Course - Laravel Framework

Query Builder

Query builder trong cơ sở dữ liệu của Laravel cung cấp một interface thuận tiện, có thể xâu chuỗi các cuộc gọi hàm trong một dòng code nhằm để tạo và chạy các truy vấn SQL của cơ sở dữ liệu.

Tags: query builder, truy van database, laravel

Giới thiệu

Query builder trong cơ sở dữ liệu của Laravel cung cấp một interface thuận tiện, có thể xâu chuỗi các cuộc gọi hàm trong một dòng code nhằm để tạo và chạy các truy vấn SQL của cơ sở dữ liệu. Nó có thể được sử dụng để thực hiện hầu hết các hoạt động cơ sở dữ liệu trong ứng dụng của bạn và hoạt động hoàn hảo với tất cả các hệ thống cơ sở dữ liệu được hỗ trợ của Laravel.

Query builder Laravel sử dụng tham số PDO để bảo vệ ứng dụng của bạn chống lại các cuộc tấn công SQL injection. Không cần phải làm sạch hoặc làm sạch các chuỗi được truyền vào query builder dưới dạng các ràng buộc truy vấn.

> **Chú ý:** PDO không hỗ trợ tên cột ràng buộc. Do đó, bạn không bao giờ được phép cho phép người dùng nhập tên cột mà truy vấn của bạn tham chiếu, bao gồm cả các cột "order by".

Chạy truy vấn database

Truy xuất tất cả các hàng từ một bảng

Bạn có thể sử dụng phương thức `table` được cung cấp bởi facade `DB` để bắt đầu truy vấn. Phương thức `table` trả về một đối tượng query builder cho bảng đã cho, cho phép bạn xâu chuỗi nhiều ràng buộc (constraint) hơn vào truy vấn và sau đó cuối cùng truy xuất kết quả của truy vấn bằng phương thức `get`:

```php
<?php

namespace App\Http\Controllers;

use App\Http\Controllers\Controller;
use Illuminate\Support\Facades\DB;

class UserController extends Controller
{
  /**
   * Show a list of all of the application's users.
   *
   * @return \Illuminate\Http\Response
```

```
     */
    public function index()
    {
        $users = DB::table('users')->get();

        return view('user.index', ['users' => $users]);
    }
}
```

Phương thức get trả về một đối tượng `Illuminate\Support\Collection` có chứa các kết quả của truy vấn. Trong đó, mỗi kết quả là một đối tượng PHP `stdClass`. Bạn có thể truy cập giá trị của từng cột bằng cách truy cập cột đó như một thuộc tính của đối tượng:

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

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

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

> Collection Laravel cung cấp nhiều phương pháp cực kỳ mạnh mẽ để ánh xạ và thu nhỏ dữ liệu. Để biết thêm thông tin về collection Laravel, hãy xem tài liệu collection.

Truy xuất một hàng/cột từ một bảng

Nếu bạn chỉ cần truy xuất một hàng đơn lẻ từ bảng cơ sở dữ liệu, bạn có thể sử dụng phương thức first của facade `DB`. Phương thức này sẽ trả về một đối tượng `stdClass`:

```
$user = DB::table('users')->where('name', 'John')->first();

return $user->email;
```

Nếu bạn không muốn lấy toàn bộ dữ liệu của một hàng, thì bạn cũng có thể truy xuất một giá trị từ một hàng bằng phương thức `value`. Phương thức này sẽ trả về trực tiếp giá trị của một cột:

```php
$email = DB::table('users')->where('name', 'John')->value('email');
```

Để truy xuất một hàng theo giá trị cột `id` của nó, hãy sử dụng phương thức `find`:

```php
$user = DB::table('users')->find(3);
```

Truy xuất một danh sách các giá trị cột

Nếu bạn muốn lấy một đối tượng `Illuminate\Support\Collection` chứa các giá trị của một cột, bạn có thể sử dụng phương thức `pluck`. Trong ví dụ này, chúng tôi sẽ truy xuất một collection các tiêu đề của người dùng:

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

$titles = DB::table('users')->pluck('title');

foreach ($titles as $title) {
  echo $title;
}
```

Bạn có thể chỉ định cột mà collection kết quả sẽ sử dụng như khóa tìm kiếm của nó bằng cách cung cấp đối số thứ hai cho phương thức `pluck`:

```php
$titles = DB::table('users')->pluck('title', 'name');

foreach ($titles as $name => $title) {
  echo $title;
}
```

Phân nhỏ đều kết quả

Nếu bạn cần làm việc với hàng nghìn record cơ sở dữ liệu, hãy xem xét sử dụng phương thức `chunk` của facade `DB`. Phương thức này truy xuất một phần nhỏ kết quả tại một thời điểm và đưa mỗi phần vào một hàm xử lý. Ví dụ: chúng ta hãy truy xuất toàn bộ bảng `users` theo từng phần có 100 record như sau:

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

DB::table('users')->orderBy('id')->chunk(100, function ($users) {
  foreach ($users as $user) {
    //
  }
});
```

Bạn có thể ngừng xử lý các phần tiếp theo bằng cách trả về `false` từ hàm xử lý:

```php
DB::table('users')->orderBy('id')->chunk(100, function ($users) {
  // Process the records...

  return false;
});
```

Nếu bạn đang cập nhật record cơ sở dữ liệu trong khi phân chia kết quả, kết quả phân đoạn của bạn có thể thay đổi theo những cách không mong muốn. Nếu bạn dự định cập nhật các record đã truy xuất trong khi phân chia, tốt nhất bạn nên sử dụng phương thức `chunkById` để thay thế. Phương pháp này sẽ tự động phân trang các kết quả dựa trên khóa chính (khóa dùng để tìm kiếm) của record:

```php
DB::table('users')->where('active', false)
  ->chunkById(100, function ($users) {
    foreach ($users as $user) {
      DB::table('users')
        ->where('id', $user->id)
        ->update(['active' => true]);
    }
  });
```

Chú ý: Khi cập nhật hoặc xóa các record bên trong lệnh callback của chunk, bất kỳ thay đổi nào đối với khóa chính hoặc khóa ngoại có thể ảnh hưởng đến truy vấn chunk. Điều này có thể dẫn đến việc các record không được đưa vào các kết quả đã phân nhỏ.

Truy xuất dần dần kết quả

Phương thức `lazy` hoạt động tương tự như phương thức `chunk` nghĩa là nó sẽ thực thi truy vấn theo từng phân đoạn. Nhưng thay vì truyền từng đoạn vào một lệnh callback, thì phương thức `lazy()` trả về một `LazyCollection`, nó cho phép bạn tương tác với các kết quả dưới dạng một luồng truy xuất duy nhất:

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

DB::table('users')->orderBy('id')->lazy()->each(function ($user) {
  //
});
```

Một điều nữa, nếu bạn định cập nhật các record đã truy xuất trong khi lặp qua chúng, tốt nhất là sử dụng các phương thức `lazyById` hoặc `lazyByIdDesc` để thay thế. Các phương thức này sẽ tự động phân trang kết quả dựa trên khóa chính của record:

```
DB::table('users')->where('active', false)
  ->lazyById()->each(function ($user) {
    DB::table('users')
      ->where('id', $user->id)
      ->update(['active' => true]);
  });
```

> **Chú ý:** Khi cập nhật hoặc xóa record trong khi lặp qua chúng, bất kỳ thay đổi nào đối với khóa chính hoặc khóa ngoại đều có thể ảnh hưởng đến truy vấn phân đoạn. Điều này có thể dẫn đến việc record không được đưa vào kết quả.

Truy vấn số liệu tổng hợp

Query builder cũng cung cấp nhiều phương thức để truy xuất các giá trị tổng hợp như `count`, `max`, `min`, `avg` và `sum`. Bạn có thể gọi bất kỳ phương thức nào sau đây sau khi tạo truy vấn của mình:

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

```php
$users = DB::table('users')->count();

$price = DB::table('orders')->max('price');
```

Tất nhiên, bạn có thể kết hợp các phương thức này với các mệnh đề khác để tinh chỉnh cách tính giá trị tổng hợp của bạn:

```php
$price = DB::table('orders')
  ->where('finalized', 1)
  ->avg('price');
```

Xác định xem record có tồn tại

Thay vì sử dụng phương thức count để xác định xem có record nào phù hợp với các ràng buộc của truy vấn hay không, bạn có thể sử dụng phương thức `exists` và phương thức `doesntExist`:

```php
if (DB::table('orders')->where('finalized', 1)->exists()) {
  // ...
}

if (DB::table('orders')->where('finalized', 1)->doesntExist()) {
  // ...
}
```

Các lệnh Select

Mô tả mệnh đề select

Có thể không phải lúc nào bạn cũng muốn chọn hết tất cả các cột từ bảng cơ sở dữ liệu. Bằng việc sử dụng phương thức `select`, bạn có thể tạo được mệnh đề "select" theo cách bạn muốn cho truy vấn của bạn:

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

```php
$users = DB::table('users')
  ->select('name', 'email as user_email')
  ->get();
```

Phương thức `distinct` cho phép truy vấn trả về các kết quả một cách riêng biệt:

```php
$users = DB::table('users')->distinct()->get();
```

Nếu bạn đã có một đối tượng query builder và bạn muốn thêm một cột vào mệnh đề select hiện có của nó, bạn có thể sử dụng phương thức `addSelect`:

```php
$query = DB::table('users')->select('name');

$users = $query->addSelect('age')->get();
```

Các biểu thức thuần

Đôi khi bạn có thể cần phải chèn một chuỗi tùy ý vào một truy vấn. Để tạo một biểu thức với chuỗi thuần túy, bạn có thể sử dụng phương thức `raw` được cung cấp bởi facade `DB`:

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

> **Chú ý:** Các câu lệnh thuần sẽ được đưa vào truy vấn dưới dạng chuỗi, vì vậy bạn phải cực kỳ cẩn thận để tránh các cuộc tấn công SQL injection.

Các phương thức raw khác

Thay vì sử dụng phương thức DB::raw, thì bạn cũng có thể sử dụng các phương thức sau để chèn một biểu thức thuần vào các phần khác nhau của truy vấn của mình. Hãy nhớ rằng, Laravel không thể đảm bảo rằng bất kỳ truy vấn nào sử dụng biểu thức thuần cũng đều được bảo vệ khỏi các lỗ hổng SQL injection.

`selectRaw`

Phương thức `selectRaw` có thể được sử dụng thay cho `addSelect(DB::raw(...))`. Phương thức này chấp nhận một mảng ràng buộc không bắt buộc làm đối số thứ hai của nó:

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

`whereRaw` / `orWhereRaw`

Phương thức `whereRaw` và `orWhereRaw` có thể được sử dụng để đưa mệnh đề "where" thuần vào truy vấn của bạn. Các phương thức này chấp nhận một mảng ràng buộc không bắt buộc làm đối số thứ hai của chúng:

```
$orders = DB::table('orders')
  ->whereRaw('price > IF(state = "TX", ?, 100)', [200])
  ->get();
```

`havingRaw` / `orHavingRaw`

Các phương thức `havingRaw` và `orHavingRaw` có thể được sử dụng để cung cấp một chuỗi thuần làm giá trị của mệnh đề "having". Các phương thức này chấp nhận một mảng ràng buộc không bắt buộc làm đối số thứ hai của chúng:

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

`orderByRaw`

Phương thức `orderByRaw` có thể được sử dụng để cung cấp một chuỗi thuần làm giá trị của mệnh đề "order by":

```
$orders = DB::table('orders')
```

```
    ->orderByRaw('updated_at - created_at DESC')
    ->get();
```

groupByRaw

Phương thức `groupByRaw` có thể được sử dụng để cung cấp một chuỗi thuần làm giá trị của mệnh đề `group by`:

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

Truy vấn join

Mệnh đề inner join

Query builder cũng có thể được sử dụng để thêm các mệnh đề join vào các truy vấn của bạn. Để thực hiện truy vấn "inner join" cơ bản, bạn có thể sử dụng phương thức `join` trên một đối tượng query builder. Đối số đầu tiên được truyền cho phương thức `join` là tên của bảng mà bạn cần tham gia, trong khi các đối số còn lại chỉ định các cột cho lệnh `join`. Bạn thậm chí có thể kết hợp nhiều bảng trong một truy vấn duy nhất:

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

$users = DB::table('users')
    ->join('contacts', 'users.id', '=', 'contacts.user_id')
    ->join('orders', 'users.id', '=', 'orders.user_id')
    ->select('users.*', 'contacts.phone', 'orders.price')
    ->get();
```

Mệnh đề left join/right join

Nếu bạn muốn thực hiện lện "left join" hoặc "right join" thay vì "inner join", thì hãy sử dụng các phương thức `leftJoin` hoặc `rightJoin`. Các phương thức này có cùng hình thức với phương thức `join`:

```php
$users = DB::table('users')
  ->leftJoin('posts', 'users.id', '=', 'posts.user_id')
  ->get();

$users = DB::table('users')
  ->rightJoin('posts', 'users.id', '=', 'posts.user_id')
  ->get();
```

Mệnh đề cross join

Bạn có thể sử dụng phương thức `crossJoin` để thực hiện "cross join". Các phép nối chéo tạo ra một tích cartesian giữa bảng đầu tiên và bảng đã nối:

```php
$sizes = DB::table('sizes')
  ->crossJoin('colors')
  ->get();
```

Mệnh đề advanced join

Bạn cũng có thể chỉ định thêm các mệnh đề join. Để bắt đầu, hãy truyền một hàm xử lý làm đối số thứ hai cho phương thức `join`. Hàm xử lý sẽ nhận được một đối tượng `Illuminate\Database\Query\JoinClause` cho phép bạn chỉ định các ràng buộc đối với mệnh đề "join":

```php
DB::table('users')
  ->join('contacts', function ($join) {
    $join->on('users.id', '=', 'contacts.user_id')->orOn(...);
  })
  ->get();
```

Nếu bạn muốn sử dụng mệnh đề "where" trên các lệnh `join` của mình, thì bạn có thể sử dụng các phương thức `where` và `orWhere` được cung cấp bởi đối tượng `JoinClause`. Thay vì so sánh hai cột, các phương thức này sẽ so sánh cột với một giá trị:

```php
DB::table('users')
```

```
  ->join('contacts', function ($join) {
    $join->on('users.id', '=', 'contacts.user_id')
      ->where('contacts.user_id', '>', 5);
  })
  ->get();
```

Lệnh join trong chuỗi truy vấn con

Bạn có thể sử dụng các phương thức `joinSub`, `leftJoinSub` và `rightJoinSub` để nối một truy vấn với một truy vấn con. Mỗi phương thức này nhận được ba đối số: truy vấn con, alias của bảng và một hàm xử lý xác định các cột liên quan. Trong ví dụ này, chúng tôi sẽ truy xuất một tập hợp người dùng trong đó mỗi record người dùng cũng chứa dấu thời gian `created_at` của bài đăng blog được xuất bản gần đây nhất của người dùng:

```
$latestPosts = DB::table('posts')
  ->select('user_id', DB::raw('MAX(created_at) as last_post_created_at'))
  ->where('is_published', true)
  ->groupBy('user_id');

$users = DB::table('users')
  ->joinSub($latestPosts, 'latest_posts', function ($join) {
    $join->on('users.id', '=', 'latest_posts.user_id');
  })->get();
```

Các lệnh union

Query builder cũng cung cấp một phương thức thuận tiện để kết hợp (union) hai hoặc nhiều truy vấn lại với nhau. Ví dụ: bạn có thể tạo một truy vấn ban đầu và sử dụng phương thức `union` để kết hợp nó với các truy vấn khác:

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

$first = DB::table('users')
  ->whereNull('first_name');

$users = DB::table('users')
```

```
    ->whereNull('last_name')
    ->union($first)
    ->get();
```

Ngoài phương thức `union`, query builder sẽ cung cấp phương thức `unionAll`. Các truy vấn được kết hợp bằng cách sử dụng phương thức `unionAll` sẽ không bị xóa các kết quả trùng lặp. Phương thức `unionAll` có cùng hình thức với phương thức `union`.

Các mệnh đề where căn bản

Mệnh đề where

Bạn có thể sử dụng phương thức where của query builder để thêm mệnh đề "where" vào truy vấn. Lệnh gọi cơ bản nhất đến phương thức where yêu cầu ba đối số. Đối số đầu tiên là tên của cột. Đối số thứ hai là một toán tử, có thể là bất kỳ toán tử nào được hỗ trợ bởi cơ sở dữ liệu. Đối số thứ ba là giá trị để so sánh với giá trị của cột.

Ví dụ: truy vấn sau đây truy xuất người dùng trong đó giá trị của cột phiếu bầu bằng 100 và giá trị của cột tuổi lớn hơn 35:

```
$users = DB::table('users')
    ->where('votes', '=', 100)
    ->where('age', '>', 35)
    ->get();
```

Để thuận tiện, nếu bạn muốn xác minh rằng một cột bằng với một giá trị nhất định, bạn có thể truyền giá trị làm đối số thứ hai cho phương thức `where`. Laravel sẽ giả sử bạn muốn sử dụng toán tử `=`:

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

Như đã đề cập trước đây, bạn có thể sử dụng bất kỳ toán tử nào được hỗ trợ bởi hệ thống cơ sở dữ liệu của bạn:

```
$users = DB::table('users')
    ->where('votes', '>=', 100)
```

```
    ->get();

$users = DB::table('users')
    ->where('votes', '<>', 100)
    ->get();

$users = DB::table('users')
    ->where('name', 'like', 'T%')
    ->get();
```

Bạn cũng có thể truyền một mảng các điều kiện cho hàm `where`. Mỗi phần tử của mảng phải là một mảng chứa ba đối số thường được truyền cho phương thức `where`:

```
$users = DB::table('users')->where([
    ['status', '=', '1'],
    ['subscribed', '<>', '1'],
])->get();
```

> **Chú ý:** PDO không hỗ trợ tên cột ràng buộc. Do đó, bạn không bao giờ được phép cho phép người dùng nhập tên cột mà truy vấn của bạn sẽ tham chiếu, bao gồm cả các cột "order by".

Mệnh đề or where

Khi xâu chuỗi cùng nhau các cuộc gọi phương thức `where` của query builder, các mệnh đề "where" sẽ được nối với nhau bằng cách sử dụng toán tử `and`. Tuy nhiên, bạn có thể sử dụng phương thức `orWhere` để nối một mệnh đề với truy vấn bằng toán tử `or`. Phương thức `orWhere` chấp nhận các đối số giống như phương thức `where`:

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

Nếu bạn cần nhóm một điều kiện "or" trong dấu ngoặc đơn, bạn có thể truyền một hàm xử lý làm đối số đầu tiên cho phương thức `orWhere`:

```php
$users = DB::table('users')
  ->where('votes', '>', 100)
  ->orWhere(function($query) {
    $query->where('name', 'Abigail')->where('votes', '>', 50);
  })
  ->get();
```

Ví dụ trên sẽ tạo ra SQL sau:

```sql
select * from users where votes > 100 or (name = 'Abigail' and votes > 50)
```

> **Chú ý:** Bạn nên luôn nhóm các cuộc gọi `orWhere` để tránh hành vi không mong muốn khi phạm vi chung được áp dụng.

Các mệnh đề JSON Where

Laravel cũng hỗ trợ truy vấn các loại cột JSON cho những cơ sở dữ liệu hỗ trợ các loại cột JSON. Hiện tại, nó có trong MySQL 5.7+, PostgreSQL, SQL Server 2016 và SQLite 3.9.0 (với phần mở rộng JSON1). Để truy vấn một cột JSON, hãy sử dụng toán tử `->`:

```php
$users = DB::table('users')
  ->where('preferences->dining->meal', 'salad')
  ->get();
```

Bạn có thể sử dụng whereJsonContains để truy vấn mảng JSON. Tính năng này không được cơ sở dữ liệu SQLite hỗ trợ:

```php
$users = DB::table('users')
  ->whereJsonContains('options->languages', 'en')
  ->get();
```

Nếu ứng dụng của bạn sử dụng cơ sở dữ liệu MySQL hoặc PostgreSQL, bạn có thể truyền một mảng giá trị cho phương thức `whereJsonContains`:

```php
$users = DB::table('users')
  ->whereJsonContains('options->languages', ['en', 'de'])
  ->get();
```

Bạn có thể sử dụng phương thức `whereJsonLength` để truy vấn các mảng JSON theo độ dài của chúng:

```php
$users = DB::table('users')
  ->whereJsonLength('options->languages', 0)
  ->get();

$users = DB::table('users')
  ->whereJsonLength('options->languages', '>', 1)
  ->get();
```

Mệnh đề where bổ sung

`whereBetween` / `orWhereBetween`

Phương thức `whereBetween` xác minh rằng giá trị của cột nằm giữa hai giá trị quy định:

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

`whereNotBetween` / `orWhereNotBetween`

Phương thức `whereNotBetween` xác minh rằng giá trị của cột nằm ngoài hai giá trị quy định:

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

`whereIn` / `whereNotIn` / `orWhereIn` / `orWhereNotIn`

Phương thức `whereIn` xác minh rằng giá trị của một cột nhất định có trong mảng giá trị đã cho:

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

Phương thức `whereNotIn` xác minh rằng giá trị của cột đã cho không được chứa trong mảng đã cho:

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

> **Chú ý:** Nếu bạn đang thêm một mảng lớn các liên kết số nguyên vào truy vấn của mình, phương thức `whereIntegerInRaw` hoặc `whereIntegerNotInRaw` có thể được sử dụng để giảm đáng kể mức sử dụng bộ nhớ của bạn.

`whereNull` / `whereNotNull` / `orWhereNull` / `orWhereNotNull`

Phương thức `whereNull` xác minh rằng giá trị của cột đã cho là `NULL`:

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

Phương thức `whereNotNull` xác minh rằng giá trị của cột không phải là `NULL`:

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

`whereDate` / `whereMonth` / `whereDay` / `whereYear` / `whereTime`

Phương thức `whereDate` có thể được sử dụng để so sánh giá trị của cột với một ngày quy định:

```php
$users = DB::table('users')
  ->whereDate('created_at', '2016-12-31')
  ->get();
```

Phương thức `whereMonth` có thể được sử dụng để so sánh giá trị của cột với một tháng cụ thể:

```php
$users = DB::table('users')
  ->whereMonth('created_at', '12')
  ->get();
```

Phương thức `whereDay` có thể được sử dụng để so sánh giá trị của cột với một ngày cụ thể trong tháng:

```php
$users = DB::table('users')
  ->whereDay('created_at', '31')
  ->get();
```

Phương thức `whereYear` có thể được sử dụng để so sánh giá trị của cột với một năm cụ thể:

```php
$users = DB::table('users')
  ->whereYear('created_at', '2016')
  ->get();
```

Phương thức `whereTime` có thể được sử dụng để so sánh giá trị của cột với một thời điểm cụ thể:

```php
$users = DB::table('users')
  ->whereTime('created_at', '=', '11:20:45')
  ->get();
```

`whereColumn` / `orWhereColumn`

Phương thức `whereColumn` có thể được sử dụng để xác minh rằng hai cột bằng nhau:

```php
$users = DB::table('users')
  ->whereColumn('first_name', 'last_name')
  ->get();
```

Bạn cũng có thể truyền một toán tử so sánh cho phương thức `whereColumn`:

```php
$users = DB::table('users')
  ->whereColumn('updated_at', '>', 'created_at')
  ->get();
```

Bạn cũng có thể truyền một mảng cột so sánh vào phương thức `whereColumn`. Các điều kiện này sẽ được kết hợp bằng cách sử dụng toán tử `and`:

```php
$users = DB::table('users')
  ->whereColumn([
    ['first_name', '=', 'last_name'],
    ['updated_at', '>', 'created_at'],
  ])
  ->get();
```

Nhóm logic

Đôi khi bạn có thể cần phải nhóm một số mệnh đề "where" trong dấu ngoặc đơn để đạt được nhóm logic mong muốn của truy vấn của bạn. Trên thực tế, bạn thường cần phải nhóm các lệnh gọi đến phương thức `orWhere` trong dấu ngoặc đơn để tránh hành vi truy vấn không mong muốn. Để thực hiện điều này, bạn có thể truyền một hàm xử lý vào phương thức `where`:

```php
$users = DB::table('users')
  ->where('name', '=', 'John')
  ->where(function ($query) {
    $query->where('votes', '>', 100)
      ->orWhere('title', '=', 'Admin');
  })
  ->get();
```

Như bạn có thể thấy, việc truyền một hàm xử lý vào phương thức `where` sẽ cho query builder biết cách bắt đầu một nhóm ràng buộc. Hàm xử lý sẽ nhận được một đối tượng của query builder mà bạn có thể sử dụng để đặt các ràng buộc nên được chứa trong nhóm dấu ngoặc đơn. Ví dụ trên sẽ tạo ra SQL sau:

```sql
select * from users where name = 'John' and (votes > 100 or title = 'Admin')
```

> **Chú ý:** Bạn nên luôn nhóm các cuộc gọi `orWhere` để tránh hành vi không mong muốn khi phạm vi chung được áp dụng.

Các mệnh đề where khác

Các mệnh đề where exists

Phương thức `whereExists` cho phép bạn viết các mệnh đề SQL "where exists". Phương thức `whereExists` chấp nhận một hàm xử lý sẽ nhận được một đối tượng query builder, cho phép bạn xác định truy vấn sẽ được đặt bên trong mệnh đề "exists":

```php
$users = DB::table('users')
  ->whereExists(function ($query) {
    $query->select(DB::raw(1))
      ->from('orders')
      ->whereColumn('orders.user_id', 'users.id');
  })
  ->get();
```

Cách build truy vấn như trên sẽ tạo ra SQL như sau:

```
select * from users
where exists (
  select 1
  from orders
  where orders.user_id = users.id
)
```

Các mệnh đề where trong truy vấn phụ

Đôi khi bạn có thể cần phải xây dựng một mệnh đề "where" để so sánh kết quả của một truy vấn con với một giá trị nào đó. Bạn có thể thực hiện điều này bằng cách truyền một hàm xử lý và một giá trị cho phương thức where. Ví dụ: truy vấn sau sẽ truy xuất tất cả người dùng có "tư cách thành viên" (membership) gần đây của một loại nào đó;

```
use App\Models\User;

$users = User::where(function ($query) {
  $query->select('type')
    ->from('membership')
    ->whereColumn('membership.user_id', 'users.id')
    ->orderByDesc('membership.start_date')
    ->limit(1);
}, 'Pro')->get();
```

Hoặc, bạn có thể cần phải xây dựng một mệnh đề "where" nhằm so sánh một cột với kết quả của một truy vấn phụ. Bạn có thể thực hiện điều này bằng cách truyền một cột, toán tử và hàm xử lý vào phương thức `where`. Ví dụ: truy vấn sau sẽ truy xuất tất cả các bản ghi thu nhập trong đó số tiền nhỏ hơn mức trung bình;

```
use App\Models\Income;

$incomes = Income::where('amount', '<', function ($query) {
  $query->selectRaw('avg(i.amount)')->from('incomes as i');
})->get();
```

Các lệnh order, group, limit, và offset

Lệnh Order

Phương thức `orderBy`

Phương thức `orderBy` cho phép bạn sắp xếp các kết quả của truy vấn theo một cột nhất định. Đối số đầu tiên được phương thức `orderBy` chấp nhận phải là cột bạn muốn sắp xếp, trong khi đối số thứ hai xác định hướng sắp xếp và có thể là `asc` hoặc `desc`:

```
$users = DB::table('users')
  ->orderBy('name', 'desc')
  ->get();
```

Để sắp xếp theo nhiều cột, bạn có thể chỉ cần gọi `orderBy` nhiều lần nếu cần:

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

Các phương thức `latest` và `oldest`

Các phương thức latest và oldest cho phép bạn dễ dàng sắp xếp kết quả theo ngày. Theo mặc định, kết quả sẽ được sắp xếp theo cột `created_at` của bảng. Hoặc, bạn có thể chuyển tên cột mà bạn muốn sắp xếp theo:

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

Xếp thứ tự mặc định

Phương thức `inRandomOrder` có thể được sử dụng để sắp xếp các kết quả truy vấn một cách ngẫu nhiên. Ví dụ: bạn có thể sử dụng phương thức này để tải một người dùng ngẫu

nhiên:

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

Loại bỏ các thứ tự hiện có

Phương thức `reorder` loại bỏ tất cả các mệnh đề "order by" đã được áp dụng trước đó cho truy vấn:

```
$query = DB::table('users')->orderBy('name');

$unorderedUsers = $query->reorder()->get();
```

Bạn có thể truyền một cột và hướng sắp xếp khi gọi phương thức reorder lại để loại bỏ tất cả các mệnh đề "order by" hiện có và áp dụng một thứ tự hoàn toàn mới cho truy vấn:

```
$query = DB::table('users')->orderBy('name');

$usersOrderedByEmail = $query->reorder('email', 'desc')->get();
```

Nhóm

Phương thức `groupBy` và `having`

Như bạn có thể mong đợi, các phương thức `groupBy` và `having` có thể được sử dụng để nhóm các kết quả truy vấn. Hình thức của phương thức `having` tương tự như phương thức `where`:

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

Bạn có thể sử dụng phương thức `havingBetween` để lọc kết quả trong một khoảng giá trị nào đó:

```
$report = DB::table('orders')
  ->selectRaw('count(id) as number_of_orders, customer_id')
  ->groupBy('customer_id')
  ->havingBetween('number_of_orders', [5, 15])
  ->get();
```

Bạn có thể truyền nhiều đối số cho phương thức groupBy cách nhóm theo nhiều cột:

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

Để xây dựng các câu lệnh `having` nâng cao hơn, hãy xem phương thức `havingRaw`.

Lệnh Limit và Offset

Các phương thức `skip` và `take`

Bạn có thể sử dụng phương thức `skip` và `take` để giới hạn số lượng kết quả trả về từ truy vấn hoặc để bỏ qua một số kết quả nhất định trong truy vấn:

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

Ngoài ra, bạn có thể sử dụng các phương thức `limit` và `offset.` Các phương thức này tương đương về mặt chức năng với các phương thức `take` và `skip`:

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

Các mệnh đề điều kiện

Đôi khi bạn có thể muốn các mệnh đề truy vấn nào đó áp dụng cho một truy vấn dựa trên một điều kiện khác. Ví dụ: bạn có thể chỉ muốn áp dụng câu lệnh `where` nếu một giá trị đầu vào có trong HTTP request được gửi đến. Bạn có thể thực hiện điều này bằng cách sử dụng phương thức `when`:

```
$role = $request->input('role');

$users = DB::table('users')
  ->when($role, function ($query, $role) {
    return $query->where('role_id', $role);
  })
  ->get();
```

Phương thức `when` chỉ thực hiện một hàm xử lý đã cho khi đối số đầu tiên là `true`. Nếu đối số đầu tiên là `false`, thì hàm xử lý sẽ không được thực hiện. Vì vậy, trong ví dụ trên, hàm xử lý được cung cấp cho phương thức `when` sẽ chỉ được gọi nếu trường `role` hiện diện trong yêu cầu gửi đến và đánh giá là `true`.

Bạn có thể truyền một hàm xử lý khác làm đối số thứ ba cho phương thức `when`. Hàm xử lý này sẽ chỉ thực thi nếu đối số đầu tiên đánh giá là `false`. Để minh họa cách sử dụng tính năng này, chúng tôi sẽ sử dụng nó để cấu hình thứ tự mặc định của một truy vấn:

```
$sortByVotes = $request->input('sort_by_votes');

$users = DB::table('users')
  ->when($sortByVotes, function ($query, $sortByVotes) {
    return $query->orderBy('votes');
  }, function ($query) {
    return $query->orderBy('name');
  })
  ->get();
```

Các lệnh insert

Query builder cũng cung cấp một phương thức `insert` có thể được sử dụng để chèn các

record vào bảng cơ sở dữ liệu. Phương thức `insert` chấp nhận một mảng tên cột và giá trị của nó:

```
DB::table('users')->insert([
  'email' => 'kayla@example.com',
  'votes' => 0
]);
```

Bạn có thể chèn nhiều record cùng một lúc bằng cách truyền một mảng liệt kê các mảng. Mỗi mảng đại diện cho một record cần được chèn vào bảng:

```
DB::table('users')->insert([
  ['email' => 'picard@example.com', 'votes' => 0],
  ['email' => 'janeway@example.com', 'votes' => 0],
]);
```

Phương thức `insertOrIgnore` sẽ bỏ qua lỗi khi chèn record vào cơ sở dữ liệu:

```
DB::table('users')->insertOrIgnore([
  ['id' => 1, 'email' => 'sisko@example.com'],
  ['id' => 2, 'email' => 'archer@example.com'],
]);
```

> **Chú ý:** `insertOrIgnore` sẽ bỏ qua các bản ghi trùng lặp và cũng có thể bỏ qua các loại lỗi khác tùy thuộc vào công cụ cơ sở dữ liệu. Ví dụ: `insertOrIgnore` sẽ bỏ qua chế độ thắt chặt của MySQL.

Các ID tự động đếm số

Nếu bảng có `id` tự động đếm số, hãy sử dụng phương thức `insertGetId` để chèn record và sau đó truy xuất ID của record vừa chèn:

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

> **Chú ý:** Khi sử dụng PostgreSQL, phương thức `insertGetId` yêu cầu cột tự động tăng dần được đặt tên là id. Nếu bạn muốn truy xuất ID từ một "sequence" khác, bạn có thể truyền tên cột làm tham số thứ hai cho phương thức `insertGetId`.

Upserts

Phương thức `upsert` sẽ chèn các record không tồn tại và cập nhật các bản ghi đã tồn tại với các giá trị mới mà bạn có thể chỉ định. Đối số đầu tiên của phương thức bao gồm các giá trị để chèn hoặc cập nhật, trong khi đối số thứ hai liệt kê (các) cột dùng để nhận dạng các record trong bảng được liên kết. Đối số thứ ba và cuối cùng của phương thức là một mảng cột cần được cập nhật nếu bản ghi phù hợp đã tồn tại trong cơ sở dữ liệu:

```
DB::table('flights')->upsert([
    ['departure' => 'Oakland', 'destination' => 'San Diego', 'price' => 99],
    ['departure' => 'Chicago', 'destination' => 'New York', 'price' => 150]
], ['departure', 'destination'], ['price']);
```

Trong ví dụ trên, Laravel sẽ cố gắng chèn hai record. Nếu một record đã tồn tại với cùng giá trị cột khởi hành và cột đích, Laravel sẽ cập nhật cột giá của record đó.

> **Chú ý:** Tất cả các cơ sở dữ liệu ngoại trừ SQL Server yêu cầu các cột trong đối số thứ hai của phương thức `upsert` phải có chỉ mục "chính" hoặc "duy nhất". Ngoài ra, driver cơ sở dữ liệu MySQL bỏ qua đối số thứ hai của phương thức `upsert` và luôn sử dụng các chỉ mục "primary" và "unique" của bảng để phát hiện các bản ghi hiện có.

Các lệnh Update

Ngoài việc chèn các record vào cơ sở dữ liệu, query builder cũng có thể cập nhật các record hiện có bằng phương thức `update`. Phương thức `update`, giống như phương thức `insert`, chấp nhận một mảng các cặp cột và giá trị cho biết các cột sẽ được cập nhật. Phương thức `update` trả về số lượng hàng bị ảnh hưởng. Bạn có thể hạn chế truy vấn cập nhật bằng mệnh đề `where`:

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

Lệnh Update hay Insert

Đôi khi bạn có thể muốn cập nhật một record hiện có trong cơ sở dữ liệu hoặc tạo nó nếu không có record phù hợp nào tồn tại. Trong trường hợp này, phương thức `updateOrInsert` có thể được sử dụng. Phương thức `updateOrInsert` chấp nhận hai đối số: một mảng các điều kiện để tìm record và một mảng các cặp giá trị và cột cho biết các cột sẽ được cập nhật.

Phương thức `updateOrInsert` sẽ cố gắng xác định một bản ghi cơ sở dữ liệu phù hợp bằng cách sử dụng các cặp giá trị và cột của đối số đầu tiên. Nếu bản ghi tồn tại, nó sẽ được cập nhật với các giá trị trong đối số thứ hai. Nếu không thể tìm thấy record, record mới sẽ được chèn với các thuộc tính đã hợp nhất của cả hai đối số:

```php
DB::table('users')
  ->updateOrInsert(
    ['email' => 'john@example.com', 'name' => 'John'],
    ['votes' => '2']
  );
```

Cập nhật các cột JSON

Khi cập nhật một cột JSON, bạn nên sử dụng dấu `->` trong cú pháp để cập nhật khóa thích hợp trong đối tượng JSON. Thao tác này được hỗ trợ trên MySQL 5.7+ và PostgreSQL 9.5+:

```php
$affected = DB::table('users')
  ->where('id', 1)
  ->update(['options->enabled' => true]);
```

Tăng dần và giảm dần

Query builder cũng cung cấp các phương thức thuận tiện để tăng hoặc giảm giá trị của một

cột nhất định. Cả hai phương thức này đều chấp nhận ít nhất một đối số: cột cần sửa đổi. Đối số thứ hai có thể được cung cấp để chỉ định số lượng cột sẽ được tăng hoặc giảm:

```
DB::table('users')->increment('votes');
DB::table('users')->increment('votes', 5);
DB::table('users')->decrement('votes');
DB::table('users')->decrement('votes', 5);
```

Bạn cũng có thể chỉ định các cột bổ sung để cập nhật trong quá trình hoạt động:

```
DB::table('users')->increment('votes', 1, ['name' => 'John']);
```

Các lệnh delete

Phương thức `delete` của query builder có thể được sử dụng để xóa các record khỏi bảng. Phương thức `delete` trả về số hàng bị ảnh hưởng. Bạn có thể hạn chế các câu lệnh xóa bằng cách thêm mệnh đề "where" trước khi gọi phương thức `delete`:

```
$deleted = DB::table('users')->delete();

$deleted = DB::table('users')->where('votes', '>', 100)->delete();
```

Nếu bạn sử dụng lệnh *truncate* cho toàn bộ bảng, thì nó sẽ xóa tất cả các record khỏi bảng và đặt lại ID tự động tăng dần về 0, bạn có thể sử dụng phương thức `truncate`:

```
DB::table('users')->truncate();
```

Cắt bỏ bảng và PostgreSQL

Khi cắt bỏ cơ sở dữ liệu PostgreSQL, hành vi `CASCADE` sẽ được áp dụng. Điều này có nghĩa là tất cả các bản ghi liên quan đến khóa ngoại trong các bảng khác cũng sẽ bị xóa.

Khóa Pessimistic (Pe-si-met)

Query builder cũng bao gồm một vài chức năng để giúp bạn đạt được "khóa pe-si-met" khi

thực hiện các câu lệnh select của mình. Để thực hiện một câu lệnh với "khóa shared", bạn có thể gọi phương thức `sharedLock`. Khóa như thế này dùng để ngăn các hàng đã chọn không được sửa đổi cho đến khi giao dịch transaction của bạn được chốt lệnh:

```
DB::table('users')
  ->where('votes', '>', 100)
  ->sharedLock()
  ->get();
```

Ngoài ra, bạn có thể sử dụng phương thức `lockForUpdate`. Khóa để cập nhật sẽ ngăn không cho sửa đổi các record đã chọn hoặc không được chọn bằng một khóa được chia sẻ khác:

```
DB::table('users')
  ->where('votes', '>', 100)
  ->lockForUpdate()
  ->get();
```

Gỡ rối (Debugging)

Bạn có thể sử dụng phương thức `dd` và phương thức `dump` trong khi xây dựng truy vấn để kết xuất các ràng buộc truy vấn hiện tại và SQL. Phương thức `dd` sẽ hiển thị thông tin gỡ lỗi và sau đó ngừng thực hiện yêu cầu. Phương thức `dump` sẽ hiển thị thông tin gỡ lỗi nhưng cho phép yêu cầu tiếp tục thực hiện:

```
DB::table('users')->where('votes', '>', 100)->dd();

DB::table('users')->where('votes', '>', 100)->dump();
```