DATABASE LARAVEL

Phương thức truy vấn Database Laravel

Trong Laravel có 3 phương thức truy vấn với Database

- Truy vấn SQL thuần
- Query Builder
- Eloquent ORM

Tùy theo nhu cầu, thói quen, độ phức tạp của bài toán lập trình viên có thể lựa chọn phương thức phù hợp

Thiết lập cấu hình kết nối Database Laravel

Để cấu hình Database trong Laravel, bạn thực hiện cấu hình trong file `config/database.php` hoặc file `.env`

Thiết lập file .env

```
DB_CONNECTION=mysql
DB_HOST=localhost
DB_DATABASE=ten_csdl
DB_USERNAME=user
DB_PASSWORD=pass
```

Thiết lập file config/database.php

```
'default' => env('DB_CONNECTION', 'mysql'),
'connections' => [
...

    'mysql' => [
        'driver'    => 'mysql',
```

```
        'host'      => env('DB_HOST', 'localhost'),
        'database'  => env('DB_DATABASE', 'ten_db'),
        'username'  => env('DB_USERNAME', 'user'),
        'password'  => env('DB_PASSWORD', 'pass'),
        'charset'   => 'utf8',
        'collation' => 'utf8_unicode_ci',
        'prefix'    => '',
        'strict'    => false,
    ],
...
],
```

Laravel hỗ trợ kiến trúc đa máy chủ Database. Sau đây là ví dụ tách riêng việc đọc, ghi dữ liệu ở 2 máy chủ khác nhau

```
'mysql' => [
    'read' => [
        'host' => '192.168.1.1',
    ],
    'write' => [
        'host' => '196.168.1.2'
    ],
    'driver'    => 'mysql',
    'host'      => env('DB_HOST', 'localhost'),
    'database'  => env('DB_DATABASE', 'ten_csdl'),
    'username'  => env('DB_USERNAME', 'user'),
    'password'  => env('DB_PASSWORD', 'pass'),
    'charset'   => 'utf8',
    'collation' => 'utf8_unicode_ci',
    'prefix'    => '',
    'strict'    => false,
],
```

Ngoài ra, Laravel hỗ trợ việc kết nối tới nhiều CSDL bằng cách sử dụng phương thức `connection()`

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

Trong đó **connection_name** được khai báo trong file `config/database.php`

Raw Query trong Laravel

Raw Query là việc các bạn sẽ dùng câu lệnh SQL thuần để thực hiện việc truy vấn dữ liệu trong Database

Facade DB cung cấp các phương thức để truy vấn Database: select(), insert(), update(), delete() và statement()

Như vậy, trước khi sử dụng bạn cần phải `use Illuminate\Support\Facades\DB` hoặc `use DB` để sử dụng các phương thức của lớp `DB()`

Phương thức select()

Phương thức `select()` sẽ trả về một mảng, mỗi kết quả trong mảng là một đối tượng `StdClass` trong PHP

Ví dụ 1:

```
$users = DB::select('select * from users where status = ?', [1]);
```

Ví dụ 2:

```
$results = DB::select('s
elect * from users where email = :email', ['email' => 'unicode.vn@gmail.com']);
```

Qua 2 ví dụ trên bạn sẽ thấy giống cách truyền dữ liệu của PDO vì Laravel cũng dùng PDO

Để đọc dữ liệu, bạn tham khảo ví dụ sau:

```
<table>
    <tr>
        <th>User ID</th>
        <th>User name</th>
    </tr>
@foreach($users as $user)
    <tr>
        <th>{{ $user->id }}</th>
        <th>{{ $user->name }}</th>
    </tr>
@endforeach
<table>
```

Lưu ý: Mỗi item của mảng lúc này là stdClass nên bạn cần sử dụng cách gọi của Object

Phương thức insert()

Phương thức này sẽ thực hiện thêm dữ liệu vào Database

```
DB::insert(INSERT INTO users (id, name) VALUES (?, ?)',
[1, 'Unicode Academy']);
```

Phương thức update()

Phương thức này sẽ thực hiện việc cập nhật dữ liệu trong Database

```
$updated = DB::update('UPDATE users SET name = 'Unicode'
where id = ?', [1]);
```

Phương thức delete()

Phương thức này sẽ thực hiện xoá dữ liệu trong Database

```
$deleted = DB::delete('DELETE FROM users WHERE id = ?',
[1]);
```

Phương thức statement()

Phương thức này dùng để thực thi bất kỳ câu lệnh SQL

```
DB::statement('DROP TABLE users');
```

Transaction Laravel

Giao dịch (transaction) là một nhóm các hành động có thứ tự trong Database nhưng lại được người dùng xem như là một đơn vị thao tác duy nhất.

Ví dụ: Một giao dịch thanh toán trực tuyến trên web TMDT

- Trừ tiền từ tài khoản người dùng.
- Cộng tiền vào tài khoản nhà cung cấp.
- Giảm số lượng hàng trong kho

Khi đó, bất kỳ một hành động nào bị lỗi sẽ dẫn đến giao dịch lỗi. Trong thực tế khi lập trình, người ta thường nhóm các nhiều các hành động trên CSDL vào thành một nhóm và thực hiện chúng như một giao dịch, nếu một hành động lỗi, hệ thống sẽ thực hiện các lệnh sao cho quay về trạng thái ban đầu (rollback). Laravel cũng hỗ trợ quản lý transaction, thực hiện chúng khá đơn giản với cú pháp:

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

    DB::table('posts')->delete();
}, 5);
```

Tham số thứ 2 của phương thức transaction() là số lần thử thực hiện lại khi gặp tình trạng deadlock

Quản lý transaction thủ công

Ngoài ra bạn có thể thao tác quản lý transaction một cách thủ công như khi nào mới rollback hoặc khi nào mới commit một cách tùy ý:

```
DB::beginTransaction();
// Bắt đầu các hành động trên CSDL

...
//Gặp lỗi nào đó mới rollback
DB::rollBack();
...

//Commit dữ liệu khi hoàn thành kiểm tra
DB::commit();
```

Query Builder trong Laravel

Query Builder là một class trong Laravel, cung cấp cho lập trình các phương thức để dễ dàng truy vấn Database mà không cần sử dụng đến câu lệnh SQL (Trừ 1 số trường hợp đặc biệt)

Class này sử dụng PDO để thực thi các câu lệnh, điều này giúp ứng dụng của bạn tránh được lỗi bảo mật SQL Injection. Chính vì vậy, bạn không cần xử lý dữ liệu trước khi truy vấn, chèn vào Database

Để sử dụng Query Builder trong Laravel, bạn cần gọi 1 trong 2 đoạn code sau ở đầu file muốn sử dụng

```
use DB;
```

```
use Illuminate\Support\Facades\DB;
```

Đây chính là **Facade DB** đã được sử dụng trong phần **Raw Query**

Lấy tất cả dữ liệu trong bảng

```
DB::table('tablename')->get();
```

Lấy bản ghi đầu tiên

```
$userInfo = DB::table('users')->first();
```

Select cột trong bảng

```
DB::table('tablename')->select('columnfirst','columnsecon
d')->get();
```

Truy vấn có điều kiện (Where)

Điều kiện bằng (=)

```
DB::table('tablename')->where('column','value')->get();
```

Điều kiện lớn hơn (>)

```
DB::table('tablename')->where('column','>','value')->get(
);
```

Điều kiện nhỏ hơn (<)

```
DB::table('tablename')->where('column','<','filter')->get
();
```

Điều kiện khác (<>)

```
DB::table('tablename')->where('column','<>','filter')->ge
t();
```

Kết hợp điều kiện (AND, OR)

```
$users = DB::table('users')->where('votes', '>',
100)->orWhere('name', 'John')->get();
```

Truy vấn tìm kiếm (LIKE)

```
DB::table('tablename')->where('column','like','filter')->
get();
```

Truy vấn trong khoảng (whereBetween)

```
$users = DB::table('users')->whereBetween('votes', [1,
100])->get();
```

Truy vấn không trong khoảng (whereNotBetween)

```
$users = DB::table('users')->whereNotBetween('votes', [1,
100])->get();
```

Truy vấn toán tử IN (whereIn, whereNotIn)

```
$users = DB::table('users')->whereIn('id', [1, 2,
3])->get();
```

```
$users = DB::table('users')->whereNotIn('id', [1, 2,
3])->get();
```

Truy vấn kiểm tra tra NULL (whereNull, whereNotNull)

```
$users =
DB::table('users')->whereNull('update_at')->get();
```

```
$users =
DB::table('users')->whereNotNull('update_at')->get();
```

Truy vấn Date (whereDate, whereMonth, whereDay, whereYear)

```
$users = DB::table('users')->whereDate('create_at',
'2021-10-10')->get();
```

```
$users = DB::table('users')->whereMonth('create_at',
'10')->get();
```

```
$users = DB::table('users')->whereDay('create_at',
'10')->get();
```

```
$users = DB::table('users')->whereYear('create_at',
'2021')->get();
```

Truy vấn giá trị cột

```
//So sánh 2 cột bằng nhau
$users = DB::table('users')->whereColumn('first_name',
'last_name')->get();
```

```
//So sánh với các toán tử so sánh
$users = DB::table('users')->whereColumn('updated_at',
'>', 'created_at')->get();
```

```
//Kết hợp điều kiện (AND)
$users = DB::table('users')
```

```
                ->whereColumn([
                    ['first_name', '=', 'last_name'],
                    ['updated_at', '>', 'created_at']
                ])->get();
```

Nối bảng (join)

```
//Inner join
$users = DB::table('users')->join('contacts', 'users.id',
'=', 'contacts.user_id')->get();
```

```
//Left join
$users = DB::table('users')->leftJoin('contacts',
'users.id', '=', 'contacts.user_id')->get();
```

```
//Right join
$users = DB::table('users')->rightJoin('contacts',
'users.id', '=', 'contacts.user_id')->get();
```

Sắp xếp (orderBy)

```
//Sắp xếp 1 cột
$users = DB::table('users')->orderBy('name',
'desc')->get();
```

```
//Sắp xếp nhiều cột
$users = DB::table('users')->orderBy('name',
'desc')->orderBy('email', 'asc')->get();
```

```
//Sắp xếp ngẫu nhiên
$randomUsers =
DB::table('users')->inRandomOrder()->get();
```

Truy vấn theo nhóm (groupBy, having)

```
$users =
DB::table('users')->groupBy('account_id')->having('accoun
t_id', '>', 100)->get();
```

Giới hạn (limit, offset)

```
$users = DB::table('users')->offset(10)->limit(5)->get();
```

hoặc

```
$users = DB::table('users')->skip(10)->take(5)->get();
```

Thêm dữ liệu vào table (Insert)

```
//Thêm 1 bản ghi
DB::table('users')->insert(
    ['email' => 'john@example.com', 'votes' => 0]
);

//Thêm nhiều bản ghi
DB::table('users')->insert([
    ['email' => 'taylor@example.com', 'votes' => 0],
    ['email' => 'dayle@example.com', 'votes' => 0]
]);
```

Cập nhật bản ghi (Update)

```
DB::table('users')
            ->where('id', 1)
            ->update(['votes' => 1]);
```

Xoá bản ghi (Delete)

```
DB::table('users')->where('id', 1)->delete();
```

Lấy id sau khi Insert

```
$id = DB::getPdo()->lastInsertId();
```

```
$id = DB::table('users')->insertGetId(
    ['email' => 'john@example.com', 'votes' => 0]
);
```

DB Raw Query

DB::raw()

```
$users = DB::table('users')
             ->select(DB::raw('count(*) as user_count,
status'))
             ->where('status', '<>', 1)
             ->groupBy('status')
             ->get();
```

selectRaw()

```
$orders = DB::table('orders')
                ->selectRaw('price * ? as
price_with_tax', [1.0825])
                ->get();
```

whereRaw(), orWhereRaw()

```
$orders = DB::table('orders')->whereRaw('price > IF(state
```

```
= "TX", ?, 100)', [200])->get();
```

orderByRaw()

```
$orders = DB::table('orders')
                ->orderByRaw('updated_at - created_at
DESC')
                ->get();
```

groupByRaw()

```
$orders = DB::table('orders')
                ->select('city', 'state')
                ->groupByRaw('city, state')
                ->get();
```

havingRaw()

```
$orders = DB::table('orders')
                ->select('department',
DB::raw('SUM(price) as total_sales'))
                ->groupBy('department')
                ->havingRaw('SUM(price) > ?', [2500])
                ->get();
```

Debug câu lệnh SQL

```
$sql = DB::table('users')->toSql();
```

```
DB::enableQueryLog();

//Truy vấn Query Builder ở đây
```

```
dd(DB::getQueryLog());
```

Eloquent ORM trong Laravel

ORM (Object Relational Mapping): là kỹ thuật ánh xạ các record dữ liệu trong hệ quản trị cơ sở dữ liệu sang dạng đối tượng

Eloquent Model là một module trong Laravel core.

Laravel sử dụng ORM để tương tác với dữ liệu trong database một cách đơn giản linh hoạt hơn.

Khi sử dụng Eloquent thì mỗi một Table trong Database sẽ được gắn với một Model. Và chúng ta có thể tương tác với dữ liệu trong bảng đó như đọc, thêm, sửa và xóa (CRUD) qua Eloquent Model.

Tạo Eloquent Model

Để tạo Model trong Laravel, hãy sử dụng câu lệnh sau:

```
php artisan make:model TenModel
```

Trong đó: ***TenModel*** là tên model muốn tạo

File model sẽ được tự động tạo trong folder: `app/Models`

```
namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    //
```

```
}
```

Cấu hình Eloquent Model

Cấu hình tên table

Mặc định khi tạo Model trong Laravel, tên table sẽ được quy ước mặc định theo cú pháp sau:

- Chữ thường
- Mỗi chữ cách nhau bởi dấu gạch dưới _
- Dạng số nhiều

Ví dụ: Tạo Model có tên **Post** thì tên table sẽ là **posts**

Trong trường hợp muốn cấu hình riêng tên table, hãy thêm thuộc tính sau:

```
protected $table = 'posts';
```

Lưu ý: Phạm vi bắt buộc phải là `protected` và tên thuộc tính bắt buộc phải là `$table`

Cấu hình Primary Key

Mặc định, Laravel sẽ nhận trường `id` làm khóa chính. Tuy nhiên, chúng ta có thể thay đổi bằng cách thêm thuộc tính sau:

```
protected $primaryKey = 'id';
```

Trong trường hợp khóa chính không ở chế độ **Auto_Increment** và không phải kiểu số, hãy khai báo thêm thuộc tính sau:

```
public $incrementing = false;
```

Thay đổi kiểu dữ liệu cho khóa chính

```
protected $keyType = 'string';
```

Cấu hình Timestamp

Mặc định, Laravel sẽ ngầm hiểu table có sẵn 2 trường `created_at` và `updated_at`

Khi thêm mới hoặc sửa bản ghi, Laravel sẽ update dữ liệu cho 2 trường này

Nếu không muốn, hãy thêm thuộc tính sau để vô hiệu hoá

```
public $timestamps = false;
```

Trong trường hợp muốn đổi tên trường `created_at` và `updated_at` hãy bổ sung vào 2 hằng số sau:

```
const CREATED_AT = 'ten_truong_moi';
const UPDATED_AT = 'ten_truong_moi';
```

Cấu hình giá trị mặc định

```
protected $attributes = [
    'ten_truong' => gia_tri,
];
```

Cấu hình Database connection

Trong trường hợp muốn thay đổi Database Connection của 1 Model nào đó, hãy bổ sung thuộc tính sau

```
protected $connection = 'ten_connection';
```

Query Eloquent Model

Về bản chất, Query trong Eloquent Model giống như Query Builder với 1 table cụ thể (Thông qua Model)

Lấy tất cả bản ghi

```
TenModel::all()
```

Lấy 1 bản ghi

```
TenModel::find($id)
```

Sử dụng các phương thức của Query Builder

```
TenModel::where('status',
1)->orderBy('name')->limit(5)->get();
```

Collections

Các phương thức của Eloquent trả về nhiều bản ghi sẽ tự động chuyển sang dạng **Collections**. **Collections** trong Laravel cung cấp cho lập trình viên rất nhiều phương thức hữu ích để xử lý kết quả trả về.

Các phương thức Collections: https://laravel.com/docs/8.x/collections

```
$posts = Post::all();
$posts = $posts->reject(function ($post) {
    return $post->status==1;
});
```

Chunk

Nếu muốn xử lý dữ liệu lớn, sử dụng phương thức `chunk()` để tiết kiệm memory. Phương thức này sẽ chia nhỏ dữ liệu và cung cấp qua `Closure` để xử lý

```
Post::chunk(200, function ($posts) {
    foreach ($posts as $post) {
        echo $post->title. "<br>";
    }
});
```

Cursors

Phương thức `cursors()` sẽ duyệt qua records bằng cách sử dụng một con trỏ. Khi dữ liệu lớn, phương thức này được sử dụng để giảm memory

```
foreach(Post::where('status', 1)->cursor() as $post) {
    //code
};
```

Insert dữ liệu

```
TenModel::create($data);
```

- `$data`: Mảng chứa dữ liệu cần insert
- Phương thức này sẽ trả về instance của model chứa dữ liệu của record vừa insert

Lưu ý: Cần phải thiết lập thuộc tính `$fillable` trong Model

```
protected $fillable = ['name', 'password', 'email'];
```

Ngoài ra, chúng ta có thể dùng phương thức `insert()` giống như Query Builder

```
TenModel::insert($data);
```

Phương thức firstOrCreate()

Phương thức `firstOrCreate()` có tác dụng lấy ra bản ghi đầu tiên của dữ liệu phù hợp với query, nếu không có nó sẽ insert dữ liệu query vào Database và trả về bản ghi đó

```
TenModel::firstOrCreate($attributes, $values);
```

- `$attribute`: là một mảng chứa các điều kiện bạn muốn query để tìm kiếm bản ghi trong Database.
- `$values:`: là mảng chứa dữ liệu sẽ được insert vào trong database khi bản ghi cần tìm không tồn tại trong Database

Phương thức save()

```
$tenModel = new TenModel;

$tenModel->thuoctinh1 = $request->thuoctinh1;

$tenModel->thuoctinh2 = $request->thuoctinh2;

$tenModel->thuoctinh3 = $request->thuoctinh3;

$tenModel->save();
```

Update dữ liệu

Có rất nhiều cách để update dữ liệu trong Laravel, hãy tham khảo các cách sau đây:

```php
$tenModel = TenModel::find($id);

$tenModel->thuoctinh1 = $request->thuoctinh1;

$tenModel->thuoctinh2 = $request->thuoctinh2;

$tenModel->thuoctinh3 = $request->thuoctinh3;

$tenModel->save();
```

hoặc

```php
$tenModel = TenModel::find($id);

$tenModel->update($data);
```

hoặc

```php
TenModel::updateOrCreate($attributes, $values);
```

Phương thức này sẽ update nếu tìm thấy bản ghi dựa vào điều kiện của `$attributes`, nếu không tìm thấy sẽ insert

Xoá dữ liệu

```php
//Xóa 1 bản ghi
TenModel::destroy($id);

//Xóa nhiều bản ghi
TenModel::destroy($id1, $id2, $id3);

// hoặc
TenModel::destroy([$id1, $id2, $id3]);
```

```
// hoặc
TenModel::destroy(collect([$id1, $id2, $id3]));
```

Xóa mềm

Hiểu đơn giản đây là chức năng chuyển bản ghi vào thùng rác

Để thực hiện được chức năng này, chúng ta cần thêm trường `deleted_at` vào table có kiểu dữ liệu *timestamp* và cho phép *NULL*

Tiếp theo, tại Model cần thêm **Trait** sau:

```
use Illuminate\Database\Eloquent\SoftDeletes;
```

Sau đó gọi Trait trong Class của Model và khai báo thuộc tính `$dates`:

```
use SoftDeletes;

protected $dates = ['deleted_at'];
```

Lúc này, khi xoá bản ghi thay vì bản ghi đó sẽ bị xoá đi thì sẽ cập nhật thời gian vào trường `deleted_at`

Để kiểm tra bản ghi đã được xóa hay chưa, chúng ta dùng phương thức `trashed()`

```
if ($post->trashed()) {
    // Sản phẩm này đã được đánh dấu là đã xóa
}
```

Để khôi phục bản ghi đã bị xoá mềm, chúng ta sử dụng phương thức `restore()`

```
$post->restore();
```

Để lấy tất cả bản ghi kể cả bản ghi đã xoá mềm, sử dụng phương thức `withTrashed()`

```
TenModel::withTrashed()->where('id', $id)->restore();
```

Nếu muốn xoá vĩnh viễn 1 bản ghi, dùng phương thức `forceDelete()`

```
TenModel::find($id)->forceDelete();
```

Nếu muốn lấy tất cả bản ghi đã xoá mềm, sử dụng phương thức `onlyTrashed()`

```
TenModel::onlyTrashed()
```

Eloquent Relationships

Eloquent Relationship là tính năng trong Laravel nó cho phép chúng ta định nghĩ ra các mối quan hệ (relationship) giữa các model với nhau. Từ đó có thể query, làm việc với các model được định nghĩa quan hệ một cách đơn giản.

Quan hệ One To One

Quan hệ One To One là quan hệ 1 - 1 trong Database

Ví dụ: *Mỗi User chỉ có duy nhất 1 số điện thoại*

Để định nghĩa, chúng ta sử dụng phương thức `hasOne()`

```
hasOne($relationModel, $foreignKey, $localKey');
```

- `$relationModel` là model sẽ được liên kết với model hiện tại với mỗi quan hệ 1-1
- `$foreignKey` là khóa ngoại của table `$relationModel` ở trên dùng để liên kết giữa 2 bảng với nhau. Mặc định thì `$foreignKey` sẽ là tên bảng của Model hiện tại cộng với '_id', ví dụ model User thì sẽ là user_id
- `$localKey` là cột chứa dữ liệu để liên kết với bảng của `$relationModel`. Mặc định thì `$localKey` sẽ là khóa chính của Model hiện tại.

Ví dụ mẫu:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
```

```
     * Get the phone associated with the user.
     */
    public function phone()
    {
        return $this->hasOne(Phone::class);
    }
}
```

Lúc này thì `$foreignKey` và `$localKey` sẽ là:

- `$foreignKey`: user_id
- `$localKey`: id

Sau khi đã định nghĩa xong relationship cho Model lúc này chúng ta có thể query đến relation model bằng cách gọi đến phương thức định nghĩa relation như một thuộc tính trong model hiện tại

Cú pháp:

```
TenModel::find($id)->tenPhuongThuc;
```

Lưu ý: Định nghĩa là phương thức, nhưng khi truy vấn là thuộc tính

Để đảo ngược quan hệ, chúng ta sử dụng phương thức `belongsTo()`

```
belongsTo($relatedModel, $foreignKey, $ownerKey);
```

- `$relatedModel` là model của bạn muốn liên kết.
- `$foreignKey` là tên trường của bảng hiện tại sẽ dùng để liên kết. Mặc định `$foreignKey` sẽ là tên của phương thức cộng với primary key của `$relatedModel`
- `$ownerKey` là column của bảng `$relatedModel` sẽ dùng để liên kết. Mặc định `$ownerKey` là khóa chính của `$relatedModel` model

Ví dụ mẫu:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

Quan hệ One To Many

Quan hệ One To Many là quan hệ 1 - N trong Database

Để định nghĩa quan hệ này, chúng ta sử dụng phương thức `hasMany()`

Ví dụ mẫu:

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany(Comment::class);
    }
}
```

Ở ví dụ này khi query đến relationship nó sẽ trả về một collection với các item sẽ là model object

```php
use App\Models\Post;

$comments = Post::find(1)->comments;

foreach ($comments as $comment) {
    //
}
```

Nếu muốn đảo ngược quan hệ, chúng ta cần sử dụng phương thức `belongsTo()`

```php
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Comment extends Model
{
    /**
     * Get the post that owns the comment.
     */
    public function post()
    {
        return $this->belongsTo(Post::class);
    }
}
```

Truy vấn dữ liệu:

```php
use App\Models\Comment;

$comment = Comment::find(1);

return $comment->post->title;
```

Kết quả chỉ trả về 1 post

Quan hệ Has One Through

Quan hệ Has One Through là quan hệ 1-1 nhưng sẽ liên kết thông qua Model khác

Cùng phân tích Database sau:

mechanics
- id - integer
- name - string

cars
id - integer
model - string
mechanic_id - integer

owners
id - integer
name - string
car_id - integer

Lúc này để định nghĩa owner của mechanics các bạn có thể sử dụng phương thức `hasOneThrough()`

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Mechanic extends Model
{
    /**
     * Get the car's owner.
     */
    public function carOwner()
    {
        return $this->hasOneThrough(Owner::class,
Car::class);
    }
}
```

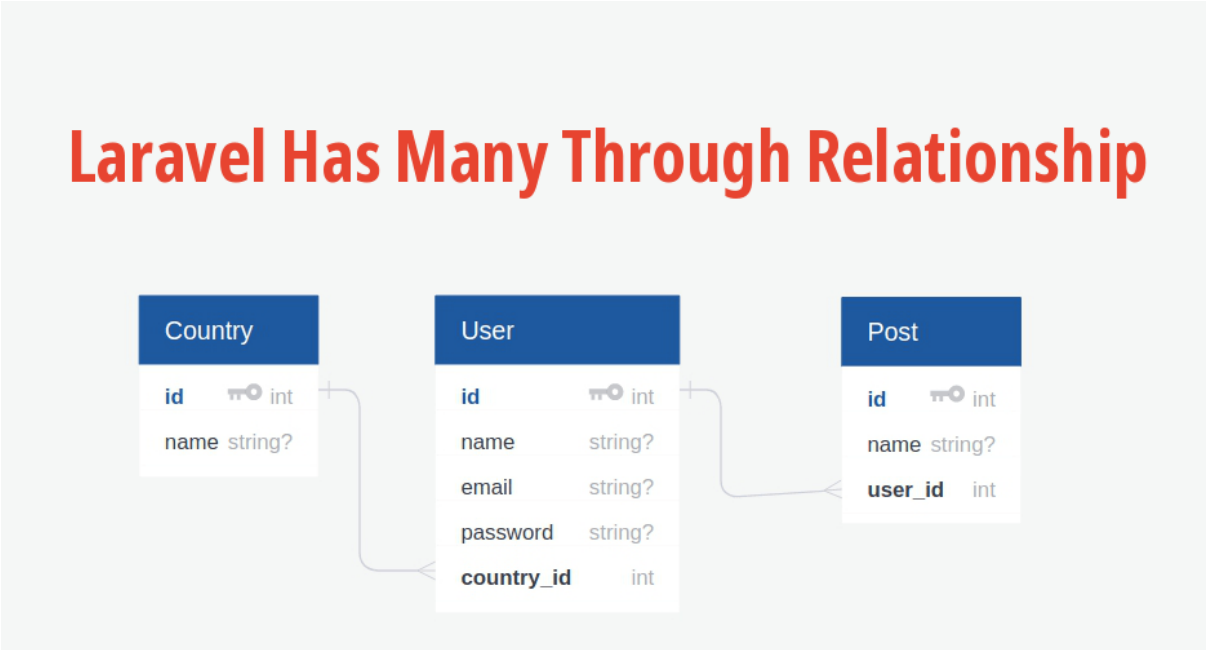
Nếu như khoá ngoại và khoá chính không tuân thủ nguyên như cấu trúc table trên, cần thiết lập lại như sau

```
class Mechanic extends Model
{
```

```
    /**
     * Get the car's owner.
     */
    public function carOwner()
    {
        return $this->hasOneThrough(
            Owner::class,
            Car::class,
            'mechanic_id', // Foreign key on the cars
table...
            'car_id', // Foreign key on the owners
table...
            'id', // Local key on the mechanics table...
            'id' // Local key on the cars table...
        );
    }
}
```

Quan hệ Has Many Through

Quan hệ Has Many Through là quan hệ 1 - N nhưng sẽ liên kết qua bảng trung gian



Để định nghĩa quan hệ này các bạn sử dụng phương thức `hasManyThrough()`

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Country extends Model
{

    public function posts()
    {
        return $this->hasManyThrough(Post::class,
User::class);
    }
}
```

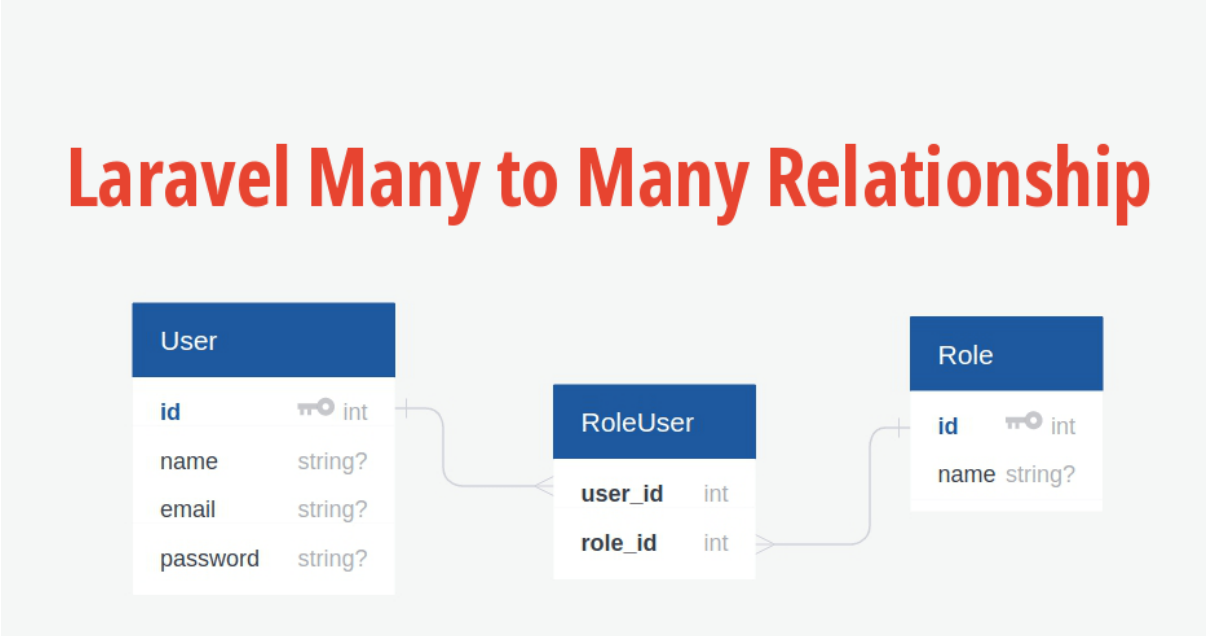
Nếu như khoá ngoại và khoá chính không tuân thủ nguyên như cấu trúc table trên, cần thiết lập lại như sau

```php
class Country extends Model
{
    public function posts()
    {
        return $this->hasManyThrough(
            Post::class,
            User::class,
            'country_id', // Foreign key on the users
table...
            'user_id', // Foreign key on the posts
table...
            'id', // Local key on the country table...
            'id' // Local key on the users table...
```

```
        );
    }
}
```

Quan hệ Many To Many

Quan hệ Many To Many là quan hệ N - N trong Database. Quan hệ này sẽ có bảng trung gian



Để thiết lập quan hệ này, chúng ta sử dụng `belongsToMany()`

```php
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    /**
     * The roles that belong to the user.
```

```
     */
    public function roles()
    {
        return $this->belongsToMany(Role::class);
    }
}
```

Nếu muốn thiết lập quan hệ đảo ngược, chúng ta thiết lập giống như trên:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Role extends Model
{
    /**
     * The users that belong to the role.
     */
    public function users()
    {
        return $this->belongsToMany(User::class);
    }
}
```

Trong một số trường hợp, bạn muốn lấy ra thêm data của bảng trung gian, chúng ta có thể sử dụng thuộc tính `pivot`

Ví dụ:

```
use App\Models\User;

$user = User::find(1);
```

```php
foreach ($user->roles as $role) {
    echo $role->pivot->created_at;
}
```

Default Model

Trong các phương thức liên kết quan hệ trong ORM sẽ trả về `null` nếu dữ liệu quan hệ giữa các table không tồn tại. Trong trường hợp này laravel có cung cấp cho mọi người định nghĩa thêm giá trị default với phương thức `withDefault()`.

```php
public function user()
{
    return $this->belongsTo(User::class)->withDefault();
}
```

Lúc này, Laravel sẽ trả về model object và không có thuộc tính mặc định

Nếu muốn thêm thuộc tính mặc định vào, hãy tham khảo cú pháp sau:

```php
public function user()
{
    return $this->belongsTo(User::class)->withDefault([
        'name' => 'Khách',
    ]);
}
```

```php
public function user()
{
    return
$this->belongsTo(User::class)->withDefault(function
($user, $post) {
        $user->name = 'Khách';
```

```
    });
}
```

Truy vấn dữ liệu trong Eloquent Relationships

Sau khi các dạng quan hệ được định nghĩa trong Eloquent thông qua các phương thức, bạn có thể gọi các phương thức để lấy dữ liệu về như các phương thức sử dụng trong Query Builder.

Lấy dữ liệu liên kết

```
$user = User::find(1);

$posts = $user->posts;
```

Lọc dữ liệu liên kết

```
$user = User::find(1);

$posts = $user->posts()->where('status', 1)->get();
```

Lấy kết quả dựa vào ràng buộc liên kết

```
//Lấy danh sách bài viết có ít nhất 1 comment
$posts = Post::has('comments')->get();

//Lấy danh sách bài viết có từ 3 comments trở lên
$posts = Post::has('comments', '>=', 3)->get();
```

Ngoài ra, Eloquent cung cấp các phương thức `whereHas()` và `orWhereHas()` cho phép thêm các điều kiện **where** vào trong truy vấn

```
$posts = Post::whereHas('comments', function ($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

Trái ngược với phương thức `has()`, `whereHas()`, `orWhereHas()` Eloquent cung cấp phương thức `doesntHave()`, `whereDoesntHave()`

```
$posts = Post::doesntHave('comments')->get();

$posts = Post::whereDoesntHave('comments', function
($query) {
    $query->where('content', 'like', 'foo%');
})->get();
```

Eloquent cung cấp thêm phương thức `withCount()` giúp đếm kết quả trả về và đặt vào một cột có tên mặc định là `{relation}_count`

```
$posts = Post::withCount('comments')->get();

foreach ($posts as $post) {
    echo $post->comments_count;
}
```

withCount() với nhiều quan hệ

```
$posts = Post::withCount(['votes', 'comments' => function
($query) {
    $query->where('content', 'like', 'foo%');
}])->get();

echo $posts[0]->votes_count;
echo $posts[0]->comments_count;
```

Nếu muốn đặt tên cho cột kết quả, tham khảo cú pháp sau:

```php
$posts = Post::withCount([
    'comments',
    'comments AS pending_comments' => function ($query) {
        $query->where('approved', false);
    }
])->get();

echo $posts[0]->comments_count;

echo $posts[0]->pending_comments_count;
```

Eager load - Tải dữ liệu một lần

Với cách mặc định trong Laravel, số lượng truy vấn sẽ rất nhiều. Tuy nhiên, với Eager Load số lượng truy vấn sẽ ít hơn

```php
$posts = Posts::with('group')->get();

foreach ($posts as $post) {
    echo $posts->group->name.'<br/>';
}
```

Với đoạn code trên, câu lệnh SQL sẽ chạy như sau:

```
select * from posts
select * from groups where id in (1, 2, 3, 4,...)
```

Nếu bạn dùng cách mặc định, câu lệnh SQL sẽ chạy như sau:

```
select * from posts
select * from groups where id = 1
select * from groups where id = 2
select * from groups where id = 3
```

Như vậy, với cách mặc định truy vấn sẽ được load từng phần (Hay còn

gọi là Lazy Load)

Mỗi phương pháp sẽ có ưu điểm và nhược điểm riêng. Tuỳ vào từng tình huống cụ thể, chúng ta chọn phương pháp cho phù hợp.

Thêm ràng buộc với phương thức with()

```
$posts = Posts::with(['groups' => function ($query) {
    $query->where('id', '>', 1);
}])->get();
```

Lazy eager load

Kỹ thuật này sẽ trả được xử lý sau khi truy vấn lấy danh sách bản ghi

```
$posts = Posts::all();
$posts->load('groups');
```

Cập nhật dữ liệu vào các Model có quan hệ

Phương thức save()

```
$comment = new Comment(
[
'content' => 'A new comment',
'name' => 'New Name'
]);

$post = Post::find(1);

$post->comments()->save($comment);
```

Phương thức saveMany()

```
$post = Post::find(1);

$post->comments()->saveMany([
    new Comment(['message' => 'A new comment',
'name'=>'Name 1']),
    new Comment(['message' => 'Another comment', 'name'
=> 'Name 2']),
]);
```

Phương thức create()

Giống như chức năng của save() nhưng đầu vào của create() là một mảng thay vì một instance của Model

```
$post = Post::find(1);

$comment = $post->comments()->create([
    'message' => 'A new comment.',
    'title' => 'New Name'
]);
```