Introduction to

LARAVEL 4

PHP Framework for Web Artisan

SLC Research and Development

Bina Nusantara University

Table of Contents

I
Introducing Laravel 4

At a Glance

Laravel is a framework for PHP language which heavily utilized Object-Oriented Programming concept. This resolution is visible from its requirement of PHP 5.3, which emphasizes on OOP. As of now, Laravel 4 is the newest version which is still in beta stage, but it is ready to be used for website project.

As taken from its official website,

> "Laravel is a web application framework with expressive, elegant syntax. We believe development must be an enjoyable, creative experience to be truly fulfilling. Laravel attempts to take the pain out of development by easing common tasks used in the majority of web projects, such as authentication, routing, sessions, and caching."
>
> "Laravel aims to make the development process a pleasing one for the developer without sacrificing application functionality. Happy developers make the best code. To this end, we've attempted to combine the very best of what we have seen in other web frameworks, including frameworks implemented in other languages, such as Ruby on Rails, ASP.NET MVC, and Sinatra."
>
> "Laravel is accessible, yet powerful, providing powerful tools needed for large, robust applications. A superb inversion of control container, expressive migration system, and tightly integrated unit testing support give you the tools you need to build any application with which you are tasked."

Is Laravel Suitable for Me?

Do you enjoy:

- Programming in Object Oriented Paradigm?
- Using framework with rigid structure?
- Utilizing Object Oriented Paradigm in PHP?

- Writing beautiful, understandable code?
- Using templates to render organized HTML file?
- Having control over your website routing?

Then this framework is suited for you.

System Requirements

Software Requirements:

- Windows XP SP3 or above, with IIS 5.1 or above and FastCGI Extension/Module.
- PHP-compliant web server
- PHP version 5.3.7 or above.
- MCrypt PHP Extension
- Text Editor. Recommending **Notepad++** (http://notepad-plus-plus.org/download/)

Hardware Requirements:

- Pentium 233-megahertz (MHz) processor or faster (300 MHz is recommended)
- At least 64 megabytes (MB) of RAM (128 MB is recommended)
- At least 200 megabytes (MB) of available space on the hard disk

II
Installation

Composer Setup

Laravel utilizes Composer as its dependency manager. First you would need to install Composer in your server. Download it here (http://getcomposer.org/download/). Windows user can download the installer version for easier installation.

Composer uses command line interface for its usage. In Windows you can use command prompt, while in Linux you will use Shell. This module would base its example in Windows. However, Linux user will find that the syntax for the command is almost always the same.

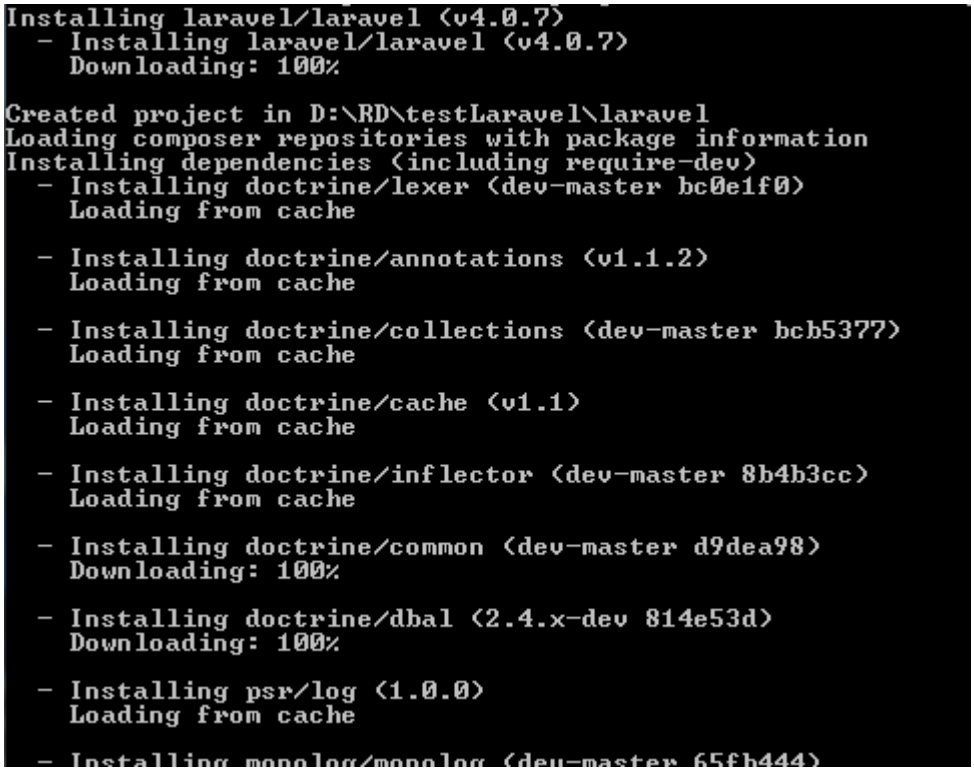
Getting Laravel

There are 2 ways to get Laravel, either by using Composer or downloading archived version of the framework.

- Composer
  Put this line in command line inside your project folder (may need to navigate using `cd`)

```
D:\RD\testLaravel>composer create-project laravel/laravel --prefer-dist
```

  If you are connected to the internet, Composer will download all necessary components for the framework.



- Archive version
  Download the latest version in github. Here's the link for the "master" version (the stable one). If you want to use experimental build or latest alpha version, you may need to look for it yourself in github repository.

  http://github.com/laravel/laravel/archive/master.zip

  Download the archive and extract it to your project folder. It will create a folder named "laravel-master"

Either way, your copy of Laravel is ready. You can try Laravel immediately by opening browser to your project folder, and then to **public** folder of Laravel framework. Laravel has policy to only give authorities for users to access the public folder, although you can go around it by taking the content of public folder outside.

If everything went as expected, the result would be something like this.

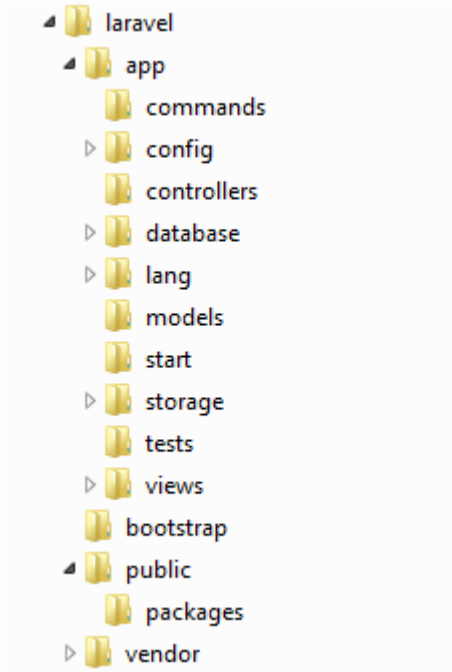
localhost:8099/testLaravel/laravel/public/

You have arrived.

III
Laravel Basic Foundation

Organization

Like many other frameworks, Laravel uses folder organization for its system. Inside your project folder, you will find folders and sub-folders similar to this screenshot.



Now we have a lot of folders to cover, but you actually don't have to change all of them. We will only change the one which is needed for our applications. In the big picture, there are 4 major folders right below laravel folder.

- App
  This is the place for you to change configurations, place models, views, and controllers, and many other things directly changing how your application works.
- Bootstrap

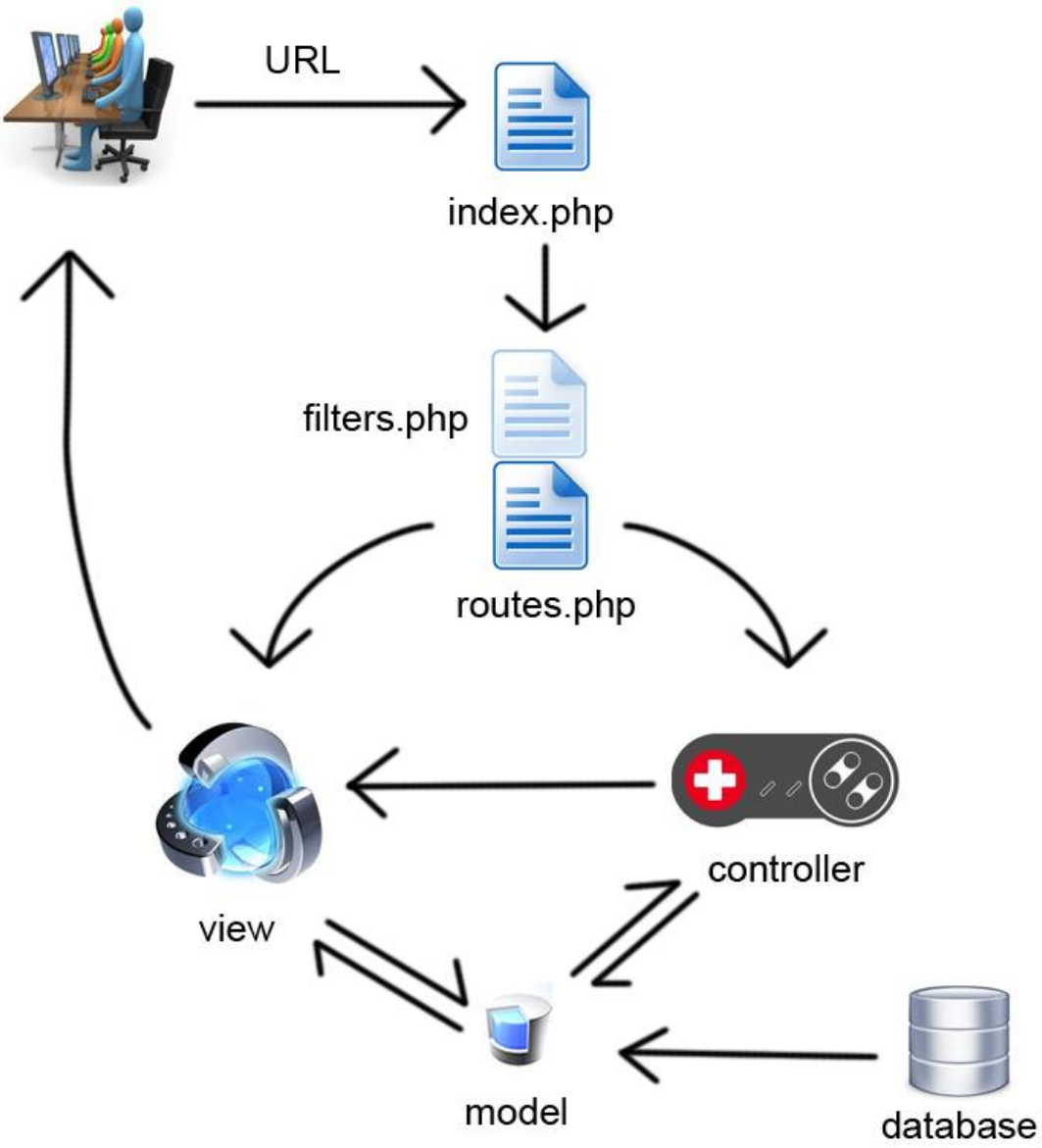
This folder only contains 3 items in the beginning: autoload.php, paths.php, and start.php. As you can see from the name, all of them are used to configure how our applications would start. For example, if we want to load a plugin for each page, and so on.

- Public
  This is the place which can be accessed by visitors and users. Laravel suggests that you put the other 3 folders to be accessible only by applications and admins, so the only point of entry is here. In public there is an index.php which is the very thing which really starts our application after the scripts in bootstrap folder finished preparation.
- Vendor
  Laravel actually uses a lot of 3rd party vendors for its components. This folder contains all of those 3rd party components. Interestingly you may find many familiar names, like symfony and monolog.

Now that you know how the folders are arranged, the common guidelines for making application in Laravel are as follows.

1. Put accessible resources (images, videos, etc.) in public folder.
2. Create views for output to users.
3. Create controllers if there needs additional processing in the application.
4. Create models if there are needs for database access.
5. Configure routing so user can access your view / controller through URL.

The process from user's URL request until the result is shown in the screen can be summed in this diagram. Note that this is only a simple example. You may need to make a lot of configuration, user-made functions and plugins, before-processing and after-processing events, and many other things for serious applications. But in the nutshell, you will always use these basic steps.



Routing

Basic Routing

As we can see from above diagram, routing becomes an important part in Laravel. Rather than using htaccess, we would find it easier and faster to use Laravel's built-in routing function. You can find it in the **app** folders with the name **routers.php**. Try to open it and see the only one example: routing for "/" path (the root path for index.php).

```
Route::get('/', function()
{
    return View::make('hello');
});
```

Reading from this example, we make assumption that for "/" path (which is, empty URL) the framework will make a view named "hello". This is how Laravel make their view, and this notation is PHP way for accessing static methods in a class. So

```
Route::get
```

Means we accessed the static method `get()` inside `Route` class. This is the basic of all routing done by Laravel. The syntax is

```
Route::request_type ( path_or_regex, callback )
```

Request_type can be get or post, depending on what you want to use. **Get** is used for retrieving information, while **post** is usually used for transferring data from user to server. If you want to make a route for any type, use **any**. Most of the time we will be using **get**.

Path_or_regex is the path which will be "caught" by the route. Basically if you put the same URL as the path, Laravel will do whatever you code in **callback**. Path is taken as it is. If the URL missed even one character, then an exception page will be shown.

Callback is a function which is called when the URL matched with the path. As you can see in the example, callback function always returned something. The returned value is the one shown to user, so take care not to forget returning your value.

Try this example.

```
Route::get('user', function()
{
```

```
    return "this is user page";
});
```

When you put “user” in your URL, it will show a blank screen with “this is user page” printed on it. And that is basically how routing is done.

Parameterized Routing

Sometimes, we need to use URL routing to get input from user. For example, imagine showing a route to an article. It could be something like this:

/laravel/public/article/**computer**/**00123-apple-new-iphone**

Computer might be the article topic, while **00123-apple-new-iphone** is the title for our article. In routing, of course **computer** can be changed into something else, maybe **cars** or **music**. Title is also the same. In our routes.php, we might put it like this.

```
Route::get('article/{topic}/{title}', function($topic, $title){
    return
    "
            Topic : $topic<br/>
            Title : $title<br/>
    ";
});
```

Intuition shows that the variable `$topic` and `$title` is taken from `{topic}` and `{title}` in the path. Note that you don’t need to have the same name between the ones in the path and thes one in function. As you can see, the way to show that **topic** and **title** is used as parameter for the callback is by putting them inside { }

You can also make the parameter optional. In previous example, it could be that when there is no title in the URL, we will show the list of titles in that particular topic. To make optional parameter, simply add **?** after the word inside { }

Here’s an example for optional parameter. Note that although **title** is optional, not passing a topic will still result in an exception.

```
Route::get('article/{topic}/{title?}', function($topic, $title = null){
    if($title == null){
        return "List of Topic for : $topic";
    }
    else{
        return
        "
            Topic : $topic<br/>
            Title : $title<br/>
        ";
    }
});
```

Regular Expression in Routing

Using parameterized routing above, we can create rules for our path. Using above example, imagine that **topic** can only be alphabets (no space or numbers) and **title** must start with numbers. We would use Regular Expression to make the rules.

Unfortunately Regular Expression will not be covered in this module specifically. For more information about Regular Expression, head to this link (http://www.regular-expressions.info/tutorial.html)

Now let's move on to how we implement the regular expression. Using above example, we will tweak it a bit into something like this.

```
Route::get('article/{topic}/{title?}', function($topic, $title = null){
    if($title == null){
        return "List of Topic for : $topic";
    }
    else{
        return
        "
            Topic : $topic<br/>
            Title : $title<br/>
        ";
    }
})->where('topic', '[a-zA-Z]+');
```

The difference is in the last line, with **where** statement. This is what used to give regular expression constraint to our parameter. The syntax is

```
Where(param_name, regex)
```

Or

```
Where( array(param_name => regex, param_name => regex, …) )
```

The first one is used if we only want to put one parameter with regular expression. For more than one parameter, we will use the second syntax. Note that you don't have to use regular expression in all parameter; just put it where you really need it.

Using the second syntax, our example would become like this.

```
Route::get('article/{topic}/{title?}', function($topic, $title = null){
    if($title == null){
          return "List of Topic for : $topic";
    }
    else{
          return
          "
                Topic : $topic<br/>
                Title : $title<br/>
          ";
    }
})->where(array(
    'topic' => '[a-zA-Z]+',
    'title' => '[0-9]{3}-\w+'
));
```

The application of this feature is limitless. You can adjust it to your need depending on how the application would work.

Named Routes

This feature helps when we have created a long path, which would take longer time to type rather than a shortened, easy-to-remember name. It can also be used to make routes easier to manage.

To use named routes, put an array in the callback section of the syntax. This is an example.

```
Route::get('user/section/private/home', array('name' => 'userHome',
function()
{
    return "this is user page";
})
);
```

It would give no difference when you call it in URL, but it will be easier for us to call it later in our scripts. For example, above route can now be called as “userHome” rather than the long path.

```
Redirect::route('userHome');
```

View

Actually we can put any HTML or PHP file in public folder, and call it directly in the browser. However, doing so would make our application cluttered with files, and soon we will lost about which is what.

To create better environment for application development, current frameworks used separation concept to differ between files for processing, database access, and output. The concept is MVC (Model-View-Controller). There are other variants of this concept and there is no definite “best” between them. MVC is the concept I used so far and it is satisfactory, while reader might find other concept like MVVM (Model-View-ViewModel) better suited for their project. Laravel by default uses concept similar to MVC. The View we’re talking here refers to **output displayed to user**.

A View is simply pure-HTML or PHP file called by routing (or later, Controller) to be shown to user. In previous example we simply returned plain texts. In real life situation we will output a View. A View is not put in public folder, in other words users could not access them directly. This makes a View safe to use because users could not take them without given authority by routes or Controller.

How to make a View? Simply make a PHP file and save it to **app/views** folder. If this is a vanilla Laravel framework, you will likely find a **hello.php** in this folder. This is the file which shows Laravel logo and "You Have Arrived" text which we saw in 2nd chapter. The file name is important because this is the one we will call in our code, so make sure the name is meaningful to you as developer.

Next we simply call it using either route or Controller. No matter the way, the syntax remained the same.

```
View::make( view_name );
```

With `view_name` the name of your view file minus extension. So if your view file is "user.php", we will put "user" inside the `make` method. Note that if you put "user.php" inside another folder inside view folder, you will have to put the full path in `make`. So if it is "folder1/folder2/user.php", put "folder1/folder2/user" in `make` method.

This will output your PHP file entirely. Sometimes we would need to pass data to our View, maybe from database or other processing results. We can put it either with a single variable or push an array of data at once.

```
View::make('user')->with( var_name , value );

Or

View::make('user', arr );
```

The first line is used if we only want to put one variable. Using with method, we can put `value` inside a variable `var_name`. We then call it in our View as `$var_name`.

The second line put an array inside the View. All indices inside the array will be a variable. So if we have `$arr['name1']`, we can access it as `$name1` in our View.

Here is an example for making a View with some data with route. Note that like using text data, in order for the route to output the View to browser, we have to use **return** statement.

```
Route::get('user', function()
{
    $data = array();
```

```
    $data['arden'] = 'cat';
    $data['budi'] = 'dog';

    return View::make('user', $data);
});
```

Controller

Controller Basics

Controller will become one of the most important parts of our application so far. This is the component which is responsible for processing our data, either from database or user inputs. Controller is a **class**, and to use a Controller means we have to create a new class inheriting from Laravel's core Controller class.

Like View, to make a Controller you will have to create a .php file and call it using its file name in routing. Controllers are located in **app/controllers** folder. By default, you will find **HomeController.php** and **BaseController.php** in that folder. BaseController is a Controller which inherits directly from Laravel's core Controller class. Meanwhile, our user-made Controllers will inherit from this BaseController. Thus we can use BaseController to create a method or attribute which will be shared by all of our Controller class. HomeController is just a sample Controller. Its output is the same as the default "/" route, which displays Laravel logo and a text "You have arrived".

This is the bare minimum of a Controller class' code. All Controllers will be using this as its template.

```
class Controller_name extends BaseController {

    public function method_name ( params )
    {
        //code
    }

}
```

What should we use our controller for? The guideline for MVC concept is to use Controller as our hub, which will be called by our routing and process all necessary transaction logics before outputting it to user, usually by using Views.

Here's an example of a Controller. Notice that Controller name usually starts with a capital letter, and its filename use the same name as its class name. For this example we will give the file name as **ProductController.php**. As with routes, to output the view we need to use **return** statement.

```
class ProductController extends BaseController {

   public function process ( $id )
    {
        $data = 0;
          if($id == 1) $data = 10;
          else if($id == 2) $data = 20;

          return $this->show($data);
    }

   public function show ( $data ){
          return View::make('product')->with('data', $data);
   }

}
```

Now we will call this in our routes.php by using this syntax.

```
Route::get('product/{num}', 'ProductController@process');
```

What can we infer from this code? **{num}** is, as explained before, a parameter which can be inserted in URL. Different from before, this time {num} will be transferred not to anonymous function but to a method defined in our Controller. That is the **ProductController** in the second parameter of **Route::get**. Notice the "@" symbol after the Controller name. It means that the name after that symbol is the method we want to invoke, in this example it is process. As Process takes one input parameter, **$id**, the value of {num} is inserted to it.

Controller can achieve the same capability with anonymous function in routing, but using Controller will make not only routing easier to read, but also simplify routing because there's no need to clutter routes.php with processing logic.

RESTful Controller

While we can put each method into their respective routing, sometimes it is better to make one routing on a Controller, and let Laravel handle how it is routed in the URL. In other words, you only need to have one route to access all methods in a Controller. This is helpful if one Controller have many methods.

This is an example of a Controller with many methods, designed for RESTful concept.

```
class ManyController extends BaseController {

    public function getIndex () { return 'start'; }

    public function getMethod1 () { return '1'; }

    public function getMethod2 ($a, $b) { return $a . $b; }

    public function getMethod3 () { return '3'; }

    public function postMethod4 () { return '4'; }

    public function postMethod5 () { return '5'; }

}
```

Notice something different? Each method is prefixed with either **get** or **post**. This is a must, and it denotes the HTTP verb you want to use for that method. So, in normal route style method4 and method5 will be called by `Route::post`, while the others will be using `Route::get`.

Now, we will call this Controller in routes.php with this syntax.

```
Route::controller('many', 'ManyController');
```

That's it. When you put "many" in URL, it will call ManyController and refer directly to `getIndex` method. If you don't have `getIndex` method, it will simply throw an exception. To call other methods, you only need to call "many/method_name" in the URL. So if you call "many/method1", it will call `method1`.

For a method with input parameters (`method2` in example above), just put the values after the method name in URL, separated by "/" character. So for `method2`, you will call it in URL as "many/method2/param_value1/param_value2" and so on.

Missing Method

Sometimes when you called methods in a Controller, user might put wrong name in the URL. When it produced error, Laravel will throw exception, which is useful for developers but **extremely mortifying** for users. In order to prevent this behavior we can put a "catch-all" method in that controller. The syntax is always the same, the only thing different is the content of the method.

```
public function missingMethod ( $parameters )
{
    //code containing what the application should do in this situation
}
```

This is particularly useful for RESTful Controller, because we can not use routing to prevent user from getting to undefined method name in the URL.

$parameters is a variable which contains the name of undefined method which the user tried to call, and its input parameters. This is useful to create dynamic message to user, for example by saying that this particular URL is not found in the website.

IV
Database Processing

While PHP already provided decent database API, using database processing utilities provided by Laravel will make it scalable and easier, because Laravel's utilities abstracted the query process for various database engine. In other words, we can create one code for virtually all database engine supported by Laravel. As of version 4, Laravel has support for MySQL, SQLite, Postgre SQL, and SQL Server.

Database processing is a necessity in web application, and we will soon see how Laravel handles it gracefully.

Configuring your Engine

Before we even start querying, we need to set up our database engine. In **app/config/database.php**, you will find a lot of options to change depending on which engine you want to use.

One of the most important parts is this line

```
'default' => 'mysql',
```

This sets our application to use MySQL engine by default. You can actually use multiple database engines using the provided library, but for now we will use this default database. You may change it to the database engine you're using at the moment. The choices are **mysql** (MySQL), **pgsql** (PostgreSQL), **sqlite** (SQLite), and **sqlsrv** (SQL Server).

Next you only need to set your preferred database settings in **connections** array. You can see for yourself in **database.php** what parameters you would need to set. This will depends on what database engine you use and the policy of your database server.

Querying Database

At the heart of all relational database processing is SQL. Laravel provided an interface to use SQL syntax to manipulate our database. In the most generic sense, you will use this syntax to do SQL query.

```
DB::statement( sql_query );
```

With `sql_query` being the string containing your query. It will return a Boolean value **true** if the statement succeeded and **false** otherwise. As we can see, although we can do most SQL operation with this, we still need more specialized capabilities for some SQL task. I will explain the syntax using examples.

Select Statement

```
DB::select ( "SELECT * FROM phones WHERE id = ?", array( 10 ) );
```

This syntax will **return an array of rows**. Notice that rather than supplying `id` with direct number, we put an "?" in its place. This is a **prepared statement**, and using this ensures that user will not put a hole in security. The content of "?" will be replaced with the array in the second parameter. The first value inside the array will replace the first "?", and so on. Other operations also use this style of statement, which help in ensuring Laravel security.

An example for fetching all rows and taking the column values is like this.

```
$data = DB::select ( "SELECT * FROM phones" );

foreach($data as $row){
    echo '<h2>' . $row->name  .'</h2>';
}
```

To take the value in a column, we use `$var->col_name` syntax.

Insert Statement

```
DB::insert ( "INSERT INTO phones VALUES ( ?,? )", array( 1, "Nokia") );
```

Insert is straightforward. It will return **true** if succeeded, and **false** otherwise.

```
DB::insert (
   "INSERT INTO phones (name) VALUES ( ? ), ( ? ), ( ? )",
   array("Samsung", "Sony Ericsson", "Motorola")
);
```

This is how we do multiple insert using MySQL. It is not recommended because different database engine might not support this syntax.

Update Statement

```
DB::update (
   "UPDATE phones SET name = ? WHERE id = ?",
   array( 'Apple', 6 )
);
```

Update functions the same as insert. However, update (and delete) returned not true or false, but the **number of rows affected** by the statement. For example, if you update the entire rows of phones table, it will give 7 as its return value.

Delete Statement

```
DB::update (
   "DELETE FROM phones WHERE id = ?",
   array( 5 )
);
```

Like update, delete will return the **number of rows affected** by the statement.

Transactions

Sometimes, we need to do more than one statement at once, but we need to make sure that if one of them failed, the others should fail as well. We can do this using **transaction**. The syntax is as follows.

```
DB::transaction(function()
{
    //SQL statements
});
```

You can put as many SQL statements as you like inside a transaction. If one of them failed, the others will be rolled back. This is useful if you are doing “all-or-nothing” transactions.

Query Builder

Rather than manually create the query string, it is better for us to use Laravel’s query-building capabilities. Query builder automatically translate your query object into working SQL query, minimizing the error possible from mistyping your query string.

The first and foremost thing in using query builder is to choose which table you want to manipulate.

```
DB::table( table_name )
```

After that, we just need to call the methods equivalent to SQL query syntax using **->** notation chains. Here’s some example and its SQL equivalent. I will be using MySQL syntax here.

Query Builder	SQL syntax
SELECT Statements	
`DB::table('phones')->get()`	`SELECT * FROM phones`

`DB::table('phones')->where('id', 2)->get()`	`SELECT * FROM phones WHERE id = 2`
`DB::table('phones')->select('name')->get()`	`SELECT name FROM phones`
`DB::table('phones')->first()`	`SELECT * FROM phones LIMIT 0, 1`
`DB::table('phones')->skip(2)->take(3)->get()`	`SELECT * FROM phones LIMIT 2, 3`
INSERT Statements	

`DB::table('phones')->insert(`
`  array('id' => 1, 'name' => 'BenQ')`
`);`
	`INSERT INTO phones`
	`VALUES`
	`(1, 'BenQ')`

`DB::table('phones')->insertGetId(`
`  array('name' => 'BenQ')`
`);`
	`INSERT INTO phones (name)`
	`VALUES`
	`('BenQ')`

`DB::table('phones')->insertGetId(array(`
`  array('name' => 'HTC'),`
`  array('name' => 'Huawei'),`
`  array('name' => 'Cross')`
`  )`
`);`
	`INSERT INTO phones (name)`
	`VALUES`
	`('HTC'),`
	`('Huawei'),`
	`('Cross')`

UPDATE Statements	
`DB::table('phones')->update('name', 'Xbit')`	`UPDATE phones SET name = 'Xbit'`

`DB::table('phones')`
`  ->where('id', 1)`
`  ->update('name', 'Xbit')`
	`UPDATE phones`
	`SET name = 'Xbit'`
	`WHERE id = 1`

DELETE Statements	
`DB::table('phones')->delete()`	`DELETE FROM phones`

`DB::table('phones')`
`  ->where('id', 1)`
`  ->delete()`
	`DELETE FROM phones`
	`WHERE id = 1`

Advanced Chaining

The examples above used only simple SQL conditions in their syntax. In real world application, IN and EXISTS conditions and other comparisons would be necessary.

1. **where()**

   `where()` is straightforward. Where would take either 2 or 3 parameters. When it takes 2, it means "=" comparison. 3 parameters mean that `where()` will do comparisons based on what the 2nd parameter is. If you put ">" inside the 2nd parameter then you will do greater than comparison. Chaining `where()` will gives you equivalence to AND operation.

   Query builder:

```
DB::table('phones')
    ->where('id', '<', 5')
    ->where('name', 'like', '%x%')
    ->where('model', 'clamshell')
    ->get();
```

   SQL syntax:

```
SELECT * FROM phones
WHERE id < 5
AND name LIKE '%x%'
AND model = 'clamshell'
```

2. **orWhere()**

   `orWhere()` is similar to OR operations, and should only be chained after at least one `where()`. It takes an anonymous function as its parameter, with the function taking one variable (usually called `$query`). This variable is the same as our `DB::table()`, and will be used to further chain `where()` methods.

   Query builder:

```
DB::table('phones')
    ->where('name', 'like', '%x%')
    ->orWhere(function($query){
                $query
                      ->where('id', '<', 5)
                      ->where('model', 'clamshell')
          })
    ->get();
```

SQL syntax:

```
SELECT * FROM phones
WHERE name LIKE '%x%'
OR(
id < 5
AND model = 'clamshell'
)
```

3. **whereIn()** and **whereNotIn()**

   You can also use IN with query builder. `whereIn()` takes 2 parameters, the column name being compared and either an array containing the values or an anonymous function containing a subquery. `whereNotIn()` is the same, except it is the SQL equivalent of NOT IN.

   Query Builder:

```
DB::table('phones')
    ->whereIn('id', function($query){
                $query->from('transactions')
                       ->where('transactionDate', '2013-09-28')
                       ->select('id');
          })
    ->select('name')
    ->get();
```

SQL Syntax:

```
SELECT name FROM phones
WHERE id IN(
    SELECT id FROM transactions
    WHERE transactionDate = '2013-09-28'
)
```

4. **whereExists()** and **whereNotExists()**

   The same as `whereIn()` and `whereNotIn()`, but this time it always take anonymous function containing the subquery.

   Query builder:

```
DB::table('phones')
    ->whereExists(function($query){
                $query->from('transactions')
                       ->where('transactionDate', '2013-09-28')
                       ->whereRaw('transactions.phoneId = phones.id')
                       ->select(DB::raw(1));
          })
    ->select('name')
    ->get();
```

   SQL syntax:

```
SELECT name FROM phones
WHERE EXISTS(
    SELECT 1 FROM transactions
    WHERE transactionDate = '2013-09-28'
    AND transactions.phoneId = phone.id
)
```

5. **orderBy()**

   Similar to ORDER BY in SQL, this is used to order the result. The first parameter is the ordered column, and the second is how the column will be ordered, ascending (asc) or descending (desc).

   Query Builder:

```
DB::table('phones')
    ->orderBy('name', 'desc')
    ->get();
```

SQL Syntax:

```
SELECT * FROM phones ORDER BY name DESC
```

6. groupBy()

Similar to GROUP BY, this is used if you have aggregate inside your query.

Query Builder:

```
DB::table('phones')
    ->groupBy('name', 'model')
    ->get();
```

SQL syntax:

```
SELECT * FROM phones
GROUP BY (name, model)
```

A complex query builder chains could be something like this.

```
DB::table('phones')
    ->where('name', 'like', 'a%')
    ->orWhere(function($query){
          $query->where('id', '<', 10)
                ->where('name', 'not like', '%n')
          })
    ->orderBy('name', 'asc')
    ->groupBy('id')
    ->select('id', 'name', 'model')
    ->get()
```

Which is equivalent to this SQL syntax.

```
SELECT id, name, model FROM phones
WHERE name LIKE 'a%'
OR (
    id < 10 AND name NOT LIKE 'n%'
)
GROUP BY id
ORDER BY name ASC
```

Joining Tables

Joining tables is simple using `join()` method.

```
DB::table('phones')
    ->join('transactions', 'transactions.phoneId', '=', 'phones.id')
    ->select(phones.name, transactions.date)
    ->get();
```

The equivalent SQL syntax is like this.

```
SELECT phones.name, transactions.date
FROM phones JOIN transactions ON transactions.phoneId = phones.id
```

If you want to do left join, simply use `leftJoin()` method.

Aggregate Functions

Query Builder	SQL syntax
`DB::table('phones')->count();`	`SELECT COUNT(*) FROM phones`
`DB::table('phones')->max('price');`	`SELECT MAX(price) FROM phones`
`DB::table('phones')->min('price');`	`SELECT MIN(price) FROM phones`

`DB::table('phones')->avg('price');`	`SELECT AVG(price) FROM phones`
`DB::table('phones')->sum('price');`	`SELECT SUM(price) FROM phones`

Increments and Decrements

We can increment and decrement numeric values in table columns (similar to updating a column).

```
DB::table('phones')->increment('price');

DB::table('phones')->increment('price', 50);
```

Without 2nd parameter, increment only does +1. The line below it will add price by +50.

```
DB::table('phones')->decrement('price');

DB::table('phones')->decrement('price', 50);
```

Decrement works the same way.

```
DB::table('phones')->increment('price', 3, array('culprit' => 'admin'));
```

You can add additional column to change. In this example, after price is incremented by +3, the value of culprit column will be updated into “admin”.

Truncate Table

The syntax for truncating a table is really simple. Make sure that you are sure to do it because all data in the table will be deleted.

```
DB::table( table_name )->truncate();
```

Schema

Database Schema is the description of its structure. In Laravel, Schema class is the one used to create and maintain a database. In other words, its DDL.

Creating Table

The most fundamental method is the `create()` method, which creates a table based on `Blueprint` object put into its parameter. A `Blueprint` is Laravel's name for the table definition, which includes column names, data types, and indexes.

```
Schema::create ( table_name, function($table){
   //tabel definitions
});
```

For example, we will make the "phones" table.

```
Schema::create('phones', function($table){
   $table->increments('id');
   $table->string('name', 100);
   $table->string('model', 100);
   $table->bigInteger('price');
});
```

This code is equivalent to this SQL statement.

```
CREATE TABLE phones(
   id INT AUTO_INCREMENT PRIMARY KEY,
   name VARCHAR(100),
   model VARCHAR(100),
   price BIGINT
)
```

As we can see, creating table using Schema is closer to Object Oriented programming style.

Here is a complete list of column types.

Command	Description
`$table->increments('id');`	Incrementing ID to the table (primary key).
`$table->bigIncrements('id');`	Incrementing ID using a "big integer" equivalent.
`$table->string('email');`	VARCHAR equivalent column
`$table->string('name', 100);`	VARCHAR equivalent with a length
`$table->integer('votes');`	INTEGER equivalent to the table
`$table->bigInteger('votes');`	BIGINT equivalent to the table
`$table->smallInteger('votes');`	SMALLINT equivalent to the table
`$table->float('amount');`	FLOAT equivalent to the table
`$table->decimal('amount', 5, 2);`	DECIMAL equivalent with a precision and scale
`$table->boolean('confirmed');`	BOOLEAN equivalent to the table
`$table->date('created_at');`	DATE equivalent to the table
`$table->dateTime('created_at');`	DATETIME equivalent to the table
`$table->time('sunrise');`	TIME equivalent to the table
`$table->timestamp('added_on');`	TIMESTAMP equivalent to the table
`$table->timestamps();`	Adds created_at and updated_atcolumns
`$table->softDeletes();`	Adds deleted_at column for soft deletes
`$table->text('description');`	TEXT equivalent to the table
`$table->binary('data');`	BLOB equivalent to the table
`$table->enum('choices', array('foo', 'bar'));`	ENUM equivalent to the table
`->nullable()`	Designate that the column allows NULL values
`->default($value)`	Declare a default value for a column
`->unsigned()`	Set INTEGER to UNSIGNED

Drop Table

Dropping a table is achieved with a simple method.

```
Schema::drop( table_name );
```

If you want to check before dropping whether the table exists or not, you can use another method.

```
Schema::dropIfExists( table_name );
```

Alter Table

Altering a table definition is easier using Schema. We do this using `table()` method. The first parameter is the table name, and the second is similar to `create()` method in which we put an anonymous function containing the table definition.

```
Schema::table('phones', function($table){
   //table definition
});
```

Inside this method, we can just add more columns just like when we are creating a new table.

If we want to rename a column:

```
$table->renameColumn ( old_name, new_name );
```

If we want to delete a column:

```
$table->deleteColumn ( col );
```

Or if we want to delete multiple columns at once:

```
$table->deleteColumn ( col_1, col_2, col_3, ...);
```

Adding Keys

We can put foreign keys on our table definitions using `foreign()` method and its chains.

```
$table->foreign('phoneId')
    ->references('id')
    ->on('phones')
    ->onDelete('cascade')
    ->onUpdate('cascade');
```

Which is equivalent to the following SQL syntax.

```
FOREIGN KEY (phoneId) REFERENCES phones(id)
ON DELETE CASCADE ON UPDATE CASCADE
```

As ON DELETE and ON UPDATE are optional, we can also omit them from our table definition if necessary.

Other than foreign key, there are other keys supported by Schema. One of them is the most important one, the primary key. Here is the complete list.

Command	Description
`$table->primary('id');`	Adding a primary key
`$table->primary(array('first', 'last'));`	Adding composite keys
`$table->unique('email');`	Adding a unique index
`$table->index('state');`	Adding a basic index

Once we have established the keys, we can drop them using the following methods.

```
$table->dropPrimary( column_name );

$table->dropForeign( column_name );

$table->dropUnique( column_name );

$table->dropIndex( column_name );
```

Migration

Migration allows for **version control** in you database structure. This feature allows us to update database structures to the latest design or rollback the newest structures to the older one if it fails to meet our demand. Using Migration makes teamwork easier because development team can share and test new database design each time changes are needed.

Migration utilizes the **command line interface** for Laravel, the **Artisan**. It is a PHP script located in your Laravel folder. To call Artisan, simply call your terminal and move to your Laravel's uppermost folder and then type the following command.

```
php artisan
```

Artisan has a lot of capabilities. If you simply typed the command above, you will see a list of Artisan commands. We will focus on **migrate** command.

Creating Migration

A Migration is a file containing the database structures. First and foremost, we need to create that file. Laravel provided the template for us. To make it, we use Artisan like this.

```
php artisan migrate:make migration_file_name
```

With `migration_file_name` is the name of the file you will make. Make sure that the name is something meaningful because you can only use one name for each migration file. Having the same name for two or more file will produce errors.

The result of using this command is a php file located in **app/database/migrations** folder. The filename will include the creation date, random id number, and the name you specified earlier. The content is something like this. In this example the `migration_file_name` is "CreateBirdsTable".

```
use Illuminate\Database\Migrations\Migration;

class CreateBirdsTable extends Migration {

   /**
    * Run the migrations.
    *
    * @return void
    */
   public function up()
   {
         //
   }

   /**
    * Reverse the migrations.
    *
    * @return void
    */
   public function down()
   {
         //
   }

}
```

Migrations have two main methods: **up** and **down.**

1. **Up**
   This method is used when migration is called. In this method we usually create new tables and do similar database operations.
2. **Down**

This method is used when a newer version of migration is called and the current version needs to be changed. It is also called when we do rollback to previous version. Usually the content is the opposite of **up** method. e.g. if we create a new table in **up**, we will drop it in **down**.

Running Migration

After we have created a new Migration version, we just need to run it to apply the changes. To run Migration, simply call Artisan again in command line. The command is as follows.

php artisan migrate

The changes will be applied immediately. Now if you want to change your database Schema, you only need to create new Migration and then run it again. Laravel will take care of the versioning.

Rollback

If you feel that the current Schema fails to meet our expectation, we can simply return to the previous Schema using rollback function in Artisan. The command is as follows.

```
php artisan migrate:rollback
```

Or you can also wipe the database clean with this command.

```
php artisan migrate:reset
```

We can also rollback the database and then run the newest Migration again, usually to clean the data while maintaining the Schema. The command is as follows.

```
php artisan migrate:refresh
```

Model

Creating Model

We finally come to the concept of ORM (Object-Relational Mapping). The goal is to allow us to manipulate SQL database, which use relational concept, using objects. Thus, rather than making SQL queries, we simply create a new object and save it to the table. Laravel uses Eloquent ORM as its underlying mechanism.

Model is located in **app/models**. Creating a model is simply creating a new PHP class. This is the barebone code for a model.

```
class singular_table_name extends Eloquent {
}
```

That's all. Laravel will connect your model with the appropriate table based on your `singular_table_name`. Remember that in previous examples, table names are always plural: thus, if your table is **users**, your `singular_table_name` would be **User**. Take care that Laravel does this automatically. When you save the file, usually it will be the same name as your class name, so in this case it will be **User.php**.

However, if you want the model to connect to different table, you can specify the table manually with `$table` attribute.

```
class singular_table_name extends Eloquent {
    protected $table = table_name;
}
```

With `table_name` is a string containing your desired table's name.

Querying with Model

A Model is the representation of database table. Therefore, we can do various manipulation of the table through our Model. The fundamental thing is to get data from the database, equivalent to a SELECT statement.

To get all data from the table, we can do it like this example.

```
$birds = Bird::all();

//print all the data
foreach($birds as $bird){
    echo "Name: $bird->name<br/>";
    echo "Species: $bird->species<br/><br/>";
}
```

`Bird::all()` is a method to get all rows from **birds** table. All Model have this method, so if your table is **users**, the method would be `User::all()`. After we put it into `$birds` variable, we can access it using foreach. Each item inside `$birds` is a single object of **Bird** class. We can get the value of each column by calling the column name using `->` notation, as shown in the example.

If we want to get only a specific row using the primary key, we can do it like this.

```
$birds = Bird::find(1);
```

`find()` method will search for a row which primary key matched the parameter value. Then you can access it as a single object.

Finally, if you want to create a complex SQL SELECT statement, you can use similar syntax with Query Builder from previous examples. For example, we can make a complex search like this.

```
$birds = Bird::where('name', 'like', '%j%')
                ->select('name')
                ->get();

foreach($birds as $bird){
    echo "Name: $bird->name<br/>";
}
```

Inserting Model

Inserting, updating, and deleting with Model is abstracted such that interacting with the database is similar to creating a new object.

For example, inserting a new row will look like this.

```
$bird = new Bird;

$bird->name = 'Jack';
$bird->species = 'American Bald Eagle';

$bird->save();
```

From the code above, we can see that first thing to do would be to initialize the object with `new` keyword. And then, we assign values to the object's attribute using `->` notation. The attributes is named exactly as the name of your table row. Finally, we call `save()` method to insert the data permanently.

Updating a Model

When we insert a new Model to database, we create the Model from scratch using `new` keyword. However, when we want to update, we do it to the **existing** Model in the database.

First we would need to find the row which we want to update.

```
$birds = Bird::find(1);
```

Then, we just have to change its content as if we're inserting a new row. Unlike when inserting a row, we don't have to specify all column values, just the ones we want to update.

```
$birds->name = 'Jack Version 2';

$birds->save();
```

Deleting a Model

Finally, to delete a Model we just need to find the rows we want to delete, and then call `delete()` method.

```
$birds = Bird::find(2);

$birds->delete();
```

References

Laravel Documentation (http://laravel.com/docs)

Code Bright by Dayle Rees (http://codebright.daylerees.com/)