Parte 2 – Creación Blog en Laravel

Información Básica

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Consultar: <https://code.tutsplus.com/es/tutorials/using-faker-to-generate-filler-data-for-automated-testing--cms-26824>

<https://styde.net/generando-datos-de-prueba-con-faker-en-laravel/>

Llenar base de datos usando factories

1. Se deben crear factories para cada entidad excepto para User, que por default se crea

Texto, Escala de tiempo

Descripción generada automáticamente

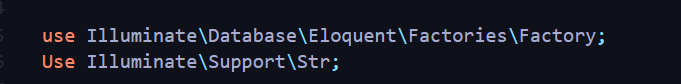
1. Luego se abre cada una de estas clases para poder llenar la base de datos con información de prueba o falsa (fake):

2.1. Abrimos CategoriaFactory:

En el return se especifica los campos que queremos diligenciar, pero antes es necesario definir la variable $nombre, donde decimos que llenaremos el campo respectivo con una palabra (Word) de 20 caracteres. Igualmente, es necesario llamar al helper Str en la parte superior:

Texto

Descripción generada automáticamente



Luego en el return especificamos los campos a llenar:

Pantalla de video juego

Descripción generada automáticamente con confianza media

También, queremos especificar que el nombre nunca se repita, así:

Una captura de pantalla de un celular

Descripción generada automáticamente con confianza media

2.2. Ahora abrimos PostFactory:

Hacemos algo similar.

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

Texto

Descripción generada automáticamente

No olvidar que en las clases que se implementan o llaman en la parte superior deben aparecer estas:

Texto

Descripción generada automáticamente

2.3. Procedemos con el factory tag (etiquetas):

Texto

Descripción generada automáticamente

2.4. Seguimos con el factory Image:

Normalmente las imágenes que subimos al servidor se guardan en la ruta storage/app pero queremos que se almacenen en storage/app/public por esta vez.

Esto se hace así porque no queremos que las imágenes se sincronicen con nuestro repositorio de github, el cual tomará solo en cuenta lo que esté en storage. Pero como aquí son imágenes de prueba, o datos falsos (fake), no queremos que se sincronicen a github ni se suban al servidor real. Por eso, necesitamos la ruta public que está en storage/app, donde claramente se especifica con el archivo .gitignore que no habrá sincronización.

Para hacer una simple prueba de esto, podemos crear un archivo llamado ImagenPrueba.txt dentro de storage/app/public y se verá inmediatamente en el acceso directo de storage que está en la carpeta public del proyecto

Interfaz de usuario gráfica, Texto, Aplicación

Descripción generada automáticamente

Y precisamente es allí donde queremos acceder directamente y no a storage/app.

* **Recordar que lo anterior es aplicable exclusivamente en modo desarrollo.**

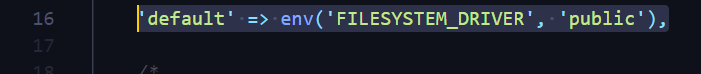
|  |
| --- |
| **Apuntes adicionales:**  **De acuerdo al procedimiento aquí descrito, las imágenes en modo desarrollo se guardarán en la carpeta public/storage. En este caso, el navegador sí puede acceder al archivo. Observe el siguiente ejemplo:**    **Obviamente, si queremos hacer referencia a una imagen o archivo que está allí en public/storage debemos escribir la ruta respectiva. Eso sí, la palabra public no se debe usar. Observe el siguiente ejemplo:** |

Para lograr que todas las imágenes que subamos en modo Desarrollo queden en storage/app/public, se debe hacer una modificación en config/filesystems (o en su default, utilizar el comando php artisan storage:link. Para mayor información, visite: <https://www.youtube.com/watch?v=PjEutNUZjj0&ab_channel=Aprendible> y <https://www.youtube.com/watch?v=tDgFOKvQajg&ab_channel=Aprendible>). Se deben seguir estos pasos:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Luego modificamos la línea 'default' => env('FILESYSTEM\_DRIVER', 'local'), por:



\*Nota: Hay que tener precaución con el procedimiento anterior, ya que esto solo se hace para introducir datos de prueba o fake en la BD.

3) ***Lo siguiente ya no es necesario, pero es recomendable:*** Se debe trabajar sobre un dominio para que lo anterior funcione. En Xampp lo creamos de la siguiente manera (\*Nota: En Laragon se crea automáticamente).

3.1. Abrimos bloc de notas o Notepad++ en modo administrador.

3.2. Desde la aplicación, buscamos la ruta C:\Windows\System32\drivers\etc y abrimos el archivo hosts. Allí agregamos la siguiente línea, indicando así que el dominio que queremos usar ahora es ese:

Texto

Descripción generada automáticamente

3.3. Para que dicho dominio, apunte al proyecto que estamos desarrollando, hacemos lo siguiente:

Abrimos desde el mismo bloc de notas o desde Notepad++ el archivo httpd-vhosts.conf y agregamos al final lo siguiente:

NameVirtualHost \*

<VirtualHost \*>

DocumentRoot "C:\xampp\htdocs"

ServerName localhost

</VirtualHost>

<VirtualHost \*>

DocumentRoot "C:\xampp\htdocs\proyectoblog\blog\public"

ServerName blog.test

<Directory "C:\xampp\htdocs\proyectoblog\blog\public">

Options All

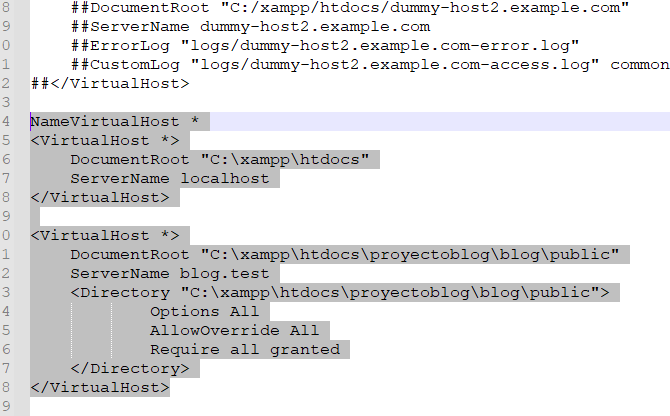
AllowOverride All

Require all granted

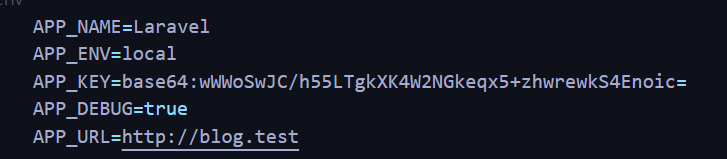
</Directory>

</VirtualHost>

Como puede apreciarse en esta captura:



Modificamos el APP\_URL en el archivo env de LaraveL:



***Aquí terminan las líneas que son más bien opcionales por ahora***

En ImageFactory ponemos lo siguiente:

Texto

Descripción generada automáticamente

Con lo anterior tendríamos listos todos los factories.

Es hora de ir a la base de datos en MySQL, a través de la interfaz de PhpMyAdmin y revisar que todo esté en orden.

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Interfaz de usuario gráfica, Texto, Aplicación, Correo electrónico

Descripción generada automáticamente

**Relacionar Posts con los tags**

En la base de datos, en la tabla de post vemos esto:

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

Por ejemplo, quiero que un post contenga dos tags. Para eso, tenemos que introducir datos en la tabla intermedia:

Diagrama

Descripción generada automáticamente

Para lograrlo, es necesario:

1. Observemos el modelo Post y el método público tags() que contiene, el cual especifica la relación de muchos a muchos con la clase modelo tag.

Texto

Descripción generada automáticamente

1. Abrimos el seeder PostSeeder y digitamos esto:

Texto

Descripción generada automáticamente

* 1. Pero ¿cómo sabe Laravel que debe insertar esta información fake en una tabla intermedia y no en otro lugar? Es gracias al método tags() que de hecho hace referencia a una relación de muchos a muchos, la cual se definió como ya se dijo en el modelo Post(). Luego necesitamos saber a qué id del tag va a hacer referencia. Se lo indicamos manualmente. Si ponemos un 1 como parámetro de attach(), entonces hará referencia al id 1 de la tabla tags. Luego, el primer registro tendrá el id del post respectivo, haciendo referencia al id de ese post, y al id 1 de tags. Así (siga los pasos):

Pantalla de computadora con letras

Descripción generada automáticamente con confianza media

* 1. Pero cómo relaciono, por ejemplo, un post específico con los id 1 y 2 de los tags. Crearemos un array dentro de attach para lograrlo:

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

Aunque podríamos probar que funciona, sería más interesante aún hacerlo con id’s al azar de la tabla tags, pero ¿cómo?

* 1. Para el primer id de tags con el que relacionaremos el post, llamamos al método rand() y le decimos que escoja al azar un número entre el 1 y el 4, y luego para el segundo id de tags que requerimos, hacemos algo parecido pero que se escoja al azar un número entre 5 y 8:

Texto

Descripción generada automáticamente

Como puede apreciarse borramos las líneas de los pasos 2.1 y 2.2 para dejar definitivamente esto último.

* 1. Ahora volvemos a ejecutar la migración con los seeders: