CORE360 IT SERVICES

PHP MVC Framework using Laravel

Implementing a Cleaner, Scalable, Robust and Secure Application Design Pattern using Modular Design Logic.

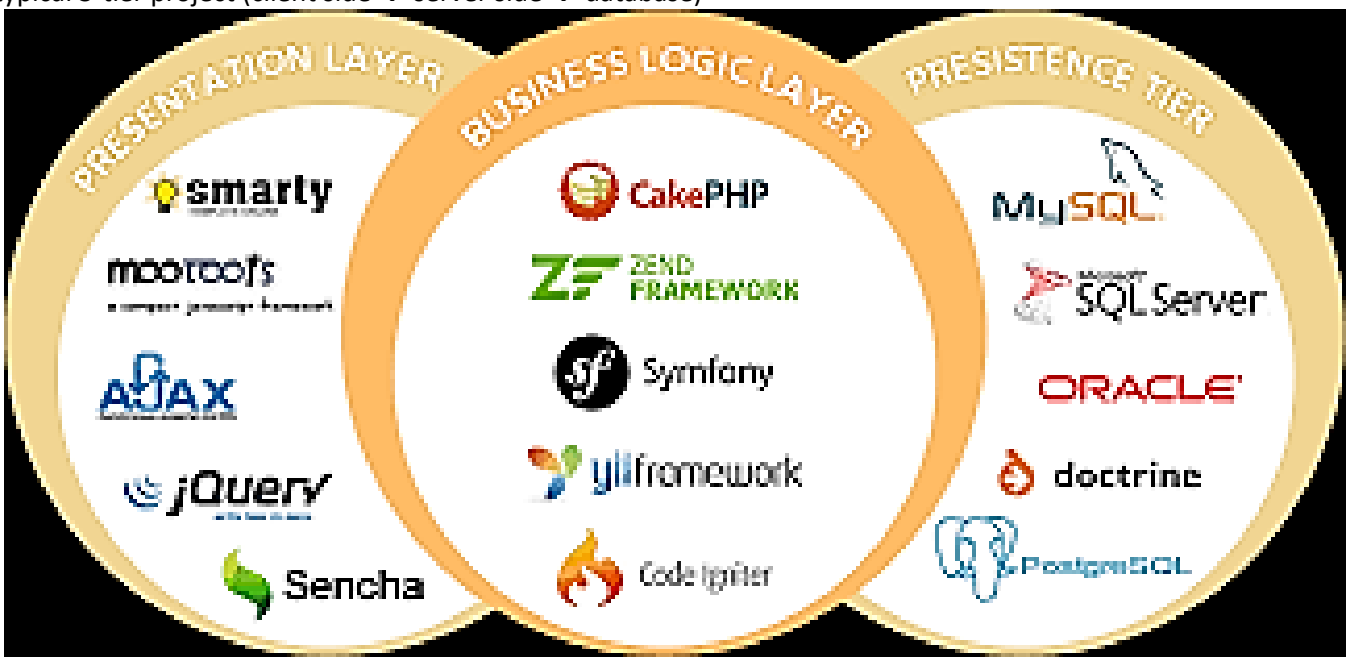
Core360
Technology | Solutions | Training

CORE360 IT SERVICES
TECHNOLOGY | BUSINESS SOLUTIONS | TRAINING

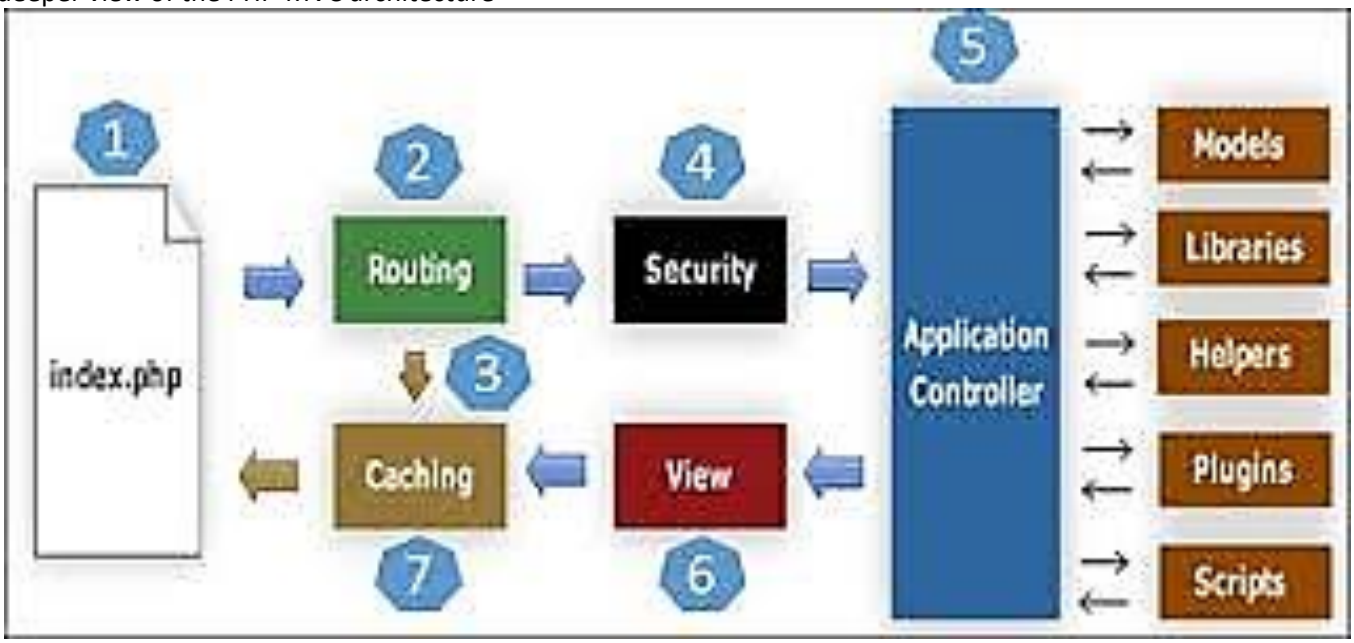
1-A F.Jacinto Street Galguierra Compound
General T. De Leon, Marulas, Valenzuela City
Mobile(s): 09228720135 | 09954334855
Email: johnreygoh@gmail.com
Facebook: https://www.facebook.com/johnrey.goh

[THE PHP-MVC DESIGN WORKFLOW]

*A typical 3-tier project (client side → server side → database)



*A deeper view of the PHP-MVC architecture



The Basics:

1. Understand PHP frameworks
   (discussed)

2. Understand Model-View-Controller

   •Model: A model is a representation of a database row. Models should be the only part of your application that interacts directly with the database

   •View: A view is the user facing HTML and server side code. There should be no business logic or database interactions within a view. Views are dumb and should just render and display data to the user

   •Controller: A controller "controls" all the data that is either requested by the user or data that is retrieved from the database via a model. All business logic should be within the controllers

Popular FRONT-END / PRESENTATION LAYER languages and technologies:

1. HTML / XHTML -structure and universal document type definition.
2. CSS -layout and designing.
3. Javascript -front-end/client-side script (Core Javascript, AngularJS, JQuery, AJAX, etc..).

Popular PHP-MVC Frameworks:

1. Zend Framework
2. Laravel
3. Yii2
4. Symfony
5. CakePHP
6. CodeIgniter

   And more!

Popular Databases (Back-End)

1. Persistent Data for Mobile (Android / iOS)
   a. SQLite
   b. CoreData
   c. Realm

      And more…

2. Database Technologies
   a. MS SQL
   b. MySQL
   c. MongoDB
   d. PostgreSQL
   e. Oracle
   f. Etc..

Contents

Installing Laravel

1. wamp->php->php extensions(php_curl,php_openssl,php_sockets)
2. wamp->apache->apache modules(rewrite_module,ssl_module)
3. c:/wamp/bin/php/php5.x/php.ini -> enable openssl
4. As of June 2018, WAMP does not carry openssl functions (XAMPP does), I have found openssl here:

   http://slproweb.com/products/Win32OpenSSL.html
   *I have tested version 1.0.2 both for Windows 7 and Windows 10

5. Laravel 10 needs PHP v8.1 and above
6. install composer for windows
7. run command in your /www or /htdocs

   *to test if composer is good:

```
composer
```

   *to download composer vendor files, type in command:

```
composer create-project --prefer-dist laravel/laravel <proj-name> <version>
```

   *example:

```
composer create-project --prefer-dist laravel/laravel project-name "10.*"
```

8. Or, you may copy the existing laravel skeleton application from another project
9. You may also use the built-in development server

```
php artisan serve
```

   This will start a local development server at http://localhost:8000.

Creating a home page

[resources/views/home.php]

```
<html>
<head><title>MY HOME PAGE</title></head>
<body>
This is your home page
</body></html>
```

Setting your default project home page

[app/Http/routes.php]

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

or

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

Creating a Blade template for your pages

[resources/views/mylayout.blade.php]

```
<!DOCTYPE html>
<head>
<title>Laravel 7 Employee Records</title>
</head>
<body>
<div id="container">

<h1>My Laravel 7 practice</h1>
<div>
@yield('body')
</div>

<p>
<a href="#">Visit my page</a>
</p>

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

*note: the '@yield(?)' is used to hold the '@section(?)' on other pages that will be extending our layout. These two tags should have the same parameters.

Creating other views pages that uses the blade template

*Edit your views page name
[resources/views/home.blade.php]

```
@extends('home')
@section('body')
This is your home page
@stop
```

*create the other pages for the activity
[resources/views/htmlform.blade.php]

```
@extends('home')
@section('body')
This is your htmlform page
@stop
```

[resources/views/activity.blade.php]

```
@extends('home')
@section('body')
This is your activity page
@stop
```

```
 [resources/views/htmlprocess.blade.php]
@extends('home')
@section('body')
This is your htmlprocess page
@stop
```

Create Routes for your pages

*edit
[app/Http/Routes.php]

```
Route::get('/',function(){return view('home');});
Route::get('/htmlform',function(){return view('htmlform');});
Route::get('/activity',function(){return view('activity');});
```

*try calling the views pages from the URL

Ex:
localhost/projectfolder/publicfolder/htmlform

Calling Views from a Controller class

*run cmd in your project

```
/>php artisan make:controller TestController
```

*this will create a controller class

```
[app/http/controllers/testController]
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;
use App\Http\Requests;
use View;

class testController extends Controller
{

	public function index(){
	return "this is your controller's index";
	}

	public function htmlform(){
	return View::make('htmlform');
	}

	public function activity(){
	return View::make('activity');
	}

}
```

*you have to route the function calls to your controller in
[app/routes/web.php]

```
use App\Http\Controllers\TestController;
Route::get('/', [TestController::class, 'index']);
```

or

```
Route::get('/home','App\Http\Controller\testController@index')
Route::get('/','App\Http\Controller\testController@index')
Route::get('/htmlform','App\Http\Controller\testController@htmlform')
Route::get('/activity','App\Http\Controller\testController@activity')
```

Getting and submitting form values

Anytime you define a HTML form in your application, you should include a hidden CSRF token field in the form so that the CSRF protection middleware can validate the request. You may use the csrf_field helper to generate the token field:

```
<form method="POST" action="/profile">
   {{ csrf_field() }}
   ...
</form>
```

The VerifyCsrfToken middleware, which is included in the web middleware group, will automatically verify that the token in the request input matches the token stored in the session.

[resources/views/htmlform.blade.php]

```
@extends('layout')
@section('body')

<h1>Contact Us.</h1>
<p>Please contact us by sending a message using the form below:</p>

<form method="post" action="htmlprocess">
Subject <input type="text" name="text1"><br />
Message: <br />
<textarea name="area1" cols="30" rows="4"></textarea><br />
<input type="submit" name="btn1" value="submit">
{{ csrf_field() }}
</form>

@stop
```

*passing value to views via route

```
use Illuminate\Http\Request;

Route::post('/htmlprocess',function(Request $request){
        $data=$request->all();
        return View::make('htmlprocess')->with('data',$data);

});
```

*passing values from view->controller->view

//add to route:

```
Route::post('/htmlprocess','testController@htmlprocess');
```

//on controller

```
use Illuminate\Http\Request;
use View;

public function htmlprocess(Request $request)
  {
        $data=$request->all();
        return View::make('htmlprocess')->with('data',$data);
  }
```

[resources/views/htmlprocess.blade.php]

```
@extends('layout')
@section('body')

<h1>MESSAGE DETAILS:</h1>
Subject: {!!$data['text1']!!} <br />
Message Content: {!!$data['area1']!!} <br /><br />

<a href="htmlform">back to form</a>

@stop
```

Using Laravel's Request and Response

Accessing the Request

To obtain an instance of the current HTTP request via dependency injection, you should type-hint the Illuminate\Http\Request class on your controller method. The incoming request instance will automatically be injected by the service container:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class UserController extends Controller
{
   public function store(Request $request)
   {
      $name = $request->input('name');
   }
}
```

Route Parameters

If your controller method is also expecting input from a route parameter you should list your route parameters after your other dependencies. For example, if your route is defined like so:

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

You may still type-hint the Illuminate\Http\Request and access your route parameter id by defining your controller method as follows:

```
<?php
namespace App\Http\Controllers;
use Illuminate\Http\Request;

class UserController extends Controller
{
   public function update(Request $request, $id)
   {
      //
   }
}
```

Accessing the Request Via Route Closures

You may also type-hint the Illuminate\Http\Request class on a route Closure. The service container will automatically inject the incoming request into the Closure when it is executed:

```
use Illuminate\Http\Request;

Route::get('/', function (Request $request) {
   //
});
```

Request Path & Method

The Illuminate\Http\Request instance provides a variety of methods for examining the HTTP request for your application and extends the Symfony\Component\HttpFoundation\Request class. We will discuss a few of the most important methods below.

Retrieving the Request Path

The path method returns the request's path information. So, if the incoming request is targeted at http://domain.com/foo/bar, the path method will return foo/bar:

```
$uri = $request->path();
```

Retrieving the Request URL

To retrieve the full URL for the incoming request you may use the url or fullUrl methods. The url method will return the URL without the query string, while the fullUrl method includes the query string:

```
// Without Query String...
$url = $request->url();

// With Query String...
$url = $request->fullUrl();
```

Input Trimming & Normalization

By default, Laravel includes the TrimStrings and ConvertEmptyStringsToNull middleware in your application's global middleware stack. These middlewares are listed in the stack by the App\Http\Kernel class. These middlewares will automatically trim all incoming string fields on the request, as well as convert any empty string fields to null. This allows you to not have to worry about these normalization concerns in your routes and controllers.

If you would like to disable this behavior, you may remove the two middleware from your application's middleware stack by removing them from the $middleware property of your App\Http\Kernel class.

Retrieving Input

Retrieving All Input Data

You may also retrieve all of the input data as an array using the all method:

```
$input = $request->all();
```

Retrieving an Input Value

Using a few simple methods, you may access all of the user input from your Illuminate\Http\Request instance without worrying about which HTTP verb was used for the request. Regardless of the HTTP verb, the input method may be used to retrieve user input:

```
$name = $request->input('name');
```

You may pass a default value as the second argument to the input method. This value will be returned if the requested input value is not present on the request:

```
$name = $request->input('name', 'Sally');
```

When working with forms that contain array inputs, use "dot" notation to access the arrays:

```
$name = $request->input('products.0.name');
$names = $request->input('products.*.name');
```

Retrieving A Portion of The Input Data

If you need to retrieve a subset of the input data, you may use the only and except methods. Both of these methods accept a single array or a dynamic list of arguments:

```
$input = $request->only(['username', 'password']);
$input = $request->only('username', 'password');
$input = $request->except(['credit_card']);
$input = $request->except('credit_card');
```

Determining If an Input Value Is Present

You should use the has method to determine if a value is present on the request. The has method returns true if the value is present and is not an empty string:

```
if ($request->has('name')) {
    //
}
```

Creating Responses

Strings & Arrays

All routes and controllers should return a response to be sent back to the user's browser. Laravel provides several different ways to return responses. The most basic response is simply returning a string from a route or controller. The framework will automatically convert the string into a full HTTP response:

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

In addition to returning strings from your routes and controllers, you may also return arrays. The framework will automatically convert the array into a JSON response:

```
Route::get('/', function () {
   return [1, 2, 3];
});
```

Redirects

Redirect responses are instances of the Illuminate\Http\RedirectResponse class, and contain the proper headers needed to redirect the user to another URL. There are several ways to generate a RedirectResponse instance. The simplest method is to use the global redirect helper:

```
Route::get('dashboard', function () {
   return redirect('home/dashboard');
});
```

Using Sessions in Laravel

Configuration

The session configuration file is stored at config/session.php. Be sure to review the options available to you in this file. By default, Laravel is configured to use the file session driver, which will work well for many applications.

Using the Session

Retrieving Data

There are two primary ways of working with session data in Laravel: the global session helper and via a Request instance. First, let's look at accessing the session via a Request instance, which can be type-hinted on a controller method. Remember, controller method dependencies are automatically injected via the Laravel service container:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserController extends Controller
{
   public function show(Request $request, $id)
   {
```

```
        $value = $request->session()->get('key');
    }
}
```

When you retrieve a value from the session, you may also pass a default value as the second argument to the get method. This default value will be returned if the specified key does not exist in the session. If you pass a Closure as the default value to the get method and the requested key does not exist, the Closure will be executed and its result returned:

```
$value = $request->session()->get('key', 'default');

$value = $request->session()->get('key', function () {
    return 'default';
});
```

The Global Session Helper

You may also use the global session PHP function to retrieve and store data in the session. When the session helper is called with a single, string argument, it will return the value of that session key. When the helper is called with an array of key / value pairs, those values will be stored in the session:

```
Route::get('home', function () {
    // Retrieve a piece of data from the session...
    $value = session('key');

    // Specifying a default value...
    $value = session('key', 'default');

    // Store a piece of data in the session...
    session(['key' => 'value']);
});
```

Retrieving All Session Data

If you would like to retrieve all the data in the session, you may use the all method:

```
$data = $request->session()->all();
```

Determining If an Item Exists in The Session

To determine if a value is present in the session, you may use the has method. The has method returns true if the value is present and is not null:

```
if ($request->session()->has('users')) {
    //
}
```

To determine if a value is present in the session, even if its value is null, you may use the exists method. The exists method returns true if the value is present:

```
if ($request->session()->exists('users')) {
    //
}
```

Storing Data

To store data in the session, you will typically use the put method or the session helper:

```
// Via a request instance...
$request->session()->put('key', 'value');

// Via the global helper...
session(['key' => 'value']);
```

Retrieving an Item

The pull method will retrieve and delete an item from the session in a single statement:

```
$value = $request->session()->pull('key', 'default');
```

Deleting Data

The forget method will remove a piece of data from the session. If you would like to remove all data from the session, you may use the flush method:

```
$request->session()->forget('key');

$request->session()->flush();
```

File System Management in Laravel

The filesystem configuration file is located at config/filesystems.php. Within this file you may configure all of your "disks". Each disk represents a particular storage driver and storage location. Example configurations for each supported driver are included in the configuration file. So, simply modify the configuration to reflect your storage preferences and credentials.

The Public Disk

The public disk is intended for files that are going to be publicly accessible. By default, the public disk uses the local driver and stores these files in storage/app/public. To make them accessible from the web, you should create a symbolic link from public/storage to storage/app/public. This convention will keep your publicly accessible files in one directory that can be easily shared across deployments when using zero down-time deployment systems like Envoyer.

To create the symbolic link, you may use the storage:link Artisan command:

```
php artisan storage:link
```

Of course, once a file has been stored and the symbolic link has been created, you can create a URL to the files using the asset helper:

```
echo asset('storage/file.txt');
```

The Local Driver

When using the local driver, all file operations are relative to the root directory defined in your configuration file. By default, this value is set to the storage/app directory. Therefore, the following method would store a file in storage/app/file.txt:

```
Storage::disk('local')->put('file.txt', 'Contents');
```

Retrieving Files

The get method may be used to retrieve the contents of a file. The raw string contents of the file will be returned by the method. Remember, all file paths should be specified relative to the "root" location configured for the disk:

```
$contents = Storage::get('file.jpg');
```

File Metadata

In addition to reading and writing files, Laravel can also provide information about the files themselves. For example, the size method may be used to get the size of the file in bytes:

```
use Illuminate\Support\Facades\Storage;

$size = Storage::size('file1.jpg');
```

The lastModified method returns the UNIX timestamp of the last time the file was modified:

```
$time = Storage::lastModified('file1.jpg');
```

Storing Files

The put method may be used to store raw file contents on a disk. You may also pass a PHP resource to the put method, which will use Flysystem's underlying stream support. Using streams is greatly recommended when dealing with large files:

```
use Illuminate\Support\Facades\Storage;

Storage::put('file.jpg', $contents);
Storage::put('file.jpg', $resource);
```

Prepending & Appending to Files

The prepend and append methods allow you to write to the beginning or end of a file:

```
Storage::prepend('file.log', 'Prepended Text');
Storage::append('file.log', 'Appended Text');
```

File Uploads

In web applications, one of the most common use-cases for storing files is storing user uploaded files such as profile pictures, photos, and documents. Laravel makes it very easy to store uploaded files using the store method on an uploaded file instance. Simply call the store method with the path at which you wish to store the uploaded file:

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

use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class UserAvatarController extends Controller
{
   public function update(Request $request)
   {
      $path = $request->file('avatar')->store('avatars');
      return $path;
   }
}
```

There are a few important things to note about this example. Note that we only specified a directory name, not a file name. By default, the store method will generate a unique ID to serve as the file name. The path to the file will be returned by the store method so you can store the path, including the generated file name, in your database.

You may also call the putFile method on the Storage facade to perform the same file manipulation as the example above:

```
$path = Storage::putFile('avatars', $request->file('avatar'));
```

Specifying A File Name

If you would not like a file name to be automatically assigned to your stored file, you may use the storeAs method, which receives the path, the file name, and the (optional) disk as its arguments:

```
$path = $request->file('avatar')->storeAs(
    'avatars', $request->user()->id
);
```

Of course, you may also use the putFileAs method on the Storage facade, which will perform the same file manipulation as the example above:

```
$path = Storage::putFileAs(
    'avatars', $request->file('avatar'), $request->user()->id
);
```

Deleting Files

The delete method accepts a single filename or an array of files to remove from the disk:

```
use Illuminate\Support\Facades\Storage;

Storage::delete('file.jpg');
Storage::delete(['file1.jpg', 'file2.jpg']);
```

Directories

Get All Files Within A Directory

The files method returns an array of all of the files in a given directory. If you would like to retrieve a list of all files within a given directory including all sub-directories, you may use the allFiles method:

```
use Illuminate\Support\Facades\Storage;

$files = Storage::files($directory);
$files = Storage::allFiles($directory);
```

Get All Sub-Directories Within A Directory

The directories method returns an array of all the directories within a given directory. Additionally, you may use the allDirectories method to get a list of all directories within a given directory and all of its sub-directories:

```
$directories = Storage::directories($directory);

// Recursive...
$directories = Storage::allDirectories($directory);
```

Create A Directory

The makeDirectory method will create the given directory, including any needed sub-directories:

```
Storage::makeDirectory($directory);
```

Delete A Directory

Finally, the deleteDirectory may be used to remove a directory and all of its files:

```
Storage::deleteDirectory($directory);
```

The Eloquent Object Relational Mapper (ORM)

Eloquent Model Conventions

Now, let's look at an example Flight model, which we will use to retrieve and store information from our flights database table:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
   //
}
```

Table Names

Note that we did not tell Eloquent which table to use for our Flight model. By convention, the "snake case", plural name of the class will be used as the table name unless another name is explicitly specified. So, in this case, Eloquent will assume the Flight model stores records in the flights table. You may specify a custom table by defining a table property on your model:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
   protected $table = 'my_flights';
}
```

Primary Keys

Eloquent will also assume that each table has a primary key column named id. You may define a $primaryKey property to override this convention.

In addition, Eloquent assumes that the primary key is an incrementing integer value, which means that by default the primary key will be cast to an int automatically. If you wish to use a non-incrementing or a non-numeric primary key you must set the public $incrementing property on your model to false.

Timestamps

By default, Eloquent expects created_at and updated_at columns to exist on your tables. If you do not wish to have these columns automatically managed by Eloquent, set the $timestamps property on your model to false:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
   public $timestamps = false;
}
```

If you need to customize the format of your timestamps, set the $dateFormat property on your model. This property determines how date attributes are stored in the database, as well as their format when the model is serialized to an array or JSON:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
   protected $dateFormat = 'U';
}
```

If you need to customize the names of the columns used to store the timestamps, you may set the CREATED_AT and UPDATED_AT constants in your model:

```
<?php

class Flight extends Model
{
   const CREATED_AT = 'creation_date';
   const UPDATED_AT = 'last_update';
}
```

Database Connection

By default, all Eloquent models will use the default database connection configured for your application. If you would like to specify a different connection for the model, use the $connection property:

```
<?php

namespace App;
use Illuminate\Database\Eloquent\Model;

class Flight extends Model
{
   protected $connection = 'connection-name';
}
```

Retrieving Models

Once you have created a model and its associated database table, you are ready to start retrieving data from your database. Think of each Eloquent model as a powerful query builder allowing you to fluently query the database table associated with the model. For example:

```
<?php

use App\Flight;

$flights = App\Flight::all();

foreach ($flights as $flight) {
   echo $flight->name;
}
```

Adding Additional Constraints

The Eloquent all method will return all of the results in the model's table. Since each Eloquent model serves as a query builder, you may also add constraints to queries, and then use the get method to retrieve the results:

```
$flights = App\Flight::where('active', 1)
          ->orderBy('name', 'desc')
          ->take(10)
          ->get();
```

Collections

For Eloquent methods like all and get which retrieve multiple results, an instance of Illuminate\Database\Eloquent\Collection will be returned. The Collection class provides a variety of helpful methods for working with your Eloquent results:

```
$flights = $flights->reject(function ($flight) {
   return $flight->cancelled;
});
```

Of course, you may also simply loop over the collection like an array:

```
foreach ($flights as $flight) {
   echo $flight->name;
}
```

Retrieving Single Models / Aggregates

Of course, in addition to retrieving all of the records for a given table, you may also retrieve single records using find or first. Instead of returning a collection of models, these methods return a single model instance:

```
// Retrieve a model by its primary key...
$flight = App\Flight::find(1);

// Retrieve the first model matching the query constraints...
$flight = App\Flight::where('active', 1)->first();
```

You may also call the find method with an array of primary keys, which will return a collection of the matching records:

```
$flights = App\Flight::find([1, 2, 3]);
```

Retrieving Aggregates

You may also use the count, sum, max, and other aggregate methods provided by the query builder. These methods return the appropriate scalar value instead of a full model instance:

```
$count = App\Flight::where('active', 1)->count();

$max = App\Flight::where('active', 1)->max('price');
```

Inserting & Updating Models

Inserts

To create a new record in the database, simply create a new model instance, set attributes on the model, then call the save method:

```
<?php

namespace App\Http\Controllers;
use App\Flight;
use Illuminate\Http\Request;
use App\Http\Controllers\Controller;

class FlightController extends Controller
{
   public function store(Request $request)
   {
      // Validate the request...

      $flight = new Flight;
      $flight->name = $request->name;
      $flight->save();
   }
}
```

In this example, we simply assign the name parameter from the incoming HTTP request to the name attribute of the App\Flight model instance. When we call the save method, a record will be inserted into the database. The created_at and updated_at timestamps will automatically be set when the save method is called, so there is no need to set them manually.

Updates

The save method may also be used to update models that already exist in the database. To update a model, you should retrieve it, set any attributes you wish to update, and then call the save method. Again, the updated_at timestamp will automatically be updated, so there is no need to manually set its value:

```
$flight = App\Flight::find(1);
$flight->name = 'New Flight Name';
$flight->save();
```

Deleting Models

To delete a model, call the delete method on a model instance:

```
$flight = App\Flight::find(1);
$flight->delete();
```

Deleting an Existing Model By Key

In the example above, we are retrieving the model from the database before calling the delete method. However, if you know the primary key of the model, you may delete the model without retrieving it. To do so, call the destroy method:

```
App\Flight::destroy(1);
App\Flight::destroy([1, 2, 3]);
App\Flight::destroy(1, 2, 3);
```

Deleting Models By Query

Of course, you may also run a delete statement on a set of models. In this example, we will delete all flights that are marked as inactive. Like mass updates, mass deletes will not fire any model events for the models that are deleted:

```
$deletedRows = App\Flight::where('active', 0)->delete();
```

LARAVEL'S Models and DBMS functions

LARAVEL's MODEL class and DB class for DBMS Manipulation

1. Create a database, table and some records. Set a user with privileges.
   (ex: laradb→table1)

2. Set database configuration
   [app/config/database.php]

3. Edit the .env file on your project root directory
4. Create a model class for your table. Navigate to your project root directory then use the ff command:

   ```
   />php artisan make:model table1
   ```

5. The model will be in app/.

   Edit the model class and specify the table name.

   ```
   <?php
   namespace App;
   use Illuminate\Database\Eloquent\Model;

   class table1 extends Model
   {
      protected $table='table1';
   }
   ```

Displaying records in a view

[app/Http/Requests/routes.php]

```
Route::get('/viewtable1','testController@viewtable1');
```

Getting the values from the table using the model class and passing it to the a view page

[app/Http/Controllers/testController]

```
use App\table1;

public function viewtable1()
  {
        $records = table1::all();
        return View::make('viewtable1')->with('records',$records);
  }
```

Displaying the data on the view

[app/Resources/views/viewtable1]

```
foreach ($records as $record) {
                echo $record->id.'<br />';
                echo $record->fname.'<br />';
                echo $record->message.'<br />';
        }
```

Running sql queries using laravel's DB class

Create a controller for your sample dbms project. Add functions for the corresponding CRUD operations.

*In your project directory, open cmd then type:

```
/>php artisan make:controller dbmsController
```

*add the highlighted references (input, model,DB,view)
Sample:

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;
use App\Http\Requests;
use Illuminate\Support\Facades\Input;
use View;
use App\table1;
use DB;

class DbmsController extends Controller
{
```

```
    public function index()
    {   }

    public function viewall()
    {   }

    public function addrecord()
    {   }

    public function deleterecord()
    {   }

    public function updaterecord()
    {   }

}
```

*Create views for the controller functions in project/resources/resources/views

*Add routes for your controller functions (project/app/http/routes.php)

```
Route::get('/view','DbmsController@viewall');
Route::get('/add','DbmsController@addrecord');
Route::get('/delete','DbmsController@deleterecord');
Route::get('/edit','DbmsController@updaterecord');
```

Calling views from controller

Sample:

```
public function viewall()
{
    return View::make('view');
}
```

Executing raw sql queries in controller:

//add in reference:

```
Use App/yourmodelname;
Use DB;
```

//in function, direct execution of query after getting input from users:

```
DB::select(DB::raw('select * from table'));
```

//or through PDO

```
DB::connection()->getPdo()->exec( $sql );
```

//or

```
$sql = 'UPDATE my_table SET updated_at = FROM_UNIXTIME(nonce) WHERE id = ' . strval($this->id);
DB::statement($sql);
```

//or

```
DB::connection()->getPdo()->exec('USE '.$dbConfig['database']);
```

//or

Note: Laravel's DB class also supports running raw SQL select, insert, update, and delete queries, like:

```
$users = DB::select('select id, name from users');
DB::insert(sql here);
DB::update(sql here);
DB::delete(sql here);
```