



Practicum Module

Web Programming

Hands on Lab

[MONTH] [YEAR]

Information in this document, including URL and other Internet Web site references, is subject to change without notice. This document supports a preliminary release of software that may be changed substantially prior to final commercial release, and is the proprietary information of Binus University.

This document is for informational purposes only. BINUS UNIVERSITY MAKES NO WARRANTIES, EITHER EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

The entire risk of the use or the results from the use of this document remains with the user. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Binus University.

Binus University may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Binus University, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, email address, logo, person, place or event is intended or should be inferred.

© 2011 Binus University. All rights reserved.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.

DAFTAR ISI

BAB 1 INTRODUCTION TO LARAVEL	1
1.1 INTRODUCTION	2
1.2 MVC (MODEL-VIEW-CONTROLLER)	2
1.3 INSTALLING LARAVEL	4
1.4 ENVIRONMENT & APP CONFIG.....	11
1.5 STRUKTUR FOLDER DAN ROUTING.....	17
1.6 OOP CONCEPT	31
1.7 BLADE TEMPLATES.....	31
BAB 2 CONTROLLER & MODEL	41
2.1 CONTROLLER	42
2.2 MODEL / ELOQUENT MODEL.....	59
2.3 MIGRATION DAN SEEDING.....	65
2.4 CRUD (CREATE, READ, UPDATE, DELETE).....	77
2.5 QUERY	92
BAB 3 PAGING & SEARCHING.....	98
3.1 PAGING & SEARCHING.....	99
3.2 INFINITE SCROLLING	108
BAB 4 AUTHENTICATION & VALIDATION.....	115
4.1 OTENTIKASI (AUTHENTICATION).....	116
4.2 VALIDASI.....	123
4.3 PESAN ERROR	128
4.4 HALAMAN ERROR	130
BAB 5 ADVANCED.....	133
5.1 FILE UPLOAD	134
5.2 OTORISASI (AUTHORIZATION)	137
5.3 SESSION.....	144
5.4 COOKIES	150

OVERVIEW

BAB 1

- Introduction to Laravel

Memperkenalkan Laravel sebagai Framework PHP untuk mengembangkan aplikasi website secara mudah dan dengan struktur yang tertata rapih.

BAB 2

- Controller & Model

Memahami bagaimana *controller* pada Framework Laravel menerima dan mengolah permintaan dari pengguna, menggunakan *model* untuk mendapatkan data dan kemudian mengirimkannya ke *view*.

BAB 3

- Paging & Searching

Melakukan *paging & searching* untuk memudahkan pengguna dalam mencari data yang diinginkan dan meringankan beban kerja sistem untuk mengambil data yang banyak.

BAB 4

- Otentikasi & Validasi

Melakukan otentikasi agar pengguna yang menggunakan aplikasi kita adalah pengguna yang sudah terdaftar atau terverifikasi dan validasi untuk memastikan kesesuaian input dari pengguna dengan ketentuan dari sistem.

BAB 5

- Advanced

Mengenal Framework PHP Laravel lebih dalam seperti melakukan pengunggahan file, otorisasi, session dan cookies.



1.1 Introduction

Laravel adalah generasi baru dari *web framework* yang cukup unggul dan populer. Laravel dibuat oleh Taylor Otwell dan merupakan *web framework* yang tidak berbayar dan terbuka untuk umum (*open-source*). Sebelum membahas lebih lanjut tentang Laravel, perlu anda ketahui dahulu bahwa *framework* berbeda dengan *library*. Apabila pada *library* (contoh: Emgu CV, OpenGL, JQuery) terdapat fungsi-fungsi yang telah disediakan oleh *library* tersebut untuk dapat langsung digunakan, *framework* adalah sebuah kerangka yang didesain untuk mendukung pengembangan suatu aplikasi, dimana setiap kode ditempatkan dan alur suatu aplikasi berjalan diatur.

Laravel menggunakan arsitektur MVC (Model-View-Controller) sebagai dasar pembuatan *framework*-nya. Untuk mengetahui lebih lanjut mengenai arsitektur MVC, anda dapat membaca subbab 1.1 MVC (Model-View-Controller).

Laravel sering disebut “*The PHP Framework for Web Artisans*” yang berarti Laravel adalah *framework* PHP yang ditujukan untuk “pengrajin”. Laravel mengharapkan pengembangan aplikasi dengan struktur dan kode yang bersih / baik dari sejak awal pengembangan. Laravel berada dibawah MIT License, dan menggunakan GitHub sebagai tempat / repository untuk berbagi kode.

1.2 MVC (Model-View-Controller)

MVC atau **Model-View-Controller** adalah sebuah *design-pattern* yang digunakan untuk mengembangkan aplikasi dengan data yang dipisahkan dari tampilan, sehingga data dan tampilan tidak berhubungan langsung melainkan harus melalui proses / logika yang dibuat oleh pengembang. Selain Laravel, arsitektur MVC juga diterapkan pada beberapa *framework* lain seperti Rails, ASP.NET MVC, dan lain-lain.

Arsitektur MVC membagi aplikasi menjadi 3 bagian, yaitu:

1. Model

Sebagai representasi dari data / sumber daya dari aplikasi. Biasanya dibuat sesuai dengan kebutuhan data-data yang ada pada database. Model dapat dianggap sebagai representasi dari tabel-tabel pada database. Dalam Laravel 5.2, model terdapat pada folder app.

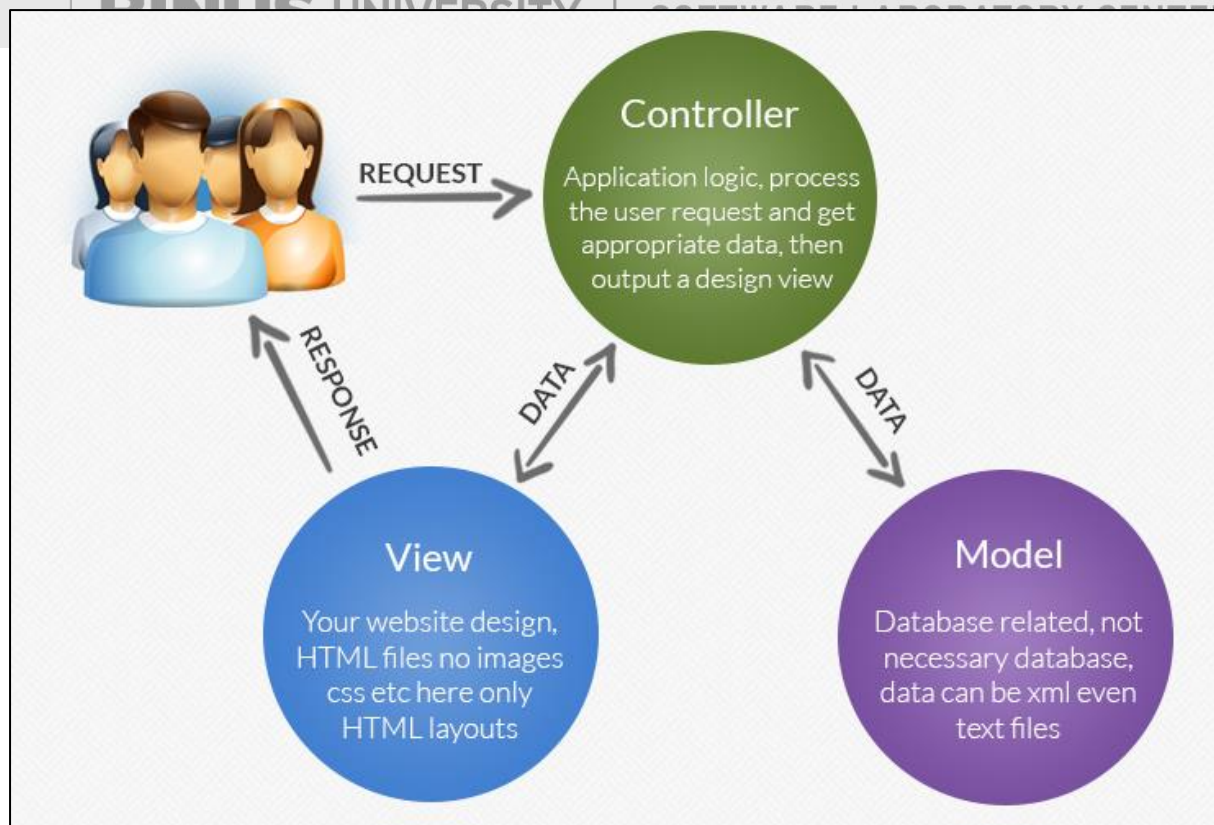
2. Controller

Bertanggung-jawab untuk menangkap *request* dari pengguna, memproses permintaan dari pengguna, dan kemudian mengirim respon dari *request* yang dikirimkan. Dalam Laravel 5.2, *controller* terdapat pada folder `app/Http/Controllers`.

3. View

Bertanggung-jawab untuk menampilkan respon yang dikembalikan dari suatu *controller* dalam format yang sesuai, biasanya dalam bentuk halaman web HTML. Dalam Laravel 5.2, *view* terdapat pada folder `resources/views`.

Gambar 1.1 menunjukkan penggambaran dari alur kerja MVC dimana pengguna akan melakukan *request* ke *controller*, kemudian *request* tersebut akan diproses oleh *controller* menggunakan logika yang sudah dibuat oleh pengembang aplikasi. Setelah diproses oleh *controller*, *controller* akan me-*request* data ke model yang merupakan representasi dari *database* sesuai dengan *request* yang sudah diproses. Model akan mengembalikan respon berupa data yang diminta kembali ke *controller*. Kemudian *controller* akan melanjutkan data yang diterima dari model ke *view* untuk ditampilkan ke pengguna sebagai respon dari *request* yang sudah dilakukan.



MVC sendiri memiliki banyak keuntungan dalam penerapannya untuk mengembangkan sebuah aplikasi. Keuntungan MVC ini dikarenakan tiap-tiap bagian bersifat independen dari bagian lain. Sebagai contoh bagian *View* tidak akan terpengaruh apabila terjadi *bug* di bagian *Model*. Berikut adalah keuntungan-keuntungan aplikasi yang menggunakan arsitektur MVC:

Keuntungan	Pengertian
<i>Simultaneous Development</i>	Banyak pengembang yang bisa bekerja secara bersama-sama di tiap-tiap bagian.
<i>High Cohesion</i>	Tiap bagian dari aplikasi yang menggunakan arsitektur MVC memiliki tugas yang spesifik.
<i>Low Coupling</i>	Tiap bagian dari aplikasi yang menggunakan arsitektur MVC bersifat independen satu sama lain.
<i>Ease of Modification</i>	Karena tiap bagian bertanggung-jawab atas fungsinya masing-masing, perubahan dapat dilakukan dengan mudah.
<i>High Code Reuseability</i>	Kode yang sudah ada bisa dengan mudah digunakan kembali.

Selain keuntungan, MVC juga memiliki beberapa kerugian. Berikut adalah kerugian-kerugian aplikasi yang menggunakan arsitektur MVC:

Kerugian	Pengertian
<i>Code Navigability</i>	Struktur dari <i>web framework</i> yang menggunakan arsitektur MVC bisa sangat kompleks dan butuh waktu lama dalam membiasakan diri untuk menggunakannya.
<i>Multi-Artifact Consistency</i>	Bagian-bagian dari aplikasi yang menggunakan arsitektur MVC memiliki tugas yang spesifik, hal ini dapat menjadi sulit untuk tetap mempertahankan fungsinya yang spesifik.
<i>Pronounced learning curve</i>	Pengembang yang menggunakan arsitektur MVC harus menguasai berbagai teknologi.

1.3 Installing Laravel

Laravel memiliki banyak versi yang dapat digunakan dalam mengembangkan aplikasi. Dalam HOL ini versi Laravel yang akan digunakan adalah Laravel 5.2. Sebelum mulai membahas

tentang cara menginstal Laravel. Ada beberapa persyaratan yang harus dipenuhi agar dapat menginstal dan menggunakan Laravel 5.2, yaitu:

1. PHP >= 5.5.9

Untuk dapat menggunakan Laravel tentu anda harus memiliki PHP terlebih dahulu. Versi minimal PHP untuk bisa menggunakan Laravel 5.2 adalah PHP 5.5.9. Apabila anda pernah menginstall XAMPP maka anda telah memiliki PHP di komputer anda, jika tidak maka anda harus menginstal PHP terlebih dahulu. Cara agar anda bisa mengetahui versi PHP yang anda gunakan adalah dengan menjalankan perintah “php -v” pada *command prompt*.

```
D:\>php -v
```

Lalu hasilnya,

```
PHP 5.6.23 (cli) (built: Jun 22 2016 12:15:20)
Copyright (c) 1997-2016 The PHP Group
Zend Engine v2.6.0, Copyright (c) 1998-2016 Zend Technologies
```

2. OpenSSL PHP Extension

Selain versi PHP yang harus minimal 5.5.9, untuk dapat menggunakan Laravel diperlukan ekstensi OpenSSL PHP agar fitur keamanan yang disediakan oleh *framework* Laravel dapat digunakan.

3. PDO PHP Extension

Ekstensi PDO PHP digunakan untuk dapat membuat koneksi antara PHP dengan database server yang digunakan untuk Laravel.

4. Mbstring PHP Extension

Ekstensi Mbstring PHP digunakan untuk dapat menggunakan fungsi-fungsi yang berfungsi untuk memanipulasi string.

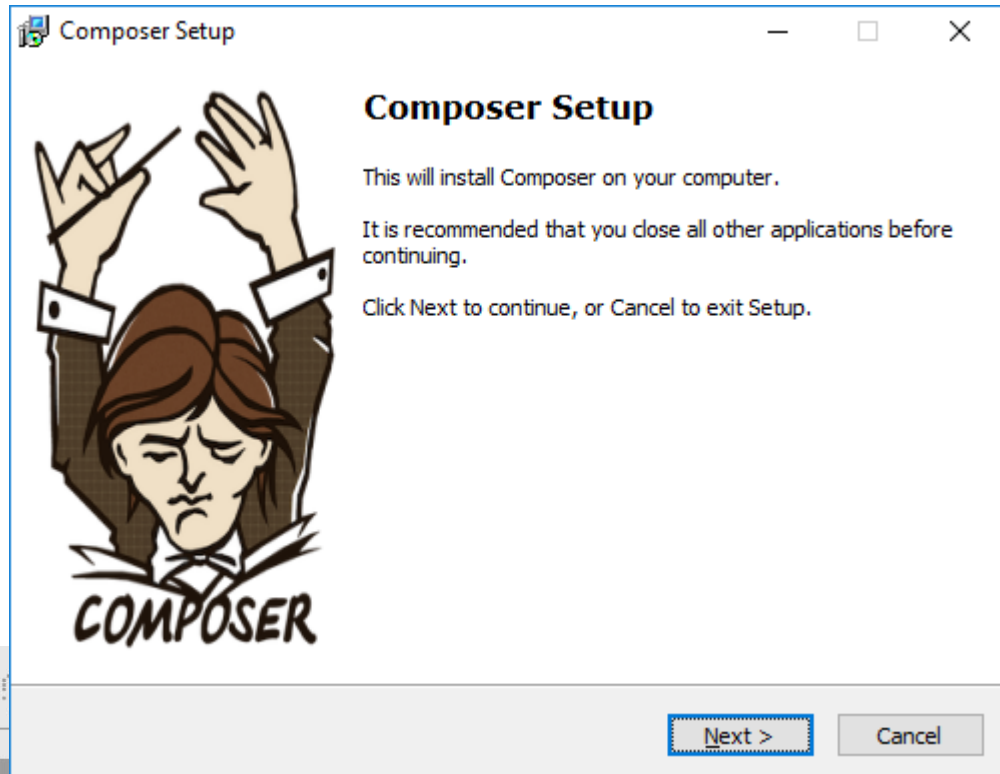
5. Tokenizer PHP Extension

Ekstensi Tokenizer PHP digunakan untuk dapat memanipulasi string menjadi kode PHP.

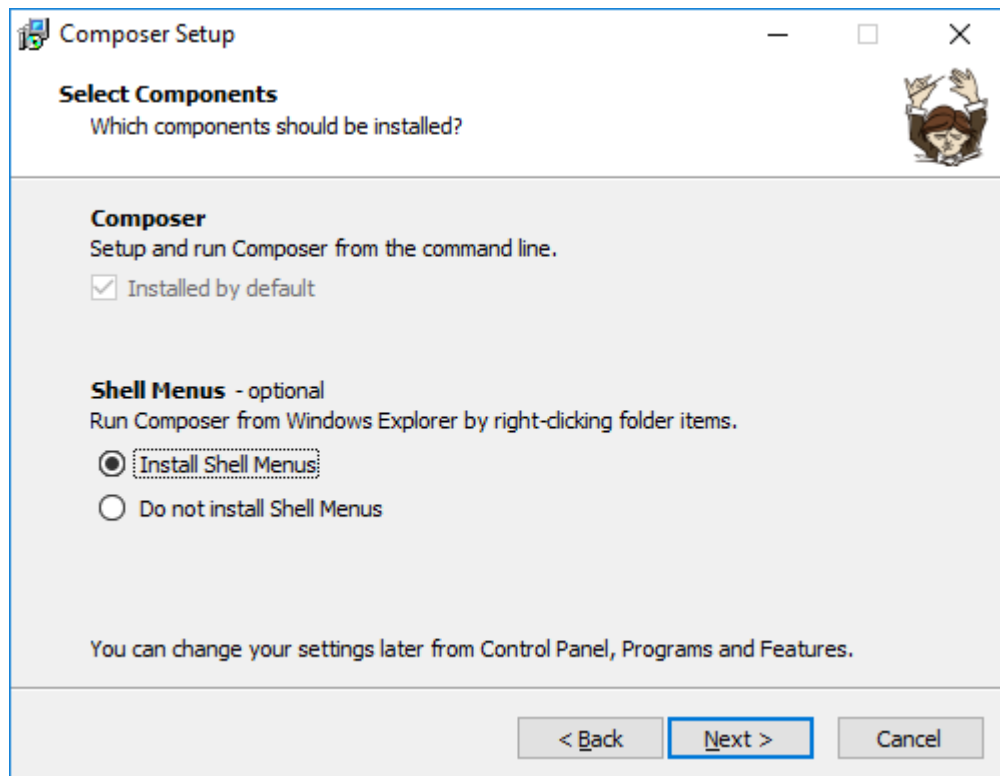
Setelah memenuhi semua persyaratan agar dapat menggunakan Laravel 5.2, maka Laravel 5.2 sudah dapat diinstal dengan menggunakan **Composer**. Composer adalah alat untuk mengelola *dependency* di PHP. Dengan Composer, *library* yang digunakan pada proyek Laravel dapat di deklarasikan. Composer juga sekaligus dapat mengatur *library* tersebut yang dalam hal ini adalah menginstal dan memperbaharainya jika diperlukan.

Composer dapat didownload dari link berikut <https://getcomposer.org/Composer-Setup.exe>. Berikut ini adalah langkah-langkah dalam menginstal composer.

1. Tampilan pertama saat composer-setup.exe dijalankan.

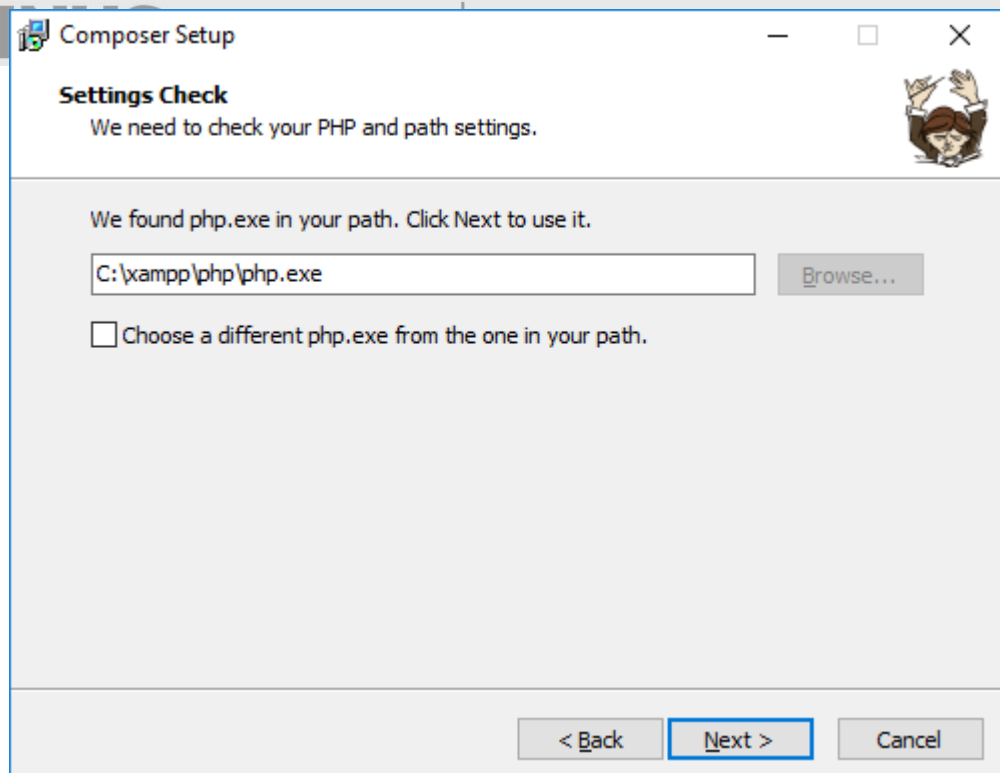


2. Tampilan setelah klik next.



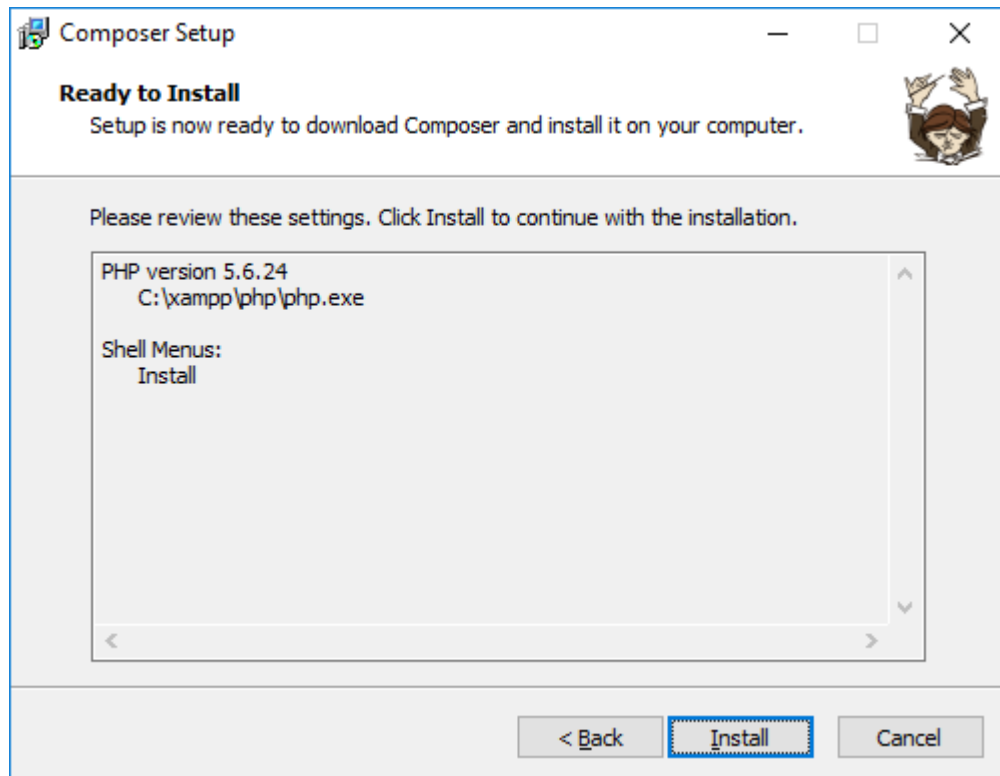
Shell menus adalah menu composer yang akan muncul ketika klik kanan disuatu folder.

3. Tampilan setelah klik next.



Masukkan lokasi dari file php.exe berada

4. Tampilan setelah klik next.



Klik install untuk mulai menginstal composer, tunggu hingga proses instalasi selesai dan composer siap digunakan.

Ada 2 cara yang dapat digunakan untuk menginstal Laravel 5.2 yaitu:

1. Menggunakan Laravel Installer

Untuk dapat menginstal proyek Laravel dengan Laravel Installer, anda perlu mengunduh dan menginstal Laravel Installer terlebih dahulu. Laravel Installer dapat diinstal dengan melakukan langkah-langkah berikut:

Buatlah satu buah folder “LaravelInstaller”.

Didalam folder LaravelInstaller yang baru saja dibuat, klik kanan lalu pilih **Use Composer here**, kemudian jalankan perintah dibawah ini:

```
D:\LaravelInstaller>composer require laravel/installer
```

- **composer**

Perintah yang digunakan untuk menjalankan program Composer yang telah terinstal.

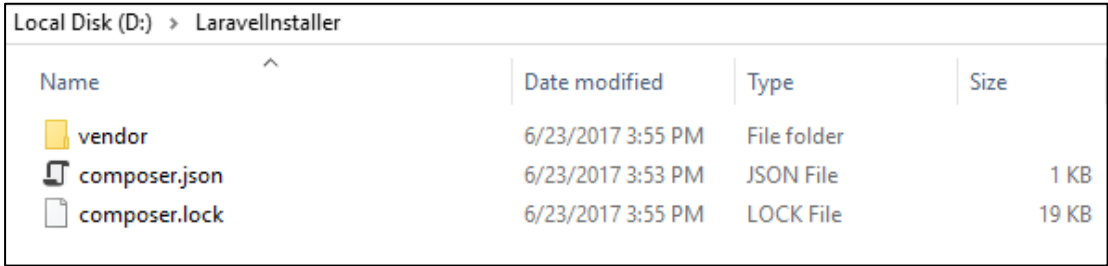
- **require**

Perintah composer yang digunakan untuk menambahkan / menginstal *package* baru yang ditentukan oleh pengguna.

- **laravel/installer**

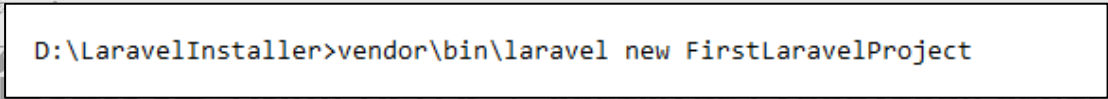
Nama *package* yang ingin ditambahkan / diinstal saat perintah dijalankan, yang dalam contoh ini adalah installer untuk laravel.

Jika Laravel Installer telah berhasil diinstal, maka akan terbuat folder dan file seperti gambar dibawah ini:



Name	Date modified	Type	Size
vendor	6/23/2017 3:55 PM	File folder	
composer.json	6/23/2017 3:53 PM	JSON File	1 KB
composer.lock	6/23/2017 3:55 PM	LOCK File	19 KB

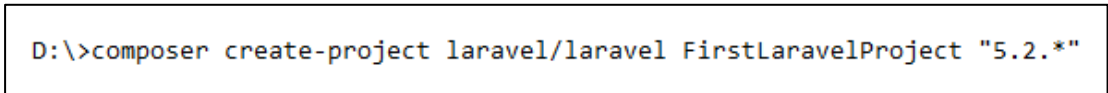
Setelah menginstal laravel installer anda dapat menginstal laravel dengan menjalankan perintah dibawah ini:



```
D:\LaravelInstaller>vendor\bin\laravel new FirstLaravelProject
```

2. Menggunakan Composer

Untuk dapat menginstal proyek Laravel dengan composer, dapat dilakukan dengan menjalankan perintah dibawah ini:



```
D:\>composer create-project laravel/laravel FirstLaravelProject "5.2.*"
```

- **composer**

Perintah yang digunakan untuk menjalankan program Composer yang telah terinstal.

- **create-project**

Perintah composer yang digunakan untuk membuat proyek baru berdasarkan nama *package* yang ditentukan oleh pengguna.

- **laravel/laravel**

Nama *package* yang akan diunduh dan diinstal saat perintah dijalankan yang dalam contoh ini *package* yang akan diinstal adalah proyek baru laravel.

- **FirstLaravelProject**

Nama proyek Laravel yang akan diinstal sehingga bisa diganti dengan nama project yang anda inginkan.

- **5.2.***

Versi Laravel yang akan diinstal. Apabila ketika menginstal Laravel tidak ditentukan versinya maka secara default versi Laravel yang akan diinstal adalah versi yang paling baru.

Jika Laravel telah selesai diinstal, maka akan terbuat satu folder proyek Laravel yang terdapat pada lokasi tempat anda menjalankan perintah tersebut. Pada contoh diatas, proyek laravel akan terbuat di folder D:\.

Untuk memastikan bahwa proyek Laravel yang telah diinstal dapat dijalankan dengan baik, maka anda dapat menjalankan proyek Laravel anda dengan menjalankan perintah dibawah ini:

```
D:\FirstLaravelProject>php artisan serve
```

- **php**

Perintah untuk menjalankan program PHP yang sudah dimiliki. Untuk dapat menjalankan program PHP di lokasi manapun yang diinginkan, maka lokasi file php.exe perlu ditempatkan pada *environment variables path*. Jika tidak, maka perintah “php” harus diganti menjadi “[lokasi file php.exe]\php” (contoh: C:\xampp\php).

- **artisan**

Perintah yang digunakan untuk menjalankan file artisan pada proyek laravel. Artisan adalah *comand-line interface* yang menyediakan sejumlah perintah yang dapat digunakan selama pengembangan proyek. Perintah-perintah yang disediakan oleh artisan dapat dilihat dengan menjalankan perintah seperti dibawah ini:

```
D:\FirstLaravelProject>php artisan list
```

Pastikan juga perintah ini benar-benar merujuk ke file artisan yang ada di dalam proyek laravel.

- **serve**

Perintah artisan yang digunakan untuk menjalankan proyek laravel.

Jika proyek Laravel telah berhasil dijalankan, maka akan tampil pesan yang menandakan bahwa aplikasi dapat dibuka melalui server “http://localhost:8000/” seperti gambar dibawah ini:

```
Laravel development server started on http://localhost:8000/
```

Sekarang anda sudah bisa membuka proyek anda di browser dengan memasukkan URI “localhost:8000” dan hasilnya adalah seperti gambar dibawah ini:



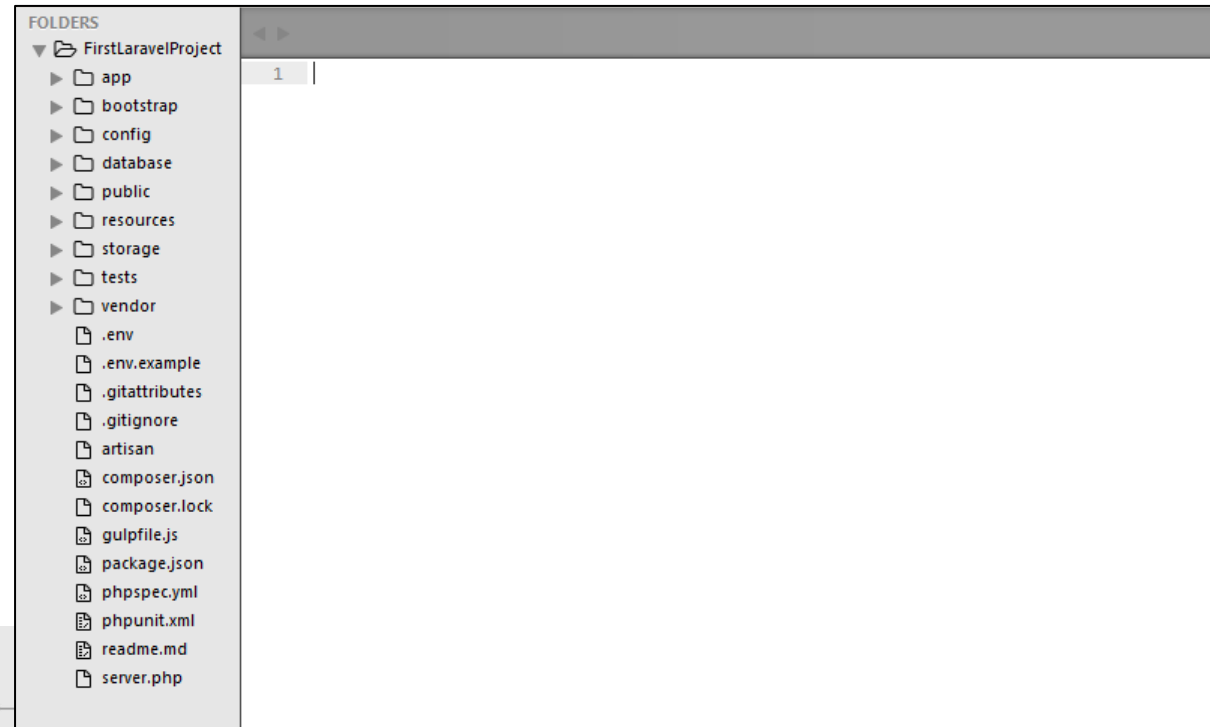
1.4 Environment & App Config

Sebelum memulai untuk mengembangkan aplikasi web dengan menggunakan Laravel, tentu kita perlu melakukan beberapa konfigurasi agar proyek yang akan dikembangkan sesuai dengan kebutuhan. Pada *framework* Laravel, semua konfigurasi dari aplikasi terdapat pada folder *config*. Beberapa hal yang dapat dikonfigurasi adalah database yang akan digunakan untuk aplikasi pada file *database.php*, zona waktu dan bahasa yang akan digunakan untuk aplikasi pada file *app.php*, dan masih banyak lagi konfigurasi lainnya.

Project Laravel dapat dibuka dengan menggunakan *text editor* seperti Notepad, Notepad++, dan Sublime Text ataupun dengan menggunakan IDE seperti PHPStorm. Untuk contoh dalam HOL ini yang akan digunakan adalah Sublime Text. Untuk membuka 1 proyek Laravel ke dalam **Sublime Text** dapat dilakukan dengan mengikuti langkah-langkah berikut:

1. Pilih menu File > Open Folder
2. Pilih folder proyek yang akan dibuka (dalam contoh ini adalah **FirstLaravelProject**)

Jika proyek yang telah dipilih berhasil dibuka maka proyek tersebut akan tampil seperti gambar dibawah ini:



1.4.1 Konfigurasi Environment

Sebagai pengembang aplikasi tentu akan sangat membantu jika terdapat konfigurasi yang dapat disesuaikan dengan *environment* pengembangan aplikasi. Sebagai contoh, misalnya anda ingin membedakan database yang digunakan selama pengembangan dan produksi dari aplikasi.

Untuk memenuhi hal ini Laravel menyediakan *library* DotEnv PHP yang menyediakan beberapa konfigurasi yang dapat diatur sesuai dengan kebutuhan dari pengembang. Konfigurasi tersebut dapat diatur pada file “.env” yang terdapat pada folder utama Laravel. Berikut adalah tampilan dari file “.env”:


```

.env
1 APP_ENV=local
2 APP_DEBUG=true
3 APP_KEY=base64:bn4QH3+jqcKd2uZ9u19kM6ZjplawPP0rUQczM35sMU8=
4 APP_URL=http://localhost
5
6 DB_CONNECTION=mysql
7 DB_HOST=127.0.0.1
8 DB_PORT=3306
9 DB_DATABASE=homestead
10 DB_USERNAME=homestead
11 DB_PASSWORD=secret
12
13 CACHE_DRIVER=file
14 SESSION_DRIVER=file
15 QUEUE_DRIVER=sync
16
17 REDIS_HOST=127.0.0.1
18 REDIS_PASSWORD=null
19 REDIS_PORT=6379
20
21 MAIL_DRIVER=smtp
22 MAIL_HOST=mailtrap.io
23 MAIL_PORT=2525
24 MAIL_USERNAME=null
25 MAIL_PASSWORD=null
26 MAIL_ENCRYPTION=null

```

- APP_ENV menentukan jenis *environment* pengembangan, nilainya dapat diisi dengan “local” atau “production”.
- APP_DEBUG menentukan aktif atau tidaknya debug errors.
- APP_KEY berisi 32 karakter acak yang berfungsi sebagai *key* yang akan digunakan sebagai alat untuk enkripsi untuk menjamin keamanan aplikasi.
- APP_URL menentukan alamat url yang akan digunakan untuk mengakses sumber daya dari aplikasi.
- DB_CONNECTION menentukan koneksi database yang akan digunakan, ada 4 koneksi database yang didukung oleh Laravel yaitu **sqlite** (SQLite), **pgsql** (Postgres), **sqlsrv** (SQL Server), dan **mysql** (MySQL).
- DB_HOST lokasi server database yang akan digunakan berada, jika database anda berada pada satu server / komputer yang sama dengan aplikasi, maka dapat diisi dengan nilai “127.0.0.1” atau “localhost”.
- DB_DATABASE nama database yang akan digunakan pada aplikasi.
- DB_USERNAME nama pengguna yang digunakan pada DBMS.
- DB_PASSWORD kata sandi yang digunakan pada DBMS.

1.4.2 App Config

Seperti yang sudah dijelaskan sebelumnya bahwa semua konfigurasi untuk aplikasi terdapat pada folder *config*. Salah satu file konfigurasi yang ada adalah *app.php*, file ini berisi konfigurasi-konfigurasi umum yang akan diterapkan pada keseluruhan aplikasi. Beberapa konfigurasi yang dapat diatur adalah sebagai berikut:

1. Debug

```
'debug' => env('APP_DEBUG', false),
```

Konfigurasi di atas sama dengan konfigurasi yang ada pada file “.env” namun hanya akan berjalan apabila konfigurasi pada file “.env” kosong (tidak diisi) sehingga nilai pada *APP_DEBUG* secara *default* akan menjadi *false*, tetapi apabila konfigurasi pada file “.env” diisi maka nilai dari *APP_DEBUG* akan mengikuti konfigurasi yang diatur pada file “.env”.

2. Zona waktu

```
'timezone' => 'UTC',
```

Konfigurasi diatas digunakan untuk menentukan zona waktu yang akan digunakan pada aplikasi dan juga sebagai acuan pengambilan waktu saat anda memakai fungsi-fungsi tanggal seperti *date* atau *datetime* pada PHP.

3. Bahasa

```
'locale' => 'en',
```

Konfigurasi diatas digunakan untuk menentukan bahasa yang akan digunakan pada aplikasi. Nantinya anda bisa memberi fitur untuk mengganti bahasa aplikasi anda menggunakan [Localization](#).

4. Enkripsi

```
'key' => env('APP_KEY', 'SomeRandomString'),  
'cipher' => 'AES-256-CBC',
```

Konfigurasi diatas digunakan untuk menentukan *key* dari aplikasi dan metode enkripsi yang akan digunakan pada aplikasi.

1. *Key* diatas sama dengan *APP_KEY* yang ada pada file “.env”. Sama seperti konfigurasi *debug* di atas, konfigurasi ini juga hanya akan berjalan apabila konfigurasi *APP_KEY* pada file “.env” kosong (tidak diisi).

2. *Cipher* adalah metode yang akan digunakan sebagai metode enkripsi pada aplikasi.

5. Providers

```
'providers' => [

    /*
     * Laravel Framework Service Providers...
     */
    Illuminate\Auth\AuthServiceProvider::class,
    Illuminate\Broadcasting\BroadcastServiceProvider::class,
    Illuminate\Bus\BusServiceProvider::class,
    Illuminate\Cache\CacheServiceProvider::class,
    Illuminate\Foundation\Providers\ConsoleSupportServiceProvider::class,
    Illuminate\Cookie\CookieServiceProvider::class,
    Illuminate\Database\DatabaseServiceProvider::class,
    Illuminate\Encryption\EncryptionServiceProvider::class,
    Illuminate\Filesystem\FilesystemServiceProvider::class,
    Illuminate\Foundation\Providers\FoundationServiceProvider::class,
    Illuminate\Hashing\HashServiceProvider::class,
    Illuminate\Mail\MailServiceProvider::class,
    Illuminate\Pagination\PaginationServiceProvider::class,
    Illuminate\Pipeline\PipelineServiceProvider::class,
    Illuminate\Queue\QueueServiceProvider::class,
    Illuminate\Redis\RedisServiceProvider::class,
    Illuminate\Auth\Passwords>PasswordResetServiceProvider::class,
    Illuminate\Session\SessionServiceProvider::class,
    Illuminate\Translation\TranslationServiceProvider::class,
    Illuminate\Validation\ValidationServiceProvider::class,
    Illuminate\View\ViewServiceProvider::class,

    /*
     * Application Service Providers...
     */
    App\Providers\AppServiceProvider::class,
    App\Providers\AuthServiceProvider::class,
    App\Providers\EventServiceProvider::class,
    App\Providers\RouteServiceProvider::class,

],
```

Konfigurasi diatas menampung semua *service providers* yang akan dimuat secara otomatis pada setiap *request* ke aplikasi anda.

6. Alias

```
'aliases' => [
    'App' => Illuminate\Support\Facades\App::class,
    'Artisan' => Illuminate\Support\Facades\Artisan::class,
    'Auth' => Illuminate\Support\Facades\Auth::class,
    'Blade' => Illuminate\Support\Facades\Blade::class,
    'Cache' => Illuminate\Support\Facades\Cache::class,
    'Config' => Illuminate\Support\Facades\Config::class,
    'Cookie' => Illuminate\Support\Facades\Cookie::class,
    'Crypt' => Illuminate\Support\Facades\Crypt::class,
    'DB' => Illuminate\Support\Facades\DB::class,
    'Eloquent' => Illuminate\Database\Eloquent\Model::class,
    'Event' => Illuminate\Support\Facades\Event::class,
    'File' => Illuminate\Support\Facades\File::class,
    'Gate' => Illuminate\Support\Facades\Gate::class,
    'Hash' => Illuminate\Support\Facades\Hash::class,
    'Lang' => Illuminate\Support\Facades\Lang::class,
    'Log' => Illuminate\Support\Facades\Log::class,
    'Mail' => Illuminate\Support\Facades\Mail::class,
    'Password' => Illuminate\Support\Facades>Password::class,
    'Queue' => Illuminate\Support\Facades\Queue::class,
    'Redirect' => Illuminate\Support\Facades\Redirect::class,
    'Redis' => Illuminate\Support\Facades\Redis::class,
    'Request' => Illuminate\Support\Facades\Request::class,
    'Response' => Illuminate\Support\Facades\Response::class,
    'Route' => Illuminate\Support\Facades\Route::class,
    'Schema' => Illuminate\Support\Facades\Schema::class,
    'Session' => Illuminate\Support\Facades\Session::class,
    'Storage' => Illuminate\Support\Facades\Storage::class,
    'URL' => Illuminate\Support\Facades\URL::class,
    'Validator' => Illuminate\Support\Facades\Validator::class,
    'View' => Illuminate\Support\Facades\View::class,
],
```

Konfigurasi diatas menampung *class* dan disertai alias sebagai indexnya. Setiap *class* dihubungkan dengan aliasnya menggunakan [array asosiatif](#) (array dengan index yang berupa string). *Aliases* ini berfungsi untuk menyederhanakan struktur folder yang kompleks saat anda memanggil class. Anda dapat dengan bebas menambahkan alias lain tanpa perlu khawatir dengan performa aplikasi karena semua alias disini hanya akan dimuat saat anda menggunakannya.

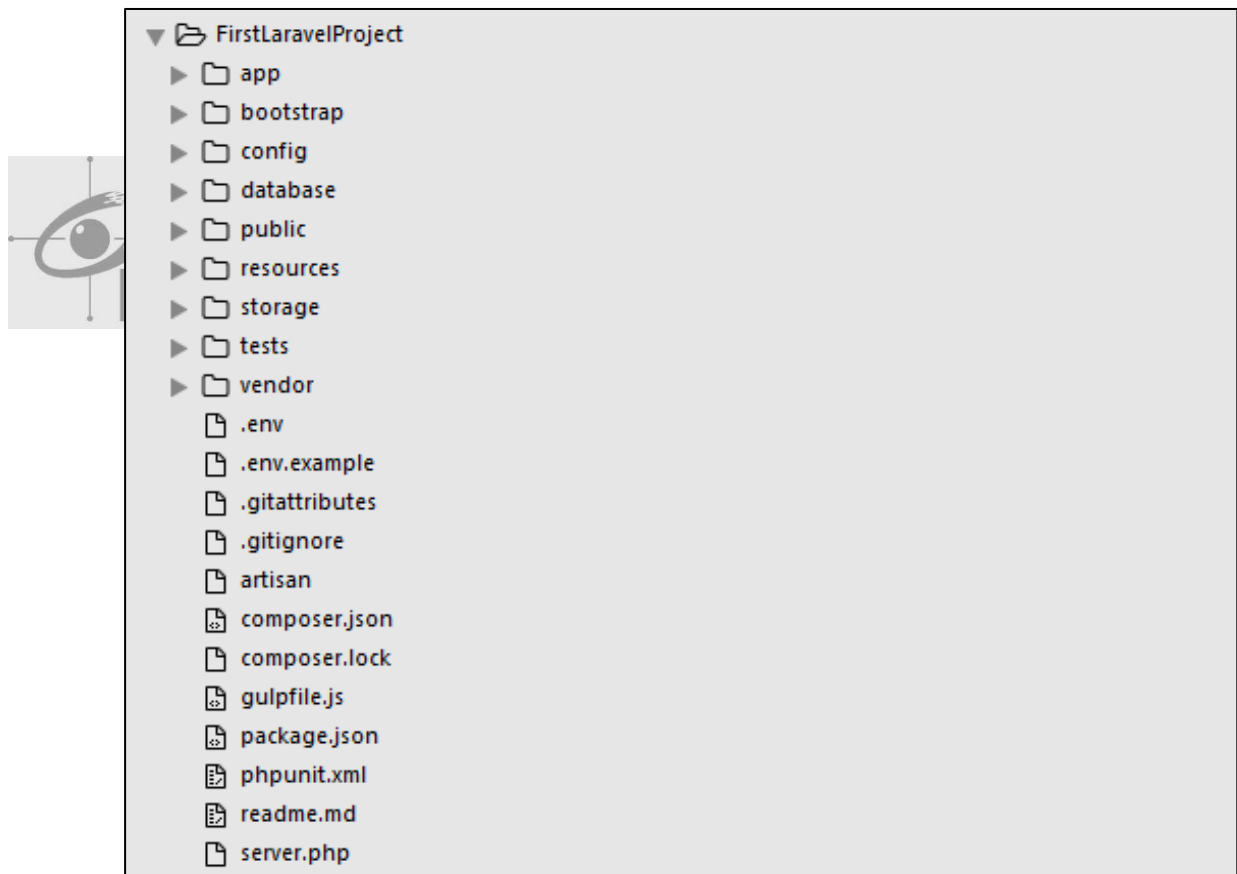
1.5 Struktur Folder dan Routing

Struktur folder *default* dari Laravel dibuat dengan maksud untuk menyediakan struktur pengembangan aplikasi yang baik dan rapih sejak awal pengembangan dimulai, baik untuk aplikasi yang besar ataupun aplikasi kecil. Struktur folder *default* ini juga dapat berubah sesuai dengan versi pengembangan dari *framework* Laravel.

1.5.1 Struktur Folder

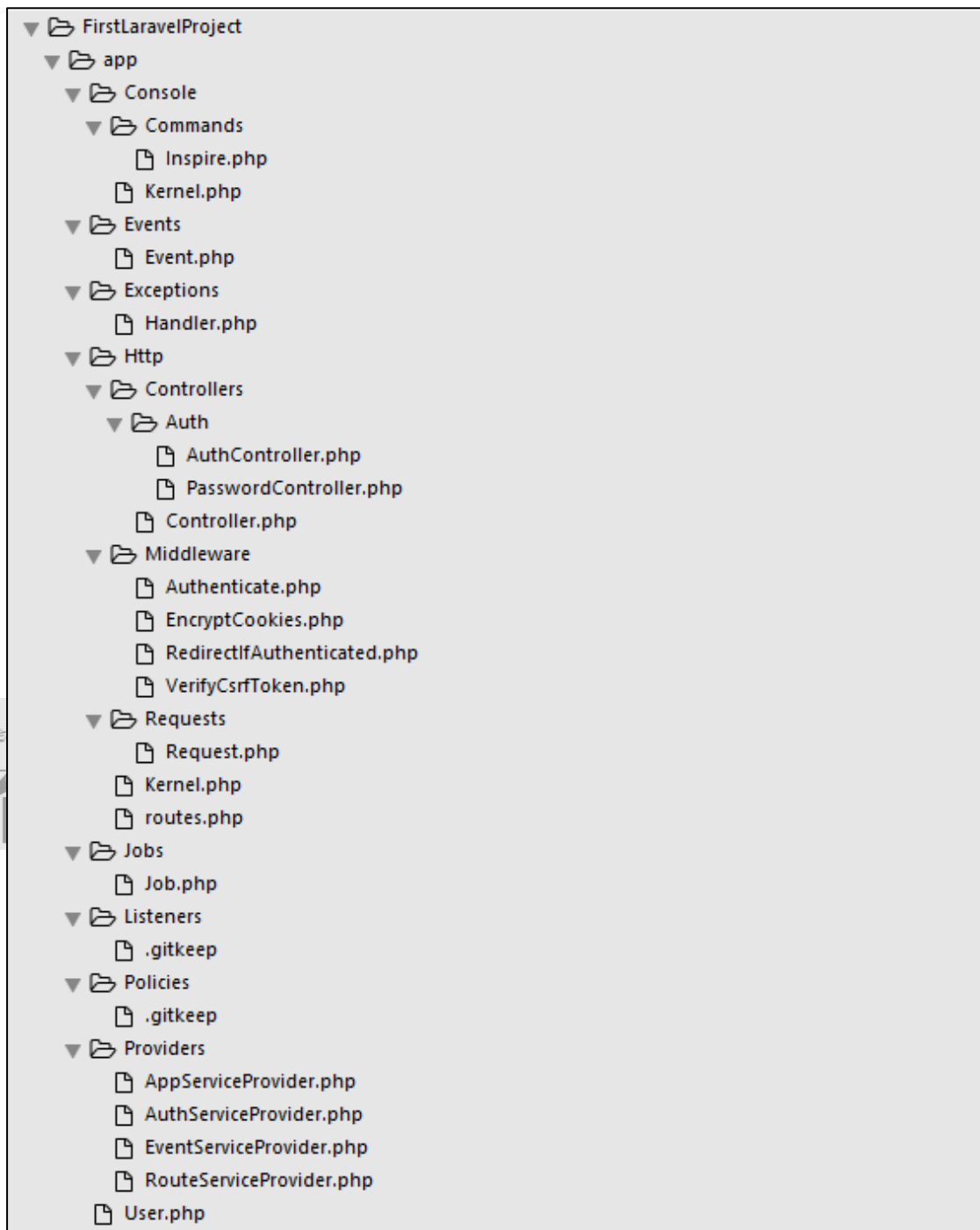
Struktur folder *default* dari Laravel memang sudah dibuat agar pengembangan aplikasi dapat dilakukan dengan mengikuti aturan *framework* yang sudah ditentukan oleh Laravel. Tetapi anda bebas untuk mengubah susunan dari proyek untuk pengembangan aplikasi anda selama Composer masih dapat mendukung / mengatur susunan tersebut. Berikut ini adalah folder-folder yang terdapat pada struktur *default* Laravel.

1. root



Folder root adalah folder paling awal dari instalasi proyek Laravel. Pada folder ini terdapat semua folder yang akan mendukung pengembangan aplikasi dan sudah dipisahkan sesuai dengan fungsinya masing-masing.

2. app



Folder app adalah folder yang berisi semua file yang mengandung kode inti dari aplikasi. Folder ini akan digunakan oleh pengembang untuk meletakkan kode yang mendukung pembuatan logika dan sumber daya yang akan digunakan pada aplikasi. Pada folder app terdapat beberapa folder dan file yang mendukung fungsinya yaitu:

- **Console**

Folder ini berisi semua perintah artisan yang dapat digunakan selama pengembangan aplikasi Laravel.

- **Http**

Folder ini berisi folder “Controllers” yang berisi kelas-kelas php yang berfungsi mengatur logika dari aplikasi anda, folder “Middleware” yang berisi kelas-kelas php untuk menyaring setiap request yang masuk ke aplikasi, folder Requests yang berisi kelas request untuk mengatur request dan file routes.php yang berfungsi untuk mengatur alur jalannya aplikasi yang akan dijelaskan di subbab berikutnya

- **Providers**

Folder ini berisi *Service Provider* dari aplikasi yang telah diregistrasikan di config/app.php.

- **Events**

Folder ini berisi *events* yang bisa diberi *listener* yang terdapat pada folder “Listeners”. Folder ini akan digunakan untuk memberi tahu suatu bagian dari aplikasi bahwa suatu kejadian telah terjadi. Contohnya adalah saat menambahkan data ke database, aplikasi akan mengirim email konfirmasi.

- **Exceptions**

Folder ini berisi *exception handler* dari aplikasi dan merupakan lokasi yang bagus untuk menempatkan setiap *exception* yang terjadi pada aplikasi.

- **Jobs**

Folder ini berisi kelas-kelas php yang berfungsi mengatur sistem antrian proses dari aplikasi. File-file pada folder ini akan membuat antrian proses agar berjalan secara sinkron dalam siklus request.

- **Listeners**

Folder ini berisi kelas-kelas yang akan mengatur *event* yang terdapat pada folder “Events”. Kelas *listener* akan menerima suatu *event* dan menampilkan logika untuk mengembalikan respon yang diinginkan.

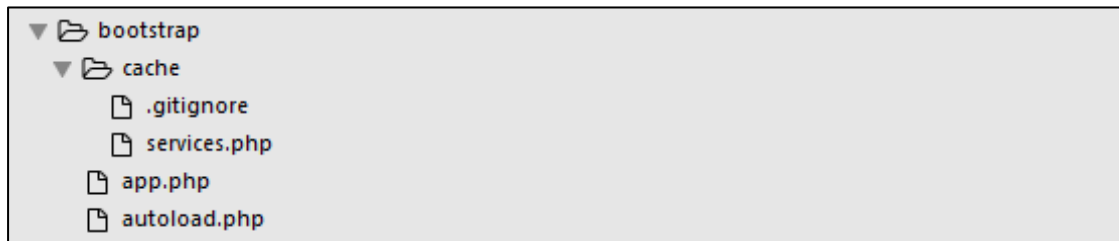
- **Policies**

Folder ini berisi kelas-kelas yang berfungsi mengatur logika otorisasi pada aplikasi. Kelas *policies* akan digunakan untuk menentukan kegiatan apa saja yang dalam dilakukan oleh seorang pengguna.

- **Model**

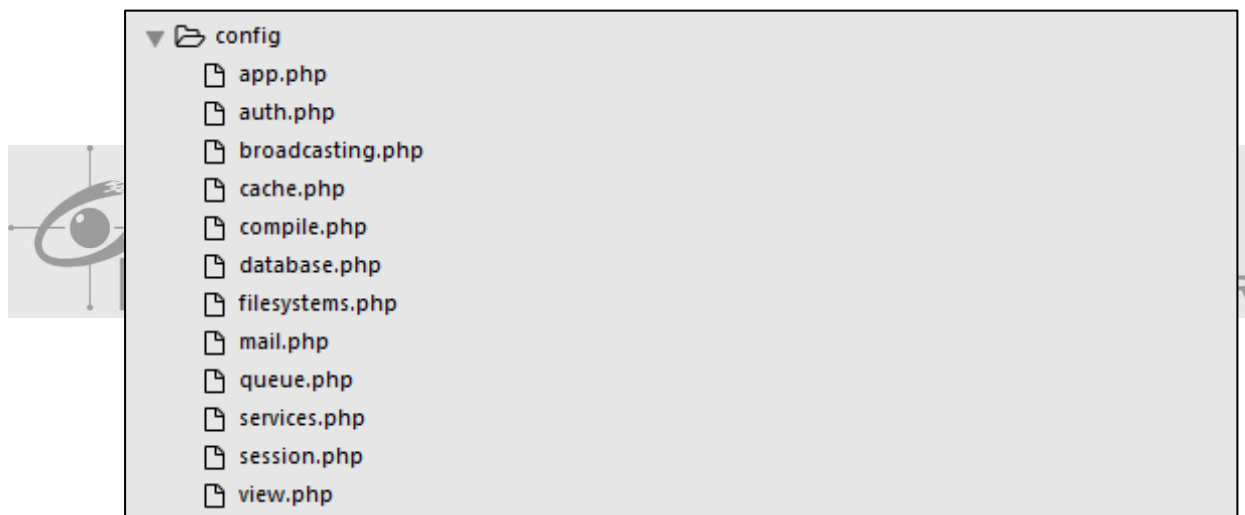
Pada folder app juga akan terdapat file-file model untuk mendukung pengembangan aplikasi.

3. bootstrap



Folder ini berisi folder cache yang berisi file-file yang berguna untuk optimisasi kerangka aplikasi:

4. config



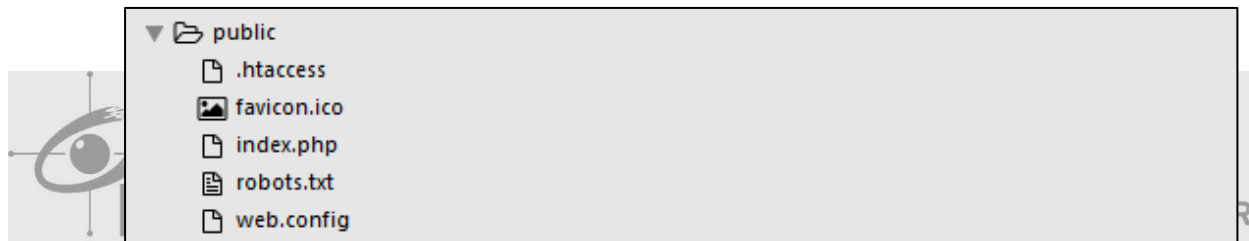
Folder ini berisi semua file konfigurasi untuk digunakan pada aplikasi.

5. database



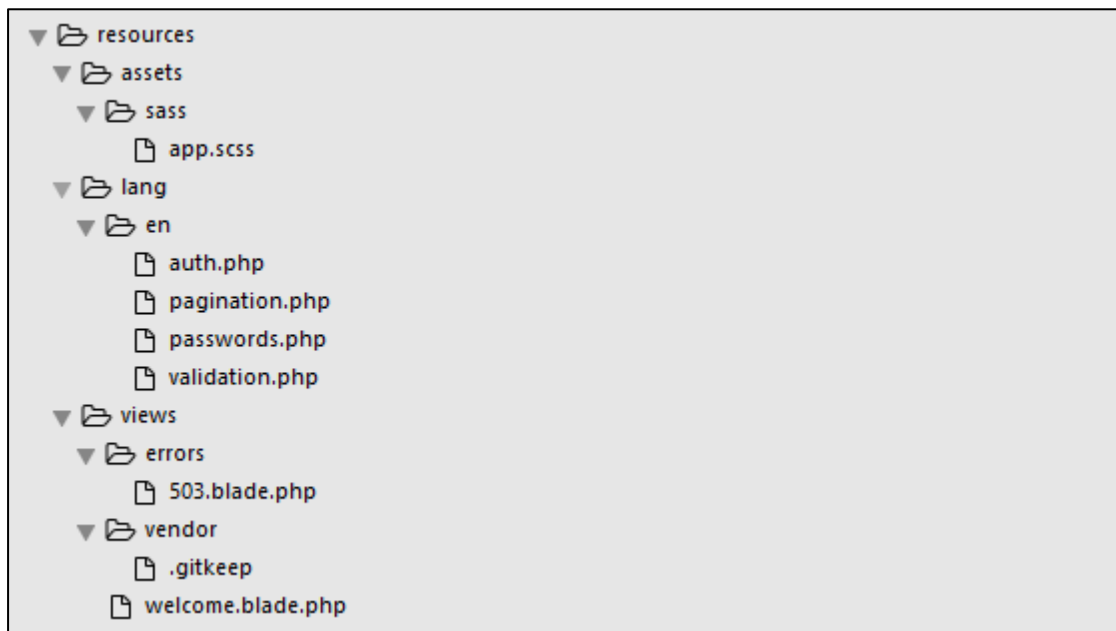
Folder ini berisi folder “factories” yang berfungsi untuk membuat banyak data secara mudah agar dapat digunakan saat melakukan test pada aplikasi, folder “migrations” yang berfungsi untuk mendefinisikan table pada database, dan folder “seeds” yang berguna untuk mengisi database dengan data-data testing secara otomatis.

6. public



Folder ini berisi *controller* untuk bagian depan aplikasi (yang dapat terlihat oleh pengguna) dan aset-aset yang digunakan pada aplikasi seperti file JavaScript, CSS, gambar, dan lain-lain.

7. resources



Folder ini berisi folder “assets” berfungsi untuk menyimpan file-file aset yang perlu di *compile* oleh *preprocessor* seperti sass, less, dan lain-lain, folder “lang” berfungsi untuk menyimpan file-file yang digunakan untuk lokalisasi, dan folder “views” yang berisi file-file halaman web dari aplikasi.

8. storage



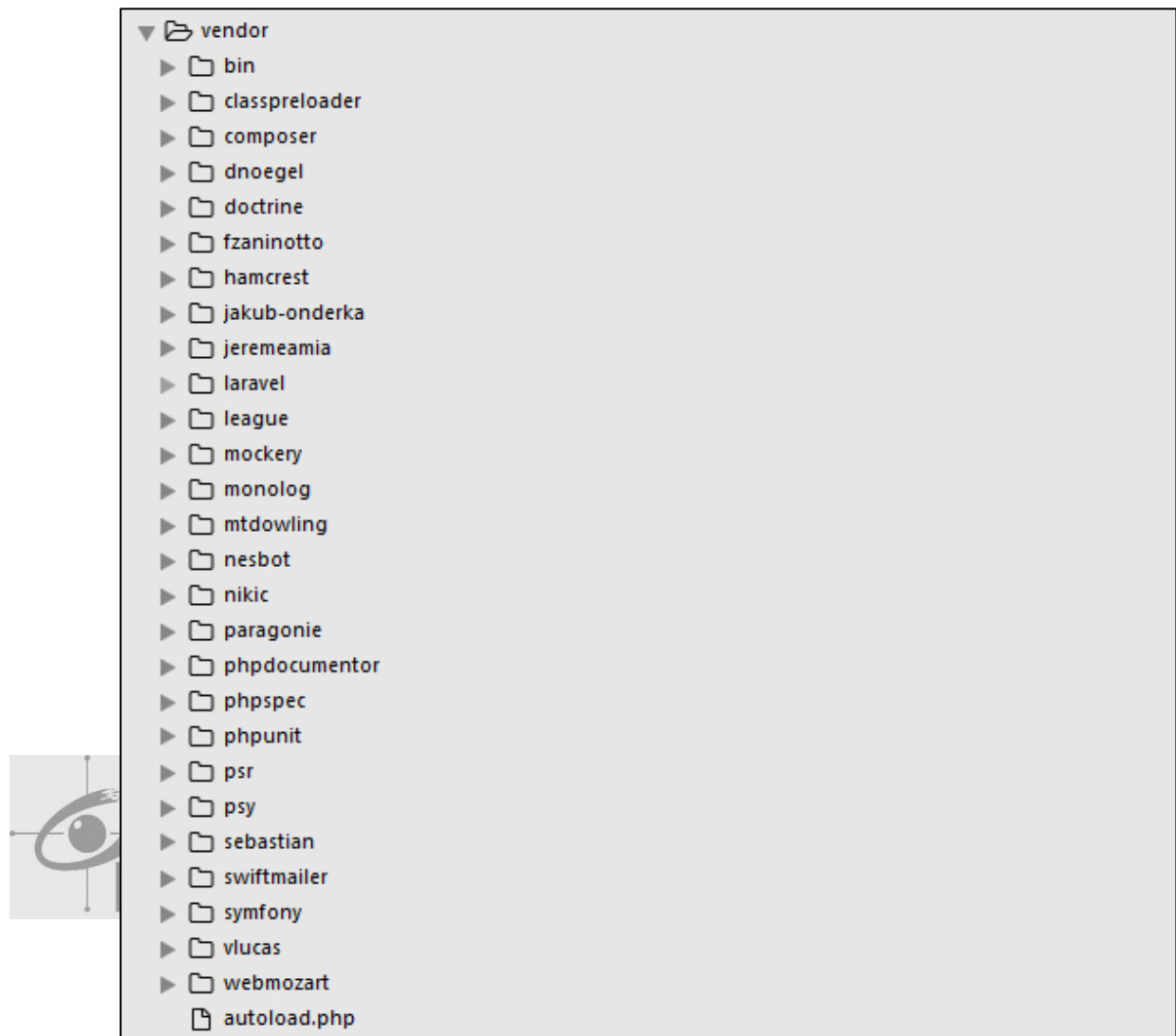
Folder ini berisi cache, data file yang dibuat secara otomatis oleh framework Laravel seperti hasil compile dari blade template, cache, dan session. Di folder ini juga akan terdapat file-file log dari aplikasi.

9. Tests



Folder ini berisi file-file yang berfungsi untuk menyimpan test case dalam tahap testing aplikasi.

10. vendor



Folder ini berisi semua *dependecy* yang dibuat oleh composer.

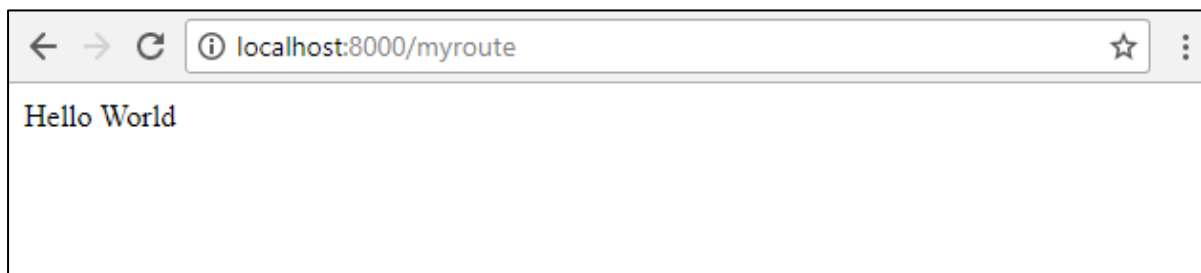
1.5.2 Routing

Routing adalah cara Laravel membaca URI yang di-request oleh pengguna dan menentukan apa yang harus dilakukan sebagai respon sesuai dengan rute yang sudah dibuat oleh pengembang dan *HTTP method*-nya. *HTTP method* adalah metode yang digunakan saat pengguna mengirimkan *request* ke aplikasi. Routing pada Laravel 5.2 dapat diatur di dalam file `routes.php` yang terdapat pada folder `app/Http/`. Setiap URI yang di-request oleh pengguna akan dicocokkan dengan setiap rute yang telah dibuat oleh pengembang dan *HTTP method* -nya, jika terdapat rute dengan *HTTP method* yang sesuai dengan URI yang di-request maka akan dijalankan fungsi yang sudah ditentukan sebagai responnya, fungsi ini dapat berupa fungsi tak bernama (*anonymous function*) yang bisa berisi logika dan dilengkapi dengan *return statement* atau string yang menunjukkan fungsi dari suatu *controller*.

Salah satu contoh pembuatan rute yang sangat sederhana adalah seperti gambar dibawah ini:

```
Route::get('/myroute', function () {
    return 'Hello World';
});
```

Rute di atas akan berjalan apabila terdapat *request* dengan *HTTP method* “GET” ke URI “/myroute”. Contoh pemanggilannya adalah dengan memasukan URI “localhost:8000/myroute” ke *address bar* browser dan kemudian Laravel akan mencari rute yang sesuai dengan *request* tersebut, lalu menjalankan fungsi yang ada di parameter kedua sehingga akan menghasilkan tampilan seperti gambar dibawah ini:



Selain memberikan respon berupa data / variabel, kita juga bisa memberikan respon berupa sebuah halaman web (*view*). Sebagai contoh kita akan menggunakan `welcome.blade.php` yang ada di dalam folder `resource/views` sebagai respon. Rute yang dibuat akan menjadi seperti gambar dibawah ini:

```
Route::get('/myroute', function () {
    return view('welcome');
});
```

Sehingga hasilnya akan seperti ini:



Selain *HTTP method* GET yang sudah dibahas diatas, masih ada beberapa *HTTP method* yang dapat digunakan selama pengembangan aplikasi Laravel. Berikut adalah *HTTP method* yang dapat digunakan:

1. GET
2. POST
3. PUT
4. PATCH
5. DELETE
6. OPTIONS

HTTP method ini akan dibahas lebih lanjut pada BAB 2 subtopik 2.4

Terkadang mungkin diperlukan rute yang akan menerima request dengan lebih dari satu *HTTP method*. Pada Laravel rute seperti ini dapat dilakukan dengan menggunakan kata kunci ***match*** seperti gambar dibawah ini:

```
Route::match(['get', 'post'], '/myroute', function () {  
    return view('welcome');  
});
```

Rute diatas akan memberikan respon berupa halaman welcome.blade.php pada setiap *request* dengan URI “localhost:8000/myroute” dan *HTTP method* “get” ataupun “post”.

Ataupun jika diperlukan rute yang dapat menerima request dengan semua *HTTP method* yang didukung Laravel, rute tersebut dapat dibuat dengan menggunakan kata kunci **any** seperti gambar dibawah ini:

```
Route::any('/myroute', function () {
    return view('welcome');
});
```

Rute diatas akan memberikan respon berupa halaman welcome.blade.php pada setiap *request* dengan URI “localhost:8000/myroute” dan *HTTP method* apapun yang didukung oleh Laravel.

Parameter pada Route

Selain menerima URI, Laravel juga memungkinkan untuk membuat rute yang menerima parameter yang dibutuhkan untuk dapat memberikan respon yang sesuai. Untuk dapat melakukan hal itu, rute dapat dibuat dengan cara seperti gambar dibawah ini:

```
Route::get('/myroute/{name}', function ($name) {
    return 'Hello my name is '.$name;
});
```

Rute diatas akan menerima URI “localhost:8000/myroute/<parameter_nama>” dan parameter yang dikirimkan pada rute akan diterima oleh variabel yang berada pada fungsi, sehingga akan memberikan respon berupa teks bertuliskan “Hello <parameter_nama>”. Contohnya adalah seperti gambar dibawah ini:

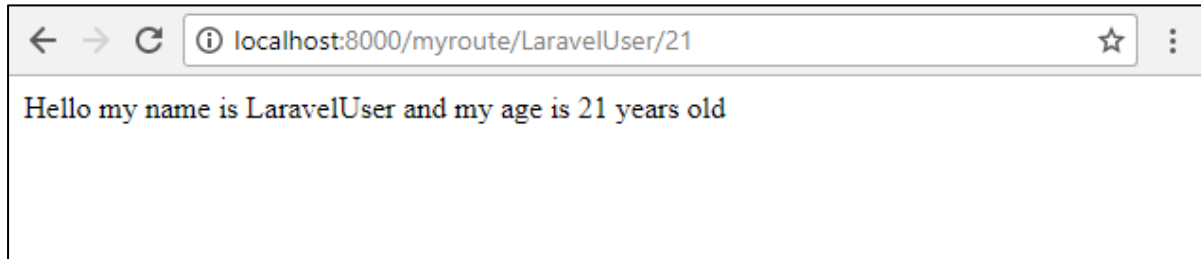


Jika diperlukan lebih dari satu parameter, cukup tambahkan parameter yang diperlukan pada rute yang akan dibuat.

contoh 1

```
Route::get('/myroute/{name}/{age}', function ($name, $age) {
    return 'Hello my name is '.$name.' and my age is '.$age.' years old';
});
```

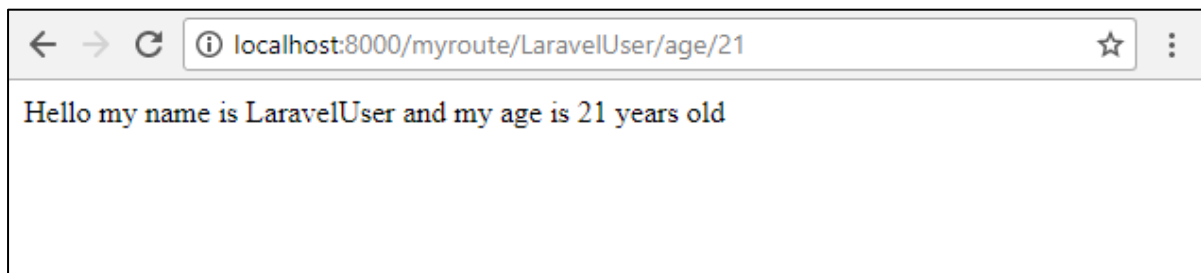
Rute diatas akan menerima URI “localhost:8000/myroute/<parameter_nama>/<parameter_umur>” dan akan memberikan respon berupa teks bertuliskan “Hello <parameter_nama> and my age is <parameter_umur> years old”. Contohnya adalah seperti gambar dibawah ini:



contoh 2

```
Route::get('/myroute/{name}/age/{age}', function ($name, $age) {
    return 'Hello my name is '.$name.' and my age is '.$age.' years old';
});
```

Rute diatas akan menerima URI “localhost:8000/myroute/<parameter_nama>/age/<parameter_umur>” dan akan memberikan respon berupa teks bertuliskan “Hello <parameter_nama> and my age is <parameter_umur> years old”. Contohnya adalah seperti gambar dibawah ini:



Terkadang diperlukan juga rute yang menerima parameter yang bersifat optional. Rute seperti ini dapat dibuat dengan menambahkan **tanda tanya** pada akhir nama parameter dan **nilai default** jika parameter tersebut tidak diterima seperti gambar dibawah ini:

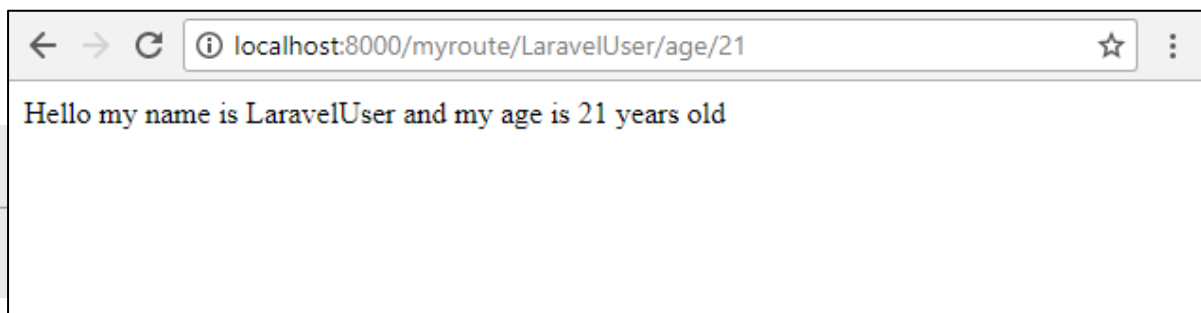

```
Route::get('/myroute/{name}/age/{age?}', function ($name, $age = null) {
    if ($age == null) {
        return 'Hello my name is '.$name;
    }
    return 'Hello my name is '.$name.' and my age is '.$age.' years old';
});
```

Hasil dari rute diatas adalah seperti gambari dibawah ini:

tanpa parameter umur



dengan parameter umur



Rute dengan Nama

Dalam pengembangan aplikasi web, tidak jarang diperlukan rute dan URI yang cukup panjang, untuk memudahkan pemanggilan rute dengan URI yang cukup panjang. Laravel memungkinkan untuk memberikan nama pada setiap rute yang dibuat sehingga dapat dipanggil dengan lebih mudah. Untuk memberikan nama pada rute yang dibuat, dapat dilakukan dengan cara seperti gambar dibawah ini:

```
Route::get('/this-is-my-first-route', function () {
    return 'Hello World';
})->name('myroute');
```

Rute diatas sudah diberi nama “myroute”.

Setelah suatu rute dengan URI yang panjang diberi nama, akan lebih mudah dalam pemanggilan rute tersebut. Untuk memanggil rute yang sudah diberi nama cukup dengan menggunakan **route(<nama_rute>)**. Berikut adalah contoh pemanggilan rute diatas yang sudah diberi nama.

```
Route::get('/gotomyroute', function() {
    return redirect()->route('myroute');
});
```

route('myroute') akan menghasilkan URI “http://localhost:8000/this-is-my-first-route”.

Rute yang Dikelompokkan

Pembuatan rute yang banyak dan tidak teratur tentu akan membuat pengembangan menjadi lebih sulit ketika akan mencari suatu rute yang ingin diubah. Selain itu juga akan sulit dalam menentukan nama bagi rute baru jika sudah banyak rute yang dibuat dan dinamai.

Untuk menangani hal tersebut, Laravel memungkinkan untuk mengelompokkan rute-rute yang ada sesuai kebutuhan dan keinginan pengembang. Pengelompokkan ini dapat dilakukan dengan menggunakan **Route::group** seperti gambar dibawah ini:

```
Route::group(['as' => 'admin::'], function() {
    Route::get('/admin/manage', function() {
        return 'This is administrator management page';
    })->name('manage');

    Route::get('/admin/register', function() {
        return 'This is administrator registration page';
    })->name('register');
});
```

Rute diatas dikelompokkan menjadi satu buah kelompok berdasarkan pengguna, yaitu administrator dan diberikan nama “admin::”. Didalam kelompok admin:: terdapat dua buah rute yang sudah diberi nama juga, yaitu “manage” dan “register”.

Setelah mengelompokkan rute-rute yang ada, sekarang akan lebih mudah untuk memanggil rute-rute tersebut dengan cara route(<nama_kelompok><nama_rute>'). Dalam contoh ini pemanggilan rute dapat dilakukan dengan:

- **route('admin::manage')** => “http://localhost:8000/admin/manage”
- **route('admin::register')** => “http://localhost:8000/admin/register”

1.6 OOP Concept

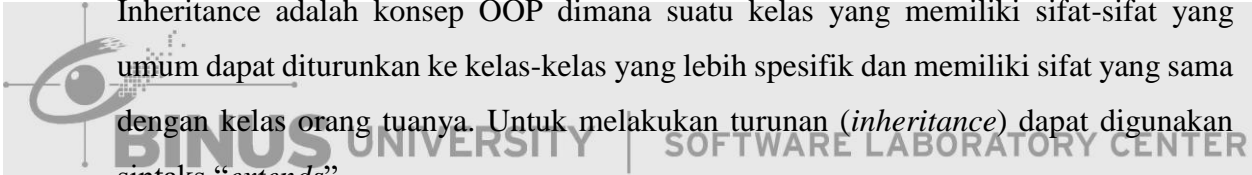
Konsep OOP atau Object Oriented Programming adalah konsep yang sangat erat hubungannya dengan pembentukan *class* dan obyek. Bahasa pemrograman PHP sepenuhnya mendukung penggunaan konsep Object Oriented Programming, begitu juga dengan Laravel yang menggunakan bahasa PHP. Hal ini tentu sangat membantu dalam pengembangan Laravel yang berarsitektur MVC.

Secara umum konsep OOP dibagi menjadi 3 bagian, yaitu

1. Enkapsulasi

Enkapsulasi adalah konsep OOP yang paling mendasar dimana suatu obyek akan dibuat berdasarkan atribut-atribut yang telah dikelompokkan menjadi satu dan dengan hak akses (*access modifier*) yang telah ditentukan. Pengelompokkan ini nantinya akan dibuat dalam bentuk *class*.

2. Inheritance



Inheritance adalah konsep OOP dimana suatu kelas yang memiliki sifat-sifat yang umum dapat diturunkan ke kelas-kelas yang lebih spesifik dan memiliki sifat yang sama dengan kelas orang tuanya. Untuk melakukan turunan (*inheritance*) dapat digunakan sintaks “*extends*”.

3. Polymorphism

Polymorphism merupakan konsep OOP dimana suatu kelas yang bersifat umum dapat dibuat menjadi beberapa jenis obyek yang lebih spesifik. Sebagai contoh, kelas hewan bisa dibuat menjadi obyek kucing, anjing, ataupun kelinci. Penerapan polymorphism ditandai dengan adanya kelas abstrak.

1.7 Blade Templates

Blade adalah *templating engine* yang disediakan oleh Laravel. *Templating engine* ini sangat yang sederhana tetapi cukup *powerful* untuk digunakan. Tidak seperti *templating engine* yang lain, blade tidak membatasi pengembang untuk menggunakan *tag php* yang normal pada *view* dengan mesin blade. Blade bekerja dengan menkompilasikan semua kode dengan template blade menjadi kode *php* normal dan disimpan pada *cache* sampai terdapat perubahan.

Selain blade, masih ada beberapa *templating engine* untuk php lain seperti Moustache, Twig, dan Smarty. Tetapi *default templating engine* yang didukung oleh Laravel adalah mesin blade.

Blade menawarkan fleksibilitas dan efisiensi dengan sintaks-sintaks PHP yang dipersingkat. Untuk dapat menggunakan mesin blade pada *view*, dapat dilakukan dengan mengubah ekstensi file menjadi “.blade.php”.

Berikut adalah contoh perbandingan kode dengan PHP normal dan menggunakan mesin blade:

dengan blade:

```
@php
    $word = 'World';
@endphp
Hello {{ $word }}
```

tanpa blade:

```
<?php
    $word = 'World';
?>
Hello <?= $word ?>
```

Secara *default*, *tag* blade akan menganggap sebuah string sebagai teks biasa meskipun terdapat *tag* HTML di dalam string tersebut. Apabila anda ingin tetap menampilkan *tag* HTML yang terdapat pada sebuah string sesuai dengan fungsinya maka anda dapat menggunakan *tag* berikut.

```
Hello {!! '<h2>World</h2>' !!}
```

Selain *tag* yang sudah dibahas diatas, blade masih menyediakan banyak *tag* yang dapat digunakan untuk memudahkan pengembangan aplikasi. Berikut adalah *tag* yang dapat digunakan pada *view* dengan mesin blade:

1. @extends, @yield, @section

Kebanyakan aplikasi web menggunakan tampilan yang serupa pada setiap halamannya tetapi dengan isi konten yang berbeda sesuai fungsinya masing-masing. Beberapa bagian dari halaman web seperti *header* dan *footer* biasanya menggunakan tampilan yang serupa di setiap halaman. *Tag @extends* berfungsi untuk menurunkan *view* dengan bagian-bagian serupa tersebut pada halaman web yang diinginkan sehingga dapat mengurangi penulisan kode yang berulang-ulang.

Untuk mencobanya marilah kita buat satu folder bernama “layouts” di dalam folder resources/views. Kemudian buatlah satu file bernama “app.blade.php”. file app.blade.php ini akan menjadi tampilan umum yang akan digunakan pada setiap halaman.



Setelah membuat file `app.blade.php`, buatlah kode berikut di dalam file tersebut.

```
<!DOCTYPE html>
<html>
<head>
  <title>FirstLaravelProject - @yield('title')</title>
</head>
<body>
  <ul class="nav-bar">
    @section('nav-bar')
      <li><a href="#"#>Home</a></li>
    @show
  </ul>

  <header>Hello World !!</header>

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

  <footer>Copyright &copy; Software Laboratory Center</footer>
</body>
</html>
```

@section

Menyatakan dan menampilkan suatu bagian konten dari halaman web. Tag **@show** menandakan bahwa *section* yang terakhir dibuat belum selesai atau masih bisa ditambahkan.

@yield

Menandai bagian yang akan menampilkan konten dari **@section** pada halaman lain yang diturunkan dari halaman dengan **@yield** sesuai dengan nama yang telah ditentukan.

Contoh:

halaman *parent*

```
Hello @yield('word')
```

halaman *child*

```
@section('word', 'World !!')
```

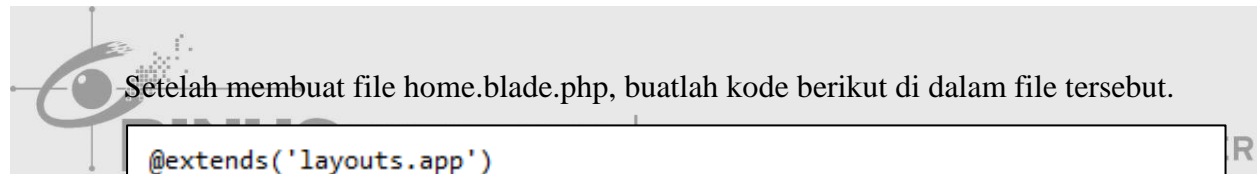
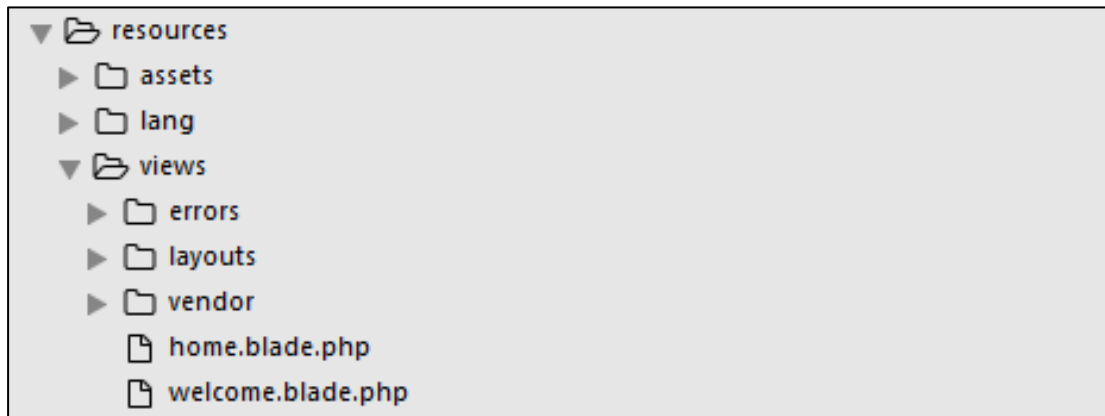
atau

```
@section('word')
  World !!
@endsection
```

hasil pada halaman *child*



Setelah selesai membuat halaman umum (halaman *parent*) yang akan digunakan dibanyak halaman lain. Sekarang buatlah satu file bernama “home.blade.php” didalam resources/views dan diluar folder layouts.



Setelah membuat file home.blade.php, buatlah kode berikut di dalam file tersebut.

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

@section('title', 'Home')

@section('nav-bar')
    @parent

    <li><a href="#">Login</a></li>
    <li><a href="#">Register</a></li>
@endsection

@section('content')
    Ini adalah halaman <strong>Home</strong>
@endsection
```

@extends

Menurunkan kode pada suatu file ke file lain. Dalam contoh ini yang diturunkan adalah file app.blade.php yang ada di dalam folder layouts.

@section

Menyatakan dan menampilkan suatu bagian konten dari halaman web. Dalam contoh ini

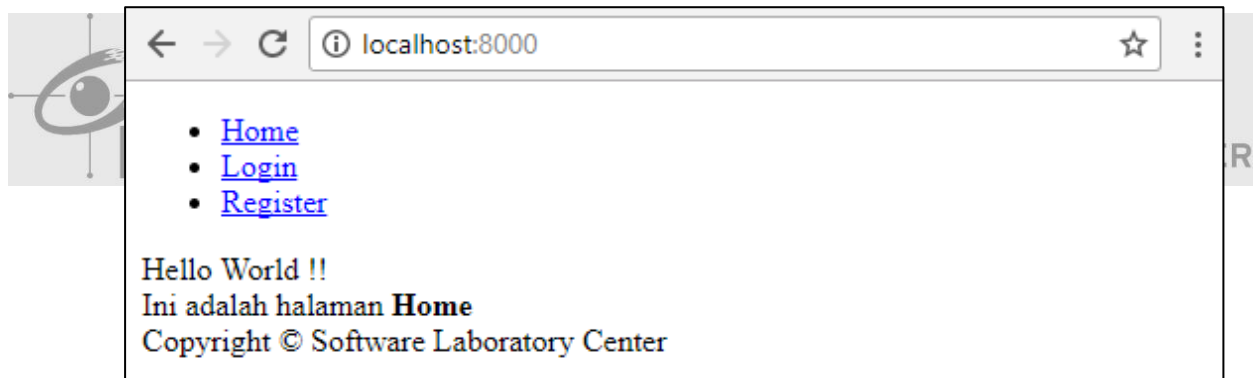
- Bagian **@yield('title')** diganti dengan “Home”.
- Bagian **@section('nav-bar')** ditambahkan dengan menu **Login** dan **Register**.
- Bagian **@yield('content')** diganti dengan “Ini adalah halaman **Home**”.

Tag **@section** hanya akan merubah **@yield** dengan nama yang sesuai, jika halaman tersebut (*child*) sudah diturunkan dari halaman dengan **@yield** (*parent*) dengan menggunakan tag **@extends**. Dalam contoh ini halaman `home.blade.php` diturunkan dari halaman `app.blade.php`.

@parent

Menampilkan isi konten dari *section* yang sudah ada sebelumnya pada file *parent*, yang dalam contoh ini adalah file `app.blade.php`.

Hasil pada halaman `home.blade.php` (*child*) adalah seperti gambar dibawah ini:

**2. @include**

Berfungsi untuk memasukkan isi kode dari suatu file *view* ke dalam file *view* lain. Sebagai contoh, kita akan memisahkan bagian *header* pada file lain dan melakukan *include* terhadap file tersebut.

Buatlah satu file bernama “header.blade.php” di dalam folder layouts.



Setelah selesai membuat file header.blade.php, buatlah kode berikut di dalam file tersebut.

header.blade.php

```
<header>Hello {{ $user or 'Guest' }} !!</header>
```

Penggunaan **or** pada kode diatas berfungsi untuk mengganti nilai yang akan ditampilkan menjadi “Guest” jika variabel **\$user** kosong.

Setelah itu bukalah file app.blade.php, kemudian ubahlah bagian header menjadi seperti berikut:

app.blade.php

```

<!DOCTYPE html>
<html>
<head>
    <title>FirstLaravelProject - @yield('title')</title>
</head>
<body>
    <ul class="nav-bar">
        @section('nav-bar')
            <li><a href="#"#>Home</a></li>
        @show
    </ul>

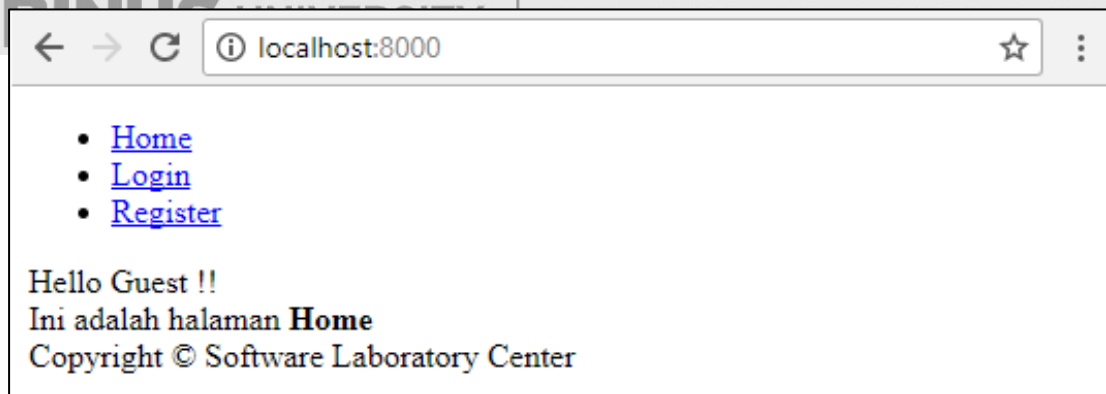
    <!-- <header>Hello World !!</header> -->
    @include('layouts.header')

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

    <footer>Copyright &copy; Software Laboratory Center</footer>
</body>
</html>

```

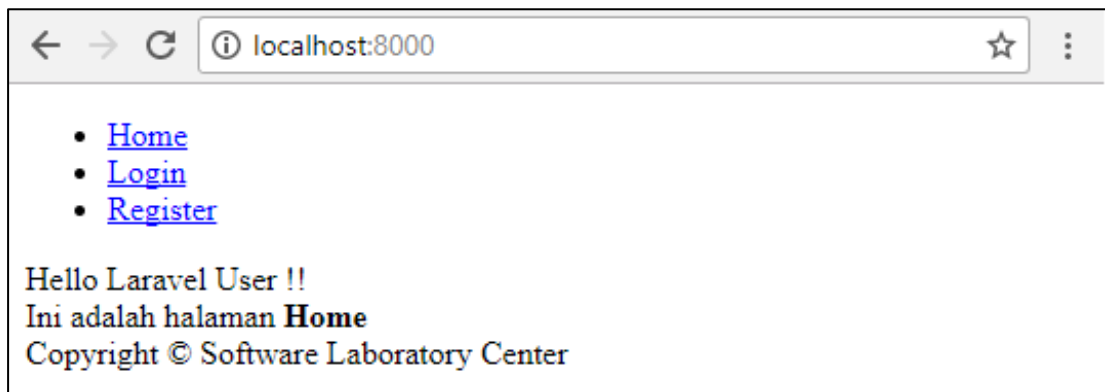
Hasil yang akan ditampilkan pada browser adalah seperti gambar dibawah ini.



Selain itu apabila anda ingin melakukan *include* sambil mengirim data, maka pada tag **@include** dapat ditambahkan parameter kedua yang berbentuk array assosiatif yang dapat diisi dengan data-data yang ingin dikirimkan seperti gambar dibawah ini.

```
@include('layouts.header', ['user' => 'Laravel User'])
```

Hasil yang akan ditampilkan pada browser akan menjadi seperti gambar dibawah ini.



Gambar 6.2 Penggunaan Tag @include dengan Mengirim Data

3. @if, @unless

Tag **@if** dan **@unless** digunakan untuk membuat kondisi pada file blade. Untuk menyatakan lebih dari satu kondisi dengan aksi yang berbeda **@if** dapat dibantu dengan **@elseif** dan **@else**. Tag **@unless** merupakan negasi dari **@if**. Contoh:

```
@unless($user == "admin")
```

sama dengan

```
@if($user != "admin")
```

atau

```
@if( !($user == "admin") )
```

Gambar dibawah ini adalah contoh penggunaan dari:

@if, @elseif, @else

```
@if( $user == "admin" )
    <h1>Welcome Back Admin</h1>
@elseif( $user == "owner" )
    <h1>Welcome Back Owner</h1>
@else
    <h1>Welcome Back {{ $user }}</h1>
@endif
```

@unless

```
@unless($user != "admin")
    <h2>Show Administrator Features</h2>
@endunless
```

4. @while, @for, @foreach

Tag **@while**, **@for**, dan **@foreach** digunakan untuk membuat perulangan pada file blade. Berikut adalah contoh penggunaan dari:

@while

```
@while($i < 5)
    <h2>{{ $i++ }}</h2>
@endwhile
```

@for

```
@for($i = 0 ; $i < 5 ; $i++)
    <h2>{{ $i }}</h2>
@endfor
```

@foreach

```
@foreach(array(0, 1, 2, 3, 4) as $i)
    <h2>{{ $i }}</h2>
@endforeach
```

Ketiga contoh diatas mengeluarkan hasil yang sama yaitu angka 0 sampai 4 dengan ukuran tulisan *header* dua.

5. @php

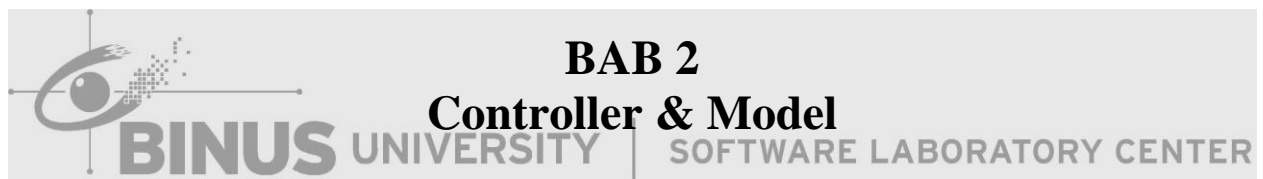
Tag **@php** digunakan untuk dapat menyisipkan potongan kode PHP ke dalam view dengan mesin blade. Berikut adalah contoh penggunaannya:

```
@php
    $word = 'Hello World !!';
    echo $word;
@endphp
```

6. komentar

Untuk menambahkan komentar PHP di dalam view dengan mesin blade dapat menggunakan *tag* seperti dibawah ini.

```
{{-- kalimat ini adalah sebuah komentar --}}
```



Seperti yang sudah dijelaskan pada bab sebelumnya bahwa Laravel menggunakan arsitektur MVC (Model-View-Controller) sebagai dasar dari pembangunan frameworknya. Jika pada bab sebelumnya sudah banyak dibahas mengenai *view*, *blade templating engine*, dan *routing* yang menentukan rute jalannya aplikasi. Pada bab ini yang akan lebih banyak dibahas adalah komponen lain dari arsitektur MVC, yaitu *controller* dan model.

2.1 Controller

Controller adalah sebuah *class* dimana semua logika dari aplikasi akan dibuat. Controller akan memuat semua logika yang menghubungkan model (data) dengan *view* (tampilan), dimana semua logika itu akan dibagi-bagi menjadi fungsi-fungsi sesuai kegunaannya masing-masing. Controller biasanya dibuat berdasarkan obyek yang akan dikontrolnya. Misalkan dalam suatu aplikasi, akan dilakukan proses menambahkan, mengubah, menghapus, atau menampilkan produk-produk yang ada, maka semua proses tersebut akan dibuat di dalam sebuah *controller* bernama “ProductController”.

2.1.1 Introduction

Pada Laravel, sebuah controller dapat dibuat dengan sangat mudah, yaitu dengan menggunakan perintah “**make:controller <nama_controller>**” seperti gambar dibawah ini:

```
D:\FirstLaravelProject>php artisan make:controller ExampleController
```

Setelah menjalankan perintah di atas, maka pada folder `app/Http/Controllers` akan terbuat file baru yaitu `ExampleController.php` yang berisi:

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;

class ExampleController extends Controller
{
    //
}
```

Laravel juga dapat membuat *controller* yang sudah menyediakan fungsi-fungsi yang dapat digunakan untuk proses CRUD (Create, Read, Update, Delete) dengan menjalankan perintah seperti gambar dibawah ini:

```
D:\FirstLaravelProject>php artisan make:controller ExampleController --resource
```

Perintah diatas akan menghasilkan *controller* yang sudah dilengkapi dengan fungsi-fungsi yang dapat digunakan untuk proses CRUD. Fungsi-fungsi yang disediakan diantaranya adalah:

Nama Fungsi	Penjelasan
index	Digunakan untuk menampilkan daftar dari semua data yang ada.
create	Digunakan untuk menampilkan form untuk menambahkan data baru.
store	Digunakan untuk menyimpan data yang sudah ditambahkan pada tempat penyimpanan data / database.
show	Digunakan untuk menampilkan data yang spesifik sesuai dengan id yang diterima.
edit	Digunakan untuk menampilkan form untuk mengubah data yang sudah ada sesuai dengan id yang diterima.
update	Digunakan untuk mengubah data yang diinginkan pada tempat penyimpanan data / database.
destroy	Digunakan untuk menghapus data yang spesifik sesuai dengan id yang diterima dari tempat penyimpanan data / database.

Seperti yang dilihat semua *controller* yang dibuat akan diturunkan dari *class* Controller.

Sebagai contoh awal, kita akan membuat fungsi bernama “foo” di dalam *class* ExampleController:

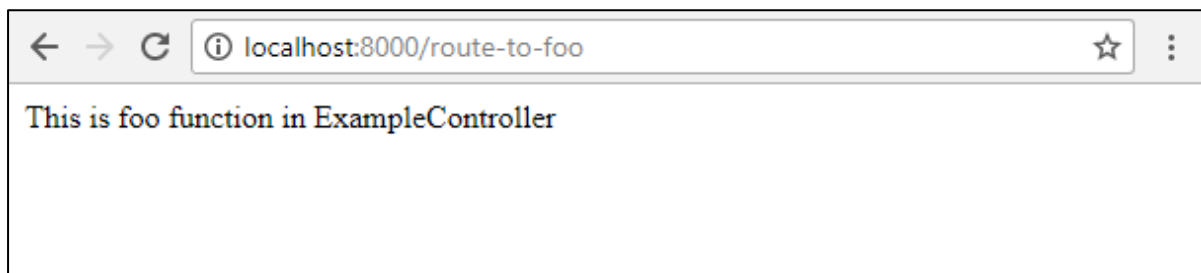
```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
class ExampleController extends Controller
{
    public function foo() {
        return "This is foo function in ExampleController";
    }
}
```

Setelah itu, kita akan membuat rute baru pada file `app/Http/routes.php` yang akan memanggil fungsi `foo` pada ExampleController:

```
Route::get('/route-to-foo', 'ExampleController@foo');
```

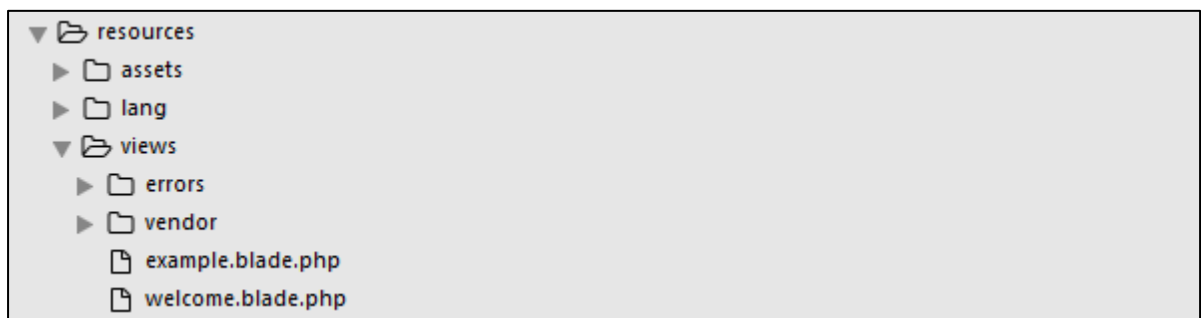
Fungsi yang ada di parameter kedua pada rute bisa diganti dengan string yang merujuk pada nama *controller* dan fungsi yang ada di dalam *controller* tersebut dengan memisahkannya menggunakan karakter “@”. Yang pada contoh diatas adalah rute tersebut akan merujuk pada ExampleController dan fungsi foo. Pastikan string pada parameter kedua ini benar-benar merujuk pada suatu fungsi.

Hasilnya bisa kita buka di browser dengan mengakses URI “localhost:8000/route-to-foo” seperti gambar dibawah ini:



Apabila kita ingin memunculkan suatu halaman web melalui *controller* maka kita dapat menggunakan function **view**(‘<nama_view>’).

Sebagai contoh buatlah sebuah halaman web yang ingin dimunculkan pada folder resources/views. Misalnya halaman bernama example.blade.php.



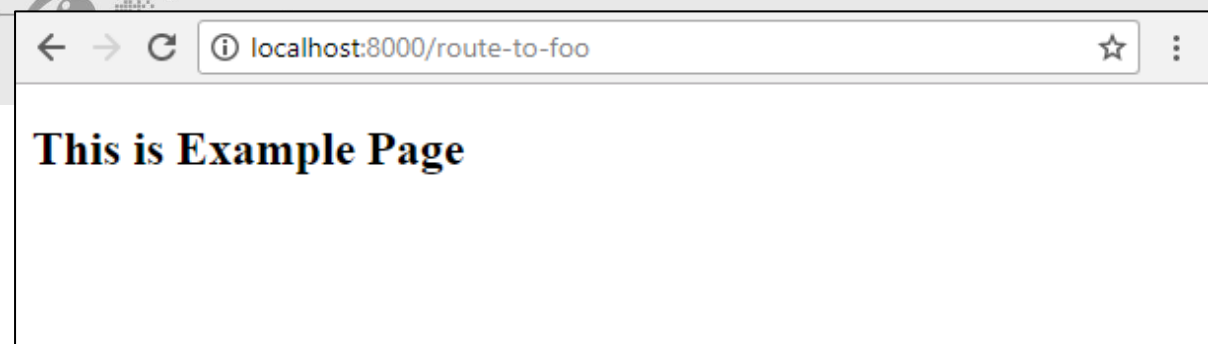
Setelah membuat file `example.blade.php`, buatlah kode dibawah ini didalam file tersebut:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>This is Example Page</h2>
</body>
</html>
```

Setelah selesai membuat halaman web yang ingin ditampilkan, ubahlah definisi dari fungsi `foo` pada file `ExampleController.php` menjadi seperti gambar dibawah ini:

```
class ExampleController extends Controller
{
    public function foo() {
        return view('example');
    }
}
```

Maka hasil yang akan ditampilkan pada browser akan menjadi seperti gambar dibawah ini:



2.1.2 Mengirimkan Data dari *Controller* ke *View*

Dalam pengembangan aplikasi tidak jarang diperlukan pengiriman data yang diperlukan ke dalam tampilan yang diinginkan. Terdapat 2 cara yang bisa dipakai untuk mengirimkan data ke dalam *view* melalui *controller*:

1. Menggunakan array asosiatif

Untuk mengirimkan data menggunakan array asosiatif, dapat dilakukan dengan cara menambahkan parameter pada fungsi `view` berupa array asosiatif yang diisi dengan semua data yang ingin dikirimkan. Contohnya adalah seperti gambar dibawah ini:

```
public function foo(){
    return view('example', ['nama'=>'Eric', 'umur'=> 3] );
}
```

Fungsi view diatas akan mengirimkan data dengan kata kunci “nama” bernilai “Eric” dan kata kunci “umur” bernilai 3.

2. Menggunakan fungsi compact

Untuk mengirimkan data dengan menggunakan fungsi compact, dapat dilakukan dengan mengganti array asosiatif dengan fungsi **compact**(‘<nama_parameter_1>’, ‘<nama_parameter_2>’, ...). Contohnya adalah seperti gambar dibawah ini:

```
class ExampleController extends Controller
{
    public function foo() {
        $nama = "Eric";
        $umur = 3;

        return view('example', compact("nama", "umur"));
    }
}
```

Fungsi compact akan secara otomatis membuat array asosiatif dengan nama index (kata kunci) dari parameteranya dan berisi variabel dengan nama yang sama, sehingga contoh di atas akan sama dengan array asosiatif seperti gambar dibawah ini:

```
public function foo() {
    $nama = "Eric";
    $umur = 3;

    return view('example', ['nama' => $nama, 'umur' => $umur]);
}
```

3. Menggunakan fungsi with

Untuk mengirimkan data dengan menggunakan fungsi with, dapat dilakukan dengan menambahkan fungsi **with**(‘<kata_kunci>’, ‘<nilai>’) diakhir fungsi view. Contohnya adalah seperti gambar dibawah ini:

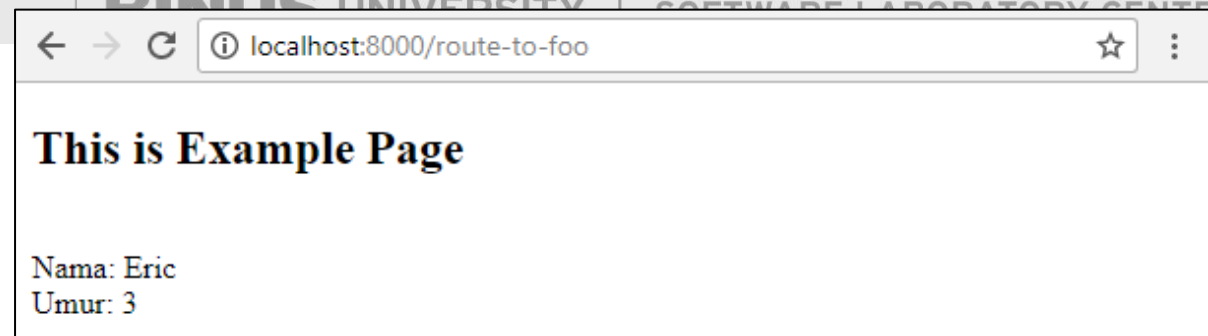
```
public function foo() {
    return view('example')
        ->with('nama', 'Eric')
        ->with('umur', 3);
}
```

Fungsi `foo` diatas akan mengembalikan respon berupa tampilan halaman `example.blade.php` dengan membawa data dengan kata kunci “nama” bernilai “Eric” dan kata kunci “umur” bernilai 3.

Setelah mengirimkan data ke *view* yang ditentukan melalui *controller*, sekarang kita dapat mengakses data tersebut menggunakan variabel dengan nama yang sesuai dengan kata kunci yang sudah ditentukan saat mengirimkan data, yang dalam contoh ini adalah nama dan umur. Sebagai contoh ubahlah isi file `example.blade.php` menjadi seperti gambar dibawah ini:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>This is Example Page</h2>
    <br>
    Nama: {{ $nama }}
    <br>
    Umur: {{ $umur }}
</body>
</html>
```

Hasil yang akan ditampilkan pada browser akan menjadi seperti gambar dibawah ini:



Selain dapat mengirimkan sebuah nilai seperti yang sudah dicontohkan, ketiga cara diatas juga dapat digunakan untuk mengirimkan array ataupun obyek. Sebagai contoh, tambahkanlah variabel hobi berbentuk array di dalam fungsi `foo` dan mengirimkannya menggunakan fungsi `compact`, seperti gambar dibawah ini:

```
public function foo() {
    $nama = "Eric";
    $umur = 3;
    $hobi = [ "Makan", "Dota", "Nonton" ];

    return view('example', compact("nama", "umur", "hobi"));
}
```

Kemudian tambahkanlah kode untuk menampilkan data hobi tersebut di dalam file `example.blade.php` menjadi seperti gambar dibawah ini:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>This is Example Page</h2>
    <br>
    Nama: {{ $nama }}
    <br>
    Umur: {{ $umur }}
    <br>
    <ul>
        @foreach($hobi as $item)
            <li>{{ $item }}</li>
        @endforeach
    </ul>
</body>
</html>
```

Hasil yang akan ditampilkan pada browser akan menjadi seperti gambar dibawah ini:



Karena fungsi `foo` pada `ExampleController.php` akan langsung mengembalikan respon berupa halaman web, maka akan cukup sulit mencari error / bug pada fungsi tersebut. Pada kondisi ini, fungsi **dd()** yang merupakan gabungan dari **var_dump()** dan **die()** dapat digunakan. Fungsi **dd()** digunakan untuk menampilkan data yang dimasukan sebagai parameter dan menghentikan proses yang sedang berjalan, jika tidak ada data yang dimasukan ke dalam parameter fungsi ini hanya akan menghentikan proses yang sedang berjalan.

Sebagai contoh, kode dibawah ini akan menampilkan nilai dari variabel nama, umur, dan hobi yang berbentuk array:

```
public function foo() {  
    $nama = "Eric";  
    $umur = 3;  
    $hobi = [ "Makan", "Dota", "Nonton" ];  
  
    dd($nama, $umur, $hobi);  
  
    return view('example', compact("nama", "umur", "hobi"));  
}
```

Hasil yang ditampilkan pada browser akan menjadi seperti gambar dibawah ini:



Jika fungsi dd() dipanggil atau digunakan maka semua kode yang ada dibawahnya tidak akan dijalankan lagi (proses yang sedang berjalan akan dihentikan).

2.1.3 Mengirimkan Data dari View ke Controller

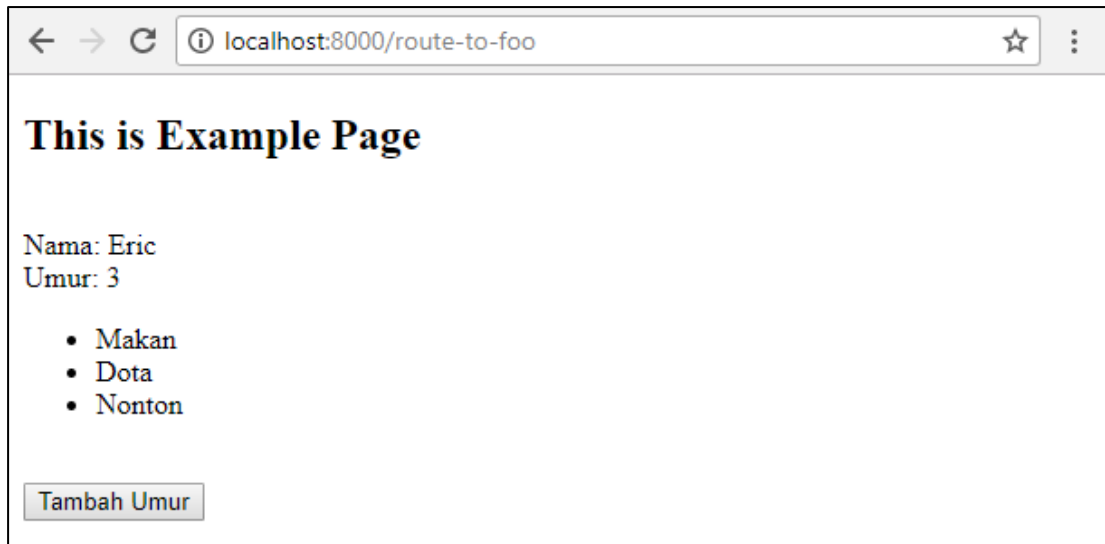
Kita telah mempelajari cara bagaimana *controller* bisa mengirim data ke *view* seperti contoh diatas. Apabila kita ingin melakukan yang sebaliknya, yaitu mengirimkan data dari *view* kepada *controller*, maka ada 2 cara yang dapat digunakan:

1. Parameter pada route

Seperti yang sudah dijelaskan pada Bab 1, bahwa route dapat digunakan untuk mengirimkan data melalui parameternya. Sebagai contoh, tambahkan tombol “Tambah Umur” pada file `example.blade.php` menjadi seperti gambar dibawah ini:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>This is Example Page</h2>
    <br>
    Nama: {{ $nama }}
    <br>
    Umur: {{ $umur }}
    <br>
    <ul>
        @foreach($hobi as $item)
            <li>{{ $item }}</li>
        @endforeach
    </ul>
    <br>
    <a href="{{ URL::to( '/route-to-foo/' . ($umur + 10) ) }}">
        <button>Tambah Umur</button>
    </a>
</body>
</html>
```

URL di atas akan mengirimkan data umur yang didapat dari variabel umur yang diterima dari *controller* dan ditambahkan dengan 10. Hasil yang akan ditampilkan pada browser akan menjadi:



Untuk bisa dapat mengirimkan data umur yang telah kita tambahkan, kita perlu mengubah rute `route-to-foo` agar dapat menerima parameter umur. Rute `route-to-foo` akan menjadi seperti gambar dibawah ini:

```
Route::get('/route-to-foo/{umur?}', 'ExampleController@foo');
```

Rute diatas akan menerima parameter umur yang sifatnya optional. Maksudnya adalah rute diatas akan menerima URI **“localhost:8000/route-to-foo”** atau **“localhost:8000/route-to-foo/<parameter_umur>”**.

Setelah mengubah rute `route-to-foo` kita juga perlu mengubah fungsi `foo` pada `ExampleController` menjadi seperti gambar dibawah ini:

```
public function foo($param_umur = 3) {
    $nama = "Eric";
    $umur = $param_umur;
    $hobi = [ "Makan", "Dota", "Nonton" ];

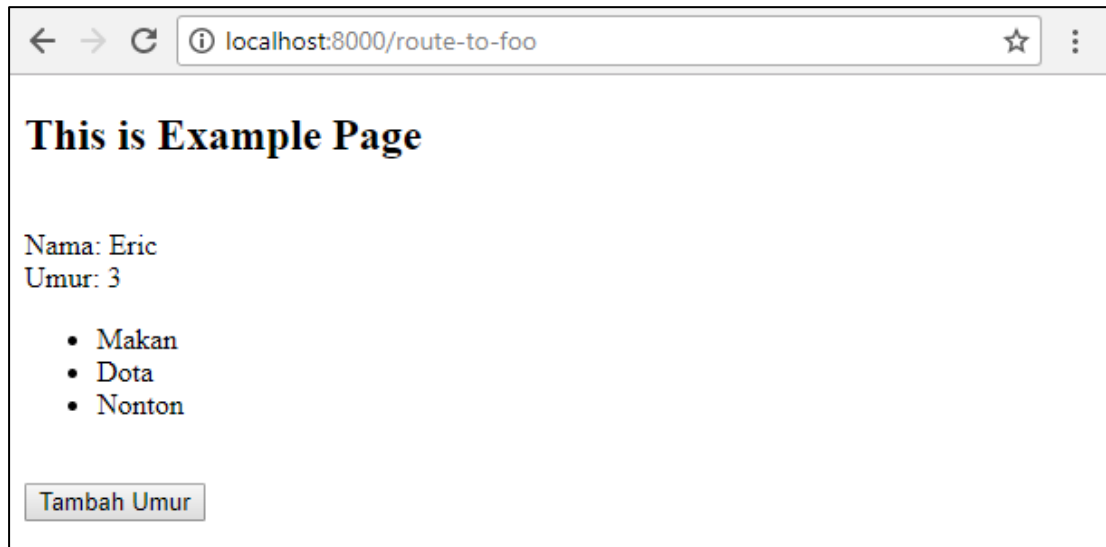
    return view('example', compact("nama", "umur", "hobi"));
}
```

Fungsi `foo` diatas akan menerima parameter umur dari *view* sesuai dengan rute yang sudah dibuat. Karena parameter umur dibuat dengan sifat optional pada rute, sehingga parameter umur pada fungsi `foo` harus diberikan nilai *default*, yang pada contoh diatas nilai *default*-nya adalah 3.

Setelah selesai mengubah fungsi `foo`, sekarang jika tombol **“Tambah Umur”** yang telah anda buat tadi ditekan, maka umur yang diterima dari *controller* sebelumnya akan

ditambahkan dengan 10 kemudian dikirimkan kembali ke *controller*, dan *controller* akan mengirimkan variabel umur tersebut kembali ke *view*. Sehingga umur yang akan ditampilkan pada browser adalah umur yang sudah ditambahkan dengan 10. Contoh hasilnya adalah seperti gambar dibawah ini:

before



after



2. Http Request

Http request atau sering disebut request dapat digunakan untuk mengirimkan data dari *view* ke *controller* dengan menggunakan 2 cara berdasarkan metode dari request tersebut dikirimkan.

1. GET

Pengiriman data dengan menggunakan metode GET sering disebut juga sebagai query string. Cara ini dapat dilakukan dengan cara menempatkan parameter request pada url. Sebagai contoh, misalnya data nama dan umur akan dikirimkan dari view ke controller menggunakan request dengan metode GET, hal tersebut dapat dilakukan dengan memasukan url seperti gambar dibawah ini:



Url diatas akan mengakses rute register dan mengirimkan parameter request berupa data nama yang bernilai “Laravel User” dan umur yang bernilai 20.

Note: %20 diantara Laravel dan User mewakili karakter spasi.



Atau bisa juga dilakukan dengan membuat form seperti gambar dibawah ini:

```
<form method="GET" action="{{ URL::to('/register') }}">
  <label>Nama :</label> <input type="text" name="nama">
  <br>
  <label>Umur :</label> <input type="text" name="umur">
  <br>
  <input type="submit" value="Register">
</form>
```

Form diatas akan menghasilkan url yang sama dengan contoh sebelumnya ketika tombol register ditekan.

2. POST

Pengiriman data dengan metode POST berbeda dengan metode GET, jika dengan metode GET semua parameter request yang dikirimkan terlihat pada url, dengan metode POST parameter itu tidak akan dikirimkan ataupun ditampilkan pada url. Contohnya penggunaan metode POST adalah pada form untuk registrasi, data yang terdapat pada form tersebut akan ada bermacam-macam dan tidak ingin terlihat pada url (misalnya password tidak mungkin ditampilkan pada url). Metode POST dapat digunakan dengan menggunakan cara membuat form seperti gambar dibawah ini:

```
<form method="POST" action="{{ URL::to('/register') }}">
    {!! csrf_field() !!}
    <label>Nama :</label> <input type="text" name="nama">
    <br>
    <label>Umur :</label> <input type="text" name="umur">
    <br>
    <input type="submit" value="Register">
</form>
```

Form diatas akan mengirimkan semua data yang ada pada form ke *controller* sesuai dengan rute yang ditentukan dan menggunakan metode POST ketika tombol register ditekan.

Setelah membuat form untuk mengirim data, rute pengiriman data juga perlu dibuat agar data dapat dikirimkan ke *controller* yang sesuai. Untuk contoh diatas rute yang dibuat akan menjadi seperti gambar dibawah ini:

Untuk metode GET

```
Route::get('/register', 'ExampleController@register');
```

Untuk metode POST

```
Route::post('/register', 'ExampleController@register');
```

Kedua rute diatas akan mengarahkan url 'localhost:3000/register' ke fungsi register pada ExampleController yang akan dibuat.

Selanjutnya buatlah fungsi register dengan parameter request pada ExampleController seperti gambar dibawah ini:

```
public function register(Request $request) {
    $nama = $request->nama;
    $umur = $request->umur;

    $hobi = [ "Makan", "Dota", "Nonton" ];

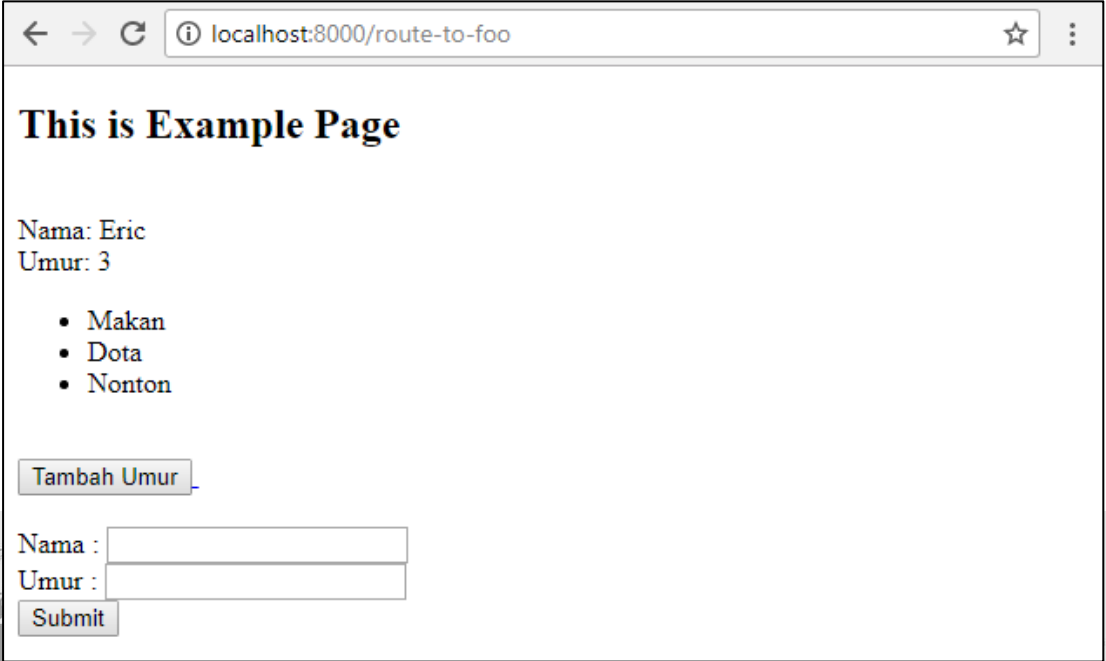
    return view('example', [
        'nama' => $nama,
        'umur' => $umur,
        'hobi' => $hobi,
    ]);
}
```

Bisa dilihat variabel request akan berisi data-data dari form yang telah dikirim. Data-data yang terdapat pada form dapat diakses sesuai dengan atribut "*name*" pada elemen

html pada halaman web. Selain menggunakan operator *single arrow* ('->'), kita juga mengakses data-data tersebut dengan menggunakan *brackets* ('[]') seperti penggunaan array.

Hasil dari statement di atas adalah sebagai berikut:

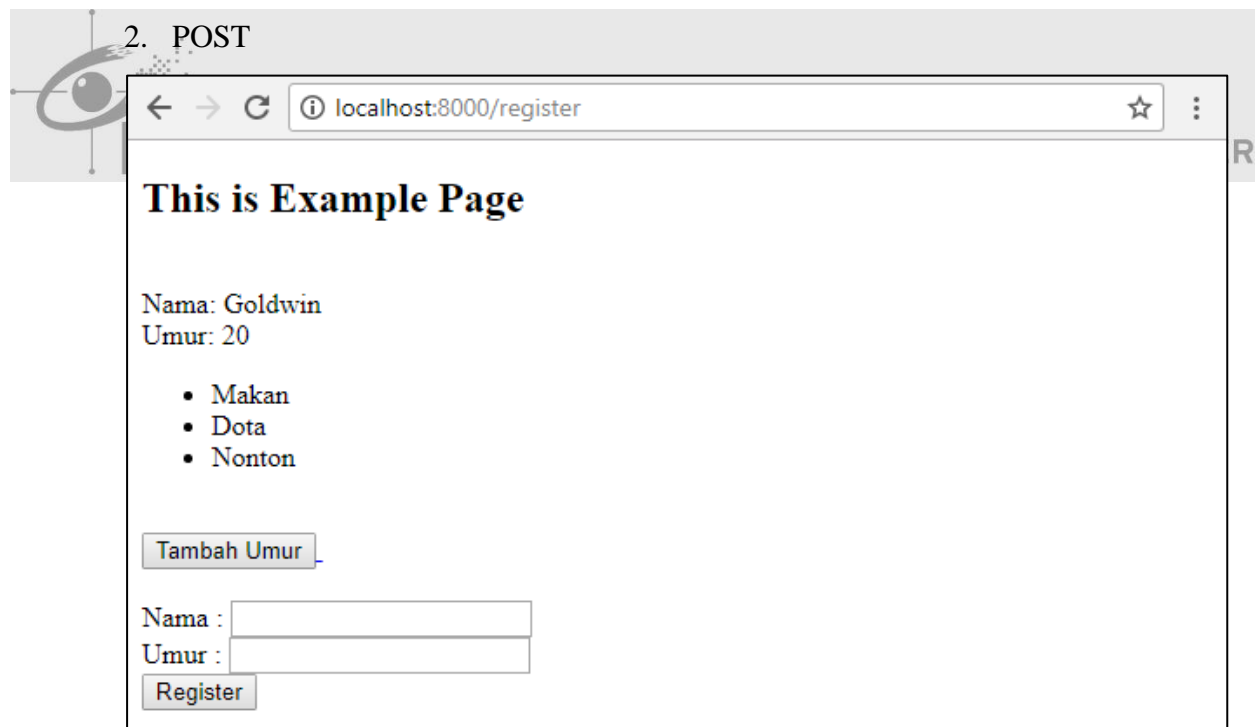
Before



The screenshot shows a web browser window with the address bar displaying 'localhost:8000/route-to-foo'. The page content includes a heading 'This is Example Page', followed by the text 'Nama: Eric' and 'Umur: 3'. Below this is a bulleted list containing 'Makan', 'Dota', and 'Nonton'. A button labeled 'Tambah Umur' is positioned above a form section. The form section contains two input fields labeled 'Nama :' and 'Umur :', and a 'Submit' button.

After**1. GET**

A screenshot of a web browser window. The address bar shows the URL `localhost:8000/register?nama=Goldwin&umur=20`. The page content includes the heading "This is Example Page", the text "Nama: Goldwin" and "Umur: 20", a bulleted list with items "Makan", "Dota", and "Nonton", a "Tambah Umur" button, and a registration form with input fields for "Nama" and "Umur", and a "Register" button.

2. POST

A screenshot of a web browser window. The address bar shows the URL `localhost:8000/register`. The page content is identical to the GET request result, showing the heading "This is Example Page", the text "Nama: Goldwin" and "Umur: 20", a bulleted list with items "Makan", "Dota", and "Nonton", a "Tambah Umur" button, and a registration form with input fields for "Nama" and "Umur", and a "Register" button.

Kedua form diatas adalah hasil dari form dengan nama yang diinput “Goldwin” dan umur yang diinput “20” yang sudah dikirimkan ke *controller*.

2.1.4 Jenis-Jenis Controller Response

Seperti yang sudah dibahas bahwa *controller* adalah lokasi dimana logika dari suatu aplikasi ditempatkan dan berfungsi untuk mengatur jalannya aplikasi. *Controller* banyak memiliki kegunaan dalam pengembangan suatu aplikasi, sehingga controller dituntut untuk dapat mengembalikan nilai (*response*) dengan beberapa jenis, Beberapa jenis pengembalian nilai (*response*) tersebut diantaranya adalah:

1. Data Umum

Controller yang digunakan untuk mengembalikan data biasa sudah pernah dilihat saat membahas *controller* pertama kali. Contoh penggunaannya adalah untuk mendapatkan nama pengguna dari suatu id yang diinginkan.

```
return $nama
```

2. View

Contoh penggunaan *controller* untuk mengembalikan *view* sudah banyak dicontohkan diatas. Pengembalian *view* digunakan oleh *controller* yang bertujuan untuk menampilkan suatu halaman web baik dengan data maupun tanpa data yang dikirimkan melalui *controller*. Contoh penggunaannya adalah seperti gambar dibawah ini:

tanpa data

```
return view('example');
```

dengan data

```
return view('example', [
    'nama' => $nama,
    'umur' => $umur,
    'hobi' => $hobi,
]);
```

3. Redirect

Fungsi `redirect()` digunakan pada *controller* dengan tujuan untuk mengarahkan pengguna ke rute lain untuk mengatur alur jalan dari suatu proses. Contohnya adalah jika pengguna sudah menginput data registrasi dengan benar maka *controller* akan menambahkan data pengguna baru dan diarahkan ke rute untuk menampilkan halaman login.


```
return redirect('/login');
```

Selain redirect ke suatu url atau rute, Laravel juga menyediakan fungsi back(). Fungsi back() digunakan pada controller dengan tujuan untuk menampilkan halaman sebelumnya yang memicu fungsi tersebut. Contoh penggunaan fungsi back() pada *controller* adalah untuk mengembalikan halaman register jika terdapat suatu inputan yang salah atau tidak sesuai.

```
return back();
```

4. Json

Fungsi json() digunakan pada *controller* dengan tujuan untuk mengembalikan suatu nilai atau obyek dalam bentuk JSON. JSON adalah *JavaScript Object Notation*, json adalah suatu data berupa string yang merepresentasikan suatu obyek dengan notasi sesuai aturan JSON.



```
return response()->json([  
    'nama' => 'Eric',  
    'umur' => 20  
]);
```

R

5. Download

Fungsi download() digunakan pada *controller* dengan tujuan untuk mengembalikan *response* yang membuat browser mendownload suatu file berdasarkan lokasi yang ditentukan.

```
return response()->download($lokasi_file);
```

2.2 Model / Eloquent Model

Setelah membahas tentang *controller*, selanjutnya akan dibahas tentang komponen dari MVC yang terakhir yaitu Model. Model adalah representasi data-data yang ada pada database. Model digunakan untuk membantu dalam mengakses data pada database. Pada Laravel, Model terdapat pada folder “app/” dan juga sudah dilengkapi dengan *Eloquent* sehingga sering disebut sebagai *Eloquent Model*.

Eloquent adalah suatu *object-relational mapper (ORM)* yang disediakan Laravel. ORM ini bertujuan untuk merepresentasikan struktur dari suatu database menjadi struktur obyek yang direlasikan sehingga menyerupai struktur database aslinya. *Eloquent* menyediakan fungsi-fungsi yang berguna untuk merepresentasikan query-query yang biasa dijalankan sehingga lebih mudah untuk dipahami. Meskipun begitu *Eloquent* tidak membatasi user untuk menggunakan query pada umumnya.

2.2.1 Introduction

Pada Laravel model dapat dibuat dengan mudah menggunakan perintah “**make:model <nama_model>**” seperti gambar dibawah ini:

```
D:\FirstLaravelProject>php artisan make:model Story
```

Perintah diatas akan menghasilkan suatu model Story pada folder “app/” yang merepresentasikan table Story pada database yang akan dibuat. Isi dari model Story yang baru saja dibuat adalah seperti gambar dibawah ini:

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class Story extends Model
{
    //
}
```

Pada saat membuat model, terdapat parameter tambahan yang dapat digunakan untuk mendefinisikan table pada database yang akan direpresentasikan oleh model tersebut. Contoh penggunaannya adalah seperti gambar dibawah ini:

```
D:\FirstLaravelProject>php artisan make:model Story -m
```

Parameter `-m` atau yang bisa diganti dengan `-migration` ini berfungsi untuk membuat file migration yang berguna untuk mendefinisikan table yang akan dibuat pada database. Penjelasan mengenai migration akan dibahas lebih lanjut pada subbab 2.3.1.

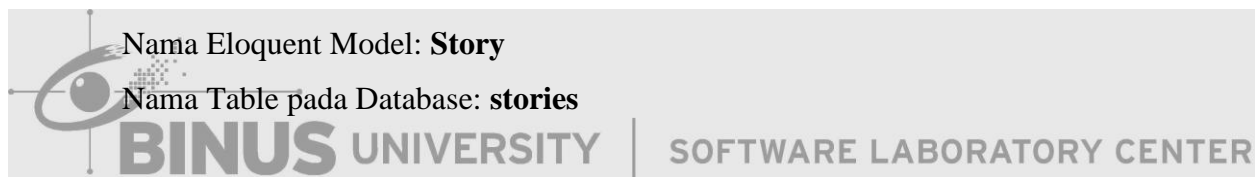
2.2.2 Ketentuan *Eloquent*

Eloquent disediakan oleh Laravel untuk dapat memudahkan pengembang dalam mendeklarasi dan memanipulasi database yang digunakan pada aplikasi. Untuk dapat memanfaatkan fitur-fitur dari *Eloquent*, *Eloquent* memiliki beberapa ketentuan yang harus diikuti. Berikut adalah beberapa ketentuan dari *Eloquent*:

1. Nama Table pada Database

Untuk menghubungkan model dengan table pada database *Eloquent* memiliki ketentuan tersendiri yaitu, “*snake case and plural name*” dari nama model akan digunakan sebagai nama untuk mengidentifikasi nama table pada database.

Contoh:



Nama Eloquent Model: **Story**

Nama Table pada Database: **stories**

Nama Eloquent Model: **ZipCode**

Nama Table pada Database: **zip_codes**

Dengan mengikuti aturan ini, pengembang tidak perlu memikirkan mengenai table mana akan diwakilkan dengan model yang mana. Meskipun begitu Laravel tidak membatasi pengembang untuk harus mengikuti ketentuan tersebut. Jika ketentuan tersebut tidak diikuti, maka pengembang harus memberi tahu model mengenai table yang akan direpresentasikan. Hal tersebut dapat dilakukan dengan menambahkan kode seperti gambar dibawah ini:

```
protected $table = 'nama_table';
```

2. Primary Key

Selain mana table dan model, *Eloquent* Laravel juga mengasumsikan bahwa setiap table pada database akan memiliki kolom *primary key* bernama *id* dengan tipe data *int* dan bersifat *autoincrement*. Sama seperti nama table, *Eloquent* tidak memaksakan

pengembang untuk harus mengikuti aturan ini, jika pengembang ingin menggunakan nama lain untuk diidentifikasi sebagai primary key, maka pengembang harus mengatur ulang properti primary key, dengan cara seperti gambar dibawah ini:

```
protected $primarykey = 'nama_primary_key';
```

jika pengembang ingin menggunakan primary key dengan tipe data lain selain *int* atau tidak ingin kolom *primary key*-nya bersifat *autoincrement*, maka pengembang harus mengatur ulang properti *incrementing*, dengan cara seperti gambari dibawah ini:

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

3. Timestamps

Secara *default*, Eloquent akan mengasumsikan bahwa table yang direpresentasikan-nya mengandung kolom “created_at” dan “updated_at” yang mengidentifikasi tanggal dan waktu satu buah data dibuat dan diubah terakhir kali. Jika pengembang tidak menginginkan adanya kolom-kolom ini, pengembang dapat menambahkan kode seperti gambar dibawah ini:

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



BINUS UNIVERSITY

SOFTWARE LABORATORY CENTER

2.2.3 Hubungan antar Model / Eloquent Model Relationships

Table-table pada database biasanya berhubungan antar satu dengan yang lain, misalnya table pengguna berhubungan dengan table transaksi penjualan, atau table produk berhubungan dengan table jenis produk. *Eloquent* membuat hubungan-hubungan ini dapat diatur dengan mudah pada model. Dengan menggunakan *Eloquent* pengembang tidak perlu memikirkan query *join* untuk membuat hubungan antar table. *Eloquent* menyatakan hubungan-hubungan ini dalam fungsi-fungsi yang merepresentasikan hubungannya, sehingga untuk menghubungkan suatu table dengan table lain dapat dilakukan dengan memanggil *Eloquent dynamic property* untuk hubungan tersebut. *Eloquent dynamic property* adalah properti dari suatu model yang memungkinkan pengembang untuk memanggil fungsi hubungan yang sudah dibuat. Contohnya adalah seperti gambar dibawah ini:

```
$address = User::find(1)->address;
```

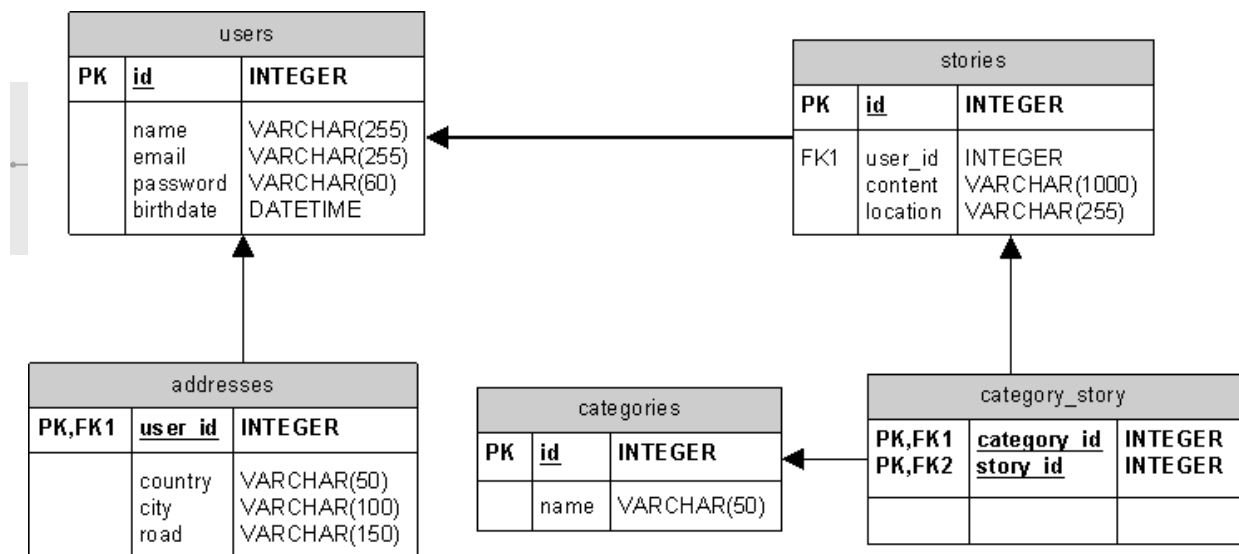
Query diatas digunakan untuk mendapatkan alamat dari pengguna yang memiliki id 1.

Eloquent mendukung beberapa tipe hubungan antar table diantaranya adalah:

- a. *One to One*
- b. *One to Many*
- c. *Many to Many*
- d. *Has Many Through*
- e. *Polymorphic Relations*
- f. *Many to Many Polymorphic Relations*

Dari keenam tipe hubungan diatas, sebenarnya hubungan yang paling dasar adalah *one to one* dan *one to many*. Tipe hubungan yang lain merupakan tipe hubungan yang dapat dibuat dengan menggunakan dua hubungan dasar ini.

Untuk membantu dalam menjelaskan tentang hubungan-hubungan diatas, akan digunakan rancangan database seperti gambar ERD dibawah ini:



Penjelasan:

Dalam diagram database diatas seorang *user* hanya dapat memiliki satu alamat, dan satu alamat hanya akan dimiliki oleh seorang *user*, setiap *user* bisa menambahkan banyak *story* (seperti tweet pada twitter) sedangkan satu *story* sudah pasti dimiliki oleh seorang *user*. Untuk setiap *story* dapat ditambahkan banyak *category* yang menggambarkan *story* tersebut dan setiap *category* dapat memiliki banyak *story*.

Sebelum mulai mendefinisikan hubungan antar table seperti pada ERD diatas, buatlah model untuk masing-masing table yang ada pada ERD tersebut. Model-model tersebut dapat dibuat dengan menggunakan perintah seperti gambar dibawah ini:

```
D:\FirstLaravelProject>php artisan make:model User
D:\FirstLaravelProject>php artisan make:model Address
D:\FirstLaravelProject>php artisan make:model Story
D:\FirstLaravelProject>php artisan make:model Category
```

a. *One to One*

Hubungan *one to one* adalah tipe hubungan dimana satu buah data pada suatu table hanya memiliki hubungan dengan satu buah data pada table lain dan sebaliknya. Contoh penggunaannya pada ERD diatas adalah hubungan antar table *users* dan *addresses*. Setiap data pengguna hanya akan memiliki hubungan dengan satu data alamat, dan sebaliknya setiap data alamat hanya akan memiliki hubungan dengan satu data pengguna.

Hubungan ini dapat didefinisikan dengan menambahkan fungsi *address()* pada model *User* yang menyatakan hubungannya dengan model *Address*. Contohnya adalah seperti gambar dibawah ini:

```
public function address()
{
    return $this->hasOne(Address::class);
}
```

Untuk menyatakan hubungan sebaliknya, maka dapat ditambahkan fungsi *user()* pada model *Address* seperti gambar dibawah ini:

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

b. *One to Many*

Hubungan *one to many* adalah tipe hubungan dimana satu buah data pada suatu table dapat memiliki banyak hubungan dengan data pada table lain. Contoh penggunaannya

pada ERD diatas adalah hubungan antar table *users* dan *stories*. Setiap data pengguna dapat memiliki banyak hubungan dengan data *stories* (seorang pengguna bisa menambahkan banyak *story*).

Hubungan ini dapat didefinisikan dengan menambahkan fungsi *stories()* pada model *User* yang menyatakan hubungannya dengan model *Story*. Contohnya adalah seperti gambar dibawah ini:

```
public function stories()
{
    return $this->hasMany(Story::class);
}
```

Untuk menyatakan hubungan sebaliknya, maka dapat ditambahkan fungsi *user()* pada model *Story* seperti gambar dibawah ini:

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

c. Many to Many

Hubungan *many to many* adalah tipe hubungan dimana satu buah data pada table A dapat memiliki banyak hubungan dengan data pada table B dan data pada table B tersebut dapat memiliki banyak hubungan dengan data pada table A. Contoh penggunaannya pada ERD diatas adalah hubungan antar table *stories* dan *categories*. Setiap data *story* dapat memiliki banyak hubungan dengan data *categories*.

Hubungan ini dapat didefinisikan dengan menambahkan fungsi *stories()* pada model *Category* yang menyatakan hubungannya dengan model *Story*. Contohnya adalah seperti gambar dibawah ini:

```
public function stories()
{
    return $this->belongsToMany(Story::class);
}
```

Untuk menyatakan hubungan sebaliknya, maka dapat ditambahkan fungsi *categories()* pada model *Story* seperti gambar dibawah ini:

```

public function categories()
{
    return $this->hasMany(Category::class);
}

```

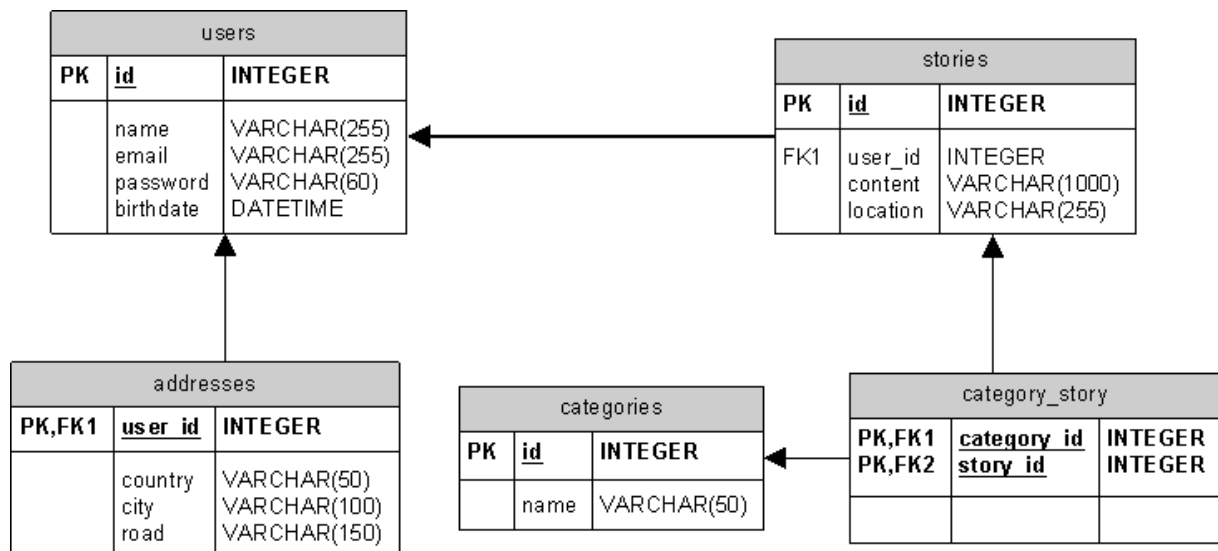
2.3 Migration dan Seeding

2.3.1 Migration

Migration dapat diartikan sebagai suatu jenis *version control* pada Laravel untuk mengatur database yang digunakan. *Version control* adalah suatu sistem yang mencatat semua perubahan pada suatu dokumen atau kumpulan dokumen, yang dalam hal ini adalah suatu database. Migration memungkinkan sebuah tim dapat dengan mudah mengubah dan berbagi rancangan database dari aplikasi.

Keuntungan dari menggunakan migration daripada mendefinisikan table-table secara langsung pada DBMS adalah kompatibilitas yang tinggi karena dapat digunakan untuk berbagai jenis DBMS dan berbagi dengan lebih mudah.

Untuk membantu penjelasan tentang penggunaan migration pada subbab ini, maka akan digunakan rancangan database seperti pada ERD dibawah ini:



Penjelasan:

Dalam diagram database diatas seorang *user* dapat memiliki lebih dari satu alamat, tetapi satu alamat hanya akan dimiliki oleh seorang *user*, setiap *user* bisa menambahkan banyak *story* (seperti tweet pada twitter) sedangkan satu *story* sudah pasti dimiliki oleh seorang *user*. Untuk

setiap *story* dapat ditambahkan banyak *category* dan setiap *category* dapat memiliki banyak *story*.

Setiap tabel yang dibuat, akan diwakilkan oleh 1 buah file migration. Migration dapat dibuat dengan menggunakan perintah “**make:migration <nama_migration>**” seperti gambar dibawah ini:

```
D:\FirstLaravelProject>php artisan make:migration create_users_table
```

Jika perintah diatas dijalankan maka akan terbuat satu buah file migration yang terdapat pada folder “database/migrations”. Nama file migration dilengkapi dengan *timestamp* yang membantu Laravel dalam menentukan urutan dari migration yang akan dijalankan. Pada contoh ini file yang terbuat akan memiliki format:

“**yyyy_mm_dd_hhMMss_create_users_table.php**”

yyyy => tahun migration dibuat
 mm => bulan migration dibuat
 dd => tanggal migration dibuat
 hh => jam migration dibuat
 MM => menit migration dibuat
 ss => detik migration dibuat

Pada saat menjalankan perintah untuk membuat migration diatas, terdapat beberapa parameter yang dapat digunakan, diantaranya adalah:

1. --create

Parameter --create menandakan bahwa migration yang dibuat akan digunakan untuk membuat table baru. Dengan menambahkan parameter **--create=<nama_table>**, maka pada file migration yang terbuat akan dilengkapi dengan template untuk membuat table baru sesuai dengan nama table yang dimasukan pada parameter.

```
D:\FirstLaravelProject>php artisan make:migration create_users_table --create=users
```

2. --table

Parameter --table digunakan untuk menspesifikasi nama table pada database yang akan diatur / dimanipulasi. Berbeda dengan parameter --create, parameter --table hanya akan

membuatkan template yang mengandung nama table sesuai dengan yang dimasukkan pada parameter (tidak menyediakan template membuat table baru).

```
D:\FirstLaravelProject>php artisan make:migration create_users_table --table=users
```

3. --path

Parameter `--path` digunakan untuk menspesifikasi lokasi file migration ingin disimpan. Jika parameter ini tidak dituliskan, maka file migration yang terbuat akan secara *default* diletakkan pada folder “database/migrations”.

Setelah mengetahui cara untuk membuat migration pada Laravel, dengan menggunakan struktur database yang sama, buatlah migration untuk membuat table baru berdasarkan pada table-table yang ada pada database tersebut.

```
php artisan make:migration create_addresses_table --create=addresses
php artisan make:migration create_stories_table --create=stories
php artisan make:migration create_categories_table --create=categories
php artisan make:migration create_category_story_table --create=category_story
```

Untuk table *users*, secara *default* proyek Laravel sudah membuatkan migration untuk table *users* sehingga tidak perlu dibuat lagi.

Karena file migration yang terbuat akan dilengkapi dengan *timestamp* untuk menentukan migration mana yang akan dijalankan terlebih dahulu, maka urutan perintah “**make:migration**” yang benar menjadi salah satu syarat agar migration dapat berjalan dengan baik. Yang dimaksud urutan migration yang benar adalah dengan tidak membuat table dengan *foreign key* yang merujuk pada *primary key* dari suatu table yang **belum dibuat**.

Contoh:

1. Table *users* harus dibuat terlebih dulu sebelum membuat table *addresses* ataupun *stories*, karena kedua table tersebut membutuhkan data dari table *users* sebagai *foreign key*-nya
2. Table *stories* dan *categories* harus dibuat terlebih dulu sebelum membuat table *category_story*, karena kedua table tersebut dibutuhkan oleh table *category_story* sebagai *foreign key*-nya

Setelah semua perintah untuk membuat migration diatas berhasil dijalankan maka akan terbuat file-file migration seperti gambar dibawah ini:



Setelah semua migration telah selesai dibuat, selanjutnya adalah menentukan definisi dari masing-masing table sesuai kebutuhan. Pada setiap file migration yang terbuat, akan disediakan dua buah fungsi secara *default* yang akan dijelaskan dengan tabel berikut ini:

Fungsi	Keterangan
up	Digunakan dengan tujuan untuk menambahkan table, kolom, atau index baru ke dalam database.
down	Digunakan dengan tujuan untuk mengembalikan operasi yang dilakukan oleh fungsi <i>up</i> ke keadaan semua sebelum fungsi <i>up</i> dijalankan.

Sebagai contoh untuk masing-masing migration akan ditentukan definisinya sama seperti gambar ERD diatas.

- **create_users_table**

Table

users		
PK	<u>id</u>	INTEGER
	name	VARCHAR(255)
	email	VARCHAR(255)
	password	VARCHAR(60)
	birthdate	DATETIME

Migration


```

public function up()
{
    Schema::create('users', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name');
        $table->string('email')->unique();
        $table->string('password', 60);
        $table->date('birthdate');
        $table->rememberToken();
        $table->timestamps();
    });
}

```

Fungsi *create* di atas akan membuat table baru dengan nama sesuai pada **parameter pertama** (dalam contoh ini adalah “*users*”) dan definisi sesuai dengan atribut-atribut yang terdapat didalam fungsi pada **parameter kedua**.

Untuk menambahkan atribut baru pada table yang akan dibuat cukup tambahkan kode dengan format:

`$table-><type_data>('<nama_kolom>, ...)`

`$table` => mewakili table yang akan dibuat
`type_data` => tipe data dari atribut / kolom baru yang akan dibuat
`nama_kolom` => nama atribut / kolom baru yang akan dibuat
`...` => parameter lain yang disesuaikan dengan tipe data

Daftar tipe data yang dapat dipakai pada migration dapat dilihat dalam dokumentasi [Migration](#).

- **create_addresses_table**

Table

addresses		
PK,FK1	<u>user_id</u>	INTEGER
	country	VARCHAR(50)
	city	VARCHAR(100)
	road	VARCHAR(150)

Migration

```

public function up()
{
    Schema::create('addresses', function (Blueprint $table) {
        $table->integer('user_id')->unsigned();
        $table->primary('user_id');
        $table->foreign('user_id')->references('id')->on('users');
        $table->string('country', 50);
        $table->string('city', 100);
        $table->string('road', 150);
        $table->timestamps();
    });
}

```

Pada migration di atas kita ingin membuat kolom `user_id` yang merupakan foreign key yang menunjuk tabel `users`. Kita juga bisa menambahkan foreign key constraint seperti *on delete* atau *on update* dengan cara seperti statement di atas.

- **create_stories_table**

Table

stories		
PK	<u>id</u>	INTEGER
FK1	user_id content location	INTEGER VARCHAR(1000) VARCHAR(255)

Migration

```

public function up()
{
    Schema::create('stories', function (Blueprint $table) {
        $table->increments('id');

        $table->integer('user_id')->unsigned();
        $table->foreign('user_id')
            ->references('id')
            ->on('users');

        $table->string('content', 1000);
        $table->string('location');

        $table->timestamps();
    });
}

```

- **create_categories_table**

Table

categories		
PK	<u>id</u>	INTEGER
	name	VARCHAR(50)

Migration

```
public function up()
{
    Schema::create('categories', function (Blueprint $table) {
        $table->increments('id');
        $table->string('name', 50);
        $table->timestamps();
    });
}
```

- **create_category_story_table**

Table

category_story		
PK,FK1	<u>category_id</u>	INTEGER
PK,FK2	<u>story_id</u>	INTEGER

Migration

```
public function up()
{
    Schema::create('category_story', function (Blueprint $table) {
        $table->integer('category_id')->unsigned();
        $table->foreign('category_id')->references('id')->on('categories');

        $table->integer('story_id')->unsigned();
        $table->foreign('story_id')->references('id')->on('stories');

        $table->primary(['category_id', 'story_id']);

        $table->timestamps();
    });
}
```

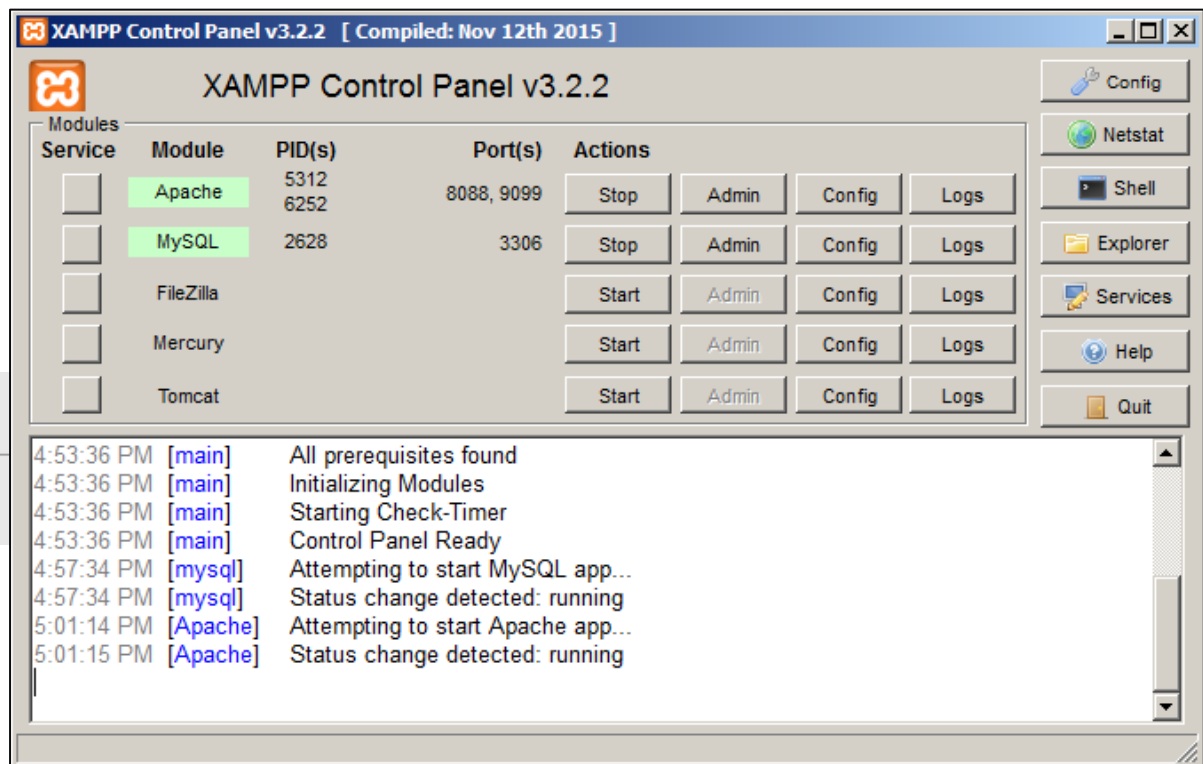
Setelah selesai menentukan definisi dari setiap migration, maka table-table yang akan dibuat pada setiap migration siap untuk dibuat di dalam database.

Sebelum mulai membuat table, pastikan dahulu konfigurasi database pada file .env sudah benar dan database yang ditentukan pada file .env sudah terbuat. Dalam contoh ini koneksi database yang akan digunakan adalah *mysql* dengan database **prk**. Untuk di lab *username* dan *password*

yang digunakan untuk mysql adalah **prk** dan **prk**. Jika XAMPP diinstall pada komputer lain, secara *default username* yang digunakan adalah *root* dan *password*-nya kosong.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_DATABASE=prk
DB_USERNAME=prk
DB_PASSWORD=prk
```

Pastikan juga koneksi MySQL dan Apache telah diaktifkan melalui XAMPP control panel / *web server control* lainnya.



Setelah koneksi database sudah siap, maka kita sudah dapat menjalankan migration yang sudah dibuat dengan menggunakan perintah seperti gambar dibawah ini:

```
D:\FirstLaravelProject>php artisan migrate
```

Perintah diatas akan menjalankan setiap migration yang telah dibuat dan memanggil fungsi *up* untuk membuat table-table baru pada database yang sudah ditentukan. Jika migration berhasil dijalankan, maka akan tampil pesan seperti gambar dibawah ini:

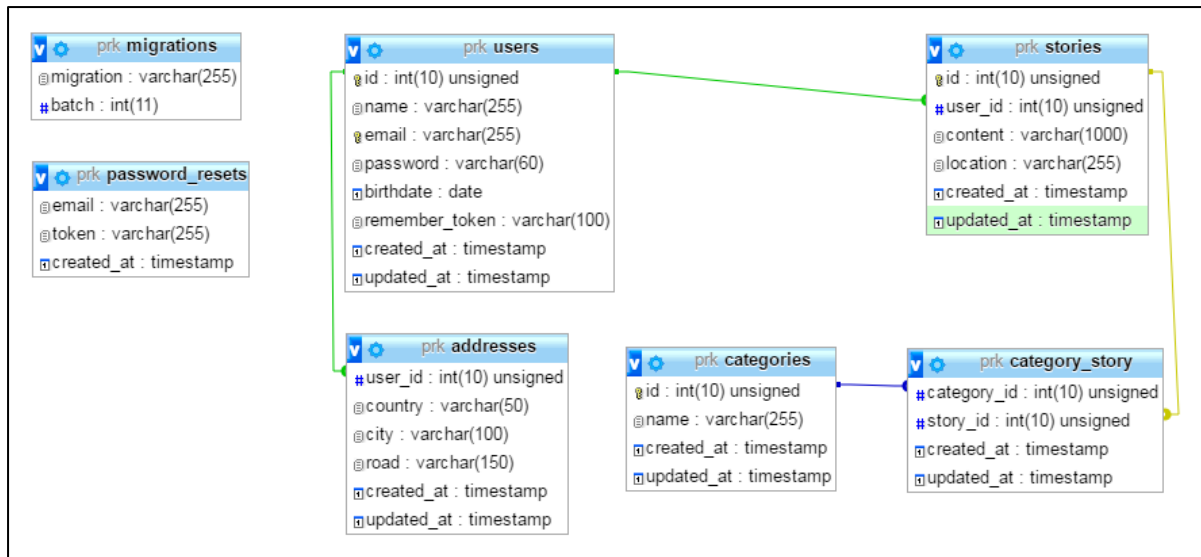
```
Migration table created successfully.
Migrated: 2014_10_12_000000_create_users_table
Migrated: 2014_10_12_100000_create_password_resets_table
Migrated: 2017_03_14_075114_create_stories_table
Migrated: 2017_03_15_080429_create_categories_table
Migrated: 2017_03_15_080500_create_category_story_table
Migrated: 2017_03_15_080519_create_addresses_table
```

Untuk memastikan table-table pada migration benar-benar terbuat, maka dapat dicek dengan membuka url “localhost:8088/phpmyadmin” dan memilih database **prk**:

Table	Action	Rows	Type	Collation	Size	Overhead
addresses	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	32 KiB	-
categories	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	16 KiB	-
category_story	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	48 KiB	-
migrations	★ Browse Structure Search Insert Empty Drop	6	InnoDB	utf8_unicode_ci	16 KiB	-
password_resets	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	16 KiB	-
stories	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	32 KiB	-
users	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8_unicode_ci	16 KiB	-
7 tables	Sum	6	InnoDB	latin1_swedish_ci	176 KiB	0 B

Jika dilihat hasilnya terbuat satu buah table tambahan yang tidak ada dalam folder “database/migrations” yaitu table “*migrations*”. Table ini terbuat secara otomatis ketika perintah “**migrate**” dijalankan. Table ini akan berisi sejarah dibuatnya migration-migration yang ada berdasarkan batch / gelombang dari migration tersebut dijalankan.

Untuk memastikan bahwa table yang terbuat sudah terhubung sesuai dengan yang sudah didefinisikan, maka hubungan antar table dapat dilihat pada menu *designer*:



Selain perintah untuk menjalankan fungsi *up*, terdapat juga perintah-perintah lain yang dapat digunakan untuk mengatur migration, beberapa perintah-perintah tersebut diantaranya adalah:

Perintah	Keterangan
migrate:rollback	Digunakan untuk mengembalikan kondisi semula sebelum operasi migration terakhir dijalankan. Parameter <code>--step</code> dapat digunakan untuk menentukan seberapa banyak batch dari operasi migration akan dikembalikan. Contoh: <code>php artisan migrate:rollback --step=3</code> Perintah diatas akan mengembalikan operasi migration yang telah dijalankan sebanyak 3 batch.
migrate:reset	Digunakan untuk mengembalikan kondisi semua sebelum semua operasi migration dijalankan.
migrate:refresh	Digunakan untuk mengembalikan kondisi semua sebelum semua operasi migration dijalankan dan menjalankan perintah migrate kembali.

2.3.2 Seeding

Seeding adalah kegiatan untuk mengisi database dengan data testing. Laravel menyediakan metode yang mudah untuk melakukan seeding pada database dengan menggunakan kelas-kelas untuk mengisi table-table pada database. Kelas-kelas ini disebut sebagai *seeder*. Setiap *seeder* yang dibuat akan terdapat didalam folder “database/seeds”

Seeder dapat dibuat dengan menggunakan perintah “**make:seeder <nama_seeder>**” seperti gambar dibawah ini:

```
php artisan make:seeder UsersTableSeeder
```

Perintah diatas akan menghasilkan satu buah file seeder dengan nama UsersTableSeeder didalam folder “database/seeds”. Isi dari file tersebut adalah seperti gambar dibawah ini:

```
<?php
use Illuminate\Database\Seeder;

class UsersTableSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        //
    }
}
```

Setiap file *seeder* yang dibuat akan dilengkapi dengan fungsi *run*. Fungsi ini akan dipanggil saat *seeder* dijalankan, sehingga data-data testing yang ingin dimasukkan ke dalam database harus didefinisikan didalam fungsi ini.

Berikut adalah cara untuk memasukkan data ke dalam database menggunakan *Query Builder* yang akan didefinisikan didalam fungsi *run*:

```
DB::table('users')->insert([
    'name' => 'Admin',
    'email' => 'admin@mail.com',
    'password' => 'admin',
    'birthdate' => Carbon::now(),
]);
```

Query builder diatas akan memasukkan data user dengan nama = “**Admin**”, email = “**admin@mail.com**”, password = “**admin**”, birthdate = **tanggal saat seeder dijalankan**.

Untuk menjalankan file *seeder* yang telah dibuat, maka *seeder* tersebut harus didaftarkan pada file DatabaseSeeder.php dengan cara seperti gambar dibawah ini:

```
<?php
use Illuminate\Database\Seeder;

class DatabaseSeeder extends Seeder
{
    /**
     * Run the database seeds.
     *
     * @return void
     */
    public function run()
    {
        $this->call(UsersTableSeeder::class);
    }
}
```

Fungsi *call()* diatas akan menjalankan kelas *UsersTableSeeder* dan memanggil fungsi *run* didalamnya sehingga data yang sudah disiapkan sebelumnya dapat dimasukkan ke dalam database.

Setelah semua seeder telah didaftarkan, maka *seeder* dapat dijalankan dengan menggunakan perintah “**db:seed**” seperti gambar dibawah ini:

```
php artisan db:seed
```

Perintah diatas akan menjalankan semua file seeder yang telah didaftarkan pada file *DatabaseSeeder.php*. Jika hanya ingin menjalankan satu buah file *seeder* saja dapat dispesifikasi dengan menggunakan parameter *--class* seperti gambar dibawah ini:

```
php artisan db:seed --class=UsersTableSeeder
```

Perintah diatas hanya akan menjalankan file *seeder* yang sudah ditentukan saja. Dalam contoh ini yang akan dijalankan adalah *UsersTableSeeder*.

2.4 CRUD (Create, Read, Update, Delete)

Setelah mempelajari semua komponen dari arsitektur MVC secara masing-masing, pada subbab ini akan dibahas tentang alur kerja MVC dalam melakukan operasi menambahkan (*create*), membaca / menampilkan (*read*), mengubah (*update*), dan menghapus (*delete*) data dalam database. Sebagai contoh, akan digunakan model User untuk melakukan operasi-operasi tersebut.

Dalam melakukan operasi-operasi diatas, tentu tidak terlepas dari *request* dari *view*. Pada subbab ini akan dibahas lebih lanjut lagi tentang metode request yang dapat digunakan dalam melakukan operasi-operasi diatas, diantaranya adalah:

1. GET

Metode GET digunakan untuk menerima informasi dari suatu server yang telah ditentukan menggunakan URI yang dimasukan.

2. POST

Metode POST digunakan untuk mengirimkan informasi atau data ke suatu server yang telah ditentukan menggunakan URI yang dimasukan.

3. PUT / PATCH

Metode PUT / PATCH digunakan untuk mengganti semua nilai dari suatu sumber daya yang telah ditentukan dengan nilai baru yang telah diinput.

4. DELETE

Metode DELETE digunakan untuk menghapus suatu sumber daya yang telah ditentukan menggunakan URI yang dimasukan.

Setelah memahami metode request diatas, selanjutnya akan dibahas cara kerja MVC dalam melakukan operasi CRUD. Pertama-tama buatlah komponen-komponen dari MVC untuk model User.

Model

```
D:\FirstLaravelProject>php artisan make:model User
```

```
<?php
namespace App;

use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    //
}
```

Controller

```
D:\FirstLaravelProject>php artisan make:controller UserController
```

```
<?php
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;

class UserController extends Controller
{
    //
}
```

View

Buatlah folder “user” di folder “resources/views”. Kemudian buatlah file *view* untuk mendukung proses CRUD didalam folder user yang baru saja dibuat. File *view* yang dibuat diantaranya adalah:

1. index.blade.php

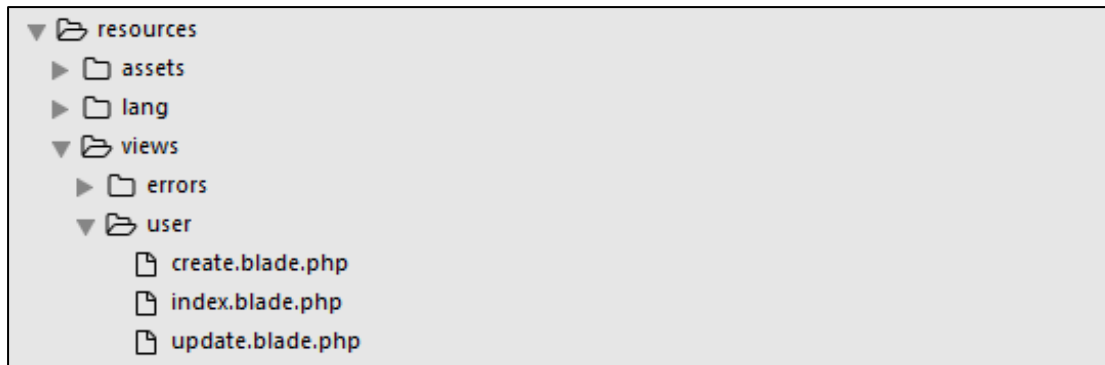
Digunakan untuk menampilkan semua data *user* yang sudah ada di dalam database.

2. create.blade.php

Digunakan untuk membuat data user baru yang akan dimasukkan ke dalam database.

3. update.blade.php

digunakan untuk mengubah data user yang sudah ada, sesuai dengan datab baru yang akan diinput.



2.4.2 Menampilkan Data (read)

Untuk dapat menampilkan data user, akan ditambahkan fungsi **index()** pada UserController. Fungsi ini akan berfungsi untuk mendapatkan semua data *user* dan dikirimkan ke *view* untuk ditampilkan.

```
public function index()
{
    $users = User::all();

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

Karena kelas User tidak berada pada folder yang sama dengan UserController maka lokasi kelas User perlu didefinisikan dengan cara menambahkan kode “**use <lokasi_kelas_User>**” diluar kelas UserController seperti pada gambar dibawah ini:

```
use App\User;

class UserController extends Controller
```

Setelah menambahkan fungsi **index()** untuk menampilkan semua data *user*, selanjutnya buatlah kode berikut ini pada halaman **index.blade.php**

```

<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>User - Index</h2>
    <br>
    <table border="1">
        <thead>
            <td>Name</td>
            <td>Email</td>
            <td>Birthdate</td>
        </thead>
        @foreach($users as $user)
            <tr>
                <td>{{ $user->name }}</td>
                <td>{{ $user->email }}</td>
                <td>{{ $user->birthdate }}</td>
            </tr>
        @endforeach
    </table>
</body>
</html>

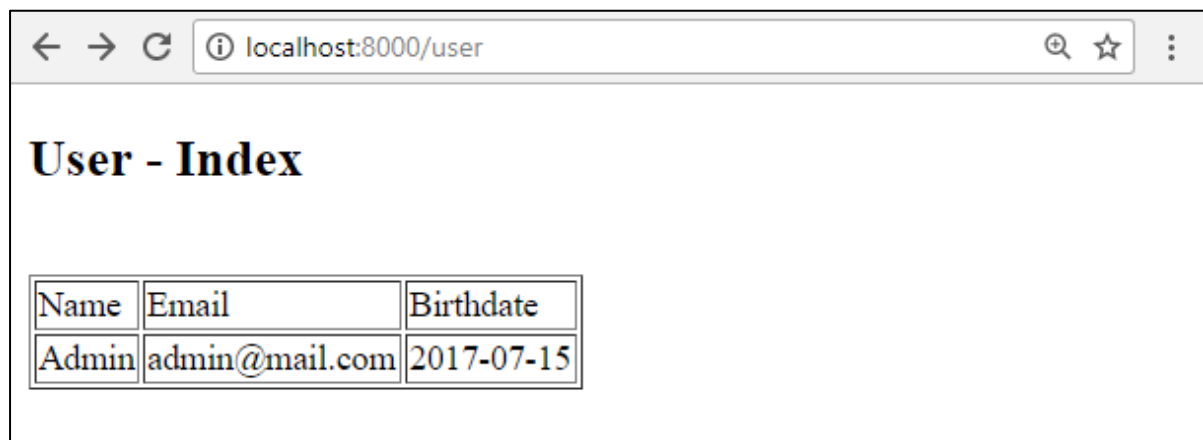
```

Setelah membuat *view* dan *controller*. Selanjutnya buatlah rute yang menghubungkan *view* (**index.blade.php**) dengan *controller* (**UserController fungsi index**).

```
Route::get('/user', 'UserController@index');
```

Rute diatas akan memanggil fungsi index pada UserController.

Jika sudah selesai, maka halaman index.blade.php sudah dapat diakses melalui browser dengan memasukkan URL “**localhost:8000/user**”.



Penjelasan:

Ketika URL “**localhost:8000/user**” diakses, Laravel akan membaca URL yang masuk dan mengarahkannya pada rute yang sesuai. Dalam contoh ini, akan diarahkan ke fungsi **index()**

pada **UserController**. Didalam fungsi **index()** akan diambil semua data *user* yang ada dan dikirimkan ke **index.blade.php**. Kemudian pada file **index.blade.php**, data yang dikirimkan tersebut akan ditampilkan dalam bentuk table. Dalam contoh ini sudah terdapat satu baris data hasil dari seeder yang sudah dibuat pada subbab 2.3.2.

2.4.3 Menambahkan Data Baru (create)

Untuk dapat menambahkan data user, akan ditambahkan fungsi **create()** dan **store()** pada **UserController**. Fungsi-fungsi ini akan berfungsi untuk **menampilkan form** untuk membuat data *user* yang baru dan untuk **menyimpan data** yang diinput ke dalam database.

```
public function create()
{
    return view('user.create');
}

public function store(Request $request)
{
    $user = new User();

    $user->name = $request->name;
    $user->email = $request->email;
    $user->password = bcrypt($request->password);
    $user->birthdate = Carbon::parse($request->birthdate);

    $user->save();

    return redirect('/user');
}
```

Fungsi **create()** akan berfungsi untuk mengembalikan halaman **create.blade.php** yang akan digunakan untuk meminta user untuk menginput data-data yang diperlukan untuk membuat data *user* baru.

Fungsi **store()** akan berfungsi untuk membuat data *user* baru dan menyimpan data-data yang sudah diinput pengguna, kemudian akan langsung diarahkan ke rute **“/user”** yang akan mengambil semua data *user* terbaru dan dikirimkan ke **index.blade.php**.

Untuk data tanggal, dalam contoh ini digunakan *Carbon* yang merupakan ekstensi PHP API untuk memanipulasi tanggal maupun waktu. Kelas *Carbon* merupakan turunan dari kelas *DateTime* dari PHP. Karena kelas *Carbon* tidak berada pada lokasi yang sama dengan **UserController**, maka lokasi kelas *Carbon* perlu didefinisikan dengan menambahkan kode **“use Carbon\Carbon”** diluar kelas **UserController** seperti gambar dibawah ini:

```

use App\User;
use Carbon\Carbon;

class UserController extends Controller

```

Setelah menambahkan fungsi **create()** dan **store()** untuk menambahkan data *user* baru, selanjutnya buatlah kode berikut ini pada halaman **create.blade.php**

```

<!DOCTYPE html>
<html>
<head>
    <title>User</title>
</head>
<body>
    <h2>User Create</h2>
    <br>
    <form action="{{ URL::to('/user') }}" method="POST">
        {!! csrf_field() !!}
        Name: <input type="text" name="name">
        <br>
        Email: <input type="text" name="email">
        <br>
        Password: <input type="password" name="password">
        <br>
        Birthdate: <input type="text" name="birthdate">
        <br>
        <input type="submit" value="Add User">
    </form>
</body>
</html>

```

`{!! csrf_field() !!}` atau `{{ csrf_field() }}` digunakan untuk membuat komponen input yang disembunyikan untuk menyimpan token yang akan digunakan untuk melindungi aplikasi dari serangan cross-site request forgery (CSRF). CSRF adalah serangan yang menampilkan suatu perintah yang tidak sah (tidak terotorisasi) atas nama pengguna yang sudah terotentikasi.

Tambahkan juga kode dibawah ini pada **index.blade.php** untuk menampilkan tombol “Insert”:

```

<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>User - Index</h2>
    <br>
    <table border="1">
        <thead>
            <td>Name</td>
            <td>Email</td>
            <td>Birthdate</td>
        </thead>
        @foreach($users as $user)
            <tr>
                <td>{{ $user->name }}</td>
                <td>{{ $user->email }}</td>
                <td>{{ $user->birthdate }}</td>
            </tr>
        @endforeach
    </table>
    <br>
    <a href="{{ URL::to('/user/create') }}">
        <input type="submit" value="Insert">
    </a>
</body>
</html>

```

Setelah membuat *view* dan *controller*. Selanjutnya buatlah rute yang menghubungkan *view* (`create.blade.php`) dengan *controller* (**UserController fungsi `create` dan `store`**).

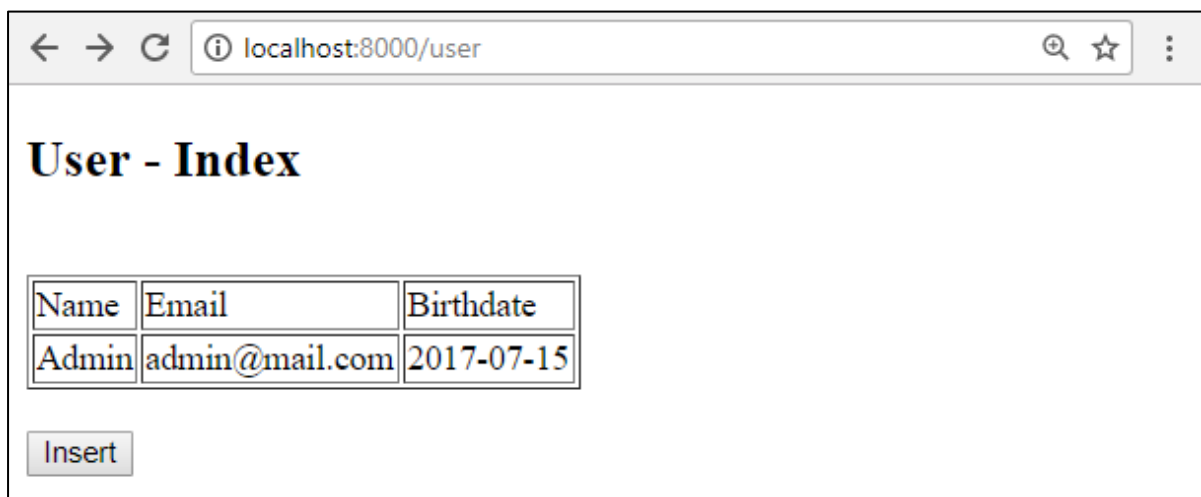
```

Route::get('/user/create', 'UserController@create');
Route::post('/user', 'UserController@store');

```

Rute diatas akan memanggil fungsi `create` dan `store` pada `UserController`.

Jika sudah selesai, maka halaman `index.blade.php` sudah dapat diakses kembali melalui browser dengan memasukkan URL “**localhost:8000/user**”.

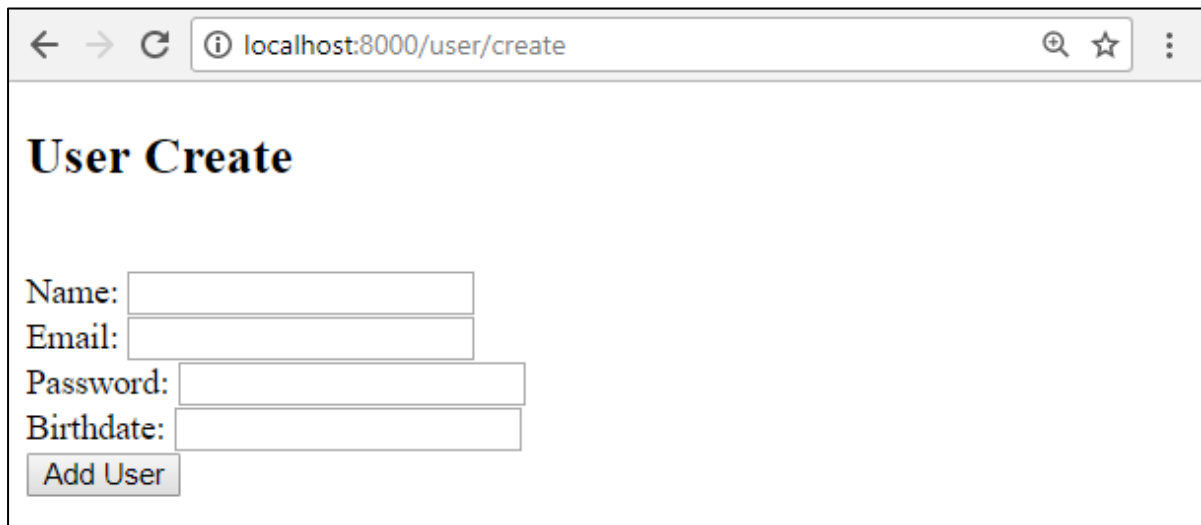


The screenshot shows a web browser window with the address bar displaying 'localhost:8000/user'. The page title is 'User - Index'. Below the title is a table with three columns: 'Name', 'Email', and 'Birthdate'. The table contains one row of data: 'Admin', 'admin@mail.com', and '2017-07-15'. Below the table is a button labeled 'Insert'.

Name	Email	Birthdate
Admin	admin@mail.com	2017-07-15

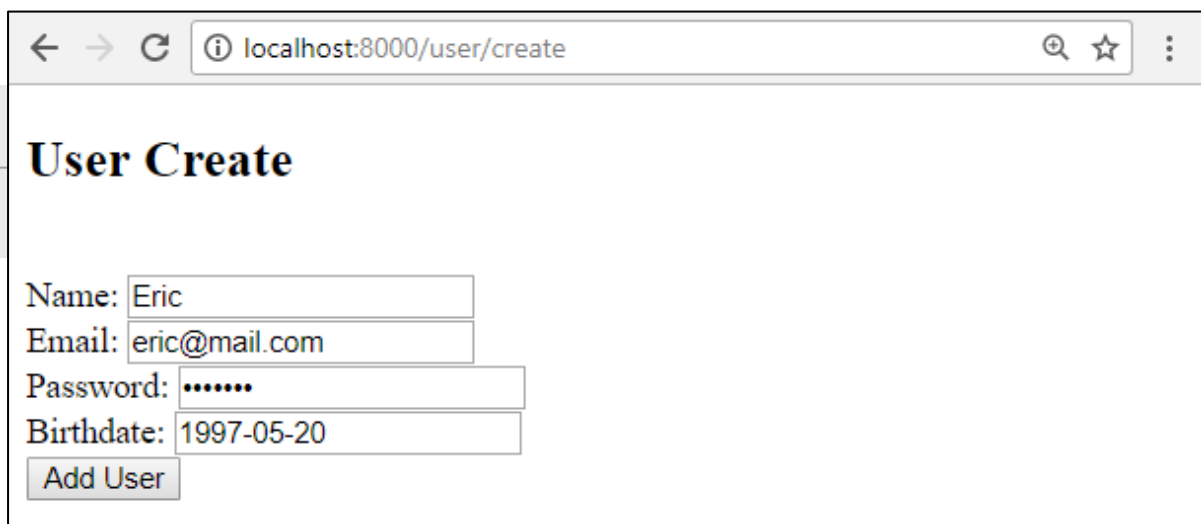
Insert

Ketika tombol “Insert” ditekan, halaman akan berubah menjadi halaman **create.blade.php** seperti gambar dibawah ini:



A screenshot of a web browser window showing a form titled "User Create". The browser's address bar displays "localhost:8000/user/create". The form contains four input fields: "Name:", "Email:", "Password:", and "Birthdate:". Below these fields is a button labeled "Add User".

Setelah semua data diisi dan tombol “Add User” ditekan, maka halaman akan berganti menjadi halaman **index.blade.php** dan menampilkan semua data *user* terbaru.



A screenshot of the same "User Create" form, but now the input fields are filled with data. The "Name" field contains "Eric", the "Email" field contains "eric@mail.com", the "Password" field contains "*****", and the "Birthdate" field contains "1997-05-20". The "Add User" button remains at the bottom.



User - Index

Name	Email	Birthdate
Admin	admin@mail.com	2017-07-15
Eric	eric@mail.com	1997-05-20

Penjelasan:

Ketika tombol “Insert” pada `index.blade.php` ditekan, maka Laravel akan mencari rute yang sesuai. Dalam contoh ini adalah rute dengan URL “**/user/create**” yang akan memanggil fungsi `create` pada `UserController`. Fungsi `create` akan langsung mengembalikan halaman **`create.blade.php`**.

Pada halaman **`create.blade.php`** setelah semua data diisi dan tombol “Add User” ditekan, maka *request* yang dikirimkan akan dicocokkan dengan rute yang sesuai. Dalam contoh ini *request* yang dikirimkan menggunakan metode POST ke URL “**/user**” sehingga akan langsung diarahkan ke fungsi `store` pada `UserController`. Di dalam fungsi `store`, akan dibuat sebuah obyek *user* baru, kemudian diisi dengan data yang sudah diinput dan disimpan ke dalam database. Setelah itu fungsi `store` akan mengarahkan pengguna ke URL “**/user**” dengan metode GET yang akan memanggil fungsi `index` pada `UserController`.

2.4.4 Mengubah Data (update)

Untuk dapat mengubah data user yang sudah ada, akan ditambahkan fungsi `edit()` dan `update()` pada `UserController`. Fungsi-fungsi ini akan berfungsi untuk **menampilkan form** untuk mengubah data *user* yang sudah ada dan untuk **menyimpan data** sesuai perubahan yang diinput oleh pengguna ke dalam database.

```

public function edit($id)
{
    $user = User::find($id);

    return view('user.update', [
        'user' => $user
    ]);
}

public function update(Request $request, $id)
{
    $user = User::find($id);

    $user->name = $request->name;
    $user->email = $request->email;

    $user->save();

    return redirect('/user');
}

```

Fungsi **edit()** akan berfungsi untuk mencari data *user* yang ingin diubah dan mengirimkannya ke halaman **update.blade.php**. Halaman ini akan digunakan untuk meminta user untuk menginput data-data yang akan diubah dari data user yang dikirimkan dari UserController.

Fungsi **update()** akan berfungsi untuk mencari data *user* yang akan diubah dan menyimpan data-data yang sudah diinput oleh pengguna, kemudian akan langsung diarahkan ke rute **"/user"** yang akan mengambil semua data *user* terbaru dan dikirimkan ke **index.blade.php**.

Setelah menambahkan fungsi **edit()** dan **update()** untuk mengubah suatu data *user*, selanjutnya buatlah kode berikut ini pada halaman **update.blade.php**

```

<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>User - Update</h2>
    <br>
    <form action="{{ URL::to('/user/' . $user->id) }}" method="POST">
        {{ csrf_field() }}
        {{ method_field('PUT') }}
        Name: <input type="text" name="name" value="{{ $user->name }}">
        <br>
        Email: <input type="text" name="email" value="{{ $user->email }}">
        <br>
        <input type="submit" value="Update User">
    </form>
</body>
</html>

```

{{ method_field('PUT') }} digunakan untuk mengubah metode *request* menjadi PUT. Hal ini diperlukan karena form pada HTML tidak mendukung metode-metode seperti PUT, PATCH, dan DELETE.

Tambahkan juga kode dibawah ini pada **index.blade.php** untuk menampilkan tombol “Update” di masing-masing baris data:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>User - Index</h2>
    <br>
    <table border="1">
        <thead>
            <td>Name</td>
            <td>Email</td>
            <td>Birthdate</td>
            <td>Actions</td>
        </thead>
        @foreach($users as $user)
            <tr>
                <td>{{ $user->name }}</td>
                <td>{{ $user->email }}</td>
                <td>{{ $user->birthdate }}</td>
                <td>
                    <a href="{{ URL::to('/user/' . $user->id . '/edit') }}">
                        <input type="submit" value="Update">
                    </a>
                </td>
            </tr>
        @endforeach
    </table>
    <br>
    <a href="{{ URL::to('/user/create') }}">
        <input type="submit" value="Insert">
    </a>
</body>
</html>
```

Setelah membuat *view* dan *controller*. Selanjutnya buatlah rute yang menghubungkan *view* (**update.blade.php**) dengan *controller* (**UserController** fungsi **edit** dan **update**).

```
Route::get('/user/{id}/edit', 'UserController@edit');
Route::put('/user/{id}', 'UserController@update');
```

Rute diatas akan memanggil fungsi **edit** dan **update** pada **UserController**.

Jika sudah selesai, maka halaman **index.blade.php** sudah dapat diakses kembali melalui browser dengan memasukkan URL “**localhost:8000/user**”.



User - Index

Name	Email	Birthdate	Actions
Admin	admin@mail.com	2017-07-15	Update
Eric	eric@mail.com	1997-05-20	Update

[Insert](#)

Ketika tombol “Update” ditekan, halaman akan berubah menjadi halaman **update.blade.php** dengan data-data yang sudah terisi seperti gambar dibawah ini:



User - Update

Name:

Email:

[Update User](#)

Setelah semua data yang ingin diubah sudah diubah dan tombol “Update User” ditekan, maka halaman akan berganti menjadi halaman **index.blade.php** dan menampilkan semua data *user* terbaru.

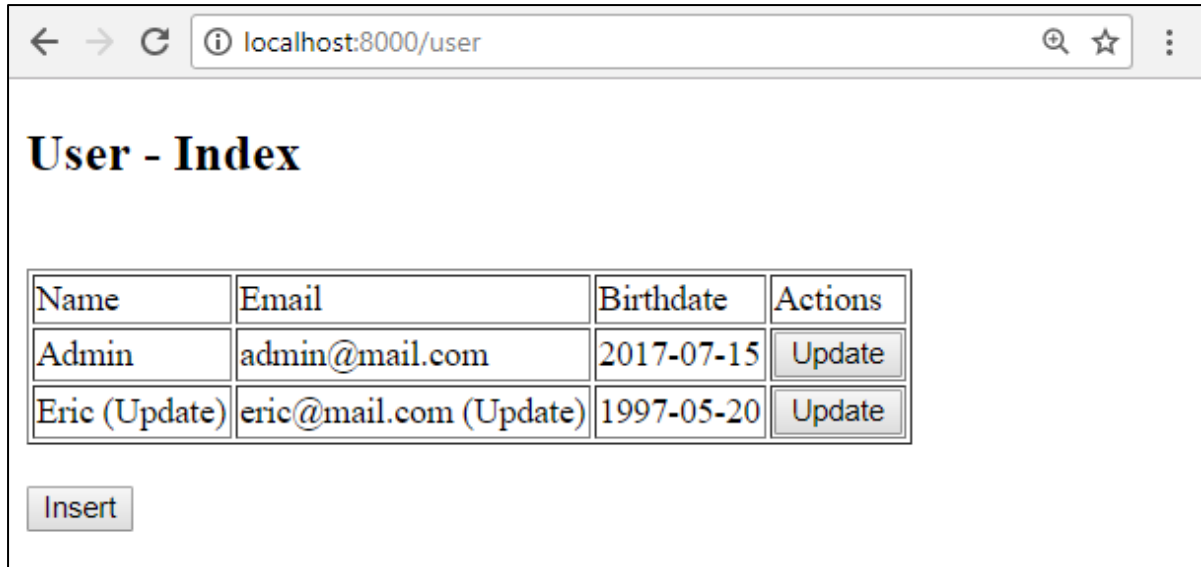


User - Update

Name:

Email:

[Update User](#)



Name	Email	Birthdate	Actions
Admin	admin@mail.com	2017-07-15	Update
Eric (Update)	eric@mail.com (Update)	1997-05-20	Update

Insert

Penjelasan:

Ketika tombol “Update” pada `index.blade.php` ditekan, maka Laravel akan mencari rute yang sesuai. Dalam contoh ini adalah rute dengan URL “`/user/{id}/edit`” yang akan memanggil fungsi edit pada `UserController`. Fungsi edit akan mencari data *user* sesuai id yang dikirimkan dari **`index.blade.php`** dan langsung mengirimkannya ke halaman **`update.blade.php`**.

Pada halaman **`update.blade.php`** data-data yang dikirimkan dari `UserController` akan ditampilkan untuk dapat diubah oleh pengguna. Setelah semua data selesai diubah dan tombol “Update User” ditekan, maka *request* yang dikirimkan akan dicocokkan dengan rute yang sesuai. Dalam contoh ini *request* yang dikirimkan menggunakan metode PUT ke URL “`/user/{id}`” sehingga akan langsung diarahkan ke fungsi update pada `UserController`. Di dalam fungsi update, akan dicari data *user* sesuai dengan id yang dikirimkan, kemudian data *user* tersebut diubah dengan data yang sudah diinput dan disimpan ke dalam database. Setelah itu fungsi update akan mengarahkan pengguna ke URL “`/user`” dengan metode GET yang akan memanggil fungsi index pada `UserController`.

2.4.5 Menghapus Data (delete)

Untuk dapat menghapus data user yang sudah ada, akan ditambahkan fungsi **`destroy()`** pada `UserController`. Fungsi ini akan berfungsi untuk **menghapus data** dari database sesuai id yang dikirimkan.

```
public function destroy($id)
{
    $user = User::find($id);
    $user->delete();
    return redirect('/user');
}
```

Fungsi **destroy()** akan berfungsi untuk mencari data *user* yang ingin dihapus dan menghapusnya dari database. Kemudian akan langsung diarahkan ke URL “/user” untuk menampilkan halaman **index.blade.php** dengan data *user* terbaru.

Setelah menambahkan fungsi **destroy()** untuk menghapus suatu data *user*, selanjutnya tambahkanlah kode berikut ini pada halaman **index.blade.php** untuk menampilkan tombol “Delete” di masing-masing baris data:



```

<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>User - Index</h2>
    <br>
    <table border="1">
        <thead>
            <td>Name</td>
            <td>Email</td>
            <td>Birthdate</td>
            <td colspan="2">Actions</td>
        </thead>
        @foreach($users as $user)
        <tr>
            <td>{{ $user->name }}</td>
            <td>{{ $user->email }}</td>
            <td>{{ $user->birthdate }}</td>
            <td>
                <a href="{{ URL::to('/user/' . $user->id . '/edit') }}">
                    <input type="submit" value="Update">
                </a>
            </td>
            <td>
                <form action="{{ URL::to('/user/' . $user->id) }}" method="POST">
                    {{ csrf_field() }}
                    {{ method_field('DELETE') }}
                    <input type="submit" value="Delete">
                </form>
            </td>
        </tr>
        @endforeach
    </table>
    <br>
    <a href="{{ URL::to('/user/create') }}">
        <input type="submit" value="Insert">
    </a>
</body>
</html>

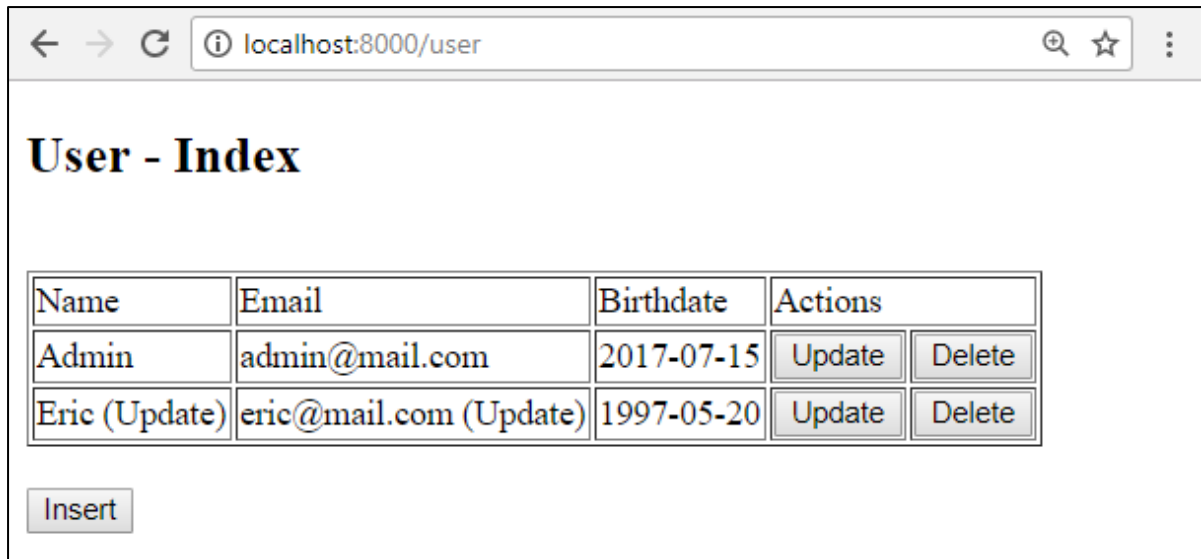
```

Setelah membuat *view* dan *controller*. Selanjutnya buatlah rute yang menghubungkan *view* (**index.blade.php**) dengan *controller* (**UserController fungsi destroy**).

```
Route::delete('/user/{id}', 'UserController@destroy');
```

Rute diatas akan memanggil fungsi destroy pada UserController.

Jika sudah selesai, maka halaman index.blade.php sudah dapat diakses kembali melalui browser dengan memasukkan URL “localhost:8000/user”.

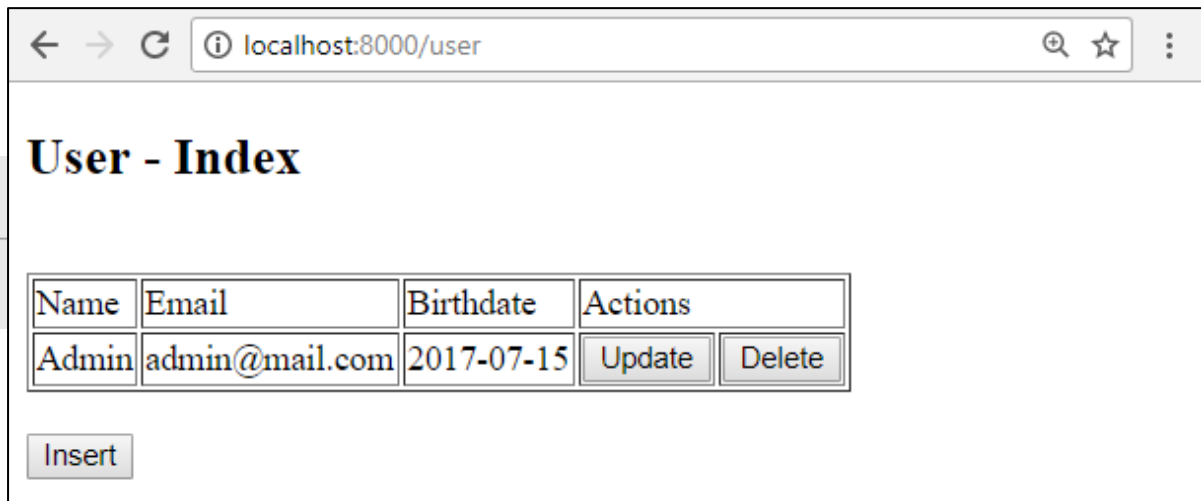


User - Index

Name	Email	Birthdate	Actions	
Admin	admin@mail.com	2017-07-15	Update	Delete
Eric (Update)	eric@mail.com (Update)	1997-05-20	Update	Delete

Insert

Ketika tombol “Delete” ditekan, halaman akan dimuat ulang dengan menghilangkan data yang sudah dihapus. Dalam contoh ini yang dihapus adalah data kedua.



User - Index

Name	Email	Birthdate	Actions	
Admin	admin@mail.com	2017-07-15	Update	Delete

Insert

Penjelasan:

Ketika tombol “Delete” pada `index.blade.php` ditekan, maka Laravel akan mencari rute yang sesuai. Dalam contoh ini adalah rute dengan method DELETE dan URL “/user/{id}” yang akan memanggil fungsi `destroy` pada `UserController`. Fungsi `destroy` akan mencari data *user* sesuai id yang dikirimkan dari **index.blade.php** dan menghapusnya dari database. Kemudian akan langsung mengirimkannya ke URL “/user” dengan metode GET untuk memanggil fungsi `index` yang akan mengembalikan halaman **index.blade.php** dengan data *user* terbaru.

2.5 Query

Query merupakan salah satu hal yang sangat penting dalam mengatur data dalam database untuk pengembangan aplikasi. Sebelum dapat menggunakan query pastikan database yang akan digunakan dalam aplikasi sudah diatur dengan benar pada file “`database.php`” yang ada didalam

folder “config” atau jika aplikasi masih dalam pengembangan lokal, database dapat diatur pada file “.env”.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=prk
DB_USERNAME=prk
DB_PASSWORD=prk
```

2.5.1 Using Eloquent Model

Laravel menyediakan query dengan cara yang lebih mudah menggunakan Eloquent Model. Eloquent model membuat query-query menjadi fungsi-fungsi yang bisa digunakan dengan mudah. Berikut ini adalah fungsi-fungsi yang dapat digunakan untuk mendapatkan data dalam eloquent query:

- **All**

Fungsi **all()** akan mengembalikan semua data yang ada pada table yang direpresentasikan oleh suatu model tertentu.

	SQL Query
	<code>SELECT * FROM users</code>
	Eloquent Query
	<code>User::all()</code>

- **Find**

Fungsi **find()** digunakan untuk menampilkan data pada suatu table sesuai dengan *primary key* yang ditentukan oleh pengembang, fungsi ini akan menerima parameter berupa *primary key* yang diinginkan dan mencari data-data tersebut untuk dikembalikan.

SQL Query
<code>SELECT * FROM users WHERE id = 1</code>
Eloquent Query
<code>User::find(1)</code>

SQL Query
<code>SELECT * FROM users WHERE id in (1, 2, 3)</code>
Eloquent Query

```
User::find([1, 2, 3])
```

- **Where**

Fungsi **where()** digunakan untuk menspesifikasikan syarat yang dibutuhkan dalam suatu query. Setelah menambahkan fungsi **where**, untuk mendapatkan hasil dari query, perlu ditambahkan fungsi **get()** diakhir query yang menandakan query telah selesai.

SQL Query

```
SELECT * FROM users WHERE name = 'Eric'
```

Eloquent Query

```
User::where('name', 'Eric')->get()
```

SQL Query

```
SELECT * FROM users WHERE stock > 10
```

Eloquent Query

```
Product::where('stock', '>', 10)->get()
```

- **Order by**

Fungsi **orderBy()** digunakan untuk mengurutkan suatu data hasil query terakhir sesuai dengan atribut table yang ditentukan dan dengan cara pengurutan (*ascending* atau *descending*) yang juga sudah ditentukan. Setelah menggunakan fungsi **orderBy()**, perlu ditambahkan juga fungsi **get()** yang menandakan query yang sedang dibuat telah selesai.

SQL Query

```
SELECT * FROM users where name like '%a%' order by name
```

Eloquent Query

```
User::where('name', 'like', '%a%')->orderBy('name')
```

SQL Query

```
SELECT * FROM users where name like '%a%' order by name desc
```

Eloquent Query

```
User::where('name', 'like', '%a%')->orderBy('name', 'desc')
```

- **Multiple Table Queries**

Untuk membuat query yang melibatkan lebih dari 1 table atau biasa disebut *join* maka Eloquent memiliki syarat tersendiri yaitu model-model yang dibuat harus sudah dilengkapi dengan fungsi-fungsi yang menyatakan hubungannya dengan model lain.

```
class User extends Model
{
    public function stories()
    {
        return $this->hasMany(Story::class);
    }

    public function address()
    {
        return $this->hasOne(Address::class);
    }
}
```

Jika fungsi-fungsi tersebut sudah dilengkapi maka untuk membuat query dengan menggabungkan table dapat dilakukan dengan mudah menggunakan *dynamic property* sesuai dengan fungsi-fungsi yang sudah dibuat pada masing-masing model.



SQL Query
<pre>SELECT stories.* FROM users join stories on users.id = stories.user_id WHERE users.name = 'Eric'</pre>
Eloquent Query
<pre>User::where('name', 'Eric')->stories</pre>
Atau
<pre>User::where('name', 'Eric')->stories()->get()</pre>

2.5.2 Using Query Builder

Selain menggunakan Eloquent model, Laravel juga menyediakan pilihan lain, yaitu menggunakan Query Builder. Cara penggunaan query builder tidak jauh berbeda dengan yang digunakan pada Eloquent model.

Pada Query Builder setiap query yang digunakan tetap akan digantikan oleh fungsi-fungsi yang telah disediakan Laravel. Perbedaannya dengan menggunakan Eloquent model adalah tentu saja tidak memerlukan model untuk membuat query dengan Query builder, selain itu untuk

membuat query dengan lebih dari satu table, dengan query builder, table-table tersebut harus digabungkan secara manual.

Berikut adalah contoh penggunaan query builder jika dibandingkan dengan query SQL biasa:

- SELECT

SQL Query
<code>SELECT * FROM users</code>
Eloquent Query
<code>DB::table('users')->select('users.*')->get()</code>

SQL Query
<code>SELECT name, birthdate FROM users</code>
Eloquent Query
<code>DB::table('users')->select('name', 'birthdate as Born_Date')->get()</code>

WHERE
SQL Query
<code>SELECT * FROM users WHERE name = 'Eric'</code>
Eloquent Query
<code>DB::table('users')->select('users.*')->where('name', '=', 'Eric')->get()</code>

- JOIN

SQL Query
<code>SELECT * FROM users join stories on users.id = stories.user_id WHERE users.name = 'Eric'</code>
Eloquent Query
<code>DB::table('users') ->join('stories', 'users.id', '=', 'stories.user_id') ->select('users.*', 'stories.*') ->where('users.name', '=', 'Eric') ->get()</code>

- GROUP BY

SQL Query

```
SELECT name
FROM users
GROUP BY birthdate
```

Eloquent Query

```
DB::table('users')->select('name')->groupBy('birthdate')->get()
```

- AGGREGATE

SQL Query

```
SELECT COUNT(*) FROM users
```

Eloquent Query

```
DB::table('users')->count()
```

SQL Query

```
SELECT MAX(birthdate) from users
```

Eloquent Query

```
DB::table('users')->max('birthdate')
```

- UNION

SQL Query

```
SELECT * FROM users WHERE gender = 'Male'
UNION
SELECT * FROM users WHERE gender = 'Female'
```

Eloquent Query

```
$query_first = DB::table('users')->where('gender', 'Male')->get()

$query_second = DB::table('users')
                ->where('gender', 'Female')
                ->unions($query_first)
                ->get()
```

BAB 3

Paging & Searching



UNIVERSITY

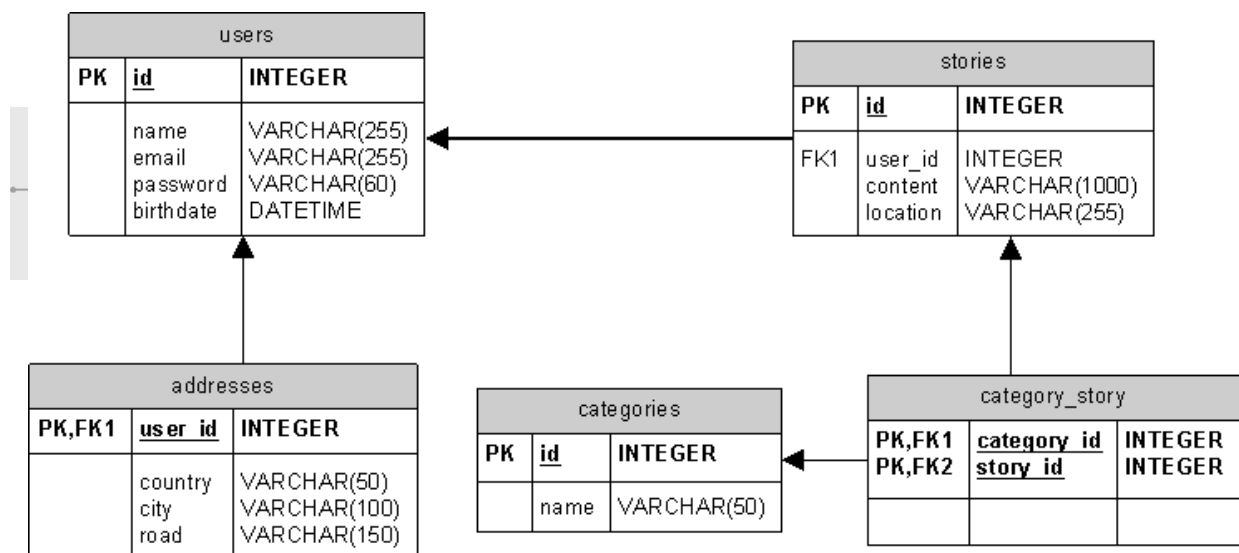
SOFTWARE LABORATORY CENTER

3.1 Paging & Searching

Dalam pembuatan suatu aplikasi, fitur seperti pencarian dan pembagian item untuk ditampilkan dalam suatu halaman menjadi dua hal yang wajib disertakan. Selain untuk memudahkan pengguna dalam menggunakan aplikasi, fitur ini juga dapat meningkatkan performa yang dapat dicapai oleh aplikasi.

3.1.1 Paging

Paging merupakan salah satu cara memunculkan data dengan membaginya menjadi beberapa halaman. Paging diperlukan karena saat memunculkan data yang sangat banyak dari database, tidak mungkin untuk memunculkan semua data tersebut sekaligus. Hal tersebut dapat menurunkan performa aplikasi karena harus memuat data yang sangat banyak sekaligus. Hal itu juga dapat merusak tampilan (*user interface*) dan pengalaman pengguna (*user experience*) saat menggunakan aplikasi.



Pada subbab ini, ERD database yang digunakan masih sama dan untuk mencontohkan penggunaan paging yang akan dipakai adalah table *stories*. Data-data pada table *stories* akan dibagi menjadi beberapa halaman dengan menampilkan 5 buah *story* untuk setiap halaman.

Pertama-tama pastikan komponen-komponen dari MVC untuk model *Story* sudah selesai disiapkan.

1. Model

```
D:\FirstLaravelProject>php artisan make:model Story
```

```
<?php

namespace App;

use Illuminate\Database\Eloquent\Model;

class Story extends Model
{
    //
}
```

2. Controller

```
D:\FirstLaravelProject>php artisan make:controller StoryController
```

```
<?php

namespace App\Http\Controllers;

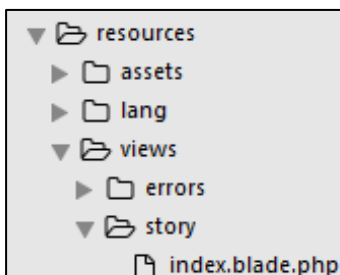
use Illuminate\Http\Request;

use App\Http\Requests;

class StoryController extends Controller
{
    //
}
```

3. View

Buatlah halaman yang akan mendukung dalam pembuatan paging untuk model *story*. Pertama-tama buatlah folder “story” didalam folder “resources/views”. Kemudian di dalam folder “story” yang baru saja dibuat, buatlah halaman **index.blade.php** yang akan menjadi halaman pendukung dalam pembuatan paging.



```
▼ resources
  ► assets
  ► lang
  ▼ views
    ► errors
    ▼ story
      index.blade.php
```

Setelah semua komponen MVC siap, selanjutnya tambahkanlah fungsi **index()** pada StoryController yang akan digunakan untuk mengambil semua data *story* dan membaginya menjadi beberapa halaman.

Untuk membagi data-data yang ada dalam database, fungsi **paginate()** dapat digunakan dengan sangat mudah, fungsi **paginate()** ini akan menerima parameter **N**, dimana **N** adalah jumlah item yang ingin ditampilkan dalam suatu halaman. Contoh penggunaannya adalah seperti gambar dibawah ini:

```
public function index()
{
    $stories = Story::paginate(5);

    return view('story.index', compact('stories'));
}
```

Fungsi **index()** diatas akan mengambil semua data *story* sesuai dengan jumlah item yang sudah ditentukan oleh pengembang untuk dibagi perhalamannya. Dalam contoh ini jumlah item yang akan ditampilkan adalah 5 buah per halaman. Kemudian data *story* yang sudah dibagi itu akan dikirimkan ke halaman **index.blade.php** yang ada di dalam folder “story”.

Setelah menambahkan fungsi **index()** pada StoryController, selanjutnya ubahlah halaman **index.blade.php** untuk menampilkan data yang dikirimkan dari *controller* dalam bentuk table seperti gambar dibawah ini:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Story Index</h2>
    <br>
    <table border="1">
        <tr>
            <th>Location</th>
            <th>Content</th>
        </tr>
        @foreach($stories as $story)
            <tr>
                <td>{{ $story->location }}</td>
                <td>{{ $story->content }}</td>
            </tr>
        @endforeach
    </table>
</body>
</html>
```

Setelah membuat *view* dan *controller*. Selanjutnya buatlah rute yang menghubungkan *view* (**index.blade.php**) dengan *controller* (**StoryController fungsi index**).

```
Route::get('/story', 'StoryController@index');
```

Rute diatas akan memanggil fungsi index pada StoryController.

Jika sudah selesai, maka halaman index.blade.php yang ada di dalam folder “story” sudah dapat diakses melalui browser dengan memasukkan URL “localhost:8000/story”.



Bisa dilihat bahwa data yang muncul pertama kali hanya 5 data dari table *stories*.

Untuk dapat mengakses 5 data selanjutnya, maka dapat ditambahkan parameter pada URL dalam bentuk *query string*, yaitu dengan menambahkan “?page=2” diakhir URL yang dimasukkan pada browser. Hasil yang akan tampil adalah seperti gambar dibawah ini:



Bisa dilihat bahwa data yang tampil setelah URL ditambahkan parameter *page* adalah lanjutan 5 data sebelumnya dari table *stories*. Pada gambar diatas data yang tampil hanya 4 buah karena dalam contoh ini data yang ada pada table *stories* hanya 9 data.

Dalam melakukan paging tentu dibutuhkan juga link yang dapat memudahkan pengguna dalam mengganti halaman untuk mencari data yang diinginkan. Untuk memunculkan link tersebut maka dapat menambahkan kode berikut ini pada **index.blade.php** yang ada di folder “story”:

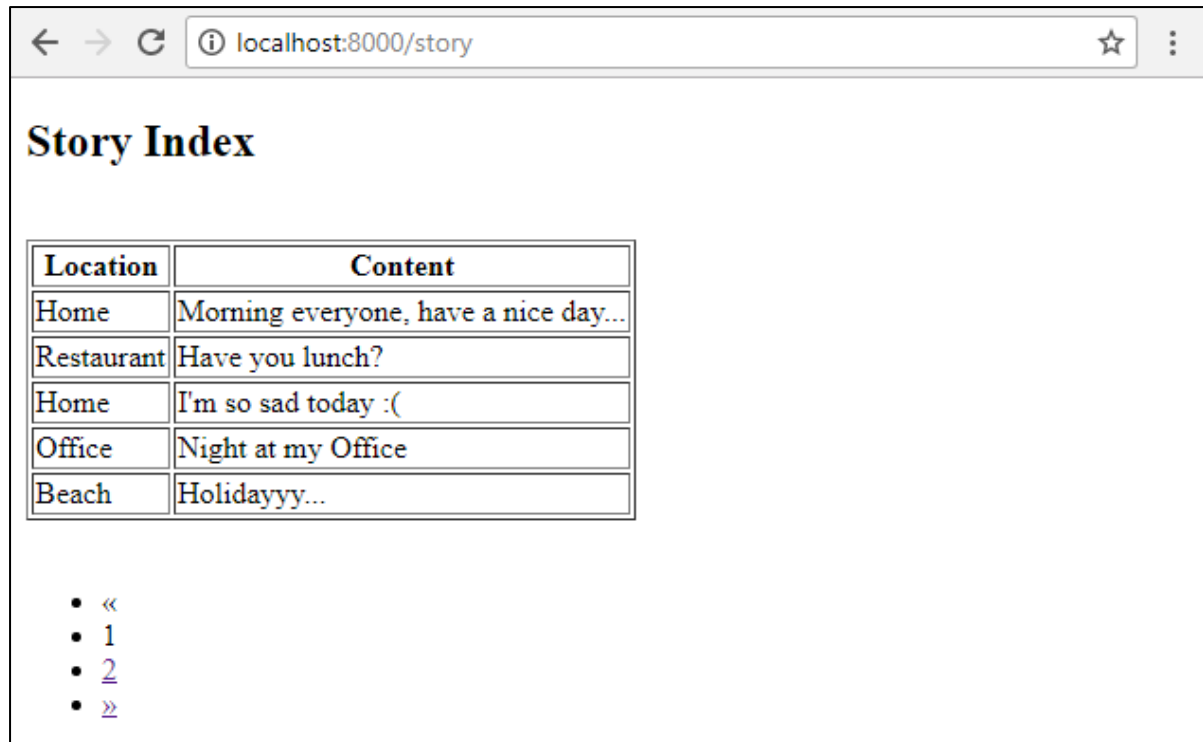
```
{{ $stories->render() }}
```

Atau

```
{{ $stories->links() }}
```

Setelah menambahkan kode diatas maka tampilan pada halaman index.blade.php akan menjadi seperti gambar dibawah ini:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Story Index</h2>
    <br>
    <table border="1">
        <tr>
            <th>Location</th>
            <th>Content</th>
        </tr>
        @foreach($stories as $story)
            <tr>
                <td>{{ $story->location }}</td>
                <td>{{ $story->content }}</td>
            </tr>
        @endforeach
    </table>
    <br>
    {{ $stories->links() }}
</body>
</html>
```



3.1.2 Searching

Searching merupakan fitur dari suatu aplikasi yang memiliki peran penting karena dapat mempercepat pengguna dalam mencari data yang diinginkan.

Pada subbab ini, akan dibahas mengenai cara membuat dan mengguakan fitur pencarian ini. Untuk mencontohkan cara pembuatan fitur ini, akan digunakan model dan halaman yang sudah dibuat saat membahas tentang paging.

Untuk mulai membuat fitur pencarian, ubahlah fungsi **index()** pada StoryController untuk dapat melakukan pencarian sekaligus paging seperti gambar dibawah ini:

```
public function index(Request $request)
{
    $search = $request->search;

    $stories = Story::where('content', 'like' , '%'.$search.'%')->paginate(5);

    return view('story.index', compact('stories', 'search'));
}
```

Setelah mengubah fungsi **index()** pada StoryController, tambahkanlah kode pada halaman **index.blade.php** yang ada di folder “story” untuk menampilkan form untuk memasukan kata kunci pencarian.

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Story Index</h2>
    <br>
    <form action="{{ URL::to('/story') }}" method="get">
        <input type="text" name="search" value="{{ $search }}">
        <input type="submit" value="Search">
    </form>
    <br>
    <table border="1">
        <tr>
            <th>Location</th>
            <th>Content</th>
        </tr>
        @foreach($stories as $story)
            <tr>
                <td>{{ $story->location }}</td>
                <td>{{ $story->content }}</td>
            </tr>
        @endforeach
    </table>
    <br>
    {{ $stories->links() }}
</body>
</html>
```

Setelah menambahkan kode untuk menampilkan form, tampilan pada browser akan menjadi seperti gambar dibawah ini:

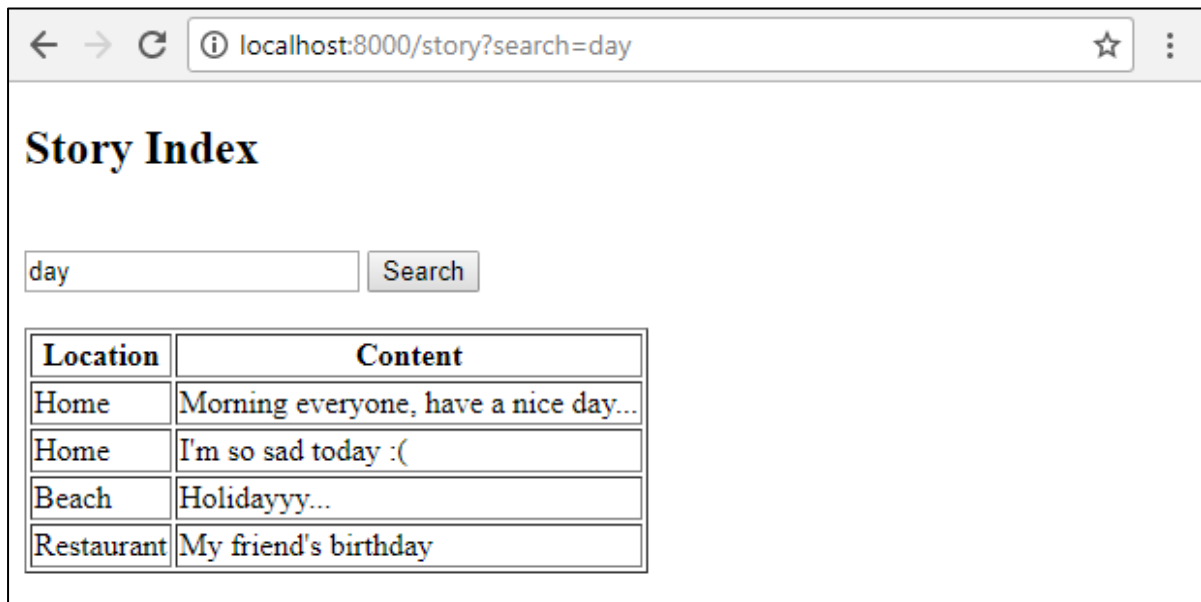
[←](#) [→](#) [↻](#) localhost:8000/story [☆](#) [⋮](#)

Story Index

Location	Content
Home	Morning everyone, have a nice day...
Restaurant	Have you lunch?
Home	I'm so sad today :(
Office	Night at my Office
Beach	Holidayyy...

- [«](#)
- [1](#)
- [2](#)
- [»](#)

Ketika teks pencarian diisi dan tombol “Search” ditekan maka, halaman **index.blade.php** akan menampilkan data-data sesuai dengan teks pencarian yang dimasukkan.



Pada gambar diatas link yang dibuat saat paging tidak tampil karena jumlah data hasil pencarian tidak lebih dari 5 data.

Fitur pencarian yang baru saja dibuat berjalan dengan baik jika data yang dihasilkan tidak melebihi 5 data. Jika hasil pencarian sudah melebihi 5 data, maka saat link untuk menampilkan halaman lain ditekan, pencarian tidak dilakukan lagi.

Hal ini terjadi karena saat link untuk menampilkan halaman lain ditekan. Yang dilakukan oleh link tersebut adalah mengambil data sesuai dengan urutan yang ada tanpa disaring lagi berdasarkan kata kunci pencarian yang telah dimasukan.

Untuk mencegah hal tersebut dan membuat link agar menampilkan halaman lain tetapi tetap menyaring data sesuai dengan kata kunci, maka kode yang digunakan untuk membuat link harus ditambahkan dengan parameter kata kunci pencarian yang diinginkan. Cara untuk menambahkan parameter tersebut adalah seperti gambar dibawah ini:

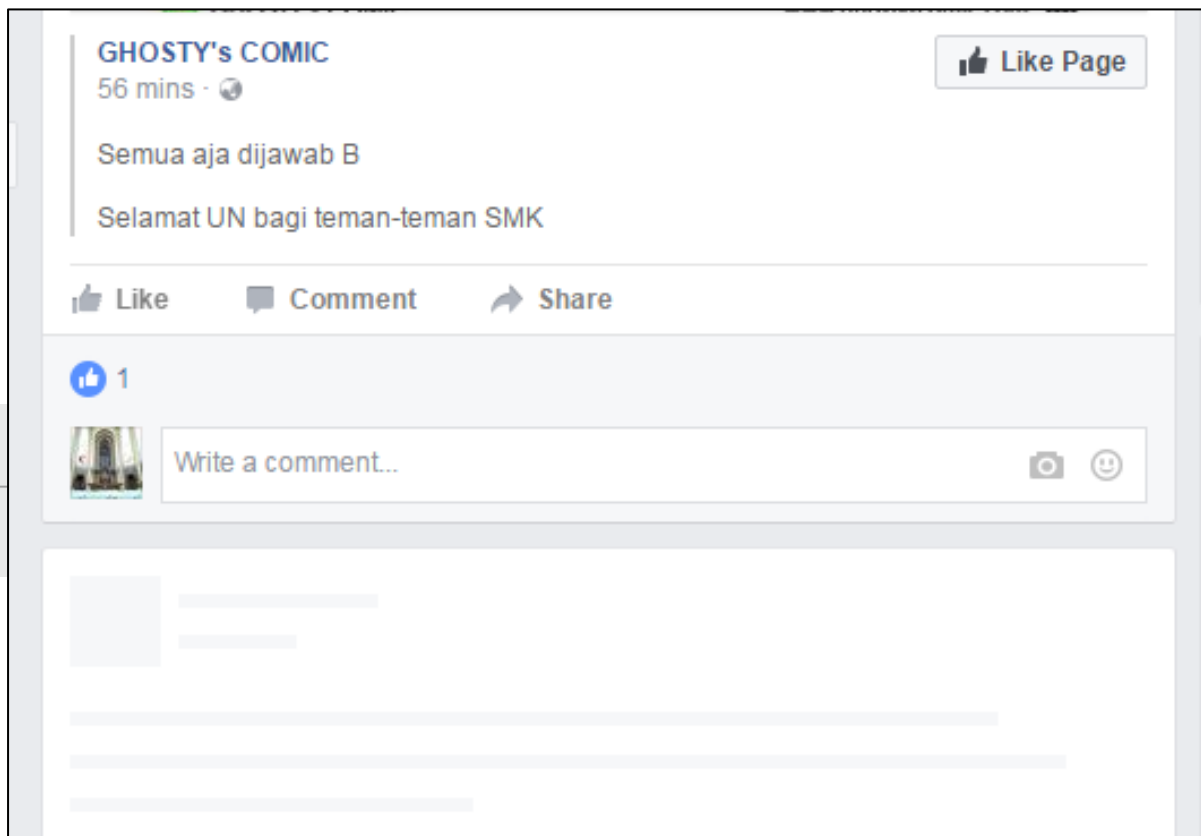
```
$stories->appends(compact('search'))->links()
```

Atau

```
$stories->appends(compact('search'))->render()
```

3.2 Infinite Scrolling

Dalam perkembangannya suatu website yang memiliki konten yang sangat banyak tidak hanya dapat menggunakan paging, tetapi juga dapat menggunakan teknik lain. Salah satu Teknik yang cukup sering digunakan adalah *infinite scrolling*. Infinite Scrolling merupakan cara dimana untuk menampilkan konten yang sangat banyak konten tersebut akan dimuat secara bertahap. Dimana setiap tahap pemuatan konten tersebut, akan ditampilkan beberapa data dan akan bertambah terus apabila *scroll bar* sudah mencapai batas maksimumnya saat itu. Contohnya adalah seperti Facebook, 9GAG, dll yang menggunakan *infinite scrolling* dalam menyajikan kontennya.



Pada subbab ini akan dibahas tentang cara membuat dan menggunakan *infinite scrolling* dalam menyajikan data. Untuk melakukan *infinite scrolling* akan digunakan model yang sama dengan yang digunakan pada paging yang dibahas pada subbab sebelumnya.

Logika dari *infinite scrolling* cukup sederhana dan mirip dengan paging sederhana yaitu apabila pengguna melakukan *scroll* hingga akhir halaman (paling bawah) maka halaman akan memuat data-data selanjutnya (dapat dianggap halaman selanjutnya pada paging sederhana) lalu menambahkannya ke halaman tersebut.

Untuk melakukan logika tersebut, akan dibutuhkan bantuan JavaScript untuk mengetahui waktu ketika pengguna sudah melakukan *scrolling* pada halaman sampai akhir. Oleh karena itu buatlah suatu folder “script” di dalam folder “public”, kemudian buatlah suatu file JavaScript dengan nama **script.js** di dalam folder “script” yang baru saja dibuat.



Kemudian buatlah kode berikut ini didalam file **script.js** yang baru saja dibuat.

```

currentPage = 1;

document.onscroll = function() {
    var body = document.body;

    if (window.innerHeight + body.scrollTop == body.scrollHeight) {
        currentPage++;
        ajax('get', '/story/next?page=' + currentPage, function(response) {
            var data = JSON.parse(response).data;

            for (var i = 0; i < data.length; i++) {
                var table = document.getElementById('stories');
                var row = createRow(data[i]);
                table.appendChild(row);
            }
        });
    }

    function createRow(data) {
        var row = document.createElement('tr');
        var column_1 = document.createElement('td');
        var column_2 = document.createElement('td');

        column_1.innerHTML = data.location;
        column_2.innerHTML = data.content;

        row.appendChild(column_1);
        row.appendChild(column_2);

        return row;
    }

    function ajax(method, url, callback) {
        var httpRequest = new XMLHttpRequest();

        httpRequest.onreadystatechange = function() {
            if (this.readyState == 4 && this.status == 200) {
                callback(this.responseText);
            }
        };

        httpRequest.open(method, url, true);
        httpRequest.send();
    }
}

```

Penjelasan:

Fungsi **ajax(method, url, callback)** adalah fungsi yang dibuat untuk mendapatkan respon secara asynchronous dari server. Ajax merupakan singkatan dari Asynchronous Javascript and XML. Ajax memungkinkan javascript untuk berkomunikasi langsung dengan server, sehingga untuk mendapatkan data dari server, keseluruhan halaman tidak harus dimuat ulang. Fungsi ini menerima 3 parameter, yaitu:

1. method

Metode pengiriman *request* yang akan dilakukan.

2. url

Rute yang akan digunakan untuk berkomunikasi dengan server (controller).

3. Callback

Fungsi yang akan dijalankan juga *request* berhasil dikirimkan dan respon berhasil diterima. Keadaan saat request berhasil dikirimkan dan respon sudah siap diterima ditandai dengan variable *readyState*. Berikut adalah status-status yang menandai *readyState*:

Status	Keterangan
0	Request yang akan dikirimkan belum diinisialisasi
1	Koneksi ke server telah terbuat
2	Request sudah diterima oleh server
3	Request yang dikirimkan sedang diproses
4	Request selesai diproses dan respon telah siap

Selain itu untuk memastikan bahwa respon yang dikirimkan bukanlah error dapat digunakan pengecekan terhadap **status == 200** yang menyatakan bahwa response yang diberikan berstatus OK dan tidak error.

Fungsi **createRow(data)** akan digunakan untuk membuat baris baru yang akan dimasukkan kedalam table pada halaman untuk menampilkan *story* yang akan dibuat. Fungsi ini akan membuat sebuah baris baru dengan 2 kolom yang akan diisi dengan data konten dan lokasi dari data *story* yang didapat dari parameter.

Untuk mendeteksi bahwa pengguna melakukan *scroll* pada halaman aplikasi, dapat digunakan “**document.onscroll**” dan diisi dengan fungsi yang akan dijalankan ketika halaman di-*scroll*. Ketika tinggi halaman browser yang tampil (**window.innerHeight**) ditambahkan dengan jauh suatu halaman di-*scroll* (**body.scrollTop**) sudah sampai pada akhir halaman yang bisa di-*scroll* (**body.scrollHeight**), maka akan dimuat data-data pada halaman selanjutnya dari paging yang telah dilakukan.

Buat fungsi yang **index()** dan **next()** pada StoryController yang akan berfungsi untuk menampilkan data-data pada halaman **index.blade.php** yang ada di folder “story” dan untuk melakukan paging dan mengembalikan data yang sudah dibagi dengan paging. Fungsi-fungsi tersebut akan menjadi seperti gambar dibawah ini:

```
public function index()
{
    $stories = Story::paginate(5);

    return view('story.index', compact('stories'));
}

public function next()
{
    $stories = Story::paginate(5);

    return $stories;
}
```

Setelah selesai dengan StoryController, ubahlah halaman **index.blade.php** menjadi seperti gambar dibawah ini:

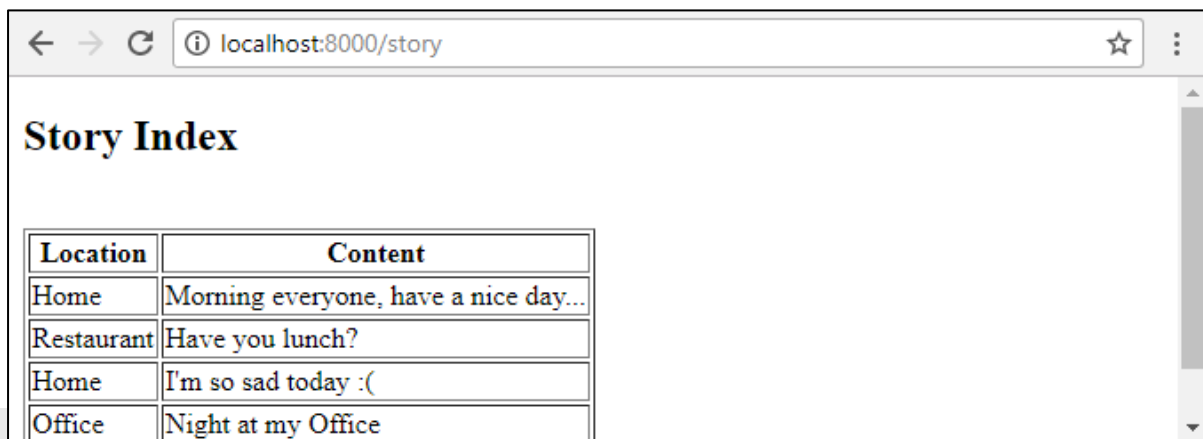
```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Story Index</h2>
    <br>
    <table id="stories" border="1">
        <tr>
            <th>Location</th>
            <th>Content</th>
        </tr>
        @foreach($stories as $story)
            <tr>
                <td>{{ $story->location }}</td>
                <td>{{ $story->content }}</td>
            </tr>
        @endforeach
    </table>
    <script type="text/javascript" src="{{ asset('script/script.js') }}"></script>
</body>
</html>
```

Selanjutnya buatlah rute untuk menghubungkan *controller* (StoryController fungsi **index** dan **next**) dan *view* (**index.blade.php** dalam folder **story**). Rute yang dibuat akan menjadi seperti gambar dibawah ini:

```
Route::get('/story', 'StoryController@index');
Route::get('/story/next', 'StoryController@next');
```

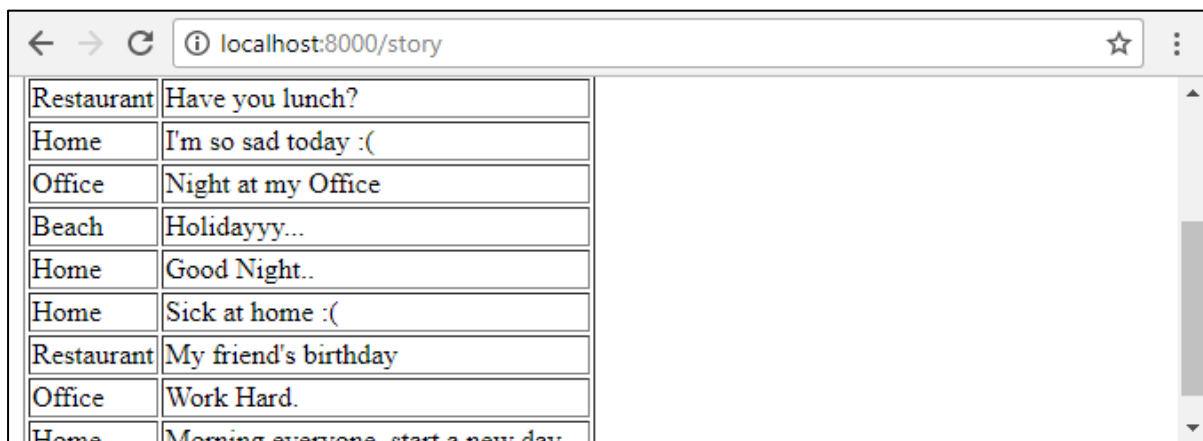
Rute pertama akan memanggil fungsi **index** pada StoryController untuk **menampilkan** data, rute kedua akan memanggil fungsi **next** pada StoryController untuk **mengambil** data yang sudah di paging.

Jika sudah selesai, sekarang teknik *infinite scrolling* untuk menampilkan data sudah dapat dicoba dengan mengakses URL "localhost:8000/story".



Location	Content
Home	Morning everyone, have a nice day...
Restaurant	Have you lunch?
Home	I'm so sad today :(
Office	Night at my Office

Jika halaman di *scroll* ke bawah, maka halaman akan memuat data selanjutnya yang masih ada dalam table *stories* sebanyak jumlah data yang ditentukan oleh pengembang saat menggunakan fungsi **paginate()**.



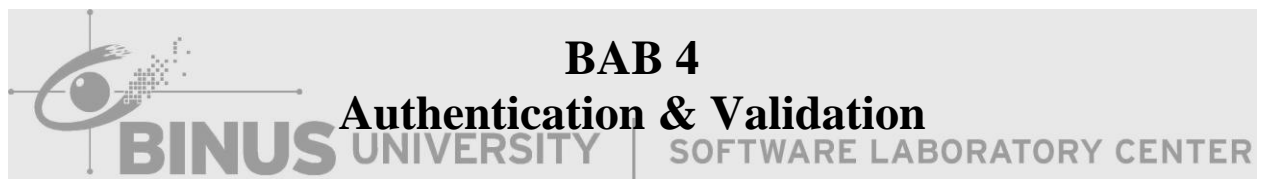
Restaurant	Have you lunch?
Home	I'm so sad today :(
Office	Night at my Office
Beach	Holidayyy...
Home	Good Night..
Home	Sick at home :(
Restaurant	My friend's birthday
Office	Work Hard.
Home	Morning everyone, start a new day...

Data tersebut akan terus dimuat sampai data yang ada di table *stories* sudah habis, ketika data pada database sudah habis maka halaman tidak akan dapat di *scroll* lagi.



A screenshot of a web browser window. The address bar shows 'localhost:8000/story'. The main content area displays a table with two columns: a location and a corresponding activity. The table has 8 rows. To the right of the table is a large, empty white rectangular area. The browser interface includes back, forward, and refresh buttons, as well as a star icon for bookmarks.

Restaurant	Alone dinner..
Home	Today is a great day
Office	Time to go home
Mall	Shopping time...
Home	Cooking
Home	Bed time...
Campus	Final exam finished
Office	I love my job



4.1 Otentikasi (Authentication)

Otentikasi adalah suatu proses untuk memverifikasi identitas dari seorang pengguna. Hal ini biasanya dilakukan saat user melakukan login dan register. Saat login, pengguna memasukkan username dan password atau data-data lain yang diperlukan system untuk memverifikasi bahwa pengguna tersebut telah memiliki identitas pada database. Jika username dan password atau data-data lain tersebut telah dapat diverifikasi identitasnya maka pengguna tersebut akan diberikan hak akses untuk masuk ke halaman utama dari suatu aplikasi dan bisa menggunakan fitur aplikasi lainnya.

4.1.1 Laravel Manual Authentication

Untuk dapat melakukan proses otentikasi, pertama-tama, pastikan database yang digunakan sudah dapat diakses dan sudah terbuat table *users*. Pada contoh ini, akan digunakan migration “create_table_users” yang sudah disediakan oleh Laravel. Untuk membuat table users jalankan perintah berikut ini:

```
D:\FirstLaravelProject>php artisan migrate
```

Setelah table users terbuat pada database, selanjutnya buatlah model User dan table users pada database. Setelah itu buatlah halaman yang akan mendukung proses otentikasi seperti gambar dibawah ini:

1. login.blade.php (resources/views/auth)

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Login</h2>
    <br>
    <form method="post" action="{{ URL::to('/login') }}">
        {{ csrf_field() }}

        Email:
        <input type="text" name="email">
        <br>
        Password:
        <input type="password" name="password">
        <br>
        <input type="checkbox" name="remember"> Remember me
        <br><br>
        <input type="submit" value="Login">
    </form>
</body>
</html>
```

2. register.blade.php (resources/views/auth)

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Register</h2>
    <br>
    <form method="post" action="{{ URL::to('/register') }}">
        {{ csrf_field() }}

        Name:
        <input type="text" name="name">
        <br>
        Email:
        <input type="text" name="email">
        <br>
        Password:
        <input type="password" name="password">
        <br>
        Confirm Password:
        <input type="password" name="password_confirmation">
        <br><br>
        <input type="submit" value="Register">
    </form>
</body>
</html>
```

3. home.blade.php (resources/views)

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Home</h2>
    <br>
    Welcome, {{ Auth::user()->name }}
</body>
</html>
```

Setelah membuat halaman login dan register, buatlah *controller* bernama AuthController, kemudian buatlah beberapa fungsi pada AuthController seperti gambar berikut ini:

```
public function showHome()
{
    return view('home');
}

public function showLoginForm()
{
    return view('auth.login');
}

public function login(Request $request)
{
    if (Auth::viaRemember())
    {
        return redirect('/home');
    }
    else if (Auth::attempt([
        'email' => $request['email'],
        'password' => $request['password']
    ], $request['remember']))
    {
        return redirect('/home');
    }

    return back();
}

public function showRegisterForm()
{
    return view('auth.register');
}

public function register(Request $request)
{
    $user = User::create([
        'name' => $request['name'],
        'email' => $request['email'],
        'password' => bcrypt($request['password']),
    ]);

    return redirect('/login');
}
```

Fungsi **attempt()** digunakan untuk melakukan otentikasi dengan berdasarkan data yang diberikan pada fungsi **attempt()** tersebut. Dalam contoh diatas fungsi **attempt()** akan mengotentikasi pengguna menggunakan email dan password.

Fungsi **viaRemember()** merupakan fungsi yang akan melakukan otentikasi dengan melakukan pengecekan terhadap fitur “remember me” yang disertakan pada fungsi **attempt()**.

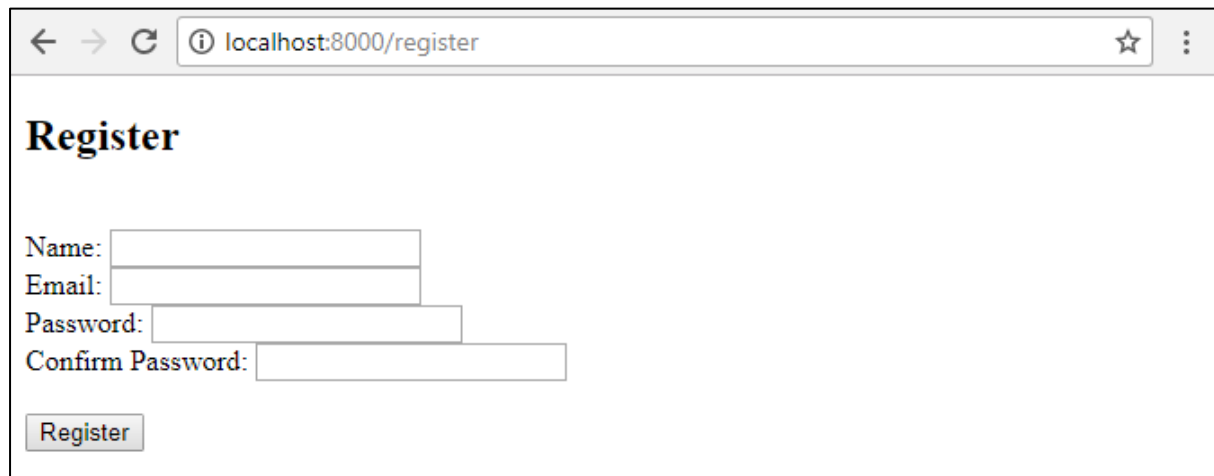
Setelah membuat *view* (**login.blade.php**, **register.blade.php**, **home.blade.php**) dan *controller* (**AuthController**), selanjutnya buatlah rute untuk menghubungkan *view* dan *controller* seperti gambar dibawah ini:

```
Route::get('/login', 'UserController@showLoginForm');
Route::post('/login', 'UserController@login');

Route::get('/register', 'UserController@showRegisterForm');
Route::post('/register', 'UserController@register');

Route::get('/home', 'UserController@showHome');
```

Setelah semua siap, proses otentikasi sudah dapat dicoba dengan mengakses URL “localhost:8000/register”.



The screenshot shows a web browser window with the address bar displaying "localhost:8000/register". The page content includes a heading "Register" and four input fields labeled "Name:", "Email:", "Password:", and "Confirm Password:". Below the input fields is a button labeled "Register".

← → ↻ ⓘ localhost:8000/register ☆ ⋮

Register

Name:

Email:

Password:

Confirm Password:

← → ↻ ⓘ localhost:8000/login ☆ ⋮

Login

Email:

Password:

☐ Remember me

← → ↻ ⓘ localhost:8000/login ☆ ⋮

Login

Email:

Password:

☒ Remember me

← → ↻ ⓘ localhost:8000/home 🔑 ☆ ⋮

Home

Welcome, Eric

4.1.2 Laravel Built-In Authentication

Pad dasarnya Laravel sudah menyediakan fitur yang dapat digunakan untuk membuat otentikasi secara otomatis, fitur ini baru dapat digunakan dengan menggunakan Laravel versi 5.2 keatas dan sangatlah mudah untuk diimplementasikan. Dokumentasi mengenai fitur otentikasi otomatis ini bisa dilihat pada link berikut: <https://laravel.com/docs/5.2/installation>.

Untuk dapat menggunakan otentikasi yang telah disediakan oleh Laravel, maka jalankan perintah seperti gambar dibawah ini:

```
D:\FirstLaravelProject>php artisan make:auth
```

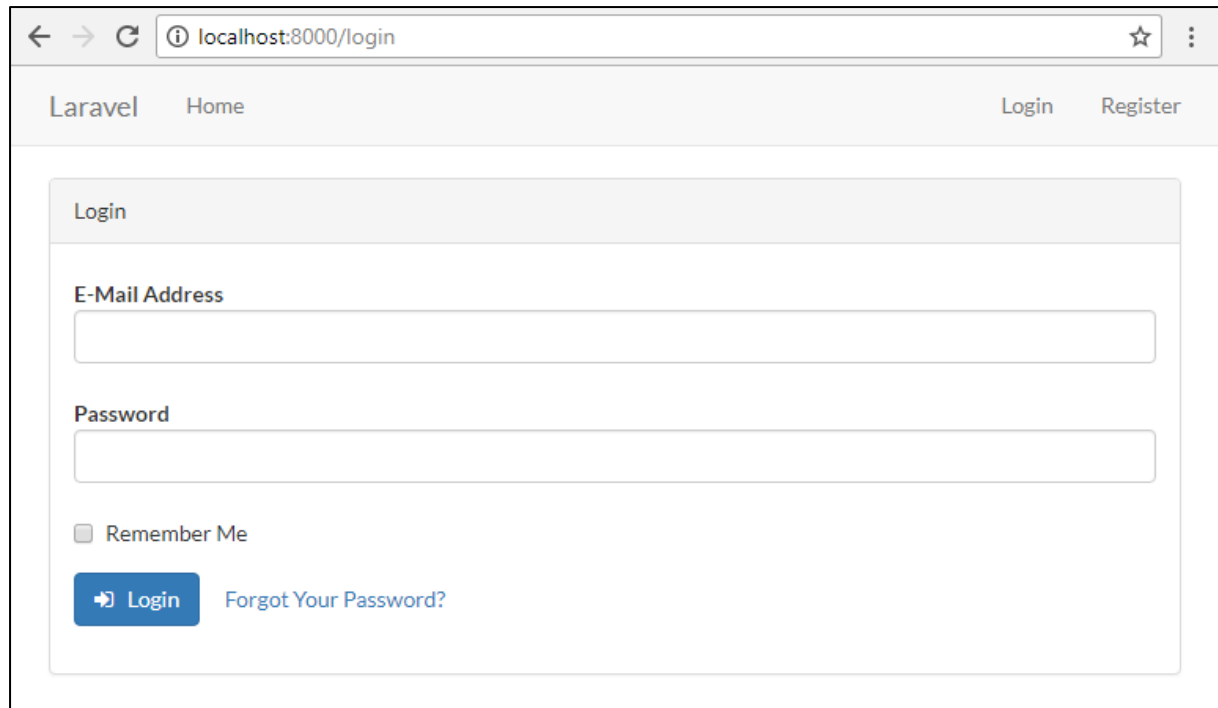
Setelah menjalankan perintah tersebut dapat dilihat bahwa Laravel akan secara otomatis membuat beberapa halaman atau *view* untuk mendukung proses otentikasi yaitu:

- resources/views/auth/login.blade.php
- resources/views/auth/register.blade.php
- resources/views/auth/passwords/email.blade.php
- resources/views/auth/passwords/reset.blade.php
- resources/views/auth/emails/password.blade.php
- resources/views/layout/app.blade.php
- resources/views/home.blade.php
- resources/views/welcome.blade.php

Selain itu Laravel juga akan membuat sebuah *controller* yaitu HomeController dan mengubah isi dari route file dan halaman welcome.blade.php.

Setelah selesai menjalankan perintah “**make:auth**” diatas, maka fitur otentikasi yang telah disediakan Laravel sudah dapat dicoba dengan mengakses “**localhost:8000**”.





A screenshot of a web browser displaying the Laravel login page. The address bar shows 'localhost:8000/login'. The page has a header with 'Laravel' and 'Home' on the left, and 'Login' and 'Register' on the right. The main content area is titled 'Login' and contains a form with the following elements: an 'E-Mail Address' input field, a 'Password' input field, a 'Remember Me' checkbox, a blue 'Login' button with a right-pointing arrow, and a link 'Forgot Your Password?'.

localhost:8000/login

Laravel Home Login Register

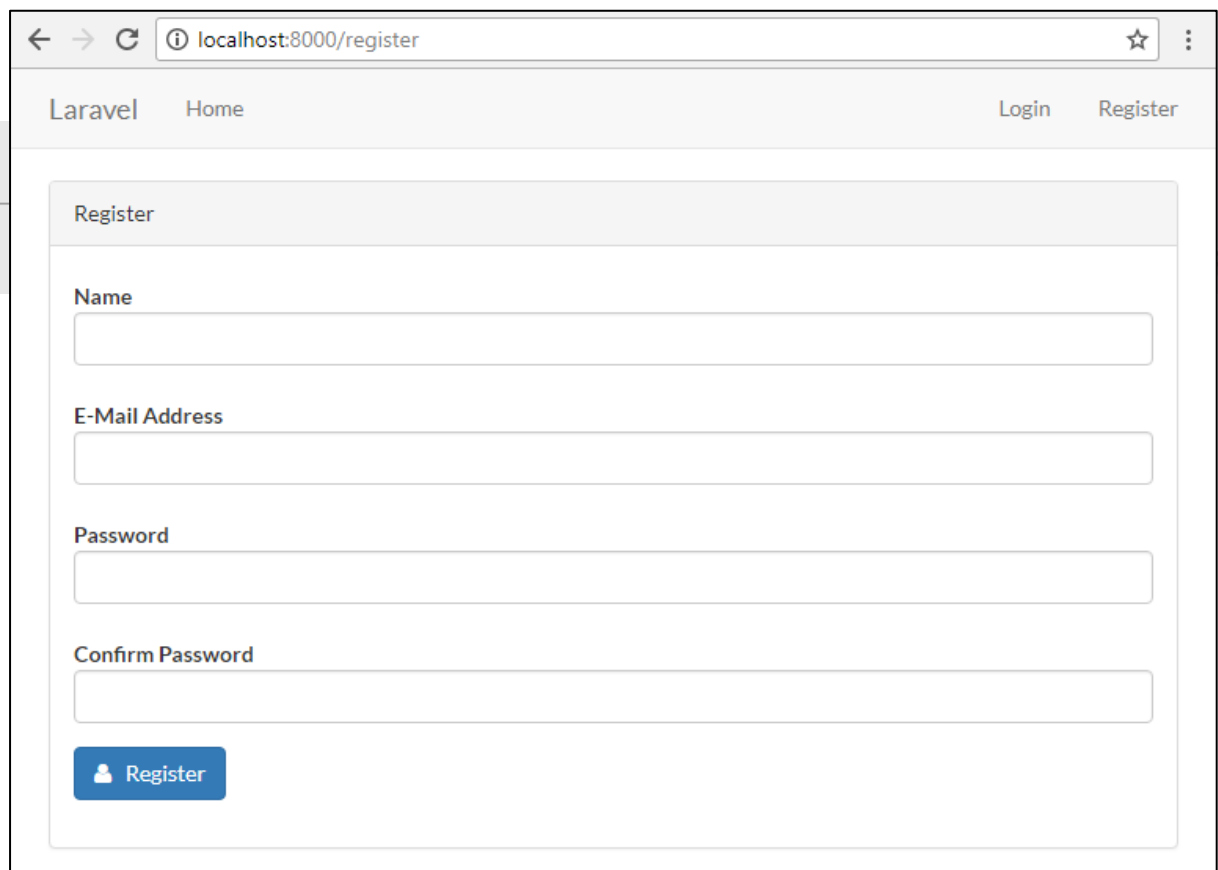
Login

E-Mail Address

Password

☐ Remember Me

Login Forgot Your Password?



A screenshot of a web browser displaying the Laravel register page. The address bar shows 'localhost:8000/register'. The page has a header with 'Laravel' and 'Home' on the left, and 'Login' and 'Register' on the right. The main content area is titled 'Register' and contains a form with the following elements: a 'Name' input field, an 'E-Mail Address' input field, a 'Password' input field, a 'Confirm Password' input field, and a blue 'Register' button with a person icon.

localhost:8000/register

Laravel Home Login Register

Register

Name

E-Mail Address

Password

Confirm Password

Register



4.2 Validasi

Laravel menyediakan beberapa cara yang berbeda untuk melakukan validasi pada data yang masuk ke aplikasi. Secara default setiap *controller* yang dibuat sudah dilengkapi dengan `ValidatesRequests` trait. Trait adalah sekumpulan fungsi yang dapat digunakan pada suatu class dengan tujuan mengurangi penulisan kode yang berulang. Trait `ValidatesRequests` ini menyediakan sebuah fungsi yang mudah digunakan untuk melakukan validasi data dengan berbagai macam aturan. Fungsi tersebut adalah **`validate()`**, contoh penggunaan dari fungsi ini adalah seperti gambar dibawah ini:

```
$this->validate($request, [
    'name' => 'required|max:255',
    'email' => 'required|email|unique:users|max:255',
    'password' => 'required|max:255',
    'birthdate' => 'required',
]);
```

Kode diatas akan melakukan validasi terhadap data yang dikirimkan dengan detail sebagai berikut:

Atribut	Validasi
name	<ul style="list-style-type: none"> Tidak boleh kosong Panjang maksimal adalah 255 karakter
email	<ul style="list-style-type: none"> Tidak boleh kosong Harus sesuai dengan format email Harus unik berdasarkan email yang ada pada table users Panjang maksimal adalah 255 karakter
password	<ul style="list-style-type: none"> Tidak boleh kosong Panjang maksimal adalah 255 karakter
birthdate	<ul style="list-style-type: none"> Tidak boleh kosong

Untuk membahas lebih lanjut mengenai fungsi **validate()** dan aturan-aturan validasi lainnya, dalam subbab ini, akan dijelaskan beberapa aturan-aturan yang dapat dipakai dalam fungsi **validate()**:

1. required

Atribut form yang diberikan aturan ini harus tersedia dan nilainya tidak kosong. Dalam aturan ini suatu atribut form disebut kosong jika memenuhi kondisi-kondisi dibawah ini:

- Memiliki nilai null
- Memiliki nilai string kosong
- Memiliki nilai array atau sejenisnya yang kosong
- Memiliki nilai berupa file yang tidak ada *path* lokasinya.

Contoh:

```
'name' => 'required'
```

2. max / min

Atribut form yang diberikan aturan max, harus memiliki panjang karakter lebih kecil atau sama dengan nilai yang diberikan. Sebaliknya atribut form yang diberikan aturan min, harus memiliki panjang karakter dengan minimum nilai sesuai yang diberikan.

Validasi ini hanya akan berfungsi jika atribut form yang ditentukan tersedia dan memiliki nilai.

Contoh:

```
'name' => 'max:255'
```

```
'name' => 'min:10'
```

3. boolean / date / integer / string

Atribut form yang diberikan salah satu dari aturan ini, harus sesuai dengan tipe data yang diberikan. Nilai-nilai yang diterima sesuai dengan tipe datanya dijelaskan lebih detail pada table berikut:

Aturan	Nilai yang diterima
boolean	true, false, 1, 0, "1", "0".

date	Tanggal valid berdasarkan fungsi <i>strtotime</i> dari php.
integer	Suatu angka integer.
string	Nilai yang berupa string.

Contoh:

```
'send_updates' => 'boolean'
```

```
'birthdate' => 'date'
```

```
'quantity' => 'integer'
```

```
'name' => 'string'
```

4. accepted

Atribut form yang diberikan aturan ini harus memiliki nilai yes, on, 1, atau true. Aturan ini biasanya dipakai untuk validasi “Terms of Service” telah disetujui oleh pengguna.

Contoh:

```
'terms' => 'accepted'
```

5. alpha / alpha_dash / alpha_num / numeric

Atribut form yang diberikan salah satu aturan ini, harus memiliki format sesuai dengan aturan yang diberikan. Penjelasan detil tentang ketiga aturan ini dapat dilihat pada tabel berikut:

Aturan	Keterangan
alpha	Harus berupa karakter alfabet.
alpha_dash	Harus berupa karakter alfabet, numerik, karakter '-' atau '_'.
alpha_num	Harus berupa karakter alfabet atau numerik.
numeric	Harus berupa karakter numerik.

Contoh:

```
'password' => 'alpha'
```

```
'password' => 'alpha_dash'
```

```
'password' => 'alpha_num'
```

```
'password' => 'numeric'
```

6. after / before (date)

Atribut form yang diberikan aturan **after**, harus berupa tanggal yang lebih besar dari tanggal yang diberikan. Sebaliknya, atribut form yang diberikan aturan **before**, harus berupa tanggal yang lebih kecil dari tanggal yang diberikan. Tanggal yang diberikan pada aturan ini juga harus mengikuti format tanggal yang valid berdasarkan fungsi `strtotime` pada php. Selain itu juga tanggal yang diberikan juga dapat digantikan dengan nama atribut yang ada pada form.

Contoh:

```
'start_date' => 'after:tomorrow'
```

```
'end_date' => 'before:2018-01-30'
```

7. in / not_in

Atribut form yang diberikan aturan **in**, harus memiliki nilai yang merupakan salah satu dari daftar nilai yang diberikan. Sebaliknya, atribut form yang diberikan aturan **not_in**, harus memiliki nilai yang **bukan** merupakan salah satu dari daftar nilai yang diberikan. Setiap nilai yang didaftarkan pada aturan ini akan dicocokkan secara *case sensitive* (huruf besar dan huruf kecil berpengaruh).

Contoh:

```
'gender' => 'in:male,female'
```

```
'level' => 'not_in:easy,medium'
```

8. confirmed

Atribut form yang diberikan aturan ini, harus memiliki nilai yang sama dengan atribut form yang memiliki nama yang sama dan diikuti dengan “_confirmation”. Aturan ini

juga melakukan validasi bahwa atribut form yang diikuti dengan “_confirmation” harus tersedia.

Contoh:

```
'password' => 'confirmed'
```

9. email

Atribut form yang diberikan aturan ini, harus memiliki nilai dengan format email.

Contoh:

```
'email' => 'email'
```

10. unique:table

Atribut form yang diberikan aturan ini, harus memiliki nilai yang berbeda daripada semua data yang ada pada table yang ditentukan dalam suatu database. Nama atribut form tersebut akan menjadi nama kolom yang dipakai untuk melakukan validasi nilai

unik.

Contoh:

```
'email' => 'unique:users'
```

Untuk melakukan validasi dengan menggunakan lebih dari satu aturan, aturan-aturan tersebut dapat dipisahkan dengan tanda ‘|’ seperti gambar dibawah ini:

```
'password' => 'required|alpha_num|max:255'
```

Dalam pembuatan suatu aplikasi website, tidak jarang ditemukan inputan yang membutuhkan array, untuk melakukan validasi terhadap inputan array dapat dilakukan dengan menggunakan tanda ‘*’ seperti gambar dibawah ini:

```
'email.*' => 'email|unique:users'
```

Selain inputan array, tidak jarang juga dibutuhkan inputan yang bersifat *nested* (memiliki penampung yang mengelompokkan beberapa inputan). Untuk melakukan validasi terhadap inputan yang bersifat *nested* dapat dilakukan dengan menggunakan tanda ‘.’ seperti gambar dibawah ini:

```
'user.email' => 'required|email'
```

Fungsi **validate()** akan mengembalikan respon ke halaman sebelumnya jika tidak memenuhi semua aturan yang sudah ditentukan.

Selain menggunakan fungsi **validate()**, pengembang juga dapat membuat sendiri validasi yang diinginkan dengan tujuan untuk mengubah respon yang akan dikembalikan saat inputan tidak memenuhi validasi yang telah ditentukan. Untuk membuat validasi secara manual, dapat digunakan kelas **Validator**. Cara menggunakan validasi dengan kelas **Validator** tidak jauh berbeda dengan fungsi **validate()** bahkan cenderung sama. Cara membuat validasi dengan kelas **Validator** adalah seperti gambar dibawah ini:

```
$validator = Validator::make($request->all(), [
    'name' => 'required|max:255',
    'email' => 'required|email|unique:users|max:255',
    'password' => 'required|max:255',
    'birthdate' => 'required',
]);
```

Kode diatas hanya akan membuat validator dengan aturan-aturannya. Untuk melakukan validasi menggunakan validator tersebut, dapat digunakan fungsi **fails()** seperti gambar dibawah ini:

```
if ($validator->fails())
{
    // validation failed
}
```

Dengan membuat validator secara manual, pengembang dapat dengan bebas menentukan apa yang akan dilakukan jika inputan yang dimasukkan tidak memenuhi aturan validasi yang telah ditentukan.

4.3 Pesan Error

Setelah melakukan validasi tentu akan sangat baik jika kesalahan dari inputan ditampilkan pada halaman agar tidak membuat pengguna bingung tentang apa yang harus dilakukan. Laravel

menyediakan cara yang mudah untuk menampilkan pesan error berdasarkan atribut form dan aturan-aturan yang ditentukan pada fungsi **validate()** atau kelas **Validator**.

Pada subbab ini akan dibahas lebih detil mengenai pesan error pada Laravel. Seperti yang sudah dituliskan diatas bahwa Laravel telah menyediakan pesan error berdasarkan atribut form dan aturan-aturan yang didukung oleh Laravel. Jika suatu validasi telah gagal semua aturan yang gagal akan dicari pesan error yang sesuai dan akan disimpan pada suatu **session**. Session adalah suatu cara untuk menampung informasi dalam variabel untuk dapat digunakan pada halaman yang berbeda.

Session yang digunakan Laravel untuk menyimpan pesan error tersebut ditampung pada suatu variable yang bernama **\$errors**. Setelah semua pesan error yang didapat dari validasi ditampung ke dalam variabel **\$errors**, maka pesan error tersebut akan dapat diakses pada halaman web menggunakan variabel **\$errors**. Cara menampilkan pesan error menggunakan variabel **\$errors** adalah seperti gambar dibawah ini:

```
<ul>
    @foreach($errors->all() as $error)
        <li>{{ $error }}</li>
    @endforeach
</ul>
```

Kode diatas akan menampilkan pesan-pesan error berdasarkan pada setiap aturan validasi yang gagal.

Pesan-pesan error yang akan ditampilkan tidak hanya bisa ditampilkan secara menyeluruh, tetapi juga dapat ditampilkan satu per satu berdasarkan nama kolom yang divalidasi. Untuk menampilkan pesan error berdasarkan nama kolom yang divalidasi, dapat dilakukan dengan cara seperti gambar dibawah ini:

```
<ul>
    @foreach($errors->get('name') as $error)
        <li>{{ $error }}</li>
    @endforeach
</ul>
```

Fungsi **get** pada variable **\$errors** diatas digunakan untuk mendapatkan pesan error hanya untuk kolom dengan nama yang ditentukan dalam fungsi tersebut.

Pada dua cara diatas pesan error yang ditampilkan adalah semua pesan error yang ada sesuai dengan fungsi yang dipakai. Dalam suatu aplikasi website tentu akan lebih baik bila hanya menampilkan satu buah pesan error berdasarkan urutan aturan yang diberikan. Untuk dapat melakukan hal tersebut dapat digunakan fungsi **first()** untuk mendapatkan pesan error yang paling pertama.

```
$errors->first('name')
```

Fungsi **first()** akan menerima satu buah parameter, yaitu nama kolom yang ingin ditampilkan pesan errornya.

Untuk melakukan pengecekan bahwa untuk kolom tertentu memiliki error dapat digunakan fungsi **has()** seperti gambar dibawah ini.

```
$errors->has('name')
```

Seperti yang sudah dijelaskan sebelumnya bahwa Laravel telah menyediakan pesan error untuk setiap aturan yang didukungnya. Tetapi Laravel tidak membatasi pengembang untuk harus menggunakan pesan error yang sudah disediakan untuk mengubah pesan error sesuai dengan keinginan dapat dilakukan dengan menambahkan variabel yang berisikan array asosiatif dengan aturan Laravel sebagai kata kuncinya dan pesan error yang diinginkan sebagai nilainya. Contohnya adalah seperti gambar dibawah ini:

```
$messages = [
    'required' => 'This :attribute must be filled',
    'min' => 'This :attribute must be more than or equals :min',
];

$this->validate($request, [
    'name' => 'required|min:5'
], $messages);
```

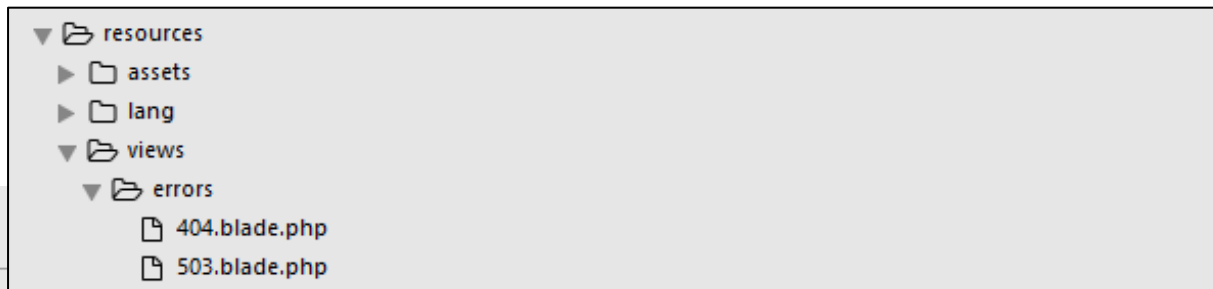
4.4 Halaman Error

Dalam pembuatan suatu aplikasi web, tentu tidak baik jika pengguna sampai mengalami error saat menggunakan aplikasi. Apalagi sampai menunjukkan halaman error yang dapat membuat pengguna bingung atau bahkan panik saat terjadi error. Sebagai pengembang tentu ingin jika saat aplikasinya mengalami error setidaknya tidak menampilkan halaman yang dapat membuat

panik pengguna. Laravel menyediakan cara yang sangat mudah untuk mengubah tampilan halaman error yang akan ditampilkan saat terjadi error pada aplikasi.

Mengubah tampilan halaman error pada Laravel dapat dilakukan dengan membuat suatu file berekstensi “.blade.php” didalam folder “resources/views/errors”. Laravel akan menentukan halaman error yang akan tampil dengan mencocokkan kode status error aplikasi dengan nama file yang ada di dalam folder “resources/views/errors”.

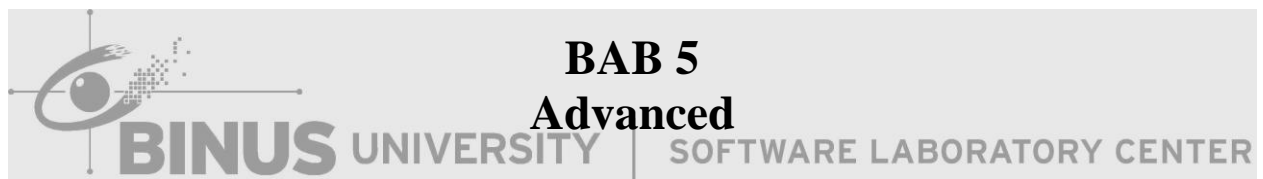
Sebagai contoh, misalnya error yang ingin diubah tampilan halaman errornya adalah error “page not found” yang memiliki kode status error 404, maka pada folder “resources/views/errors” dapat dibuat file 404.blade.php sebagai halaman yang akan ditampilkan saat error “page not found” terjadi.



Berikut adalah beberapa kode status error yang sering terjadi:

Kode Status	Keterangan Status
200	OK
300	Multiple choices
301	Moved permanently
302	Found
304	Not modified
307	Temporary redirect
400	Bad request
401	Unauthorized
403	Forbidden
404	Not found

410	Gone
500	Internal service error
501	Not implemented
503	Service unavailable
550	Permission denied



5.1 File Upload

File upload di laravel tidak berbeda jauh dengan file upload pada PHP. Untuk mencobanya, kita harus menambahkan atribut “enctype” dan diisi dengan nilai “multipart/form-data” pada form yang digunakan untuk melakukan upload file.

Sebagai contoh, buatlah folder “user” didalam folder “resources/views”, kemudian didalam folder “user” yang baru saja dibuat, buatlah halaman **profile.blade.php** untuk menampilkan form yang akan digunakan untuk melakukan upload file gambar:

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Upload Image</h2>
    <form enctype="multipart/form-data"
        method="post"
        action="{{ URL::to('/user/profile') }}">
        <input type="file" name="profile_picture">
        <br><br>
        <input type="submit" value="Upload">
    </form>
</body>
</html>
```

Setelah membuat halaman untuk upload file gambar, buatlah *controller* dengan nama UserController, kemudian buatlah fungsi **profile_create()** dan **profile()** pada UserController yang akan digunakan untuk menampilkan halaman **profile.blade.php** dan menyimpan file gambar yang sudah diupload.

```
public function profile_create()
{
    return view('user.profile');
}

public function profile(Request $request)
{
    $image = $request['profile_picture'];
    $file_name = $image->getClientOriginalName();
    $image->move('uploads', $file_name);

    return redirect('/user/profile/create');
}
```

Fungsi **profile_create()** akan digunakan untuk menampilkan form untuk melakukan upload file gambar. Sedangkan fungsi **profile()** akan digunakan untuk mendapatkan file yang diinput

pengguna kemudian disimpan pada folder “uploads” yang akan terbuat di dalam folder “public” pada proyek Laravel dengan nama sesuai dengan nama file aslinya.

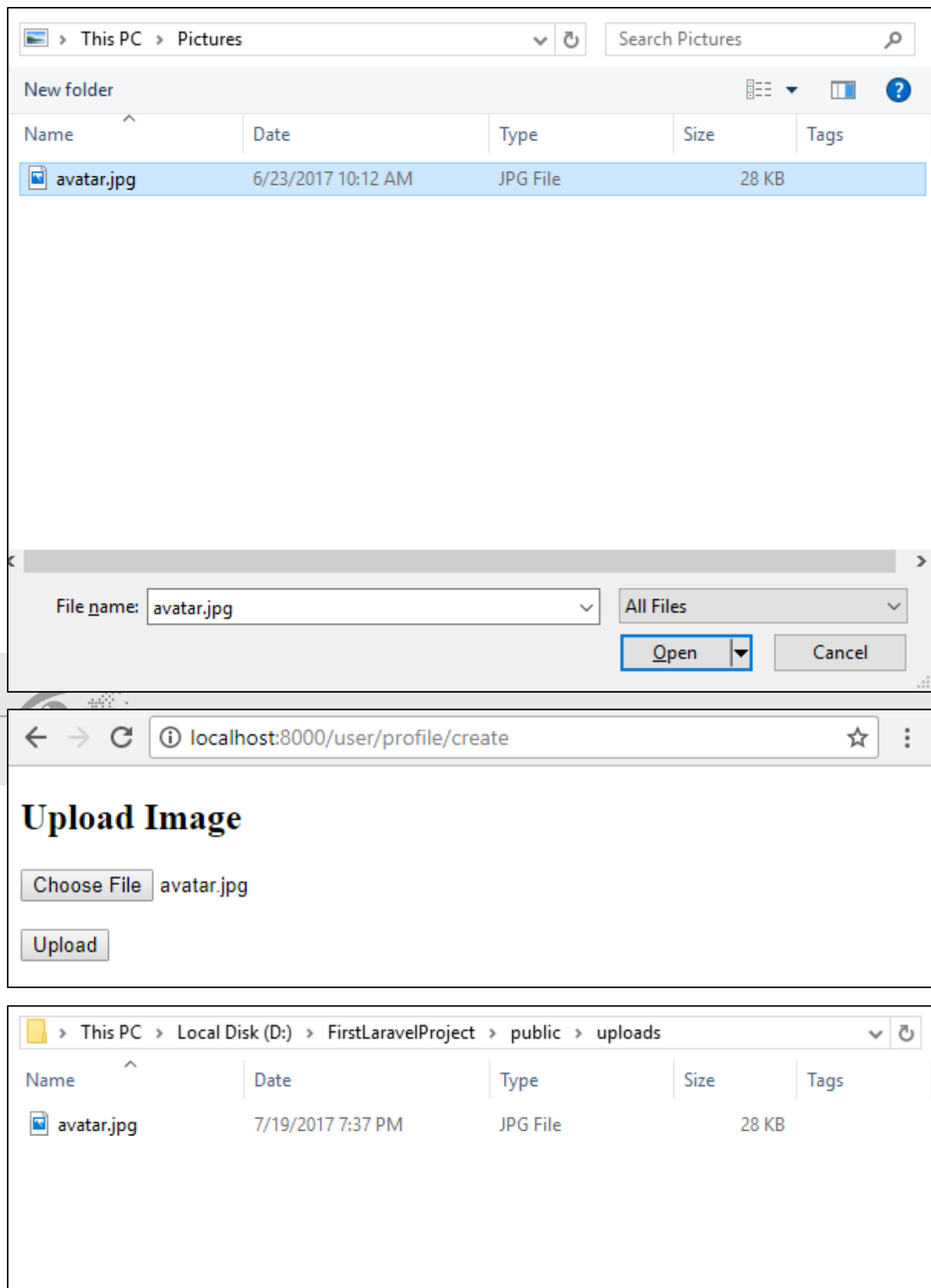
Setelah membuat halaman dan fungsi-fungsi untuk melakukan upload file. Selanjutnya buatlah rute untuk menghubungkan *view* (**profile.blade.php**) dengan *controller* (**UserController** fungsi **profile_create** dan **profile**).

```
Route::get('/user/profile/create', 'UserController@profile_create');  
Route::post('/user/profile', 'UserController@profile');
```

Setelah selesai, maka upload file gambar sudah dapat dilakukan dengan mengakses URL “localhost:8000/user/profile/create” seperti pada gambar berikut ini:



Jika tombol “Choose File” ditekan dan pengguna memilih file yang ingin diupload, kemudian tombol Upload ditekan maka file yang sudah ingin diupload tersebut akan di duplikasi dan dipindahkan ke folder “uploads” yang ada di folder “public” pada proyek Laravel.



Contoh diatas adalah contoh sederhana untuk melakukan upload file. Masih terdapat beberapa fungsi yang dapat dipakai saat bekerja dengan upload file.

1. extension

Fungsi **extension()** digunakan untuk mendapatkan ekstensi dari file yang akan diupload.

2. path

Fungsi **path()** digunakan untuk mendapatkan lokasi sementara dari file yang akan diupload.

3. hasFile / isValid

Fungsi **hasFile()** dan **isValid()** digunakan untuk memastikan bahwa nilai yang didapat dari inputan file mengandung file yang sesungguhnya.

4. getClientSize

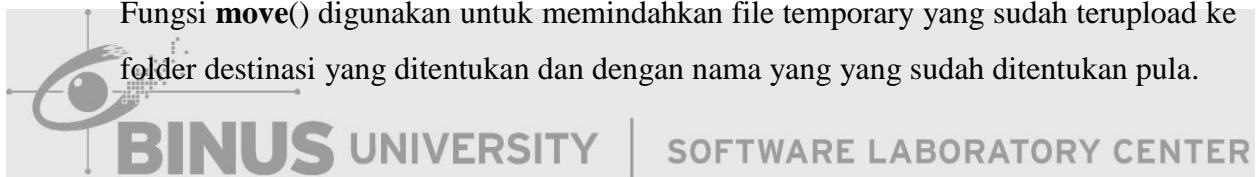
Fungsi **getClientSize()** digunakan untuk mendapatkan ukuran dari file yang akan diupload.

5. getClientOriginalName

Fungsi **getClientOriginalName()** digunakan untuk mendapatkan nama file dari file yang akan diupload.

6. move(lokalasi_destinasi, nama_file)

Fungsi **move()** digunakan untuk memindahkan file temporary yang sudah terupload ke folder destinasi yang ditentukan dan dengan nama yang sudah ditentukan pula.



5.2 Otorisasi (Authorization)

Otorisasi adalah proses menspesifikasikan hak akses yang akan diberikan pada seorang pengguna. Setelah proses otentikasi selesai biasanya akan dilanjutkan dengan proses otorisasi yang akan menentukan fungsi-fungsi yang akan dapat diakses oleh seorang pengguna. Pada subbab ini akan dibahas mengenai cara melakukan otorisasi pada Laravel.

Sebelum mulai menspesifikasikan hak akses dari seorang pengguna, tentu saja pengguna sudah harus melewati proses otentikasi. Untuk memastikan pengguna sudah melewati proses otentikasi, dapat digunakan fungsi **Auth::check()**. Fungsi ini akan mengembalikan nilai **true** jika pengguna sudah terverifikasi pada proses otentikasi, dan nilai **false** jika pengguna gagal terverifikasi pada proses otentikasi. Contoh penggunaan fungsi **Auth::check()** adalah seperti gambar dibawah ini:

```

@if(Auth::check())
    Hi, {{ Auth::user()->name }}
@else
    Hi, Guest
@endif

```

5.2.1 Aturan Otorisasi

Setelah memastikan bahwa pengguna sudah terverifikasi dalam proses otentikasi, proses otorisasi akan dilakukan dengan menggunakan kelas **Gate** untuk membuat aturan yang akan dipakai dalam proses otorisasi. Laravel menyediakan kelas **AuthServiceProvider** yang akan menjadi tempat dimana pengembang menempatkan aturan untuk proses otorisasi. Kelas **AuthServiceProvider** ini terletak pada folder “app/Providers/”. Contoh sederhana dalam membuat aturan untuk proses otorisasi adalah sebagai berikut:

```

<?php

namespace App\Providers;

use Illuminate\Contracts\Auth\Access\Gate as GateContract;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
    ];

    /**
     * Register any application authentication / authorization services.
     *
     * @param \Illuminate\Contracts\Auth\Access\Gate $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);

        $gate->define('update-story', function($user, $story) {
            return $user->id == $story->user_id;
        });
    }
}

```

Kode yang ditandai dengan kotak merah diatas akan membuat suatu aturan dengan nama “update-story” yang akan menentukan bahwa seorang pengguna hanya dapat mengubah *story*

jika *story* yang akan diubah adalah milik pengguna tersebut. Pengguna dan *story* akan dicocokkan berdasarkan parameter yang ditetapkan pada aturan yang dibuat.

5.2.2 Pengecekan Otorisasi

Setelah aturan selesai dibuat, selanjutnya adalah menggunakan aturan tersebut untuk mengotorisasi pengguna yang akan mengubah *story*. Untuk melakukan hal tersebut, buatlah sebuah controller dengan nama **StoryController**. Kemudian di dalam controller tersebut buatlah fungsi **update()** untuk mengubah *story* seperti gambar dibawah ini:

```
class StoryController extends Controller
{
    public function update(Request $request, $id)
    {
        $story = Story::find($id);

        if(Gate::denies('update-story', $story))
        {
            return "You are not authorized";
        }

        return "You are authorized";
    }
}
```

Kode diatas akan mencari *story* sesuai dengan id yang diterima oleh parameter, kemudian melakukan otorisasi pengguna yang sudah terotentikasi atau sudah berhasil *login* menggunakan fungsi **denies()**. Fungsi **denies()** akan menghasilkan nilai **true** jika pengguna gagal dalam proses otorisasi atau tidak memiliki hak akses untuk melakukan *update-story*. Sebaliknya akan menghasilkan nilai **true** jika pengguna berhasil dalam proses otorisasi atau memiliki hak akses untuk melakukan *update-story*.

Jika diperhatikan fungsi **denies()** yang digunakan tidak diberikan parameter *\$user* yang diminta saat membuat aturan **update-story**. Hal ini terjadi karena Laravel akan langsung mengganti parameter *\$user* tersebut dengan data pengguna yang sudah terotentikasi. Jika tidak ada pengguna yang sudah terotentikasi maka secara otomatis proses otorisasi akan digagalkan.

Meskipun begitu Laravel tidak membatasi pengembang untuk hanya dapat melakukan otorisasi pada pengguna yang sudah terotentikasi, tetapi juga dapat melakukan pengecekan otorisasi pada pengguna lainnya. Untuk dapat melakukan hal tersebut maka dapat digunakan kode seperti dibawah ini:

```
if(Gate::forUser($user)->denies('update-story', $story))
{
    return "You are not authorized";
}
```

Kode diatas akan melakukan pengecekan otorisasi pada pengguna yang dispesifikasikan pada fungsi **forUser()**.

Selain menggunakan fungsi **denies()**, pengecekan otorisasi pengguna juga dapat dilakukan dengan menggunakan fungsi **allows()** atau **check()**.

```
public function update(Request $request, $id)
{
    $story = Story::find($id);

    if(Gate::allows('update-story', $story))
    {
        return "You are authorized";
    }

    return "You are not authorized";
}
```

```
public function update(Request $request, $id)
{
    $story = Story::find($id);

    if(Gate::check('update-story', $story))
    {
        return "You are authorized";
    }

    return "You are not authorized";
}
```

Cara kerja dari fungsi **allows()** dan **check()** adalah kebalikan dari fungsi **denies()**, yaitu menghasilkan nilai **true** jika pengguna berhasil terotorisasi, sebaliknya nilai **false** jika pengguna gagal terotorisasi.

Selain menggunakan ketiga fungsi diatas (**denies()**, **allows()**, **check()**) yang berasal dari kelas Gate. Proses otorisasi juga dapat dilakukan menggunakan model user dengan memanfaatkan fungsi **can()** dan **cannot()**, seperti gambar dibawah ini:

```
if ($user->can('update-story', $story)) {
    return 'User authorized';
}
```



```
if ($user->cannot('update-story', $story)) {
    return 'User not authorized';
}
```

Untuk melakukan otorisasi pada blade, dapat digunakan tag **@can** seperti gambar dibawah ini:

```
@can('update-story', $story)
    <!-- The authenticated user is authorized -->
@else
    <!-- The authenticated user is not authorized -->
@endcan
```

5.2.3 Policy

Pembuatan aturan untuk otorisasi yang sudah dicontohkan diatas akan menjadi sangat rumit jika aturan yang dibuat sudah sangat banyak dan ditempatkan hanya pada kelas **AuthServiceProvider**. Untuk mengatasi hal tersebut, Laravel membuat pengembang dapat membagi aturan-aturan otorisasi yang akan dibuat menjadi kelas-kelas *Policy*.

Policy adalah kelas PHP yang mengelompokkan aturan otorisasi berdasarkan sumber daya yang akan diotorisasi dari pengguna. Contohnya adalah aturan update-story dan destroy-story akan berada dalam kelas *policy* StoryPolicy. Untuk membuat kelas *policy* dapat digunakan perintah **make:policy <nama_policy>** seperti gambar dibawah ini.

```
D:\FirstLaravelProject>php artisan make:policy StoryPolicy
```

Perintah diatas akan membuat sebuah kelas policy dengan nama **StoryPolicy** yang akan terletak pada folder “app/Policies”.

Setelah kelas **StoryPolicy** terbuat, maka aturan yang terdapat pada kelas **AuthServiceProvider** dapat dipindahkan ke dalam kelas **StoryPolicy**, seperti gambar dibawah ini:

```
<?php

namespace App\Policies;

use Illuminate\Auth\Access\HandlesAuthorization;

use App\User;
use App\Story;

class StoryPolicy
{
    use HandlesAuthorization;

    /**
     * Create a new policy instance.
     *
     * @return void
     */
    public function __construct()
    {
        //
    }

    public function update(User $user, Story $story)
    {
        return $user->id == $story->user_id;
    }
}
```

Jika sudah memindahkan aturan otorisasi sesuai dengan sumber dayanya masing-masing, selanjutnya adalah mendaftarkan semua *policy* yang sudah dibuat pada kelas **AuthServiceProvider** seperti gambar dibawah ini:

```

<?php

namespace App\Providers;

use Illuminate\Contracts\Auth\Access\Gate as GateContract;
use Illuminate\Foundation\Support\Providers\AuthServiceProvider as ServiceProvider;

class AuthServiceProvider extends ServiceProvider
{
    /**
     * The policy mappings for the application.
     *
     * @var array
     */
    protected $policies = [
        'App\Model' => 'App\Policies\ModelPolicy',
        'App\Story' => 'App\Policies\StoryPolicy',
    ];

    /**
     * Register any application authentication / authorization services.
     *
     * @param \Illuminate\Contracts\Auth\Access\Gate $gate
     * @return void
     */
    public function boot(GateContract $gate)
    {
        $this->registerPolicies($gate);
    }
}

```

Setelah selesai membuat *policy* dan mendaftarkannya pada **AuthServiceProvider** maka otorisasi dapat dilakukan dengan cara yang sama yaitu menggunakan fungsi **denies()**, **allows()**, **check()** dari kelas **Gate**, fungsi **can()**, **cannot()** dari **model**, dan tag **@can** pada **blade template**.

Terdapat satu cara otorisasi tambahan yang hanya dapat dilakukan jika menggunakan kelas *policy*, yaitu dengan menggunakan fungsi **authorize()** pada *controller*. Contoh penggunaannya adalah seperti gambar dibawah ini:

```

public function update(Request $request, $id)
{
    $story = Story::find($id);

    $this->authorize($story);

    return "You are authorized";
}

```

Fungsi **authorize()** diatas akan secara langsung menggunakan *policy* sesuai dengan model yang sudah didaftarkan pada **AuthServiceProvider** dan aturan sesuai dengan nama fungsi dimana

fungsi **authorize()** dipanggil. Karena fungsi **authorize()** dipanggil pada fungsi update maka aturan yang akan digunakan adalah aturan update.

```
protected $policies = [
    'App\Model' => 'App\Policies\ModelPolicy',
    'App\Story' => 'App\Policies\StoryPolicy',
];
```

Jika nama aturan dan fungsi berbeda maka nama aturan yang ingin digunakan dapat dispesifikasi pada fungsi **authorize()** seperti gambar dibawah ini:

```
public function update(Request $request, $id)
{
    $story = Story::find($id);

    $this->authorize('update-story', $story);

    return "You are authorized";
}
```

Untuk melakukan otorisasi pada pengguna lain menggunakan cara yang sama seperti fungsi **authorize()** dapat juga dilakukan dengan menggunakan fungsi **authorizeForUser()** seperti gambar dibawah ini:

```
$this->authorizeForUser($user , 'update-story', $story);
```

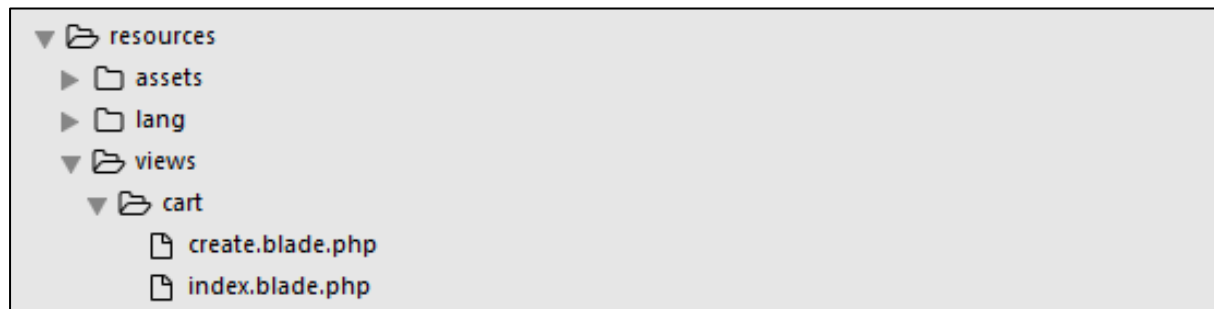
5.3 Session

Session adalah suatu cara untuk menyimpan informasi dalam variable yang dapat digunakan pada file yang berbeda. Session biasa digunakan ketika suatu informasi dibutuhkan untuk dapat diakses dibanyak file atau halaman web.

Pada subbab ini, akan dicontohkan penggunaan session untuk menambahkan suatu item atau data ke dalam keranjang belanja atau *cart* dan menampilkannya. Logika dari hal tersebut adalah id dari data yang ingin ditambahkan ke dalam *cart* akan disimpan pada suatu array, kemudian array tersebut akan disimpan di dalam session agar dapat dengan diakses melalui halaman lain yang akan menampilkan data-data yang telah ditambahkan ke dalam *cart*.

Untuk menimplentasikan logika diatas, buatlah folder “cart” di dalam folder “resources/views”, kemudian di dalam folder cart yang baru saja dibuat, buatlah dua buah halaman yaitu

create.blade.php dan **index.blade.php** yang akan digunakan untuk menambahkan suatu data ke dalam *cart* dan untuk melihat semua data yang telah ditambahkan ke dalam *cart* tersebut.



create.blade.php

```
<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Cart Create</h2>
    <br>
    <table border="1">
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Description</th>
            <th>Action</th>
        </tr>
        @foreach($products as $product)
            <tr>
                <td>{{ $product->id }}</td>
                <td>{{ $product->name }}</td>
                <td>{{ $product->description }}</td>
                <td>
                    <a href="{{ URL::to('/cart/' . $product->id . '/store') }}">
                        <input type="submit" value="Add to Cart">
                    </a>
                </td>
            </tr>
        @endforeach
    </table>
    <br><br>
    <a href="{{ URL::to('/cart') }}">
        <input type="submit" value="View Cart">
    </a>
</body>
</html>
```

index.blade.php

```

<!DOCTYPE html>
<html>
<head>
    <title></title>
</head>
<body>
    <h2>Cart Index</h2>
    <br>
    <table border="1">
        <tr>
            <th>ID</th>
            <th>Name</th>
            <th>Quantity</th>
            <th>Action</th>
        </tr>
        @foreach($items as $item)
            <tr>
                <td>{{ $item->id }}</td>
                <td>{{ $item->name }}</td>
                <td>{{ $cart[$item->id] }}</td>
                <td>
                    <a href="{{ URL::to('/cart/' . $item->id . '/delete') }}">
                        <input type="submit" value="Delete">
                    </a>
                </td>
            </tr>
        @endforeach
    </table>
    <br><br>
    <a href="{{ URL::to('/cart/create') }}">
        <input type="submit" value="View Products">
    </a>
</body>
</html>

```

Setelah membuat *view* (**index.blade.php** dan **create.blade.php**), selanjutnya buatlah *controller* dengan nama “CartController” dan di dalam *controller* tersebut buatlah beberapa fungsi dibawah ini:

1. index

```

public function index()
{
    $cart = session('cart');

    $items = Product::find(array_keys($cart));

    return view('cart.index', compact('items', 'cart'));
}

```

2. create

```
public function create()
{
    $products = Product::all();

    return view('cart.create', compact('products'));
}
```

3. store

```
public function store($id)
{
    $cart = session('cart');

    if (isset($cart[$id]))
        $cart[$id] += 1;
    else
        $cart[$id] = 1;

    session(['cart' => $cart]);

    return redirect('/cart/create');
}
```

4. destroy

```
public function destroy($id)
{
    $cart = session('cart');

    $cart[$id] -= 1;

    if ($cart[$id] == 0)
        unset($cart[$id]);

    session(['cart' => $cart]);

    return redirect('/cart');
}
```

Setelah membuat *view* (**index.blade.php** dan **create.blade.php**) dan *controller* (**CartController.php**), selanjutnya buatlah rute untuk menghubungkan *view* dan *controller* tersebut:

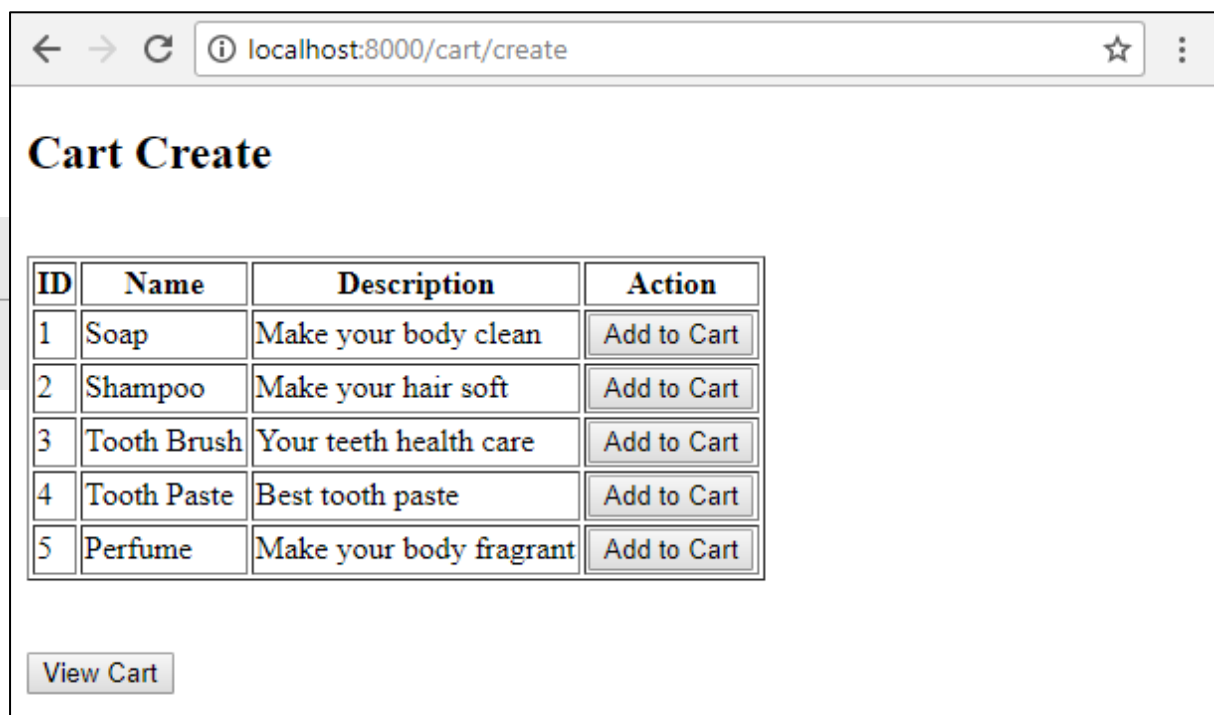
```
Route::get('/cart', 'CartController@index');







Route::get('/cart/create', 'CartController@create');

Route::get('/cart/{id}/store', 'CartController@store');

Route::get('/cart/{id}/delete', 'CartController@destroy');
```

Setelah selesai, maka penggunaan **session** untuk menambahkan produk ke dalam keranjang belanja (*cart*) sudah dapat dicoba dengan mengakses “**localhost:8000/cart**”.









    localhost:8000/cart  

Cart Index

ID	Name	Quantity	Action
1	Soap	2	Delete
2	Shampoo	1	Delete
4	Tooth Paste	3	Delete






View Products

    localhost:8000/cart  

Cart Index

ID	Name	Quantity	Action
1	Soap	1	Delete
2	Shampoo	1	Delete
4	Tooth Paste	3	Delete

View Products

    localhost:8000/cart  

Cart Index

ID	Name	Quantity	Action
1	Soap	1	Delete
4	Tooth Paste	3	Delete

View Products

5.4 Cookies

Cookies adalah informasi-informasi yang dikirimkan oleh aplikasi website dan disimpan pada komputer pengguna aplikasi, tepatnya adalah pada web browser yang digunakan saat mengakses suatu aplikasi website.

Pada Laravel semua cookies yang dibuat sudah dienkripsi dan ditandai dengan kode otentikasi, sehingga suatu cookies akan dianggap invalid atau tidak berlaku lagi jika diubah dari sisi *client* atau pengguna.

Untuk dapat membuat cookie dan mengirimkannya, diperlukan suatu obyek Response yang akan membawa cookie ke halaman atau respon aplikasi lainnya. Setelah membuat obyek Response, cookie dapat dikirimkan menggunakan fungsi **withCookie()** seperti gambar dibawah ini:

```
$response = new Response(view('home'));  
$response->withCookie('mycookie', 'this is my cookie', 5);  
return $response;
```

Fungsi **withCookie** akan menerima 3 buah parameter:

- **Nama cookie**

Nama cookie ini akan menjadi nama untuk mengidentifikasi cookie yang terbuat

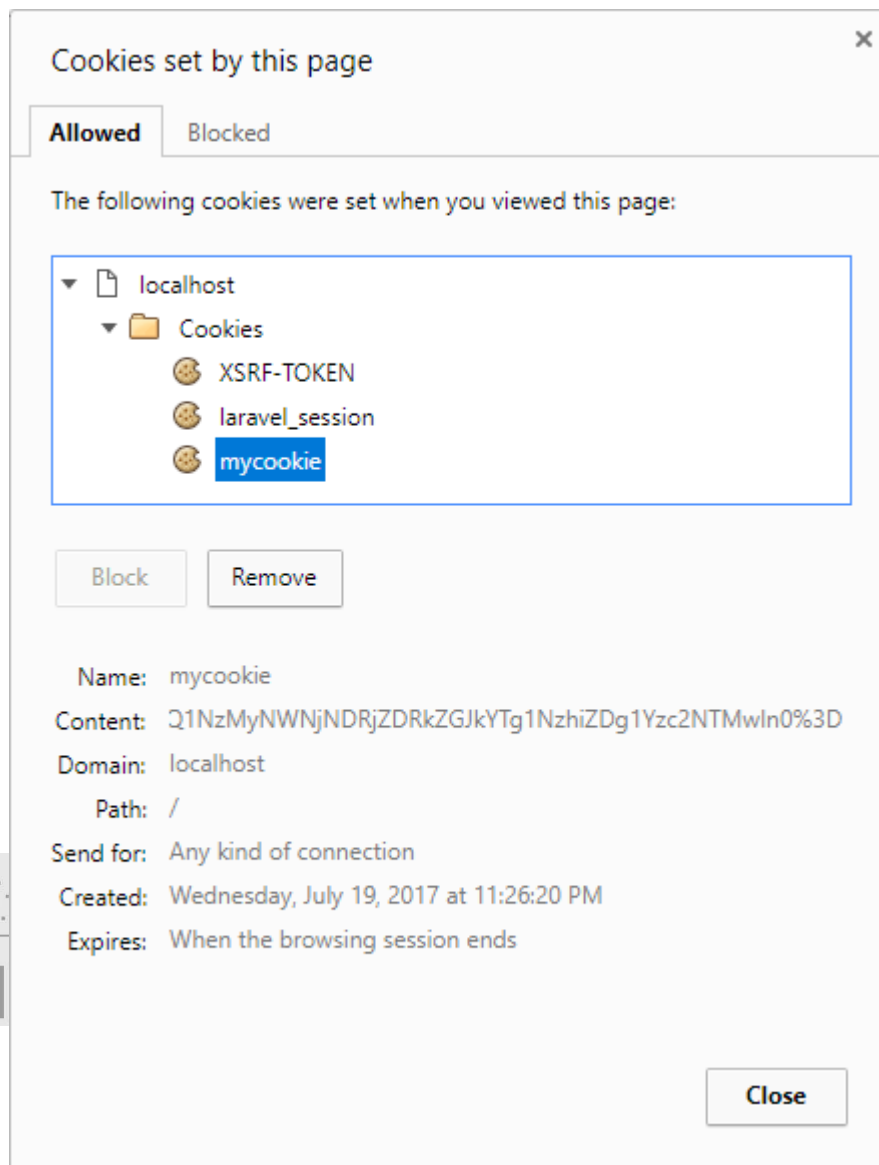
- **Nilai cookie**

Nilai cookie ini akan menjadi nilai yang akan diisi pada cookie yang akan dibuat. Nilai ini akan dienkripsi sebelum dikirimkan bersama response.

- **Masa berlaku cookie**

Nilai pada parameter ini akan menjadi lama waktu cookie yang terbuat akan bertahan dalam satuan menit, jika parameter ini tidak diberikan, maka cookie akan bertahan sampai browser ditutup.

Cookie yang sudah terbuat dapat dilihat dengan membuka cookie yang ada pada browser. Pada contoh ini browser yang digunakan adalah browser “chrome”. Hasil dari kode diatas adalah seperti gambar dibawah ini:



Untuk mendapatkan nilai dari cookie, dapat digunakan fungsi **cookie()** yang akan menerima satu buah parameter yaitu **nama** dari cookie. Contoh penggunaan fungsi cookie adalah seperti gambar dibawah ini:

```
$request->cookie('mycookie');
```

Selain menggunakan cara di atas, masih terdapat cara lain yang juga bisa digunakan yaitu class dengan menggunakan kelas **Cookie** dan **CookieJar**. Contoh penggunaannya adalah seperti gambar dibawah ini:

Cookie

```
Cookie::queue('mycookie', 'this is my cookie', 5);
```

CookieJar

```
public function addCookie(CookieJar $cookieJar)
{
    $cookieJar->queue(cookie('mycookie', 'this is my cookie', 5));
    return back();
}
```

Parameter-parameter yang digunakan pada kedua cara diatas memiliki urutan yang sama dengan parameter yang digunakan pada fungsi **withCookie()** yang sudah dijelaskan diatas.