



Bài 17

ORM và Eloquent

Module: BOOTCAMP WEB-BACKEND DEVELOPMENT

- Trình bày được cấu hình CSDL
- Trình bày được Migration
- Trình bày được Seeding
- Thực thi được các câu lệnh SQL
- Trình bày được khái niệm ORM
- Trình bày được Eloquent
- Thực hiện được các thao tác CRUD cơ bản

Cơ bản về CSDL

Giới thiệu Cấu hình

Đọc & ghi các kết nối

Sử dụng nhiều kết nối cơ sở dữ liệu

Thực thi SQL thuần

Listening tới các Query Events

Database Transactions

- Laravel giúp cho việc kết nối tới các CSDL và thực thi các query cực kì đơn giản
- Có một số cơ chế khác nhau để làm việc với CSDL:
 - Raw SQL: Thực thi câu lệnh SQL thuần
 - Query builder: Giúp xây dựng các câu lệnh SQL
 - Eloquent ORM: Tự động ánh xạ và liên kết dữ liệu với CSDL
- Laravel hỗ trợ sẵn bốn CSDL:
 - MySQL
 - Postgres
 - SQLite
 - SQL Server

- Cấu hình CSDL trong file `config/database.php`
- Cấu hình các connection
- Xác định connection sẽ được sử dụng mặc định
- Có thể tùy chỉnh file `.env` để thay đổi cấu hình CSDL

Cấu hình SQL Server



- Laravel hỗ trợ sẵn cho SQL Server
- Tuy nhiên bạn vẫn cần thêm vào thông số cấu hình cho cơ sở dữ liệu tại file cấu hình
- `config/database.php`

```
'sqlsrv' => [  
    'driver' => 'sqlsrv',  
    'host' => env('DB_HOST', 'localhost'),  
    'database' => env('DB_DATABASE', 'forge'),  
    'username' => env('DB_USERNAME', 'forge'),  
    'password' => env('DB_PASSWORD', ''),  
    'charset' => 'utf8',  
    'prefix' => '',  
],
```

Đọc & ghi các kết nối



```
'mysql' => [  
  'read' => [  
    'host' => '192.168.1.1',  
  ],  
  'write' => [  
    'host' => '196.168.1.2'  
  ],  
  'driver'    => 'mysql',  
  'database'  => 'database',  
  'username'  => 'root',  
  'password'  => '',  
  'charset'   => 'utf8',  
  'collation' => 'utf8_unicode_ci',  
  'prefix'    => '',  
],
```

Sử dụng nhiều kết nối cơ sở dữ liệu



- Có thể truy cập vào mỗi kết nối qua phương thức **connection** trong **DB** façade
- **name** truyền vào trong phương thức connection cần tương ứng với tên của kết nối trong file cấu hình config/database.php

```
$users = DB::connection('foo')->select(...);
```

```
$pdo = DB::connection()->getPdo();
```


Thực thi lệnh SQL thuần



- Sử dụng **DB façade** để thực thi lệnh SQL thuần
- DB facade cung cấp các hàm để thực hiện các kiểu query: **select, update, insert, delete, và statement.**

Thực thi lệnh select

- Sử dụng hàm **select** trong **DB** facade để thực thi một câu lệnh truy vấn

```
foreach ($users as $user) {  
    echo $user->name;  
}
```

```
<?php  
  
namespace App\Http\Controllers;  
  
use Illuminate\Support\Facades\DB;  
use App\Http\Controllers\Controller;  
  
class UserController extends Controller  
{  
    /**  
     * Show a list of all of the application's users.  
     *  
     * @return Response  
     */  
    public function index()  
    {  
        $users = DB::select('select * from users where active = ?'  
  
        return view('user.index', ['users' => $users]);  
    }  
}
```



Sử dụng Named Bindings

- Thay vì sử dụng ? để tượng trưng cho liên kết parameter, có thể thực thi câu query sử dụng liên kết đặt tên:

```
$results = DB::select('select * from users where id = :id', ['id' => 1]);
```

Thực thi câu lệnh insert



- Sử dụng hàm **insert** trong DB facade
- Giống như **select**, hàm **insert** nhận câu raw SQL query ở tham số đầu tiên, và bindings ở tham số thứ hai

```
DB::insert('insert into users (id, name) values (?, ?)', [1, 'Dayle']);
```

Thực thi câu lệnh update



- Hàm **update** được dùng để **update** các bản ghi đang có trong CSDL
- Hàm **update** trả về lượng các dòng được cập nhật

```
$affected = DB::update('update users set votes = 100 where name = ?', ['John']);
```

Thực thi câu lệnh delete



- Hàm **delete** cần được sử dụng để xoá các records khỏi cơ sở dữ liệu.
- Giống như **update**, số lượng dòng bị xoá sẽ được trả về:

```
$deleted = DB::delete('delete from users');
```

Thực thi một câu lệnh chung

- Sử dụng hàm statement trong DB façade để thực thi các câu lệnh SQL nói chung

```
DB::statement('drop table users');
```

Database Transactions



- Sử dụng phương thức **transaction** trong DB facade
- Khi có một **exception** được ném ra trong transaction **Closure**, transaction sẽ tự động được roll back.
- Khi **Closure** thực thi thành công, transaction sẽ tự động được commit.
- Không cần phải lo lắng về việc thực hiện thủ công các thao tác **roll back** hay **commit** khi sử dụng hàm **transaction**

Database Transactions



```
DB::transaction(function () {  
    DB::table('users')->update(['votes' => 1]);  
  
    DB::table('posts')->delete();  
});
```

Tự quản lý transaction

- Sử dụng các phương thức `beginTransaction()`, `rollback()` và `commit()` của DB façade để tự quản lý transaction
- Bắt đầu một transaction:

- Roll-back nếu có lỗi:

```
DB::beginTransaction();
```

- Commit nếu thành công:

```
DB::rollback();
```

```
DB::commit();
```

Migrations

Làm việc với Migrations

Rolling Back Migrations

Tạo và thay đổi cấu trúc của Bảng

- Migration được coi như là version control cho database
- Dễ dàng xây dựng cấu trúc cho CSDL
- Có thể dễ dàng thay đổi và chia sẻ cấu trúc (schema) của CSDL
- Schema facade hỗ trợ việc tạo và thao tác trên các bảng mà không cần làm việc trực tiếp với CSDL

Tạo Migrations



- Sử dụng câu lệnh **make:migration**

```
php artisan make:migration create_users_table
```

```
php artisan make:migration create_users_table --create=users
```

```
php artisan make:migration add_votes_to_users_table --table=users
```

Lớp migration



- Một lớp migration chứa hai hàm cơ bản là up() và down()
 - Hàm up() được dùng để tạo table, cột hay index mới trong CSDL
 - Hàm down thực hiện ngược lại những thao tác ở hàm up()

```
class CreateFlightsTable extends Migration
{

    public function up()
    {
        Schema::create('flights', function (Blueprint $table) {
            $table->increments('id');
            $table->string('name');
            $table->string('airline');
            $table->timestamps();
        });
    }

    public function down()
    {
        Schema::drop('flights');
    }

}
```

Thực thi Migration



- Để thực thi tất cả các migration trong chương trình, sử dụng lệnh **migrate**

```
php artisan migrate
```

- Ép buộc thực thi migration chạy cho môi trường production

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

Rolling Back Migrations



- Sử dụng câu lệnh rollback để quay lại trước câu lệnh migrate cuối cùng

```
php artisan migrate:rollback
```

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

```
php artisan migrate:reset
```


Rollback & Migrate trong cùng một câu lệnh



- Lệnh migrate:refresh sẽ rollback lại toàn bộ migration của chương trình, và thực hiện câu lệnh migrate
- Câu lệnh sẽ thực hiện tái cấu trúc toàn bộ database:

```
php artisan migrate:refresh
```

```
// Refresh the database and run all database seeds...
```

```
php artisan migrate:refresh --seed
```

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



- Để tạo một bảng mới, sử dụng hàm `create` trong `Schema` facade.
- Kiểm tra sự tồn tại của một bảng hay cột sử dụng hàm `hasTable` and `hasColumn`

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

```
if (Schema::hasTable('users')) {  
    //  
}  
  
if (Schema::hasColumn('users', 'email')) {  
    //  
}
```

Đổi tên / xóa bảng



- Sử dụng hàm rename để đổi tên bảng

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

- Sử dụng hàm drop hoặc dropIfExists để xóa bảng

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

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

- Hàm table trong Schema facade sử dụng để cập nhật một bảng đã tồn tại
- Như hàm create, hàm table nhận hai tham số: tên của bảng và một Closure nhận một đối tượng Blueprint để thực hiện thao tác với bảng

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

Chỉnh sửa cột



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

Modifier	Description
<code>->after('column')</code>	Đặt column "after" một column khác (MySQL Only)
<code>->comment('my comment')</code>	Thêm một comment cho column.
<code>->default(\$value)</code>	Đặt giá trị "mặc định" vào column
<code>->first()</code>	Đặt column "first" vào trong bảng (MySQL Only)
<code>->nullable()</code>	Cho phép dữ liệu kiểu NULL có thể chèn vào column.
<code>->storedAs(\$expression)</code>	Tạo một cột stored (MySQL Only)
<code>->unsigned()</code>	Đặt cột <code>integer</code> sang <code>UNSIGNED</code>
<code>->virtualAs(\$expression)</code>	Tạo một cột virtual (MySQL Only)

- Cần khai báo dependency là doctrine/dbal vào trong file composer.json
- Thư viện Doctrine DBAL được dùng để xác định trạng thái hiện tại của column và tạo câu SQL query cần thiết để chỉnh sửa column:

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

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

- Sử dụng hàm renameColumn trong Schema builder
- Cần thêm doctrine/dbal dependency vào file composer.json

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



Hiện tại chưa hỗ trợ đổi tên cột có kiểu là `enum`.

- Sử dụng hàm dropColumn trong the Schema builder

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

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



Drop hay thay đổi nhiều columns trong một file migration trong khi sử dụng SQLite chưa được hỗ trợ.

Database: Seeding

Giới thiệu

Viết Seeders

Sử dụng Model Factories

Gọi các Seeders bổ sung

Thực thi Seeders



- Seeding là cơ để chèn dữ liệu mẫu vào trong CSDL của ứng dụng
- Seeding được thực hiện thông qua các lớp Seeder
- Các lớp Seeder được lưu trong thư mục `database/seeds`
- Nên đặt tên các lớp Seeder theo một quy ước để dễ nhận biết

Tạo các lớp Seeder



- Sử dụng lệnh `make:seeder`
- Các seeder được sinh ra bởi framework sẽ được đặt trong thư mục `database/seeds`:

```
php artisan make:seeder UsersTableSeeder
```

Tạo các lớp Seeder



```
<?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('users')->insert([
            'name' => str_random(10),
            'email' => str_random(10).'@gmail.com',
            'password' => bcrypt('secret'),
        ]);
    }
}
```

Sử dụng Model Factory



- Model factory giúp sinh ra lượng lớn các dữ liệu vào trong CSDL

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    factory(App\User::class, 50)->create()->each(function($u) {
        $u->posts()->save(factory(App\Post::class)->make());
    });
}
```

Gọi các Seeder



- Bên trong class DatabaseSeeder, dùng hàm call để gọi các lớp Seeder
- Giúp tách các Seeder thành nhiều file

```
/**
 * Run the database seeds.
 *
 * @return void
 */
public function run()
{
    $this->call(UsersTableSeeder::class);
    $this->call(PostsTableSeeder::class);
    $this->call(CommentsTableSeeder::class);
}
```

Chạy seeding

- Sử dụng câu lệnh db:seed
- Sử dụng tùy chọn --class để chỉ định chạy một Seeder nhất định
- Có thể chạy seeding cùng với migrate bằng cách sử dụng tham số --seed

```
php artisan db:seed
```

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

```
php artisan migrate:refresh --seed
```

Eloquent: Bắt đầu

Giới thiệu Định nghĩa Models

Lấy nhiều Models, Lấy một Models / Aggregates

Thêm & Cập nhật Models, Deleting Models

Query Scopes, Events

- Eloquent ORM đi kèm với Laravel cung cấp một API ActiveRecord đơn giản và tuyệt vời khi làm việc với database. Mỗi database table sẽ có một "Model" tương ứng để tương tác với table đó. Model cho phép bạn query dữ liệu trong table, cũng như chèn thêm các dữ liệu mới.
- Trước khi bắt đầu, hãy đảm bảo cấu hình kết nối database trong file `config/database.php`. Để biết thêm thông tin chi tiết cho cấu hình database, hãy xem the documentation.

Định nghĩa Models



- Để bắt đầu, hãy cùng tạo một Eloquent model. Model về cơ bản nằm trong thư mục app nhưng bạn có thể tùy ý đặt chúng ở bất cứ đâu mà được cấu hình autoload trong file composer.json . Tất cả các Eloquent model đều kế thừa từ class Illuminate\Database\Eloquent\Model .
- Cách đơn giản nhất để tạo một model là sử dụng make:model Artisan command:

```
php artisan make:model User
```

Quy tắc cho Eloquent Model



- Bây giờ, hãy cùng nhau coi ví dụ về class model Flight, mà chúng ta sẽ dùng để lấy và lưu thông tin vào trong bảng flights:

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    //
}
```

- Để ý là chúng ta không hề cho Eloquent biết là bảng nào được sử dụng cho model Flight. Vì kiểu "snake case", tên class ở số nhiều sẽ được sử dụng như tên table trừ khi có một tên khác được khai báo. Vì thế, trong trường hợp này, Eloquent sẽ coi model Flight lưu dữ liệu vào trong bảng flights. Bạn có thể chỉ định tên table khác cho model bằng cách khai báo thuộc tính table trong model:

Tên bảng



```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The table associated with the model.
     *
     * @var string
     */
    protected $table = 'my_flights';
}
```

Primary Keys



- Eloquent cũng coi mỗi table có một column là primary key tên là id . Bạn có thể định nghĩa một `$primaryKey` để đổi tên column này.
- Ngoài ra, Eloquent cũng coi primary key là một giá trị nguyên tăng dần, có nghĩa là về mặc định primary key sẽ được cast về kiểu int tự động. Nếu bạn muốn sử dụng primary không tăng dần hay không phải là dạng số, bạn cần thay đổi thuộc tính `$incrementing` trong model thành false .

Timestamps



- Mặc định, Eloquent cần hai cột `created_at` và `updated_at` có mặt trong các bảng. Nếu bạn không muốn những columns này tự động được quản lý bởi Eloquent, thiết lập thuộc tính `$timestamps` trong model thành `false` :

```
<?php

namespace App;

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



```
class Flight extends Model
{
    /**
     * Indicates if the model should be timestamped.
     *
     * @var bool
     */
    public $timestamps = false;
}
```

Timestamps




- Nếu bạn muốn thay đổi định dạng của timestamp, thiết lập vào thuộc tính \$dateFormat trong model. Thuộc tính này xác định cách mà các thuộc tính kiểu date được lưu trong database cũng như cách format khi được serialize thành array hay JSON:

```
<?php

namespace App;

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



```
class Flight extends Model
{
    /**
     * The storage format of the model's date columns.
     *
     * @var string
     */
    protected $dateFormat = 'U';
}
```


Kết nối database



- Tất cả các Eloquent model sẽ sử dụng kết nối database mặc định được cấu hình. Nếu bạn muốn sử dụng một kết nối khác cho model, sử dụng thuộc tính \$connection:

```
<?php

namespace App;

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



```
class Flight extends Model
{
    /**
     * The connection name for the model.
     *
     * @var string
     */
    protected $connection = 'connection-name';
}
```

Lấy nhiều Models



- Khi bạn đã tạo được một model và its associated database table, bạn có thể sẵn sàng truy xuất dữ liệu từ database. Hãy coi mỗi Eloquent model như một query builder mạnh mẽ cho phép bạn thực hiện query tới database một cách liền mạch. Ví dụ:

```
<?php

use App\Flight;

$flights = App\Flight::all();

foreach ($flights as $flight) {
    echo $flight->name;
}
```

Thêm ràng buộc bổ sung



- Phương thức all sẽ trả về tất cả các kết quả trong table của model. Vì mỗi Eloquent model phục vụ như một query builder, nên bạn có thể tạo ràng buộc cho các query, và cuối cùng sử dụng hàm get để lấy kết quả:

```
$flights = App\Flight::where('active', 1)
    ->orderBy('name', 'desc')
    ->take(10)
    ->get();
```



Vì các Eloquent model là các query builder, bạn nên xem qua tất cả các hàm có thể sử dụng trên [query builder](#). Bạn có thể áp dụng bất kì hàm nào trong này với Eloquent query.

- Vì các hàm của Eloquent như all và get đều trả về nhiều kết quả, một instance từ Illuminate\Database\Eloquent\Collection sẽ được trả về. Class Collection cung cấp các hàm hữu ích cho phép làm việc với các kết quả Eloquent:

```
$flights = $flights->reject(function ($flight) {  
    return $flight->cancelled;  
});
```

- Tất nhiên, bạn có thể foreach vòng lặp collection như một array:

```
foreach ($flights as $flight) {  
    echo $flight->name;  
}
```

Chunking Results



- Nếu bạn muốn xử lý hàng ngàn kết quả từ Eloquent, sử dụng hàm chunk. Hàm chunk này sẽ lấy từng "chunk" của Eloquent models, cung cấp chúng thông qua Closure để xử lý. Sử dụng hàm chunk sẽ tiết kiệm được memory khi thao tác với tập dữ liệu kết quả lớn:

```
Flight::chunk(200, function ($flights) {  
    foreach ($flights as $flight) {  
        //  
    }  
});
```

Chunking Results



- Tham số đầu truyền vào là số record bạn muốn lấy từng "chunk". Closure truyền vào ở tham số thứ hai sẽ được gọi cho mỗi chunk được lấy từ database. Một database sẽ được thực thi để lấy mỗi "chunk" của records truyền vào Closure.

Sử dụng Cursors



- Hàm cursor cho phép bạn duyệt qua records bằng cách sử dụng một cursor, nó chỉ thực thi cho một truy vấn. Khi dữ liệu lớn, hàm cursor có thể được sử dụng để giảm memory sử dụng:

```
foreach (Flight::where('foo', 'bar')->cursor() as $flight) {  
    //  
}
```

Lấy một Models / Aggregates



- Ngoài việc lấy tất cả dữ liệu, bạn có thể lấy một kết quả sử dụng hàm `find` hoặc `first`. Thay vì trả về một collection model, những hàm này trả về một model instance:

```
// Retrieve a model by its primary key...
```

```
$flight = App\Flight::find(1);
```

```
// Retrieve the first model matching the query constraints...
```

```
$flight = App\Flight::where('active', 1)->first();
```


Lấy một Models / Aggregates



- Bạn có thể gọi hàm find với một mảng các primary key, với kết quả trả về là một collection các kết quả tìm thấy:

```
$flights = App\Flight::find([1, 2, 3]);
```

Not Found Exceptions

- Sẽ có lúc bạn muốn bắn ra một exception nếu một model không được tìm thấy. Điều này thực sự hữu ích khi làm việc trên route hay controller. Hàm `findOrFail` và `firstOrFail` sẽ trả lại kết quả đầu tiên của query. Tuy nhiên, nếu không có kết quả, thì `Illuminate\Database\Eloquent\ModelNotFoundException` sẽ được ném ra:

```
$model = App\Flight::findOrFail(1);
```

```
$model = App\Flight::where('legs', '>', 100)->firstOrFail();
```

Not Found Exceptions

- Nếu exception mà không được bắt, một HTTP response 404 sẽ tự động được gửi lại cho user, vì thế, không cần thiết phải viết code riêng để kiểm tra để trả về 404 khi sử dụng những hàm này:

```
Route::get('/api/flights/{id}', function ($id) {  
    return App\Flight::findOrFail($id);  
});
```

Lấy Aggregates



- Bạn cũng có thể sử dụng các hàm như count , sum , max , and other hàm aggregate cung cấp bởi query builder. Những hàm này trả về một kết quả thay vì một model instance

```
$count = App\Flight::where('active', 1)->count();
```

```
$max = App\Flight::where('active', 1)->max('price');
```

Thêm & Cập nhật Models



- Thêm
- Cập nhật

- Để thêm dữ liệu mới vào database, đơn giản hãy tạo một model instance mới, thiết lập các attributes vào model rồi gọi hàm save :

```
<?php

namespace App\Http\Controllers;

use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;
```

```
class FlightController extends Controller
{
    /**
     * Create a new flight instance.
     *
     * @param Request $request
     * @return Response
     */
    public function store(Request $request)
    {
        // Validate the request...

        $flight = new Flight;

        $flight->name = $request->name;

        $flight->save();
    }
}
```



- Ở ví dụ này, chúng ta có thể đơn giản chỉ gán tham số name từ HTTP request vào thuộc tính name của model App\Flight. Khi gọi hàm save , một record sẽ được thêm vào database. Timestamp created_at và updated_at timestamps sẽ tự động được thêm khi hàm save được gọi, và bạn không cần thay đổi thủ công giá trị này.

- Hàm save cũng được dùng để cập nhật model đã tồn tại sẵn trong database. Để update, bạn cần lấy model instance ra trước, thay đổi các attribute bạn muốn, rồi gọi hàm save . Một lần nữa, giá trị của updated_at sẽ tự động được cập nhật, và bạn không cần thay đổi thủ công giá trị này:

```
$flight = App\Flight::find(1);  
  
$flight->name = 'New Flight Name';  
  
$flight->save();
```

Mass Updates

- Update cũng có thể được thực hiện cho nhiều model mà thoả mãn một điều kiện query. Ở ví dụ này, tất cả các flights mà active và có một destination là San Diego sẽ bị đánh dấu là delayed:

- Hàm update tương ứng với các cột được cập

```
App\Flight::where('active', 1)
->where('destination', 'San Diego')
->update(['delayed' => 1]);
```



Khi thực hiện một mass update bằng Eloquent, hàm `saved` và `updated` model events sẽ không được bắn ra cho các model updated. Vì các model thực ra chưa bao giờ lấy khi mass update.

Mass Assignment

- Bạn cũng có thể sử dụng hàm create để tạo một model mới chỉ trong một dòng. Model instance được thêm mới sẽ được trả lại từ hàm. Tuy nhiên, để làm được điều đó, bạn cần thiết phải chỉ định thuộc tính fillable hoặc guarded trong model, để Eloquent model được bảo vệ trước mass-assignment.
- Lỗi bảo mật mass-assignment xảy ra khi một user truyền vào một tham số HTTP không mong muốn trong request, và tham số đó sẽ có thể thay đổi một column trong database mà bạn không ngờ tới. Ví dụ, một user xấu có thể gửi một tham số is_admin qua HTTP request, và khi giá trị này được map vào trong model qua hàm create, sẽ cho phép user thay đổi để biến thành một admin.

Mass Assignment




- Vì thế, để bắt đầu, bạn cần khai báo thuộc tính bạn muốn cho phép mass-assignment. Bạn có thể thiết lập qua thuộc tính \$fillable. Ví dụ, hãy làm cho thuộc tính name của model Flight mass assignable:

```
<?php

namespace App;

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



```
class Flight extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @var array
     */
    protected $fillable = ['name'];
}
```

Mass Assignment



- Sau đó, chúng ta có thể sử dụng hàm create để tạo một record mới trong database. Hàm create sẽ trả về một model instance được lưu:

```
$flight = App\Flight::create(['name' => 'Flight 10']);
```

Thuộc tính guarding

- Trong khi \$fillable dùng để lưu danh sách các thuộc tính "white list" được mass assignable, bạn có thể sử dụng \$guarded . Thuộc tính \$guarded để lưu các thuộc tính mà không được phép mass assignable. Các thuộc tính khác không lưu trong nó sẽ được mass assignable. Vì vậy, \$guarded giống như là một "black list". Tất nhiên, Bạn có thể sử dụng một trong hai, \$fillable hoặc \$guarded - không cả hai. Trong ví dụ dưới, tất cả các thuộc tính ngoại trừ price price sẽ được mass assignable:

Thuộc tính guarding



```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
    /**
     * The attributes that aren't mass assignable.
     *
     * @var array
     */
    protected $guarded = ['price'];
}
```

Thuộc tính guarding

- Nếu bạn muốn tất cả các thuộc tính mass assignable, bạn định nghĩa thuộc tính \$guarded là một mảng rỗng:

```
/**  
 * The attributes that aren't mass assignable.  
 *  
 * @var array  
 */  
protected $guarded = [];
```


Các hàm tạo khác

- Còn hai hàm khác bạn có thể sử dụng để model bằng cách mass-assignment các thuộc tính: `firstOrCreate` và `firstOrCreate`. Hàm `firstOrCreate` sẽ cố gắng tìm trong database sử dụng cặp column và giá trị truyền vào. Nếu model không được tìm thấy trong database, một dòng record mới sẽ được thêm vào với các attributes được truyền vào.

Các hàm tạo khác

- Hàm `firstOrCreate` giống như hàm `firstOrNew` sẽ cố gắng tìm record trong database khớp với các attribute truyền vào. Tuy nhiên, nếu model không tìm thấy, một model instance mới sẽ được trả về. Chú ý là model được trả về bởi `firstOrNew` vẫn chưa được lưu vào database. Bạn cần gọi hàm `save` để lưu nó lại:

```
// Retrieve the flight by the attributes, or create it if it doesn't exist.  
$flight = App\Flight::firstOrCreate(['name' => 'Flight 10']);  
  
// Retrieve the flight by the attributes, or instantiate a new instance...  
$flight = App\Flight::firstOrNew(['name' => 'Flight 10']);
```

Xóa Models



- Để xóa một model, gọi hàm delete trong model instance:

```
$flight = App\Flight::find(1);
```

```
$flight->delete();
```



Xoá model tồn tại bằng một key

- Ví dụ trên, chúng ta lấy model từ database trước khi gọi hàm delete. Tuy nhiên, nếu bạn đã biết primary key của model, bạn có thể xoá model mà không cần lấy nó ra. Để làm được việc này, bạn chỉ cần gọi hàm destroy :

```
App\Flight::destroy(1);
```

```
App\Flight::destroy([1, 2, 3]);
```

```
App\Flight::destroy(1, 2, 3);
```

Xóa các Models bằng truy vấn

- Bạn cũng có thể thực hiện gọi một query để xóa một tập hợp các model. Ở ví dụ này, chúng ta sẽ xóa tất cả các flights được đánh dấu là inactive:

```
$deletedRows = App\Flight::where('active', 0)->delete();
```



Khi thực thi một mass delete qua Eloquent, the `deleting` và `deleted` model events sẽ không được bắn ra cho các model deleted. Bởi vì các models thực ra chưa bao giờ lấy khi thực thi lệnh delete.

Hướng dẫn

- Hướng dẫn làm bài thực hành và bài tập
- Chuẩn bị bài tiếp: ***Eloquent nâng cao***