Generate Dummy Laravel Data with Model Factories

A while back, if I needed dummy data for my local application environment — it involved importing a database partial of an existing site, or modifying **MySQL's** sakila database, or just some funky way to get data. Laravel gave us seeding (not a new concept). Seeding lets the developer input data into the local database like it would appear on the main site.

Take this scenario for example. You need to create a database with a thousand users, each user has a certain number of posts (0 - n) attached to them. You would not do this:

PHP

```php
// manually make a list of users
$users = [
    [
        'username' => 'firstuser',
        'name' => 'first user',
        'email' => 'firstuser@email.com'
    ],
    [
        'username' => 'seconduser',
        'name' => 'second user',
        'email' => 'seconduser@email.com'
    ]
    ...
```

```
];
```

This is silly, nobody wants to do this a thousand time. Thanks to libraries like `fzaninotto/faker`, we can create a lot of fake data that makes sense.

The `fzaninotto/faker` library is available on packagist so we can install it via composer. You do not need to install this package if you are using laravel 5 or higher.

PHP

```
composer require fzaninotto/faker --dev
```

After installing `faker`, we can then use our seeder class to create as many users as we want. In the `run` method of the `UserTableSeeder` class or whatever you named your seed class, do this.

PHP

```
public function run() {
    $faker = Faker\Factory::create();

    for($i = 0; $i < 1000; $i++) {
```

```
            App\User::create([
                'username' => $faker->userName,
                'name' => $faker->name,
                'email' => $faker->email
            ]);
        }
    }
```

With the above snippet, we can create a thousand users without having to go through a lot of manual labor.

Although this method saves us a lot of time and effort, it is not the most effective solution out there. When you want to create fake data for testing, you then have to copy this code and paste it in your test class. This method is not maintanable because you now have duplicate, changing a table schema would be a nightmare.

Don't forget, you should always keep your code **DRY**.

Model Factories

Model Factory lets us define a pattern used in generating fake data. A while back, Jeffrey Way of Laracasts created a package called TestDummy. TestDummy allowed developers to define a pattern for creating fake data. Model Factory became part of Laravel's features as of Laravel 5.

To use factories, we have to first define them. In the `database/factories/ModelFactory.php` we can create a factory there. Laravel provides a `$factory` global object which we can extend to define our factories.

PHP

```
$factory->define(App\User::class, function
(Faker\Generator $faker) {
    return [
         'username' => $faker->userName,
        'email' => $faker->email,
        'name' => $faker->name
    ];
});
```

As you can see, the `define` method being called on the `$factory` object takes in two parameters. The first one is an identifier (model FQN), used to later reference the factory.

The second parameter is a closure which takes in `Faker\Generator` class and returns an array of `users`.

Using the Factory

Now that we have our factory defined, we can spin up fresh users anytime we want. To use the defined factory (either from your tests or seed), we use the `factory` function provided by laravel.

PHP

```php
// create a user and save them to the database
$user = factory(App\User::class)->create();
```

This creates a single user. To create many users — just pass a second parameter to the `factory` function.

PHP

```php
// create 1000 users with just one line
$users = factory(App\User::class, 1000)->create();
```

Overriding Attributes

If you want to override some of the default values of your model, you may pass an array of values to the `create` method. Only the specified values will change — while the rest of the values remain set to their default values as specified by the factory.

PHP

```php
$user = factory(App\User::class)->create([
    'username' => 'pizzamuncher'
]);
```

Factory

Make

Sometimes we may not want to save the data created to the database. For that we use the `make` method on the `factory` object instead of `create`.

PHP

```
$user = factory(App\User::class)->make();
```

Just like we did with the `create` method, we can also overwrite attributes of the `method` also. It works on the same principle as the `create` method.

PHP

```
$user = factory(App\User::class)->make([
    'username' => 'pitzanotpizza'
]);
```

Multiple Factory Types

A model can be of different types; we can configure a model factory to use many types. For example say we want some users to be an admin, we can create a factory with `$factory->defineAs()` which takes in three parameters. The first parameter is the base model, the second parameter is a type and the third parameter is a closure that generates data.

PHP

```
$factory->defineAs(App\User::class, 'admin', function
(Faker\Generator $faker) {
    return [
         'username' => $faker->userName,
        'email' => $faker->email,
        'name' => $faker->name,
        'admin' => true
    ];
});
```

Or if you want to extend a base model factory, you can do this:

PHP

```php
$factory->defineAs(App\User::class, 'admin', function
($faker) use ($factory) {
    $post = $factory->raw('App\User');

    return array_merge($post, ['admin' => true]);
});
```

To use this factory created using `defineAs` method, we can change the second argument to the `factory` from number of posts to the new factory identifier.

PHP

```php
factory(App\User::class, 'admin')->create();
```

If you want to create admin users, you can pass a third parameter to the factory function.

PHP

```php
factory(App\User::class, 'admin', 10)->create();
```

Create a Model with Relation

A basic `User` model usually has relations with other models. For example a user can have posts, if we wanted to make a user have posts using model factories, we can do this.

NOTE: This is assuming that you setup the Eloquent model relationship e.g `hasMany` etc.

PHP

```
factory(App\User::class,
50)->create()->each(function($u) {
    $u->posts()->save(factory(App\Post::class)->make());
});
```

The create method returned a `collection` of created users. We can now loop through the users, fetch a post from the database and attach it to the user.

Using Model Factories While Testing

With Model Factories setup, you can now setup database for testing. If you change your schema, all you need to do is update your factory definition and that's all.

PHP

```php
<?php

use Illuminate\Foundation\Testing\WithoutMiddleware;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\DatabaseTransactions;

class ExampleTest extends TestCase
{
    public function setUp() {
        parent::setUp();

        Artisan::call('migrate'); //run migrations
        Eloquent::unguard(); // disable eloquent guard
    }

    public function
it_creates_at_least_hundred_fake_users() {
        $users = factory(App\Users::class, mt_rand(100,
```

```
1000))->create();
        $user_count = count($users) >= 100;

        $this->assertTrue($user_count);
    }
}
```

This is a simple and silly example. But using model factories to handle database seeding a lot more efficient that doing this in every test you write.

PHP

```
public function it_creates_at_least_hundred_fake_users()
 {
    $users = [];
    $faker = Faker\Factory::create();

    for ($i = 0; $i < mt_rand(100, 1000); $i++) {
        $users[] = App\User::create([
            'username' => $faker->userName,
            'email' => $faker->email,
            'name' => $faker->name
        ]);
    }

    $this->assertTrue(count($users) >= 100);
}
```

Conclusion

Laravel's model factories are a powerful tool for testing and seeding. Having them integrated into Laravel makes it easier for developers to test and seed data.