FABRICANDO DATOS

Datos ficticios:

Es habitual realizar la inserción de datos “ficticios” en nuestra aplicación:

- Para poder probar nuestros programas
- Para poder realizar pruebas
- etc

Por datos ficticios entendemos aquellos que no son añadidos por el usuario en producción, es decir, son datos “inventados”, a veces se les referencia como datos “falsos”.

Patrón Factory:

Factory es un patrón de diseño orientado a objetos. Nos permite abstraer la creación de objetos, de forma que los programas no tienen que saber cómo se crean, simplemente eligen un tipo de objeto u otro.

Este patrón es uno de los 23 patrones populares de diseño orientado a objetos. Estos patrones resuelven problemas de diseño y facilitan el mantenimiento y la reutilización del software.

Laravel ha automatizado este patrón para facilitar el desarrollo. En concreto, para facilitar la creación de datos en nuestra base de datos.

Factories: Estructura para crear los datos

Cuando creamos nuestros modelos, Laravel genera, mediante la opción -f, unas factorías por defecto para cada una de las tablas.

En el capítulo anterior utilizamos esta opción, y se crearon las factorías de cada una de nuestras tablas, a excepción de la tabla de relación tag_thread, que va a ser rellenada automáticamente por Laravel.

A continuación, veremos cómo se define la estructura de la información en una factoría.

Ejemplo de factoría: Factory de la tabla users

Función fake

Laravel ha desarrollado la tabla de usuarios, y también su factoría. Si nos fijamos en la definición, utiliza una función fake (ficticio), que genera datos ficticios de forma automática.

Factoría de la tabla user

Crea un usuario de ejemplo. La información la obtendrá de:

UserFactory.php

```
public function definition()
{
  return [
    'name' => fake()->name(),
    'email' => fake()->unique()->safeEmail(),
    'email_verified_at' => now(),
    'password' =>
       '$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi',
       // password
    'remember_token' => Str::random(10),
  ];
}
```

Como puede verse, en la factoría se generan los datos de la tabla que se está rellenando. Salvo los timestamps y el id, debemos generar un valor ficticio para todos los campos.

Los campos se rellenan en la tabla de usuarios de la forma:

- name: Generará un nombre de usuario ficticio con fake
- email: Generará un email único ficticio con fake
- email_verified_at: Fecha y hora actual
- password: La password sigue la convención de encriptación. El usuario utilizará la password “password”, en la tabla se guarda su encriptación.
- remember_token: Aleatorio

Factoría de la tabla categories

Para fabricar categorías, vamos a la fábrica, que generó artisan al poner la opción -f, y editamos el fichero. La función definition es la que define la estructura de los datos y como se generan.

CategoryFactory.php

```
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;
...
    public function definition()
    {
        return [
            //
            'nombre' => $name = fake()->unique()->word(),
            'slug' => Str::slug($name)
        ];
    }
...
```

En la tabla categorías, los valores de los campos se obienene de:

- nombre de la categoría lo generará Laravel a partir de palabras reales (word), de forma única.
- slug se obtendrá a partir del nombre asignado. Para ello, utilizaremos la clase Str (importada previamente con use), y su función para generar slugs a partir de otros textos.

Factoría de la tabla de comentarios

La tabla de comentarios tiene dos campos de relación.

CommentFactory.php

```
use App\Models\Thread;
use App\Models\User;
...
    public function definition()
    {
        return [
            //
            'thread_id' => Thread::factory(),
            'user_id'   => User::factory(),
            'body'      => fake()->text(500)
        ];
    }
```

Los campos los completamos de la forma siguiente:

- thread_id: Cada vez que se cree un comentario, quiero que se cree una pregunta, a la que quedará asociado
- user_id: Cada vez que se cree un comentario, quiero que se cree un usuario, al que quedará asociado
- body: Se generará un texto ficticio de longitud 500 caracteres

Factoría de la tabla de etiquetas

La tabla de etiquetas es similar a la de categorías, por lo que su factoría es similar:

TagFactory.php

```
use Illuminate\Database\Eloquent\Factories\Factory;
use Illuminate\Support\Str;
...
    public function definition()
    {
        return [
            //
            'nombre' => $name = fake()->unique()->word(),
            'slug' => Str::slug($name)
        ];
    }
```

Factoría de la tabla de preguntas

ThreadFactory

```
namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;
use App\Models\Categories;
use App\Models\User;
use Illuminate\Support\Str;
...
    public function definition()
    {
        return [
            'category_id' => Category::factory(),
            'user_id'     => User::factory(),
            'titulo'      => $titulo = fake()->unique()->sentence(),
            'slug'        => Str::slug($titulo),
            'body'        => fake()->text(1300)
        ];
    }
```

Los campos de la tabla de preguntas son:

- category_id: Se pondrá el id de una categoría que crearemos para crear la pregunta
- user_id: Se pondrá el id de un usuario que crearemos para crear la pregunta
- titulo: Crearemos un título ficticio a partir de una frase única autogenerada
- slug: Utilizaremos la función slug para obtener un slug a partir del título
- body: Generaremos el cuerpo de la pregunta con un texto ficticio de 1300 caracteres

Creación de los datos.

Seeders: DatabaseSeeder.php

Para crear los datos, Laravel proporciona el fichero

`/database/seeders/DatabaseSeeder.php`

Este fichero viene con un código por defecto, comentado, que crea usuarios en la tabla user. Utilizaré la creación que más me interese, en este caso:

DatabaseSeeder.php

```
    public function run()
    {
        \App\Models\User::factory(10)->create();
    }
```

El código anterior, cuando sea lanzado, creará 10 usuarios en la tabla users de la base de datos.

Migración con creación de datos

Creación de datos en la tabla de usuarios

Vamos a utilizar la opción de migración fresh, que permite la opción --seed, que lanza el seeder que alimenta de datos las tablas creadas.

```
$ php artisan migrate:fresh --seed
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacionBD$ php artisan migrate:fresh --seed

  Dropping all tables ..................................................................... 318ms DONE

  INFO  Preparing database.

  Creating migration table ................................................................ 41ms DONE

  INFO  Running migrations.

  2014_10_12_000000_create_users_table ..................................................... 90ms DONE
  2014_10_12_100000_create_password_resets_table ........................................... 130ms DONE
  2019_08_19_000000_create_failed_jobs_table ................................................ 72ms DONE
  2019_12_14_000001_create_personal_access_tokens_table ..................................... 109ms DONE
  2023_01_14_163743_create_categories_table ................................................. 76ms DONE
  2023_01_14_164745_create_threads_table ................................................... 354ms DONE
  2023_01_14_165014_create_comments_table .................................................. 374ms DONE
  2023_01_14_165448_create_tags_table ....................................................... 91ms DONE
  2023_01_15_084108_create_tag_thread_table ................................................. 443ms DONE

  INFO  Seeding database.
```

Como podemos ver, el comando:

- Hace drop de todas las tablas de la base de datos (en vez de rollback de las migraciones, como hacía refresh)
- Crea todas las tablas conforme a la configuración de las migraciones
- Crea los datos en la base de datos (seeding), ejecutando el fichero DatabaseSeeder.php.

Teníamos programada la creación de 10 usuarios. Si vamos a phpMyAdmin:

id	name	email	email_verified_at	password	remember_token	created_at	updated_at
1	Maxwell Heidenreich III	kautzer.keven@example.org	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	L2Rwlp9Jd7	2023-01-15 12:10:39	2023-01-15 12:10:39
2	Beulah Gutmann MD	xprosacco@example.net	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	ql4XYRlAef	2023-01-15 12:10:39	2023-01-15 12:10:39
3	Mr. Adelbert Roberts DDS	zieme.tara@example.net	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	3LnAauaNRv	2023-01-15 12:10:39	2023-01-15 12:10:39
4	Vince Ritchie Jr.	oosinski@example.net	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	rl3Cf9TsoE	2023-01-15 12:10:39	2023-01-15 12:10:39
5	Heidi Pollich	pflatley@example.org	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	TjrngTYhPZ	2023-01-15 12:10:39	2023-01-15 12:10:39
6	Prof. Alysha McDermott I	clarissa.upton@example.org	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	GBxfCFgZqy	2023-01-15 12:10:39	2023-01-15 12:10:39
7	Lonzo Emard	lukas74@example.org	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	KJHPemWHT7	2023-01-15 12:10:39	2023-01-15 12:10:39
8	Hayden Harber	veum.daniella@example.net	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	4XsaI3WOeu	2023-01-15 12:10:39	2023-01-15 12:10:39
9	Lyda Hill II	spencer.kiley@example.com	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	a5GQ0WMiAT	2023-01-15 12:10:39	2023-01-15 12:10:39
10	Ansel Spinka IV	parker.stoltenberg@example.net	2023-01-15 12:10:39	$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2...	NXM04PjIpf	2023-01-15 12:10:39	2023-01-15 12:10:39

Los nombres e emails de los usuarios se han generado automáticamente a partir de datos aleatorios.

La password es la misma, está encriptada y corresponde a la palabra “password”, con la que todos los usuarios podrán entrar en el sistema.

Creación de datos en la tabla de categorías

Añadimos la creación de 5 categorías:

DatabaseSeeder.php

```
\App\Models\Category::factory(5)->create();
```

Aquí no tenemos que hacer use de la clase porque hacemos referencia a la misma con todo el espacio de nombres.

Luego hacemos una migración “fresca” (fresh) con la opción de añadir los datos (--seed)

```
$ php artisan migrate:fresh --seed
```

```
alumno@alumno-VirtualBox:~/proyectos/aplicacionBD$ php artisan migrate:fresh --seed

  Dropping all tables ..................................................... 276ms DONE

  INFO  Preparing database.

  Creating migration table ................................................ 56ms DONE

  INFO  Running migrations.

  2014_10_12_000000_create_users_table ..................................... 102ms DONE
  2014_10_12_100000_create_password_resets_table ........................... 175ms DONE
  2019_08_19_000000_create_failed_jobs_table ................................ 77ms DONE
  2019_12_14_000001_create_personal_access_tokens_table .................... 136ms DONE
  2023_01_14_163743_create_categories_table ................................. 109ms DONE
  2023_01_14_164745_create_threads_table ................................... 405ms DONE
  2023_01_14_165014_create_comments_table .................................. 307ms DONE
  2023_01_14_165448_create_tags_table ....................................... 78ms DONE
  2023_01_15_084108_create_tag_thread_table ................................ 288ms DONE

  INFO  Seeding database.
```

Si vamos a phpMyAdmin, veremos que los 10 usuarios creados esta vez son diferentes.

En cuanto a categorías, se han creado 5 categorías, con un slug relacionado con el nombre:

id	nombre	slug	created_at	updated_at
1	ipsam	ipsam	2023-01-15 12:21:03	2023-01-15 12:21:03
2	et	et	2023-01-15 12:21:03	2023-01-15 12:21:03
3	est	est	2023-01-15 12:21:03	2023-01-15 12:21:03
4	perspiciatis	perspiciatis	2023-01-15 12:21:03	2023-01-15 12:21:03
5	dicta	dicta	2023-01-15 12:21:03	2023-01-15 12:21:03

Resto de creación de datos.

Necesitamos crear preguntas y comentarios para completar con datos todas las tablas. El seeder quedará finalmente de la siguiente forma:

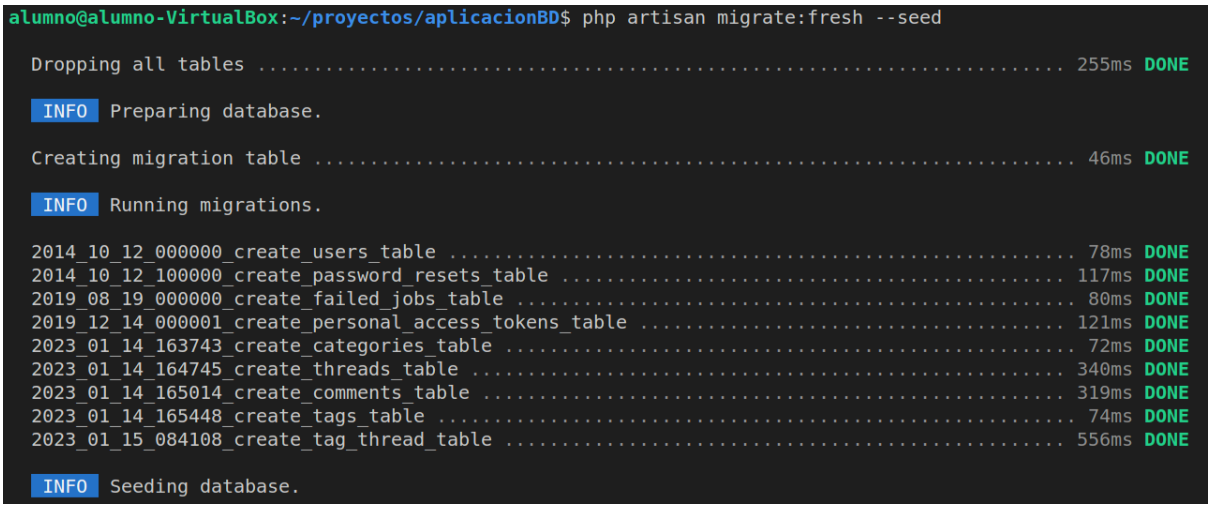
DatabaseSeeder.php

```
        \App\Models\User::factory(10)->create();
        \App\Models\Category::factory(5)->create();
        \App\Models\Thread::factory(40)->create();
        \App\Models\Comment::factory(80)->create();
        \App\Models\Tag::factory(18)->create();
```

Con este código, pido que se generen automáticamente 40 preguntas, 80 comentarios, y 18 etiquetas.

Ejecuto de nuevo la migración y la creación de datos:

```
$ php artisan migrate:fresh --seed
```



Veremos en phpMyAdmin que se han creado los datos solicitados… ¡ y muchos más !:

- Se crean 80 comentarios, tal y como solicitamos
- Se crean 18 etiquetas, como solicitamos
- Se crean 125 categorías, aunque habíamos pedido 5 !!
- Se crean 120 preguntas, aunque habíamos pedido 40!!
- Se crean 210 usuarios, aunque habíamos pedido 5 !!

¿ Por qué es así ?.
En varias factorías, habíamos invocado la función factory(), para crear algunos datos adicionales. Veamos una de las factorías:

CommentFactory

```
            'thread_id' => Thread::factory(),
            'user_id'   => User::factory(),
            'body'      => fake()-
```

Esta factoría, al crear un nuevo comentario, a su vez crea, con la función factory:

- Una nueva pregunta
- Un nuevo usuario

No es lo más interesante para la creación de nuestros datos.

Organización de los datos

Vamos a ver cómo programar el seeder para que los datos queden más organizados.

DatabaseSeeder.php

Para empezar importaremos con use las clases que vayamos a utilizar, para no tener que poner el espacio de nombres completo.

Luego crearemos lo que nos interese, de forma que no se añadan más datos no controlados por mí. Para eso, mi código tendrá en cuenta las relaciones y asociará la información de forma más controlada.

DatabaseSeeder.php

```
use App\Models\Tag;
use App\Models\Thread;
use App\Models\Category;
...
        \App\Models\User::factory(10)->create();

        Tag::factory(6)->create();

        // Crea 6 categorías. Por cada una, creará 10 preguntas,
        // y por cada pregunta 8 comentarios
        Category::factory(6)
            ->has (
                Thread::factory(10)->hasComments(8)
            )
            ->create();

        // Asocia, a cada pregunta generada anteriormente,
        // un array de entre 1 y 6 ids de etiquetas,
        // tomadas de forma aleatoria entre las 6 creadas previamente
        $tags = Tag::all();
        Thread::all()->each(function($thread) use ($tags) {
            $tag = $tags->random(rand(1,6))
            ->pluck('id')
            ->toArray();

            $thread->tags()->attach($tag); // asocia los id
```

Este seeder, tal y como está ahora, no puede funcionar, pues está utilizando funciones como hasComments o attach, que asumen que nuestro modelo en Laravel tiene implementadas las relaciones.

Las relaciones existen en la base de datos, pues están en las migraciones. Sin embargo, no están implementadas en nuestro modelo de datos.

Si ejecutamos migrate:fresh --seeder, fallará porque no va a encontrar la función comments en la clase Thread de mi modelo, etc.

Creación de las relaciones en el modelo

Tabla threads:

Para que el modelo tenga la relación uno a muchos de pregunta a comentarios, añadimos la función comments.

Thread.php

```
    public function comments() {
        return $this->hasMany(Comment::class);
    }
```

Creamos una función para obtener todos los comentarios de una pregunta. Se utiliza la función hasMany, que indica que una pregunta tiene muchos comentarios.

Tabla categories:

Para que el modelo tenga la relación uno a muchos de categoría a preguntas, añadimos la función threads.

Category.php

```
    public function threads() {
        return $this->hasMany(Thread::class);
    }
```

Aquí también utilizamos la función hasMany, que indica que una categoría tiene muchas preguntas.

Tabla tags:

Para que el modelo tenga la relación muchos a muchos entre tags y threads, añadiremos una función a Thread.php para poder recuperar las etiquetas de una pregunta.

Thread.php

```
    public function tags() {
        return $this->belongsToMany(tag::class);
```

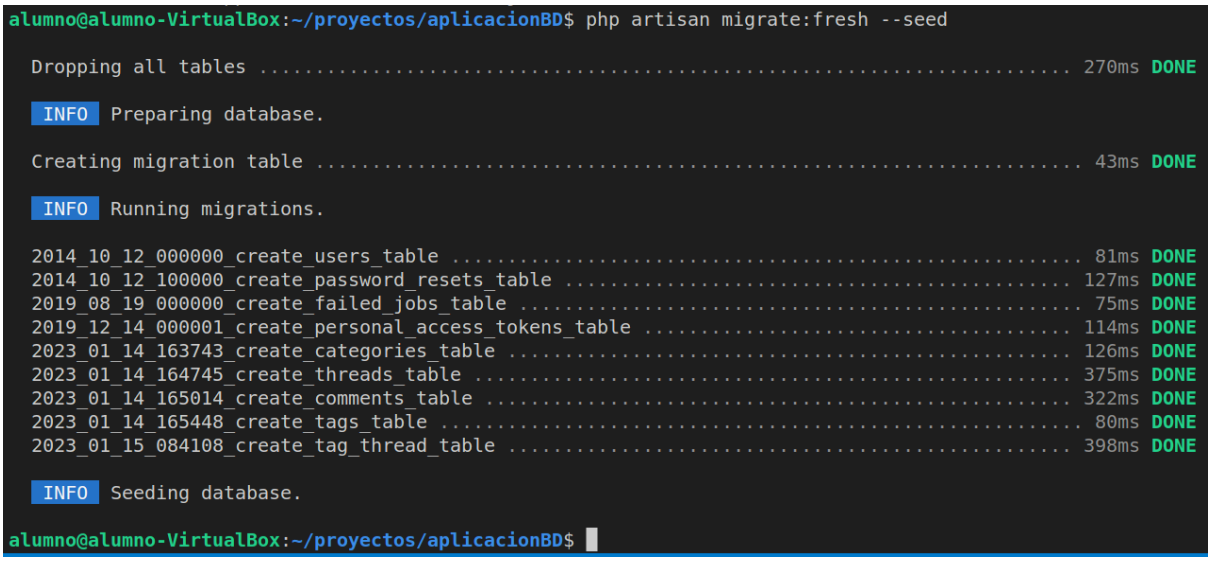
```
    }
```

En este caso, como la relación es n a n, el método es belongsToMany, que significa que pertenece y tiene muchas etiquetas.

Ejecución de las migraciones con --seed

Una vez implementadas las relaciones, tanto en la migración como en el modelo, podremos utilizar nuestro modelo para insertar los datos.

```
$ php artisan migrate:fresh --seed
```



Si escribimos correctamente las funciones necesarias en los modelos, para implementar la relación, nuestro seeder funcionará correctamente y creará los datos en las tablas:

Si ahora vamos a phpMyAdmin:

- Tendremos 6 categorías. No ha creado nuevas, porque al crear los threads los habíamos relacionado con una categoría concreta.
- Tengo 60 preguntas (10 por cada categoría)
- Tengo 6 etiquetas
- Tengo los usuarios necesarios, se ha creado uno nuevo por cada thread y uno nuevo por cada comentarios, en total 550
- La tabla tag_thread contiene 213 registros, que implementan la relación entre etiquetas y preguntas. Cada pregunta tendrá un número variable de etiquetas, entre 1 y 6, de forma aleatoria.
- Como se han generado 8 comentarios por cada una de las preguntas, se han generado un total de 480.