Migration và Seeding trong Laravel

Laravel Migration - Tạo phiên bản Database

Laravel Migration được sử dụng để quản lý các phiên bản Database giúp làm việc nhóm dễ dàng hơn và thuận tiện hơn trong việc thay đổi, chia sẻ Database

Để tạo Migration trong Laravel, chúng ta sử dụng câu lệnh Artisan sau:

```
php artisan make:migration ten_migration
```

Khi đó một file migration sẽ được tạo ra trong thư mục database/migrations, trong tên các file migration có chứa thông tin thời gian tạo ra giúp cho Laravel sắp xếp và xác định được các file migration tại các thời điểm cần thiết

File migration chứa Migration class với hai phương thức `up()` và `down()`

Phương thức `up()` sử dụng để thêm các bảng, cột hoặc tạo một index cho cột nào đó trong CSDL

Phương thức `down()` sử dụng để làm những việc ngược lại với phương thức `up()`

Ví dụ:

```
use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Migrations\Migration;

class CreateProductsTable extends Migration
{
    /**
```

```
     * Run the migrations.
     *
     * @return void
     */
    public function up()
    {

    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down()
    {

    }
}
```

Để chạy tất cả các file migrations, chúng ta dùng câu lệnh sau

```
php artisan migrate
```

Các migration file có thể ảnh hưởng đến cấu trúc database hiện tại, do đó khi chạy lệnh trên có thể nó sẽ thông báo xác nhận chạy hay không

Nếu muốn bắt buộc tạo, chúng ta hãy sử dụng thêm tuỳ chọn `--force`

```
php artisan migrate --force
```

Trong quá trình migration có thể bạn muốn quay trở lại cấu trúc CSDL trước lúc thực thi các migration, hoàn toàn có thể được bằng cách sử dụng rollback và reset.

Rollback thì cần cung cấp thêm tùy chọn --step là rollback lại bao nhiêu lần

Reset sẽ đưa cấu trúc CSDL về thời điểm chưa có bất kỳ một migration nào

```
php artisan migrate:rollback --step=2
```

```
php artisan migrate:reset
```

Laravel cung cấp thêm lệnh migrate:refresh giúp reset về thời điểm chưa có migration và tạo lại từ đầu

Laravel Schema - Thao tác với bảng và cột trong Database

Laravel Schema là một Class rất quan trọng trong thiết kế các thực thể cơ sở dữ liệu mà không cần phải viết bất kỳ một dòng lệnh SQL nào

Schema giúp xây dựng tạo ra, xóa, cập nhật các cột cũng như bảng trong cơ sở dữ liệu

Các thiết lập khác như index, khóa chính, khóa phụ cũng có thể thực hiện với Schema

Thao tác với bảng

Phương thức create()

```
Schema::create($table, $callback);
```

- **$table**: Tên table
- **$callback**: Closure chứa các khai báo cột trong Table

```
Schema::create('products', function (Blueprint $table) {
    $table->increments('id');
```

```
    $table->string('name');
    $table->integer('price');
    $table->mediumText('content');
    $table->boolean('active');
    $table->timestamps();
});
```

Khi tạo bảng có thể bảng đó đã tồn tại, chúng ta cần kiểm tra trước khi tạo, tương tự với các cột trong bảng thông qua các phương thức `hasTable()` và `hasColumn()`:

```
if (!Schema::hasTable('products')) {
        Schema::create('products', function (Blueprint
$table) {
            $table->increments('id');
            $table->string('name');
            $table->integer('price');
            $table->mediumText('content');
            $table->boolean('active');
            $table->timestamps();
        });
}

if (Schema::hasColumn('products', 'warranty_info')) {
    //
}
```

Phương thức rename() - drop() - dropIfExists()

```
// Đổi tên bảng products thành san_pham
Schema::rename('products', 'san_pham');
// Xóa bảng products
Schema::drop('products');
// Xóa bảng products nếu nó tồn tại trong CSDL
Schema::dropIfExists('products');
```

Thao tác với cột dữ liệu

```
$table->increments('id');
$table->string('name');
$table->integer('price');
$table->mediumText('content');
$table->boolean('active');
$table->timestamps();
```

Laravel cũng cho phép thay đổi các tham số trong dạng cột, tên cột... để thực hiện trước tiên phải cài đặt một gói doctrine/dbal, sử dụng lệnh composer

```
composer require doctrine/dbal
```

Thay đổi thuộc tính cột

```
Schema::table('products', function (Blueprint $table) {
    $table->string('name', 50)->change();
    $table->string('warranty')->nullable()->change();
});
```

Thay đổi tên cột

```
Schema::table('products', function (Blueprint $table) {
    $table->renameColumn('warranty_info',
'thong_tin_bao_hanh');
});
```

Xoá cột trong bảng

```
Schema::table('products', function (Blueprint $table) {
    $table->dropColumn(['price', 'content',
'warranty_info']);
```

```
});
```

Thao tác liên quan đến index

Để tạo một index chúng ta sử dụng cú pháp:

```
$table->string('product_code')->unique();
```

Hoặc tạo index sau khi đã tạo cột

```
$table->unique('product_code');
```

Tạo index cho nhiều cột và đặt tên cho index

```
$table->index(['price', 'product_code'],
'price_code_index');
```

Danh sách các dạng index cho phép

Phương thức	**Mô tả**
$table->primary('id');	Thêm khóa chính
$table->primary(['first', 'last']);	Thêm nhiều khóa chính
$table->unique('email');	Thêm Unique Index
$table->unique('state', 'my_index_name');	Thêm index với name
$table->unique(['first', 'last']);	Thêm nhiều Unique Index
$table->index('state');	Thêm index cơ bản

Xóa Index khỏi Database

Phương thức	Mô tả
$table->dropPrimary('users_id_primary');	Xoá khoá chính
$table->dropUnique('users_email_unique');	Xóa Unique Index
$table->dropIndex('geo_state_index');	Xóa Index cơ bản

Thiết lập khóa ngoại

```
Schema::table('products', function (Blueprint $table) {
    $table
    ->integer('admin_id')
    ->unsigned();
    $table
    ->foreign('admin_id')
    ->references('id')
    ->on('admin');
});
```

Trong bảng **products** có trường **admin_id** là trường chứa ID của quản trị viên đã tạo sản phẩm, trường **admin_id** sẽ tham chiếu sang bảng admin chứa danh sách các quản trị viên hệ thống

Khi đã thiết lập khóa ngoại, chúng ta có thể thiết lập các hành động như update hay delete

```
$table->foreign('admin_id')
      ->references('id')->on('admin')
      ->onDelete('cascade');
```

Chúng ta cũng có thể xóa khóa ngoại bằng câu lệnh

```
$table->dropForeign('products_admin_id_foreign');
```

Tên khóa ngoại giống như quy ước đặt tên trong index, tên bảng + tên cột + hậu tố _foreign. Laravel cho phép enable hoặc disable khóa ngoại trong CSDL bằng lệnh

```
Schema::enableForeignKeyConstraints();
Schema::disableForeignKeyConstraints();
```

Laravel Seeding - Đưa dữ liệu vào Database

Có một số tình huống chúng ta cần đưa dữ liệu hàng loạt vào database như tạo dữ liệu test ứng dụng, đổ dữ liệu hàng loạt phục vụ cho chuyển đổi hệ thống…

Laravel cung cấp các phương thức đơn giản để thực hiện bằng cách tạo các các lớp mở rộng của lớp Seeder

Các lớp Seeder được lưu trong thư mục **database/seeds**.

Chúng ta có thể tạo bằng tay một file có phần mở rộng php hoặc tạo tự động một Seeder bằng cách sử dụng lệnh artisan như sau:

```
php artisan make:seeder TenSeeder
```

Mỗi lớp Seeder sẽ có một phương thức `run()`, phương thức này sẽ được thực thi khi thực hiện câu lệnh

Trong folder **database/seeds** sẽ có 1 seeder mặc định là **DatabaseSeeder**

```
<?php

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

class DatabaseSeeder extends Seeder
```

```
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        DB::table('products')->insert([
           ['name' => str_random(255), 'price' =>
'300000', 'content' => str_random(255), 'active' => 1],
           ['name' => str_random(255), 'price' =>
'400000', 'content' => str_random(255), 'active' => 1],
           ['name' => str_random(255), 'price' =>
'500000', 'content' => str_random(255), 'active' => 1],
           ['name' => str_random(255), 'price' =>
'600000', 'content' => str_random(255), 'active' => 1],
        ]);
    }
}
```

Trong phương thức run() của Seeder mặc định DatabaseSeeder chúng ta cũng có thể gọi đến các Seeder khác do chúng ta tạo ra:

```
public function run()
{
    $this->call(UsersTableSeeder::class);
    $this->call(PostsTableSeeder::class);
    $this->call(CommentsTableSeeder::class);
}
```

Để chạy tất cả Seeder, chúng ta sử dụng câu lệnh sau:

```
php artisan db:seed
```

Để chạy 1 seeder cụ thể, chúng ta sử dụng câu lệnh sau:

```
php artisan db:seed --class=TenSeeder
```

Faker - Tạo dữ liệu test cho ứng dụng

Laravel Seeding là một trong những tính năng rất hay của Laravel giúp đưa dữ liệu vào Cơ sở dữ liệu, tuy nhiên khi tạo dữ liệu kiểm thử ứng dụng, chúng ta cần tạo ra nhiều bản ghi với các dữ liệu được tạo ra ngẫu nhiên và Faker là một thư viện tuyệt vời cho mục đích này. Faker là một thư viện PHP được sử dụng để tạo ra dữ liệu giả cho mục đích kiểm thử ứng dụng

Cài đặt Faker trong Laravel

Laravel 5 trở lên cài đặt Faker mặc định, do đó bạn không cần cài đặt gì thêm. Bạn có thể kiểm tra các thư viện được cài đặt trong file `composer.json` nằm trong thư mục gốc.

```
"require-dev": {
    "facade/ignition": "^2.5",
    "fakerphp/faker": "^1.9.1",
    "laravel/sail": "^1.0.1",
    "mockery/mockery": "^1.4.4",
    "nunomaduro/collision": "^5.10",
    "phpunit/phpunit": "^9.5.10"
}
```

Faker có thể sử dụng để sinh ra các dữ liệu giả như số, văn bản text, thông tin người dùng (tên, email, giới tính...), địa chỉ, số điện thoại, dữ liệu thời gian, màu sắc, ảnh

Thông tin chi tiết về thư viện:
https://github.com/fzaninotto/Faker#formatters

Tạo dữ liệu test bằng Faker

Faker có rất nhiều dạng dữ liệu, chi tiết chúng ta có xem link chi tiết của thư viện.

Sau đây là 1 ví dụ cơ bản

```
$faker = Faker\Factory::create();
$limit = 1000;
$customers = [];
for ($i = 0; $i < $limit; $i++) {
    $customers[$i] = [
        'name' => $faker->name,
        'email' => $faker->unique()->email,
        'phone' => $faker->phoneNumber,
        'website' => $faker->domainName,
        'age' => $faker->numberBetween(20,100),
        'address' => $faker->address
    ];
}
```

Tham khảo thêm 1 số kiểu dữ liệu khác

```
$faker->randomDigit;
$faker->numberBetween(1,100);
$faker->word;
$faker->paragraph;
$faker->lastName;
$faker->city;
$faker->year;
$faker->domainName;
$faker->creditCardNumber;
```

Chúng ta có thể chọn ngôn ngữ cho Faker

```
$faker = Faker\Factory::create('fr_FR');
```

Faker support chúng ta rất nhiều ngôn ngữ khác nhau (Trừ Tiếng Việt)

Model Factory - Tạo dữ liệu test phức tạp

Định nghĩa Model Factory

```
$factory->define(App\Category::class, function
(Faker\Generator $faker) {
    return [
        'name' => implode(' ', $faker->words(2)),
        'description' => $faker->sentence(),
    ];
});

$factory->define(App\Topic::class, function
(Faker\Generator $faker) {
    return [
        'category_id' => null,
        'user_id' => 1,
        'title' => $faker->sentence,
        'body' => $faker->paragraph(7),
        'views' => $faker->numberBetween(0, 10000),
        'created_at' => $faker->datetimeBetween('-5
months'),
    ];
});

$factory->define(App\Post::class, function
(Faker\Generator $faker) {
    return [
        'topic_id' => null,
        'user_id' => 1,
        'body' => $faker->sentence,
```

```
    ];
});
```

Phương thức `define()` với hai tham số, một là tên đầy đủ của model, hai là một **instance** của **Faker** và nó trả về một mảng các thuộc tính của đối tượng với dữ liệu có thể được tạo giả.

Tạo ra các đối tượng có dữ liệu giả bằng Model Factory

Sau khi định nghĩa xong chúng ta có thể tạo ra một đối tượng với các thuộc tính có dữ liệu giả bằng cách gọi phương thức `create()`

```
$category = factory(Category::class)->create();
```

Đối tượng với dữ liệu giả được tạo ra và create() cũng thực hiện ghi dữ liệu này xuống database luôn. Nếu muốn tạo ra nhiều hơn 1 đối tượng, truyền thêm tham số vào như sau:

```
$category = factory(Category::class, 5)->create();
```

Tạo 5 danh mục và cũng ghi luôn xuống database. Đôi khi chúng ta chỉ muốn tạo ra đối tượng dữ liệu giả mà không ghi xuống database, bạn có thể sử dụng phương thức make() thay cho create():

```
$category = factory(Category::class)->make();
```

Định nghĩa các dạng đối tượng khác nhau

Có thể bạn muốn tạo ra các dạng đối tượng khác nhau khi sử dụng Model Factory, ví dụ một bài viết dài và một bài viết ngắn, chúng ta cùng xem đoạn code sau:

```
$factory->defineAs(App\Post::class, 'short-post',
function ($faker) {
```

```
    return [
        'title' => $faker->sentence,
        'body' => $faker->paragraph
    ];
});

$factory->defineAs(App\Post::class, 'long-post', function
($faker) {
    return [
        'title' => $faker->sentence,
        'body' => implode("\n\n", $faker->paragraphs(10))
    ];
});
```

Tạo các đối tượng có quan hệ

Với các đối tượng có quan hệ, chúng ta có thể thực hiện tạo ra các đối tượng này

Ví dụ dưới đây tạo ra 5 danh mục và mỗi danh mục tạo ra 5 chủ đề liên quan.

```
factory(Category::class, 5)->create()->each(function($c)
{
    $c->topics()->saveMany(
        factory(Topic::class, 50)->create([
            'category_id' => $c->id
        ])
    );
});
```

Chú ý, đoạn code trên chỉ chạy được khi đối tượng Category đã định nghĩa topics() với mối quan hệ là hasMany()

Phương pháp làm việc theo nhóm

Trong thực tế, khi xuất hiện một yêu cầu mới hoặc chỉnh sửa một yêu cầu trước đó, các lập trình viên có thể phải thay đổi một số bảng trong database, tuy nhiên có thể nhiều các thành viên khác không biết việc này. Laravel Migrate ra đời nhằm mục đích như vậy, nó giúp cho làm việc theo nhóm có thể dễ dàng chia sẻ, thay đổi kiến trúc của cơ sở dữ liệu.

Các bước thực hiện để quản lý phiên bản cơ sở dữ liệu như sau:

Khi có yêu cầu thay đổi kiến trúc cơ sở dữ liệu (database schema) thực hiện thông qua Laravel Migrate:

- Tạo file migrate với câu lệnh Artisan
  Chú ý khi tạo ra file migrate, hệ thống đã tự động đưa thêm yếu tố thời gian vào tên file, do đó có thể biết được file migrate này tạo ra khi nào, chúng ta chỉ cần đặt tên file sao cho gợi nhớ ví dụ: create_users_table, alter_users_table…
- Các thay đổi cần thực hiện trên kiến trúc cơ sở dữ liệu cần được chuyển thành các câu lệnh và đưa vào phương thức up().
- Trong phương thức down() cần viết code để rollback lại những gì đã thực hiện trong phương thức up(), nếu không thực hiện công việc này cũng không thể quản lý phiên bản cơ sở dữ liệu một cách tự động được

Sau khi thực hiện viết hoặc thay đổi code trong ứng dụng, thực hiện commit code trong đó có cả file migrate.

Như vậy khi cần thiết chúng ta có thể quay lại phiên bản đã commit, khi đó trong thư mục **database/migrations** chứa các file migrate điều chỉnh kiến trúc cơ sở dữ liệu đến thời điểm cần, chúng ta chỉ cần thực hiện chạy php artisan migrate là nó sẽ build lại toàn bộ kiến trúc cơ sở dữ liệu

Chú ý: Trên đây chúng ta mới chỉ nói đến bộ khung cho cơ sở dữ liệu, các dữ liệu kiểm thử cho ứng dụng chúng ta cũng cần thực hiện thông

qua Laravel Seeding và đưa vào danh sách các file cần commit cho lần sửa đổi này.

Như vậy, khi quay lại phiên bản cũ chúng ta hoàn toàn có thể chạy câu lệnh `php artisan db:seed` sau khi đã thực hiện migrate để có dữ liệu test.