Course - Laravel Framework

Migration

Các migration quản lý phiên bản cho cơ sở dữ liệu của bạn, cho phép đội của bạn hoạch định và chia sẻ schema cơ sở dữ liệu của ứng dụng. Nếu bạn đã từng phải yêu cầu đồng đội thêm một cột vào cơ sở dữ liệu local của họ theo cách thủ công sau khi pull các thay đổi của bạn từ phần quản lý mã source, thì bạn đã phải đối mặt với việc giả quyết các phiên bản cơ sở dữ liệu.

Tags: migration, laravel

Giới thiệu

Các migration quản lý phiên bản cho cơ sở dữ liệu của bạn, cho phép đội của bạn hoạch định và chia sẻ schema cơ sở dữ liệu của ứng dụng. Nếu bạn đã từng phải yêu cầu đồng đội thêm một cột vào cơ sở dữ liệu local của họ theo cách thủ công sau khi pull các thay đổi của bạn từ phần quản lý mã source, thì bạn đã phải đối mặt với việc giả quyết các phiên bản cơ sở dữ liệu.

Laravel có facade Schema, nó hỗ trợ tạo và thao tác các bảng trên tất cả các hệ thống cơ sở dữ liệu được hỗ trợ của Laravel. Thông thường, migration sẽ sử dụng facade này để tạo và sửa đổi các bảng và cột cơ sở dữ liệu.

Tạo ra các migration

Bạn có thể sử dụng lệnh Artisan `make:migration` để tạo một migration cơ sở dữ liệu. Migration mới sẽ được đặt trong thư mục *database/migration* của bạn. Mỗi tên tập tin migration đều chứa một đoạn mô tả thời gian cho phép Laravel xác định thứ tự của chạy lệnh tạo DB:

```
php artisan make:migration create_flights_table
```

Laravel sẽ sử dụng tên của migration để cố gắng suy đoán tên của bảng và migration có tạo một bảng mới hay không. Nếu Laravel có thể xác định tên bảng từ tên migration, Laravel sẽ điền trước tập tin migration đã tạo với bảng được chỉ định. Nếu không, bạn có thể chỉ định bảng trong tập tin migration theo cách thủ công.

Nếu bạn muốn chỉ định một đường dẫn tùy chỉnh cho migration đã tạo, bạn có thể sử dụng tùy chọn `--path` khi thực hiện lệnh `make:migration`. Đường dẫn đã cho phải liên quan đến đường dẫn cơ sở của ứng dụng của bạn.

> Những migration stub có thể được tùy chỉnh bằng cách sử dụng stub publishing.

Tích ép migration

Khi bạn xây dựng ứng dụng của mình, bạn có thể tích lũy ngày càng nhiều migration theo thời gian. Điều này có thể dẫn đến thư mục *database/migrations* của bạn trở nên cồng kềnh với hàng trăm migration. Nếu bạn muốn, bạn có thể "ép" migration của mình trở thành một tập tin SQL duy nhất. Để bắt đầu, hãy thực hiện lệnh `schema:dump`:

```
php artisan schema:dump
```

Kết xuất lược đồ cơ sở dữ liệu hiện tại và cắt bỏ tất cả các migration hiện có ...

```
php artisan schema:dump --prune
```

Khi bạn thực thi lệnh này, Laravel sẽ ghi một tập tin "lược đồ" vào thư mục `database/schema` của ứng dụng của bạn. Bây giờ, khi bạn cố gắng chạy migration cơ sở dữ liệu của mình và không có migration nào khác được thực hiện, Laravel sẽ thực thi các câu lệnh SQL của tập tin schema trước. Sau khi thực hiện các câu lệnh của tập tin schema, Laravel sẽ thực thi bất kỳ migration nào còn lại mà không phải là một phần của kết xuất lược đồ.

Bạn nên chốt tập tin schema cơ sở dữ liệu của mình cho source nguồn để các nhà phát triển mới khác trong đội của bạn có thể nhanh chóng tạo cấu trúc cơ sở dữ liệu ban đầu cho ứng dụng của bạn.

> **Chú ý:** Tích ép migration chỉ khả dụng cho cơ sở dữ liệu MySQL, PostgreSQL và SQLite và sử dụng dòng lệnh cli của cơ sở dữ liệu. Các kết xuất schema có thể không được khôi phục vào cơ sở dữ liệu SQLite trong bộ nhớ.

Cấu trúc migration

Một lớp di chuyển chứa hai phương thức: `up` và `down`. Phương thức `up` được sử dụng để thêm bảng, cột hoặc chỉ mục mới vào cơ sở dữ liệu của bạn, trong khi phương thức `down` sẽ đảo ngược các thao tác được thực hiện bởi phương thức `up`.

Trong cả hai phương thức này, bạn có thể sử dụng schema builder của Laravel để tạo và sửa đổi các bảng một cách rõ ràng. Để tìm hiểu về tất cả các phương thức có sẵn trên `Schema` builder, hãy xem tài liệu của nó. Ví dụ: bản migration sau đây sẽ tạo ra một bảng `flights`:

```
<?php

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

class CreateFlightsTable extends Migration
```

```
{
  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('flights', function (Blueprint $table) {
      $table->id();
      $table->string('name');
      $table->string('airline');
      $table->timestamps();
    });
  }

  /**
   * Reverse the migrations.
   *
   * @return void
   */
  public function down()
  {
    Schema::drop('flights');
  }
}
```

Các bản migration nặc danh

Như bạn có thể đã nhận thấy trong ví dụ trên, Laravel sẽ tự động gán tên class cho tất cả các migration mà bạn tạo bằng lệnh `make:migration`. Tuy nhiên, nếu muốn, bạn có thể trả về một class ẩn danh từ tập tin migration của mình. Điều này đặc biệt hữu ích nếu ứng dụng của bạn tích lũy nhiều migration và hai trong số chúng có xung đột tên class:

```
<?php
use Illuminate\Database\Migrations\Migration;
return new class extends Migration
```

```
{
  //
};
```

Cài đặt kết nối migation

Nếu migration của bạn tương tác với kết nối cơ sở dữ liệu khác với kết nối cơ sở dữ liệu mặc định của ứng dụng, thì bạn nên đặt thuộc tính `$connection` của migration như sau:

```
/**
 * The database connection that should be used by the migration.
 *
 * @var string
 */
protected $connection = 'pgsql';

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

Chạy các migration

Để chạy tất cả các migration chưa hoàn thành của bạn, hãy thực hiện lệnh Artisan `migrate`:

```
php artisan migrate
```

Nếu bạn muốn xem những migration nào đã chạy cho đến nay, thì bạn có thể sử dụng lệnh Artisan `migrate:status`:

```
php artisan migrate:status
```

Chạy bắt buộc các migration trong production

Một số hoạt động migration có tính chất phá hoại, có nghĩa là chúng có thể khiến bạn mất dữ liệu. Để bảo vệ bạn khỏi việc chạy các lệnh này với cơ sở dữ liệu trong môi trường product của bạn, thì bạn sẽ được nhắc xác nhận trước khi các lệnh được thực thi. Để buộc các lệnh chạy mà không có lời nhắc, hãy sử dụng cờ `--force`:

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

Khôi phục lại migration

Để khôi phục hoạt động migration mới nhất, bạn có thể sử dụng lệnh Artisan `rollback`. Lệnh này sẽ quay trở lại "đợt" migration cuối cùng, có thể bao gồm nhiều tập tin migration:

```
php artisan migrate:rollback
```

Bạn có thể khôi phục một số lần hạn chế các migration bằng cách cung cấp tùy chọn `step` cho lệnh `rollback`. Ví dụ: lệnh sau sẽ khôi phục lại năm lần di chuyển cuối cùng:

```
php artisan migrate:rollback --step=5
```

Lệnh `migrate:reset` sẽ khôi phục tất cả các migration của ứng dụng của bạn:

```
php artisan migrate:reset
```

Chạy và khôi phục migrate bằng một dòng lệnh

Lệnh `migrate:refresh` sẽ khôi phục tất cả các migration của bạn và sau đó thực thi lệnh `migrate`. Lệnh này tạo lại toàn bộ cơ sở dữ liệu của bạn một cách hiệu quả:

```
php artisan migrate:refresh


// Refresh the database and run all database seeds...
php artisan migrate:refresh --seed
```

Bạn có thể khôi phục và chạy migrate lại một migration hạn chế bằng cách cung cấp tùy chọn step cho lệnh refresh. Ví dụ: lệnh sau sẽ khôi phục và migrate lại năm lần bản migration cuối cùng:

```
php artisan migrate:refresh --step=5
```

Xóa bỏ tất cả các bảng và migrate

Lệnh `migrate:fresh` sẽ xóa tất cả các bảng khỏi cơ sở dữ liệu và sau đó thực thi lệnh `migrate`:

```
php artisan migrate:fresh
```

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

> **Chú ý:** Lệnh `migrate:fresh` sẽ loại bỏ tất cả các bảng cơ sở dữ liệu bất kể tiền tố của chúng là gì. Lệnh này nên được sử dụng một cách thận trọng khi phát triển trên cơ sở dữ liệu được chia sẻ với các ứng dụng khác.

Bảng dữ liệu

Tạo bảng dữ liệu

Để tạo một bảng cơ sở dữ liệu mới, hãy sử dụng phương thức create trên facade `Schema`. Phương thức `create` chấp nhận hai đối số: đối số đầu tiên là tên của bảng, trong khi đối số thứ hai là một hàm xử lý nhận một đối tượng Blueprint có thể được sử dụng để xác định bảng mới:

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

Schema::create('users', function (Blueprint $table) {
  $table->id();
  $table->string('name');
  $table->string('email');
  $table->timestamps();
});
```

Khi tạo bảng, bạn có thể sử dụng bất kỳ phương thức cột nào của schema builder để tạo các cột của bảng.

Kiểm tra sự tồn tại của bảng/cột

Bạn có thể kiểm tra sự tồn tại của một bảng hoặc cột bằng các phương thức `hasTable` và `hasColumn`:

```
if (Schema::hasTable('users')) {
  // The "users" table exists...
}

if (Schema::hasColumn('users', 'email')) {
  // The "users" table exists and has an "email" column...
}
```

Kết nối database và các tùy chọn cho bảng

Nếu bạn muốn thực hiện thao tác schema trên một kết nối cơ sở dữ liệu không phải là kết nối mặc định của ứng dụng, thì hãy sử dụng phương thức `connection`:

```
Schema::connection('sqlite')->create('users', function (Blueprint $table) {
  $table->id();
});
```

Ngoài ra, một số thuộc tính và phương thức khác có thể được sử dụng để xác định các khía cạnh khác của việc tạo bảng. Thuộc tính `engine` có thể được sử dụng để chỉ định engine lưu trữ của bảng khi sử dụng MySQL:

```
Schema::create('users', function (Blueprint $table) {
  $table->engine = 'InnoDB';

  // ...
});
```

Các thuộc tính `charset` và `collation` có thể được sử dụng để chỉ định bộ ký tự và sự đối chiếu cho bảng đã tạo khi sử dụng MySQL:

```
Schema::create('users', function (Blueprint $table) {
```

```
    $table->charset = 'utf8mb4';
    $table->collation = 'utf8mb4_unicode_ci';

    // ...
});
```

Phương thức `temporary` có thể được sử dụng để chỉ ra rằng bảng phải là "temporary". Các bảng tạm thời chỉ hiển thị với session cơ sở dữ liệu của kết nối hiện tại và tự động bị loại bỏ khi kết nối bị đóng:

```
Schema::create('calculations', function (Blueprint $table) {
    $table->temporary();

    // ...
});
```

Cập nhật bảng

Phương thức `table` trên facade `Schema` có thể được sử dụng để cập nhật các bảng hiện có. Giống như phương thức `create`, phương thức `table` chấp nhận hai đối số: tên của bảng và một hàm xử lý nhận một đối tượng Blueprint mà bạn có thể sử dụng để thêm cột hoặc chỉ mục vào bảng:

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

Schema::table('users', function (Blueprint $table) {
    $table->integer('votes');
});
```

Đổi tên hoặt xóa bỏ bảng

Để đổi tên một bảng cơ sở dữ liệu hiện có, hãy sử dụng phương thức rename:

```
use Illuminate\Support\Facades\Schema;
```

```
Schema::rename($from, $to);
```

Để xóa bỏ một bảng hiện có, bạn có thể sử dụng các phương thức `drop` hoặc `dropIfExists`:

```
Schema::drop('users');

Schema::dropIfExists('users');
```

Đổi tên bảng bằng khóa ngoại

Trước khi đổi tên bảng, bạn nên xác minh rằng bất kỳ ràng buộc khóa ngoại nào trên bảng đều có tên rõ ràng trong tập tin migrate của bạn thay vì để Laravel gán tên dựa trên quy ước. Nếu không, tên ràng buộc khóa ngoại sẽ tham chiếu đến tên bảng cũ.

Các cột dữ liệu

Tạo các cột dữ liệu

Phương thức table trên facade Schema có thể được sử dụng để cập nhật các bảng hiện có. Giống như phương thức create, phương thức table chấp nhận hai đối số: tên của bảng và một hàm xử lý nhận một đối tượng `Illuminate\Database\Schema\Blueprint` mà bạn có thể sử dụng để thêm cột vào bảng:

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

Schema::table('users', function (Blueprint $table) {
  $table->integer('votes');
});
```

Các kiểu cột được hỗ trợ

Kế hoạch chi tiết (Blueprint) của schema builder cung cấp nhiều phương thức tương ứng

với các loại cột khác nhau mà bạn có thể thêm vào các bảng cơ sở dữ liệu của mình. Từng phương thức hỗ trợ được liệt kê trong bảng dưới đây:

`bigIncrements()`

Phương thức `bigIncrements` sẽ tạo một cột tương đương `UNSIGNED BIGINT` (khóa chính) tự động tăng dần:

```
$table->bigIncrements('id');
```

`bigInteger()`

Phương thức `bigInteger` tạo một cột tương đương `BIGINT`:

```
$table->bigInteger('votes');
```

`binary()`

Phương thức `binary` tạo một cột tương đương `BLOB`:

```
$table->binary('photo');
```

`boolean()`

Phương thức `boolean` tạo một cột tương đương `BOOLEAN`:

```
$table->boolean('confirmed');
```

`char()`

Phương thức `char` tạo một cột tương đương `CHAR` có độ dài cho trước:

```
$table->char('name', 100);
```

`dateTimeTz()`

Phương thức `dateTimeTz` tạo cột tương đương `DATETIME` (với múi giờ) có độ chính xác

tùy chọn (tổng số):

```
$table->dateTimeTz('created_at', $precision = 0);
```

dateTime()

Phương thức `dateTime` tạo một cột tương đương `DATETIME` có độ chính xác tùy chọn (tổng số):

```
$table->dateTime('created_at', $precision = 0);
```

date()

Phương thức `date` tạo một cột tương đương `DATE`:

```
$table->date('created_at');
```

decimal()

Phương thức `decimal` tạo ra một cột tương đương `DECIMAL` có độ chính xác đã cho (tổng số) và tỷ lệ (chữ số thập phân):

```
$table->decimal('amount', $precision = 8, $scale = 2);
```

double()

Phương thức `double` sẽ tạo ra một cột tương đương `DOUBLE` có độ chính xác đã cho (tổng số) và tỷ lệ (chữ số thập phân):

```
$table->double('amount', 8, 2);
```

enum()

Phương thức `enum` tạo một cột tương đương `ENUM` với các giá trị hợp lệ đã cho:

```
$table->enum('difficulty', ['easy', 'hard']);
```

float()

Phương thức `float` tạo một cột tương đương `FLOAT` với độ chính xác (tổng số) và tỷ lệ (chữ số thập phân) đã cho:

```
$table->float('amount', 8, 2);
```

foreignId()

Phương thức `foreignId` tạo một cột tương đương `UNSIGNED BIGINT`:

```
$table->foreignId('user_id');
```

foreignIdFor()

Phương thức `foreignIdFor` thêm một cột tương đương `{column}_id UNSIGNED BIGINT` cho một model nào đó:

```
$table->foreignIdFor(User::class);
```

foreignUuid()

Phương thức `foreignUuid` tạo một cột `UUID` tương đương:

```
$table->foreignUuid('user_id');
```

geometryCollection()

Phương thức `geometryCollection` tạo ra một cột tương đương `GEOMETRYCOLLECTION`:

```
$table->geometryCollection('positions');
```

geometry()

Phương thức `geometry` tạo ra một cột tương đương `GEOMETRY`:

```php
$table->geometry('positions');
```

id()

Phương thức `id` là một bí danh của phương thức `bigIncrements`. Theo mặc định, phương thức sẽ tạo một cột `id`; tuy nhiên, bạn có thể chuyển một tên cột nếu bạn muốn gán một tên khác cho cột:

```php
$table->id();
```

increments()

Phương thức `increments` sẽ tạo ra một cột tự động tăng dần tương đương `UNSIGNED INTEGER` làm khóa chính:

```php
$table->increments('id');
```

integer()

Phương thức `integer` tạo một cột tương đương `INTEGER`:

```php
$table->integer('votes');
```

ipAddress()

Phương thức `ipAddress` tạo một cột tương đương `VARCHAR`:

```php
$table->ipAddress('visitor');
```

json()

Phương thức `json` tạo một cột tương đương `JSON`:

```php
$table->json('options');
```

jsonb()

Phương thức `jsonb` tạo một cột tương đương `JSONB`:

```
$table->jsonb('options');
```

`lineString()`

Phương thức `lineString` tạo một cột tương đương `LINESTRING`:

```
$table->lineString('positions');
```

`longText()`

Phương thức `longText` tạo một cột tương đương `LONGTEXT`:

```
$table->longText('description');
```

`macAddress()`

Phương thức `macAddress` tạo một cột dùng để chứa địa chỉ `MAC`. Một số hệ thống cơ sở dữ liệu, chẳng hạn như PostgreSQL, có kiểu cột dành riêng cho kiểu dữ liệu này. Các hệ thống cơ sở dữ liệu khác sẽ sử dụng một cột tương đương chuỗi câu:

```
$table->macAddress('device');
```

`mediumIncrements()`

Phương thức `mediumIncrements` tạo cột tương đương `UNSIGNED MEDIUMINT` tự động tăng dần làm khóa chính:

```
$table->mediumIncrements('id');
```

`mediumInteger()`

Phương thức `mediumInteger` tạo một cột tương đương `MEDIUMINT`:

```
$table->mediumInteger('votes');
```

mediumText()

Phương thức `mediumText` tạo một cột tương đương `MEDIUMTEXT`:

```
$table->mediumText('description');
```

morphs()

Phương thức `morphs` là một phương thức tiện lợi thêm một cột tương đương `{column}_id UNSIGNED BIGINT` và một cột tương đương `VARCHAR {column}_type`.

Phương pháp này nhằm mục đích được sử dụng khi xác định các cột cần thiết cho mối quan hệ Eloquent đa hình. Trong ví dụ sau, các cột `taggable_id` và `taggable_type` sẽ được tạo:

```
$table->morphs('taggable');
```

multiLineString()

Phương thức `multiLineString` tạo một cột tương đương `MULTILINESTRING`:

```
$table->multiLineString('positions');
```

multiPoint()

Phương thức `multiPoint` tạo ra một cột `MULTIPOINT` tương đương:

```
$table->multiPoint('positions');
```

multiPolygon()

Phương thức `multiPolygon` tạo ra một cột tương đương `MULTIPOLYGON`:

```
$table->multiPolygon('positions');
```

nullableTimestamps()

Phương thức `nullableTimestamps` là một bí danh của phương thức `timestamps`:

```
$table->nullableTimestamps(0);
```

`nullableMorphs()`

Phương thức này tương tự như phương thức morphs; tuy nhiên, các cột được tạo sẽ là "nullable":

```
$table->nullableMorphs('taggable');
```

`nullableUuidMorphs()`

Phương thức này tương tự như phương thức `uuidMorphs`; tuy nhiên, các cột được tạo sẽ là "nullable":

```
$table->nullableUuidMorphs('taggable');
```

`point()`

Phương thức `point` tạo một cột tương đương `POINT`:

```
$table->point('position');
```

`polygon()`

Phương thức `polygon` tạo một cột tương đương `POLYGON`:

```
$table->polygon('position');
```

`rememberToken()`

Phương thức `rememberToken` tạo ra một cột tương đương `VARCHAR(100)` và có thể nullable, nhằm mục đích lưu trữ mã token xác thực "remember me" hiện tại:

```
$table->rememberToken();
```

set()

Phương thức `set` tạo một cột tương đương `SET` với danh sách các giá trị hợp lệ đã cho:

```
$table->set('flavors', ['strawberry', 'vanilla']);
```

smallIncrements()

Phương thức `smallIncrements` tạo một cột tương đương `UNSIGNED SMALLINT` tự động tăng dần làm khóa chính:

```
$table->smallIncrements('id');
```

smallInteger()

Phương thức `smallInteger` tạo một cột tương đương `SMALLINT`:

```
$table->smallInteger('votes');
```

softDeletesTz()

Phương thức `softDeletesTz` thêm cột tương đương với `TIMESTAMP` (với múi giờ) có thể xóa được nullable với độ chính xác tùy chọn (tổng số). Cột này nhằm lưu trữ dấu thời gian `deleted_at` cho tính năng "soft delete" của Eloquent:

```
$table->softDeletesTz($column = 'deleted_at', $precision = 0);
```

softDeletes()

Phương thức `softDeletes` thêm cột có tên `deleted_at` tương đương `TIMESTAMP` và có thể null với độ chính xác tùy chọn (tổng số). Cột này nhằm lưu trữ dấu thời gian `deleted_at` cho chức năng "xóa mềm" của Eloquent:

```
$table->softDeletes($column = 'deleted_at', $precision = 0);
```

string()

Phương thức `string` tạo một cột tương đương `VARCHAR` có độ dài đã cho:

```
$table->string('name', 100);
```

`text()`

Phương thức `text` tạo một cột `TEXT` tương đương:

```
$table->text('description');
```

`timeTz()`

Phương thức `timeTz` tạo cột tương đương `TIME` (với múi giờ) với độ chính xác tùy chọn (tổng số):

```
$table->timeTz('sunrise', $precision = 0);
```

`time()`

Phương thức `time` tạo một cột tương đương `TIME` với độ chính xác tùy chọn (tổng số):

```
$table->time('sunrise', $precision = 0);
```

`timestampTz()`

Phương thức `timestampTz` tạo cột tương đương `TIMESTAMP` (với múi giờ) với độ chính xác tùy chọn (tổng số):

```
$table->timestampTz('added_at', $precision = 0);
```

`timestamp()`

Phương thức `timestamp` sẽ tạo cột tương đương `TIMESTAMP` với độ chính xác tùy chọn (tổng số):

```
$table->timestamp('added_at', $precision = 0);
```

timestampsTz()

Phương thức `timestampsTz` tạo các cột tương đương với `created_at` và `updated_at` `TIMESTAMP` (với múi giờ) với độ chính xác tùy chọn (tổng số):

```
$table->timestampsTz($precision = 0);
```

timestamps()

Phương thức `timestamps` tạo các cột tương đương với `created_at` và `updated_at` `TIMESTAMP` với độ chính xác tùy chọn (tổng số):

```
$table->timestamps($precision = 0);
```

tinyIncrements()

Phương thức `tinyIncrements` tạo cột tương đương `UNSIGNED TINYINT` tự động tăng dần làm khóa chính:

```
$table->tinyIncrements('id');
```

tinyInteger()

Phương thức `tinyInteger` tạo một cột tương đương `TINYINT`:

```
$table->tinyInteger('votes');
```

tinyText()

Phương thức `tinyText` tạo một cột tương đương `TINYTEXT`:

```
$table->tinyText('notes');
```

unsignedBigInteger()

Phương thức `unsignedBigInteger` tạo một cột tương đương `UNSIGNED BIGINT`:

```php
$table->unsignedBigInteger('votes');
```

unsignedDecimal()

Phương thức `unsignedDecimal` tạo cột tương đương `UNSIGNED DECIMAL` với độ chính xác tùy chọn (tổng số) và tỷ lệ (chữ số thập phân):

```php
$table->unsignedDecimal('amount', $precision = 8, $scale = 2);
```

unsignedInteger()

Phương thức `unsignedInteger` tạo một cột tương đương `UNSIGNED INTEGER`:

```php
$table->unsignedInteger('votes');
```

unsignedMediumInteger()

Phương thức `unsignedMediumInteger` tạo một cột tương đương `UNSIGNED MEDIUMINT`:

```php
$table->unsignedMediumInteger('votes');
```

unsignedSmallInteger()

Phương thức `unsignedSmallInteger` tạo một cột tương đương `UNSIGNED SMALLINT`:

```php
$table->unsignedSmallInteger('votes');
```

unsignedTinyInteger()

Phương thức `unsignedTinyInteger` tạo một cột tương đương `UNSIGNED TINYINT`:

```php
$table->unsignedTinyInteger('votes');
```

uuidMorphs()

Phương thức uuidMorphs là một phương thức tiện lợi để thêm một cột tương đương `{column}_id CHAR(36)` và một cột tương đương `{column}_type VARCHAR`.

Phương thức này nhằm mục đích được sử dụng khi xác định các cột cần thiết cho mối quan hệ Eloquent đa hình sử dụng các định danh `UUID`. Trong ví dụ sau, các cột `taggable_id` và `taggable_type` sẽ được tạo:

```
$table->uuidMorphs('taggable');
```

`ulid()`

Phương thức `ulid` tạo một cột tương đương `ULID`:

```
$table->ulid('id');
```

`uuid()`

Phương thức `uuid` tạo một cột tương đương `UUID`:

```
$table->uuid('id');
```

`year()`

Phương thức `year` tạo một cột tương đương `YEAR`:

```
$table->year('birth_year');
```

Các thao tác dành cho cột dữ liệu

Ngoài các loại cột được liệt kê ở trên, bạn có thể sử dụng một số thao tác cột sau khi thêm cột vào bảng cơ sở dữ liệu. Ví dụ: để làm cho cột là "nullable", bạn có thể sử dụng phương thức `nullable`:

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

```php
Schema::table('users', function (Blueprint $table) {
  $table->string('email')->nullable();
});
```

Bảng sau đây chứa tất cả các thao tác sửa đổi cột. Danh sách này không bao gồm các thao tác sửa đổi chỉ mục:

Thao tác	Mô tả
`->after('column')`	Đặt cột "sau" một cột khác (MySQL).
`->autoIncrement()`	Đặt các cột INTEGER làm tự động tăng (khóa chính).
`->charset('utf8mb4')`	Chỉ định một bộ ký tự cho cột (MySQL).
`->collation('utf8mb4_unicode_ci')`	Chỉ định đối chiếu cho cột (MySQL / PostgreSQL / SQL Server).
`->comment('my comment')`	Thêm ghi cú vào một cột (MySQL / PostgreSQL).
`->default($value)`	Chỉ định giá trị "mặc định" cho cột.
`->first()`	Đặt cột "đầu tiên" trong bảng (MySQL).
`->from($integer)`	Đặt giá trị bắt đầu của trường tự động tăng (MySQL / PostgreSQL).
`->invisible()`	Làm cho cột "ẩn" với các truy vấn `SELECT *` (MySQL).
`->nullable($value = true)`	Cho phép chèn giá trị `NULL` vào cột.
`->storedAs($expression)`	Tạo một cột được tạo được lưu trữ (MySQL / PostgreSQL).
`->unsigned()`	Đặt các cột `INTEGER` là `UNSIGNED` (MySQL).
`->useCurrent()`	Đặt cột `TIMESTAMP` để sử dụng `CURRENT_TIMESTAMP` làm giá trị mặc định.
`->useCurrentOnUpdate()`	Đặt cột `TIMESTAMP` để sử dụng `CURRENT_TIMESTAMP` khi bản ghi được cập nhật.

Thao tác	Mô tả
`->virtualAs($expression)`	Tạo một cột được tạo ảo (MySQL).
`->generatedAs($expression)`	Tạo một cột nhận dạng với các tùy chọn trình tự được chỉ định (PostgreSQL).
`->always()`	Xác định mức độ ưu tiên của các giá trị trình tự so với đầu vào cho một cột nhận dạng (PostgreSQL).
`->isGeometry()`	Đặt kiểu cột không gian thành `geometry` - kiểu mặc định là `geography` (PostgreSQL).

Biểu thức mặc định

Công cụ sửa đổi default chấp nhận một giá trị hoặc một đối tượng `Illuminate\Database\Query\Expression`. Sử dụng một đối tượng Expression sẽ ngăn Laravel gói giá trị trong dấu ngoặc kép và cho phép bạn sử dụng các chức năng cụ thể của cơ sở dữ liệu. Một tình huống mà điều này đặc biệt hữu ích là khi bạn cần gán giá trị mặc định cho các cột JSON:

```php
<?php

use Illuminate\Support\Facades\Schema;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Database\Query\Expression;
use Illuminate\Database\Migrations\Migration;

return new class extends Migration
{
  /**
   * Run the migrations.
   *
   * @return void
   */
  public function up()
  {
    Schema::create('flights', function (Blueprint $table) {
```

```
            $table->id();
            $table->json('movies')->default(new Expression('(JSON_ARRAY())'));
            $table->timestamps();
        });
    }
};
```

> **Chú ý:** Hỗ trợ cho các biểu thức mặc định phụ thuộc vào driver cơ sở dữ liệu, đối tượng cơ sở dữ liệu và loại trường của bạn. Vui lòng tham khảo tài liệu cơ sở dữ liệu của bạn. Ngoài ra, không thể kết hợp các biểu thức mặc định thô (sử dụng `DB::raw`) với các thay đổi cột thông qua phương thức thay đổi.

Thứ tự của cột dữ liệu

Khi sử dụng cơ sở dữ liệu MySQL, phương thức `after` có thể được sử dụng để thêm các cột vào sau một cột hiện có trong lược đồ:

```
$table->after('password', function ($table) {
    $table->string('address_line1');
    $table->string('address_line2');
    $table->string('city');
});
```

Điều chỉnh các cột

Các điều kiện tiên quyết

Trước khi sửa đổi một cột, bạn phải cài đặt package `doctrine/dbal` bằng trình quản lý package *composer*. Thư viện Doctrine DBAL được sử dụng để xác định trạng thái hiện tại của cột và tạo các truy vấn SQL cần thiết để thực hiện các thay đổi được yêu cầu đối với cột của bạn:

```
composer require doctrine/dbal
```

Nếu bạn định sửa đổi các cột được tạo bằng phương thức `timestamp`, bạn cũng phải thêm cấu hình sau vào tập tin cấu hình *config/database.php* của ứng dụng:

```
use Illuminate\Database\DBAL\TimestampType;

'dbal' => [
  'types' => [
    'timestamp' => TimestampType::class,
  ],
],
```

> **Chú ý:** Nếu ứng dụng của bạn đang sử dụng Microsoft SQL Server, hãy đảm bảo rằng bạn cài đặt `doctrine/dbal: ^ 3.0`.

Cập nhật các thuộc tính của cột

Phương thức change cho phép bạn sửa đổi loại và thuộc tính của các cột hiện có. Ví dụ, bạn có thể muốn tăng kích thước của một cột chuỗi. Để nhìn thấy phương thức change hoạt động, hãy giả dụ tăng kích thước của cột tên từ 25 lên 50. Để thực hiện điều này, chúng ta chỉ cần xác định trạng thái mới của cột và sau đó gọi phương thức change:

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

Chúng ta cũng có thể sửa đổi một cột thành có giá trị có thể null:

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

> **Chú ý:** Các kiểu cột sau có thể chỉnh sửa được: `bigInteger`, `binary`, `boolean`, `char`, `date`, `dateTime`, `dateTimeTz`, `decimal`, `double`, `integer`, `json`, `longText`, `mediumText`, `smallInteger`, `string`, `text`, `time`, `tinyText`, `unsignedBigInteger`, `unsignedInteger`, `unsignedSmallInteger`, và `uuid`. Để chỉnh sửa một kiểu cột `timestamp`.

Các cột thay đổi tên

Để đổi tên một cột, bạn có thể sử dụng phương thức `renameColumn` được cung cấp bởi kế hoạch chi tiết (blueprint) của schema builder. Trước khi đổi tên một cột, hãy đảm bảo rằng bạn đã cài đặt package `doctrine/dbal` thông qua trình quản lý gói Composer:

```
Schema::table('users', function (Blueprint $table) {
  $table->renameColumn('from', 'to');
});
```

> **Chú ý:** Đổi tên cột `enum` hiện không được hỗ trợ.

Gở bỏ cột dữ liệu

Để bỏ một cột, bạn có thể sử dụng phương thức `dropColumn` trên bản thiết kế của schema builder. Nếu ứng dụng của bạn đang sử dụng cơ sở dữ liệu SQLite, bạn phải cài đặt gói `doctrine/dbal` thông qua trình quản lý package *Composer* trước khi phương thức `dropColumn` có thể được sử dụng:

```
Schema::table('users', function (Blueprint $table) {
  $table->dropColumn('votes');
});
```

Bạn có thể bỏ nhiều cột từ một bảng bằng cách chuyển một mảng tên cột vào phương thức `dropColumn`:

```
Schema::table('users', function (Blueprint $table) {
  $table->dropColumn(['votes', 'avatar', 'location']);
});
```

> **Chú ý:** Loại bỏ hoặc sửa đổi nhiều cột trong một lần di chuyển trong khi sử dụng cơ sở dữ liệu SQLite không được hỗ trợ.

Các bí danh lệnh

Laravel cung cấp một số phương thức thông dụng để loại bỏ các loại cột phổ biến. Các phương thức này được mô tả trong bảng dưới đây:

Lệnh	Mô tả
`$table->dropMorphs('morphable');`	Gở bỏ các cột `morphable_id` và `morphable_type`.
`$table->dropRememberToken();`	Gở bỏ cột `remember_token`.
`$table->dropSoftDeletes();`	Gở bỏ cột `deleted_at`.
`$table->dropSoftDeletesTz();`	Bí danh của phương thức `dropSoftDeletes()`.
`$table->dropTimestamps();`	Gở bỏ các cột `created_at` và `updated_at`.
`$table->dropTimestampsTz();`	Bí danh của phương thức `dropTimestamps()`.

Chỉ số

Tạo chỉ số

Schema builder của Laravel hỗ trợ một số loại chỉ mục. Ví dụ sau tạo một cột email mới và chỉ định rằng các giá trị của nó phải là unique (duy nhất). Để tạo chỉ mục này, chúng ta có thể xâu chuỗi phương thức `unique` vào định nghĩa cột:

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

Schema::table('users', function (Blueprint $table) {
  $table->string('email')->unique();
});
```

Ngoài ra, bạn có thể tạo chỉ mục sau khi khai báo cột. Để làm như vậy, bạn nên gọi phương thức `unique` trên bản thiết kế của schema builder. Phương thức này chấp nhận tên của cột sẽ nhận được một chỉ mục *unique*:

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

Bạn thậm chí có thể truyền một mảng cột tới một phương thức chỉ mục nào đó để tạo một

chỉ mục phức hợp hơn (hoặc có tính kết hợp):

```
$table->index(['account_id', 'created_at']);
```

Khi tạo chỉ mục, Laravel sẽ tự động tạo tên chỉ mục dựa trên bảng, tên cột và kiểu chỉ mục, nhưng bạn có thể truyền đối số thứ hai vào phương thức để tự chỉ định tên chỉ mục bạn muốn:

```
$table->unique('email', 'unique_email');
```

Các kiểu index

Class blueprint của schema builder trong Laravel cung cấp các phương thức để tạo từng loại chỉ mục được Laravel hỗ trợ. Mỗi phương thức chỉ mục sẽ chấp nhận một đối số thứ hai không bắt buộc để chỉ định tên của chỉ mục. Nếu bỏ qua, tên sẽ được lấy từ tên của bảng và (các) cột được sử dụng cho chỉ mục, cũng như kiểu chỉ mục. Mỗi phương thức chỉ mục có sẵn được mô tả trong bảng dưới đây:

Lệnh	Mô tả
`$table->primary('id');`	Thêm một khóa chính.
`$table->primary(['id', 'parent_id']);`	Thêm các khóa tổng hợp.
`$table->unique('email');`	Thêm một chỉ mục duy nhất.
`$table->index('state');`	Thêm chỉ mục.
`$table->fullText('body');`	Thêm chỉ mục văn bản đầy đủ (MySQL / PostgreSQL).
`$table->fullText('body')->language('english');`	Thêm chỉ mục văn bản đầy đủ của ngôn ngữ được chỉ định (PostgreSQL).
`$table->spatialIndex('location');`	Thêm chỉ mục không gian (ngoại trừ SQLite).

Index độ dài & MySQL / MariaDB

Theo mặc định, Laravel sử dụng bộ ký tự `utf8mb4`. Nếu bạn đang chạy phiên bản MySQL cũ hơn bản phát hành 5.7.7 hoặc MariaDB cũ hơn bản phát hành 10.2.2, bạn có thể cần phải cấu hình thủ công độ dài chuỗi mặc định được tạo bởi migration để MySQL tạo chỉ mục cho chúng. Bạn có thể cấu hình độ dài chuỗi mặc định bằng cách gọi phương thức `Schema::defaultStringLength` trong phương thức boot của class `App\Providers\AppServiceProvider`:

```
use Illuminate\Support\Facades\Schema;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  Schema::defaultStringLength(191);
}
```

Ngoài ra, bạn có thể bật tùy chọn `innodb_large_prefix` cho cơ sở dữ liệu của mình. Tham khảo tài liệu của cơ sở dữ liệu của bạn để biết hướng dẫn về cách bật tùy chọn này đúng cách.

Thay tên cho các chỉ mục

Để đổi tên một chỉ mục, bạn có thể sử dụng phương thức `renameIndex` được cung cấp bởi kế hoạch chi tiết của schema builder. Phương thức này chấp nhận tên chỉ mục hiện tại làm đối số đầu tiên và tên mong muốn làm đối số thứ hai:

```
$table->renameIndex('from', 'to')
```

Gở bỏ các chỉ số

Để gở bỏ một chỉ mục, bạn phải chỉ định tên của chỉ mục. Theo mặc định, Laravel tự động gán tên chỉ mục dựa trên tên bảng, tên của cột được lập chỉ mục và kiểu chỉ mục. Dưới đây là một số ví dụ:

Lệnh	Mô tả
`$table->dropPrimary('users_id_primary');`	Bỏ khóa chính khỏi bảng "users".
`$table->dropUnique('users_email_unique');`	Bỏ một chỉ mục unique khỏi bảng "users".
`$table->dropIndex('geo_state_index');`	Bỏ chỉ mục cơ bản khỏi bảng "geo".
`$table->dropFullText('posts_body_fulltext');`	Thả một chỉ mục văn bản đầy đủ từ bảng "post".
`$table->dropSpatialIndex('geo_location_spatialindex');`	Thả một chỉ mục không gian từ bảng "geo" (ngoại trừ SQLite).

Nếu bạn truyền một mảng cột vào một phương thức để gở bỏ chỉ mục, thì tên chỉ mục thông thường sẽ được tạo dựa trên tên bảng, cột và loại chỉ mục:

```
Schema::table('geo', function (Blueprint $table) {
  $table->dropIndex(['state']); // Drops index 'geo_state_index'
});
```

Các ràng buộc về khóa ngoại (foreign key)

Laravel cũng cung cấp hỗ trợ để tạo các ràng buộc khóa ngoại, được sử dụng để buộc tính toàn vẹn tham chiếu ở cấp cơ sở dữ liệu. Ví dụ: hãy quy định cột `user_id` trên bảng `posts` sẽ tham chiếu đến cột `id` trên bảng `users`:

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

Schema::table('posts', function (Blueprint $table) {
  $table->unsignedBigInteger('user_id');

  $table->foreign('user_id')->references('id')->on('users');
});
```

Vì cú pháp này khá dài dòng, Laravel cung cấp các phương thức bổ sung, ngắn gọn để sử dụng các quy ước nhằm cung cấp trải nghiệm tốt hơn cho nhà phát triển. Khi sử dụng phương thức `foreignId` để tạo cột của bạn, ví dụ trên có thể được viết lại như sau:

```
Schema::table('posts', function (Blueprint $table) {
  $table->foreignId('user_id')->constrained();
});
```

Phương thức `foreignId` tạo ra một cột tương đương `UNSIGNED BIGINT`, trong khi phương thức `constrained` sẽ sử dụng các quy ước để xác định bảng và tên cột đang được tham chiếu. Nếu tên bảng của bạn không khớp với các quy ước của Laravel, bạn có thể chỉ định tên bảng bằng cách truyền nó làm đối số cho phương thức `constrained`:

```
Schema::table('posts', function (Blueprint $table) {
  $table->foreignId('user_id')->constrained('users');
});
```

Bạn cũng có thể chỉ định hành động cụ thể cho các thuộc tính "on delete" và "on update" của ràng buộc:

```
$table->foreignId('user_id')
      ->constrained()
      ->onUpdate('cascade')
      ->onDelete('cascade');
```

Một cú pháp mang tính diễn đạt được cung cấp cho các hành động sau:

Phương thức	Mô tả
`$table->cascadeOnUpdate();`	Cập nhật sẽ cascade.
`$table->restrictOnUpdate();`	Cập nhật sẽ bị thắt chặc.
`$table->cascadeOnDelete();`	Khi xóa sẽ cascade.
`$table->restrictOnDelete();`	Việc xóa sẽ bị thắt chặt.

Phương thức	Mô tả
`$table->nullOnDelete();`	Việc xóa sẽ đặt giá trị khóa ngoại thành null.

Bất kỳ công cụ sửa đổi cột bổ sung nào phải được gọi trước phương thức `constrained`:

```
$table->foreignId('user_id')
      ->nullable()
      ->constrained();
```

Gở bỏ khóa ngoại

Để xóa khóa ngoại, bạn có thể sử dụng phương thức `dropForeign`, truyền tên của ràng buộc khóa ngoại sẽ bị xóa làm đối số. Các ràng buộc khóa ngoại sử dụng quy ước đặt tên giống như các chỉ mục. Nói cách khác, tên ràng buộc khóa ngoại dựa trên tên của bảng và các cột trong ràng buộc, theo sau là hậu tố "*_foreign*":

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

Ngoài ra, bạn có thể truyền một mảng chứa tên cột chứa khóa ngoại vào phương thức `dropForeign`. Mảng sẽ được chuyển đổi thành tên ràng buộc khóa ngoại bằng cách sử dụng các quy ước đặt tên cho ràng buộc của Laravel:

```
$table->dropForeign(['user_id']);
```

Chuyển đổi các ràng buộc khóa ngoại

Bạn có thể bật hoặc tắt các ràng buộc khóa ngoại trong migration của mình bằng cách sử dụng các phương thức sau:

```
Schema::enableForeignKeyConstraints();

Schema::disableForeignKeyConstraints();
```

Chú ý: SQLite vô hiệu hóa các ràng buộc khóa ngoại theo mặc định. Khi sử dụng SQLite, hãy đảm bảo bật hỗ trợ khóa ngoại trong cấu hình cơ sở dữ liệu của bạn trước khi cố gắng tạo chúng trong migration của bạn. Ngoài ra, SQLite chỉ hỗ trợ khóa ngoại khi tạo bảng và không hỗ trợ khi bảng bị thay đổi.

Các event

Để thuận tiện, mỗi thao tác migrate sẽ gửi một event. Tất cả các event sau đây sẽ được mở rộng từ class `Illuminate\Database \Events\MigrationEvent`:

Class	Mô tả
`Illuminate\Database\Events\MigrationsStarted`	Một migration sắp được thực thi.
`Illuminate\Database\Events\MigrationsEnded`	Một migration đã hoàn tất quá trình thực thi.
`Illuminate\Database\Events\MigrationStarted`	Một tập tin migration sắp được thực thi.
`Illuminate\Database\Events\MigrationEnded`	Một tập tin migration đã hoàn tất quá trình thực thi.
`Illuminate\Database\Events\SchemaDumped`	Việc kết xuất database schema đã được trình thực thi.
`Illuminate\Database\Events\SchemaLoaded`	Việc kết xuất database schema đang có đã được tải.