

FUNDAMENTAL



LUMEN

FUNDAMENTAL

YUDI PURWANTO

Table of contents

Kata Pengantar	4
Persiapan Lingkungan Kerja	5
<i>Tools yang Dibutuhkan</i>	5
<i>Web Browser</i>	5
<i>Web Server</i>	6
<i>PHP</i>	7
<i>Composer</i>	7
<i>Visual Studio Code</i>	9
<i>Git</i>	10
<i>Postman</i>	11
<i>Docker</i>	11
Kesimpulan	11
Pengenalan Microframework Laravel	12
Pengertian MVC	12
Sejarah MVC	12
Arsitektur MVC	13
Contoh MVC	13
Fungsi Lumen	14
Bedanya dengan Framework sejenis?	14
Fitur yang ada di Lumen	15
<i>Route</i>	15
Metode yang tersedia	16
Parameter <i>route</i>	16
Penggunaan <i>Regular Expression</i>	17
Parameter Opsional	17
Alias	17
<i>Group</i>	18
<i>Namespace</i>	18
<i>Prefix</i>	18
Middleware	19
Mendefinisikan <i>Middleware</i>	19
<i>Before/After Middleware</i>	20
Registrasi <i>Middleware</i> Secara Global	21
Registrasi <i>Middleware</i> pada <i>Route</i>	21
Parameter pada <i>Middleware</i>	22
Penggunaan <i>method terminate()</i>	23
<i>Controller</i>	23
<i>Request</i>	24
<i>Response</i>	25
<i>String & Array</i>	25

<i>JSON</i>	26
<i>Eloquent</i>	26
Pengantar	26
<i>Eloquent : Relationship</i>	31
<i>One to One</i>	31
<i>One to Many</i>	32
<i>Many to Many</i>	32
Kesimpulan	33
Instalasi	34
Kebutuhan Server	34
Intalasi Lumen	34
Struktur pada Lumen	35
Menjalankan Lumen	36
Kesimpulan	37
Konfigurasi	38
Konfigurasi <i>Database</i>	38
Registrasi <i>Facedes, Eloquent & Authentication</i>	39
Generate APP_KEY	40
Kesimpulan	40
Studi Kasus - TodoManager	41
Membuat Model dan <i>Migration</i>	41
Membuat Model dan <i>Migration Todo</i>	41
Membuat <i>URL</i> atau <i>API Endpoint</i>	42
Membuat <i>Controller</i>	43
Membuat Controller TodoManager	43
Menguji dengan <i>Postman</i>	46
Kesimpulan	54

Kata Pengantar

Bismillah

Segala puji syukur penulis panjatkan kehadiran Allah Subhanahu Wa Ta'ala karena berkat Rahmat dan Karunianya penulis dapat menyelesaikan penyusunan Buku **Lumen RESTful API** ini.

Shalawat beserta salam semoga senantiasa terlimpah curahkan kepada Nabi Muhammad ﷺ kepada keluarganya, para sahabatnya, hingga umatnya hingga akhir zaman. Aamiin.

Tidak lupa penulis ucapkan terima kasih kepada berbagai pihak baik yang telah membantu kami secara langsung atau tidak langsung dalam proses penyusunan buku ini.



Persiapan Lingkungan Kerja

Pada tahap ini kita akan mempersiapkan dan instalasi kebutuhan yang akan kita gunakan pada tahap-tahap selanjutnya, kita pastikan semua yang dibutuhkan sudah terinstal.

Tools yang Dibutuhkan

Tools-tools yang dibutuhkan untuk mengikuti panduan pada buku ini yaitu:

1. Web Browser
2. Web Server
3. PHP
4. Composer
5. Visual Studio Code
6. Git
7. Postman
8. Docker
9. Niat, Usaha dan Kerja Keras.

Web Browser

Karena merupakan aplikasi berbasis *web*, maka kita perlu *web browser* untuk mengakses aplikasi kita. Ada tiga *web browser* yang penulis rekomendasikan yaitu **Google Chrome**, **Microsoft Edge** dan **Mozilla Firefox**, silakan pilih salah satu saja dan pastikan kita menggunakan versi yang terbaru.

Penulis pribadi menggunakan *browser microsoft edge*, namun jika sudah terbiasa dengan *google chrome* atau *mozilla firefox* atau bahkan ada *browser* selain ketiga di atas silakan menggunakan *browser* tersebut.

Silahkan kunjungi situs resmi untuk proses instalasi.

- [Google Chrome - Download the Fast, Secure Browser from Google](#)
- [Unduh Browser Microsoft Edge Baru | Microsoft](#)
- [Download Firefox Browser — Fast, Private & Free — from Mozilla](#)



Figure 1: Chrome, Firefox, Edge

Web Server

Merupakan *tools* yang terletak di sisi *server* untuk menangani permintaan dari *client*. Ada dua *web server* yang umum digunakan yaitu **Apache** dan **NginX** (dibaca: *engine X*). Meskipun pada saat pengembangan kita sebenarnya tidak harus menggunakan *web server*, karena PHP telah memiliki *built-in web server* sendiri.

Silahkan kunjungi situs resmi untuk proses instalasi.

- [Welcome to The Apache Software Foundation!](#)
- [Download XAMPP \(apachefriends.org\)](#)
- [NGINX | High Performance Load Balancer, Web Server, & Reverse Proxy](#)



Figure 2: apache dan nginx

PHP

Merupakan *tools* untuk menjalankan bahasa pemrograman **PHP**. Adapun versi **PHP** yang digunakan adalah versi 7.3 keatas, hal ini berkaitan dengan Lumen yang membutuhkan minimal PHP versi 7.3.

Silahkan kunjungi situs resmi untuk proses instalasi.

<https://www.php.net/manual/en/install.php>

Composer

Composer adalah *dependency manager* untuk **PHP**, yaitu sebuah *tools* yang digunakan untuk memudahkan kita dalam mengelola *library* **PHP** yang digunakan pada aplikasi kita beserta dependensinya.

Silahkan kunjungi situs resmi untuk proses instalasi.

[Composer \(getcomposer.org\)](https://getcomposer.org)



A Dependency Manager for PHP

Latest: **1.10.10** ([changelog](#))

A preview release for our next major version is available!

Try out **2.0.0-alpha3** ([changelog](#)) now using *composer self-update --preview*

Figure 3: Install Composer

Untuk mengetahui apakah **Composer** sudah berhasil terinstal di dalam komputer, kita bisa mengecek dengan menggunakan perintah berikut ini `composer` di terminal/cmd :


```

zhiaphie@DESKTOP-MEG03LU: /mnt/d/Personal/php
$ composer

Composer version 1.10.10 2020-08-03 11:35:19

Usage:
  command [options] [arguments]

Options:
  -h, --help                Display this help message
  -q, --quiet               Do not output any message
  -V, --version             Display this application version
  --ansi                   Force ANSI output
  --no-ansi                Disable ANSI output
  -n, --no-interaction      Do not ask any interactive question
  --profile                Display timing and memory usage information
  --no-plugins              Whether to disable plugins.
  -d, --working-dir=WORKING-DIR If specified, use the given directory as working directory.
  --no-cache               Prevent use of the cache
  -v|vv|vvv, --verbose     Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug

Available commands:
  about      Shows the short information about Composer.
  archive    Creates an archive of this composer package.
  browse     Opens the package's repository URL or homepage in your browser.
  cc         Clears composer's internal package cache.
  check-platform-reqs Check that platform requirements are satisfied.
  clear-cache Clears composer's internal package cache.
  clearcache Clears composer's internal package cache.
  config     Sets config options.
  create-project Creates new project from a package into given directory.
  depends    Shows which packages cause the given package to be installed.
  diagnose   Diagnoses the system to identify common errors.
  dump-autoload Dumps the autoloader.
  dumpautoload Dumps the autoloader.
  exec       Executes a vendored binary/script.

```

Figure 4: Validasi Composer

Jika muncul seperti gambar di atas, maka sekarang sudah bisa membuat proyek baru menggunakan **Composer**.

Catatan : Composer versi 2 sudah tersedia, penulis merekomendasikan jika ada yang masih memakai `composer` versi 1 silahkan *update* menggunakan perintah berikut : `composer self-update --2` jika perintah tersebut error gunakan `sudo` .

Visual Studio Code

Visual Studio Code atau banyak dikenal dengan **VSCode** ini adalah sebuah teks editor yang dikembangkan oleh **Microsoft**.

Silahkan kunjungi situs resmi untuk proses instalasi.

[Visual Studio Code - Code Editing. Redefined](#)

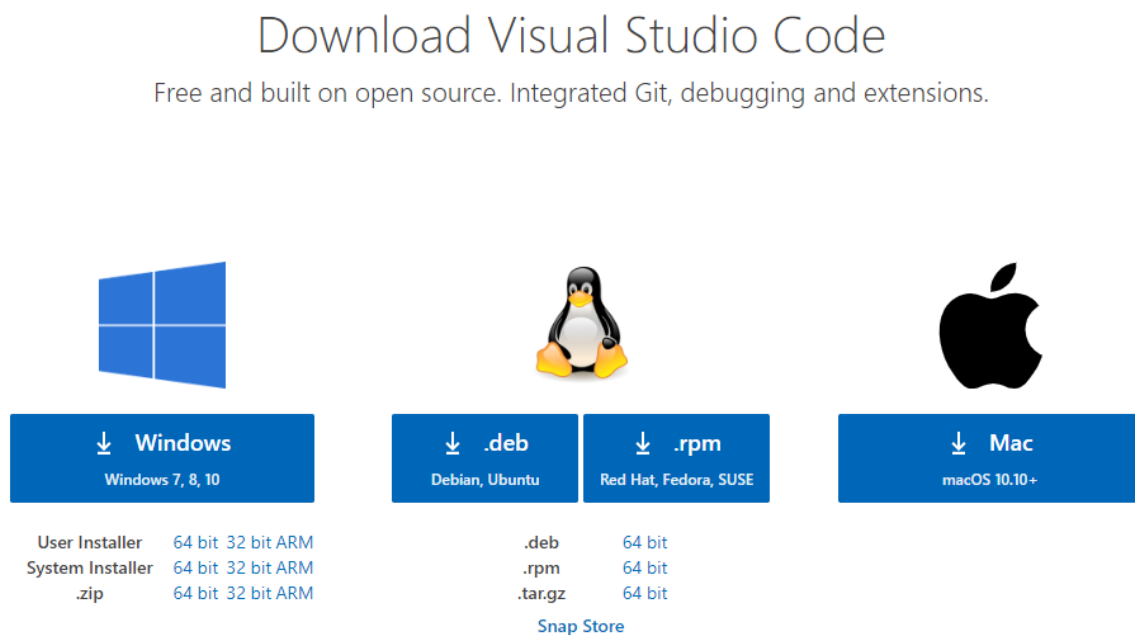


Figure 5: Visual Studio Code

Git

Git merupakan *tools* yang berfungsi untuk mengontrol revisi kode secara terdistribusi dan mengelola kode sumber (*distributed revision control and source code management/SCM*). Dengan menggunakan **Git**, kita akan lebih mudah dalam mengontrol revisi kode sumber aplikasi kita.

Silahkan kunjungi situs resmi untuk proses instalasi.

<https://git-scm.com/downloads>



Figure 6: git download

Postman

Postman adalah *tool* untuk pengujian apakah *web service* berjalan atau tidak sesuai dengan yang diharapkan maka sebenarnya kita bisa gunakan *tools HTTP client* yaitu **Postman**.

Silahkan kunjungi situs resmi untuk proses instalasi.

<https://www.postman.com/downloads/>



Figure 7: Install Postman

Docker

Docker adalah *platform* perangkat lunak yang memungkinkan kita membuat, menguji, dan menerapkan aplikasi dengan cepat. **Docker** mengemas perangkat lunak ke dalam unit standar yang disebut Kontainer yang memiliki semua yang diperlukan perangkat lunak agar dapat berfungsi termasuk pustaka, alat sistem, kode, dan waktu proses. Dengan menggunakan **Docker**, Kita dapat dengan cepat menerapkan dan menskalakan aplikasi ke lingkungan apa pun dan yakin bahwa kode Kita berjalan.

Docker ini bersifat opsional, karena kita juga bisa bekerja tanpa menggunakan *docker*, namun penulis hanya menyarankan untuk instal, karena pada materi akan dibahas juga penggunaan *docker* pada proyek kita.

Silahkan kunjungi situs resmi untuk proses instalasi.

[Empowering App Development for Developers | Docker](#)

Kesimpulan

Pada tahap ini kita sudah mempersiapkan kebutuhan dan keperluan proses pada tahap selanjutnya, maka dari itu kita sudah siap untuk masuk ke tahap pengenalan dan pembahasan dasar-dasar fitur yang ada pada Lumen.

Pengenalan Microframework Laravel

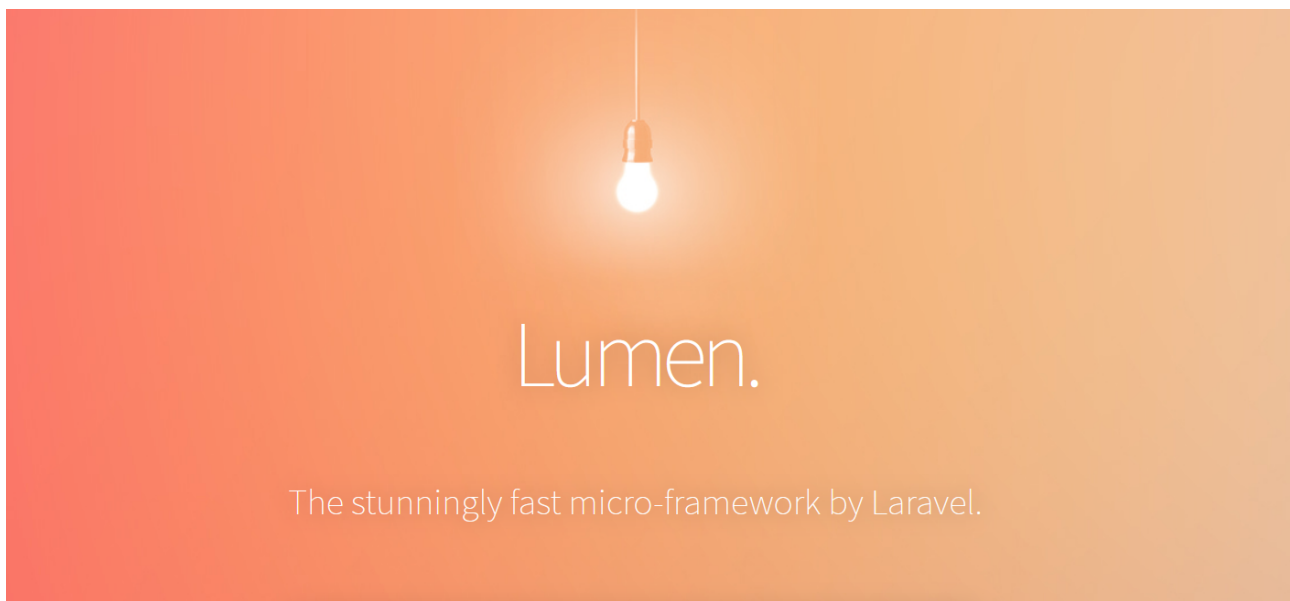


Figure 8: Lumen Framework Home Page

Lumen adalah *micro-framework* yang diciptakan pengembang Laravel untuk mengakomodasi kebutuhan *developer* yang ingin membuat aplikasi dalam skala lebih kecil dengan ekosistem Laravel. Secara konsep, Lumen menerapkan konsep MVC (*Model, View, Controller*).

Pengertian MVC

MVC adalah singkatan dari “*Model View Controller*” merupakan suatu konsep yang sangat populer dalam pembangunan sebuah aplikasi. **MVC** memisahkan pengembangan aplikasi berdasarkan 3 jenis komponen utama yaitu manipulasi *data*, *user interface*, dan bagian yang menjadi kontrol aplikasi. Komponen-komponen utama tersebut membangun suatu *MVC pattern* atau bagian yang diberi nama *Model*, *View* dan *Controller*.

Sejarah MVC

- Arsitektur **MVC** pertama kali dibahas pada tahun 1979 oleh Trygve Reenskaug.
- Model **MVC** pertama kali diperkenalkan pada tahun 1987 dalam bahasa pemrograman *Smalltalk*.
- **MVC** pertama kali diterima sebagai konsep umum, dalam artikel 1988.
- Saat ini, pola MVC banyak digunakan dalam aplikasi *web modern*.

Dengan menggunakan konsep **MVC**, suatu aplikasi dapat dikembangkan sesuai dengan kemampuan *PIC*-nya, *Developer* yang menangani bagian *Model* dan *Controller*, sedangkan *Web Designer* yang menangani bagian *View*, sehingga penggunaan arsitektur **MVC** dapat meningkatkan *maintainability* dan pengorganisasian kode.

Meskipun begitu, namun tetap dibutuhkan komunikasi yang baik *Developer* dan *Web Designer* dalam menangani variabel dan parameter *data* yang ada.

- **Model** adalah kode-kode untuk *model* bisnis dan data. biasanya berhubungan langsung dengan *database* untuk memanipulasi data (*create, read, update, delete*), menangani validasi dari bagian *controller*, namun tidak dapat berhubungan langsung dengan bagian *view*.

- **View** merupakan bagian yang menangani *presentation logic*. berisi kode-kode untuk ditampilkan.
- **Controller** merupakan bagian yang mengatur hubungan antara bagian *model* dan bagian *view*, *controller* berfungsi untuk menerima *request* dan data dari user kemudian menentukan apa yang akan diproses oleh aplikasi.

Arsitektur MVC

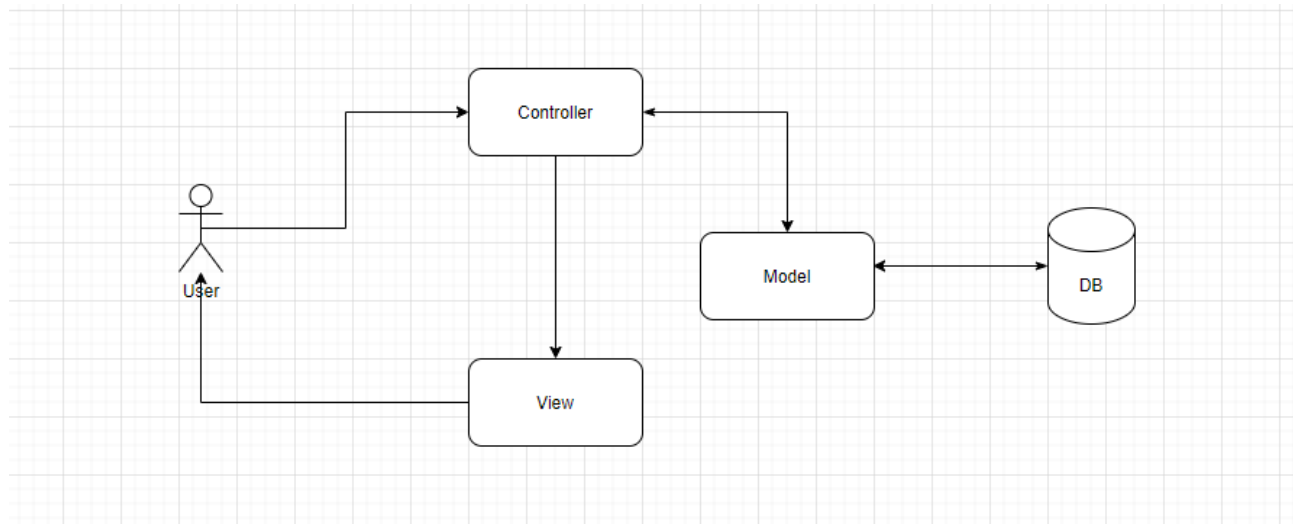


Figure 9: Konsep MVC

Contoh MVC



Figure 10: Contoh MVC

Fungsi Lumen

Secara umum Lumen digunakan untuk membuat *web service* **RESTful API** dengan serialisasinya menggunakan *data format JSON*.

Pembuatan *web service* di lumen sangatlah mudah dan praktis, karena Lumen sendiri dari awal sudah didesain untuk membuat **RESTful API**.

Namun jangan berkecil hati, Lumen juga dapat digunakan untuk membuat *website* pada umumnya, misalnya membuat portal sistem informasi. Kendalanya kembali lagi pada tujuan lumen dibuat. Ketika lumen untuk membuat aplikasi *website* maka *librarynya* kurang begitu lengkap seperti halnya Laravel.

Bedanya dengan Framework sejenis?

Ada beberapa *Micro Framework* lain yang dikhususkan untuk membuat **RESTful API** pada PHP seperti **Slim Framework**, **Silex** dan lain-lain, tentu dengan keistimewaannya masing-masing, namun yang menarik adalah Lumen dalam lamannya mampu menhandel hingga 1900 *request per second*. Lebih unggul dari **Slim Framework** yang hanya dapat menhandel 1800 *request per second*.

Perbedaan lain yaitu Lumen menggunakan PHP versi 7 sebagai syarat penggunaannya, sedangkan Slim masih bisa digunakan PHP versi 5 ke atas.

Benchmark Breaking Speed

Lumen is the perfect solution for building Laravel based micro-services and blazing fast APIs. In fact, it's one of the fastest micro-frameworks available. It has never been easier to write stunningly fast services to support your Laravel applications.

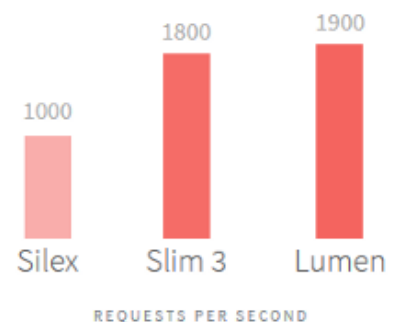


Figure 11: Benchmark 1

Fitur yang ada di Lumen

Lumen sudah dibundel dengan **Eloquent ORM** agar proses *query* lebih mudah dan tidak memakan waktu karena kita tidak perlu lagi menulis *query* yang panjang.

Lumen sendiri sangat mirip sekali dengan Laravel, dimana lumen sudah ada beberapa fitur seperti:

- Routing
- Middleware
- Controller
- Requests
- Responses
- Database
- Queues
- Event
- Unit Test
- dan Lainnya.

Mari kita belajar lebih dalam lagi mengenai fitur-fitur yang ada pada lumen.

Route

Route berfungsi untuk mengatur lalu lintas berdasarkan *request* dari pengguna. *File Route* terletak di dalam folder `/routes/web.php`.



```
blog > routes > web.php
1  <?php
2
3  /** @var \Laravel\Lumen\Routing\Router $router */
4
5  /*
6   *
7   * Application Routes
8   *
9   * Here is where you can register all of the routes for an application.
10  * It is a breeze. Simply tell Lumen the URIs it should respond to
11  * and give it the Closure to call when that URI is requested.
12  *
13  */
14
15
16  $router->get('/', function () use ($router) {
17      return $router->app->version();
18  });
```

Figure 12: route default lumen

Metode yang tersedia

Berikut adalah metode *HTTP* yang ada pada Lumen, sama persis dengan Laravel. Karena ini metode standar pada *HTTP*.

```
$router->get($uri, $callback);  
$router->post($uri, $callback);  
$router->put($uri, $callback);  
$router->patch($uri, $callback);  
$router->delete($uri, $callback);  
$router->options($uri, $callback);
```

Parameter route

Pastinya kita perlu untuk menangkap segmen **URI** dalam *route*. Misalnya, Kita perlu mengambil `id` pengguna dari **URL** itu sendiri. Kita dapat melakukannya dengan menentukan parameter *route* sebagai berikut:

```
$router->get('user/{id}', function ($id) {  
    return 'User ID : '.$id;  
});
```

Pada contoh kode di atas ketika kita jalankan dan akses *url* berikut `http://localhost:8000/user/1` maka akan otomatis tercetak pada *web browser* `User ID : 1`.

Seperti pada gambar dibawah ini:

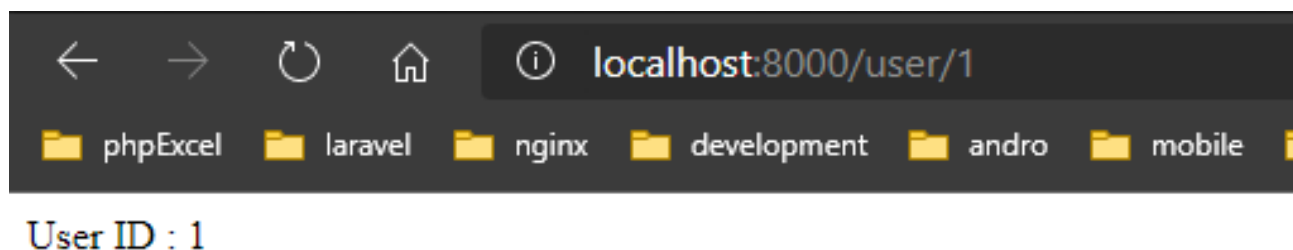


Figure 13: route parameter

Kita juga bisa memberikan parameter lebih dari satu, karena alasan kompleksitas aplikasi kita membutuhkan parameter lebih dari satu.

Seperti baris kode berikut ini:

```
$router->get('posts/{postId}/comments/{commentId}', function ($postId, $commentId) {  
    return 'Post ID : '.$postId. ' Comment ID : '.$commentId;  
});
```

Pada contoh kode di atas akan menghasilkan respon seperti gambar di bawah ini:

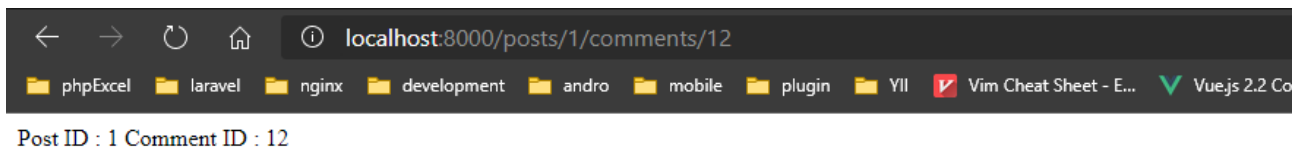


Figure 14: Route

Penggunaan *Regular Expression*

Kita juga dapat membatasi format parameter pada *route* dengan menggunakan *Regular Expression* dalam definisi *route* seperti baris kode berikut:

```
$router->get('user/{name:[A-Za-z]+}', function ($name) {
    //
});
```

Parameter Opsional

Kita dapat menentukan parameter *route* secara opsional yang artinya parameter tersebut tidak mandatori dengan menyertakan bagian dari definisi *URI*. Menempatkan parameter opsional di tengah definisi *route*:

```
$router->get('user/{name?}', function ($name = null) {
    return $name;
});
```

Alias

Kita juga dapat menetapkan alias untuk *route* menggunakan *key as* pada saat menentukan *route*:

```
$router->get('profile', ['as' => 'profile', function () {
    //
}]);
```

Secara spesifikasi kita juga bisa menentukan *controller* tertentu, seperti baris kode berikut ini:

```
$router->get('profile', [
    'as' => 'profile', 'uses' => 'UserController@showProfile'
]);
```

Group

Seperti halnya pada laravel, pada lumen sendiri juga mendukung penamaan secara pengelompokan, kita bisa menggunakan *method* `$route->group`

Kegunaan `$route->group` untuk pengelompokan *route* dalam grup tertentu, kita juga dapat menggunakan `middleware` dalam atribut grup tersebut.

Seperti baris kode berikut ini:

```
$router->group(['middleware' => 'auth'], function () use ($router) {  
    $router->get('/', function () {  
        // Uses Auth Middleware  
    });  
  
    $router->get('user/profile', function () {  
        // Uses Auth Middleware  
    });  
});
```

Namespace

Kasus penggunaan umum lainnya untuk grup *route* adalah menggunakan *namespace* **PHP** yang sama ke grup pengontrol.

Kita dapat menggunakan parameter *namespace* di atribut grup:

```
$router->group(['namespace' => 'Admin'], function() use ($router)  
{  
    // Using The "App\Http\Controllers\Admin" Namespace...  
  
    $router->group(['namespace' => 'User'], function() use ($router) {  
        // Using The "App\Http\Controllers\Admin\User" Namespace...  
    });  
});
```

Prefix

Atribut grup *prefix* dapat digunakan untuk mengawali setiap *route* dalam grup dengan **URI** tertentu. Misalnya, kita mungkin ingin memberi *prefix* semua **URI** route dalam grup dengan awalan `admin` :

```
$router->group(['prefix' => 'admin'], function () use ($router) {  
    $router->get('users', function () {  
        // Matches The "/admin/users" URL  
    });  
});
```

Contoh kode diatas akan menghasilkan *url* seperti berikut ini `http://localhost:8000/admin/users`

Atau penggunaan secara spesifikasi seperti baris kode berikut ini:

```
$router->group(['prefix' => 'accounts/{accountId}', function () use ($router) {
    $router->get('detail', function ($accountId) {
        // Matches The "/accounts/{accountId}/detail" URL
    });
});
```

Contoh kode diatas akan menghasilkan *url* seperti berikut ini `http://localhost:8000/accounts/12/detail`

Middleware

Istilah *middleware* biasa digunakan untuk menyebut sebuah perangkat lunak yang berperan sebagai “penengah” antara sebuah aplikasi dengan aplikasi lain untuk mempermudah proses integrasi antara aplikasi-aplikasi tersebut.

Dalam konteks **Lumen**, *Middleware* merupakan sebuah *Class* khusus yang berperan sebagai “penengah” antara *request* yang masuk ke *Controller* yang dituju. Secara umum, prinsip kerja *Middleware* adalah menjembatani *request* yang masuk untuk kemudian diproses terlebih dahulu sebelum diberikan kepada *Controller* yang dituju atau diarahkan ke *Controller* yang lain. Dengan menggunakan fitur ini, kita dapat membuat komponen yang *reusable* untuk melakukan pekerjaan-pekerjaan tersebut.

Mendefinisikan *Middleware*

Secara *default* Lumen telah menyediakan satu buah *Middleware* dengan nama `Authenticate.php` yang terletak pada direktori `app/Http/Middleware`. Untuk dapat menambahkan *Middleware* baru kita cukup membuat kelas dengan format seperti berikut ini:

```
namespace App\Http\Middleware;

use Closure;

class MyMiddleware
{
    /**
     * Run the request filter.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @return mixed
     */
    public function handle($request, Closure $next)
    {
        //logic
    }
}
```

Seperti yang terlihat pada kode di atas, dalam setiap *Middleware* terdapat sebuah *method* khusus yang bernama `handle()`. *Method* tersebut memiliki dua buah parameter yaitu `Illuminate\Http\Request $request` dan `Closure $next`. *Method* ini akan dipanggil secara otomatis oleh Lumen ketika kita meregistrasikan *middleware* tersebut. Lalu, kode seperti apakah yang harus kita letakkan di dalam `method handle()` tersebut. Kita bisa meletakkan logika yang kita inginkan. Misal, seperti berikut ini.

```
public function handle($request, Closure $next)
{
    if ($request->input('age') < 17) {
        return redirect('home');
    }

    return $next($request);
}
```

Contoh kode di atas merupakan kode sederhana yang melakukan pengecekan terhadap usia dari *user*. Apabila *user* mengisikan usia lebih dari 17 tahun, maka kita akan meneruskan *request* yang masuk dan memberikannya kepada *Controller* yang dituju (dengan memanggil `next(request)`). Namun, apabila *user* yang melakukan *request* usia kurang dari 17 tahun maka kita akan me-redirect *user* tersebut ke halaman *home*. Dengan cara ini kita dapat mem-filter setiap *request* yang masuk dengan mudah.

Before/After Middleware

Secara umum, *Middleware* pada lumen dapat digolongkan kedalam dua kelompok yaitu *After Middleware* dan *Before Middleware*. *After Middleware* merupakan *Middleware* yang diproses setelah *request* masuk kedalam *Controller*, sedangkan *Before Middleware* merupakan *Middleware* yang diproses sebelum *request* masuk kedalam *Controller*.

Berikut ini adalah contoh mendefinisikan kedua buah *Middleware* tersebut.

```
namespace App\Http\Middleware;

use Closure;

class BeforeMiddleware
{
    public function handle($request, Closure $next)
    {
        //lakukan sesuatu terhadap request yang masuk..

        return $next($request);
    }
}
```

Contoh kode di atas merupakan contoh bagaimana mendefinisikan sebuah *Before Middleware* pada Lumen. Pada kode tersebut terlihat bahwa *request* yang masuk akan diproses terlebih dahulu sebelum diteruskan ke *Controller* yang dituju.

```
namespace App\Http\Middleware;

use Closure;

class AfterMiddleware
{
    public function handle($request, Closure $next)
    {
        $response = $next($request);

        //lakukan sesuatu terhadap response yang diperoleh

        return $response
    }
}
```

Berbeda dengan *Before Middleware*, pada *After Middleware* kita meneruskan *request* yang masuk ke *Controller* yang dituju terlebih dahulu hingga mendapatkan *response* dari *Controller* tersebut. Setelah mendapatkan *response* yang dimaksud, kita akan memprosesnya lebih lanjut sebelum nantinya dikembalikan ke *user*.

Registrasi *Middleware* Secara Global

Setiap *Middleware* yang dibuat dapat diregistrasikan secara *global* sehingga *Middleware* tersebut akan selalu dipanggil setiap ada *request* yang masuk. Untuk dapat meregistrasikan *Middleware* yang dibuat secara *global*, kita dapat menambahkannya di dalam file `bootstrap/app.php` seperti contoh di bawah ini:

```
$app->middleware([
    App\Http\Middleware\MyMiddleware::class
]);
```

Registrasi *Middleware* pada *Route*

Selain dapat didefinisikan secara *global*, komponen *Middleware* yang dibuat juga dapat diregistrasikan secara spesifik untuk digunakan pada *Routes* tertentu dengan cara menambahkan nama *Middleware* yang dibuat kedalam file `bootstrap/app.php` seperti contoh dibawah ini:

```
$app->routeMiddleware([
    'auth' => App\Http\Middleware\Authenticate::class,
]);
```

Setelah meregistrasikan *Middleware* kita kedalam `$routeMiddleware` seperti contoh diatas, berikutnya adalah memberitahu Lumen, *Routes* mana saja yang akan menggunakan *Middleware* tersebut dengan cara menambahkan *key middleware* pada bagian *route option* yang berada di file `routes/web.php` seperti contoh dibawah ini:

```
$router->get('admin/profile', ['middleware' => 'auth', function () {
    //
}]);
```

Menggunakan *multiple middleware* sekaligus pada *route*:

```
$router->get('/', ['middleware' => ['first', 'second'], function () {
    //
}]);
```

Parameter pada *Middleware*

```
namespace App\Http\Middleware;

use Closure;

class RoleMiddleware
{
    /**
     * Run the request filter.
     *
     * @param \Illuminate\Http\Request $request
     * @param \Closure $next
     * @param string $role
     * @return mixed
     */
    public function handle($request, Closure $next, $role)
    {
        if (!$request->user()->hasRole($role)) {
            // Redirect...
        }

        return $next($request);
    }
}
```

Contoh kode di atas terlihat bahwa pada *method* `handle()` terdapat penambahan parameter `$role`. Untuk penerapan pada *route* bisa seperti kode di bawah ini.

```
$router->put('post/{id}', ['middleware' => 'role:editor', function ($id) {
    //
}]);
```

Parameter *middleware* dapat ditentukan saat menentukan *route* dengan memisahkan nama *middleware* dan parameter bisa diletakkan setelah simbol `:`

Contoh kode di atas terlihat bahwa parameter dapat diletakkan diantara tanda `:`.

Penggunaan *method terminate()*

```
namespace Illuminate\Session\Middleware;

use Closure;

class StartSession
{
    public function handle($request, Closure $next)
    {
        return $next($request);
    }

    public function terminate($request, $response)
    {
        // Store the session data...
    }
}
```

Contoh kode di atas *method terminate* harus menerima *request* dan *response*. Setelah kita menentukan *middleware*.

Saat memanggil *method terminate* di *middleware*, Lumen akan menyelesaikan *instance* baru *middleware* dari *service container*. Jika ingin menggunakan *instance middleware* yang sama saat *method handle* dan *terminate* dipanggil, daftarkan *middleware* dengan *container* menggunakan *method singleton*.

Controller

Dalam konsep **MVC** biasanya sebuah *URL* dipetakan ke sebuah *controller*, dan karena konsep **MVC** sudah mendarah daging di kalangan *Developer*, maka kita akan mencoba membuat satu halaman dengan *url /halo-dunia*.

Pertama kita harus daftarkan *controller* ke *routes*.

Buka kembali *file routes/web.php*, lalu tambahkan *route* baru seperti kode di bawah ini:

```
$router->get('halo-dunia', 'HaloController@index');
```

Penjelasan dari kode di atas, jika ada yang *request url /halo-dunia*, maka lumen akan mengeksekusi fungsi *index()* di dalam *HaloController*. Setelah itu maka langkah selanjutnya adalah membuat *controller*.

Buat *file* baru dengan nama *HaloController* pada direktori *app/Http/Controllers/HaloController.php*:

```
namespace App\Http\Controllers;

class HaloController
{
    public function index()
```

```
{  
    return 'Halo Dunia';  
}
```

Jalankan aplikasi lumen dan lalu buka <http://localhost:8000/halo-dunia> dan lihat hasilnya. Pada tahap ini kita sudah berhasil membuat halaman pada lumen dengan memanfaatkan *route* dan *controller*.

Ada tip menarik yang bisa kita terapkan pada lumen atau laravel. Ketika kita membuat *controller* dengan satu *method* atau *function* maka kita cukup menggunakan *magic function* bawaan PHP yaitu `__invoke()`.

Lakukan perubahan pada file `routes/web.php` seperti berikut ini:

```
$router->get('halo-dunia', 'HaloController');
```

dan lakukan juga perubahan pada file *controller* `app/Http/Controllers/HaloController.php` menjadi seperti berikut ini:

```
namespace App\Http\Controllers;  
  
class HaloController  
{  
    public function __invoke()  
    {  
        return 'Halo Dunia';  
    }  
}
```

Kita coba akses kembali, buka <http://localhost:8000/halo-dunia> dan lihat hasilnya. Contoh kode pada *route* dan *controller* lebih ringkas dan mudah untuk kita baca.

Request

HTTP Request digunakan untuk mengambil *field* input atau string apapun dalam bentuk metode `GET` dan `POST`.

Caranya sangat mudah untuk menerapkan pada Lumen. Cukup kita mendeklarasikan `Illuminate\Http\Request` ke dalam *controller*.

Contohnya seperti kode di bawah ini :

```
namespace App\Http\Controllers;  
  
use Illuminate\Http\Request;  
  
class UserController  
{
```



```
/**
 * Store a new user.
 *
 * @param Request $request
 * @return Response
 */
public function store(Request $request)
{
    $name = $request->input('name');

    //
}
```

Kode di atas terlihat deklarasi `use Illuminate\Http\Request`, lalu pada *method* `store()` terlihat `class Request` dipanggil. Dengan kata lain kita sudah bisa mengambil apa yang dikirim oleh *user*. Pada contoh tersebut kita hanya mengambil `name` yang dikirim oleh *user*.

Seperti yang terdapat pada Laravel, Lumen juga mendukung *method-method* berikut ini:

```
# Mengambil semua data yang dikirim oleh pengguna
$input = $request->all();

# Mengambil hanya data `username` dan `password` yang dikirim oleh pengguna
$input = $request->only(['username', 'password']);

$input = $request->only('username', 'password');

# Mengambil kecuali data `credit_card` yang dikirim oleh pengguna
$input = $request->except(['credit_card']);

$input = $request->except('credit_card');
```

Response

Laravel menyediakan beberapa cara berbeda untuk mengembalikan respon. Responnya bisa dikirim baik dari *route* maupun *controller*. Semua *route* dan *controller* harus mengembalikan tanggapan untuk dikirim kembali ke pada *user*.

String & Array

Respons paling dasar adalah mengembalikan *string* dari *route* dan *controller*. Ini secara otomatis akan mengkonversi *string* ke *Http* respon.

```
$router->get('halo', function() {
    return 'Halo semua';
});
```

Contoh kode di atas tambahkan pada `routes/web.php` dan ketika kita akses `http://localhost:8000/halo` akan mengembalikan respon `Halo semua`.

JSON

Selain mengembalikan *string* dari *route* dan *controller*, kita juga dapat mengembalikannya ke *array*. Secara otomatis lumen akan mengubah *array* menjadi *response json*.

```
$router->get('halo', function() {  
    return response()->json([  
        'first_name' => 'Yudi',  
        'last_name' => 'Purwanto'  
    ]);  
});
```

Eloquent

Pengantar

Eloquent adalah **ORM** (*Object Relational Model*) yang bisa kita gunakan di dalam Lumen. *Eloquent* sendiri mengimplementasikan *active record* dan digunakan untuk berinteraksi dengan *database*.

Penamaan Tabel Konvensi menggunakan *plural* “snake_case” untuk nama tabel dan *singular* “StudlyCase” untuk nama model. Mari kita coba jabarkan, sebagai contoh: - Tabel *cats* akan memiliki nama model *Cat* - Tabel *jungle_cats* akan memiliki nama model *JungleCat* - Tabel *users* akan memiliki nama model *User* - Tabel *people* akan memiliki nama model *Person*

Eloquent secara otomatis akan mencoba membuat model kita dengan tabel yang memiliki bentuk jamak dari nama model, seperti yang dijelaskan di atas.

Namun, kita bisa menentukan nama tabel untuk mengganti konvensi *default* tersebut.

```
class User extends Model  
{  
    protected $table = 'customers';  
}
```

Pada contoh kode diatas menunjukkan bahwa model *User* akan menghubungkan ke pada table *customers*, penamaan ini bisa kita lakukan dengan memanfaatkan definisi variabel `protected $table = 'nama_tabel'`.

Insert Data Selain membaca data dengan *Eloquent*, Kita juga dapat menggunakannya untuk memasukkan atau memperbarui data dengan *method* `save()`.

Dalam contoh ini kita akan mencoba membuat data *User*:

```
$user = new User();
$user->first_name = 'Yudi';
$user->last_name = 'Baim';
$user->email = 'yudi.baim@example.com';
$user->password = bcrypt('password');
$user->save();
```

Kita juga bisa menggunakan *method* `create()` .

```
User::create([
    'first_name' => 'John',
    'last_name' => 'Doe',
    'email' => 'john.doe@example.com',
    'password' => bcrypt('changeme'),
]);
```

Namun, perlu diingat sebelum menggunakan *method* `create()` , kita harus mendeklarasikan `$fillable` pada model:

```
class User extends Model
{
    protected $fillable = [
        'first_name',
        'last_name',
        'email',
        'password',
    ];
}
```

Sebagai alternatif lain, jika kita ingin membuat semua atribut dapat ditetapkan secara umum, kita dapat mendeklarasikan `$guarded` pada Model. Seperti kode di bawah ini:

```
class User extends Model
{
    /**
     * The attributes that aren't mass assignable.
     *
     * @var array
     */
    protected $guarded = [];
}
```

Select Data Berikut ini adalah contoh menampilkan semua data pada tabel `users` dengan menggunakan *method* `all()` :

```
$users = User::all();  
  
dd($users);
```

Untuk menampilkan data berdasarkan `id` kita bisa menggunakan *method* `find()` atau `findOrFail()`

```
$user = User::find(1);  
//or  
$user = User::findOrFail(2);  
  
dd($user);
```

Update Data Berikut ini adalah contoh memperbarui *data* dengan menggunakan *method* `find()` :

```
$user = User::find(1);  
$user->password = bcrypt('my_new_password');  
$user->save();
```

Atau seperti ini

```
$user->update([  
    'password' => bcrypt('my_new_password'),  
])->where('id', 1);
```

Delete Data Kita dapat menghapus data dengan menggunakan *method* `delete()` .

```
$user = User::find(12);  
$user->delete();
```

Alternatif lainnya, kita juga bisa menggunakan *method* `destroy()` seperti berikut ini.

```
User::destroy(1);  
User::destroy([1, 2, 3]);
```

Soft Delete Terkadang Kita tidak ingin menghapus data secara permanen, tetapi menyimpannya untuk tujuan audit atau tujuan *reporting*. *Eloquent* menyediakan fungsionalitas penghapusan data secara tidak permanen.

Untuk menambahkan fungsionalitas penghapusan ini ke model, kita perlu *import* `class SoftDeletes` dan menambahkannya ke *model*.

Seperti berikut ini.

```
namespace Illuminate\Database\Eloquent\Model;
namespace Illuminate\Database\Eloquent\SoftDeletes;
class User extends Model
{
    use SoftDeletes;
}
```

Namun, sebelum menggunakan fungsionalitas ini, kita perlu pastikan untuk membuat kolom `deleted_at` di tabel terlebih dahulu. Atau dalam migrasi kita harus memanggil *method* `softDeletes()` .

Contoh:

```
Schema::table('users', function ($table) {
    $table->softDeletes();
});
```

Jika tabel `users` sudah terbuat, dan kita ingin menambahkan fungsionalitas penghapusan ini. Kita bisa membuat satu *file migration* dengan perintah ini `php make:migration add_deleted_at_to_users` . Perintah tersebut akan menghasilkan *file* `migration/*add_deleted_at_to_users.php` .

Lalu, kita bisa mengubah *file* tersebut seperti kode di bawah ini.

```
use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

class AddDeletedAtToUsers extends Migration
{
    /**
     * Run the migrations.
     *
     * @returnvoid
     */
    public function up()
    {
        Schema::table('users', function(Blueprint $table) {
            $table->softDeletes();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @returnvoid
     */
    public function down()
    {
        Schema::table('users', function(Blueprint $table) {
```

```

    $table->dropColumn(['deleted_at']);
  });
}
}

```

Setiap *query* akan menghilangkan data yang dihapus secara sementara. Kita dapat melihat data yang telah terhapus tersebut dengan menggunakan *method* `withTrashed()` :

```
User::withTrashed()->get();
```

Kita juga bisa mengembalikan data yang sudah terhapus, menggunakan *method* `restore` .

```

$user = User::find(1);
$user->delete();
$user->restore();

```

Merubah *primary key & timestamps* Secara *default*, *primary key* pada *Model Eloquent* diberi nama `id` , namun kita bisa merubah *primary key* tersebut dengan mendefinisikan variabel `$primaryKey` .

```

class User extends Model
{
    protected $primaryKey = 'id_user';
    // ...
}

```

Selain itu, *primary key* pada *Model Eloquent* integer dan *auto-increment*. Jika *primary key* kita bukan integer (mis. GUID), kita perlu mendefinisikan pada *Model Eloquent* dengan variabel `$incrementing` diberi nilai `false` :

```

class User extends Model
{
    protected $primaryKey = 'uid_user';
    public $incrementing = false;
    // ...
}

```

Secara *default*, *Model Eloquent* mengharuskan adanya kolom `create_at` dan `updated_at` ada di tabel. Jika kita tidak ingin kolom ini secara otomatis dikelola oleh *Eloquent*, kita cukup mendefinisikan variabel `$timestamps` pada model dengan nilai `false` :

```

class User extends Model
{
    public $timestamps = false;
    // ...
}

```

Namun, jika kita perlu menyesuaikan nama kolom yang digunakan untuk menyimpan `timestamps`, kita dapat mengatur dengan cara merubah dan mendefinisikan `CREATED_AT` dan `UPDATED_AT` pada model:

```
class User extends Model
{
    const CREATED_AT = 'tanggal_pembuatan';
    const UPDATED_AT = 'tanggal_update';
    // ...
}
```

Eloquent : Relationship

Dalam memanfaatkan *database*, dikenal ada relasi antar *entity* atau antar tabel. Penggunaan fitur ini pada *database* konvensional membutuhkan penggunaan perintah *SQL* yang relatif panjang dan juga membutuhkan penanganan dari program yang membutuhkan program yang rumit pula.

Berikut adalah beberapa jenis relasi *database* yang dikenal pada umumnya dan telah diakomodasi oleh *Eloquent*.

1. Relasi *one to one* dimana sebuah data pada sebuah tabel hanya memiliki relasi ke sebuah data pada tabel yang lain. Misalnya, sebuah data tabel `users` memiliki relasi 1 nomor telepon di tabel `contacts`.
2. Relasi *one to many* dimana sebuah data pada sebuah tabel memiliki relasi ke beberapa data pada tabel yang lain. Misalnya, sebuah data tabel `categories` memiliki relasi banyak data di tabel `transactions`. Atau dengan kata lain, 1 kategori memiliki banyak data.
3. Relasi *many to one (One to many Inverse)* dimana merupakan kebalikan dari relasi *one to many*. Misalnya kita ingin mengetahui data di table `transactions` memiliki kategori apa, maka relasi ini yang akan digunakan.
4. Relasi *many to many* dimana banyak data pada sebuah tabel memiliki relasi ke banyak data juga pada tabel yang lainnya.

One to One

Relasi *One to One* adalah relasi yang mana setiap satu baris data pada tabel pertama hanya berhubungan dengan satu baris pada tabel kedua. Agar tidak bingung, mari kita coba terapkan pada *eloquent*.

```
class User extends Model
{
    /**
     * Get the phone record associated with the user.
     */
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

Pada contoh kode di atas, *model User* berelasi dengan *model Phone*, secara *default primary key* pada *eloquent* berupa `id` dan *foreign_key* berupa `user_id`. Yang mana *foreign key* itulah yang digunakan sebagai penghubung antara kedua tabel tersebut.

Definisi *Relationship* sebaliknya. Untuk mendapatkan siapa *user* yang memiliki sebuah *phone*, perlu didefinisikan *relationship* sebaliknya (*Inverse Relationships*), dengan menggunakan *method* `belongsTo`

```
class Phone extends Model
{
    /**
     * Get the user that owns the phone.
     */
    public function user()
    {
        return $this->belongsTo('App\User');
    }
}
```

One to Many

Relasi *One to Many* adalah relasi yang mana setiap satu baris data pada tabel pertama berhubungan dengan lebih dari satu baris pada tabel kedua. Agar tidak bingung, mari kita coba terapkan pada *eloquent*.

```
class Post extends Model
{
    /**
     * Get the comments for the blog post.
     */
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
```

Pada contoh kode di atas, sebuah *post* akan memiliki banyak *comments*. *Foreign key* pada contoh ini: `post_id`. Sesuai dengan *snake case* nama model diikuti `_id`.

Many to Many

Relasi *Many to Many* adalah relasi yang mana setiap lebih dari satu baris data dari tabel pertama berhubungan dengan lebih dari satu baris data pada tabel kedua. Artinya, kedua tabel masing-masing dapat mengakses banyak data dari tabel yang direlasikan. Dalam hal ini, relasi *Many to Many* akan menghasilkan tabel ketiga sebagai perantara tabel kesatu dan tabel kedua sebagai tempat untuk menyimpan *foreign key* dari masing-masing tabel. Agar tidak bingung, mari kita coba terapkan pada *eloquent*.


```
class User extends Model
{

    /**
     * The roles that belong to the user.
     */
    public function roles()
    {
        return $this->belongsToMany('App\Role');
    }
}
```

- Relasi ini juga banyak digunakan. Penggunaannya sedikit lebih kompleks dibanding 2 tipe relasi sebelumnya.
- Contoh berikut adalah di mana terdapat `users` , `roles` dan `role_user` . Seorang *user* dapat memiliki banyak role dan sebuah role dapat dimiliki banyak *user*.
- `role_user` merupakan *default* tabel perantara (*intermediate*) atau *pivot*. Nama tabel adalah dari kedua tabel *role* dan *user*
- Relasi didefinisikan dengan menggunakan *method* `belongsToMany` .

Penting, Jika kita ingin menggunakan *Eloquent*, kita harus menghapus komentar panggilan `$app->withEloquent()` pada file `bootstrap/app.php` .

Kesimpulan

Pada tahap ini, kita sudah berkenalan dengan lumen beserta fitur-fitur dasar yang ada pada Lumen itu sendiri. Seharusnya kita sudah bisa menjadikan pemahaman dasar ini untuk siap melanjutkan ke tahap selanjutnya.

Penulis tekankan kembali bahwa penjelasan di atas hanya mencakup dasar-dasar dan penggunaan secara umum dari Lumen. Selebihnya bisa membaca di halaman resmi. <https://lumen.laravel.com/docs>

Instalasi

Kebutuhan Server

Sebelum memulai instalasi, lumen membutuhkan beberapa *extension php* yang harus kita instal, pastikan kembali *extension* berikut ini sudah kita instal:

- PHP ≥ 7.3
- OpenSSL PHP Extension
- PDO PHP Extension
- Mbstring PHP Extension

Intalasi Lumen

Perlu kita ketahui bahwa untuk menginstal Lumen terdapat dua cara, yaitu cara pertama menggunakan Lumen *Installer* dan cara kedua menggunakan *Composer Create-Project*.

1. Melalui Lumen *Installer*

```
composer global require "laravel/lumen-installer"
```

Pastikan untuk mendaftarkan atau mendefinisikan direktori `~/composer/vendor/bin` di **PATH**. sehingga, `lumen` dapat dieksekusi dan dapat ditemukan oleh sistem.

Setelah terinstal, perintah `lumen` akan membuat penginstalan Lumen baru di direktori yang ditentukan. Misalnya kita akan membuat proyek baru bernama `todo`, lumen akan membuat direktori bernama `todo` yang berisi kerangka Lumen secara *default* dengan semua dependensi Lumen yang sudah terinstal. Metode penginstalan ini jauh lebih cepat daripada menginstal melalui `composer`.

Untuk membuat proyek baru dengan lumen cukup jalankan perintah berikut:

```
lumen new todo
```

2. Melalui *Composer Create-Project*

```
composer create-project --prefer-dist laravel/lumen todo
```

Setelah kita jalankan perintah di atas, maka secara otomatis akan melakukan *download repository*.

Catatan, lama tidaknya proses instalasi ini tergantung koneksi *internet* kita.

```
php
$ composer create-project --prefer-dist laravel/lumen test
Creating a "laravel/lumen" project at "./test"
Installing laravel/lumen (v8.1.1)
- Downloading laravel/lumen (v8.1.1)
- Installing laravel/lumen (v8.1.1): Extracting archive
Created project in /mnt/d/Personal/php/test
> @php -r "file_exists('.env') || copy('.env.example', '.env');"
Loading composer repositories with package information
Updating dependencies
Lock file operations: 106 installs, 0 updates, 0 removals
- Locking brick/math (0.9.1)
- Locking doctrine/inflector (2.0.3)
- Locking doctrine/instantiator (1.4.0)
- Locking doctrine/lexer (1.2.1)
- Locking dragonmantank/cron-expression (v3.1.0)
- Locking egulias/email-validator (2.1.24)
- Locking fakerphp/faker (v1.12.0)
- Locking graham-campbell/result-type (v1.0.1)
- Locking hamcrest/hamcrest-php (v2.0.1)
- Locking illuminate/auth (v8.18.1)
- Locking illuminate/broadcasting (v8.18.1)
- Locking illuminate/bus (v8.18.1)
- Locking illuminate/cache (v8.18.1)
- Locking illuminate/collections (v8.18.1)
- Locking illuminate/config (v8.18.1)
- Locking illuminate/console (v8.18.1)
- Locking illuminate/container (v8.18.1)
- Locking illuminate/contracts (v8.18.1)
- Locking illuminate/database (v8.18.1)
- Locking illuminate/encryption (v8.18.1)
- Locking illuminate/events (v8.18.1)
- Locking illuminate/filesystem (v8.18.1)
- Locking illuminate/hashing (v8.18.1)
- Locking illuminate/http (v8.18.1)
```

Figure 15: Proses download lumen

Gambar di atas menunjukkan bahwa pada `Terminal/CMD` kita sedang *download repository* lumen.

Setelah selesai maka terdapat *folder* dengan nama *folder* yang kita tuliskan ketika menginstal Lumen, contoh di atas nama proyek kita adalah `todo`.

Struktur pada Lumen

```
File Edit Selection View Go Run Terminal Help
composer.json - Untitled (Workspace) - Visual Studio Code

composer.json
1 {
2     "name": "laravel/lumen",
3     "description": "The Laravel Lumen Framework.",
4     "keywords": ["framework", "laravel", "lumen"],
5     "license": "MIT",
6     "type": "project",
7     "require": {
8         "php": "^7.3|^8.0",
9         "laravel/lumen-framework": "^8.0"
10    },
11    "require-dev": {
12        "fakerphp/faker": "^1.9.1",
13        "mockery/mockery": "^1.3.1",
14        "phpunit/phpunit": "^9.3"
15    },
16    "autoload": {
17        "psr-4": {
18            "App\\": "app/",
19            "Database\\Factories\\": "database/factories/",
20            "Database\\Seeders\\": "database/seeders/"
21        }
22    },
23 }
```

Figure 16: Struktur Pada Lumen

Gambar di atas merupakan direktori proyek lumen yang di instal, untuk melihat versi lumen nya bisa dilihat pada *file* `composer.json` pada *field* `laravel/lumen-framework`. Saat ini kita menginstal lumen yang terbaru yaitu

^8.0.

Kita melihat banyak folder di sini, tetapi hanya perlu memperhatikan *folder-folder* berikut.

`app/Models` Di sini untuk mendefinisikan model yang melakukan operasi *database*.

`app/http/controllers`: Di sini untuk mendefinisikan logika aplikasi kita.

`app/http/middlewares`: Di sini untuk mendefinisikan keamanan *API* di sini.

`database/migrations`: Di sini untuk menentukan skema *database* di sini.

`routes`: Di sini untuk menentukan tujuan akhir (`end points`) untuk *API*.

Menjalankan Lumen

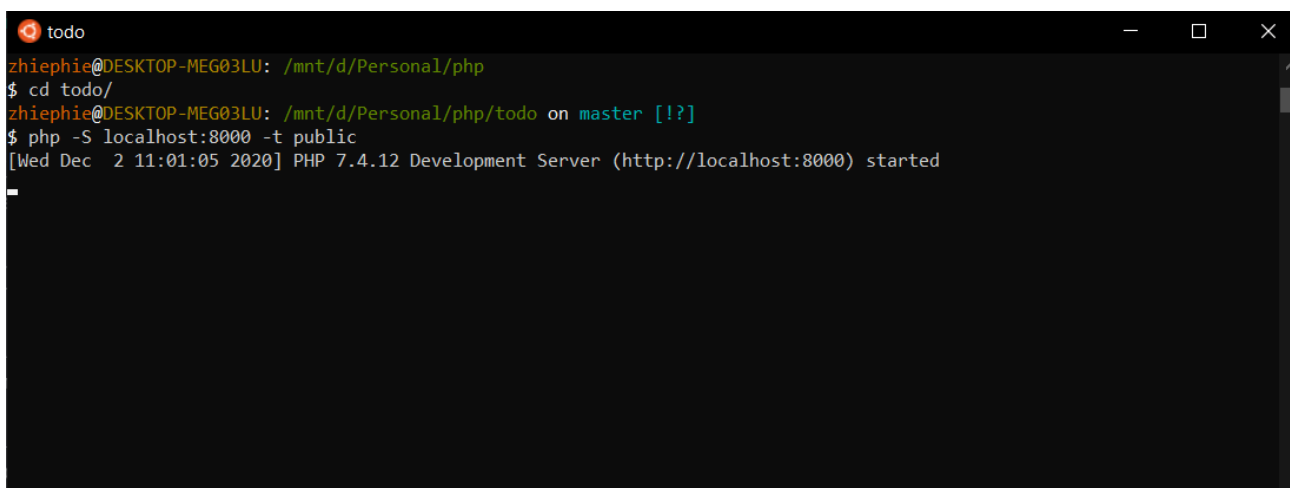
`php` sendiri sudah bisa menjalankan aplikasi secara *standalone*. Kita akan memanfaatkan server bawaan tersebut. Sehingga dalam proses development kita tidak perlu menggunakan web-server seperti apache, nginx dan web-server sejenisnya.

Untuk menjalankan proyek lumen, pastikan sudah berada didalam direktori aplikasi, selanjutnya jalankan perintah berikut ini:

```
php -S localhost:8000 -t public
```

Perintah di atas terdapat angka 8000, artinya lumen berjalan pada *port* 8000. *Port* tersebut juga dapat di ganti, misalnya ingin di ganti 8001.

Berikut hasil dari perintah diatas:



```
todo
zhiephie@DESKTOP-MEG03LU: /mnt/d/Personal/php
$ cd todo/
zhiephie@DESKTOP-MEG03LU: /mnt/d/Personal/php/todo on master [!?]
$ php -S localhost:8000 -t public
[Wed Dec 2 11:01:05 2020] PHP 7.4.12 Development Server (http://localhost:8000) started
```

Figure 17: Pertama Menjalankan Lumen

Jika muncul seperti pada gambar di atas, maka kita tinggal akses melalui browser pada *url* berikut:

`http://localhost:8000`

Lumen (8.2.1) (Laravel Components ^8.0)

Figure 18: Proses Menjalankan Lumen Sukses

Kesimpulan

Pada tahap ini, kita sudah mempelajari cara instal lumen secara benar dan berjalan sesuai harapan. Pada tahap selanjutnya kita akan membahas konfigurasi pada Lumen.



Konfigurasi

Konfigurasi *Database*

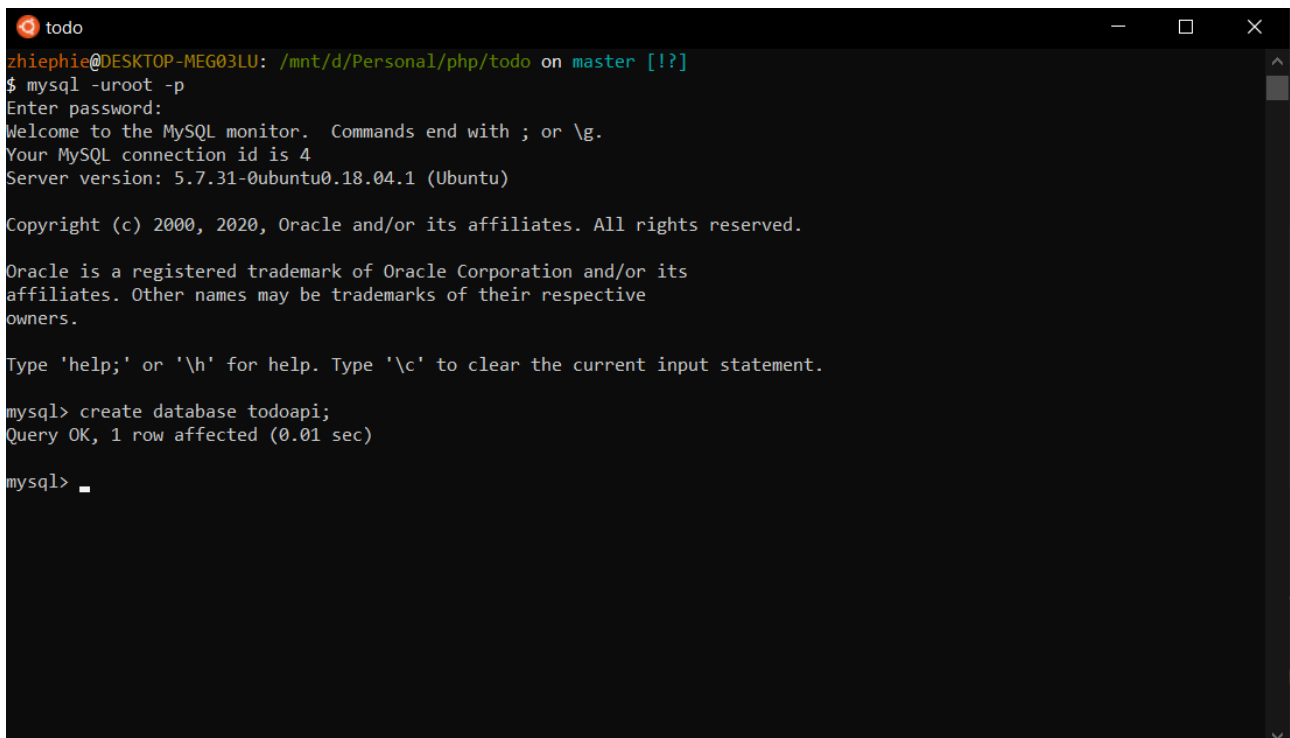
Kita akan belajar bagaimana cara menghubungkan proyek Lumen kita dengan *database*. Sekarang silahkan buka proyek Lumen kita dengan *text editor*, kemudian *copy/rename file* `.env.example` menjadi `.env` kemudian buka *file* `.env` dan cari kode berikut ini:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=homestead
DB_USERNAME=homestead
DB_PASSWORD=secret
```

Kemudian, silahkan rubah dan sesuaikan koneksi *database* berikut ini:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=todoapi
DB_USERNAME=homestead
DB_PASSWORD=secret
```

Di atas, kita akan menggunakan nama *database* `todoapi` dan untuk *password* silahkan disesuaikan dengan koneksi dari **MySQL** masing-masing, jika menggunakan **XAMPP**, maka secara *default password* adalah kosong atau tidak perlu mengisi-nya. Sekarang, kita lanjutkan membuat *database* bisa menggunakan *terminal*, *phpmyadmin* atau *client gui* seperti *navicat*, kemudian buat *database* baru dengan nama `todoapi`. Kurang lebih seperti berikut ini:



```

todo
zhiephie@DESKTOP-MEG03LU: /mnt/d/Personal/php/todo on master [!?]
$ mysql -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 4
Server version: 5.7.31-0ubuntu0.18.04.1 (Ubuntu)

Copyright (c) 2000, 2020, Oracle and/or its affiliates. All rights reserved.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql> create database todoapi;
Query OK, 1 row affected (0.01 sec)

mysql> _

```

Figure 19: Membuat Database

Registrasi *Facades*, *Eloquent* & *Authentication*

Pada *file* `bootstrap/app.php` kita harus menghapus komentar pada bagian-bagian berikut ini:

```

// $app->withFacades();

// $app->withEloquent();

// $app->routeMiddleware([
//     'auth' => App\Http\Middleware\Authenticate::class,
// ]);

// $app->register(App\Providers\AppServiceProvider::class);

// $app->register(App\Providers\AuthServiceProvider::class);

```

Menjadi seperti berikut ini:

```

$app->withFacades();
$app->withEloquent();

$app->routeMiddleware([
    'auth' => App\Http\Middleware\Authenticate::class,
]);

$app->register(App\Providers\AppServiceProvider::class);
$app->register(App\Providers\AuthServiceProvider::class);

```

Pasti ada yang bertanya-tanya mengenai baris kode di atas itu berguna untuk apa, baris kode di atas untuk mengaktifkan fitur `Facade` , `Eloquent` , `authentication` dan `middleware` .

Generate `APP_KEY`

Di Laravel memang ada fitur untuk *generate* secara otomatis dengan perintah `php artisan key:generate` , namun berbeda dengan Lumen, Lumen sendiri tidak dibekali dengan fitur tersebut. Oleh karena itu, penulis memberikan cara mudah dengan cara berikut ini. Salin kode di bawah ini ke *file* `routes/web.php` .

```
use Illuminate\Support\Str;

$route->get('/key', function() {
    return Str::random(32);
});
```

Jalankan perintah berikut ini:

```
php -S localhost:8000 -t public
```

Kemudian, akses `http://localhost:8000/key` , nanti akan menghasilkan nilai seperti `0qu89f2zHqVdDhJuSLe4acxfH5ASg2vr`

Setelah itu, buka *file* `.env` rubah bagian `APP_KEY=` dengan *key* yang dihasilkan oleh fungsi di atas menjadi seperti `APP_KEY=0qu89f2zHqVdDhJuSLe4acxfH5ASg2vr`

Kesimpulan

Pada tahap ini kita sudah belajar konfigurasi pada Lumen. Pada tahap selanjutnya kita akan melanjutkan ke tahap belajar *CRUD* (*create, read, update & delete*)

Studi Kasus - TodoManager

Pada tahap ini menjelaskan secara rinci tentang penerapan Operasi **CRUD** pada Lumen. *Create, Read, Update, & Delete* adalah operasi paling dasar dan penting yang harus dimiliki setiap aplikasi. Membuat Operasi **CRUD** dasar di lumen adalah hal yang sangat sederhana.

Untuk tahap ini kita akan membuat *TodoManager*, di mana kita dapat membuat data baru, menampilkannya sebagai daftar di halaman depan dan mengubah atau menghapusnya.

Membuat Model dan *Migration*

Di tahap sebelumnya kita telah belajar bagaimana cara menghubungkan proyek Lumen yang kita buat dengan *database*. Dan kita juga sudah belajar membuat *database*, maka sekarang kita akan lanjutkan untuk membuat model dan juga *migration*.

Membuat Model dan *Migration* *Todo*

Mari kita jalankan perintah berikut ini melalui `terminal` atau `CMD` :

```
php artisan make:migration create_todos_table --create=todos
```

Perintah di atas akan membuat *migration* untuk tabel `todos` .

Jika perintah di atas berhasil, kita akan mendapati 1 *file* baru yang ada pada direktori `database/migrations/*_create_todos_table.php`, oke mari kita buka *file* tersebut dan merubah seperti kode di bawah ini:

```
public function up()
{
    Schema::create('todos', function(Blueprint $table) {
        $table->id();
        $table->string('name');
        $table->string('description')->nullable();
        $table->timestamps();
    });
}

/**
 * Reverse the migrations.
 *
 * @returnvoid
 */
public function down()
{
    Schema::dropIfExists('todos');
}
```

Selanjutnya kita jalankan perintah berikut ini:

```
php artisan migrate
```

Jika perintah di atas berhasil, maka akan muncul pesan seperti berikut ini.

Migration table created successfully.

Migrating: 2020_12_03_092145_create_todos_table

Migrated: 2020_12_03_092145_create_todos_table (247.49ms)

Sekarang kita akan membuat model `Todo` pada direktori `app/Models/Todo.php`, dan rubah sesuai kode di bawah ini:

```
namespace App\Models;

use Illuminate\Database\Eloquent\Model;

class Todo extends Model
{
    /**
     * The attributes that are mass assignable.
     *
     * @vararray
     */
    protected $fillable = [
        'name', 'description'
    ];
}
```

Membuat *URL* atau *API Endpoint*

Tambahkan kode di bawah ini, ke dalam file `routes/web.php`

```
// API route group prefix /api
$route->group(['prefix' => 'api'], function() use ($router) {
    /**
     * Matches
     * /api/todo (post, get method)
     * /api/todo/id (get, put, delete method)
     */
    $router->post('todo', 'TodoController@store');
    $router->get('todo', 'TodoController@index');
    $router->get('todo/{id}', 'TodoController@show');
    $router->put('todo/{id}', 'TodoController@update');
    $router->delete('todo/{id}', 'TodoController@destroy');
});
```

Kita akan memberikan *prefix* atau awalan pada setiap *url* berupa `api` di semua *route*, untuk mengurangi pengulangan. Kita bisa menggunakan *method* `$router->group` untuk melakukan hal itu.

Membuat *Controller*

Membuat Controller *TodoManager*

Di sini kita akan membuat controller dengan nama `TodoController` yang digunakan untuk manajemen data pada tabel `todos`.

Mari kita buat file `app/Http/Controllers/TodoController.php` dan sesuaikan kodenya seperti di bawah ini:

```
namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Models\Todo;

class TodoController extends Controller
{
    protected $todo;

    public function __construct(Todo $todo)
    {
        $this->todo = $todo;
    }
}
```

Membuat *Create Todo API* Di tahap sebelumnya kita sudah membahas bagaimana membuat *controller*, sekarang kita coba pelajari bagaimana membuat sebuah *method create* data pada tabel `todos`.

Kita buat `function store()`, yang isinya seperti kode di bawah ini:

```
...
public function store(Request $request): JsonResponse
{
    // validate incoming request
    $data = $this->validate($request, [
        'name' => 'required|max:100',
        'description' => 'nullable'
    ]);

    try {
        $todo = $this->todo->create($data);

        //return successful response
        return response()->json([
            'status' => true,
            'message' => 'Data todo berhasil disimpan.',
            'data' => $todo
        ], 201);
    } catch(\Exception $e) {
        //return error message
        return response()->json([
```

```

        'status' => false,
        'message' => 'Create data todo gagal'
    ], 409);
}
}

```

Membuat *List Todo API* Di sini, kita akan mencoba membuat atau menampilkan semua data yang ada pada tabel `todos`, kita hanya perlu membuat `function index()`, yang mana isinya seperti di bawah ini.

```

...
public function index(): JsonResponse
{
    $todos = $this->todo->all();

    return response()->json(
        ['data' => $todos],
        200
    );
}

```

Karena kita sudah mendefinisikan variabel `$todo` pada *construct* yang kita kembalikan pada model `Todo`, maka kita tinggal memanggilnya dengan perintah `$this->todo->method()`. Untuk penggunaan `JsonResponse` ini optional, kita bisa menghapus atau memberikan spesifikasi balikan yang akan kita inginkan.

Membuat *Get Todo API* Di tahap sebelumnya kita sudah membahas bagaimana menampilkan semua data pada tabel `todos`, sekarang kita coba pelajari bagaimana menampilkan satu data `todo`.

Kita buat `function show()`, yang isinya seperti kode di bawah ini:

```

...
public function show(int $id): JsonResponse
{
    $todo = $this->todo->findOrFail($id);

    return response()->json(['data' => $todo], 200);
}

```

Tidak jauh beda fungsi pada *method show* di atas, yang membedakan hanya pada bagian `findOrFail($id)`, yang mana *method* ini akan menghasilkan *query* berupa `where clause`. Dengan kata lain, data yang dihasilkan hanya satu.

Membuat *Update Todo API* Di tahap sebelumnya kita sudah membahas bagaimana membuat data baru, sekarang kita coba pelajari bagaimana memperbaharui data pada tabel `todos`.

Kita buat `function update()`, yang isinya seperti kode di bawah ini:

```
...  
  
public function update(Request $request, int $id): JsonResponse  
{  
    //validate incoming request  
    $data = $this->validate($request, [  
        'name' => 'required|max:100',  
        'description' => 'nullable'  
    ]);  
  
    try {  
        $todo = $this->todo->findOrFail($id);  
        $todo->fill($data);  
        $todo->save();  
  
        //return successful response  
        return response()->json([  
            'status' => true,  
            'message' => 'Data todo berhasil diupdate',  
            'data' => $todo  
        ], 201);  
    } catch(\Exception $e) {  
        //return error message  
        return response()->json([  
            'status' => false,  
            'message' => 'Update data todo gagal'  
        ], 409);  
    }  
}
```

Membuat *Delete Todo API* Di tahap sebelumnya kita sudah membahas bagaimana memperbaharui data, sekarang kita coba pelajari bagaimana menghapus data pada tabel `todos` .

Kita buat `function destroy()` , yang isinya seperti kode di bawah ini:

```
...  
  
public function destroy(int $id): JsonResponse  
{  
  
    try {  
        $todo = $this->todo->findOrFail($id);  
        $todo->delete();  
  
        //return successful response  
        return response()->json([  
            'status' => true,  
            'message' => 'Data todo berhasil dihapus',  
            'user' => $todo  
        ], 201);  
    } catch(\Exception $e) {
```

```
//return error message
return response()->json([
    'status' => false,
    'message' => 'Hapus data todo gagal'
], 409);
}
```

Menguji dengan *Postman*

Pada tahap ini kita akan membahas tentang pengujian *API* yang sudah kita buat dengan **Postman**.

Postman ini merupakan *tool* wajib bagi para *developer* yang berkecukupan pada pembuatan *API*, fungsi utama *postman* ini adalah sebagai *GUI API Caller* namun sekarang *postman* juga menyediakan fitur lain yaitu *Sharing Collection API for Documentation (free)*, *Testing API (free)*, *Realtime Collaboration Team (paid)*, *Monitoring API (paid)*, *Integration (paid)*.

Setting Environment dan Collection Pastikan *postman* sudah terhubung dengan *account google* yang kita miliki karena setiap perubahan di *collection*, *postman* akan melakukan *sync* secara berkala. *setting environment* ini akan memudahkan kita dalam membedakan *environment API development* dan *environment API production*. berikut caranya:

1. Klik bagian yang dipanah merah, nantinya akan memunculkan sebuah dialog.

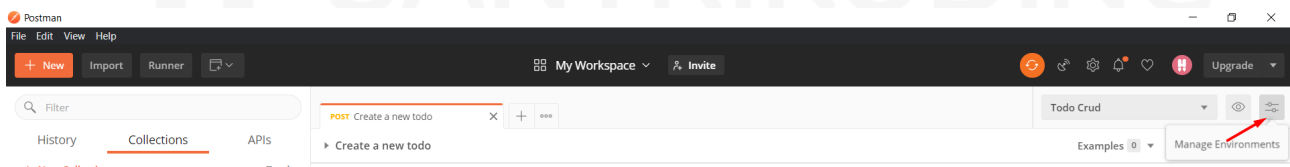


Figure 20: Environment pada Postman

2. Klik tombol *Add*.

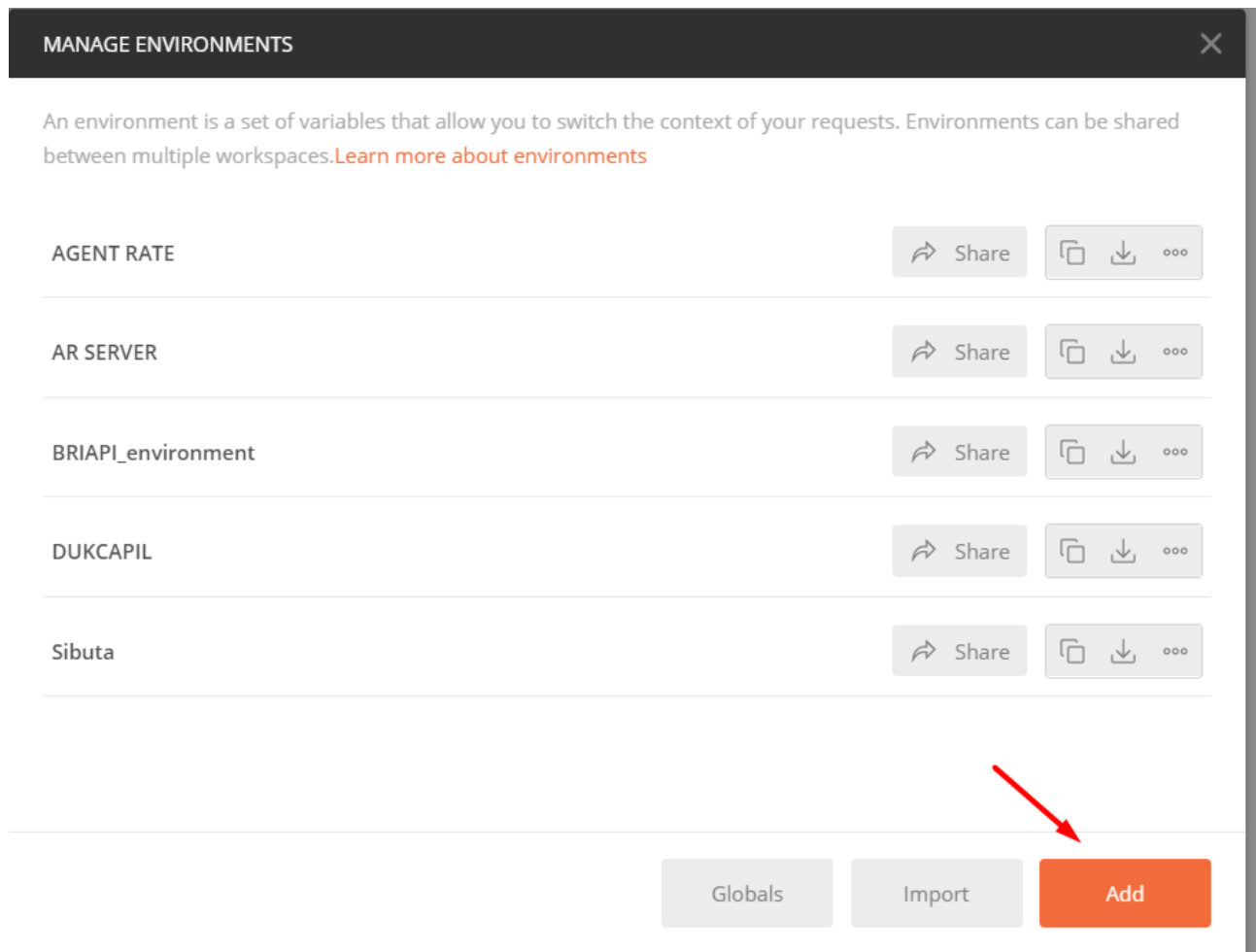


Figure 21: add env postman

3. Tambahkan variabel dan *url* seperti pada gambar di bawah ini.

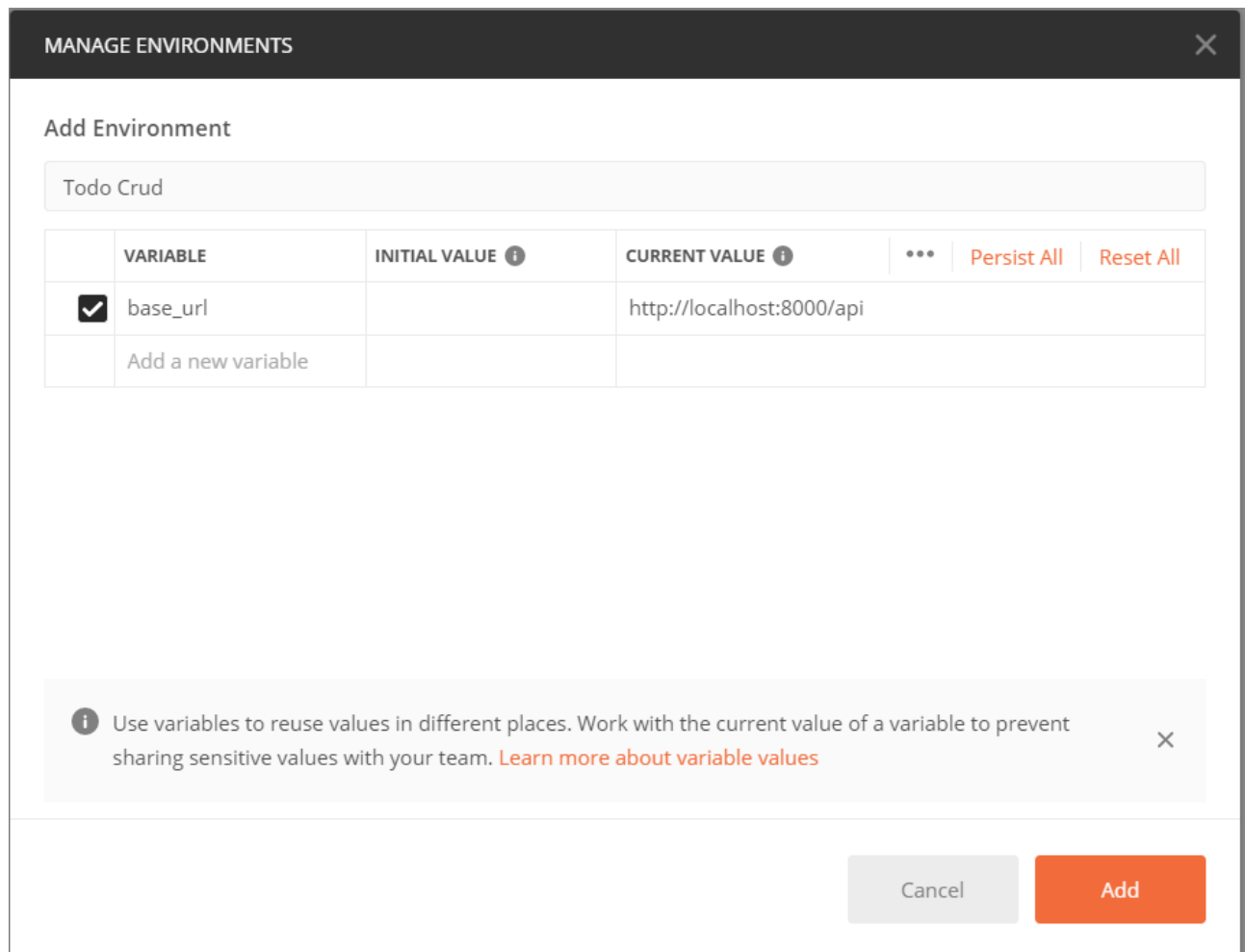


Figure 22: add new env postman

4. Klik *New Collection*

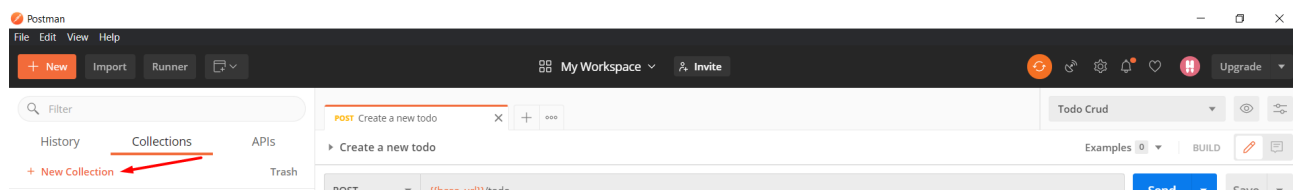


Figure 23: new collection postman

5. Isi nama *Collection* dan klik tombol *Create*, kita juga bisa memberikan sebuah deskripsi pada bagian ini.

CREATE A NEW COLLECTION

✕

Name

Collection Name

Description

Authorization

Pre-request Scripts

Tests

Variables

This description will show in your collection's documentation, along with the descriptions of its folders and requests.

Make things easier for your teammates with a complete collection description.

Descriptions support **Markdown**

Cancel

Create

Figure 24: create_a_collection_postman

Baik, konfigurasi *postman* kita sudah berhasil, sekarang kita akan ketahap pengujian *Restful API* untuk memastikan fungsi berjalan dengan lancar.

Menguji *Create Todo API*

Endpoint

Method	URL
POST	{{base_url}}/todo

HTTP Header

Header Name	Required	Values
Accept	ya	application/json
Content-type	ya	application/json

HTTP Body (x-www-form-urlencoded)

Parameter	Required	Description
name	ya	nama todo
description	ya	keterangan todo

Contoh hasil dari pengujian di atas.

The screenshot displays a REST client interface for a 'Todo Crud' API. The request is a POST to '({{base_url}})/todo' with the body 'x-www-form-urlencoded' containing 'name' and 'description'. The response is a 201 Created status with a JSON body containing details about the created todo item.

Request Details:

- Method: POST
- URL: `{{base_url}}/todo`
- Body Type: x-www-form-urlencoded
- Parameters:

KEY	VALUE	DESCRIPTION
name	Data todo 1	
description	Todo 1 sebagai deskripsi	

Response Details:

```

1 {
2   "status": true,
3   "message": "The Data has been created.",
4   "data": {
5     "name": "Data todo 1",
6     "description": "Todo 1 sebagai deskripsi",
7     "updated_at": "2020-12-05T17:54:19.000000Z",
8     "created_at": "2020-12-05T17:54:19.000000Z",
9     "id": 1
10  }
11 }

```

Status: 201 Created | **Time:** 8.18 s | **Size:** 449 B | **Save Response**

Figure 25: create new todo

Menguji *List* Todo API

Endpoint

Method	URL
GET	{{base_url}}/todo

HTTP Header

Header Name	Required	Values
Accept	ya	application/json

Contoh hasil dari pengujian di atas.

The screenshot shows a REST client interface with a GET request to `{{base_url}}/todo`. The response is a JSON array of todo items. The first item has an id of 1, name 'Data todo 1', and a description 'Todo 1 sebagai deskripsi'.

```

{
  "data": [
    {
      "id": 1,
      "name": "Data todo 1",
      "description": "Todo 1 sebagai deskripsi",
      "created_at": "2020-12-05T17:54:19.000000Z",
      "updated_at": "2020-12-05T17:54:19.000000Z"
    }
  ]
}

```

Figure 26: List All Todo

Menguji Get Todo API

Endpoint

Method	URL
GET	{{base_url}}/todo/1

HTTP Header

Header Name	Required	Values
Accept	ya	application/json

Contoh hasil dari pengujian di atas.

The screenshot shows a REST client interface with a collection of requests. The selected request is 'GET Get 1 Data Todo' with the URL `{{base_url}}/todo/1`. The 'Params' tab is active, showing a query parameter 'id' with the value '1'. The 'Body' tab is also visible, showing the JSON response:

```

{
  "data": {
    "id": 1,
    "name": "Data todo 1",
    "description": "Todo 1 sebagai deskripsi",
    "created_at": "2020-12-05T17:54:19.000000Z",
    "updated_at": "2020-12-05T17:54:19.000000Z"
  }
}

```

The status bar at the bottom indicates a successful response: Status: 200 OK, Time: 1702 ms, Size: 391 B.

Figure 27: Get Todo By ID

Menguji *Update* Todo API

Endpoint

Method	URL
PUT	<code>{{base_url}}/todo/1</code>

HTTP Header

Header Name	Required	Values
Accept	ya	application/json
Content-type	ya	application/json

HTTP Body (x-www-form-urlencoded)

Parameter	Required	Description
name	ya	nama todo
description	ya	keterangan todo

Contoh hasil dari pengujian di atas.

The screenshot shows a REST client interface with a 'PUT Update Data' request selected. The URL is `{{base_url}}/todo/1`. The request body is configured for 'x-www-form-urlencoded' with the following parameters:

KEY	VALUE	DESCRIPTION
<input checked="" type="checkbox"/> name	Data todo 1 telah terupdate	
<input type="checkbox"/> descriptio		
Key	Value	Description

The response is shown in the 'Body' tab, indicating a successful update:

```

1 {
2   "status": true,
3   "message": "The Data has been updated.",
4   "data": {
5     "id": 1,
6     "name": "Data todo 1 telah terupdate",
7     "description": "Todo 1 sebagai deskripsi",
8     "created_at": "2020-12-05T17:54:19.000000Z",
9     "updated_at": "2020-12-06T07:25:25.000000Z"
10  }
11 }

```

The status is 201 Created, Time: 2.02 s, Size: 465 B.

Figure 28: Update Todo Data

Menguji *Delete* Todo API

Endpoint

Method	URL
DELETE	<code>{{base_url}}/todo/1</code>

HTTP Header

Header Name	Required	Values
Accept	ya	application/json

Contoh hasil dari pengujian di atas.

The screenshot shows the Postman interface for a REST client. The top bar displays several request tabs: POST Create a new todo, GET Get All Todos, GET Get 1 Data Todo, PUT Update Data, and DELETE Delete Data Todo (which is the active tab). The main area shows the DELETE request configuration with the URL `{{base_url}}/todo/1` and a dropdown menu set to DELETE. The Headers tab is selected, showing a table with one header: 'Accept' with the value 'application/json'. Below the headers, the Body tab is selected, displaying a JSON response in 'Pretty' format. The response status is 201 Created, with a time of 1465 ms and a size of 465 B. The JSON body contains a success message and details about the deleted todo item.

KEY	VALUE	DESCRIPTION
Accept	application/json	
Key	Value	Description

```

1 {
2   "status": true,
3   "message": "The Data has been deleted.",
4   "data": {
5     "id": 1,
6     "name": "Data todo 1 telah terupdate",
7     "description": "Todo 1 sebagai deskripsi",
8     "created_at": "2020-12-05T17:54:19.000000Z",
9     "updated_at": "2020-12-06T07:25:25.000000Z"
10  }
11 }

```

Figure 29: Delete Data Todo

Kesimpulan

Restful API menjadi bagian penting dalam menghubungkan antara *backend* dengan *frontend* dan *mobile*. Kualitas *Restful API* bisa dilihat dari *response body*, *status code*, *error message* yang dihasilkan. Pastikan ketika *develop API*, kita tidak lupa untuk membuat *design* dan *documentation* antara *backend* dan *frontend*.

Dari tahap ini kita sudah mempelajari membuat *CRUD Todo* menggunakan *Restful API* dengan *lumen*. Dan kita juga sudah melakukan pengujian *Restful API* ini dengan *Postman*.