Course - Laravel Framework

HTTP Client

Laravel cung cấp một API tối thiểu, rõ ràng và minh bạch xung quanh ứng dụng khách Guzzle HTTP, cho phép bạn nhanh chóng thực hiện các HTTP request gửi đi để giao tiếp với các ứng dụng web khác.

Tags: laravel http client, laravel

Giới thiệu

Laravel cung cấp một API tối thiểu, rõ ràng và minh bạch xung quanh ứng dụng khách Guzzle HTTP, cho phép bạn nhanh chóng thực hiện các HTTP request gửi đi để giao tiếp với các ứng dụng web khác. Wrapper của Laravel bao quanh Guzzle tập trung vào các trường hợp sử dụng phổ biến nhất của nó và trải nghiệm tuyệt vời dành cho developer.

Trước khi bắt đầu, bạn nên đảm bảo rằng bạn đã cài đặt package Guzzle như một phần mềm phụ thuộc (dependence) vào ứng dụng của bạn. Mặc định, Laravel tự động đưa vào các phần mềm phụ thuộc này. Tuy nhiên, nếu trước đó bạn đã gỡ bỏ package, thì bạn có thể cài đặt lại package này với Composer:

```
composer require guzzlehttp/guzzle
```

Yêu cầu

Để thực hiện các request, bạn có thể sử dụng các phương thức `head`, và `get`, và được cung cấp bởi facade. Trước tiên, hãy kiểm tra cách thực hiện một request cơ bản đối với một URL khác:`post` `put` `patch` `delete` `Http` `GET`

```
use Illuminate\Support\Facades\Http;

$response = Http::get('http://example.com');
```

Phương thức `get` này trả về một đối tượng của `Illuminate\Http\Client\Response`, cung cấp nhiều phương thức có thể được sử dụng để kiểm tra response:

```
$response->body() : string;
$response->json($key = null) : array|mixed;
$response->object() : object;
$response->collect($key = null) : Illuminate\Support\Collection;
$response->status() : int;
$response->ok() : bool;
$response->successful() : bool;
$response->redirect(): bool;
$response->failed() : bool;
$response->serverError() : bool;
```

```
$response->clientError() : bool;
$response->header($header) : string;
$response->headers() : array;
```

Đối tượng `Illuminate\Http\Client\Response` cũng được thực thi với giao diện PHP `ArrayAccess`, cho phép bạn truy cập dữ liệu JSON response trực tiếp trên response kết quả:

```
return Http::get('http://example.com/users/1')['name'];
```

Dumping yêu cầu request

Nếu bạn muốn render đối tượng request gửi đi trước khi nó được gửi đi và chấm dứt việc thực thi tập lệnh, thì bạn có thể thêm phương thức `dd` vào đầu khai báo request của mình:

```
return Http::dd()->get('http://example.com');
```

Dữ liệu của request

Tất nhiên, Các request vẫn phổ biến khi thực hiện với method `POST` và `PUT`, `PATCH` sẽ gửi dữ liệu bổ sung vào request của bạn, vì vậy các phương thức này chấp nhận một mảng dữ liệu làm đối số thứ hai của chúng. Theo mặc định, dữ liệu sẽ được gửi bằng loại nội dung `application/json`:

```
use Illuminate\Support\Facades\Http;

$response = Http::post('http://example.com/users', [
  'name' => 'Steve',
  'role' => 'Network Administrator',
]);
```

Tham số truy vấn của request

Khi đưa ra `GET` request, bạn có thể nối trực tiếp một chuỗi truy vấn vào URL hoặc truyền một mảng các cặp khóa/giá trị làm đối số thứ hai cho phương thức `get`:

```php
$response = Http::get('http://example.com/users', [
  'name' => 'Taylor',
  'page' => 1,
]);
```

Gửi request mã hóa URL form

Nếu bạn muốn gửi dữ liệu bằng loại nội dung `application/x-www-form-urlencoded`, bạn nên gọi phương thức `asForm` trước khi thực hiện yêu cầu của mình:

```php
$response = Http::asForm()->post('http://example.com/users', [
  'name' => 'Sara',
  'role' => 'Privacy Consultant',
]);
```

Gửi request nội dung raw

Bạn có thể sử dụng phương thức `withBody` này nếu bạn muốn cung cấp nội dung thô khi đưa ra request. Loại nội dung (content-type) có thể được cung cấp thông qua đối số thứ hai của phương thức:

```php
$response = Http::withBody(
  base64_encode($photo), 'image/jpeg'
)->post('http://example.com/photo');
```

Gửi request multi-part

Nếu bạn muốn gửi tập tin dưới dạng multi-part, thì bạn nên gọi phương thức `attach` trước khi thực hiện request của mình. Phương thức này chấp nhận tên của tập tin và nội dung của tập tin. Nếu cần, bạn có thể cung cấp đối số thứ ba sẽ được coi là tên của tập tin:

```php
$response = Http::attach(
  'attachment', file_get_contents('photo.jpg'), 'photo.jpg'
)->post('http://example.com/attachments');
```

Thay vì truyền nội dung thô của tập tin, thì bạn có thể truyền tài nguyên luồng của nó như sau:

```
$photo = fopen('photo.jpg', 'r');

$response = Http::attach(
  'attachment', $photo, 'photo.jpg'
)->post('http://example.com/attachments');
```

Tiêu đề Header

Tiêu đề header có thể được thêm vào các request bằng phương thức withHeaders. Phương thức withHeaders chấp nhận một mảng các cặp khóa/giá trị:

```
$response = Http::withHeaders([
  'X-First' => 'foo',
  'X-Second' => 'bar'
])->post('http://example.com/users', [
  'name' => 'Taylor',
]);
```

Bạn có thể sử dụng phương thức `accept` để chỉ định loại nội dung mà ứng dụng của bạn đang mong đợi để đáp ứng yêu cầu của bạn:

```
$response = Http::accept('application/json')->get('http://example.com/users');
```

Để thuận tiện, bạn có thể sử dụng phương thức `acceptJson` để nhanh chóng chỉ định ứng dụng của bạn mong đợi loại nội dung `application/json` để đáp ứng yêu cầu của bạn:

```
$response = Http::acceptJson()->get('http://example.com/users');
```

Xác thực

Bạn có thể chỉ định thông tin xác thực cơ bản và thông báo xác thực bằng cách sử dụng các phương thức `withBasicAuth` và `withDigestAuth`, như sau:

```php
// Basic authentication...
$response = Http::withBasicAuth('taylor@laravel.com', 'secret')->post(...);

// Digest authentication...
$response = Http::withDigestAuth('taylor@laravel.com', 'secret')->post(...);
```

Token bearer

Nếu bạn muốn nhanh chóng thêm *bearer* token vào tiêu đề header `Authorization` của request, thì bạn có thể sử dụng phương thức `withToken`:

```php
$response = Http::withToken('token')->post(...);
```

Timeout của request

Phương thức `timeout` có thể được sử dụng để xác định số giây tối đa để chờ phản hồi:

```php
$response = Http::timeout(3)->get(...);
```

Nếu thời gian chờ đã cho bị vượt quá thời hạn đã cho, một đối tượng của `Illuminate\Http\Client\ConnectionException` sẽ được ném ra.

Kết nối lại request

Nếu bạn muốn ứng dụng khách HTTP tự động thử lại request nếu xảy ra lỗi máy khách hoặc máy chủ, bạn có thể sử dụng phương thức `retry`. Phương thức `retry` chấp nhận số lần yêu cầu tối đa sẽ được thử và số mili giây mà Laravel phải đợi giữa các lần thử:

```php
$response = Http::retry(3, 100)->post(...);
```

Nếu cần, bạn có thể truyền đối số thứ ba cho phương thức `retry`. Đối số thứ ba phải là một đối số có thể gọi được để xác định xem có thực sự nên thử lại hay không. Ví dụ: bạn có thể chỉ muốn thử lại request nếu request ban đầu gặp phải trường hợp `ConnectionException`:

```php
$response = Http::retry(3, 100, function ($exception) {
  return $exception instanceof ConnectionException;
})->post(...);
```

Nếu tất cả các yêu cầu không thành công, một đối tượng của `Illuminate\Http\Client\RequestException` sẽ được ném ra.

Xử lý lỗi

Không giống như hành vi mặc định của Guzzle, warpper HTTP client của Laravel không đưa ra các ngoại lệ đối với lỗi máy khách hoặc máy chủ (phản hồi cấp độ 400 và 500 từ máy chủ). Bạn có thể xác định xem một trong những lỗi này có được trả về hay không bằng cách sử dụng các phương thức `successful`, `clientError` hoặc `serverError`:

```php
// Determine if the status code is >= 200 and < 300...
$response->successful();

// Determine if the status code is >= 400...
$response->failed();

// Determine if the response has a 400 level status code...
$response->clientError();

// Determine if the response has a 500 level status code...
$response->serverError();

// Immediately execute the given callback if there was a client or server error...
$response->onError(callable $callback);
```

Đưa ra Exception

Nếu bạn có một trường hợp response và muốn đưa ra một exception `Illuminate\Http\Client\RequestException` nếu mã trạng thái của response chỉ ra lỗi máy khách hoặc máy chủ, thì bạn có thể sử dụng các phương thức `throw` hoặc `throwIf`:

```php
$response = Http::post(...);
```

```php
// Throw an exception if a client or server error occurred...
$response->throw();

// Throw an exception if an error occurred and the given condition is true...
$response->throwIf($condition);

return $response['user']['id'];
```

Đối tượng `Illuminate\Http\Client\RequestException` có thuộc tính public `$response` sẽ cho phép bạn kiểm tra response kết quả được trả về.

Phương thức `throw` sẽ trả về trường hợp phản hồi nếu không có lỗi xảy ra, cho phép bạn xâu chuỗi các thao tác khác vào phương thức `throw`:

```php
return Http::post(...)->throw()->json();
```

Nếu bạn muốn thực hiện một số logic bổ sung trước khi exception được ném ra, bạn có thể truyền một hàm cho phương thức `throw`. Exception sẽ được ném tự động sau khi hàm này được thực thi xong, vì vậy bạn không cần phải ném lại exception từ bên trong hàm này:

```php
return Http::post(...)->throw(function ($response, $e) {
  //
})->json();
```

Các tùy chọn Guzzle

Bạn có thể chỉ định các tùy chọn request Guzzle bổ sung bằng cách sử dụng phương thức withOptions. Phương thức withOptions chấp nhận một mảng các cặp khóa/giá trị:

```php
$response = Http::withOptions([
  'debug' => true,
])->get('http://example.com/users');
```

Các request đồng thời

Đôi khi, bạn có thể muốn thực hiện nhiều request HTTP đồng thời. Nói cách khác, bạn muốn gửi một số request cùng một lúc thay vì đưa ra các request một cách tuần tự. Điều này có thể dẫn đến cải thiện hiệu suất đáng kể khi tương tác với các API HTTP chậm.

Rất may, bạn có thể thực hiện điều này bằng cách sử dụng phương thức `pool`. Phương thức `pool` chấp nhận một hàm mà sẽ nhận được một đối tượng `Illuminate\Http\Client\Pool`, cho phép bạn dễ dàng thêm các request vào nhóm request để gửi đi:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
  $pool->get('http://localhost/first'),
  $pool->get('http://localhost/second'),
  $pool->get('http://localhost/third'),
]);

return $responses[0]->ok() &&
  $responses[1]->ok() &&
  $responses[2]->ok();
```

Như bạn có thể thấy, mỗi đối tượng response có thể được truy cập dựa trên thứ tự nó được thêm vào nhóm. Nếu muốn, bạn có thể đặt tên cho các request bằng cách sử dụng phương thức as, cho phép bạn truy cập các response tương ứng theo tên:

```
use Illuminate\Http\Client\Pool;
use Illuminate\Support\Facades\Http;

$responses = Http::pool(fn (Pool $pool) => [
  $pool->as('first')->get('http://localhost/first'),
  $pool->as('second')->get('http://localhost/second'),
  $pool->as('third')->get('http://localhost/third'),
]);

return $responses['first']->ok();
```

Macro

Ứng dụng khách Laravel HTTP client cho phép bạn xác định *"macro"*, có thể đóng vai trò như một cơ chế cấu hình tiêu đề và đường dẫn yêu cầu chung khi tương tác với các service trong toàn bộ ứng dụng của bạn. Để bắt đầu, bạn có thể khai báo *macro* trong phương thức `boot` của class `App\Providers\AppServiceProvider` trong ứng dụng của bạn:

```
use Illuminate\Support\Facades\Http;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  Http::macro('github', function () {
    return Http::withHeaders([
      'X-Example' => 'example',
    ])->baseUrl('https://github.com');
  });
}
```

Khi *macro* của bạn đã được cấu hình, thì bạn có thể gọi nó từ bất kỳ đâu trong ứng dụng của mình để tạo một request đang chờ xử lý với cấu hình được cài đặt:

```
$response = Http::github()->get('/');
```

Testing kiểm nghiệm

Nhiều dịch vụ của Laravel cung cấp chức năng giúp bạn viết các bài kiểm tra một cách dễ dàng và rõ ràng, và wrapper HTTP của Laravel cũng không phải là ngoại lệ. Phương thức `fake` của facade `Http` cho phép bạn hướng dẫn máy khách HTTP client trả về các phản hồi sơ khai/giả khi request được thực hiện.

Giả tạo phản hồi

Ví dụ: để cho máy khách HTTP client biết sẽ trả về các response với mã trạng thái 200, áp dụng cho mọi request thì chúng ta cho trống, bạn có thể gọi phương thức `fake` mà không có đối số:

```
use Illuminate\Support\Facades\Http;

Http::fake();

$response = Http::post(...);
```

> **Chú ý:** Khi giả tạo request, phần mềm trung gian của máy khách HTTP client không được thực thi. Bạn nên xác định các kỳ vọng cho các phản hồi giả tạo như thể các phần mềm trung gian này đã chạy chính xác.

Giả tạo URL cụ thể nào đó

Ngoài ra, bạn có thể truyền một mảng cho phương thức `fake`. Các khóa của mảng phải đại diện cho các mẫu URL mà bạn muốn giả tạo và các phản hồi liên quan của chúng. Ký tự *wildcard* (*) có thể được sử dụng như một ký tự đại diện tất cả. Bất kỳ yêu cầu nào được thực hiện đối với các URL không bị giả tạo sẽ thực sự được thực thi. Bạn có thể sử dụng phương thức `response` của facade `Http` để tạo các phản hồi sơ khai/giả tạo cho các điểm cuối này:

```
Http::fake([
  // Stub a JSON response for GitHub endpoints...
  'github.com/*' => Http::response(['foo' => 'bar'], 200, $headers),

  // Stub a string response for Google endpoints...
  'google.com/*' => Http::response('Hello World', 200, $headers),
]);
```

Nếu bạn muốn xác định một mẫu URL dự phòng sẽ tạo ra tất cả các URL, bạn có thể sử dụng ký tự *wildcard* (*):

```
Http::fake([
  // Stub a JSON response for GitHub endpoints...
```

```
    'github.com/*' => Http::response(['foo' => 'bar'], 200, ['Headers']),

    // Stub a string response for all other endpoints...
    '*' => Http::response('Hello World', 200, ['Headers']),
]);
```

Sắp xếp phản hồi giả tạo

Đôi khi bạn có thể cần xác định một URL duy nhất sẽ trả về một loạt phản hồi giả theo một thứ tự cụ thể. Bạn có thể thực hiện điều này bằng cách sử dụng phương thức `Http::sequence` để tạo phản hồi:

```
Http::fake([
    // Stub a series of responses for GitHub endpoints...
    'github.com/*' => Http::sequence()
                        ->push('Hello World', 200)
                        ->push(['foo' => 'bar'], 200)
                        ->pushStatus(404),
]);
```

Khi tất cả các phản hồi trong chuỗi phản hồi đã được sử dụng, bất kỳ yêu cầu nào khác cũng sẽ khiến chuỗi phản hồi đưa ra một exception. Nếu bạn muốn chỉ định một phản hồi mặc định sẽ được trả lại khi một chuỗi trống, bạn có thể sử dụng phương pháp `whenEmpty`:

```
Http::fake([
    // Stub a series of responses for GitHub endpoints...
    'github.com/*' => Http::sequence()
                        ->push('Hello World', 200)
                        ->push(['foo' => 'bar'], 200)
                        ->whenEmpty(Http::response()),
]);
```

Nếu bạn muốn giả tạo một chuỗi phản hồi nhưng không muốn xác định mẫu URL cụ thể sẽ được làm giả, thì bạn có thể sử dụng phương thức `Http::fakeSequence`:

```
Http::fakeSequence()
  ->push('Hello World', 200)
  ->whenEmpty(Http::response());
```

Callback giả

Nếu bạn yêu cầu logic phức tạp hơn để xác định phản hồi nào sẽ trả về cho các điểm cuối cụ thể, thì bạn có thể truyền một hàm callback cho phương thức `fake`. Hàm callback này sẽ nhận một đối tượng request của `Illuminate\Http\Client\Request` và sẽ trả về một đối tượng response. Trong hàm callback, bạn có thể thực hiện bất kỳ logic nào cần thiết để xác định loại phản hồi nào cần trả về:

```
Http::fake(function ($request) {
  return Http::response('Hello World', 200);
});
```

Kiểm tra request

Khi giả tạo phản hồi, đôi khi bạn có thể muốn kiểm tra các request mà client nhận được để đảm bảo rằng ứng dụng của bạn đang gửi dữ liệu hoặc tiêu đề chính xác. Bạn có thể thực hiện điều này bằng cách gọi phương thức `Http::assertSent` sau khi gọi phương thức `Http::fake`.

Phương thức chấp `assertSent` nhận một hàm nặc danh mà sẽ nhận một đối tượng request của `Illuminate\Http\Client\Request` và sẽ trả về một giá trị *boolean* cho biết liệu request có phù hợp với mong đợi của bạn hay không. Để bài kiểm tra vượt qua, ít nhất một yêu cầu phải được đưa ra phù hợp với các kỳ vọng đã cho:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::withHeaders([
  'X-First' => 'foo',
])->post('http://example.com/users', [
```

```
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertSent(function (Request $request) {
    return $request->hasHeader('X-First', 'foo') &&
        $request->url() == 'http://example.com/users' &&
        $request['name'] == 'Taylor' &&
        $request['role'] == 'Developer';
});
```

Nếu cần, bạn có thể khẳng định rằng một yêu cầu cụ thể đã không được gửi bằng phương thức `assertNotSent`:

```
use Illuminate\Http\Client\Request;
use Illuminate\Support\Facades\Http;

Http::fake();

Http::post('http://example.com/users', [
    'name' => 'Taylor',
    'role' => 'Developer',
]);

Http::assertNotSent(function (Request $request) {
    return $request->url() === 'http://example.com/posts';
});
```

Bạn có thể sử dụng phương thức `assertSentCount` để xác nhận có bao nhiêu yêu cầu đã được "gửi" trong quá trình kiểm tra:

```
Http::fake();

Http::assertSentCount(5);
```

Hoặc, bạn có thể sử dụng phương thức `assertNothingSent` để khẳng định rằng không

có yêu cầu nào được gửi trong quá trình kiểm tra:

```
Http::fake();

Http::assertNothingSent();
```

Event

Laravel kích hoạt ba event trong quá trình gửi yêu cầu HTTP request. Event tên là `RequestSending` sẽ được kích hoạt trước khi một request được gửi đi, trong khi một event khác là `ResponseReceived` sẽ được kích hoạt sau khi nhận được phản hồi cho một request nào đó. Event thứ ba là `ConnectionFailed` sẽ được kích hoạt nếu không nhận được phản hồi cho một request nào đó.

Cả hai event `RequestSending` và `ConnectionFailed` đều chứa một thuộc tính public `$request` mà bạn có thể sử dụng để kiểm tra đối tượng request của `Illuminate\Http\Client\Request`. Tương tự như vậy, event `ResponseReceived` cũng chứa một thuộc tính `$request` cũng như một thuộc tính `$response` có thể được sử dụng để kiểm tra đối tượng response của `Illuminate\Http\Client\Response`. Bạn có thể đăng ký chương trình theo dõi event cho các event này trong service provider `App\Providers\EventServiceProvider` của bạn:

```
/**
 * The event listener mappings for the application.
 *
 * @var array
 */
protected $listen = [
  'Illuminate\Http\Client\Events\RequestSending' => [
    'App\Listeners\LogRequestSending',
  ],
  'Illuminate\Http\Client\Events\ResponseReceived' => [
    'App\Listeners\LogResponseReceived',
  ],
  'Illuminate\Http\Client\Events\ConnectionFailed' => [
    'App\Listeners\LogConnectionFailed',
  ],
```

```
];
```