Muhammad Azamuddin
Hafid Mukhlasin

5.7

Laravel

The PHP Framework For Web Artisans

Daftar Isi

View

View merupakan tempat bagi kita untuk meletakkan kode-kode HTML. Kita tidak akan menggunakan lagi file `.html` ya. Tapi kita tidak hanya menggunakan HTML karena kita perlu menghandle tampilan dengan lebih canggih. Menampilkan data yang diberikan oleh controller. Untuk itu kita akan menggunakan templating engine, yaitu Blade.

Blade merupakan templating engine bawaan Laravel. Berguna untuk mempermudah dalam menulis kode tampilan. Dan juga memberikan fitur tambahan untuk memanipulasi data di view yang dilempar dari controller. Apa saja fitur yang ditawarkan oleh Blade?

Menampilkan view dari controller

Untuk menampilkan view dari controller pertama kita buat view terlebih dahulu. Coba buka file `index.blade.php` di path `resources/views/kategori/index.blade.php`. Lalu isi dengan kode sederhana ini:

```
<div>
  <b>TODO: list daftar kategori di view ini</b>
</div>
```

Lalu kita gunakan `CategoryController` yang telah kita buat pada latihan praktik sebelumnya. Buka file `CategoryController.php` lalu ubah action index sehingga menjadi seperti ini:

```
public function index(){
    return view("kategori.index");
}
```

Penjelasan kode: kode di atas akan menghasilkan view `kategori.index` yang baru kita buat ditampilkan di layar. Kita `return` view dari `CategoryController` action `index`.

Coba buka `http://toko-online.test/latihan/kategori/all` kita akan mendapati tampilan seperti ini:

TODO: list daftar kategori di view ini

Memberikan data ke view

Ingat kembali di bab Model dan Eloquent kita telah praktik untuk membuat route dan melakukan manipulasi data menggunakan model `Category`.

Data-data di praktik tersebut ditampilkan ke browser masih dalam bentuk `JSON`. Jika untuk keperluan `web service` maka tidak mengapa, akan tetapi jika kita ingin menggunakan view Laravel, maka seharusnya kita

`return` view dari controller plus dengan data.

Jika tadi kita belajar `return` view tanpa data. Maka untuk `return` view dari action controller beserta dengan data tertentu caranya mudah, yaitu tambahkan data yang akan dikirim sebagai parameter kedua dalam helper `view` dalam bentuk array, seperti ini:

`CategoryController`

```
//..
public function index(){
    $daftar_kategori = \App\Category::all();

    return view("kategori.index", ["daftar_kategori" =>
$daftar_kategori]);
}
//...
```

Penjelasan Kode: Dari controller action `index` pada `CategoryController` kita return sebuah view `kategori.index`. View ini harus sudah dibuat pada path `resources/views/kategori/index.blade.php`. Selain `return` view kita juga memberikan data "daftar_kategori" agar di view tersebut bisa diakses, nilainya adalah daftar kategori hasil ***query*** dengan perintah `App\Category::all()`;

Lalu buka kembali file view `kategori/index.blade.php`, dan ubah isinya menjadi seperti ini:

```
<div>
  Nama Kategori: {{$daftar_kategori[0]->name}}
</div>
```

Penjelasan Kode: Menampilkan kategori pertama (index 0) dari array `$daftar_kategori` yang diperoleh dari `CategoryController` action `index`.

Kita baru saja belajar mengenai dasar view, dan secara tidak sadar kita juga telah menggunakan blade untuk menampilkan data yang dikirim oleh controller. Setelah ini kita akan bahas fitur-fitur Blade yang ada.

Menampilkan data

Untuk menampilkan data kita cukup tuliskan variabel PHP seperti biasa tetapi diapit oleh dua kurung kurawal seperti ini `{{ $namaVariabel }}`. Tentu saja `$namaVariabel` harus sebelumnya dikirim dari controller action.

File app/routes/web.php

```
Route::get('/ucapkan-salam', 'SalamController@beriSalam');
```

Mendefinisikan route `/ucapkan-salam` yang akan mengeksekusi action `beriSalam` pada `SalamController`

File app/Http/Controllers/SalamController.php

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class SalamController extends Controller
{
   public function beriSalam(){
      return view(“salam.index”, [“kalimat” => ”Halo Selamat Datang”]);
   }
}
```

`SalamController` memiliki action beriSalam yang mengembalikan sebuah view `salam/index` dan melempar data “kalimat” yang bernilai “Halo Selamat Datang”

File resources/views/salam/index.blade.php

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

View yang dipakai oleh action beriSalam terletak pada `resources/views/salam/index.blade.php`. Karena action beriSalam pada SalamController melemparkan data “kalimat” maka kini tersedia variabel $kalimat pada view “salam.index” dan ditampilkan ke user dalam div dengan cara `{{ $kalimat }}`

Menampilkan unescaped data

Secara default semua variabel yang tersedia di view dan ditampilkan dengan kode `{{ $namaVariabel }}` otomatis dilempar dengan fungsi htmlspecialchars untuk menghindari serangan XSS. Akan tetapi, terkadang kita memerlukan view untuk menampilkan unescaped data, alias tanpa htmlspecialchars, untuk melakukannya kita gunakan kode seperti ini:

```
{{!!  $dataUnescaped !!}}
```

Sebisa mungkin kita hindari menampilkan unescaped data agar aplikasi kita terlindungi dari serangan XSS, terutama ketika kita menampilkan data dari pengguna aplikasi.

Komentar

Seperti halnya html yang memiliki tag khusus untuk komentar, kita juga bisa memberi komentar pada view yang berguna untuk dokumentasi atau catatan sementara. Untuk melakukannya kita gunakan sintak blade

```
{{-- komentar di sini --}}
```

Dengan begitu tulisan “komentar di sini” tidak akan muncul ke layar pengguna aplikasi.

Control Structure

@if

Blade memiliki fitur-fitur yang sering dipakai ketika mengembangkan sebuah aplikasi. Salah satunya adalah if statement, dengan blade kita bisa menuliskan kode kondisional dengan lebih singkat dan rapi.

```
@if($showSidebar === "left")
{{-- munculkan sidebar kiri --}}
<div> SIDEBAR KIRI! </div>

@elseIf($showSidebar === "right")
{{-- munculkan sidebar kanan --}}
<div> SIDEBAR KANAN! </div>

@else
{{-- jangan munculkan sidebar --}}
<div> TIDAK ADA SIDEBAR DITAMPILKAN </div>
@endif
```

Kode di atas tidaklah jauh berbeda dengan if statement pada PHP akan tetapi penulisannya lebih memudahkan karena tidak perlu kurung kurawal di setiap block kode statement. Jika nilai `$showSidebar` bernilai “left” maka tampilkan sidebar kiri, dan jika `$showSidebar` bernilai “right” tampilkan sidebar kanan, jika `$showSidebar` bernilai bukan “left” atau “right” tidak ada sidebar yang ditampilkan.

> **Perhatikan!** kode yang diawali dengan “@” seperti @if, @endif dll disebut dengan “directive”.

@unless

`@unless` akan mengeksekusi kode block setelahnya, kecuali criteria yang dijadikan parameter terpenuhi.

```
@unless( $status == “keren” )
    Kamu belum keren
@endunless
```

Kode blade di atas akan selalu memunculkan “Kamu belum keren” di layar kecuali nilai `$status` adalah “keren”. Akhiri blok kode `@unless` dengan `@endunless`.

@switch

Seperti `switch` pada PHP directive `@switch` digunakan untuk menuliskan kode kondisional dengan banyak kriteria.

```
@switch($vote_status)
    @case(0)
        <div> Tidak Setuju </div>
        @break

    @case(1)
        <div> Setuju </div>
        @break

    @default
        <div> Anda belum melakukan voting </div>
@endswitch
```

Mengecek apakah sebuah variabel tersedia

Gunakan `@isset` untuk mengecek apakah sebuah variabel telah didefinisikan dan tidak bernilai NULL.

```
@isset($productList)
// kode view untuk menampilkan daftar produk jika $productList telah
// didefinisikan dan tidak bernilai NULL
@endisset
```

Akhiri blok kode `@isset` dengan `@endisset`.

Mengecek apakah data kosong

Gunakan `@empty` untuk mengecek apakah sebuah variabel bernilai kosong (empty).

```
@empty($productList)
// kode jika $productList empty
@endempty
```

Akhiri blok kode `@empty` dengan `@endempty`.

Mengecek apakah pengguna sudah login

Gunakan @auth untuk mengecek apakah pengguna yang sedang mengakses sudah login ke aplikasi Laravel kita.

```
@auth
   Selamat datang!
@endauth
```

Akhiri `@auth` dengan `@endauth`. Kode yang ada di dalam block `@auth` hanya akan ditampilkan jika user telah login ke aplikasi.

Mengecek apakah pengguna belum login

Gunakan `@guest` untuk mengecek apakah pengguna yang sedang mengakses tidak login ke aplikasi. Kode yang ada didalam block `@guest` hanya akan ditampilkan ke user yang belum login.

```
@guest
   Hai, silahkan login terlebih dahulu jika sudah punya akun.
@endguest
```

Akhiri `@guest` dengan `@endguest`

Penjelasan: Kita melakukan switch terhadap variabel `@vote_status`, apabila variabel tersebut bernilai 0, maka tampilkan div dengan teks “Tidak Setuju”, jika variabel `@vote_status` bernilai 1 tampilkan div dengan teks “Setuju” dan jika selain itu, defaultnya menampilkan div dengan teks “Anda belum melakukan voting”.

Menampilkan Kumpulan Data

Menampilkan kumpulan data dilakukan dengan perulangan dari variabel yang berupa array. Digunakan misalnya bila kita memiliki variabel `$productList` yang berisi 10 data product. Lalu kita ingin menampilkan masing-masing produk sebagai item.

@foreach

Pertama kita bisa gunakan directive `@foreach`. Directive ini membutuhkan variabel bertipe iterable, misalnya array dan minimal alias untuk masing-masing item didalamnya.

```
@foreach($productList as $product)
// sekarang kita punya akses ke masing-masing produk sebagai $product
// di dalam $productList

// tampilkan masing-masing nama produk dari array $productList
{{ $product->name }}
@endforeach
```

Akhiri directive `@foreach` dengan `@endforeach`

@forelse

Directive ini hampir mirip dengan `@foreach`, bedanya kita bisa menyisipkan directive `@empty` di tengah-tengah blok kode untuk menampilkan sesuatu jika ternyata arranya kosong.

```
@forelse($productList as $product)
// sekarang kita punya akses ke masing-masing produk sebagai $product
```

```
// di dalam $productList

// tampilkan masing-masing nama produk dari array $productList
{{ $product->name }}

@empty
// jika kosong
<div>Belum ada produk</div>


@endforelse
```

Akhiri directive `@forelse` dengan `@endforelse`.

@for

For digunakan untuk melakukan perulangan seperti halnya for pada PHP.

```
@for($i = 0; $i < 10 ; $++)
   // kita akses nilai $i di sini
   // nilai $i bernilai 1 sampai 10;
@endfor
```

@while

Seperti halnya for, directive `@while` juga memiliki fungsi yang sama dengan while pada PHP biasa.

Nested Loop

Nested loop merupakan perulangan di dalam perulangan. Hal ini seringkali terjadi ketika data yang dilakukan perulangan memiliki property berupa array yang bisa dilakukan perulangan lagi.

```
@foreach($productList as $product)
    // nested tingkat 1 (parent)
   {{$product->nama}}

   Kategori:
   @foreach($product->categories as $category)
      // nested selanjutnya
      {{$category->name}}
   @endforeach
@endforeach
```

Loop Variable

Ketika kita melakukan operasi perulangan (looping), Blade menyediakan variabel yang bisa diakses dalam setiap perulangan (iteration). Variabel tersebut antaralain

Variable	Penjelasan
$loop->index	Index perulangan saat ini. Mulai dari 0
$loop->iteration	Perulangan keberapa saat ini. Mulai dari 1
$loop->remaining	Berapa perulangan lagi sampai perulangan berhenti
$loop->count	Berapa jumlah data yang dilakukan perulangan. Bernilai sama seperti panjang array.
$loop->first	Mengecek apakah ini perulangan pertama
$loop->last	Mengecek apakah ini perulangan terakhir
$loop->depth	Tingkat nested loop. Apabila merupakan nested loop.
$loop->parent	Ketika menggunakan nested loop. Akses parent loop.

@php

Suatu ketika, kita mungkin perlu menyisipkan kode PHP seperti biasa pada template blade. Untuk melakukannya, kita bisa menggunakan directive @php seperti ini

```
@php
   //kode PHP di sini
@endphp

Akhiri directive @php dengan @endphp
```

Blade Layout, Section & Component

Dari semua fitur Blade yang telah kita bahas, ada tiga fitur yang belum kita bahas. Dan tiga fitur ini merupakan fitur yang penting ketika menggunakan Blade, yaitu Layout, Section dan Component.

Layout & Section

Layout digunakan untuk membuat view master yang akan selalu ditampilkan oleh view-view child yang menggunakannya. Dalam sebuah layout kita bisa memberikan tempat-tempat yang bisa digunakan oleh child view. Tempat-tempat tersebut adalah section. Misalnya, dalam layout utama, kita definisikan section sidebar, main_content, dan footer. Selanjutnya, setiap child view yang menggunakan layout utama dapat menempatkan kode view di masing-masing section yang tersedia di layout utama.

File 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>
```

Pada layout view di atas, setiap kode html akan digunakan oleh child view layout tersebut tanpa harus menulisnya lagi. Dengan demikian kita tidak perlu mendefinisikan tag html, head, title, dll pada tiap-tiap view. Dan dari layout view tersebut dapat kita baca bahwa layout itu menyediakan section `sidebar`, `title`, dan `content` yang didefinisikan menggunakan directive `@yield` dan `@section`.

@yield

Dengan directive `@yield("nama_section")` sebuah layout mengharapkan konten dari child view. Untuk mengisinya, child view akan menggunakan `@section` dengan parameter nama yang sama dengan `@yield` seperti ini `@section("nama_section")`.

@section

Directive `@section` digunakan selain untuk mendefinisikan sebuah section, juga bisa untuk mengisi section yang diharapkan oleh parent view / layout melalui `@yield`.

@parent

Dalam child view kita bisa menampilkan juga konten yang ada pada parent dalam section tertentu, hal tersebut dilakukan dengan directive `@parent`.

@extends

Extends digunakan pada setiap child view yang ingin menggunakan sebuah view sebagai parent / layout. Mari kita coba bahas semua directive di atas dengan contoh kode.

File resources/views/layouts/app.blade.php

```
<html>
    <head>
        <title>App Name - @yield('title')</title>
    </head>
    <body>
        @section('navbar')
          Navbar dari Layout
          <hr>
        @show

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

```
    </body>
</html>
```

Penjelasan kode: Ini merupakan kode yang kita jadikan sebagai layout global aplikasi Laravel kita. Layout tersebut mengharapkan title, sidebar, dan content. Kita akan mengisinya dari child view yang mengekstend layout ini.

File resources/views/pages/about.blade.php

```
@extends("layouts.app")

@section("title")
    Tentang Kami
@endsection

@section("sidebar")
    @parent
     Sidebar dari halaman tentang kami
@endsection

@section("content")
    Kami adalah Fullstack Developer yang penuh passion untuk memecahkan
masalah yang Anda miliki melalui aplikasi yang kami bangun!
@endsection
```

Penjelasan Kode: File ini pertama melakukan @extends("layouts.app") untuk menjadikan file view `resources/views/layouts/app.blade.php`. Kemudian kita konten untuk section title dengan "Tentang Kami" yang akan direnderer sebagai `<title>` App Name - Tentang Kami `</title>` sesuai dengan file layout. Kemudian kita juga mengisi konten untuk section sidebar, akan tetapi kita menggunakan directive @parent agar selain menampilkan kalimat "Sidebar dari halaman tentang kami" juga menampilkan konten sidebar dari parent view / layout yaitu "Sidebar utama". Setelah itu kita juga isi konten untuk section bernama "content" dengan tulisan "Kami adalah Fullstack Developer dst..."

Component

Component berfungsi untuk membuat view yang dapat kita gunakan berulang kali. Berbeda dengan layout yang bertindak sebagai master yang dapat digunakan berulang, Component justru kebalikannya, yaitu ibarat child view yang bisa kita pakai di view lain yang membutuhkannya.

Misalnya, dalam pengembangan sebuah aplikasi kita akan membutuhkan view untuk alert. Berfungsi untuk memberikan notifikasi kepada pengguna aplikasi terkait informasi, peringatan ataupun pesan error. Alert tentu saja akan digunakan berulangkali di aplikasi, bukan? Oleh karena itu kita bisa membuatnya sebagai component yang bisa digunakan di view lainnya.

File `resources/views/components/alert.blade.php`

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

Penjelasan kode: Kode di atas akan kita jadikan sebagai component, untuk melakukannya kita perlu menampilkan variabel `$slot`, variabel ini dari mana? Dari view lainya yang ingin menggunakan component ini.

Misalnya, sebuah view ingin menggunakan component alert tersebut. Seperti di bawah ini

File `resources/views/about.blade.php`

```
@extends('layouts.app')
// kode…

@component("alert")
     <b>Tulisan ini akan mengisi variabel $slot</b>
@endcomponent
```

Penjelasan kode: File view about.blade.php di atas memanfaatkan component alert, caranya adalah dengan menuliskan directive `@component("alert")`. Kemudian, teks yang ada di block component itulah yang akan mengisi variabel `$slot` yang ditulis pada alert.blade.php.

Latihan praktik

1. Latihan layout & section Buat file layout di `resoureces/views/layouts/app.blade.php` lalu isikan kode berikut:

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

        <!-- BOOTSTRAP CSS -->
        <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/4.0.0/css/bootstrap.mi
n.css" integrity="sha384-
Gn5384xqQ1aoWXA+058RXPxPg6fy4IWvTNh0E263XmFcJlSAwiGgFAW/dAiS6JXm"
crossorigin="anonymous">

    </head>
    <body>
        @section('navbar')
        Navbar dari Layout
        <hr>
        @show

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

```
    </body>
</html>
```

Setelah itu buat file view yang akan menggunakan layout di atas pada `resources/views/child.blade.php` dan isikan kode berikut:

```
@extends("layouts.app")

@section("title")
    Aplikasi Toko Online
@endsection


@section("content")
    Konten dari child view
@endsection
```

Setelah itu buka file `routes/web.php` dan cukup tambahkan route view pada route group "latihan" yang pernah kita buat pada praktik bab Eloquent & Model.

```
Route::group(["prefix" => "latihan"], function(){
    // KODE KODE ROUTE SEBELUMNYA
    // ......

    // Tambahkan ini
    Route::view("layouts", "child");
});
```

Dengan route tersebut maka kini kita bisa mengakses halaman `http://toko-online.test/latihan/layouts` dan kamu akan melihat konten yang berasal dari view `resources/views/layouts/app.blade.php` dan konten dari `resources/views/child.blade.php`.

Jika kamu sudah berhasil menampilkannya maka ini adalah bukti bahwa `resources/views/child.blade.php` menggunakan layouts `resources/views/layouts/app.blade.php` dan mengisi section yang disediakan di layouts yaitu `title` dan `content`.

2. Latihan component

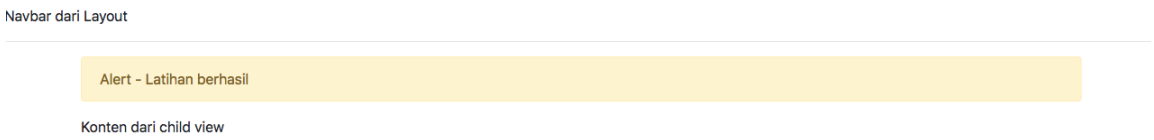
    - Buat file view yang akan kita gunakan sebagai component pada path `resources/views/alert.blade.php` lalu isikan kode berikut ini:

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

- Buka kembali file `child.blade.php` lalu kita tambahkan kode untuk menggunakan component alert yang baru kita buat

```
@component("alert")
 Alert - Latihan berhasil
@endcomponent
```

Sehingga jika kamu buka kembali `http://toko-online.test/latihan/layouts` maka kamu akan menjumpai alert telah berhasil ditampilkan seperti ini:



Tapi tunggu dulu, alert tersebut selalu menampilkan warna kuning atau type warning dari bootstrap. Gimana caranya kalo kita ingin mengganti ke type lain misalnya "success". Mari kita ubah kode component kita supaya support fitur tersebut.

- Buat component alert dapat diubah tipe alertnya. Buka file `alert.blade.php` lalu ubah kodenya menjadi seperti ini:

```
@if(isset($type))
    <div class="alert alert-{{$type}}">
        {{$slot}}
    </div>
@else
    <div class="alert alert-warning">
        {{$slot}}
    </div>
@endif
```

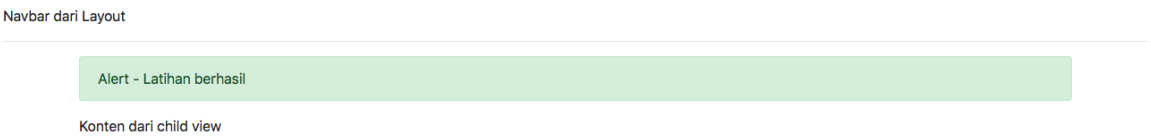
Penjelasan Kode: Kini kita memperkenalkan variabel tambahan yaitu $type. Variabel ini bisa digunakan nantinya oleh `child.blade.php` seperti ini:

```
@component("alert", ["type"=>"success"])
    // ...
@endcomponent
```

- Ubah kembali file `child.blade.php` sehingga memanfaatkan component alert untuk membuat alert success seperti ini:

```
@extends("layouts.app")

@section("title")
    Aplikasi Toko Online
@endsection


@section("content")
@component("alert", ["type"=>"success"])
    Alert - Latihan berhasil
@endcomponent
    Konten dari child view
@endsection
```

Lalu buka kembali `http://toko-online.test/latihan/layouts` maka kini warna alert akan berubah menjadi hijau.



Dengan demikian kini kita bisa menggunakan component alert di view manapun yang kita inginkan, dan kita bisa menyesuaikan tipe alertnya "success", "warning", "info" atau "danger" tanpa perlu membuat component sendiri untuk masing-masing tipe. Component merupakan cara yang baik untuk membuat view yang bisa digunakan berulangkali.

3. Latihan data pagination

Pada bab Eloquent dan Model kita telah belajar tentang melakukan pagination menggunakan model dengan fungsi `simplePaginate()` dan `paginate()`. Di situ juga telah dijelaskan perbedaan keduanya. Nah sekarang kita akan belajar untuk menampilkan pagination di view.

- Kita akan melanjutkan latihan praktik menggunakan `CategoryController` dan view `kategori/index`.
- Buka `CategoryController` dan pastikan action `index` memiliki kode berikut:

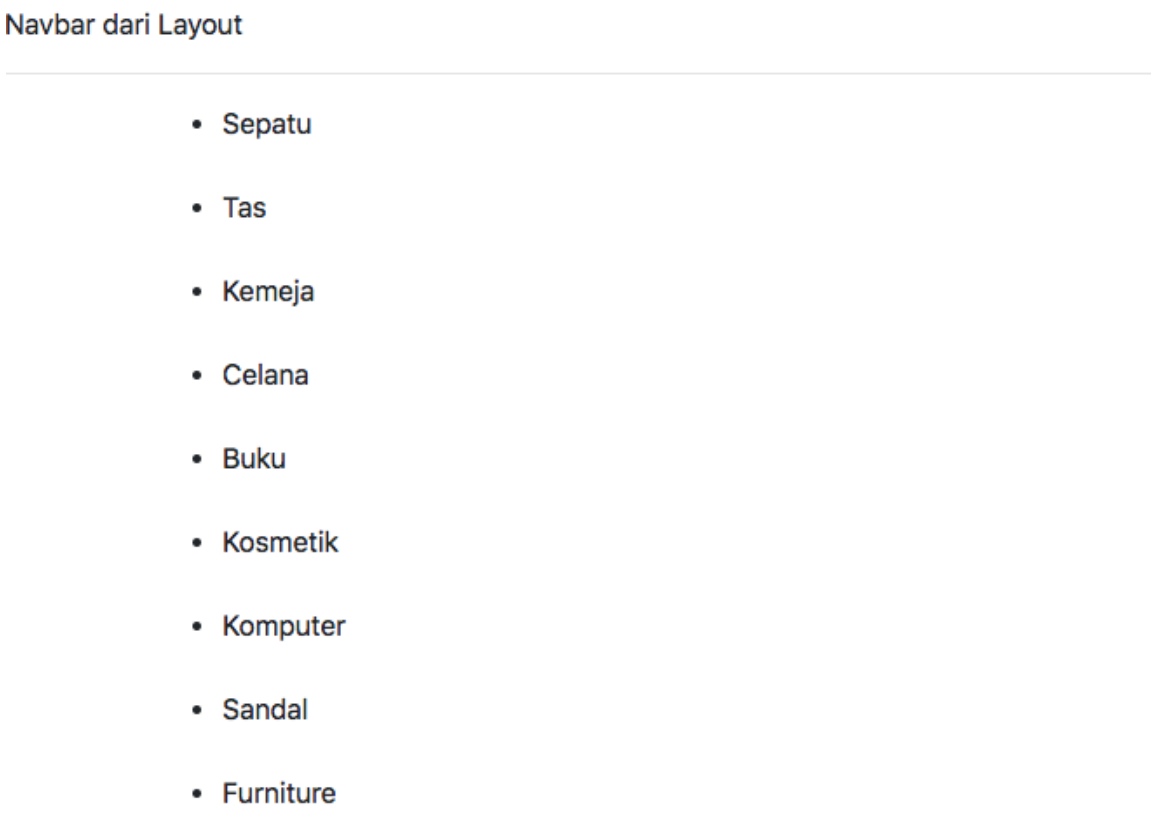
```
public function index(){

    $daftar_kategori = \App\Category::all();

    return view("kategori.index", ["daftar_kategori" =>
$daftar_kategori]);

}
```

- Setelah itu buka view `resources/views/kategori/index.blade.php` dan ubah kodenya agar menjadi seperti ini:

```
    @extends("layouts.app")

    @section("content")

        <ul>
            @foreach($daftar_kategori as $kategori)
                <li>{{ $kategori->name }}</li>
                <br/>
            @endforeach
        </ul>

    @endsection
```

Penjelasan Kode: Kita kembali menggunakan layout app ("layouts.app") yang terletak di `resources/views/layouts/app.blade.php` lalu mengisi section `content` dengan looping variable `$daftar_kategori` yang dikirim oleh action `index` di `CategoryController` untuk menampilkan masing-masing nama kategori.

Jika kamu akses kembali `http://toko-online.test/latihan/kategori/all` maka kamu akan menjumpai tampilan seperti ini:

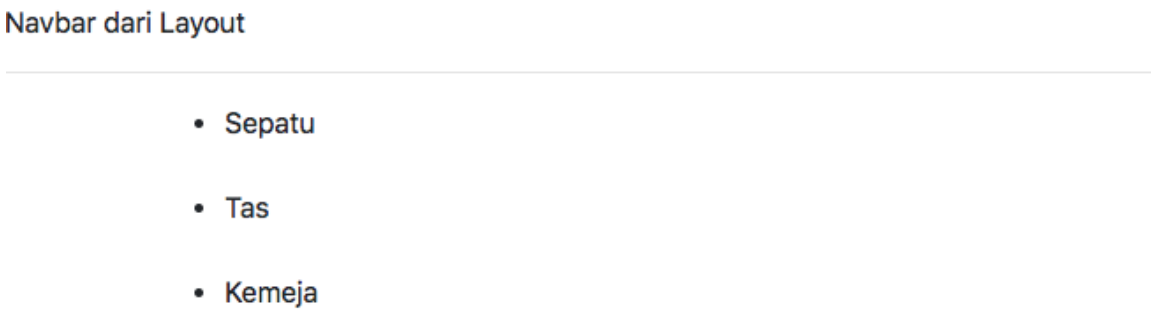


- Selanjutnya kita ingin menampilkan per halaman adalah 3 item. Maka ubah kembali action `index` pada `CategoryController` menjadi seperti ini:

```
public function index(){
    $daftar_kategori = \App\Category::paginate(3);

    return view("kategori.index", ["daftar_kategori" =>
$daftar_kategori]);
}
```

Penjelasan Kode: Kita mengubah `\App\Category::all()` menjadi `\App\Category::paginate(3)` untuk menampilkan 3 item kategori sekali tampil (per halaman)

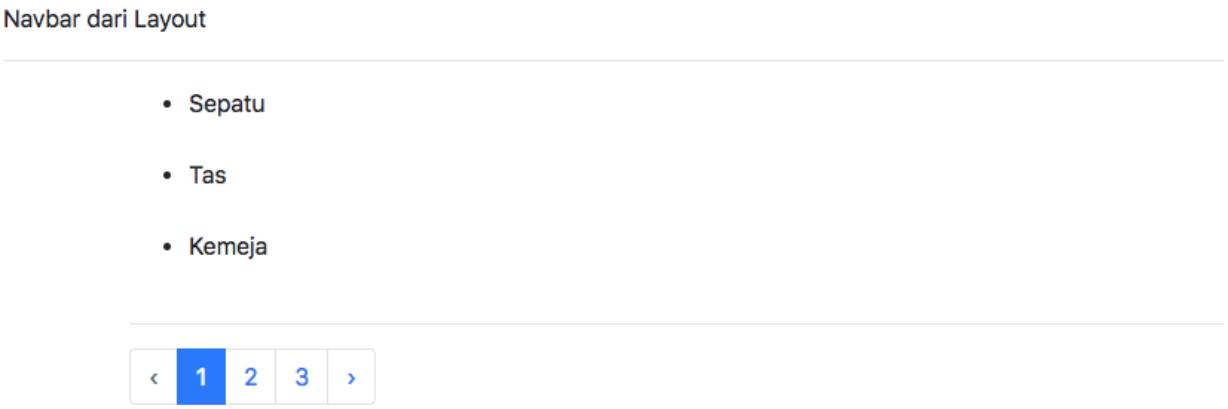
Buka kembali `http://toko-online.test/latihan/kategori/all` kamu kini hanya melihat 3 kategori teratas seperti ini:



- Nah kini kita ingin menampilkan link pagination di view kita. Buka file `resources/views/kategori/index.blade.php` dan tambahkan kode untuk menampilkan pagination sehingga secara keseluruhan kode viewnya menjadi seperti ini:

```
@extends("layouts.app")

@section("content")

<ul>
    @foreach($daftar_kategori as $kategori)
        <li>{{ $kategori->name }}</li>
        <br/>
    @endforeach
</ul>

<hr>
<!-- INI MERUPAKAN KODE UNTUK MENAMPILKAN PAGINATION -->
{{$daftar_kategori->links()}}

@endsection
```

Dengan kode di atas maka jika kamu buka kembali `http://toko-online.test/latihan/kategori/all` kamu akan memiliki link pagination seperti ini:



Dan kamu boleh mengeklik ke halaman 2 atau 3 semuanya sudah berfungsi! Keren!

Kesimpulan

Kita telah belajar tentang view dan templating enginnya yaitu Blade. Ada banyak fitur Blade yang memudahkan kita menulis kode tampilan aplikasi. Tidak hanya html tapi ada banyak directive yang sangat membantu kita sebagai developer.

Selain itu juga kita belajar tentang layouts dan section dan juga bagaimana membuat komponen yang bisa digunakan berulangkali dan dikonfigurasi dengan component. Pada akhirnya kita juga mempraktikan langung apa yang kita pelajari.

Kini kamu sudah bisa mulai untuk membuat aplikasi sederhana berbasis CRUD. Setelah ini kita akan belajar tentang Relationship. Kita akan belajar bahwa membuat relationship dan mendapatkan data relation antar model itu sangat menyenangkan di Laravel.

Relationship

Jika kamu pernah mengembangkan sebuah aplikasi sebelumnya meskipun tanpa framework saya yakin kamu pernah membuat relationship antar tabel. Dan bahkan kamu mungkin berpikir melakukan query relationship dan SQL command caranya cukup panjang. Tapi jangan kuatir, dengan Laravel semua itu akan menjadi mudah.

Dalam membuat tabel kita tidak mungkin menyimpan semua data hanya dalam satu tabel terlebih jika terdapat banyak entitas. Misalnya kita memiliki tabel-tabel berikut ini:

- user
- order
- book

Dimana setiap `user` bisa memiliki banyak `order` sementara itu `order` hanya boleh dimiliki oleh satu `user`, dengan demikian bisa dikatakan relationship antara tabel user dengan tabel `order` adalah `one-to-many`.

Sementara itu setiap `order` berisi banyak `book` dan setiap `books` bisa juga berada pada `order` lainnya. Berarti relationship antara `order` dengan `book` merupakan `many-to-many` relationship.

Baik `one-to-one`, `one-to-many` atau `many-to-many` relationship, semua itu bisa kita lakukan di Laravel dengan cara yang elegan. Relationship tersebut akan dilakukan dengan Model.

Sebelum kita mempelajari bagiamana melakukan setup relationship di Laravel, kita akan membahas relationship secara umum terlebih dahulu. Terutama dari sisi database, hal ini penting bagi kita agar kita bisa memahami setup relationship dengan Eloquent.

Pengenalan relationship

One to one

`One to one` relationship terjadi jika data A hanya boleh memiliki satu data B (A has one B) dan data B hanya boleh dimiliki oleh data A (B belongs to A).

Dalam database, kita cukup memberikan foreign key di tabel B yang akan diisi dengan ID dari data di tabel A. ID tersebut merepresentasikan data A yang memiliki data di tabel B. Kita akan menggunakan ilustrasi relation antara tabel `user` dengan tabel `phone`. Setiap user hanya boleh memiliki 1 nomor handphone dan 1 nomor handphone hanya boleh dimiliki oleh 1 user. Maka tabel keduanya adalah seperti ini:

users
id: Int {PK}
username: String
fullname: String
city: Text

Dan tabel nomor hape seperti ini:

phones
id: Int {PK}
phone_num: String

Kedua tabel tersebut belum bisa dikatakan memiliki relation, agar `one-to-one` relationship terbentuk maka kita tambahkan 1 field di `phones` sebagai foreign key yang mereferensikan ID di tabel `users` dan harus memiliki constraint unique {U} seperti ini:

phones
id: Int {PK}
phone_num: String
user_id: Int {FK} {U}

Field `user_id` akan diisi dengan user id dari tabel `users` sehingga kini kedua tabel tersebut bisa memenuhi syarat `one-to-one` relationship.

Coba kita isi masing-masing tabel seperti ini misalnya:

users

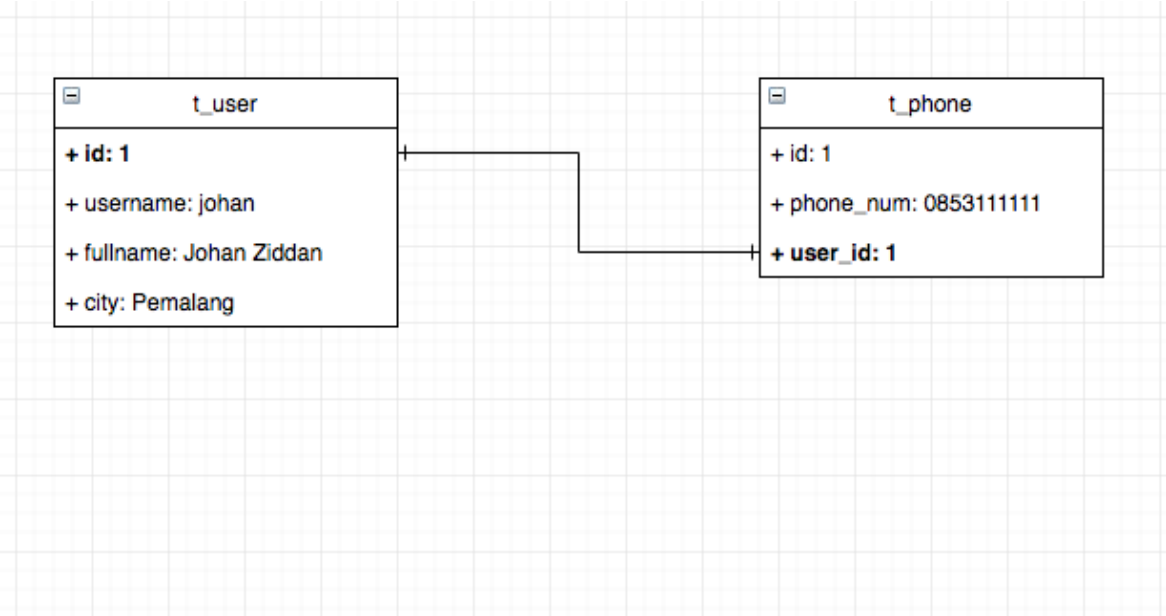
id	**username**	**fullname**	**city**
1	johan	Johan Ziddan	Pemalang
2	nadia	Nadia Nurul Mila	Jakarta
3	hafidz	Hafid Mukhlasin	Jakarta
4	fadhilah	Fadilah Rimandani	Jakarta
5	azam	Muhammad Azamuddin	Jakarta
6	adi.lukman	Adi Lukmanul	Garut

phones

id	**phone_num**	**user_id**
1	0873111111	1
2	0873222222	5

Dengan contoh data di atas kita dapat simpulkan bahwa user dengan `id` 1 yaitu Johan Ziddan memiliki nomor handphone dengan nomor 0873111111 dan user dengan `id` 5 yaitu azam memiliki nomor handhpone 0873222222.

Field `user_id` pada tabel `phones` bersifat unique sehingga tidak boleh ada nilai yang sama.



One to many

Untuk one-to-many struktur tabelnya hampir sama, misalnya kita memilik tabel `users` dan `orders`. Struktur tabel untuk masing-masing tabel dengan relation `one-to-many` adalah sebagai berikut:

users
id: Int {PK}
username: String
fullname: String
city: Text

Dan untuk tabel `orders` adalah sebagai berikut:

orders
id: Int {PK}
total_price: Int
invoice_number: String
status: Enum
user_id: Int {FK}

Struktur tabel di atas sudah memenuhi agar `users` memiliki relation `one-to-many` dengan `orders`. Perhatikan `orders` memiliki foreign key `user_id` yang mereferensikan `users` field `id`.

Sepintas struktur tabel `one-to-many` kok mirip dengan `one-to-one`, yup, bedanya foreign key pada `one-to-many` tidak memiliki constraint unique. Sehingga dibolehkan ada nilai yang sama untuk field `user_id` oleh karenanya relationshipnya disebut `one-to-many`. Sampai di sini cukup jelas bukan?

Kita coba buat kembali visualisasi dari data di kedua tabel di atas seperti ini:

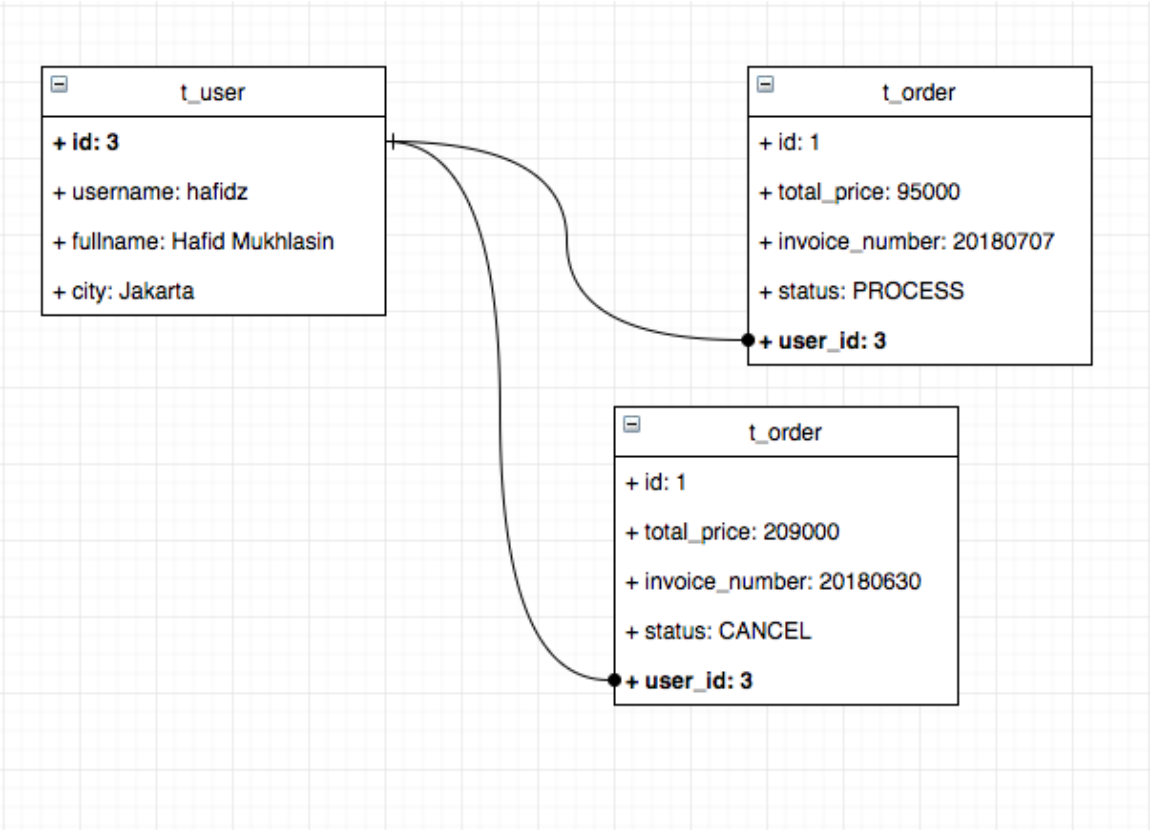
users

id	username	fullname	city
1	johan	Johan Ziddan	Pemalang
2	nadia	Nadia Nurul Mila	Jakarta
3	hafidz	Hafid Mukhlasin	Jakarta
4	fadhilah	Fadilah Rimandani	Jakarta
5	azam	Muhammad Azamuddin	Jakarta
6	adi.lukman	Adi Lukmanul	Garut

orders

id	total_price	invoice_number	status	user_id
1	95000	20180707	PROCESS	3
2	124000	20180705	DELIVER	4
3	209000	20180630	CANCEL	3
4	100000	20180629	FINISH	5

Perhatikan di tabel `orders` terdapat dua order yang memiliki `user_id` bernilai 3, itu berarti user dengan `id` 3 yaitu Hafidz memiliki dua order yaitu order dengan `id` 1 dan 3. Satu user memiliki lebih dari 1 order (user has many order), sementara itu masing-masing order hanya bisa dimiliki oleh 1 user (order belongs to one user). ini berarti relationship `one-to-many` terpenuhi antara kedua tabel ini.



Many to many

Many to many relationship sedikit berbeda struktur tabelnya dibandingkan dua relationship yang telah kita pelajari. Ini karena data A boleh memiliki banyak data B (A has many B), dan sebaliknya data B boleh dimiliki oleh banyak data A (B belongs to many A). Kita tidak bisa menghubungkan keduanya hanya dengan tabel A dan tabel B, harus ada tabel lain yang menghubungkan keduanya, tabel tersebut kita sebut dengan istilah `pivot table` atau table penghubung.

Kita ambil sebuah ilustrasi dengan table `orders` dan tabel `books`, relationship keduanya haruslah `many-to-many` karena sebuah order boleh memiliki banyak buku yang diorder, dan buku bisa masuk ke banyak order. Maka mari kita buat struktur tabel untuk masing-masing tabel kurang lebih seperti ini:

orders
id: Int {PK}
total_price: Int
invoice_number: String
status: Enum
user_id: Int {FK}

dan tabel `books`

books
id: Int {PK}

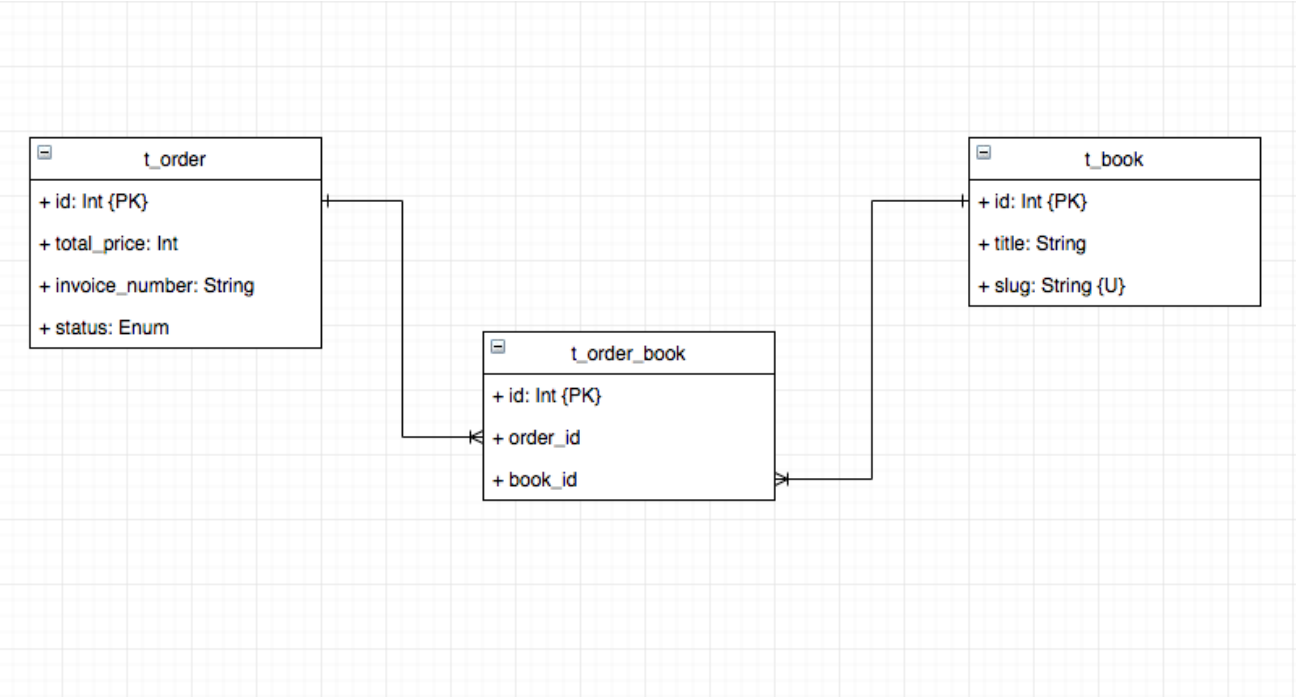
books
title: String
slug: String {U}

Sekarang mari kita berpikir, apakah struktur di atas cukup untuk membuat relationship `many-to-many`? Tentu tidak? apakah kita perlu menambahkan foreign key `order_id` misalnya di `books`? Tentu tidak juga karena ini bukan `one-to-many`.

Caranya adalah dengan membuat 1 tabel lagi sebagai penghubung kedua tabel di atas, mari kita sebut dengan `book_order` dengan struktur seperti ini:

book_order
id: Int {PK}
order_id: Int {FK}
book_id: Int {FK}

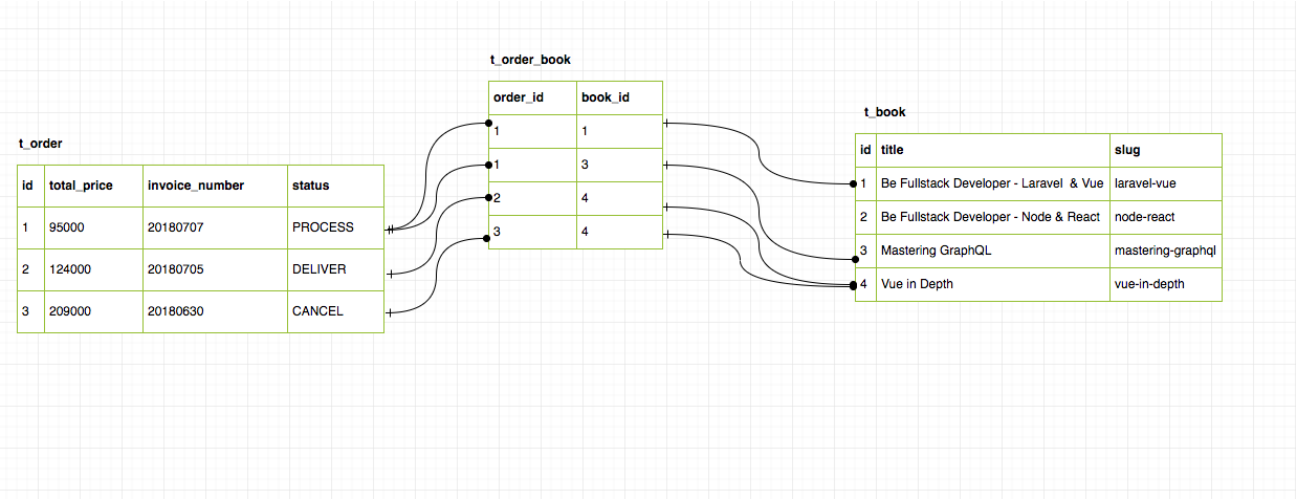
Apakah sekarang kamu sudah menangkapnya? Jika belum mari lihat visualisasi dengan diagramnya seperti ini:



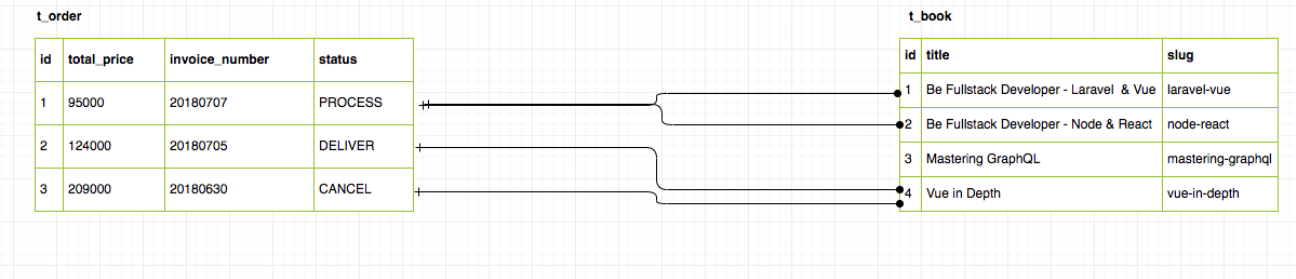
Penjelasan gambar: Diagram di atas merupakan gambaran dari relation `one-to-many` antara `orders` dengan `book_order` melalui pivot table `book_order`. Jika kita lihat dari diagram tersebut sebetulnya bisa kita pecah seperti ini:

- `orders` memiliki banyak (has many) `book_order`
- `books` juga memiliki (has many) `book_order`
- `book_order` hanya bisa dimiliki oleh 1 (belongs to one) `orders` melalui field `order_id`
- `book_order` hanya bisa dimiliki oleh 1 (belongs to one) `books` melalui field `book_id`

Itu berarti hubungan antara `orders` dengan `book_order` adalah `one-to-many` begitu juga antara `books` dengan `book_order`. Sehingga bila digabungkan ketiganya yang terjadi adalah `orders` bisa memiliki relation `many-to-many` dengan `books`.



Dari visualisasi di atas kita bisa coba untuk berpikir menghilangkan table penghubung dalam penghilatan kita sehingga akan terlihat lebih jelas seperti apa relation `many-to-many` antara table `orders` dengan `books` seperti di bawah ini:



Dengan melihat gambar di atas kini kita tahu bahwa order dengan `id` 1 memiliki 2 `book` yaitu buku dengan `id` 1 dan 2. Sementara itu `book` dengan `id` 4 dimiliki oleh 2 `order` yaitu `order` dengan `id` 2 dan 3. Terbukti bahwa relation antara tabel `orders` dengan `book` adalah `many-to-many` melalui tabel penghubung `books_order`

Apa yang kita pelajari?

- one-to-one membutuhkan foreign key dengan constraint unique
- one-to-many membutuhkan foreign key TANPA constraint unique
- many-to-many membutuhkan tabel penghubung

Relationship di Laravel

Overview

Materi sebelumnya yaitu pengenalan relationship sangat penting untuk dipahami agar kita lebih mudah memahami bagaiamana menggunakan fitur relationship di Laravel.

Karena kita menggunakan Eloquent untuk melakukan interaksi dengan database, maka kita memerlukan setup di model kita agar bisa bekerja dengan relationship.

Secara garis besar langkah-langkah yang akan kita lakukan untuk bisa menggunakan relationship di Laravel adalah sebagai berikut:

1. membuat struktur tabel untuk relationship

   Untuk menyiapkan struktur tabel kita bisa menggunakan migration kembali, dan kita akan belajar bagaimana melakukanya. Di bab sebelumnya kita telah praktik menggunakan migration akan tetapi belum ada pembahasan bagaimana menyiapkan relationship dengan migration.

2. melakukan konfigurasi terhadap model model yang terkait

   Setelah struktur tabel siap untuk relationship, maka langkah selanjutnya adalah kita konfigurasi model-model kita supaya bisa kita gunakan fitur-fitur Eloquent terkait relationship

Seperti apa sih gambaran seandainya model model kita sudah dikonfigurasi untuk relationship. Supaya kamu lebih paham mengapa kita perlu melakukan konfigurasi di atas, mari kita ambil contoh di pengenalan relationship. Misalnya `orders` diwakili oleh sebuah model bernama `Order` dan tabel `books` diwakili oleh model bernama `Book`. Maka kita ketahui bersama bahwa relasi kedua model tersebut sesuai contoh sebelumnya adalah `many-to-many` relationship.

Jika tanpa menggunakan model, kita perlu menuliskan kode SQL yang panjang seandainya ingin mendapatkan buku dari sebuah order. Tapi dengan model yang sudah kita konfigurasi relationshipnya maka kita bisa melakukan hal-hal ini:

- Mendapatkan buku-buku dari sebuah order

```
Order::find(1)->books;
```

- Mendapatkan order-order yang memasukan buku tertentu

```
Book::find(3)->orders;
```

- Menambahkan buku ke dalam order

```
// order dengan id 1
$order = Order::find(1);

// tambahkan buku dengan id 1,3,4
$order->attach([1,3,4]);
```

Penjelasan kode: Kita menambahkan 3 buku sekaligus yaitu `Book` dengan `id` 1,3 dan 4 ke dalam `Order` dengan `id` 1.

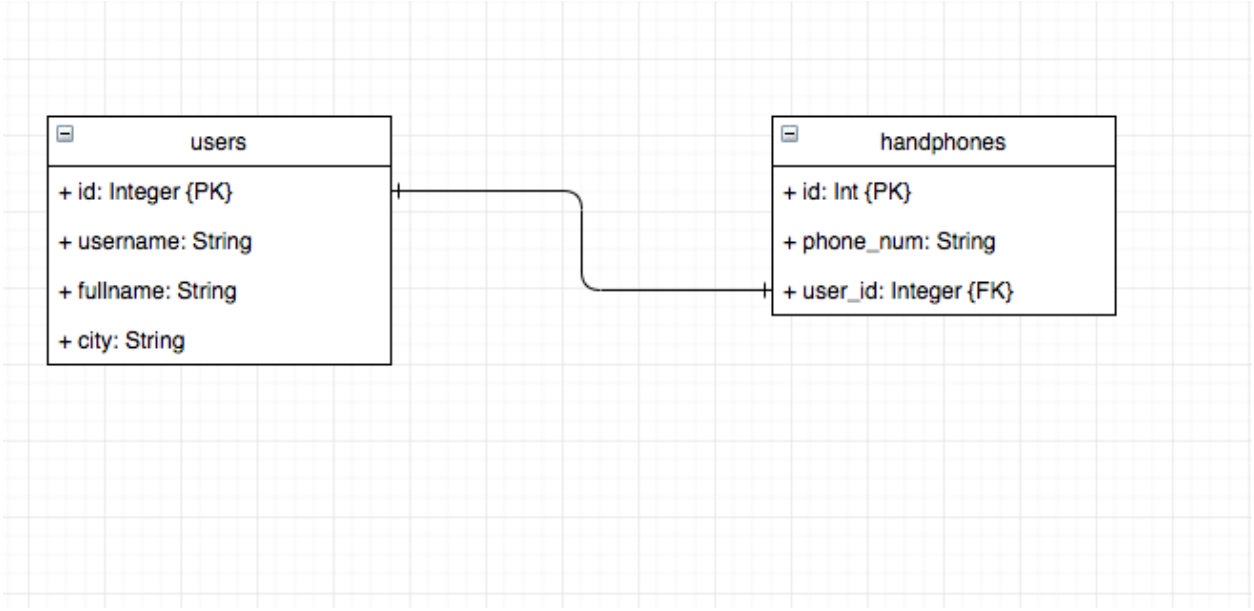
Begitu mudah dan ringkas dibaca bukan? Nah tentu masih banyak lagi fitur-fitur terkait relationship yang akan segera kita pelajari. Paling tidak 3 contoh sederhana di atas membuat kamu paham manfaat dari kita melakukan konfigurasi relationship di Laravel.

Dan sebelum kita bisa menggunakan fitur-fitur Eloquent relationship yang ada, mari kita belajar untuk mempersiapkan tabel dan model kita agar mendukung relationship di Laravel.

Struktur tabel dan model

One to one relationship

Kita akan menggunakan pemisalan antara model `User` dan `Handphone`. Kedua tabel tersebut secara berurutan mewakili tabel `users` dan `handphones` di database. Kita akan menggunakan struktur tabel seperti ini:



Menyiapkan tabel dengan migration

```
function up(){
  Schema::create("users", function(Blueprint $table){
    $table->primary("id");
    $table->string("username")->unique();
    $table->string("fullname");
    $table->string("city");
  });
}
```

Penjelasan kode: Kode migration untuk membuat struktur tabel users. Tidak ada yang khusus, menggunakan Schema builder yang pernah kita pelajari.

Karena ini konsepnya `one-to-one` tabel `users` ke `handphones` kita perlu membuat relation di migration tabel `handphones`

```
function up(){
  Schema::create("handphones", function(Blueprint $table){
    $table->primary("id");
    $table->string("phone_num")->unique();
    $table->integer("user_id")->unsigned()->unique();
```

```
    $table->foreign("user_id")->references("id")->on("users");
  });
}
```

Penjelasan kode: File migration untuk membuat struktur tabel `handphones`. Perhatikan kita membuat foreign key `user_id` di tabel ini yang mereferensikan field `id` di tabel `users` dengan kode ini:

```
$table->foreign("user_id")->references("id")->on("users")
```

Perlu diketahui juga sebelum kita bisa menggunakan kode di atas, ada hal yang perlu dilakukan yaitu:

1. field `user_id` harus sudah ada terlebih dahulu, makanya jika belum ada kita buat dengan

```
  $table->integer("user_id");
```

2. Field `user_id` harus merupakan `unsigned` oleh karena itu dalam pembuatan kita juga menggunakan method `unsigned()`

```
  $table->integer("user_id")->unsigned();
```

3. Field `user_id` harus memiliki tipe yang sama dengan referensi di tabel tujuan, itu berarti tipe `integer` di field `user_id` harus sama dengan tipe field `id` di tabel `users`. Field `id` di tabel `users` dibuat dengan method `primary()` yang secara otomatis memiliki tipe `integer`, ini berarti sudah sama. Good!

4. Karena kita akan membuat `one-to-one` relationship, maka foreign key `user_id` harus kita tambahkan constraint `unique` dengan method `unique()` seperti ini:

   ```
   $table->integer("user_id")->unsigned()->unique();
   ```

Mendefinisikan di Model

Setelah tabel kita siap untuk keperluan relationship, maka langkah terakhir yang harus kita lakukan adalah konfigurasi dua model yang akan kita hubungkan yaitu model `User` dan model `Handphone`;

Pada model `User` kita perlu menambahkan kode berikut ini:

```
public function handphone(){
  return $this->hasOne("App\Handphone");
}
```

Penjelasan kode: Dengan kode ini kita mengatakan kepada model bahwa model ini (model `User`) `hasOne` atau boleh memiliki satu `Handphone`.

Sehingga nantinya kita bisa mendapatkan handphone yang dimiliki oleh user tertentu dengan cara ini:

```
// cari user dengan ID 1
$user = User::find(1);

// dapatkan model handphone dari user di atas
// akan mengembalikan satu data dengan tipe model Handphone
$handphone = $user->handphone;
```

Kemudian pada model `Handphone` kita perlu menambahkan kode ini:

```
public function user(){
  return $this->belongsTo("App\User");
}
```

Penjelasan kode: Dengan kode ini kita mengatakan kepada model ini (model `Handphone`) `belongsTo` atau dimiliki oleh sebuah model `User`.

Sehingga kita bisa mencari tahu siapa pemiliki nomor handpone tertentu dengan cara ini:

```
// dapatkan model handphone yang memiliki phone_num = 0852111111
$handphone = Handphone::where("phone_num", "0852111111")->first();

// cari model user yang merupakan pemilik no hape di atas
$pemilik = $handphone->user;
```

Query one to one relationship

Query one-to-one relationship

Kita melanjutkan perumpamaan yang kita pakai saat mendefinisikan model one-to-one relationship sebelumnya, yaitu relationship antara model `User` dengan model `Handphone`.

Seperti yang telah kita definisikan bahwa `User` memiliki 1 `Handphone` yang kita definisikan di model dalam method `handphone()`. Untuk mencari handphone dari seorang `User` maka kita cukup gunakan kode semacam ini:

```
// cari user dengan ID 34
$user = User::find(34);

// dapatkan handphone dari user dengan ID 34
```

```
$handphone = $user->handphone;

// dapatkan nomor hape dari model handphone yang baru saja didapatkan
$no_hape_user = $handphone->phone_num;
```

Penjelasan: User dengan `id` 34 misalnya memiliki sebuah handphone, maka bisa diakses dalam model `User` sebagai properti `handphone`

> `$user->handphone` tidak perlu menggunakan `$user->handphone()` karena relation di Laravel merupakan sebuah `dynamic properties` jadi seolah-olah properti bukan method.

Misalnya kita balik, kita mencari handphone terlebih dahulu kemudian ingin mencari pemilik dari nomor tersebut juga bisa, caranya seperti ini:

```
// cari handphone dengan nomor 085211111 di DB dan dapatkan data pertama
`first()`
$handphone = Handphone::where("phone_num", "085211111")->first();

// cari siapa pemilik / user dari nomor handphone tersebut
$pemilik = $handphone->user;

// dapatkan nama lengkap dari pemiliki nomor di atas
$nama_pemilik = $pemilik->fullname;
```

One to many relationship

Kita akan menggunakan contoh sebelumnya, yaitu relation antara `User` dengan `Handphone` bedanya kini kita mengizinkan satu `User` memiliki lebih dari satu `Handphone`.

Menyiapkan tabel dengan migration

One to many relationship memiliki struktur tabel yang mirip dengan `one-to-one` relationship, bedanya apa? sudah kita pelajari di pengenalan relationship, bedanya adalah foreign key tidak memiliki constraint unique. Jadi kita bisa menggunakan contoh migration sebelumnya bedanya kita hapus method `unique()` saat membuat foreign key `user_id` di tabel `handpones`.

Migration `create_handphones_table`

```
function up(){
  Schema::create("handphones", function(Blueprint $table){
    $table->primary("id");
    $table->string("phone_num")->unique();
    $table->integer("user_id")->unsigned();

    $table->foreign("user_id")->references("id")->on("users");
  });
}
```

Sementara itu migration untuk `create_users_table` masih sama seperti pada `one-to-one` relation.

Mendefinisikan di Model

Agar relationship antara model `User` dan `Handphone` menjadi `one-to-many` maka kita tambahkan kode ini di masing-masing model:

Model User

```
public function handphones(){
  return $this->hasMany("App\Handphone");
}
```

Penjelasan kode: Mirip dengan definisi sebelumnya, bedanya kita gunakan `hasMany` bukan `hasOne` karena kita ingin model `User` boleh memiliki lebih dari satu `Handphone`. Selain itu juga kita ubah nama relation dari bentuk tunggal `handphone` menjadi bentuk jamak yaitu `handphones`, perubahan nama ini tidak wajib hanya supaya lebih mudah dipahami bahwa user punya banyak handphone.

Sehingga kita bisa mengakses `Handphones` yang dimiliki oleh user dengan cara ini:

```
// cari user dengan id 1
$user = User::find(1);

// dapatkan handphones dari user di atas
// sekarang bentuknya adalah array of handphone bukan 1 handphone
// jadi tidak hanya 1, tapi bisa lebih dari satu Handphone
$user->handphones;
```

Untuk model `Handphone` tidak ada perubahan dari contoh di `one-to-one` relationship, yaitu konfigurasinya seperti ini:

```
public function user(){
  return $this->belongsTo("App\User");
}
```

Penjelasan kode: Dengan kode ini kita tetap mengatakan bahwa setiap model `Handphone` hanya dimiliki oleh satu `User`. Tidak ada sebuah `Handphone` yang dimiliki oleh beberapa `User`.

Many to many relationship

Kita akan menggunakan contoh pada pengenalan relationship `many-to-many` yaitu antara tabel `orders` dengan tabel `books`. Dimana `orders` diwakili oleh model `Order` dan `books` diwakili oleh model `Book`. Sebuah `Order` bisa memiliki beberapa item `Book` di dalamnya, dan sebaliknya sebuah `Book` bisa masuk ke dalam beberapa `Order`.

Menyiapkan tabel dengan migration

Kita telah memahami bahwa `many-to-many` berbeda karena membutuhkan pivot table. Oleh karenanya selain membuat migration untuk tabel `orders` dan `books` kita juga perlu membuat satu tabel lagi dan kita beri nama `book_order`.

Migration `create_orders_table`

```
public function up(){
  Schema::create("orders", function(Blueprint $table){
    $table->primary("id");
    $table->integer("total_price");
    $table->string("invoice_number");
    $table->enum("status", ["SUBMIT", "PROCESS", "CANCEL", "FINISH"]);
  });
}
```

Penjelasan: Kode migration untuk membuat struktur tabel `orders`, kita tidak memberikan foreign key di tabel `orders` ke tabel `books` di sini.

Migration `create_books_table`

```
public function up(){
  Schema::create("books", function(Blueprint $table){
    $table->primary("id");
    $table->string("title");
    $table->string("slug")->unique();
  });
}
```

Penjelasan: Kode migration untuk membuat struktur tabel `books`. Kita juga tidak memberikan foreign key ke tabel `orders` di tabel ini.

Migration `create_book_order` table (pivot table)

```
public function up(){
  Schema::create("book_order", function(Blueprint $table){
    $table->primary("id");
    $table->integer("book_id")->unsigned();
    $table->integer("order_id")->unsigned();

    $table->foreign("book_id")->references("id")->on("books");
    $table->foreign("order_id")->references("id")->on("order");
  });
}
```

Penjelasan: Kode migration untuk membuat pivot tabel yang akan kita gunakan untuk menghubungkan relation `many-to-many` antara tabel `orders` dengan tabel `books`, perhatikan kita mendefinisikan foreign key di tabel ini. Ada dua foreign key yaitu:

- `book_id` yang mereferensikan `id` di tabel `books`
- `order_id` yang mereferensikan `id` di tabel `orders`

Perhatian! konvensi penamaan untuk pivot tabel agar Laravel otomatis mendeteksi pada saat mendefinisikan `many-to-many` relationship di model adalah sebagai berikut:

- nama tabel merupakan bentuk tunggal dari dua tabel yang dihubungkan;

  Misalnya tabel `orders` dan `books` maka nama pivot tabelnya adalah kombinasi antara `order` dan `book`

- kombinasi nama tabel tersebut diurutkan berdasarkan alfabet, dihubungkan dengan underscore.

  Mislanya tabel `orders` dan `books` tadi memiliki pivot tabel kombinasi antara `order` dan `book` berarti nama pivot tabelnya adalah `book_order` bukan `order_book` karena `book` / awalan "b" secara alfabet lebih dulu dibandingkan `order` yang berawalan "o"

Mendefinisikan di Model

Model `Order`

```
public function books(){
  return $this->belongsToMany("App\Book");
}
```

Penjelasan: Mendefinisikan di model `Order` bahwa model ini bisa mempunyai banyak `Book`, kita memberi nama relation tersebut sebagai `books` dengan method `$this->belongsToMany("App\Book");` sehingga kita bisa mengaksesnya dari model `Order` seperti ini:

```
// dapatkan order dengan ID 1
$order = Order::find(1);

// dapatkan buku dari orderan di atas
$buku_dipesan = $order->books;
```

> Perhatikan bahwa untuk mendefinisikan `many-to-many` relationship di kedua model kita mendefinisikan relationship menggunakan `$this->belongsToMany()`

Model `Book`

```
public function orders(){
  return $this->belongsToMany("App\Order");
}
```

Penjelasan: Mendefinisikan model `Book` bahwa model ini bisa dimiliki oleh banyak `Order`, kita definisikan relation ke model `Order` sebagai `orders` dengan `$this->belongsToMany("App\Order");`. Sehingga kita bisa mengakses order apa saja yang memuat buku tertentu seperti ini:

```
// dapatkan buku dengan id 4
$buku = Book::find(4);

// cari order yang memiliki buku dengan id 4
$orderan_yang_memuat_buku_ini = $buku->orders;
```

Menghubungkan relationship

Untuk menghubungkan sebuah model `User` dengan `Handphone` dalam `one to one` relationship kita bisa gunakan beberapa cara

save()

Yang pertama adalah `save()` method. Kita gunakan dengan cara seperti ini:

```
$hanpdhone_baru = new Handphone(["phone_num"=>"099999999"]);

$user = User::find(1);

$user->handphone()->create($handphone_baru);
```

Penjelasan: Untuk menggunakan `save()` pertama kita buat terlebih dahulu data handphone dengan `new Handphone(["..."])` (ini belum menyimpan ke database), lalu kita cari user tertentu misalnya dengan `id` 1 simpan sebagai `$user`. Setelah itu kita create handphone tadi dan sekaligus tambahkan ke user tadi dengan cara `$user->handphone()->create($handphone_baru)`

> **Perhatikan** Sekarang kita menggunakan `$user->handphone()` bukan `$user->handphone` seperti contoh sebelumnya. Itu karena kita ingin melakukan `chaining`, yaitu ingin menggunakan method lain dalam hal ini `create()`. Sama halnya jika kita ingin menggunakan method-method lainnya. Tapi jika hanya ingin mendapatkan handphone dari user maka cukup gunakan `$user->handphone`

create()

Cara berikutnya adalah menggunakan `create()` method. Kita gunakan dengan cara yang hampir sama seperti sebelumnya:

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

$user->handphone()->create(["phone_num"=>"08777777777"]);
```

Penjelasan: Mirip dengan cara `save()` method, kita menggunakan `create()` untuk membuat handphone baru untuk user dengan `id` 1. Bedanya kita langsung memberikan properti yang akan diinsert ke tabel `handphones` tanpa perlu melaukan `new Handphone` seperti method `save()`

Perbedaan `create()` dan `save()`

- Method `save()` mengharapkan sebuah argument yaitu instance object dari model yang akan ditambahkan, dalam hal ini model `Handphone` makanya kita gunakan terlebih dahulu `new Handphone()` untuk membuat instance.
- Sementara itu, method `create()` mengharapkan sebuah argument yaitu `Array dari properti` yang akan diinsert ke tabel relationship, dalam hal ini tabel `handphone`, misalnya `["phone_num" => "0877777"]` atau `["phone_num"=>"081111111"]`

associate()

Cara ketiga adalah dengan menggunakan method `associate()` seperti ini:

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

$handphone = Handphone::where("phone_num", => "0877777777");

$user->handphone()->associate($handphone);
```

Penjelasan: Method `associate()` mengharapkan sebuah argument yaitu instance dari model relationship yang akan ditambahkan, dalam hal ini model `Handphone`. Karena method ini tidak mencreate data baru ke database kita bisa menggunakan method ini untuk menghubungkan handphone yang sudah ada di database ke user tertentu seperti pada kode di atas, kita tidak mengcreate handphone baru tapi melakukan query ke tabel handphone yang memiliki `phone_num` == "0877777777" lalu menambahkan hanphone tersebut ke `User` dengan `id` 1.

createMany()

Create many berfungsi sama seperti `create()` akan tetapi digunakan untuk mengcreate banyak relationship data sekaligus. Method ini tidak bisa digunakan untuk `one-to-one` relationship.

Method ini mengharapkan sebuah parameter yaitu multidimensional Array berisi properti-properti relationship yang akan diinsert.

Misalnya kita bayangkan sebuah relationship `many to many` antara model `Product` dengan `Category`. Di mana model `Post` memiliki banyak `Category` dengan relationship properti `categories`. Kita bisa membuat kategori-kategori baru sekaligus menghubungkannya ke `Product` tertentu seperti ini:

```
$post = Post::find(23);

$post->categories()->createMany([
  [
    "name" => "Sepatu"
```

```
    ],
    [
      "name" => "Fashion Pria"
    ]
  ]);
```

Penjelasan: Membuat 2 kategori baru yaitu sepatu dan fashion pria sekaligus menambahkannya sebagai kategori dari `Post` dengan `id` 23.

Dengan begitu, maka jika kita menggunakan kode berikut ini setelahnya:

```
$post->categories()
```

Maka akan menghasilkan 2 `Category` yaitu "Sepatu" dan "Post". Seperti ini:

```
[
  [
    "id" => "ID_DARI_DB",
    "name" => "Sepatu"
  ],
  [
    "id" => "ID_DARI_DB",
    "name" => "Fashion Pria",
  ]
]
```

attach()

Khusus untuk `many to many` relationship, disediakan method tersendiri agar lebih mudah untuk mengelola relationship jenis ini. Untuk menambahkan kita relationship kita gunakan method `attach()` seperti ini:

```
$product = Product::find(231);

$product->categories()->attach([1,2,4]);
```

Penjelasan: Kita menghubungkan `Product` 231 dengan `Category` yang memiliki `id` 1,2 dan 4. Seandainya sebelumnya `Product` ini sudah memiliki kategori dengan id lain yang terhubung, misalnya 10 dan 3, maka attach akan menambahkannya. Maka sekarang `Product` tadi memiliki kategori dengan id 1,2,3,4 dan 10.

Menghapus relationship

dissociate()

Untuk menghapus relationship kita gunakan method `dissociate()` seperti ini:

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

$user->handphone()->dissociate();
```

Penjelasan: Menghapus relationship handphone yang dimiliki oleh `User` dengan `id` 1. Dengan begitu kini data handphone yang sebelumnya dimiliki oleh user tersebut kini akan diupdate dengan field `user_id` sebagai foreign key bernilai null, bukan lagi bernilai 1.

detach()

Method `detach()` merupakan cara lain untuk menghapus relationship dan merupakan kebalikan dari `attach()`.

```
$product = Product::find(231);

$product->categories()->detach([10,3]);
```

Penjelasan: Menghapus relationship `Category` dengan `id` 10 dan 3 dari `Product` 231.

Sinkronisasi relationship

Sinkronisasi digunakan untuk memudahkan kita mengelola `many to many` relationship. Method yang digunakan adalah `sync()` dan method ini merupakan gabungan antara method `attach()` sekaligus `detach()`. Dengan kata lain, menggunakan method ini kita menambahkan relationship dengan id-id tertentu sekaligus menghapus relationship dengan id-id yang lain jika sebelumnya ada.

Contoh menggunakan ilustrasi `many to many` antara model `Product` dan `Category`. Di mana sebuah `Product` dengan id 231 saat ini memiliki `Category` 1,2,3,4,10,23, yaitu berjumlah 6 kategori. Lalu kita mengeksekusi kode berikut ini:

```
$product = Product::find(231);

$product->categories()->sync([1,3]);
```

Penjelasan: Melakukan sinkronisasi relationship antara `Product` 231 dan `Category` `id` 1, `id` 3. Dengan kode tersebut maka kini `Product` tadi hanya akan memiliki `Category` 1 dan 3, sementara `Category` lainnya yang sebelumnya terhubung yaitu 2,4,10 dan 23 akan otomatis dihapus dari `Product` 231.

Querying relationship tingkat lanjut

Mendapatkan data model hanya yang memiliki relation tertentu

Untuk mendapatkan sebuah data yang memiliki relationship tertentu kita gunakan method `has()`

```

```

```
$categorized_products = Product::has("categories")->get();
```

Penjelasan: Mendapatkan semua product yang memiliki kategori.

```
$orders = Order::has("books", ">=", 4)->get();
```

Penjelasan: Mendapatkan semua order yang didalamnya terdapat 4 buku atau lebih.

Mendapatkan data berdasarkan query terhadap relationship model

Kita juga bisa mendapatkan data berdasarkan query terhadap relationship menggunakan method `whereHas()` seperti ini:

```
$orders = Order::whereHas("books", function($query){
  $query->where("title", "Advanced Fullstack Developer");
})->get();
```

Penjelasan: Mendapatkan semua order yang memiliki buku berjudul "Advanced Fullstack Developer"

Mendapatkan hanya data model yang tidak memiliki relation tertentu

Sebaliknya kita juga bisa melakukan query terhadap model yang tidak mempunyai relation tertentu, kita gunakan method `doesntHave()`.

```
$uncategorized_products = Product::doesntHave("categories")->get();
```

Penjelasan: Dapatkan semua produk yang tidak memiliki kategori.

Selain itu kita juga bisa menambahkan query lebih lanjut terhadap relationship menggunakan method `whereDoesntHave()` misalnya:

```
$orders = Order::whereDoesntHave("books", function($query){
  $query->where("title", "MS Access");
});
```

Penjelasan: Dapatkan semua order yang tidak memiliki buku berjudul "MS Access" di dalamnya.

Menghitung jumlah relationship

```
$order = Order::withCount("books")->first();
```

```
$jumlah_buku = $order->books_count
```

Penjelasan: Dapatkan order pertama dan dapatkan juga jumlah buku di orderan tersebut. Laravel akan otomatis memberikan properti dengan format {relationship}_count seperti contoh di atas `books_count`.

Contoh lain

```
$product = Product::withCount("categories")->first();

$jumlah_kategori = $product->categories_count;
```

Penjelasan: Dapatkan product pertama dan dapatakan juga jumlah kategorinya.

Method `withCount()` juga bisa kita berikan query lebih lanjut misalnya seperti ini:

```
$products = Product::withCount(["categories" => function($query){
  $query->where("name", "like", "%anak%");
}])
```

Penjelasan: Dapatkan produk-produk dan jumlah kategori yang memiliki kata-kata "anak" di dalamnya.

Lebih lanjut `withCount()` juga bisa digunakan untuk multiple relationship seperti ini:

```
$products = Product::withCount(["categories", "comments"])->get();

$jumlah_kategori = $products->categories_count;
$jumlah_komentar = $products->comments_count;
```

Penjelasan: Dapatkan produk-produk beserta dengan jumlah kategori dan jumlah komentarnya.

Foreign Key Constraint

Cascading merupakan fitur yang memungkinkan kita untuk mendefinisikan perilaku terhadap tabel-tabel yang saling berhubungan dalam relationship. Perilaku tersebut bisa kita definisikan ketika terjadi perubahan terhadap salah satu data di salah satu tabel (on update) atau jika data di salah satu tabel dihapus (on delete). Ada 4 yaitu SET NULL, CASCADE, NO ACTION dan RESTRICT.

Tipe-tipe constraint

SET NULL

Jika ada data di tabel parent yang dihapus atau diupdate, maka foreign key di tabel-tabel yang berkaitan akan otomatis diupdate dengan NULL.

Jika kamu menggunakan SET NULL, pastikan foreign key di tabel child telah menggunakan NULLABLE, atau di migration menggunakan `nullable()`

RESTRICT

Menolak penghapusan atau update terhadap data di parent tabel. Ini merupakan default di MySQL.

CASCADE

Jika terjadi penghapusan atau perubahan data di parent model maka data di child model akan mengikuti.

Itu berarti jika data di parent tabel dihapus, maka seluruh data di child tabel yang mereferensikan data di parent tabel akan dihapus juga.

NO ACTION

Secara fungsi sebetulnya sama saja dengan RESTRICT

Menerapkannya dalam migration

Kita bisa menerapkan foreign key constraint menggunakan migration. Caranya adalah seperti ini saat mendefinisikan relationship antar tabel. Kita ambil contoh tabel `handphones` yang memiliki foreign key `user_id` mereferensikan `id` di tabel `users`.

```
function up(){
  Schema::create("handphones", function(Blueprint $table){
    $table->primary("id");
    $table->string("phone_num")->unique();
    $table->integer("user_id")->unsigned()->unique();

    $table->foreign("user_id")->references("id")->on("users")-
>onDelete("cascade");
  });
}
```

Penjelasan: Mengatur foreign key constraint antara tabel `hanphones` dan `users` menjadi "CASCADE" menggunakan method `onDelete("cascade")`. Dengan demikian maka jika sebuah data user dihapus maka record handphone yang berkaitan di tabel `handphones` juga akan ikut dihapus.

Untuk menggunakan restriction lain cukup masukan paremeter di function `onDelete()`, misalnya `onDelete("restrict")`, `onDelete("set null")`. Dan untuk menerapkan on update constraint gunakan method `onUpdate()`.

Misalnya seperti ini:

```
$table->foreign("user_id")->references("id")->on("users")-
>onDelete("cascade")->onUpdate("cascade");
```

Kustomisasi definisi relationship

Pada subbab ini kita akan melakukan kustomisasi terhadap definisi relationship di model. Bab ini umumnya tidak perlu juga kamu mengikuti konvensi-konvensi penamaan tabel untuk relationship di laravel.

Tapi terkadang kamu memiliki struktur tabel yang berbeda, apalagi jika bukan kamu yang membuat databasenya. Atau kamu ingin mengubah database yang sudah ada dengan aplikasi Laravel yang sebelumnya bukan.

Intinya, terkadang kamu tidak mengikuti konvensi penamaan di Laravel tapi kamu ingin memanfaatkan fitur relationship tanpa harus mengubah struktur tabel yang sudah ada. Mari kita pelajari

Mengubah foreign key di relationship

Model `User` memiliki relation dengan `Handphone` di model `User` kita mendefinisikan sebagai relationship `handphone()`, cukup seperti ini:

```
public function handphone(){
  return $this->hasOne("App\Handphone");
}
```

Kita tidak perlu menuliskan foreign key apa yang digunakan oleh relationship ini, karena ini model User maka Laravel mengasumsikan ada field `user_id` diambil dari `{model}_id`. Hal ini tidak masalah jika di tabel `handphones` memang terdapat field `user_id`, seandainya ternyata bukan field `user_id` tapi ternyata di tabel `handphones` menggunakan tabel `owner_id` maka kita mendefinisikan relationship kita di model `User` seperti ini:

Model `User`

```
public function handphone(){
  return $this->hasOne("App\Handphone", "owner_id");
}
```

Dan lebih lanjut lagi, ternyata tabel `users` kita menggunakan primaryKey bukan field `id` tetapi field dengan nama `customer_id` padahal Laravel mengasumsikan foreign key `owner_id` mereferensikan field `id` di users, kita bisa memberitahu Laravel bahwa foreign key kita mereferensikan `customer_id` bukan `id` seperti ini:

```
public function handphone(){
  return $this->hasOne("App\Handphone", "owner_id", "customer_id");
}
```

Mengubah pivot tabel pada many-to-many relationship

Kita juga telah belajar konvensi penamaan tabel untuk pivot pada `many to many` relationship di bab ini juga. Tapi kita juga bisa memberitahu Laravel bahwa kita akan menggunakan custom nama pivot tabel misalnya

`book_order_pivot` dalam relationship antara `Order` dan `Books` maka kita definisikan relationshipnya seperti ini:

Model `Order`

```
public function books(){
  return $this->belongsToMany("App\Book", "book_order_pivot");
}
```

Kemudian di model `Book` Model `Book`

```
public function orders(){
  return $this->belongsToMany("App\Order", "book_order_pivot");
}
```

Mengubah foreign key pada many-to-many relationship

Lebih lanjut kita juga bisa melakukan perubahan terhadap field-field apa saja yang kita gunakan di tabel pivot tabel kita seperti ini:

Model `Order`

```
public function books(){
  return $this->belongsToMany("App\Book", "book_order_pivot",
"customer_id", "book_id");
}
```

Kesimpulan

Di bab ini kita belajar intense terkait model relationship. Dan memang model relationship ini membuka kemampuan kita untuk mengembangkan aplikasi yang kompleks dengan lebih mudah. Karena banyak sekali fitur-fitur yang bisa kita gunakan untuk mengelola relationship baik itu `one to one`, `one to many` atau `many to many` relationship.

Selain itu kita juga telah belajar pengenalan masing-masing tipe relationship di atas. Pengenalan tersebut memahamkan kita mengenai konsep relationship dasar dalam relationship database management system. Itu berarti pemahaman tersbebut bisa kamu pakai tidak peduli apapun frameworknya.

Dan di bab ini juga kita belajar menerapkan pemahaman kita mengenai relationship ke dalam aplikasi Laravel mulai dari menyusun stuktur tabel kita menggunakan migration kemudian melakukan definisi relationship di model. Dan terakhir kita banyak belajar mengenai fitur-fitur eloquent relationship.

Di bab ini tidak ada latihan praktik, namun kamu diharapkan untuk memahami apa yang ada di dalamnya karena kita akan langsung mempraktikan ilmu yang kita pelajari ini di bab study kasus nanti.