Internet y Negocio Electrónico, 2020-2021
Práctica 3 – Modelo-Vista-Controlador básico

Objetivo

El objetivo de esta práctica es adquirir experiencia en la construcción de una entidad del modelo (en el sistema de archivos mediante el ORM Eloquent y su migración a la base de datos), su correspondiente controlador y la inclusión en la vista, empleando para ello la interfaz Artisan de línea de instrucciones de Laravel cuando corresponda. De esta manera, la actividad consiste en la creación de la clase **`Product`**, su migración a la base de datos, la inclusión de registros en ésta, la creación de la clase controlador **`ProductController`**, el ajuste en el enrutamiento en **`web.php`**, y la modificación de **`welcome.blade.php`** para mostrar los datos almacenados en el modelo/base de datos, quedando el resultado con un aspecto similar al resultado de la práctica anterior. Previamente a la construcción del controlador, y para la visualización y el tratamiento de datos de prueba, se empleará la interfaz de instrucciones Tinker.

Requisitos previos

Para la realización de esta práctica se requiere:

- Práctica anterior funcionando correctamente.

Criterios sintácticos de este documento

- `Courier`: código fuente, nombres de archivos, paquetes, entidades, atributos, tablas y campos.
- *`Cursiva`*: términos en otro idioma.
- Negrita: contenido resaltado.
- **#**Entre almohadillas**#**: contenido no literal, dinámico o a decidir por el desarrollador.

Pablo de la Torre Moreno
Departamento de Ingeniería Informática

Creación y migración del modelo

1. Revisar la documentación facilitada

1. Repasar los aspectos básicos del ORM Eloquent, migración y Artisan.

2. Creación de clase del modelo en el proyecto

1. Crear la clase **`Product`**, mediante la ejecución, en una nueva instancia de la consola de Visual Studio Code, de[1]

```
php artisan make:model Product
```

2. Abrir el archivo recién creado **`app > Models > Product.php`**.
3. Forzar la creación de la tabla homónima (evitando el cambio a plural) mediante la inclusión, dentro de **`class Product`**, debajo de **`use HasFactory;`**, de la línea

```
protected $table = 'Product';
```

3. Inclusión de campos y creación de la tabla en la base de datos

Nota: Los campos no se crean en la clase original, sino directamente en la clase de migración.

1. Crear clase de migración de Laravel, mediante la ejecución de

```
php artisan make:migration create_product_table --create=Product
```

2. Abrir el archivo recién creado **`database > migrations > #yyyy_MM_dd_hhmmss#_create_product_table.php`**.
3. Añadir, en la instrucción **`Schema::create`** de la función **`up()`**, y justo antes de **`$table->timestamps()`**, los campos necesarios para la gestión de productos (**`name`**, **`description`**, **`imgurl`**, **`price`**, **`discountPercent`**), basándose en el contenido de la sección *Columns* de la documentación sobre migración de Laravel[2]. Deben incluirse también los campos **`discountStart_at`** y **`discountEnd_at`** (o nombres similares), como **`datetime`**, para indicar la fecha de inicio y fin de los descuentos, y que pueden contener valores nulos. Es conveniente recordar añadir también el tamaño de los campos de tipo cadena, si permite nulos y/o son índices, en su definición. Por ejemplo:

[1] Desde este momento y en sucesivas prácticas, "mediante la ejecución de" se contextualizará en la consola de instrucciones de Visual Studio Code.

[2] Se aconseja no emplear el campo de tipo **`Text`**, sino **`Varchar`** (**`string`**, en Laravel), que permite un máximo de 21 844 caracteres en formato UTF-8. No es necesario incluir el número de caracteres al definirlo en Laravel.

```
$table->string('name', 256)->nullable(false)->unique();
```

4. Enviar a la base de datos mediante

```
php artisan migrate
```

- En caso de error que indique que la tabla **user** ya existe, es necesario sacar ésta del esquema de migración y repetir la migración mediante

```
php artisan tinker
Schema::drop('user')
exit
php artisan migrate
```

- En caso de otro tipo de error, como al haber introducido los campos, revertir el proceso mediante la ejecución de

```
php artisan migrate:rollback
```

Nota: En cualquier momento del desarrollo, la ejecución de rollback *hará que se pierdan todos los datos de las tablas creadas en la migración previa.*

5. Acceder a la base de datos mediante *phpMyAdmin* y comprobar que la base de datos tiene las tablas y campos correspondientes.

Construcción de los métodos del modelo

1. Implementar los métodos para ofertas de hoy y últimos productos

1. Crear en la clase **Product** los métodos: a) **Offerings()**, que devolverá los productos en oferta como aquéllos entre cuyas fecha de inicio y fin de descuento se encuentra el día de hoy, utilizando para ello el método **where** de Eloquent; y, b) **NewProducts()** que devolverá los últimos productos insertados o modificados durante la última semana. Se facilita el código de este último[3], para que se use como referencia para desarrollar el método a):

[3] Se aconseja analizar el método facilitado con relación al trabajo con las fechas y al método **where**, que emplea con la tupla *[campo, valor]* (que equivale a *campo = valor*) o *[campo, operador, valor]*. En situaciones donde el primer parámetro no sea únicamente un nombre de campo, se emplea **DB::raw** con el código SQL directamente insertado. Esto requiere **use DB**, arriba, en la clase.

```
static function NewProducts() {

   $sNow = date('Y-m-d H:i:s');
   $sNextWeek = date('Y-m-d H:i:s',
       strtotime($sNow . ' + 1 week'));

  return Product::where(DB::raw(
     'date_format(updated_at, "%Y-%m-%d")'),
     '>=', date('Y-m-d', strtotime($sNow)))->
     where('updated_at', '<=', date('Y-m-d',
     strtotime($sNextWeek)))->get();
}
```

2. Poblar y comprobar mediante Tinker

1. Acceder a la consola de instrucciones Tinker mediante la ejecución en la consola de[4]

```
php artisan tinker
```

2. Comprobar que no hay datos de los productos en la tabla, mediante

```
Product::count()
```

3. Crear un producto mediante

```
$p = new Product
```

4. Asignar valor a los campos correspondientes (salvo **id**, **created_at** y **updated_at**), usando como valor para la imagen **"img/card-000001.jpg"**, y dando valor a **discountStart_at** y **discountEnd_at** (o como se les haya llamado) como para esperar que sea devuelto por el método **Offerings** creado en el modelo (ver más arriba). Un ejemplo de modificaciones de valor se muestra a continuación:

```
$p->imgurl = "img/card-000001.jpg"

$p->discountStart_at=new DateTime('2020-11-09 19:30',
    new DateTimeZone('Europe/Madrid'))
```

5. Guardar los datos mediante

[4] Desde este momento y en sucesivas prácticas, "mediante la ejecución de" se contextualiza en la consola de instrucciones de Visual Studio Code.

```
$p->save()
```

6. Comprobar que aparece como resultado a ambos **NewProducts** y **Offerings**.

```
Product::NewProducts()
Product::Offerings()
```

7. Crear otro producto siguiendo los pasos previos.

8. Probar la obtención de productos con

```
App\Product::get()
App\Product::first()
App\Product::where('id', 1)->get()
```

9. Crear hasta cinco productos (incluyendo los ya creados) para que tres de ellos sean devueltos por **Offerings**. Para ello puede usarse indistintamente la consola de Tinker, como se ha explicado, o hacer los cambios directamente en la tabla de la base de datos. Las imágenes de todos ellos deben tener el criterio referido en el punto 6, usando como referencia numérica el campo **id**.

10. Salir de Tinker con la instrucción **exit**.

Implementación y enrutamiento del controlador

1. Crear la clase controlador

1. Construir **ProductController** mediante

```
php artisan make:controller ProductController
```

2. Crear e implementar en dicha clase **app > Http > Controllers > ProductController.php** el método **welcome**, que consistirá en la asignación a las variables[5] **$aProduct_offering** y **$aProduct_new** de las invocaciones a

[5] En las prácticas se empleará la notación húngara, donde el prefijo "a" significa *array*. Así, **$aProduct_offering** quiere decir "un array donde cada elemento es una instancia de la clase **Product** con la semántica 'oferta'".

Product::Offerings() y **Product::NewProducts()**, respectivamente[6], seguido de la devolución de la vista *welcome* (es decir, el archivo **welcome.blade.php**) a la cual se le pasarán, como parámetros, las dos variables asignadas, mediante la instrucción

```
return view('welcome',
    compact('aProduct_offering', 'aProduct_new'));
```

2. Modificar el enrutamiento

1. En **routes > web.php**, cambiar la línea actual de enrutamiento en caso de uso de GET en el directorio raíz por la llamada al método **welcome** del controlador.

```
Route::get('/',
    'App\Http\Controllers\ProductController@welcome')
    ->name('home');
```

2. Probar que la web se muestra correctamente, lo que significa que el enrutamiento funciona y que ni el controlador ni el modelo dan errores, aunque aún no se haya modificado la vista **welcome.blade.php** para mostrar aún los valores de las variables **$aProduct_offering** y **$aProduct_new** enviadas por el controlador.

Hacer dinámica la vista

1. Sustituir mediante recorrido de las variables del controlador

1. Convertir en dinámico el contenido estático de las ofertas del día, a través de iterar **$aProduct_offering** que llega del controlador. Los códigos embebidos para iteración secuencial por valores de una lista, condicionales e impresión son, respectivamente:
    - **@foreach (#*lista*# as #*elemento*#) … @endforeach**
    - **@if (#*condición*#) … @endif**
    - **{{ #*contenido a incluir dinámicamente en el HTML, pudiendo contener código PHP*# }}**

    Un ejemplo de uso es:

```
@if($person->height > 180)
    {{ $person->name . " es alta, y mide " . number_format(
        $person->height, 2) }}
@else
    {{ $person->name . " no es especialmente alta." }}
@endif
```

[6] No debe olvidarse incluir **use App\Product;** en el lugar correspondiente.

2. Repetir el punto anterior con el contenido de **`Nuevos productos`**. En éste, se imprimirán los descuentos sólo si hay descuentos. Para ello, debe crearse un método **`public function HasDiscount()`** en la clase **`Product`** que devolverá true en caso de que haya descuento: que descuento no sea nulo y que la fecha actual se encuentre entre las fechas de descuento. Probar dicha función en Tinker e incluir en **`welcome.blade.php`**, en la sección correspondiente. Como se observará, es posible invocar desde la vista a métodos del modelo, como, en este caso, **`$product->HasDiscount()`**.