Course - Laravel Framework

Blade template

Blade là công cụ tạo template đơn giản nhưng mạnh mẽ đã được đưa vào trong Laravel. Không giống như một số công cụ tạo template PHP, Blade không hạn chế bạn sử dụng mã PHP thuần trong các template của bạn.

Tags: Blade template, laravel

Giới thiệu

Trên thực tế, tất cả các template Blade đều được biên dịch thành mã PHP thuần túy và được lưu cache cho đến khi chúng được sửa đổi mới, có nghĩa là Blade bổ sung thêm chi phí cơ bản bằng không vào ứng dụng của bạn. Các tập tin template Blade sử dụng có đuôi mở rộng là *.blade.php* và thường được lưu trữ trong thư mục *resources/view*.

Các template Blade có thể được trả về từ các route hoặc contorller bằng cách sử dụng hàm toàn cục `view`. Tất nhiên, chúng ta sẽ đề cập trong tài liệu về *view* sắp tới, dữ liệu có thể được chuyển đến template Blade bằng cách sử dụng đối số thứ hai của hàm `view`:

```
Route::get('/', function () {
  return view('greeting', ['name' => 'Finn']);
});
```

> Bạn muốn nâng các template Blade của mình lên một tầm cao mới và dễ dàng xây dựng các giao diện động? Kiểm tra Laravel Livewire.

Hiển thị dữ liệu

Bạn có thể hiển thị dữ liệu đã được truyền đến các view Blade của mình bằng cách đặt biến trong dấu ngoặc nhọn. Ví dụ, với route sau:

```
Route::get('/', function () {
  return view('welcome', ['name' => 'Samantha']);
});
```

Bạn có thể hiển thị nội dung của biến `name` như sau:

```
Hello, {{ $name }}.
```

> Các câu lệnh echo `{{}}` của Blade được gửi tự động thông qua hàm `htmlspecialchars` của PHP để ngăn chặn các cuộc tấn công XSS.

Bạn không bị giới hạn trong việc hiển thị nội dung của các biến được truyền đến template. Bạn cũng có thể lặp lại kết quả của bất kỳ hàm PHP nào. Trên thực tế, bạn có thể đặt bất kỳ

mã PHP nào bạn muốn bên trong câu lệnh Blade echo:

```
The current UNIX timestamp is {{ time() }}.
```

Mã hoá HTML entity

Mặc định, Blade (và hàm trợ giúp Laravel `e`) sẽ mã hóa kép các HTML entity. Nếu bạn muốn tắt mã hóa kép, thì hãy gọi phương thức `Blade::withoutDoubleEncoding` từ phương thức `boot` của `AppServiceProvider` của bạn:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
  /**
   * Bootstrap any application services.
   *
   * @return void
   */
  public function boot()
  {
    Blade::withoutDoubleEncoding();
  }
}
```

Hiển thị dữ liệu unescaped

Mặc định, các câu lệnh Blade `{{}}` được tự động gửi qua hàm `htmlspecialchars` của PHP để ngăn chặn các cuộc tấn công XSS. Nếu bạn không muốn dữ liệu của mình bị escape, bạn có thể sử dụng cú pháp sau:

```
Hello, {!! $name !!}.
```

> **Chú ý:** Hãy hết sức cẩn thận khi lặp lại nội dung do người dùng ứng dụng của bạn cung cấp. Thông thường, bạn nên sử dụng cú pháp dấu ngoặc nhọn kép có mã thoát để ngăn chặn các cuộc tấn công XSS khi hiển thị dữ liệu do người dùng cung cấp.

Các framework js và blade

Vì nhiều framework JavaScript cũng sử dụng dấu ngoặc nhọn "curly" để cho biết một biểu thức nhất định sẽ được hiển thị trong trình duyệt, bạn có thể sử dụng ký hiệu `@` để thông báo cho công cụ kết xuất Blade rằng một biểu thức sẽ được giữ nguyên. Ví dụ:

```
<h1>Laravel</h1>

Hello, @{{ name }}.
```

Trong ví dụ này, biểu tượng `@` sẽ bị Blade loại bỏ; tuy nhiên, biểu thức `{{name}}` sẽ vẫn không bị công cụ Blade ảnh hưởng, cho phép nó được hiển thị bởi framework JavaScript của bạn.

Biểu tượng `@` cũng có thể được sử dụng để *escape* các chỉ thị của Blade:

```
{{-- Blade template --}}
@@if()

<!-- HTML output -->
@if()
```

Hiển thị JSON

Đôi khi bạn có thể truyền một mảng vào bản view (template) của mình với ý định hiển thị nó dưới dạng JSON để khởi tạo một biến JavaScript nào đó. Ví dụ:

```
<script>
  let app = <?php echo json_encode($array); ?>;
```

```
</script>
```

Tuy nhiên, thay vì gọi `json_encode` theo cách thủ công, bạn có thể sử dụng chỉ thị phương thức `Illuminate\Support\Js::from`. Phương thức `from` cho chép truyền các đối số giống như hàm `json_encode` của PHP; tuy nhiên, nó sẽ đảm bảo rằng JSON kết quả được *escape* đúng cách để đưa vào trong các dấu ngoặc kép của HTML. Phương thức `from` sẽ trả về một câu lệnh JavaScript `JSON.parse` chuỗi mà sẽ chuyển đổi đối tượng hoặc mảng đã cho thành một đối tượng JavaScript hợp lệ:

```
<script>
  let app = {{ Illuminate\Support\Js::from($array) }};
</script>
```

Các phiên bản mới nhất của framework ứng dụng Laravel có sẵn một facade `Js`, sẽ cung cấp quyền truy cập vào chức năng này trong các template Blade của bạn:

```
<script>
  let app = {{ Js::from($array) }};
</script>
```

> **Chú ý:** Bạn chỉ nên sử dụng phương thức `Js::from` để hiển thị các biến hiện có dưới dạng JSON. Tạo template cho Blade dựa trên các biểu thức mẫu và việc cố gắng chuyển một biểu thức phức tạp đến chỉ thị có thể gây ra lỗi không mong muốn.

Chỉ thị @verbatim

Nếu bạn đang hiển thị các biến JavaScript trong một phần lớn template của mình, thì bạn có thể bọc HTML trong chỉ thị `@verbatim` để bạn không phải đặt trước mỗi câu lệnh Blade echo bằng ký hiệu `@`:

```
@verbatim
  <div class="container">
    Hello, {{ name }}.
  </div>
@endverbatim
```

Các chỉ thị Blade

Ngoài việc kế thừa template và hiển thị dữ liệu, Blade còn cung cấp các shortcut thuận tiện cho các cấu trúc điều khiển PHP phổ biến, chẳng hạn như các mệnh đề điều kiện và vòng lặp có điều kiện. Các shortcut này cung cấp một cách làm việc rất gọn gàng, nó gọn gàng với các cấu trúc điều khiển PHP trong khi vẫn quen thuộc với các đối tác PHP của chúng.

Phát biểu if

Bạn có thể tạo câu lệnh `if` bằng cách sử dụng các chỉ thị `@if`, `@elseif`, `@else` và `@endif`. Các chỉ thị này hoạt động giống hệt với các lệnh PHP của chúng:

```
@if (count($records) === 1)
  I have one record!
@elseif (count($records) > 1)
  I have multiple records!
@else
  I don't have any records!
@endif
```

Để thuận tiện, Blade cũng cung cấp chỉ thị `@unless`:

```
@unless (Auth::check())
  You are not signed in.
@endunless
```

Ngoài các mệnh đề điều kiện đã được thảo luận, các chỉ thị `@isset` và `@empty` có thể được sử dụng làm shortcut thuận tiện thay cho các hàm PHP thuần của chúng:

```
@isset($records)
  // $records is defined and is not null...
@endisset

@empty($records)
  // $records is "empty"...
@endempty
```

Chỉ thị xác thực

Các chỉ thị `@auth` và `@guest` có thể được sử dụng để nhanh chóng xác định xem người dùng hiện tại được xác thực hay là khách:

```
@auth
  // The user is authenticated...
@endauth

@guest
  // The user is not authenticated...
@endguest
```

Nếu cần, bạn có thể chỉ định cụ thể chương trình bảo vệ xác thực cần được kiểm tra khi sử dụng chỉ thị `@auth` và `@guest`:

```
@auth('admin')
  // The user is authenticated...
@endauth

@guest('admin')
  // The user is not authenticated...
@endguest
```

Chỉ thị môi trường

Bạn có thể kiểm tra xem ứng dụng có đang chạy trong môi trường *production* hay không bằng cách sử dụng chỉ thị `@production`:

```
@production
  // Production specific content...
@endproduction
```

Hoặc, bạn có thể xác định xem ứng dụng có đang chạy trong một môi trường cụ thể hay không bằng cách sử dụng chỉ thị `@env`:

```
@env('staging')
  // The application is running in "staging"...
@endenv

@env(['staging', 'production'])
  // The application is running in "staging" or "production"...
@endenv
```

Chỉ thị section

Bạn có thể xác định xem phần kế thừa template có nội dung hay không bằng cách sử dụng lệnh `@hasSection`:

```
@hasSection('navigation')
  <div class="pull-right">
    @yield('navigation')
  </div>

  <div class="clearfix"></div>
@endif
```

Bạn có thể sử dụng chỉ thị `@sectionMissing` để xác định xem một section nào đó không có nội dung:

```
@sectionMissing('navigation')
  <div class="pull-right">
    @include('default-navigation')
  </div>
@endif
```

Phát biểu switch

Các câu lệnh switch có thể được tạo bằng cách sử dụng các chỉ thị `@switch`, `@case`, `@break`, `@default` và `@endswitch`:

```
@switch($i)
  @case(1)
    First case...
    @break

  @case(2)
    Second case...
    @break

  @default
    Default case...
@endswitch
```

Vòng lặp

Ngoài các câu lệnh điều kiện, Blade cũng cung cấp các chỉ thị đơn giản để làm việc với các cấu trúc vòng lặp của PHP. Một lần nữa, mỗi chỉ thị này hoạt động giống hệt nhau đối với các lệnh PHP của chúng:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@forelse ($users as $user)
    <li>{{ $user->name }}</li>
@empty
    <p>No users</p>
@endforelse

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

> Trong khi lặp qua vòng lặp `foreach`, bạn có thể sử dụng biến vòng lặp để nhận thông tin có giá trị về vòng lặp, chẳng hạn như bạn đang ở lần lặp đầu tiên hay cuối cùng qua vòng lặp.

Khi sử dụng các vòng lặp, bạn cũng có thể kết thúc vòng lặp hoặc bỏ qua bước lặp hiện tại bằng cách sử dụng chỉ thị `@continue` và `@break`:

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

Bạn cũng có thể đưa vào các điều kiện tiếp tục hoặc ngắt quãng trong khai báo chỉ thị:

```
@foreach ($users as $user)
  @continue($user->type == 1)

  <li>{{ $user->name }}</li>

  @break($user->number == 5)
@endforeach
```

Các biến lặp

Trong khi lặp qua vòng lặp `foreach`, một biến `$loop` sẽ có sẵn bên trong vòng lặp của bạn. Biến này cung cấp quyền truy cập vào một số thông tin hữu ích như chỉ số vòng lặp hiện tại và xác định được đây là lần lặp đầu tiên hay lần cuối cùng qua vòng lặp:

```
@foreach ($users as $user)
  @if ($loop->first)
    This is the first iteration.
  @endif

  @if ($loop->last)
    This is the last iteration.
  @endif

  <p>This is user {{ $user->id }}</p>
@endforeach
```

Nếu bạn đang ở trong một vòng lặp lồng nhau, bạn có thể truy cập vào biến `$loop` của vòng lặp, vòng lặp mẹ được truy cập thông qua thuộc tính `parent`:

```
@foreach ($users as $user)
  @foreach ($user->posts as $post)
    @if ($loop->parent->first)
      This is the first iteration of the parent loop.
    @endif
```

```
    @endforeach
@endforeach
```

Biến `$loop` cũng chứa nhiều thuộc tính hữu ích khác:

Thuộc tính	Mô tả
`$loop->index`	Chỉ số của lần lặp vòng lặp hiện tại (bắt đầu từ 0).
`$loop->iteration`	Các lần đã lặp trong vòng lặp hiện tại (bắt đầu từ 1).
`$loop->remaining`	Các lần lặp còn lại trong vòng lặp.
`$loop->count`	Tổng số mục trong mảng đang được lặp.
`$loop->first`	Có phải đây là lần lặp đầu tiên trong vòng lặp.
`$loop->last`	Có phải đây là lần lặp cuối cùng trong vòng lặp.
`$loop->even`	Có phải đây là một lần lặp chẵn trong vòng lặp.
`$loop->odd`	Có phải đây là một lần lặp lẻ trong vòng lặp.
`$loop->depth`	Cấp độ lồng bên trong vòng lặp khác của vòng lặp.
`$loop->parent`	Khi ở trong một vòng lặp con, sẽ truy cập vòng lặp mẹ.

Bảng thuộc tính loop

Chỉ thị `@class` CSS

Chỉ thị `@class` biên dịch với điều kiện hiển thị một chuỗi class CSS. Chỉ thị chấp nhận một mảng các class trong đó key của mảng chứa tên CSS class hoặc các class do bạn tự chọn, trong khi giá trị là một biểu thức boolean. Nếu phần tử mảng có key số, nó sẽ luôn được đưa vào danh sách class CSS được hiển thị:

```
@php
    $isActive = false;
    $hasError = true;
@endphp

<span @class([
    'p-4',
    'font-bold' => $isActive,
```

```
  'text-gray-500' => ! $isActive,
  'bg-red' => $hasError,
])></span>

<span class="p-4 text-gray-500 bg-red"></span>
```

Đưa view con vào

> Mặc dù bạn có thể tự do sử dụng chỉ thị `@include`, nhưng các component của Blade cũng cung cấp chức năng tương tự và cung cấp một số tiện ích so với chỉ thị `@include` như liên kết dữ liệu và thuộc tính.

Chỉ thị `@include` của Blade cho phép bạn đưa template Blade vào từ bên trong một template khác. Tất cả các biến có sẵn trong template mẹ sẽ được cung cấp cho template con:

```
<div>
  @include('shared.errors')

  <form>
    <!-- Form Contents -->
  </form>
</div>
```

Mặc dù template con sẽ kế thừa tất cả dữ liệu có sẵn trong template mẹ, bạn cũng có thể chuyển một mảng dữ liệu bổ sung cần được cung cấp cho template con:

```
@include('view.name', ['status' => 'complete'])
```

Nếu bạn cố gắng `@include` một template không tồn tại, thì Laravel sẽ báo lỗi. Nếu bạn muốn đưa một template con có thể có hoặc có thể không, bạn nên sử dụng chỉ thị `@includeIf`:

```
@includeIf('view.name', ['status' => 'complete'])
```

Nếu bạn muốn `@include` một template khi một biểu thức boolean nhất định đánh giá là `true` hoặc `false`, thì bạn có thể sử dụng chỉ thị `@includeWhen` và `@includeUnless`:

```
@includeWhen($boolean, 'view.name', ['status' => 'complete'])

@includeUnless($boolean, 'view.name', ['status' => 'complete'])
```

Để đưa template đầu tiên tồn tại từ một mảng các template nhất định vào trang, bạn có thể sử dụng chỉ thị `@includeFirst`:

```
@includeFirst(['custom.admin', 'admin'], ['status' => 'complete'])
```

> **Chú ý:** Bạn nên tránh sử dụng các hằng số `__DIR__` và `__FILE__` trong template Blade của bạn, vì chúng sẽ tham chiếu đến vị trí của template đã biên dịch, được lưu trong bộ nhớ cache.

Hiển thị view với collection

Bạn có thể kết hợp các vòng lặp và gộp lại thành một dòng với chỉ thị `@each` của Blade:

```
@each('view.name', $jobs, 'job')
```

Đối số đầu tiên của chỉ thị `@each` là tên template hiển thị cho từng phần tử trong mảng hoặc collection. Đối số thứ hai là mảng hoặc collection mà bạn muốn lặp lại, trong khi đối số thứ ba là tên biến sẽ được gán cho lần lặp hiện tại trong template. Vì vậy, ví dụ: nếu bạn đang lặp lại một mảng `jobs`, thông thường, bạn sẽ muốn truy cập từng công việc dưới dạng một biến `job` trong template. Key chỉ mục trong mảng cho lần lặp hiện tại sẽ có sẵn tại biến `key` trong template.

Bạn cũng có thể truyền đối số thứ tư cho chỉ thị `@each`. Đối số này xác định chế độ xem sẽ được hiển thị nếu mảng đã cho trống.

```
@each('view.name', $jobs, 'job', 'view.empty')
```

> **Chú ý:** Các chế độ xem được hiển thị qua `@each` không kế thừa các biến từ chế độ xem gốc. Nếu chế độ xem con yêu cầu các biến này, bạn nên sử dụng các lệnh `@foreach` và `@include` để thay thế.

Chỉ thị @once

Chỉ thị `@once` cho phép bạn xác định một phần của template sẽ chỉ được đưa vào một lần. Điều này có thể hữu ích để đẩy một đoạn JavaScript nhất định vào tiêu đề của trang bằng cách sử dụng các stack. Ví dụ: nếu bạn đang hiển thị một component nhất định trong một vòng lặp, thì bạn có thể chỉ muốn đẩy JavaScript vào header ở lần lặp đầu tiên:

```
@once
  @push('scripts')
    <script>
      // Your custom JavaScript...
    </script>
  @endpush
@endonce
```

PHP thuần

Trong một số tình huống, rất hữu ích khi nhúng code PHP vào các template của bạn. Bạn có thể sử dụng chỉ thị Blade `@php` để thực thi một đoạn mã PHP thuần trong template của mình:

```
@php
  $counter = 1;
@endphp
```

Comment

Blade cũng cho phép bạn tạo các comment trong template của mình. Tuy nhiên, không giống như các comment trong HTML, các comment Blade không được để trong HTML mà ứng dụng của bạn sẽ trả về:

```
{{-- This comment will not be present in the rendered HTML --}}
```

Component

Các component và slot sẽ cung cấp các tiện ích tương tự cho các section, layout và include;

Có hai cách tiếp cận để viết các component: các component dựa trên class và các component nặc danh.

Để tạo một component dựa trên class, bạn có thể sử dụng lệnh Artisan `make:component`. Để minh họa cách sử dụng các component, chúng ta sẽ tạo một component `Alert` đơn giản. Lệnh `make:component` sẽ cài đặt component trong thư mục *App\View\Components*:

```
php artisan make:component Alert
```

Lệnh `make:component` cũng sẽ tạo một template cho component. Template sẽ được đặt trong thư mục *resources/views/components*. Khi viết các component cho ứng dụng của riêng bạn, các component sẽ tự động được tìm thấy trong thư mục *app/View/Components* và thư mục *resources/views/components*, vì vậy thường không cần đăng ký thêm component nào.

Bạn cũng có thể tạo các component trong các thư mục con:

```
php artisan make:component Forms/Input
```

Lệnh trên sẽ tạo component `Input` trong thư mục `App\View\Components\Forms` và template sẽ được đặt trong thư mục *resources/views/components/forms*.

Đăng ký thủ công các component Package

Khi viết các component cho ứng dụng của riêng bạn, các component sẽ tự động được phát hiện trong thư mục `app/View/Components` và thư mục *resources/views/components*.

Tuy nhiên, nếu bạn đang xây dựng một package trong đó vận dụng các component Blade, bạn sẽ cần phải đăng ký thủ công class component của mình và liên kết vào thẻ HTML của nó. Thông thường, bạn nên đăng ký các component của mình trong phương thức boot của service provider của gói của bạn:

```
use Illuminate\Support\Facades\Blade;


/**
 * Bootstrap your package's services.
 */
public function boot()
{
```

```
    Blade::component('package-alert', Alert::class);
}
```

Khi component của bạn đã được đăng ký, nó có thể được hiển thị bằng html-tag của nó:

```
<x-package-alert/>
```

Ngoài ra, bạn có thể sử dụng phương thức `componentNamespace` để tự động nạp các class component theo quy ước. Ví dụ: một package `Nightshade` có thể có các component `Calendar` và `ColorPicker` nằm trong namespace *Package\Views\Components*:

```
use Illuminate\Support\Facades\Blade;

/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
    Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

Bây giờ, nó sẽ cho phép sử dụng các component package bởi namespace của nhà cung cấp của họ bằng cách sử dụng cú pháp `package-name::`:

```
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade sẽ tự động phát hiện class được liên kết với component này bằng cách viết đúng tên component. Các thư mục con cũng được hỗ trợ bằng cách sử dụng ký hiệu "dấu chấm".

Trình bày các component

Để hiển thị một component, bạn có thể sử dụng tag component Blade trong một trong các template Blade của bạn. Các tag component Blade sẽ bắt đầu bằng chuỗi `x-` theo sau là tên

của class component được viết kiểu chữ viết thường:

```
<x-alert/>

<x-user-profile/>
```

Nếu class component được lồng sâu hơn trong thư mục *App\View\Components*, bạn có thể sử dụng ký tự "chấm" để chỉ ra thư mục được lồng. Ví dụ: nếu chúng ta giả sử một component được đặt tại *App\View\Components\Inputs\Button.php*, chúng ta có thể hiển thị nó như sau:

```
<x-inputs.button/>
```

Truyền dữ liệu vào trong component

Bạn có thể truyền dữ liệu đến các component của Blade bằng cách sử dụng các thuộc tính HTML. Các giá trị thuần, được mã hóa có thể được truyền đến component bằng các chuỗi thuộc tính HTML đơn giản. Các biểu thức và biến PHP phải được truyền đến component thông qua các thuộc tính sử dụng ký tự "hai chấm" như tiền tố:

```
<x-alert type="error" :message="$message"/>
```

Bạn nên xác định dữ liệu cần thiết của component trong phương thức constructor của class của nó. Tất cả các thuộc tính công khai trên một component sẽ tự động được cung cấp cho template của component đó. Không cần thiết phải truyền dữ liệu đến template từ phương thức `render` của component:

```
<?php

namespace App\View\Components;

use Illuminate\View\Component;

class Alert extends Component
{
  /**
```

```php
     * The alert type.
     *
     * @var string
     */
    public $type;

    /**
     * The alert message.
     *
     * @var string
     */
    public $message;

    /**
     * Create the component instance.
     *
     * @param  string  $type
     * @param  string  $message
     * @return void
     */
    public function __construct($type, $message)
    {
        $this->type = $type;
        $this->message = $message;
    }

    /**
     * Get the view / contents that represent the component.
     *
     * @return \Illuminate\View\View|\Closure|string
     */
    public function render()
    {
        return view('components.alert');
    }
}
```

Khi component của bạn được hiển thị, bạn có thể hiển thị nội dung của các biến công khai của component bằng cách lặp lại các biến theo tên:

```
<div class="alert alert-{{ $type }}">
    {{ $message }}
</div>
```

Kiểu viết chữ

Các đối số của hàm tạo component phải được chỉ định bằng cách sử dụng `camelCase`, trong khi `kebab-case` lại được sử dụng khi tham chiếu đến tên đối số trong các thuộc tính HTML của bạn. Ví dụ: cho hàm tạo component như sau:

```
/**
 * Create the component instance.
 *
 * @param  string  $alertType
 * @return void
 */
public function __construct($alertType)
{
  $this->alertType = $alertType;
}
```

Đối số `$alertType` có thể được cung cấp cho component như sau:

```
<x-alert alert-type="danger" />
```

Tránh việc render attribute

Vì một số framework JavaScript như *Alpine.js* cũng sử dụng các thuộc tính có tiền tố dấu hai chấm, nên bạn có thể sử dụng tiền tố dấu hai chấm kép (`::`) để thông báo cho Blade biết rằng attribute này không phải là một biểu thức PHP. Ví dụ: có component như sau:

```
<x-button ::class="{ danger: isDeleting }">
```

```
  Submit
</x-button>
```

HTML sau sẽ được Blade hiển thị:

```
<button :class="{ danger: isDeleting }">
  Submit
</button>
```

Các phương thức component

Ngoài các biến công khai có sẵn cho component template của bạn, bất kỳ phương thức công khai nào trên component đều có thể được gọi. Ví dụ, hãy tưởng tượng một component có phương thức `isSelected`:

```
/**
 * Determine if the given option is the currently selected option.
 *
 * @param  string  $option
 * @return bool
 */
public function isSelected($option)
{
  return $option === $this->selected;
}
```

Bạn có thể thực thi phương thức này từ component template của mình bằng cách gọi biến khớp với tên của phương thức:

```
<option {{ $isSelected($value) ? 'selected="selected"' : '' }} value="{{ $value }}">
    {{ $label }}
</option>
```

Truy cập vào attribute và slot với các class Component

Các component của Blade cũng cho phép bạn truy cập vào tên component, attribute và slot bên trong phương thức render của class. Tuy nhiên, để truy cập dữ liệu này, bạn nên trả về một hàm nặc danh từ phương thức `render` của component của bạn. Hàm này sẽ nhận một mảng `$data` làm đối số duy nhất của nó. Mảng này sẽ chứa một số phần tử cung cấp thông tin về component:

```
/**
 * Get the view / contents that represent the component.
 *
 * @return \Illuminate\View\View|\Closure|string
 */
public function render()
{
  return function (array $data) {
    // $data['componentName'];
    // $data['attributes'];
    // $data['slot'];

    return '<div>Components content</div>';
  };
}
```

Phần tử `componentName` bằng với tên được sử dụng trong thẻ HTML sau tiền tố `x-`. Vì vậy, `componentName` của `<x-alert />` sẽ là `alert`. Phần tử attributes sẽ chứa tất cả các attribute có trên thẻ HTML. Phần tử vị trí là một cá thể `Illuminate\Support\HtmlString` với nội dung của vị trí của thành phần.

Hàm nặc danh sẽ trả về một chuỗi. Nếu chuỗi trả về tương ứng với một template hiện có, template đó sẽ được hiển thị; nếu không, chuỗi trả về sẽ được đánh giá là template Blade ở dạng inline.

Thư viện bổ sung

Nếu component của bạn yêu cầu các thư viện từ service container của Laravel, bạn có thể liệt kê chúng trước bất kỳ attribute dữ liệu nào của component và chúng sẽ tự động được container đưa vào:

```
use App\Services\AlertCreator
```

```php
/**
 * Create the component instance.
 *
 * @param  \App\Services\AlertCreator  $creator
 * @param  string  $type
 * @param  string  $message
 * @return void
 */
public function __construct(AlertCreator $creator, $type, $message)
{
  $this->creator = $creator;
  $this->type = $type;
  $this->message = $message;
}
```

Attribute/Phương thức ẩn

Nếu bạn muốn ngăn một số phương thức hoặc thuộc tính công khai hiển thị dưới dạng biến trong mẫu thành phần của mình, bạn có thể thêm chúng vào thuộc tính mảng $ ngoại trừ trên thành phần của mình:

```php
<?php
namespace App\View\Components;
use Illuminate\View\Component;
class Alert extends Component
{
  /**
   * The alert type.
   *
   * @var string
   */
  public $type;

  /**
   * The properties / methods that should not be exposed to the component template.
   *
```

```
     * @var array
     */
    protected $except = ['type'];
}
```

Các attribute của component

Chúng ta đã kiểm chứng cách truyền các attribute dữ liệu cho một component; tuy nhiên, đôi khi bạn có thể cần chỉ định các attribute HTML bổ sung, chẳng hạn như class, không phải là một phần của dữ liệu cần thiết để một component hoạt động. Thông thường, bạn muốn truyền các attribute bổ sung này xuống phần tử root của component template. Ví dụ: hãy tưởng tượng chúng ta muốn hiển thị một component `alert` như sau:

```
<x-alert type="error" :message="$message" class="mt-4"/>
```

Tất cả các attribute không phải là một phần của constructor của component sẽ tự động được thêm vào "túi attribute" của component. Túi attribute này được tạo tự động cho component thông qua biến `$attributes`. Tất cả các attribute có thể được hiển thị trong component bằng cách dùng lại biến này như sau:

```
<div {{ $attributes }}>
    <!-- Component content -->
</div>
```

> **Chú ý:** Việc sử dụng các lệnh như `@env` trong các tag component sẽ không được hỗ trợ tại thời điểm này. Ví dụ: `<x-alert: live="@env('production')" />` sẽ không được compile.

Các attribute có sẵn hoặt được thêm

Đôi khi bạn có thể cần phải chỉ định các giá trị mặc định cho các attribute hoặc hợp nhất các giá trị bổ sung vào một số attribute của component. Để thực hiện điều này, bạn có thể sử dụng phương thức `merge` của túi attribute. Phương thức này đặc biệt hữu ích để xác định một tập hợp các lớp CSS mặc định luôn được áp dụng cho một component:

```
<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
```

```
  {{ $message }}
</div>
```

Nếu chúng ta giả sử component này được sử dụng như sau đây:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

Cuối cùng, HTML được hiển thị của component sẽ xuất hiện như sau:

```
<div class="alert alert-error mb-4">
  <!-- Contents of the $message variable -->
</div>
```

Bổ sung class có điều kiện

Đôi khi bạn có thể muốn hợp nhất các class theo một điều kiện `true`. Bạn có thể thực hiện điều này thông qua phương thức `class`, phương thức này chấp nhận một mảng các class trong đó key của mảng chứa class hoặc các class bạn muốn thêm, trong khi giá trị là một biểu thức boolean. Nếu phần tử mảng có key chỉ mục là số, thì nó sẽ luôn được đưa vào danh sách class được hiển thị:

```
<div {{ $attributes->class(['p-4', 'bg-red' => $hasError]) }}>
  {{ $message }}
</div>
```

Nếu bạn cần hợp nhất các attribute khác vào component của mình, bạn có thể dùng thêm phương thức `merge` vào phương thức `class`:

```
<button {{ $attributes->class(['p-4'])->merge(['type' => 'button']) }}>
  {{ $slot }}
</button>
```

> Nếu bạn cần biên dịch theo điều kiện với các class trên các phần tử HTML khác mà không nhận lại các attributes đã hợp nhất, thì bạn có thể sử dụng chỉ thị `@class`.

Bổ sung các attribute ngoài `class`

Khi hợp nhất các attribute không phải là attribute `class`, các giá trị được cung cấp cho phương thức `merge` sẽ được coi là giá trị "mặc định" của attribute. Tuy nhiên, không giống như attribute class, các attribute này sẽ không được `merge` với các giá trị attribute đã được đưa vào. Thay vào đó, chúng sẽ bị ghi đè. Ví dụ: về việc thực thi của một component `button` có thể giống như sau:

```
<button {{ $attributes->merge(['type' => 'button']) }}>
  {{ $slot }}
</button>
```

Để hiển thị component button với kiểu tùy chỉnh, nó có thể được chỉ định khi sử dụng component. Nếu không có `type` nào được chỉ định, thì `type` button sẽ được sử dụng:

```
<x-button type="submit">
  Submit
</x-button>
```

HTML được hiển thị của component button trong ví dụ này sẽ là:

```
<button type="submit">
  Submit
</button>
```

Nếu bạn muốn một attribute không phải là class có giá trị mặc định của nó và các giá trị được đưa vào được kết hợp với nhau, thì bạn có thể sử dụng phương thức `prepends`. Trong ví dụ này, attribute `data-controller` sẽ luôn bắt đầu bằng `profile-controller` và bất kỳ giá trị `data-controller` bổ sung nào sẽ được đặt sau giá trị mặc định này:

```
<div {{ $attributes->merge(['data-controller' => $attributes->prepends('profile-controller')]) }}
  {{ $slot }}
</div>
```

Nhận và lọc các attribute

Bạn có thể lọc các attribute bằng phương thức `filter`. Phương thức này chấp nhận một hàm nặc danh sẽ trả về `true` nếu bạn muốn giữ lại attribute trong túi attribute:

```
{{ $attributes->filter(fn ($value, $key) => $key == 'foo') }}
```

Để thuận tiện, bạn có thể sử dụng phương thức `whereStartsWith` để truy xuất tất cả các attribute có khóa bắt đầu bằng một chuỗi đã cho:

```
{{ $attributes->whereStartsWith('wire:model') }}
```

Ngược lại, phương thức `whereDoesntStartWith` có thể được sử dụng để loại trừ tất cả các attribute có khóa bắt đầu bằng một chuỗi nhất định:

```
{{ $attributes->whereDoesntStartWith('wire:model') }}
```

Sử dụng phương thức `first`, bạn có thể hiển thị attribute đầu tiên trong một túi attribute nhất định:

```
{{ $attributes->whereStartsWith('wire:model')->first() }}
```

Nếu bạn muốn kiểm tra xem một atrribute có xuất hiện trên component hay không, bạn có thể sử dụng phương thức `has`. Phương thức này chấp nhận tên attribute làm đối số duy nhất của nó và trả về một boolean cho biết attribute có hiện diện hay không:

```
@if ($attributes->has('class'))
  <div>Class attribute is present</div>
@endif
```

Bạn có thể truy xuất giá trị của một attribute cụ thể bằng cách sử dụng phương thức `get`:

```
{{ $attributes->get('class') }}
```

Từ khoá riêng

Mặc định, một số từ khóa được dành riêng cho việc sử dụng nội bộ của Blade để hiển thị các component. Các từ khóa sau không thể được xác định là attribute công khai hoặc tên phương thức trong các thành phần của bạn:

- `data`
- `render`
- `resolveView`
- `shouldRender`
- `view`
- `withAttributes`
- `withName`

Slot - vị trí

Bạn thường sẽ cần truyền nội dung bổ sung cho component của mình thông qua "slot". Các component slot được hiển thị bằng cách dùng lại biến `$slot`. Để tìm hiểu khái niệm này, hãy tưởng tượng rằng một component `alert` có html như sau:

```
<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

Chúng tôi có thể truyền nội dung vào `slot` bằng cách đưa nội dung vào component:

```
<x-alert>
  <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Đôi khi một component có thể cần hiển thị nhiều slot khác nhau ở các vị trí khác nhau trong component. Hãy sửa đổi component alert của chúng ta để cho phép đưa vào slot "title":

```
<!-- /resources/views/components/alert.blade.php -->
<span class="alert-title">{{ $title }}</span>
<div class="alert alert-danger">
```

```
    {{ $slot }}
</div>
```

Bạn có thể xác định nội dung của slot được đặt tên bằng tag `x-slot`. Mọi nội dung không nằm trong tag `x-slot` minh bạch sẽ được chuyển đến component trong biến `$slot`:

```
<x-alert>
    <x-slot name="title">
        Server Error
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Slot được định vị

Nếu bạn đã sử dụng framework JavaScript như Vue chẳng hạn, thì bạn có thể quen thuộc với "scoped slots", cho phép bạn truy cập dữ liệu hoặc phương thức từ component trong slot của bạn. Bạn có thể có được tính năng tương tự trong Laravel bằng cách xác định các phương thức hoặc thuộc tính public trên component của bạn và truy cập component trong slot của bạn thông qua biến `$component`. Trong ví dụ này, chúng tôi sẽ giả định rằng component `x-alert` có phương thức public `formatAlert` được xác định trên class component của nó:

```
<x-alert>
    <x-slot name="title">
        {{ $component->formatAlert('Server Error') }}
    </x-slot>

    <strong>Whoops!</strong> Something went wrong!
</x-alert>
```

Slot của các attribute

Giống như các component Blade, bạn có thể chỉ định các attribute bổ sung cho các slot như tên class CSS:

```
<x-card class="shadow-sm">
  <x-slot name="heading" class="font-bold">
    Heading
  </x-slot>

  Content

  <x-slot name="footer" class="text-sm">
    Footer
  </x-slot>
</x-card>
```

Để tương tác với các slot attribute, bạn có thể truy cập thuộc tính `atrributes` của biến slot. Để biết thêm thông tin về cách tương tác với các attribute, vui lòng tham khảo tài liệu về các attribute component:

```
@props([
  'heading',
  'footer',
])

<div {{ $attributes->class(['border']) }}>
  <h1 {{ $heading->attributes->class(['text-lg']) }}>
    {{ $heading }}
  </h1>

  {{ $slot }}

  <footer {{ $footer->attributes->class(['text-gray-700']) }}>
    {{ $footer }}
  </footer>
</div>
```

View của component inline

Đối với các component nhỏ, bạn sẽ có thể cảm thấy cồng kềnh khi quản lý cả class của

component và template của component. Vì lý do này, bạn có thể trả lại html của component một cách trực tiếp từ phương thức `render`:

```
/**
 * Get the view / contents that represent the component.
 *
 * @return \Illuminate\View\View|\Closure|string
 */
public function render()
{
  return <<<'blade'
    <div class="alert alert-danger">
      {{ $slot }}
    </div>
  blade;
}
```

Tạo các component có view inline

Để tạo một component mà hiển thị view inline, bạn có thể sử dụng tùy chọn `inline` khi thực hiện lệnh Artisan `make:component`:

```
php artisan make:component Alert --inline
```

Các component ẩn danh

Tương tự như các inline component, các component ẩn danh cung cấp cơ chế quản lý một component thông qua một tập tin duy nhất. Tuy nhiên, các component ẩn danh sử dụng một tập tin template duy nhất và không có class liên kết. Để tạo một component ẩn danh, bạn chỉ cần đặt một template Blade trong thư mục *resources/views/components* của mình. Ví dụ: giả sử bạn đã tạo một component tại *resources/views/components/alert.blade.php*, thì bạn có thể chỉ cần hiển thị nó như sau:

```
<x-alert/>
```

Bạn có thể sử dụng ký tự "dấu chấm" để cho biết một component cụ thể đã được lồng sâu

hơn bên trong thư mục *components*. Ví dụ: giả sử component được tạo tại *resources/views/components/inputs/button.blade.php*, bạn có thể hiển thị nó như sau:

```
<x-inputs.button/>
```

Component chỉ mục ẩn danh

Đôi khi, khi một component được tạo thành từ nhiều template Blade, bạn có thể muốn nhóm các template của component đó trong một thư mục. Ví dụ, hãy tưởng tượng một component "accordion" có cấu trúc thư mục như sau:

```
/resources/views/components/accordion.blade.php
/resources/views/components/accordion/item.blade.php
```

Cấu trúc thư mục này cho phép bạn hiển thị component `accordion` và mục của nó như sau:

```
<x-accordion>
  <x-accordion.item>
    ...
  </x-accordion.item>
</x-accordion>
```

Tuy nhiên, để hiển thị component `accordion` thông qua `x-accordion`, chúng tôi buộc phải đặt template của component accordion "index" trong thư mục *resources/views/components* thay vì lồng nó trong thư mục *accordion* với các template liên quan đến accordion khác.

Rất may, Blade cho phép bạn đặt tập tin *index.blade.php* trong thư mục *template* của component. Khi một template *index.blade.php* tồn tại cho component, nó sẽ được hiển thị dưới dạng node "root" của component. Vì vậy, chúng ta có thể tiếp tục sử dụng cùng một cú pháp Blade được đưa ra trong ví dụ trên; tuy nhiên, chúng tôi sẽ điều chỉnh cấu trúc thư mục của mình như sau:

```
/resources/views/components/accordion/index.blade.php
/resources/views/components/accordion/item.blade.php
```

Các thuộc tính/attribute dữ liệu

Vì các component ẩn danh không có bất kỳ class nào được liên kết, nên bạn có thể thắc mắc làm cách nào để phân biệt dữ liệu nào nên được truyền cho component dưới dạng biến và atrribute nào nên được đặt trong túi attribute của component.

Bạn có thể chỉ định attribute nào nên được coi là biến dữ liệu bằng cách sử dụng chỉ thị `@props` ở đầu template Blade của component của bạn. Tất cả các attribute khác trên component sẽ có sẵn thông qua túi attribute của component. Nếu bạn muốn cung cấp cho một biến dữ liệu một giá trị mặc định, bạn có thể chỉ định tên của biến làm khóa mảng và giá trị mặc định là giá trị mảng:

```
<!-- /resources/views/components/alert.blade.php -->

@props(['type' => 'info', 'message'])

<div {{ $attributes->merge(['class' => 'alert alert-'.$type]) }}>
  {{ $message }}
</div>
```

Với khai báo component ở trên, chúng ta có thể hiển thị component như sau:

```
<x-alert type="error" :message="$message" class="mb-4"/>
```

Truy cập vào dữ liệu mẹ

Đôi khi bạn có thể muốn truy cập dữ liệu từ một component mẹ bên trong một component con. Trong những trường hợp này, bạn có thể sử dụng chỉ thị `@aware`. Ví dụ: hãy tưởng tượng chúng ta đang xây dựng một component `menu` phức tạp bao gồm `<x-menu>` mẹ và con là `<x-menu.item>`:

```
<x-menu color="purple">
  <x-menu.item>...</x-menu.item>
  <x-menu.item>...</x-menu.item>
</x-menu>
```

Component `<x-menu>` có thể có cách triển khai như sau:

```
<!-- /resources/views/components/menu/index.blade.php -->
```

```
@props(['color' => 'gray'])
<ul {{ $attributes->merge(['class' => 'bg-'.$color.'-200']) }}>
  {{ $slot }}
</ul>
```

Bởi vì `color` prop chỉ được truyền vào component mẹ (`<x-menu>`), nên nó sẽ không có sẵn cho component con `<x-menu.item>`. Tuy nhiên, nếu chúng ta sử dụng lệnh `@aware`, chúng ta cũng có thể cung cấp nó bên trong component con `<x-menu.item>`:

```
<!-- /resources/views/components/menu/item.blade.php -->
@aware(['color' => 'gray'])
<li {{ $attributes->merge(['class' => 'text-'.$color.'-800']) }}>
  {{ $slot }}
</li>
```

Các component động

Đôi khi bạn có thể cần render một component nhưng không biết component nào nên được render cho đến khi chạy. Trong trường hợp này, bạn có thể sử dụng `dynamic-component` tích hợp sẵn trong Laravel để hiển thị component dựa trên giá trị hoặc biến runtime:

```
<x-dynamic-component :component="$componentName" class="mt-4" />
```

Đăng ký thủ công các component

> **Chú ý:** Tài liệu sau đây về đăng ký thủ công các component chủ yếu áp dụng cho những người đang viết các package Laravel bao gồm các component dạng view. Nếu bạn không viết một package, phần này của tài liệu component có thể không liên quan đến bạn.

Khi viết các component cho ứng dụng của riêng bạn, các component sẽ tự động được phát hiện trong thư mục *app/View/Components* và thư mục *resources/views/components*.

Tuy nhiên, nếu bạn đang xây dựng một package sử dụng các component Blade hoặc đặt các component trong các thư mục không thông dụng, bạn sẽ cần phải đăng ký thủ công class component của mình và liên kết HTML của nó để Laravel biết nơi tìm component. Thông

thường, bạn nên đăng ký các component của mình trong phương thức `boot` của service provider trong package của bạn:

```
use Illuminate\Support\Facades\Blade;
use VendorPackage\View\Components\AlertComponent;
/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
  Blade::component('package-alert', AlertComponent::class);
}
```

Khi component của bạn đã được đăng ký, nó có thể được hiển thị bằng liên kết html của nó:

```
<x-package-alert/>
```

Tự động nạp component package

Ngoài ra, bạn có thể sử dụng phương thức `componentNamespace` để tự động nạp các class component theo quy ước. Ví dụ: một package `Nightshade` có thể có các component `Calendar` và `ColorPicker` nằm trong namespace *Package\Views\Components*:

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

/**
 * Bootstrap your package's services.
 *
 * @return void
 */
public function boot()
{
  Blade::componentNamespace('Nightshade\\Views\\Components', 'nightshade');
}
```

Điều này sẽ cho phép sử dụng các component package bằng namespace của nhà cung cấp của chúng với việc sử dụng cú pháp `package-name::`:

```blade
<x-nightshade::calendar />
<x-nightshade::color-picker />
```

Blade sẽ tự động phát hiện class được liên kết với component này bằng cách viết hoa cho tên component. Các thư mục con cũng được hỗ trợ bằng cách sử dụng ký hiệu "dấu chấm".

Xây dựng layout

Các layout sử dụng component

Hầu hết các ứng dụng web duy trì cùng một layout chung trên các trang khác nhau. Sẽ vô cùng cồng kềnh và khó duy trì ứng dụng của chúng ta nếu chúng ta phải lặp lại toàn bộ HTML layout trong mọi template chúng ta tạo. Có một tiện ích khác đó là khi xác định layout như một component Blade duy nhất và sau đó sử dụng nó trong toàn bộ ứng dụng của chúng ta.

Tạo layout component

Ví dụ, hãy tưởng tượng chúng ta đang xây dựng một ứng dụng danh sách "todo". Chúng ta có thể xác định một component layout trông giống như sau:

```
<!-- resources/views/components/layout.blade.php -->

<html>
  <head>
    <title>{{ $title ?? 'Todo Manager' }}</title>
  </head>
  <body>
    <h1>Todos</h1>
    <hr/>
    {{ $slot }}
  </body>
</html>
```

Áp dụng layout component

Khi `layout` component đã được tạo, chúng ta có thể tạo template Blade sử dụng component đó. Trong ví dụ này, chúng ta sẽ xác định một template đơn giản hiển thị danh sách công việc của chúng ta:

```
<!-- resources/views/tasks.blade.php -->

<x-layout>
  @foreach ($tasks as $task)
    {{ $task }}
  @endforeach
</x-layout>
```

Hãy nhớ rằng, nội dung được đưa vào một component sẽ được cung cấp cho biến `$slot` mặc định trong layout component của chúng ta. Như bạn có thể nhận thấy, layout của chúng ta cũng trông cậy vào slot `$title` nếu slot được cung cấp; bằng không, thì một tiêu đề mặc định được hiển thị. Chúng ta có thể đưa tiêu đề tùy chỉnh từ template danh sách công việc của mình bằng cách sử dụng cú pháp slot tiêu chuẩn được thảo luận trong tài liệu component:

```
<!-- resources/views/tasks.blade.php -->
```

```
<x-layout>
  <x-slot name="title">
    Custom Title
  </x-slot>

  @foreach ($tasks as $task)
    {{ $task }}
  @endforeach
</x-layout>
```

Bây giờ chúng ta đã xác định các template layout và danh sách công việc của mình. Việc tiếp theo, chúng ta chỉ cần trả lại template công việc từ một route:

```
use App\Models\Task;

Route::get('/tasks', function () {
  return view('tasks', ['tasks' => Task::all()]);
});
```

Layout sử dụng thừa kế template

Tạo một layout

Layout cũng có thể được tạo thông qua "kế thừa template - template inheritance". Đây là cách chính để xây dựng các ứng dụng trước khi giới thiệu các component.

Để bắt đầu, chúng ta hãy xem một ví dụ đơn giản. Đầu tiên, chúng ta sẽ kiểm chứng layout trang. Vì hầu hết các ứng dụng web duy trì cùng một layout chung trên các trang khác nhau, nên thật tiện lợi khi xác định layout này dưới dạng một template Blade duy nhất:

```
<!-- resources/views/layouts/app.blade.php -->

<html>
  <head>
    <title>App Name - @yield('title')</title>
  </head>
```

```
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show

    <div class="container">
      @yield('content')
    </div>
  </body>
</html>
```

Như bạn có thể thấy, tập tin này chứa code HTML điển hình. Tuy nhiên, hãy lưu ý đến các lệnh `@section` và `@yield`. Chỉ thị `@section`, như tên của nó, khai báo một section nội dung, trong khi chỉ thị `@yield` được gọi để hiển thị nội dung của một section nào đó.

Bây giờ chúng ta đã tạo ra layout cho ứng dụng của mình, hãy tạo một trang con kế thừa layout.

Mở rộng một layout

Khi xác định template con, hãy sử dụng chỉ thị `@extends` Blade để chỉ định layout mà template con sẽ "kế thừa - inherit". Các template mở rộng layout Blade có thể đưa nội dung vào các section của layout bằng cách sử dụng lệnh `@section`. Hãy nhớ rằng, như đã thấy trong ví dụ trên, nội dung của các section này sẽ được hiển thị trong bố cục sử dụng `@yield`:

```
<!-- resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')
  @parent

  <p>This is appended to the master sidebar.</p>
@endsection
```

```
@section('content')
  <p>This is my body content.</p>
@endsection
```

Trong ví dụ này, section của sidebar đang có sử dụng lệnh `@parent` để thêm (thay vì ghi đè) nội dung vào sidebar của layout. Lệnh `@parent` sẽ được thay thế bằng nội dung của layout khi template được hiển thị.

> Trái ngược với ví dụ trước, section của sidebar này kết thúc bằng `@endsection` thay vì `@show`. Chỉ thị `@endsection` sẽ chỉ tạo một section trong khi `@show` sẽ tạo và ngay lập tức mang nó lại section đó.

Chỉ thị `@yield` cũng chấp nhận một giá trị mặc định làm đối số thứ hai của nó. Giá trị này sẽ được hiển thị nếu section đang được tạo là undefined:

```
@yield('content', 'Default content')
```

Các HTML Form

Field CSRF

Bất cứ khi nào bạn tạo một HTML form trong ứng dụng của mình, bạn nên đưa field mã token CSRF ẩn vào trong form để phần middleware chứng thực CSRF có thể xác thực yêu cầu. Bạn có thể sử dụng chỉ thị `@csrf` Blade để tạo trường mã token:

```
<form method="POST" action="/profile">
  @csrf

  ...
</form>
```

Field Method

Vì các HTML form không thể thực hiện các yêu cầu `PUT`, `PATCH` hoặc `DELETE`, bạn sẽ cần thêm field `_method` ẩn để giả mạo các HTTP method này. Chỉ thị `@method` Blade có thể tạo loại field này cho bạn:

```
<form action="/foo/bar" method="POST">
  @method('PUT')

  ...
</form>
```

Lỗi giám định form

Chỉ thị `@error` có thể được sử dụng để nhanh chóng kiểm tra xem thông báo lỗi giám định có tồn tại cho một attribute nào đó hay không. Trong chỉ thị `@error`, bạn có thể sử dụng biến `$message` để hiển thị thông báo lỗi:

```
<!-- /resources/views/post/create.blade.php -->

<label for="title">Post Title</label>

<input id="title" type="text" class="@error('title') is-invalid @enderror">

@error('title')
  <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Vì chỉ thị `@error` biên dịch thành câu lệnh "`if`", nên bạn có thể sử dụng chỉ thị `@else` để hiển thị nội dung khi không có lỗi cho một attribute nào:

```
<!-- /resources/views/auth.blade.php -->

<label for="email">Email address</label>

<input id="email" type="email" class="@error('email') is-invalid @else is-valid @enderror">
```

Bạn có thể truyền tên của một túi lỗi cụ thể làm đối số thứ hai cho chỉ thị `@error` để truy xuất thông báo lỗi giám định trên các trang chứa nhiều HTML form:

```
<!-- /resources/views/auth.blade.php -->
```

```
<label for="email">Email address</label>

<input id="email" type="email" class="@error('email', 'login') is-invalid @enderror">

@error('email', 'login')
  <div class="alert alert-danger">{{ $message }}</div>
@enderror
```

Các Stack

Blade cho phép bạn đẩy đến các stack được đặt tên có thể được hiển thị ở một nơi khác trong một template hoặc layout khác. Điều này có thể đặc biệt hữu ích để chỉ định bất kỳ thư viện JavaScript nào được yêu cầu bởi các template con của bạn:

```
@push('scripts')
  <script src="/example.js"></script>
@endpush
```

Bạn có thể đẩy đến một stack nhiều lần nếu cần. Để hiển thị toàn bộ nội dung stack, hãy truyền tên của stack vào lệnh `@stack`:

```
<head>
  <!-- Head Contents -->

  @stack('scripts')
</head>
```

Nếu bạn muốn thêm nội dung vào đầu stack, thì bạn nên sử dụng lệnh `@prepend`:

```
@push('scripts')
  This will be second...
@endpush

// Later...
```

```
@prepend('scripts')
  This will be first...
@endprepend
```

Đưa service vào

Chỉ thị `@inject` có thể được sử dụng để truy xuất một service từ service container của Laravel. Đối số đầu tiên được truyền cho `@inject` là tên của biến mà service đã được đặt vào đó, trong khi đối số thứ hai là tên class hoặc interface của service mà bạn muốn lấy vào:

```
@inject('metrics', 'App\Services\MetricsService')

<div>
  Monthly Revenue: {{ $metrics->monthlyRevenue() }}.
</div>
```

Mở rộng blade

Blade cho phép bạn tạo các chỉ thị tự tạo của riêng mình bằng cách sử dụng phương thức `directive`. Khi chương trình biên dịch Blade bắt gặp chỉ thị tự tạo, nó sẽ gọi callback đã được cung cấp với biểu thức mà chỉ thị chứa.

Ví dụ sau tạo một chỉ thị `@datetime($var)` định dạng một biến `$var` đã cho, phải là một phiên bản của `DateTime`:

```
<?php
namespace App\Providers;
use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
  /**
   * Register any application services.
   *
```

```
     * @return void
     */
    public function register()
    {
        //
    }

    /**
     * Bootstrap any application services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }
}
```

Như bạn có thể thấy, chúng ta sẽ gọi thêm phương thức `format` vào bất kỳ biểu thức nào được truyền vào chỉ thị. Vì vậy, trong ví dụ này, PHP cuối cùng được tạo bởi chỉ thị này sẽ là:

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

> **Chú ý:** Sau khi cập nhật logic của một chỉ thị Blade, bạn sẽ cần xóa tất cả các template Blade đã lưu trong bộ nhớ cache. Các template Blade đã lưu trong bộ nhớ cache có thể bị xóa bằng lệnh Artisan `view:clear`.

Tự tạo hàm echo

Nếu bạn cố gắng "echo" một đối tượng bằng Blade, phương thức `__toString` của đối tượng sẽ được gọi. Phương thức `__toString` là một trong những "phương thức ma thuật" được tích hợp sẵn trong PHP. Tuy nhiên, đôi khi bạn có thể không có quyền kiểm soát phương thức `__toString` của một class nhất định, chẳng hạn như khi class mà bạn đang

tương tác thuộc về thư viện của bên thứ ba.

Trong những trường hợp này, Blade cho phép bạn đăng ký một hàm echo tự tạo cho loại đối tượng cụ thể đó. Để thực hiện điều này, bạn nên gọi phương thức `stringable` của Blade. Phương thức `stringable` cho phép nhập một hàm nặc danh. Hàm này cho khai kiểu đối tượng mà nó chịu trách nhiệm hiển thị. Thông thường, phương thức `stringable` được gọi trong phương thức `boot` của lớp `AppServiceProvider` của ứng dụng của bạn:

```
use Illuminate\Support\Facades\Blade;
use Money\Money;

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  Blade::stringable(function (Money $money) {
    return $money->formatTo('en_GB');
  });
}
```

Khi hàm echo tự tạo của bạn đã được tạo, bạn có thể chỉ cần sử dụng lại đối tượng trong template Blade của mình như sau:

```
Cost: {{ $money }}
```

Tự tạo phát biểu if

Lập trình một chỉ thị tự tạo đôi khi phức tạp hơn mức cần thiết khi khai báo các câu lệnh điều kiện tự tạo, đơn giản. Vì lý do đó, Blade cung cấp một phương thức `Blade::if` cho phép bạn nhanh chóng tạo các chỉ thị có điều kiện tự tạo bằng cách sử dụng các hàm nặc danh. Ví dụ: hãy xác định một điều kiện tự tạo để kiểm tra "disk" mặc định được định cấu hình cho ứng dụng. Chúng ta có thể thực hiện điều này trong phương thức `boot` của `AppServiceProvider` của chúng ta:

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

/**
 * Bootstrap any application services.
 *
 * @return void
 */
public function boot()
{
  Blade::if('disk', function ($value) {
    return config('filesystems.default') === $value;
  });
}
```

Khi điều kiện tự tạo đã được khai báo, thì bạn có thể sử dụng nó trong các template của mình:

```
@disk('local')
  <!-- The application is using the local disk... -->
@elsedisk('s3')
  <!-- The application is using the s3 disk... -->
@else
  <!-- The application is using some other disk... -->
@enddisk

@unlessdisk('local')
  <!-- The application is not using the local disk... -->
@enddisk
```