como un alias, y se devolverá la cadena correspondiente asociada a este alias.
- una cadena vacía: se devolverá la URL de la petición actual;
- una cadena de texto: se devolverá sin alteraciones.

Cuando se especifique `$schema` (tanto una cadena de text como `true`), se devolverá una URL con información del host (obtenida mediante `yii\web\UrlManager::$hostInfo`). Si `$url` ya es una URL absoluta, su esquema se reemplazará con el especificado.

A continuación se muestran algunos ejemplos de uso:

```
// /index.php?r=site%2Findex
echo Url::to(['site/index']);

// /index.php?r=site%2Findex&src=ref1#name
echo Url::to(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post%2Fedit&id=100     asume que el alias "@postEdit" se
definió como "post/edit"
echo Url::to(['@postEdit', 'id' => 100]);

// the currently requested URL
echo Url::to();

// /images/logo.gif
echo Url::to('@web/images/logo.gif');

// images/logo.gif
echo Url::to('images/logo.gif');

// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', true);
```

```
// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', 'https');
```

Desde la versión 2.0.3, puedes utilizar `yii\helpers\Url::current()` para crear una URL a partir de la ruta solicitada y los parámetros GET. Puedes modificar o eliminar algunos de los parámetros GET, o también agregar nuevos pasando un parámetro `$params` al método. Por ejemplo,

```
// asume que $_GET = ['id' => 123, 'src' => 'google'], la ruta actual es
"post/view"

// /index.php?r=post%2Fview&id=123&src=google
echo Url::current();

// /index.php?r=post%2Fview&id=123
echo Url::current(['src' => null]);
// /index.php?r=post%2Fview&id=100&src=google
echo Url::current(['id' => 100]);
```

16.4.3. Recordar URLs

Hay casos en que se necesita recordar la URL y después usarla durante el procesamiento de una de las peticiones secuenciales. Se puede logar de la siguiente manera:

```
// Recuerda la URL actual
Url::remember();

// Recuerda la URL especificada. Revisar Url::to() para ver formatos de
argumentos.
Url::remember(['product/view', 'id' => 42]);

// Recuerda la URL especificada con un nombre asignado
Url::remember(['product/view', 'id' => 42], 'product');
```

En la siguiente petición se puede obtener la URL memorizada de la siguiente manera:

```
$url = Url::previous();
$productUrl = Url::previous('product');
```

16.4.4. Chequear URLs relativas

Para descubrir si una URL es relativa, es decir, que no contenga información del host, se puede utilizar el siguiente código:

```
$isRelative = Url::isRelative('test/it');
```