Using Laravel Model Factories To Generate Dummy Data

POSTED ON **3 MAY 2019** POSTED IN **LARAVEL**
TAGS: **LARAVEL MODEL FACTORIES, LARAVEL SEED, MODEL FACTORIES** 5773 VIEWS

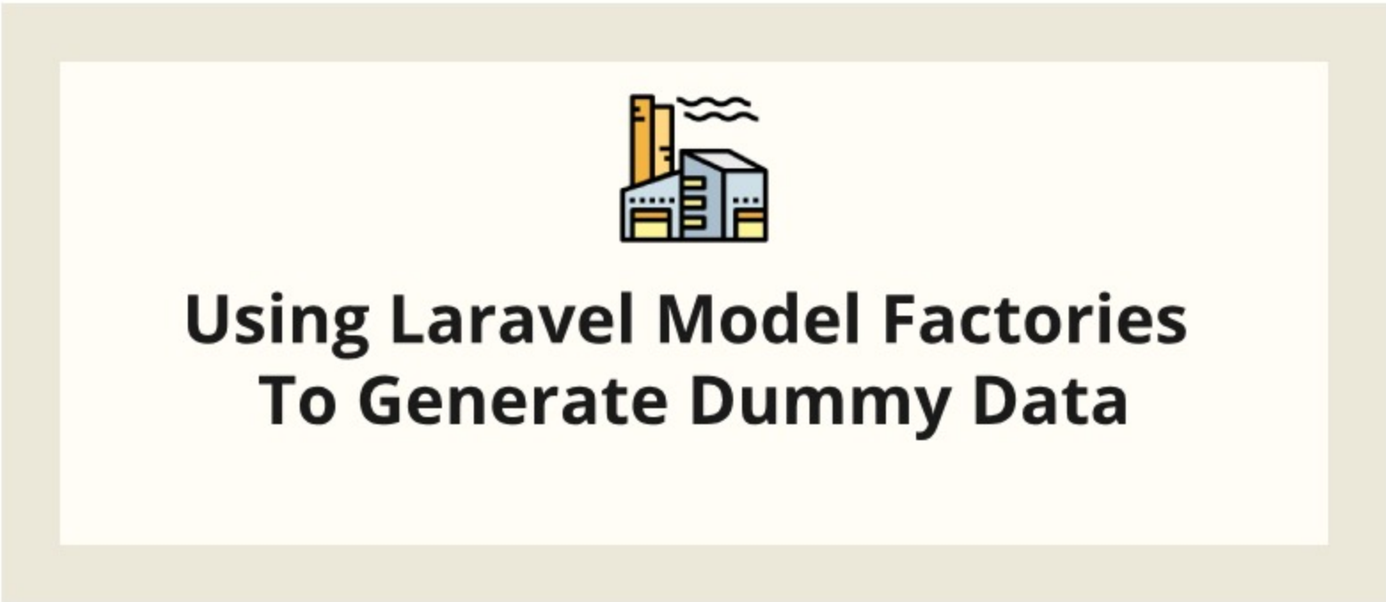


TABLE OF CONTENT

Laravel Model Factories allow you to generate fake data for your eloquent models. Model Factories are very useful for testing and seeding fake data to see if your application work as it suppose to be.

In this post, we will look at how we can create model factories and generate dummy data using the Faker Library which ships with Laravel.

Creating Laravel Application

To learn about model factories and how to use them, we will start by creating a new Laravel application for this post. Run below command in terminal to create a Laravel application.

```
composer create-project laravel/laravel modelFactories --prefer-dist
```

Above command will create a new Laravel application within a folder named **modelFactories**.

Setting Up Database

Once you have your new application created, go to **modelFactories** folder and find the `.env` file to make some changes to your database credentials.

For this post, I will be using the SQLite database engine as we are just working on a dummy project. Change **DB_CONNECTION** to `sqlite` and remove all other database related environment variables like **DB_***. Now you should have only the below environment variable for the database.

```
DB_CONNECTION=sqlite
```

Now we will create a database file called `database.sqlite` in the **database** folder which you can find in the root of your application. Now, run below command to re-establish the configuration cache.

```
php artisan config:cache
```

Creating a Model and Migration

Next, we will create a model and migration for testing purpose. To generate the model and migration run below command in the terminal.

```
php artisan make:model Post -m
```

The `-m` flag will create the migration along with the model class. Above command will generate two files one at `app/Post.php` and other in the migrations folder inside the database folder named something like `create_posts_table`.

Open migration file for posts and replace the `up()` method with the below one.

```
    /**
 * Run the migrations.
 *
 * @return void
 */
public function up()
{
    Schema::create('posts', function (Blueprint $table) {
        $table->bigIncrements('id');
        $table->string('title');
        $table->unsignedInteger('user_id');
        $table->foreign('user_id')->references('id')->on('users');
        $table->date('post_date')->nullable();
        $table->boolean('published')->default(0);
        $table->text('content')->nullable();
        $table->timestamps();
    });
}
```

In the above migration we are defining some columns we need for our `posts` table. As you might have noticed, we are also adding a `user_id` column along with the foreign key. This column will link the posts to the users table.

After we have set up the migration, we will run the `migrate` command in the terminal to migrate this table to the database.

```
php artisan migrate
```

Creating Database Seed Classes

Laravel's database seeds classes allow you to create fake data rapidly and store in the database. Believe me, it's much better than entering data manually when you are building or testing an application.

To create seeds classes we will run the below commands in the terminal.

```
php artisan make:seed UsersTableSeeder
php artisan make:seed PostsTableSeeder
```

Now we have the seed classes generated, which you can find in the **database/seeds** folder. It's time to create our model factories to generate fake data.

Creating Laravel Model Factories

If you go to **database/factories** folder, you will find that Laravel comes with the factory class for **User** model. It will look like something below.

```
/** @var \Illuminate\Database\Eloquent\Factory $factory */
use App\User;
use Illuminate\Support\Str;
use Faker\Generator as Faker;

/*
|--------------------------------------------------------------------------
| Model Factories
|--------------------------------------------------------------------------
|
| This directory should contain each of the model factory definitions for
| your application. Factories provide a convenient way to generate new
| model instances for testing / seeding your application's database.
|
*/

$factory->define(User::class, function (Faker $faker) {
    return [
        'name' => $faker->name,
        'email' => $faker->unique()->safeEmail,
        'email_verified_at' => now(),
        'password' => '$2y$10$92IXUNpkjO0rOQ5byMi.Ye4oKoEa3Ro9llC/.og/at2.uheWG/igi', // passwor
        'remember_token' => Str::random(10),
    ];
});
```

In the above model factory, we can see that this factory is using **Faker** class to generate the dummy data like name and email.

As we have already **User**'s factory class, we only need to create the factory class for **Post** model. Run below command in the terminal to create a model factory.

```
php artisan make:factory PostFactory --model=App\\Post
```

Above command will create the `PostFactory` class in the **database/factories** folder. You can generate the facotry without `--model` tag, then you have to define the model in the class yourself.

Note

When we create the model using `php artisan make:model Post` command, we can pass the `-f` flag to create the factory with the model. We used the `-m` flag to generate the migration in our above example, we can run `php artisan make:model Post -m -f` to generate model, migration and factory all together.

Our **PostFactory** class will look like below:

```
/* @var $factory \Illuminate\Database\Eloquent\Factory */

use App\Post;
use Faker\Generator as Faker;

$factory->define(Post::class, function (Faker $faker) {
    return [
        //
    ];
});
```

Replace you model factory code with below:

```
/* @var $factory \Illuminate\Database\Eloquent\Factory */

use App\Post;
use Faker\Generator as Faker;

$factory->define(Post::class, function (Faker $faker) {
    return [
        'title'         =>      $faker->sentence(6),
        'post_date'     =>      $faker->date(),
        'published'     =>      true,
        'content'       =>      $faker->realText(500),
        'user_id'       =>      function () {
                                return App\User::inRandomOrder()->first()->id;

        }
    ];
});
```

We are setting the post title, date and content using the Faker library. For `user_id` column, we are getting random user id and assigning that to this column.

Using Laravel Model Factories

As we have just defined our model factories, it's time to use them. Firstly, we will open the **DatabaseSeeder** clsss from **database/seeds** folder and make following changes to `run()` function.

```
use Illuminate\Database\Seeder;
use Illuminate\Database\Eloquent\Model;

class DatabaseSeeder extends Seeder
{
    /**
     * Seed the application's database.
     *
     * @return void
     */
    public function run()
    {
        Model::unguard(); // Disable mass assignment

        $this->call(UsersTableSeeder::class);
        $this->call(PostsTableSeeder::class);

        Model::reguard(); // Enable mass assignment
    }
}
```

Note

By default, Laravel doesn't allow the mass assignment for models. By using `Model::unguard()` method, we are disabling the mass assigmnet restrictions and after calling the seeds, we are enabling the mass assignment from models using `Model::reguard()` method.

Now, open the **UsersTableSeeder** class and add below line of code inside `run()` function.

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    factory('App\User', 10)->create();
}
```

In above code, we are using the `facotry()` helper function and creating 10 records for **User** model.

Now, in **PostsTableSeeder**'s `run()` function, we will add below code:

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    factory('App\Post', 20)->create();
}
```

Here, we are creating 20 records for the **Post** model.

Since we have already migrated our tables to database, we can run the database seed to add data to tables. Run below command to seed our **PostsTableSeeder** and **UsersTableSeeder**.

```
php artisan db:seed
```

You will recieve the response from terminal like below:

```
$ php artisan db:seed
Seeding: UsersTableSeeder
Seeding: PostsTableSeeder
Database seeding completed successfully.
```

Now to verify our data, we can use the Laravel tinker. Run below command to launch the Laravel's tinker utility.

```
php artisan tinker
```

Once you have the tinker launched, run below in the terminal:

```
App\Post::all();
```

It will list all the posts, we just created using the Laravel Model Factories.

Factory Makes, Create and Raw Methods

In our above code example we used the `create()` method for our factory like `factory('App\User')->create()`, there are three method provided by the factory helper function. The first one is `create()` which we have already used, others are `make()` and `raw()`.

The `create()` method try to store the data in the database just like the eloquent model. On the other hand, `make()` method create the model with all its properties provided by the factory and return it. The `make()` method will not persist the data to database. The third one `raw()` method will create the model instance with all its properties provided by the factory and return the model instance as an array.

```
factory('App\Post')->create(); // Store the model data to database
factory('App\Post')->make(); // Return the model
factory('App\Post')->raw(); // Return the model as an array
```

We can also use the `make()` method to save it to the database. When we pass any amount to the factory as a second argument, it will return the model collection after making it, which we can iterate through and save each of them using the `each` collection method.

```
factory('App\Post', 10)->make()->each(function ($post) {
    return $post->save();
});
```

Laravel Model Factories States

Using the model factory states, we can set the values of the boolean attributes. In our above example, in our Post model factory, we set the `published` attribute to true manually. We can handle this using the factory states.

In our **PostFactory** class, just after the factory decleration add the below code:

```
$factory->state(Post::class, 'published', function() {
    return [
        'published' =>  true
    ];
});

$factory->state(Post::class, 'draft', function() {
    return [
        'published' =>  false
    ];
});
```

Using above code, we are defining two states for our **PostFactory** `published` and `draft`. Now we can create the posts using the factory states like below:

```
// Create a new post
factory('App\Post')->create();

// Create a published post
factory('App\Post')->states('published)->create();

// Create a draft post
factory('App\Post')->states('draft)->create();
```

When we use the **published** state, it will set the `published` to true while when using the **draft** state it will set the `published` to false.

Conclusion

You can see how powerful Laravel Model Factories are and they can really speed up your development workflow. If you have any question about Laravel Model Factories or suggestion to improve this post, please leave it in the comment box below.

One comment on "Using Laravel Model Factories To Generate Dummy Data"

PedroMagno says:

JULY 3, 2019 AT 5:49 PM

Great post!
Is it possible to explain how to seed a pivot table in a relationship many to many using faker library?

Thanks in advanced.

REPLY

Leave a Reply

Your email address will not be published. Required fields are marked *

Comment *

When sharing a code snippet please wrap you code with **pre** tag and add a class **code-block** to it like below.

<pre class="code-block">you code here</pre>

Name *

Email *

SUBMIT COMMENT

This site uses Akismet to reduce spam. Learn how your comment data is processed.

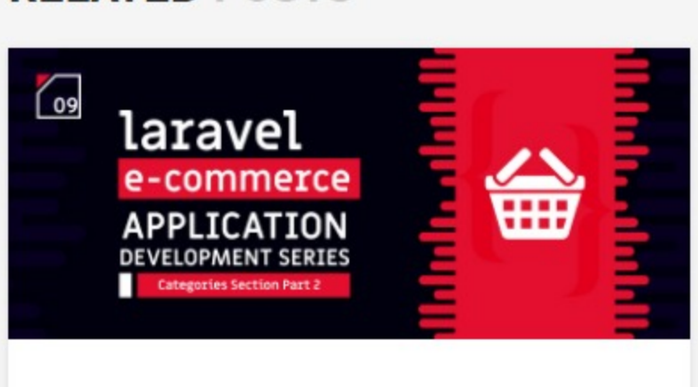
LARASHOUT NEWSLETTER

Join our newsletter to get tips and tutorials right in your inbox.

Enter email address to subscribe

SUBSCRIBE

RELATED POSTS



Laravel E-Commerce Application Development – Categories Section Part 2

POSTED IN LARAVEL

12 JUNE 2019

Adding Translations to Model Using MySQL JSON Field in Laravel

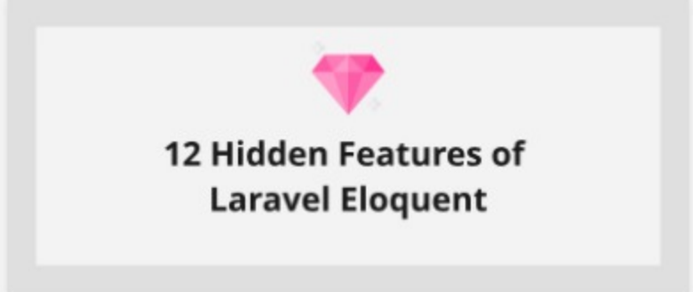
POSTED IN LARAVEL

21 FEBRUARY 2019

Custom Validation Rules in Laravel

POSTED IN LARAVEL

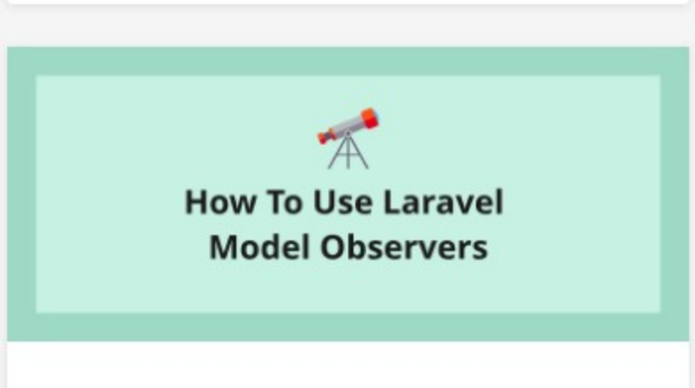
5 MARCH 2019



12 Hidden Features of Laravel Eloquent

POSTED IN LARAVEL

6 MARCH 2019



How To Use Laravel Model Observers

POSTED IN LARAVEL

28 FEBRUARY 2019

How to Host Laravel on DigitalOcean Using Cloudways

POSTED IN LARAVEL

25 OCTOBER 2019

LARASHOUT RECOMMENDS

SUPPORT LARASHOUT

Your little help will keep this site alive and help us to produce quality content for you.