

PROGRAMACIÓN WEB

Manual CODEIGNITER



Autor:

Alejandro Guerrero Medina

Grado en Ingeniería Informática

Curso 2022 - 2023

Índice

1. Prolegómeno	2
2. Instalación de CodeIgniter	2
2.1. Pasos Previos	2
2.2. Instalación de Composer	2
2.3. Instalación de CodeIgniter	3
2.4. Primer Arranque	3
3. Controladores	5
3.1. Un primer controlador	5
3.2. Páginas estáticas	6
3.3. Enrutamiento del Controlador Pages	10
3.4. Funcionamiento de los controladores	11
4. Base de Datos en CodeIgniter	12
4.1. Creación de la Base de Datos	12
4.2. Conexión con la base de datos	13
4.3. Configuración del Modelo	14
4.4. Mostrar las noticias	15
4.5. Enrutamiento del Controlador View	21
5. Inserción de nuevos objetos	23
5.1. Filtro de protección CSRF	23
5.2. Creación de un formulario	23
5.2.1. Vista	23
5.2.2. Controlador	25
5.3. Actualizando el modelo	28
5.4. Actualizando el enrutamiento	29
6. Conclusiones	31

1. Prolegómeno

Sea este documento una guía sobre el *framework* basado en PHP, CODEIGNITER, con objeto de especificar cada paso realizado para desplegar una aplicación web en correcto funcionamiento siguiendo la documentación oficial de la misma herramienta.

2. Instalación de CodeIgniter

Existen dos métodos para instalar CODEIGNITER, ellos son:

- Instalación manual.
- Mediante el manejador de dependencias COMPOSER. **En esta guía se realizará con esta opción.**

2.1. Pasos Previos

Primero que todo, será necesario disponer en su equipo de una versión igual o superior a PHP 7.4 con, al menos, las siguientes extensiones habilitadas –habrá de modificarse el archivo `php.ini`–:

- `intl`
- `mbstring`
- `json`

También existen extensiones opcionales a habilitar, como `gd` o `curl`, ambas habilitadas durante el proceso.

2.2. Instalación de Composer

El manejador de dependencias para PHP, COMPOSER, lo podemos encontrar en su página oficial. Con descargar la última versión disponible –versión 2.5.4 a fecha de resolución de este documento– e instalarla en el directorio donde tengamos localizado nuestro PHP –en esta guía usamos XAMPP–, sería suficiente.

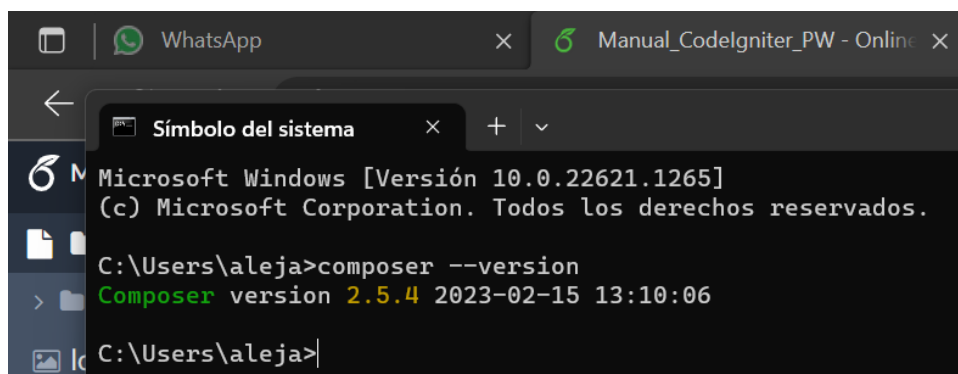


Figura 1: COMPOSER Instalado.

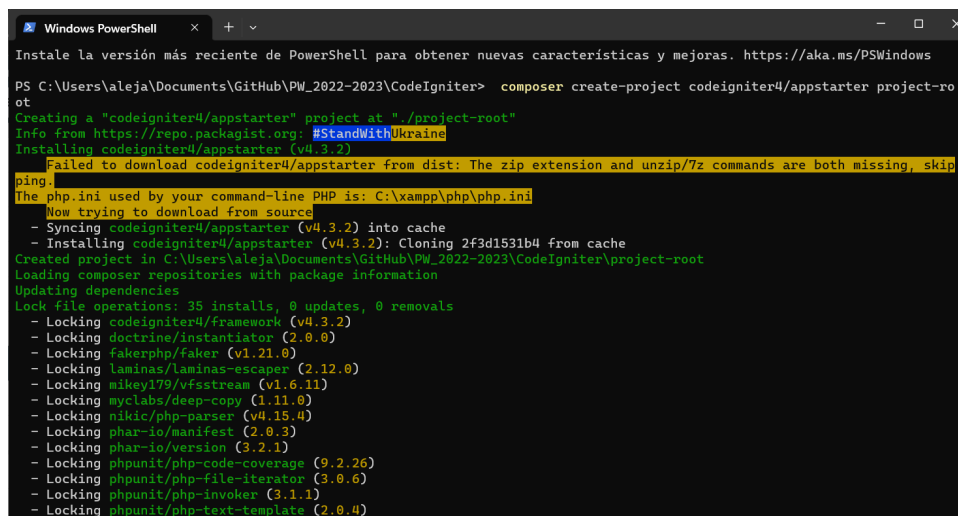
2.3. Instalación de CodeIgniter

La versión del *framework* a usar será CODEIGNITER4, la cuál podremos encontrar con facilidad en el repositorio de *GitHub* publicado en la página oficial de CODEIGNITER, en la sección de instalación.

A continuación, nos dirigiremos a un directorio en el que instalaremos el *framework* e introduciremos la siguiente línea:

```
$ composer create-project codeigniter4/appstarter project-root
```

Lo que nos crearía automáticamente el repositorio mencionado en la carpeta '**project-root**'.



```
Windows PowerShell
Instale la versión más reciente de PowerShell para obtener nuevas características y mejoras. https://aka.ms/PSWindows
PS C:\Users\aleja\Documents\GitHub\PW_2022-2023\CodeIgniter> composer create-project codeigniter4/appstarter project-root
Creating a "codeigniter4/appstarter" project at "./project-root"
Info from https://repo.packagist.org: #StandWithUkraine
Installing codeigniter4/appstarter (v4.3.2)
Failed to download codeigniter4/appstarter from dist: The zip extension and unzip/7z commands are both missing, skip ping.
The php.ini used by your command-line PHP is: C:\xampp\php\php.ini
Now trying to download from source
- Syncing codeigniter4/appstarter (v4.3.2) into cache
- Installing codeigniter4/appstarter (v4.3.2): Cloning 2f3d1531b4 from cache
Created project in C:\Users\aleja\Documents\GitHub\PW_2022-2023\CodeIgniter\project-root
Loading composer repositories with package information
Updating dependencies
Lock file operations: 35 installs, 0 updates, 0 removals
- Locking codeigniter4/framework (v4.3.2)
- Locking doctrine/instantiator (2.0.0)
- Locking fakerphp/faker (v1.21.0)
- Locking laminas/laminas-escaper (2.12.0)
- Locking mikew179/vfsstream (v1.6.11)
- Locking myclabs/deep-copy (1.11.0)
- Locking nikic/php-parser (v4.15.4)
- Locking phar-io/manifest (2.0.3)
- Locking phar-io/version (3.2.1)
- Locking phpunit/php-code-coverage (9.2.26)
- Locking phpunit/php-file-iterator (3.0.6)
- Locking phpunit/php-invoker (3.1.1)
- Locking phpunit/php-text-template (2.0.4)
```

Figura 2: Instalación de CODEIGNITER.

Se recomienda, una vez terminada la instalación, ejecutar el comando `composer update` dentro de dicho directorio en caso de que exista alguna dependencia que tenga actualizaciones pendientes y no se hayan efectuado durante el proceso de instalación.

2.4. Primer Arranque

Podemos ajustar los parámetros de nuestra aplicación web en los archivos localizados en el directorio desplegado: `app/Config/App.php` para establecer la URL del servidor –en esta guía se dejará tal y como viene por defecto, es decir, `'http://localhost:8080'`– y `app/Config/Database.php` en caso de que se desee modificar la configuración de la base de datos.

Vamos a desplegar el servidor para visualizar la página de bienvenida a CODEIGNITER. Para ello, tan solo deberemos situarnos en el directorio donde tengamos nuestro *framework* instalado y ejecutar en una terminal el siguiente comando:

```
$ php spark serve
```

En la siguiente instantánea se visualiza el contenido de la URL establecida:

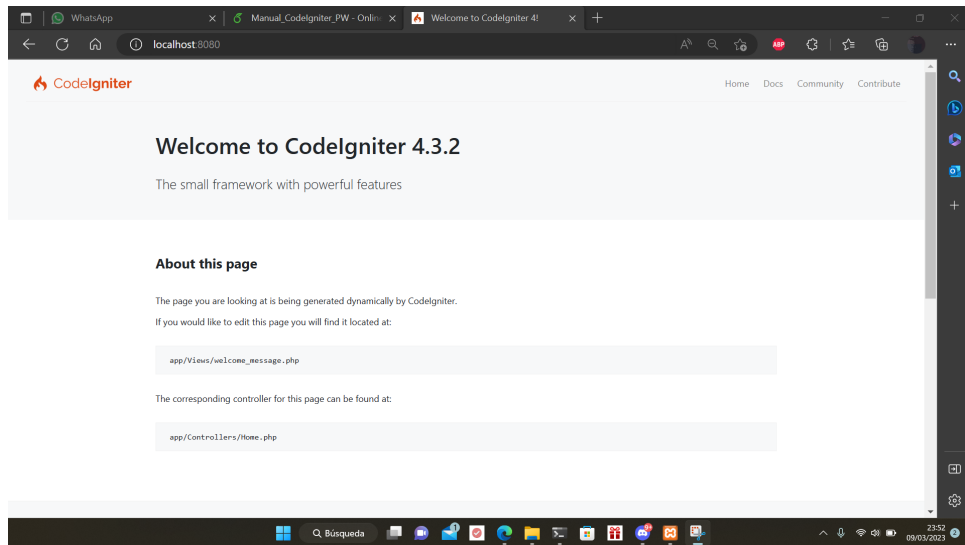


Figura 3: Página de bienvenida a CODEIGNITER.

Ya tenemos nuestro servidor funcionando, lo que significa que podremos trabajar sobre cada uno de los archivos proporcionados por el repositorio ofrecido de CODEIGNITER para personalizarlo a nuestro gusto.

En la siguiente sección veremos como manejar páginas estáticas por medio de los denominados **controladores**.

3. Controladores

Un **controlador** es una parte importante del tan utilizado patrón de diseño *Modelo-Vista-Controlador* o *MVC*, el cual se utiliza comúnmente en el desarrollo web. En este patrón, el **controlador** actúa como un intermediario entre la **vista** –que muestra la información al usuario– y el **modelo** –que maneja los datos subyacentes–.

En implementaciones de PHP, un **controlador** se trata de una clase la cual manipula una solicitud `http` y devuelve generalmente vistas al usuario en función de la solicitud enviada.

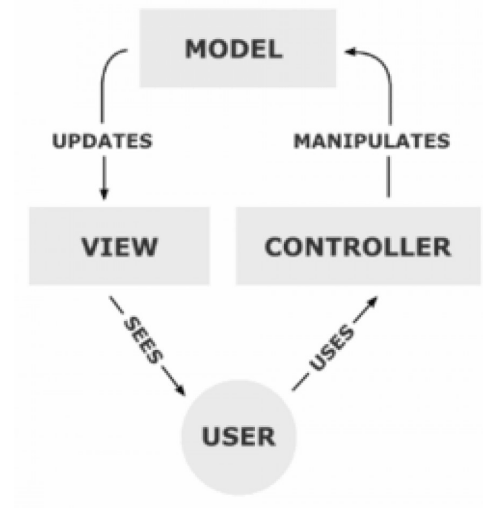


Figura 4: Estructura del *Modelo-Vista-Controlador*

3.1. Un primer controlador

Vamos a crear el **controlador** `Pages.php` en el directorio `'app/Controllers/'`, con el siguiente código incluido en él:

```
1 <?php
2
3 namespace App\Controllers;
4
5 class Pages extends BaseController
6 {
7     public function index()
8     {
9         return view('welcome_message');
10    }
11
12    public function view($page = 'home')
13    {
14        // ...
15    }
16 }
```

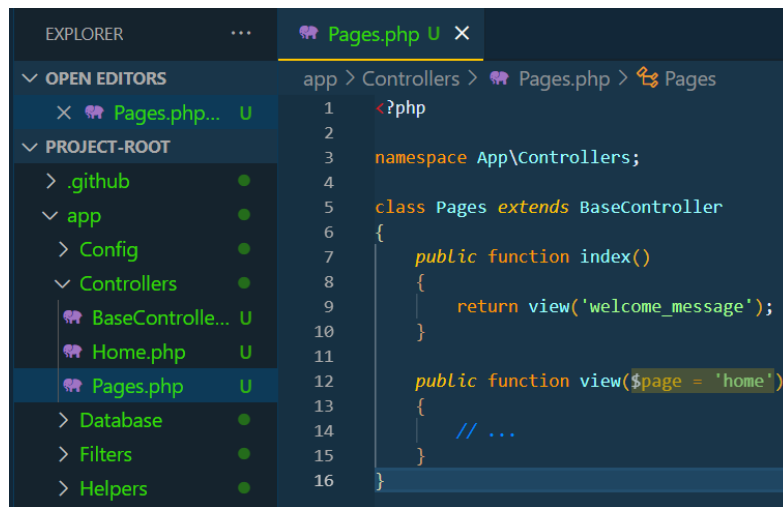


Figura 5: Controlador Pages.php

Este **controlador** tiene como objetivo mostrarnos la página de bienvenida de CODEIGNITER a través del método `index()`, el cual se encuentra implementado en `app/Controllers/Home.php`. A su vez, incluye un método `view()` que acepta un argumento llamado `$page`; dicho argumento se refiere a la página que debe cargar.

La clase `Pages` hereda de la clase `BaseController`, que a su vez esta hereda de la clase `CodeIgniter \Controller`, posibilitándose el acceso de los métodos implementados en la clase `CodeIgniter \Controller` desde la clase `Pages`.

En toda instancia, un **controlador** será aquello que gestione cada petición en la aplicación web. Para referirnos a ella, y siendo PHP, usaremos la variable `$this`.

3.2. Páginas estáticas

A continuación, añadiremos una **plantilla** para crear una página estática que incluya una **cabecera**, o *header*, y un **pie de página**, o *footer*.

Creemos un fichero en `'app/Views/templates'` llamado `header.php`:

```
1 <!doctype html>
2 <html>
3 <head>
4     <title>CodeIgniter Tutorial</title>
5 </head>
6 <body>
7
8     <h1><?= esc($title) ?></h1>
9
10 </body>
11 </html>
```

Contiene código básico HTML que mostrará un encabezado con la vista principal.

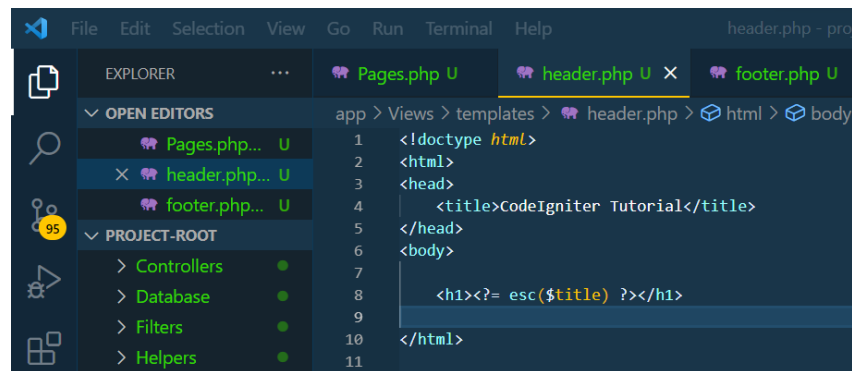


Figura 6: Contenido de header.php

- La variable \$title hace uso de la función `esc()`; una función de ámbito global de CODEIGNITER que previene ataques XSS.

Seguidamente, creamos un fichero en 'app/Views/templates' llamado footer.php:

```

1 <!doctype html>
2 <html>
3 <body>
4
5   <em>&copy; 2023</em>
6
7 </body>
8 </html>

```

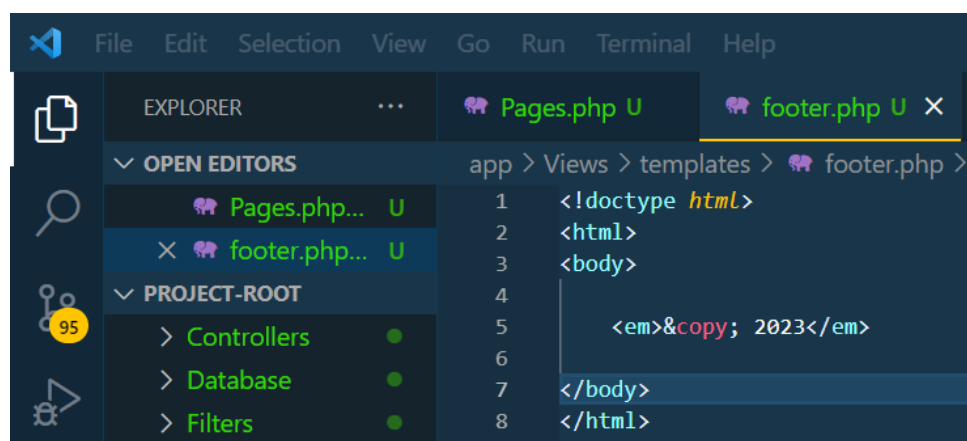


Figura 7: Contenido de footer.php

Ahora regresaremos al **controlador** Pages previamente creado para añadir más funcionalidades, no sin antes agregar en el directorio 'app/Views/pages/' los ficheros home.php y about.php, unos ficheros de ejemplo donde añadiremos texto arbitrario que nos servirá para probar las funcionalidades que a continuación implementaremos en el **controlador** Pages:

about.php

```
1 <!doctype html>
2 <html>
3 <head>
4     <title>about.php</title>
5 </head>
6 <body>
7
8     <h1>Hola, has accedido a 'about.php'</h1>
9
10 </body>
11 </html>
```

home.php

```
1 <!doctype html>
2 <html>
3 <head>
4     <title>home.php</title>
5 </head>
6 <body>
7
8     <h1>Hola, has accedido a 'home.php'</h1>
9
10 </body>
11 </html>
```

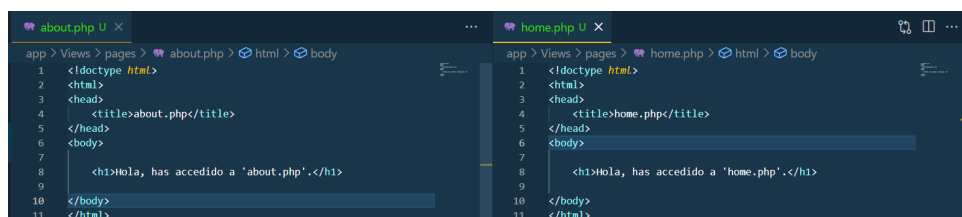


Figura 8: Contenido de about.php y home.php

No obstante, en caso de que queramos que dichas páginas se carguen, tendremos que comprobar que la petición de acceso a esa página existe. Agregamos el siguiente código al **controlador** Pages.php:

```
1 <?php
2
3 namespace App\Controllers;
4
5 use CodeIgniter\Exceptions\PageNotFoundException; // Para manejar excepciones.
6
7 class Pages extends BaseController
8 {
```

```
9      public function index()
10      {
11          return view('welcome_message');
12      }
13
14      public function view($page = 'home')
15      {
16          if (! is_file(APPPATH . 'Views/pages/' . $page . '.php')) {
17              // No se encuentra una vista disponible para esa petición.
18              throw new PageNotFoundException($page);
19          }
20
21          $data['title'] = ucfirst($page); // Convierte en mayuscula la primera letra
22          .
23
24          return view('templates/header', $data)
25              . view('pages/' . $page)
26              . view('templates/footer');
27      }
28  }
```

```
1  <?php
2
3  namespace App\Controllers;
4
5  use CodeIgniter\Exceptions\PageNotFoundException; // Para manejar excepciones.
6
7  class Pages extends BaseController
8  {
9      public function index()
10      {
11          return view('welcome_message');
12      }
13
14      public function view($page = 'home')
15      {
16          if (! is_file(APPPATH . 'Views/pages/' . $page . '.php')) {
17              // No se encuentra una vista disponible para esa petición.
18              throw new PageNotFoundException($page);
19          }
20
21          $data['title'] = ucfirst($page); // Convierte en mayuscula la primera letra.
22
23          return view('templates/header', $data)
24              . view('pages/' . $page)
25              . view('templates/footer');
26      }
27  }
```

Figura 9: Controlador pages.php actualizado

Con este código, si la página con la que se envía la petición existe, entonces se carga, incluyendo el *header* y el *footer* previamente creados, mostrándolos al usuario. En caso contrario, mostrará el error “404 Page not found”.

- En el bucle `if` se comprueba si la página existe. La función nativa de PHP, `is_file()`, se usa para comprobar si el fichero especificado se encuentra en la ruta especificada.
- La excepción `PageNotFoundException` es una implementada por CODEIGNITER cuyo propósito es mostrar la página de error por defecto.

- La variable `$data` contiene el valor que contenía la variable `$title` en la plantilla del *header*, ahora asignado como elemento del vector.
- Por último, cada una de las vistas creadas se muestran en orden, usando la función `view()` de CODEIGNITER.

3.3. Enrutamiento del Controlador Pages

Tras haber creado el **controlador** o **controladores** que hayamos deseado para nuestra aplicación web, deberemos de establecer las reglas de **enrutamiento**. El **enrutamiento** se encarga de asociar una URL con un método **controlador**.

Para ello, localizamos el fichero alojado en `'app/Config/Routes.php'` y nos fijamos en la sección denominada `'Route Definitions'`. La directiva existente funciona de tal forma que cualquier petición cuyo contenido no esté especificado será directamente manejado por el método `index()`, ubicado dentro del **controlador** `Home`.

Añadimos las siguientes líneas de código a `Routes.php`:

```
1 use App\Controllers\Pages;
2
3 // Agregar despues de "$routes->get('/', 'Home::index');"
4 $routes->get('pages', [Pages::class, 'index']);
5 $routes->get('(:segment)', [Pages::class, 'view']);
```

```
3 namespace Config;
4
5 use App\Controllers\Pages; // Añadida.
6
7 // Create a new instance of our RouteCollection class.
8 $routes = Services::routes();
9
10 /*
11  * -----
12  * Router Setup
13  * -----
14  */
15 $routes->setDefaultNamespace('App\Controllers');
16 $routes->setDefaultController('Home');
17 $routes->setDefaultMethod('index');
18 $routes->setTranslateURIDashes(false);
19 $routes->set404Override();
20 // The Auto Routing (Legacy) is very dangerous. It is easy to create vulnerable apps
21 // where controller filters or CSRF protection are bypassed.
22 // If you don't want to define all routes, please use the Auto Routing (Improved).
23 // Set '$autoRoutesImproved' to true in 'app/Config/Feature.php' and set the following to true.
24 // $routes->setAutoRoute(false);
25
26 /*
27  * -----
28  * Route Definitions
29  * -----
30  */
31
32 // We get a performance increase by specifying the default
33 // route since we don't have to scan directories.
34 $routes->get('/', 'Home::index');
35 $routes->get('pages', [Pages::class, 'index']); // Añadida.
36 $routes->get('(:segment)', [Pages::class, 'view']); //Añadida.
```

Figura 10: Enrutamiento de Pages

CODEIGNITER lee sus reglas de enrutamiento de arriba a abajo y enruta la petición a la primera regla que coincida. Cuando llega una petición, CODEIGNITER busca la primera coincidencia, y llama al **controlador** y método apropiados.

3.4. Funcionamiento de los controladores

Ahora, lanzaremos un servidor de prueba donde comprobaremos si la asignación de rutas del **controlador** creado y las plantillas de vista `header.php` y `footer.php` funcionan.

Para ello, abrimos una terminal de comandos en nuestro directorio raíz del proyecto e introducimos la siguiente línea de comando:

```
$ php spark serve
```

Dicho comando iniciará un servidor web, accesible desde el puerto 8080. Si se dejaron los campos por defecto, es decir, `localhost:8080`, debería de visualizarse la página de inicio de CODEIGNITER. Para acceder a las distintas vistas, en la barra de búsqueda, escriba, por ejemplo, `localhost:8080/home`, debería de mostrarse la siguiente página:

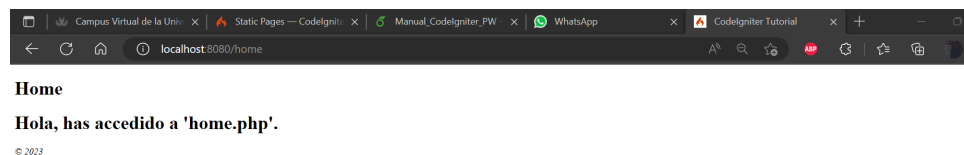


Figura 11: Vista web home.php

De igual forma, con `localhost:8080/about`, podremos visualizar el fichero `about.php` que también hemos creado.

En la siguiente sección, veremos como crear una **Base de Datos** y como conectarnos a ella.

4. Base de Datos en CodeIgniter

Antes de empezar a manejar nuestra primera base de datos, debemos tener unas consideraciones en cuenta, como:

- La **base de datos** que vayamos a crear deberá de tener compatibilidad con CODEIGNITER.
- Del mismo modo, el cliente que usemos deberá de ser adecuado para lanzar comandos. Por ejemplo, PHPMYADMIN.

4.1. Creación de la Base de Datos

A continuación, crearemos una base de datos de nombre `ci4tutorial` desde PHPMYADMIN con una tabla llamada `news`. Quedaría algo así:

#	Nombre	Tipo	Cotejamiento	Atributos	Nulo	Predeterminado	Comentarios	Extra	Acción
1	id	int(10)	utf8mb4_general_ci	unsigned	No	Ninguna		AUTO_INCREMENT	Cambiar Eliminar Más
2	title	varchar(128)	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más
3	slug	varchar(128)	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más
4	body	text	utf8mb4_general_ci		No	Ninguna			Cambiar Eliminar Más

Figura 12: Visualización de la tabla news

De igual forma, debemos añadir tuplas de datos para tener algo contenido en esta. CODEIGNITER ofrece la siguiente tupla de datos para trabajar con ella:

```

1 INSERT INTO news VALUES
2 (1,'Elvis sighted','elvis-sighted','Elvis was sighted at the Podunk internet cafe.
   It looked like he was writing a CodeIgniter app. '),
3 (2,'Say it isn\'t so!','say-it-isnt-so','Scientists conclude that some programmers
   have a sense of humor. '),
4 (3,'Caffeination, Yes!','caffeination-yes','World\'s largest coffee shop open
   onsite nested coffee shop for staff only. ');
  
```

4.2. Conexión con la base de datos

Nos dirigiremos a nuestro directorio raíz de CODEIGNITER y abriremos el archivo anteriormente modificado `.env` y asignaremos como **base de datos** predeterminada `ci4tutorial`.

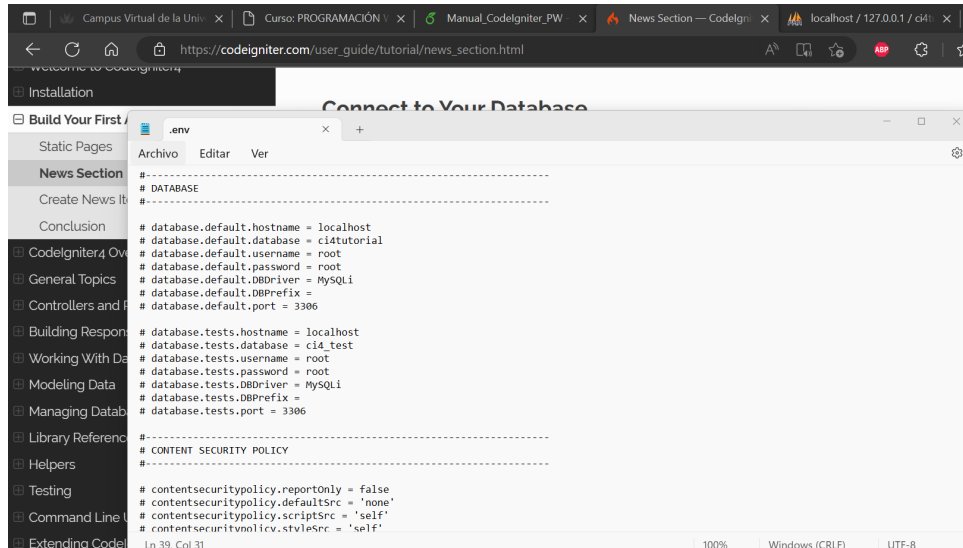


Figura 13: Fichero `.env` con base de datos predeterminada `ci4tutorial`

IMPORTANTE: En el fichero `app/Config/Database.php`, así como en el fichero `.env`, deberemos de establecer la configuración de la base de datos tal y como viene en el siguiente fragmento de código:

```
1 // ...
2
3 class Database extends Config
4 {
5     public $default = [
6         'DSN'          => '',
7         'hostname'     => 'localhost',
8         'username'     => 'root',
9         'password'     => '',
10        'database'     => 'ci4tutorial',
11        'DBDriver'     => 'MySQLi',
12        'DBPrefix'     => '',
13        // ...
14        'port'         => '3306',
15    ];
16
17    // ...
18 }
```

- Importante dejar el campo `password` vacío. Durante el desarrollo del manual, debido a que tenía dicho campo asignado con valor: `root`, CODEIGNITER informaba de un error parecido al siguiente:

```
Unable to connect to the database.Main connection [MySQLi]: Access
```

```
denied for user @ (using password: YES)
```

4.3. Configuración del Modelo

Un **Modelo** es aquella parte del *Modelo-Vista-Controlador* donde se **recupera, inserta y actualiza** la información de la base de datos –aunque puede hacerlo para más bases de datos–. En pocas palabras, proporcionan acceso a los datos.

En el directorio `app/Models/`, crearemos el fichero `NewsModel.php` y añadiremos el siguiente código:

```
1 <?php
2
3 namespace App\Models;
4 use CodeIgniter\Model;
5
6 class NewsModel extends Model
7 {
8     protected $table = 'news';
9 }
```

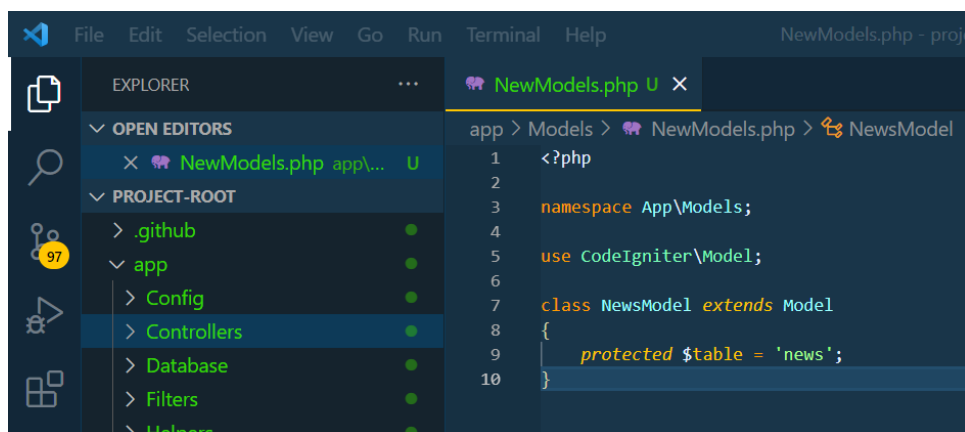


Figura 14: Fichero `NewsModel.php`

En pocas palabras, crea un **modelo** que parte del incorporado en CODEIGNITER y carga la librería de la base de datos. De esta forma podremos acceder a la clase de la base de datos mediante el objeto `$this->db`.

Aun así, necesitaremos de un método con el que podamos acceder a cada dato de nuestra base de datos. Existe un capa de abstracción incorporada en CODEIGNITER –*Query Builder*– que posibilita escribir consultas en cada sistema compatible de base de datos con el *framework*. Veamos algunas herramientas aplicadas:

```
1 public function getNews($slug = false)
2 {
3     if ($slug === false) {
4         return $this->findAll();
5     }
6
7     return $this->where(['slug' => $slug])->first();
8 }
```

```
1  <?php
2
3  namespace App\Models;
4
5  use CodeIgniter\Model;
6
7  class NewsModel extends Model
8  {
9      protected $table = 'news';
10
11     public function getNews($slug = false)
12     {
13         if ($slug === false) {
14             return $this->findAll();
15         }
16
17         return $this->where(['slug' => $slug])->first();
18     }
19 }
```

Figura 15: Función getNews

Con esta función, podremos realizar dos consultas distintas: una en la que capturemos todas las noticias, y otra que obtenga un objeto news por su *slug*.

Los métodos provistos por la clase `CodeIgniter\Model`, `findAll()` y `first()`, conocen que tabla deben usar, basada en la variable `$table` establecida en la clase `NewsModel`. *Query Builder* se ayuda de estos métodos para devolver un array de resultados en el formato que hayas elegido.

- En este caso, `findAll()` devuelve un array de arrays.

4.4. Mostrar las noticias

Ahora que tenemos las consultas escritas, el **modelo** debería estar vinculado a las **vistas** que mostrarán las noticias al usuario. Pese a que podemos realizar dicha acción con el **controlador** ya existente `Pages`, esta vez crearemos otro nuevo controlador llamado `News`. Lo creamos en `'app/Controllers/News.php'`:

```
1  <?php
2
3  namespace App\Controllers;
4
5  use App\Models\NewsModel;
6
7  class News extends BaseController
8  {
9      public function index()
10     {
11         $model = model(NewsModel::class);
12
13         $data['news'] = $model->getNews();
14     }
15 }
```



```
16     public function view($slug = null)
17     {
18         $model = model(NewsModel::class);
19
20         $data['news'] = $model->getNews($slug);
21     }
22 }
```

```
<?php

namespace App\Controllers;

use App\Models\NewsModel;

class News extends BaseController
{
    public function index()
    {
        $model = model(NewsModel::class);

        $data['news'] = $model->getNews();
    }

    public function view($slug = null)
    {
        $model = model(NewsModel::class);

        $data['news'] = $model->getNews($slug);
    }
}
```

Figura 16: Controlador News.php

Encontramos algunas similitudes con los ficheros creados anteriormente:

- Primero, se extiende el **controlador** `BaseController` que extiende una clase importante de CODEIGNITER, de nombre `Controller`, proporcionando métodos de ayuda y se asegura de que `News` tenga acceso a los objetos `Request` y `Response`, así como a la clase `Logger`, para guardar la información en el disco.
- Al igual que en el **modelo** creado previamente, existen dos métodos de semejante naturaleza: uno muestra todos los objetos y otro solo uno en específico.
- La función `model()` es una función de ayuda la cual crea una instancia de `NewsModel`. Una alternativa a esta función es usar `$model = new NewsModel();`
- La variable `$slug` la recibe el segundo método para identificar los objetos `news` que han de ser devueltos.

Sin embargo, todavía nuestra información no se está mostrando en la página –es decir, las **vistas** no están obteniendo esa información–, sino que están siendo recogidos por el **controlador**. Para ello, añadiremos las siguientes líneas de código al método `index()`, resultando tal que así:

```
1 <?php
2
3 namespace App\Controllers;
4
5 use App\Models\NewsModel;
6
7 class News extends BaseController
8 {
9     public function index()
10     {
11         $model = model(NewsModel::class);
12
13         $data = [
14             'news' => $model->getNews(),
15             'title' => 'News archive',
16         ];
17
18         return view('templates/header', $data)
19             . view('news/index')
20             . view('templates/footer');
21     }
22
23     // ...
24 }
```

```
class News extends BaseController
{
    public function index()
    {
        $model = model(NewsModel::class);

        $data = [
            'news' => $model->getNews(),
            'title' => 'News archive',
        ];

        return view('templates/header', $data)
            . view('news/index')
            . view('templates/footer');
    }
}
```

Figura 17: Controlador News.php

Con esto, se obtienen todos los registros de `news` del **modelo** y los asigna a una variable. El valor de `title` también se asigna al elemento `$data['title']` y todos los datos se pasan a las vistas.

Ahora, necesitamos crear una **vista** que muestre las noticias. Crea el fichero en `app/Views/news/index.php` y añade el siguiente fragmento de código:

```
1      <h2><?= esc($title) ?></h2>
2
3      <?php if (! empty($news) && is_array($news)): ?>
4
5          <?php foreach ($news as $news_item): ?>
6
7              <h3><?= esc($news_item['title']) ?></h3>
8
9              <div class="main">
10                  <?= esc($news_item['body']) ?>
11              </div>
12              <p><a href="/news/<?= esc($news_item['slug'], 'url') ?>">View article</
13                  a></p>
14
15          <?php endforeach ?>
16
17      <?php else: ?>
18
19          <h3>No News</h3>
20
21          <p>No se ha podido encontrar noticias.</p>
22
23      <?php endif ?>
```

```

<!doctype html>
<html>
<head>
    <title>index.php</title>
</head>
<body>

    <h2><?= esc($title) ?></h2>

    <?php if (! empty($news) && is_array($news)): ?>

        <?php foreach ($news as $news_item): ?>

            <h3><?= esc($news_item['title']) ?></h3>

            <div class="main">
                <?= esc($news_item['body']) ?>
            </div>
            <p><a href="/news/<?= esc($news_item['slug'], 'url') ?>">View article</a></p>

        <?php endforeach ?>

    <?php else: ?>

        <h3>No News</h3>

        <p>No se ha podido encontrar noticias.</p>

    <?php endif ?>

</body>
</html>

```

Figura 18: Vista index.php

Entramos en detalles:

- Cada noticia es reproducida en bucle y se muestra en la **vista**.
- En caso de que queramos una página para mostrar noticias individuales, solo deberíamos añadir unas cuántas líneas más de código al **controlador** y crear una **vista** adicional.

```

1  <?php
2
3  namespace App\Controllers;
4
5  use App\Models\NewsModel;
6  use CodeIgniter\Exceptions\PageNotFoundException;
7
8  class News extends BaseController
9  {
10     public function view($slug = null)
11     {
12         $model = model(NewsModel::class);
13
14         $data['news'] = $model->getNews($slug);
15
16         if (empty($data['news'])) {

```

```
17         throw new PageNotFoundException('No se ha podido encontrar
18         objetos news: ' . $slug);
19     }
20
21     $data['title'] = $data['news']['title'];
22
23     return view('templates/header', $data)
24         . view('news/view')
25         . view('templates/footer');
26 }
27 }
```

```
public function view($slug = null)
{
    $model = model(NewsModel::class);

    $data['news'] = $model->getNews($slug);

    if (empty($data['news'])) {
        throw new PageNotFoundException('No se ha podido encontrar objetos news: ' . $slug);
    }

    $data['title'] = $data['news']['title'];

    return view('templates/header', $data)
        . view('news/view')
        . view('templates/footer');
}
```

Figura 19: Método view en News.php actualizado

- Hay que añadir use `CodeIgniter\Exceptions\PageNotFoundException;` para importar la clase `PageNotFoundException`.

En vez de llamar al método `getNews()` sin parámetro, la variable `$slug` es pasada, de forma que devolverá el objeto `news` específico. Lo único que nos faltaría crear es la correspondiente **vista** en `'app/Views/news/view.php'`, teniendo el siguiente código incluido en él:

```
1 <h2><?= esc($news['title']) ?></h2>
2 <p><?= esc($news['body']) ?></p>
```

```
<!doctype html>
<html>
<head>
    <title>view.php</title>
</head>
<body>

    <h2><?= esc($news['title']) ?></h2>
    <p><?= esc($news['body']) ?></p>

</body>
</html>
```

Figura 20: Vista view.php

4.5. Enrutamiento del Controlador View

Al igual que hicimos con el **controlador** Pages, debemos asignar el enrutamiento para View. Añadir las siguientes líneas de código en el lugar correspondiente:

```
1 <?php
2
3 // ...
4
5 use App\Controllers\News;
6 use App\Controllers\Pages;
7
8 $routes->get('news/(:segment)', [News::class, 'view']);
9 $routes->get('news', [News::class, 'index']);
10 $routes->get('pages', [Pages::class, 'index']);
11 $routes->get('(:segment)', [Pages::class, 'view']);
12
13 // ...
```

```

use App\Controllers\News; // Añadida
use App\Controllers\Pages;

// Create a new instance of our RouteCollection class.
$route = Services::routes();

/*
 * Router Setup
 */
$route->setDefaultNamespace('App\Controllers');
$route->setDefaultController('Home');
$route->setDefaultMethod('index');
$route->setTranslateURIDashes(false);
$route->set404Override();
// The Auto Routing (Legacy) is very dangerous. It is easy to create vulnerable apps
// where controller filters or CSRF protection are bypassed.
// If you don't want to define all routes, please use the Auto Routing (Improved).
// Set 'autoRoutesImproved' to true in 'app/Config/Feature.php' and set the following to true.
$route->setAutoRoute(false);

/*
 * Route Definitions
 */

// We get a performance increase by specifying the default
// route since we don't have to scan directories.
$route->get('/', 'Home::index');
$route->get('news/{:segment}', [News::class, 'view']); // Añadida
$route->get('news', [News::class, 'index']); // Añadida
$route->get('pages', [Pages::class, 'index']);
$route->get('{:segment}', [Pages::class, 'view']);

```

Figura 21: Enrutamiento de View

Finalmente, comprobaremos si el **controlador** news funciona correctamente. Siguiendo los pasos anteriores para iniciar el servidor local, accediendo a localhost : 8080/news, vemos la siguiente **vista**:

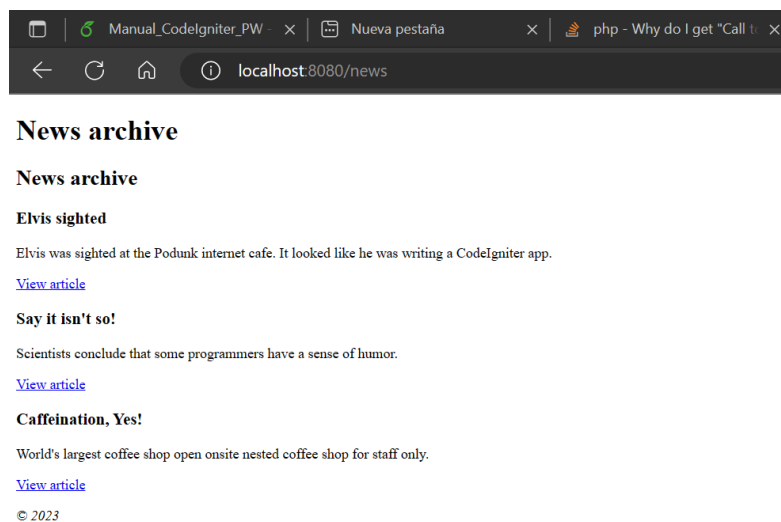


Figura 22: Vista web news

En el siguiente apartado, veremos como crear un formulario con el que podremos añadir objetos a la base de datos, además de mostrar en vistas la información dada.

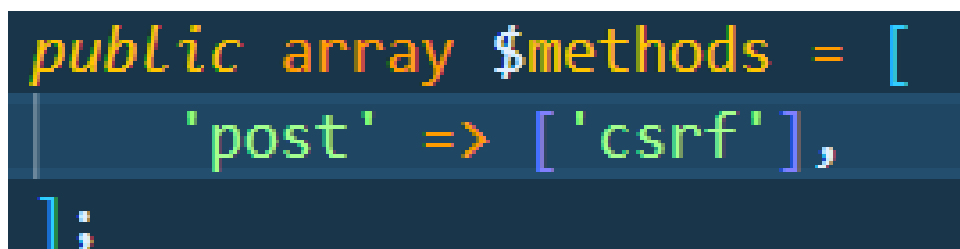
5. Inserción de nuevos objetos

A la hora de crear un formulario, conviene activar un filtro de **protección CSRF** –*Cross-Site Request Forgery*–, un tipo de ataque que ocurren en las páginas web cuando se da lugar a una acción inintencionada cuando un usuario se autentica.

5.1. Filtro de protección CSRF

Abrimos el fichero alojado en `'app/Config/Filters.php'` y actualizamos los atributos de `$methods` añadiendo lo siguiente:

```
1 <?php
2
3 namespace Config;
4
5 use CodeIgniter\Config\BaseConfig;
6
7 class Filters extends BaseConfig
8 {
9     // ...
10
11     public $methods = [
12         'post' => ['csrf'],
13     ];
14
15     // ...
16 }
```



```
public array $methods = [
    'post' => ['csrf'],
];
```

Figura 23: Filtro *CSRF* en `filters.php`

Con esto, establecemos que todas las peticiones **POST** tengan habilitado el filtro **CSRF**.

5.2. Creación de un formulario

5.2.1. Vista

Para introducir datos en la base de datos, deberemos crear un formulario en el que podamos introducir la información que ha de ser guardada. Esto significa que necesitaremos un formulario con dos campos:

- Un primer campo para el título.

- Un segundo campo para el texto.

Crea una nueva vista en 'app/Views/news/create.php':

```

1  <h2><?= esc($title) ?></h2>
2
3  <?= session()->getFlashdata('error') ?>
4  <?= validation_list_errors() ?>
5
6  <form action="/news/create" method="post">
7      <?= csrf_field() ?>
8
9      <label for="title">Titulo</label>
10     <input type="input" name="title" value="<?= set_value('title') ?>">
11     <br>
12
13     <label for="body">Texto</label>
14     <textarea name="body" cols="45" rows="4"><?= set_value('body') ?></textarea>
15     <br>
16
17     <input type="submit" name="submit" value="Crear objeto news">
18 </form>

```

```

<!doctype html>
<html>
<head>
    <title>create.php</title>
</head>
<body>

    <h2><?= esc($title) ?></h2>

    <?= session()->getFlashdata('error') ?>
    <?= validation_list_errors() ?>

    <form action="/news/create" method="post">
        <?= csrf_field() ?>

        <label for="title">Titulo</label>
        <input type="input" name="title" value="<?= set_value('title') ?>">
        <br>

        <label for="body">Texto</label>
        <textarea name="body" cols="45" rows="4"><?= set_value('body') ?></textarea>
        <br>

        <input type="submit" name="submit" value="Crear objeto news">
    </form>

</body>
</html>

```

Figura 24: Vista create.php

- La función `session()` es usada para obtener el objeto `Session`, y `session()->getFlashdata('error')`

muestra el error relacionado con la protección CSRF al usuario. No obstante, por defecto, si una validación de la protección CSRF falla, se lanzará una excepción, en caso de que no funcione.

- La función `validation_list_errors()` proporcionada por la **Ayuda de Formularios –Form Helper–** es usada para detectar errores relacionados a la validación del formulario.
- La función `csrf_field()` crea una entrada oculta con un *token* CSRF que ayuda a protegerse contra algunos ataques comunes.
- La función `set_value()` proporcionada por la **Ayuda de Formularios** es usada para mostrar los últimos datos introducidos cuando salta un error.

5.2.2. Controlador

Volvamos al controlador `News`. Añadiremos varias funcionalidades bajo los dos siguientes conceptos que tendremos que tener en cuenta:

- Comprobar cuando se ha enviado el formulario.
- Comprobar si la información enviada cumple con las reglas de validación. Para ello, usaremos el `validation method` en `Controller`.

```
1 <?php
2
3 namespace App\Controllers;
4
5 use App\Models\NewsModel;
6
7 class News extends BaseController
8 {
9     // ...
10
11     public function create()
12     {
13         helper('form');
14
15         // Comprueba si se ha enviado el formulario.
16         if (! $this->request->is('post')) {
17             // El formulario no se envia, por lo que devuelve el formulario.
18             return view('templates/header', ['title' => 'Crea un objeto news'])
19                 . view('news/create')
20                 . view('templates/footer');
21         }
22
23         $post = $this->request->getPost(['title', 'body']);
24
25         // Comprueba si los datos enviados han superado las reglas de validacion.
26         if (! $this->validateData($post, [
27             'title' => 'required|max_length[255]|min_length[3]',
```

```
28         'body' => 'required|max_length[5000]|min_length[10]',
29     ))) {
30         // La validacion falla y devuelve el formulario.
31         return view('templates/header', ['title' => 'Crea un objeto news'])
32             . view('news/create')
33             . view('templates/footer');
34     }
35
36     $model = model(NewsModel::class);
37
38     $model->save([
39         'title' => $post['title'],
40         'slug' => url_title($post['title'], '-', true),
41         'body' => $post['body'],
42     ]);
43
44     return view('templates/header', ['title' => 'Crea un objeto news'])
45         . view('news/success')
46         . view('templates/footer');
47 }
48 }
```

```
public function create()
{
    helper('form');

    // Comprueba si se ha enviado el formulario.
    if (! $this->request->is('post')) {
        // El formulario no se envía, por lo que devuelve el formulario.
        return view('templates/header', ['title' => 'Crea un objeto news'])
            . view('news/create')
            . view('templates/footer');
    }

    $post = $this->request->getPost(['title', 'body']);

    // Comprueba si los datos enviados han superado las reglas de validación.
    if (! $this->validateData($post, [
        'title' => 'required|max_length[255]|min_length[3]',
        'body' => 'required|max_length[5000]|min_length[10]',
    ])) {
        // La validación falla y devuelve el formulario.
        return view('templates/header', ['title' => 'Crea un objeto news'])
            . view('news/create')
            . view('templates/footer');
    }

    $model = model(NewsModel::class);

    $model->save([
        'title' => $post['title'],
        'slug' => url_title($post['title'], '-', true),
        'body' => $post['body'],
    ]);

    return view('templates/header', ['title' => 'Crea un objeto news'])
        . view('news/success')
        . view('templates/footer');
}
```

Figura 25: Método create implementado en news.php

- Primero cargamos la Ayuda de Formularios con la función `helper()`. La mayoría de las funciones ayudadoras requieren que el helper esté cargado antes de su uso.
- A continuación, comprobamos si tratamos la petición **POST** con el objeto `IncomingRequest`, `$this->request`. Es establecido en el controlador por el *framework*. El método `IncomingRequest::is()` comprueba el tipo de la petición. Dado que la ruta para el *endpoint* de `create()` maneja ambas peticiones: **GET** y **POST**, podemos asumir con seguridad que si la solicitud no es **POST**, entonces es de tipo **GET**. Tras esto, el formulario es cargado y se muestra por pantalla.
- Luego, obtenemos los elementos necesarios de los datos **POST** por el usuario y los establecemos en la variable `$post`. También utilizamos el objeto `IncomingRequest`, `$this->request`.
- Después de eso, la función de ayuda proporcionada por el **controlador**, `validateData()`, se utiliza para validar los datos de `$post`. En este caso, los campos `title` y `body` son obligatorios y tienen una longitud específica.

- Si la validación falla, el formulario es cargado y mostrado por pantalla.
- Si la validación pasa todas las reglas, `NewsModel` es cargado y llamado. Esto se encarga de pasar el objeto `news` al **modelo**. El método `save()` se encarga de insertar o actualizar el registro automáticamente, basándose en si encuentra una clave `array` que coincida con la clave primaria.
- Contiene una nueva función, `url_title()`. Esta función –proporcionada por el **Ayudante de URL**– elimina la cadena pasada, reemplazando todos los espacios por guiones (-) y se asegura de que todo está en minúsculas.

Tras esto, las vistas son cargadas y mostradas con un mensaje de éxito. Crea una nueva **vista** en `'app/Views/news/success.php'` y escribe un mensaje que informe de éxito:

```
1 <p>Objeto news creado exitosamente.</p>
```

```
<!doctype html>
<html>
<head>
    <title>success.php</title>
</head>
<body>
    <p>Objeto news creado exitosamente.</p>
</body>
</html>
```

Figura 26: Vista `success.php`

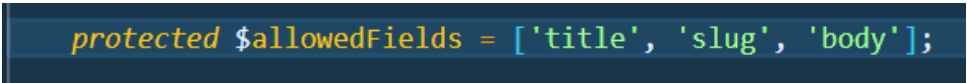
5.3. Actualizando el modelo

Lo único que queda es asegurarse de que el **modelo** está configurado para permitir que los datos se guarden correctamente. El método `save()` determinará si la información debe ser insertada o si la fila ya existe y debe ser actualizada, basándose en la presencia de una clave primaria. En este caso, no se le ha pasado ningún campo `id`, por lo que insertará una nueva fila en su tabla, `news`.

Sin embargo, por defecto, los métodos de inserción y actualización en el **modelo** no guardarán ningún dato porque no sabe qué campos son seguros para ser actualizados. Editaremos pues el modelo `NewsModel` para proporcionarle una lista de campos actualizables en la propiedad `$allowedFields`.

```
1 <?php
2
3 namespace App\Models;
```

```
4
5 use CodeIgniter\Model;
6
7 class NewsModel extends Model
8 {
9     protected $table = 'news';
10
11     protected $allowedFields = ['title', 'slug', 'body'];
12 }
```



```
protected $allowedFields = ['title', 'slug', 'body'];
```

Figura 27: \$allowedFields añadido en NewsModel

Este nuevo atributo contiene los campos permitidos para que se guarden en la **base de datos**.

5.4. Actualizando el enrutamiento

Ahora, tendremos que actualizar `app/Config/Routes.php`:

```
1 <?php
2
3 // ...
4
5 use App\Controllers\News;
6 use App\Controllers\Pages;
7
8 $routes->match(['get', 'post'], 'news/create', [News::class, 'create']);
9 $routes->get('news/(:segment)', [News::class, 'view']);
10 $routes->get('news', [News::class, 'index']);
11 $routes->get('pages', [Pages::class, 'index']);
12 $routes->get('(:segment)', [Pages::class, 'view']);
13
14 // ...
```

```
$routes->match(['get', 'post'], 'news/create', [News::class, 'create']); // Añadida.
```

Figura 28: Enrutamiento de create

Ahora, iniciamos un servidor y accedemos a `/news/create` en la *URL*. Añadimos una noticia nueva y comprobamos si hay cambios en las vistas:



Título Tutorial Completado

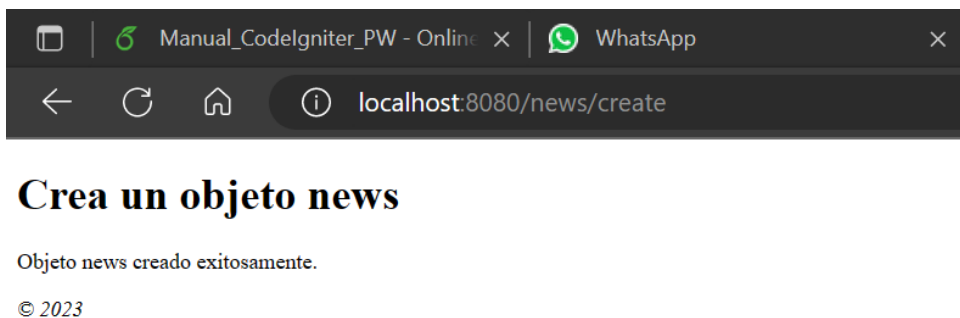
¡Enhorabuena! Has completado tu primera aplicacion web.

Texto

Crear objeto news

© 2023

Figura 29: Vista webcreate



Crea un objeto news

Objeto news creado exitosamente.

© 2023

Figura 30: Objeto news creado exitosamente

Y si volvemos a la vista `news`, observaremos que ha cambiado:

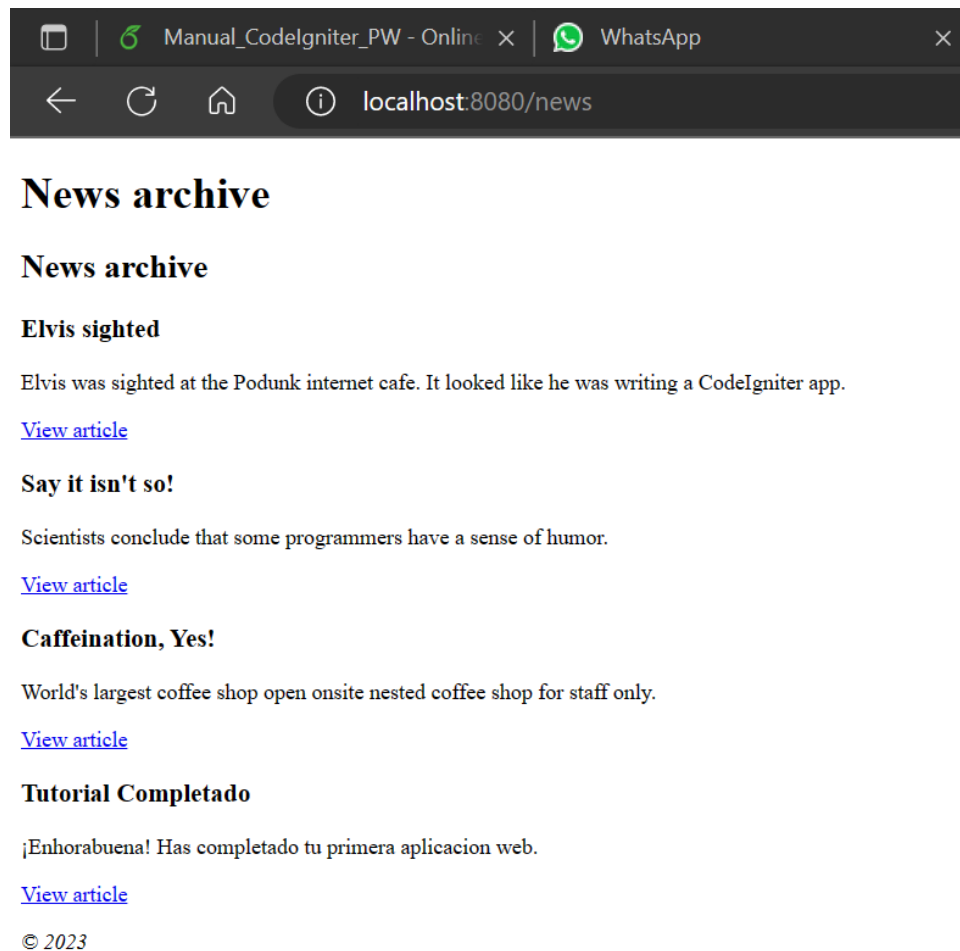


Figura 31: Vista web news actualizada

6. Conclusiones

Si hay algo que se deba destacar de la documentación de CODEIGNITER, esto sería que es una muy **ordenada** e **intuitiva**. Durante el desarrollo compaginado tanto de la aplicación web como de la memoria en L^AT_EX, no ha habido mayor problema salvo en algunos casos donde se ha tenido que revisar la documentación en una lectura rápida de la misma o consultar páginas de confianza en la comunidad informática, como pueden ser *Stackoverflow* o similares.

- El mayor problema, sin duda, se encontró en la configuración de la base de datos –problema que ya se mencionó y especificó en su respectivo apartado 4.2–.