Learn to use Model Factories in Laravel

Laravel 5.1 comes with a feature called model factories that are designed to allow you to quickly build out “fake” models.

These have several use cases with the two biggest being–testing and database seeding. Let’s take a deeper look at this feature by building out the beginning of a little fictional app.

Starting Out

Let’s pretend you have been hired by LEMON custom home builders, and they need a way for clients to submit trouble tickets after their house is built. With a name like LEMON you know they are going to get swamped with problems so you are tasked with streamlining this process.

To create the absolute minimum, we can utilize the users table for the customer and create a new issue table. So let’s get started building these out.

Create a new app:

```
laravel new lemon-support
```

Next let’s create our issues model and migration from the artisan console command. We can create both in one by using the `make:model` command passing the `-m` or `--migration` flag.

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

If you open the `app/` folder you will see Issues.php. Then inside `database/migrations/` the new `create_issues_table` migration file.

Go ahead and open the app/User.php model file and define the issues relationship:

```
public function issues()
{
return $this->hasMany('issues');
}
```

This is a good time to edit your `.env` and change your database name if you don’t want to use the default homestead.

Now let’s fill out the migration files.

Creating Migrations

Open database/migrations/datetime_create_issues_table.php and adjust the up method to the following:

```
publicfunctionup()
{
  Schema::create('issues', function(Blueprint $table){
      $table->increments('id');
      $table->integer('user_id');
      $table->string('subject');
      $table->text('description');
      $table->timestamps();
  });
}
```

In this migration we are just adding a user_id relationship to the users table, a subject of the problem, and a description so the customers can give a detailed report.

Run the migration now:

```
php artisan migrate
```

It should output the following:

```
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrated: 2015_10_03_141020_create_issues_table
```

Next let's set up database seeds so we can have sample data to work with.

Building Database Seeds

Database seeds are a way of programmatically inserting data into the database and one advantage to using them is you can quickly get dummy data into your app. This is really beneficial when working on the user interface and if you are on a team.

Let's start by creating a seed for users. This can be generated using the following command:

```
phpartisanmake:seederUserTableSeeder
```

Go ahead and create one for issues as well:

```
phpartisanmake:seederIssueTableSeeder
```

Now open the database/seeds/DatabaseSeeder.php file and adjust the run method to the following:

```
publicfunctionrun()
```

```
{
  Model::unguard();

$this->call(UserTableSeeder::class);
$this->call(IssueTableSeeder::class);

  Model::reguard();
}
```

Now before these seed classes are useful they need instructions on what to insert. Let's use model factories for that.

Creating Model Factories

When creating seed data in the past you could utilize Eloquent or Laravel's query builder and that way is still supported. However, with the introduction of model factories you can use them to build out "dummy" models which can be used for both seed data and in testing.

Open database/factories/ModelFactory.php and you will see a default one already defined:

```
$factory->define(App\User::class, function(Faker\Generator $faker){
return [
'name' => $faker->name,
'email' => $faker->email,
'password' => bcrypt(str_random(10)),
'remember_token' => str_random(10),
    ];
});
```

To explain this, we are defining the "App\User::class" model as the first parameter and then a callback that defines the data that goes in the columns. This callback also injects Faker which is a PHP library that generates fake data. Faker is powerful and can be used for a number of different field types. Hereisanexample of theonesshown:

- $fake->name – "John Smith"
- $faker->email – "tkshlerin@collins.com"

Please Note: If your app is going to send real email then you should utilize `$faker->safeEmail` instead of just `->email`. The reason for this is `->email` can potentially generate a real email, where safeEmail uses example.org that is reserved by RFC2606 for testing purposes.

Now, let's create a new factory for our Issues model. Here is the completed code:

```
$factory->define(App\Issues::class, function(Faker\Generator $faker){
return [
'subject' => $faker->sentence(5),
'description' => $faker->text(),
    ];
});
```

With this definition, we generate a sentence with five words for the subject, and finally some dummy text for the description.

Let's now switch back to our seed classes and use these factories to generate the data.

Open the UserTableSeeder and adjust the run method:

```
publicfunctionrun()
{
  factory(App\User::class, 2)->create()->each(function($u){
    $u->issues()->save(factory(App\Issues::class)->make());
  });
}
```

This may look overwhelming so lets break it down. The `factory(App\User::class, 2)->create()` in English says, build a User class and we want 2 users then create them by saving in the database.

Next is a collection each which will run through "each" of the created users. Finally, inside each save an issue associated with the created user.

Run the artisan command:

```
$ php artisan migrate --seed
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrated: 2015_10_03_141020_create_issues_table
Seeded: UserTableSeeder
```

Now our database will have our tables and they will contain the sample data generated from the model factory.

Testing with Laravel Model Factories

A major benefit to having model factories is we can now utilize these in our testing suite. Create a new test for our Issues:

```
phpartisanmake:testIssuesTest
```

Open tests/IssuesTest.php and add a new method to test the creation of an issue:

```
<?php

useIlluminate\Foundation\Testing\WithoutMiddleware;
useIlluminate\Foundation\Testing\DatabaseMigrations;
useIlluminate\Foundation\Testing\DatabaseTransactions;

classIssuesTestextendsTestCase
{
useDatabaseTransactions;
```

```
publicfunctiontestIssueCreation()
{
      factory(App\Issues::class)->create([
'subject' =>'NAIL POPS!'
      ]);

$this->seeInDatabase('issues', ['subject' =>'NAIL POPS!']);
    }
}
```

In this test, we are using the `DatabaseTransactions` trait so that each test is wrapped in a transaction. Inside testIssueCreation is the addition of our model factory with a new feature. The create method is passing an array with the column name and a preset value. This allows you to override what is stored in the original model factory definition.

For the actual test, we are using `seeInDatabase` which as you can guess searches the DB and returns green if the given column and value are found.

Now that we have seen how these model factories can be used for both seeding and in testing, lets look at some of the useful features it offers.

Other Model Factory Features

During preparation for the first meeting with LEMON homes, you notice a problem with your original setup. You've forgotten to add a way of marking issues as resolved. That would be embarrassing.

Multiple Factory Types

After adding a new migration for a "completed" field the model factory needs to be adjusted, but it would be handy to have a separate one so you can make a completed issue easily. Define a new factory like this:

```
$factory->defineAs(App\Issues::class, 'completed',
function($faker)use($factory){
    $issue = $factory->raw(App\Issues::class);

return array_merge($issue, ['completed' =>true]);
});
```

This one uses a name as the second parameter, in this case "completed", and in the callback we make a new issue and merge our completed column into it.

If you'd like to create a 'completed' issue now all that is required is running:

```
$completedIssue = factory(App\Issue::class, 'completed')->make();
```

Factory make vs create methods

In that last example, you may have noticed I called `->make()` instead of `->create()` as was done previously. These two methods do two different things, “create” attempts to store it in the database and is the same as saving in Eloquent. Make on the other hand creates the model but doesn’t actually insert. If you are familiar with Eloquent it’s the comparable to:

```
$issue = new \App\Issue(['subject' =>'My Subject']);
```

Closing

As you can see model factories are a powerful feature in Laravel and help to simplify both testing and database seeding.

Also, check out the Laravel Model Factory States in v5.3.17.