

# MODULE 5

**JSON Data Interchange Format: Syntax, Data Types, Object, JSON Schema, Manipulating JSON data with PHP**

**Web Development Frameworks: Laravel Overview-Features of Laravel-Setting up a Laravel Development Environment-Application structure of Laravel-Routing -Middleware-Controllers Route Model Binding-Views-Redirections-Request and Responses.**

# JSON-JavaScript Object Notation

- A data interchange format is a text format used to exchange data between platforms.
- Another data interchange format you may already have heard of is XML.
- The world needs data interchange formats, like XML and JSON, to exchange data between very different systems.
- JSON is easy to read and write than XML.
- JSON is language independent.
- JSON supports array, object, string, number and values.

# JSON Syntax

- JSON Objects

In JSON, objects refer to dictionaries, which are enclosed in curly brackets, i.e., { }.

These objects are written in key/value pairs, where the key has to be a string and values have to be a valid JSON data type such as string, number, object, Boolean or null.

Here the key and values are separated by a colon, and a comma separates each key/value pair.

Eg:     {"name" : "Jack", "employeeid" : 001, "present" : false}

- JSON is based on the syntactic representation of the properties of JavaScript object literals. This does not include the functions of JavaScript object literals.
- In the JSON name-value pair, the name is always surrounded by double quotes.
- JSON files use the .json extension.

# JSON Data types

- Object
- String
  - Number
  - Boolean
  - Null
- Array

- JSON Strings

Strings in JSON must be written in double quotes.

Eg: `{"name": "John"}`

- JSON Numbers

Numbers in JSON must be an integer or a floating point.

`{"age": 30}`

- JSON Booleans-Values in JSON can be true/false.

`{"sale": true}`

# Objects

Values in JSON can be objects.

```
{  
  "employee": {"name": "John", "age": 30, "city": "New  
York"}  
}
```

JSON Arrays-Values in JSON can be arrays.

```
{  
  "employees": ["John", "Anna", "Peter"]  
}
```

- JSON null-Values in JSON can be null.

```
{"middlename": null}
```

# Array of objects

*Example 3-24. Using an array of objects to represent the questions and answers of a test*

```
{
  "test": [
    {
      "question": "The sky is blue",
      "answer": true
    },
    {
      "question": "The earth is flat.",
      "answer": false
    },
    {
      "question": "A cat is a dog.",
      "answer": false
    }
  ]
}
```



# JSON Schema

- JSON Schema is a specification for JSON based format for defining the structure of JSON data
- In our very first name-value pair of our JSON, we must declare it as a schema document
- { "\$schema": "http://json-schema.org/draft-04/schema#" }
- The second name-value pair in our JSON Schema Document will be the title

```
{ "$schema": "http://json-schema.org/draft-04/schema#",  
  "title": "Cat" }
```

- In the third name-value pair of our JSON Schema Document,
- The property value is essentially a skeleton of the name-value pairs of the JSON we want.
- Instead of a literal value, we have an object that defines the data type, and optionally the description.
- Fourth name-value pair has the name “required” and a value of the array data type. The array includes the fields we require.
- Schema can be used to test the validity of the JSON code

# Keywords in schema

Sr.No.	Keyword & Description
1	<b>\$schema</b> The \$schema keyword states that this schema is written according to the draft v4 specification.
2	<b>title</b> You will use this to give a title to your schema.
3	<b>description</b> A little description of the schema.
4	<b>type</b> The type keyword defines the first constraint on our JSON data: it has to be a JSON Object.
5	<b>properties</b> Defines various keys and their value types, minimum and maximum values to be used in JSON file.
6	<b>required</b> This keeps a list of required properties.

7	<p><b>minimum</b></p> <p>This is the constraint to be put on the value and represents minimum acceptable value.</p>
8	<p><b>exclusiveMinimum</b></p> <p>If "exclusiveMinimum" is present and has boolean value true, the instance is valid if it is strictly greater than the value of "minimum".</p>
9	<p><b>maximum</b></p> <p>This is the constraint to be put on the value and represents maximum acceptable value.</p>
10	<p><b>exclusiveMaximum</b></p> <p>If "exclusiveMaximum" is present and has boolean value true, the instance is valid if it is strictly lower than the value of "maximum".</p>
11	<p><b>multipleOf</b></p> <p>A numeric instance is valid against "multipleOf" if the result of the division of the instance by this keyword's value is an integer.</p>
12	<p><b>maxLength</b></p> <p>The length of a string instance is defined as the maximum number of its characters.</p>
13	<p><b>minLength</b></p> <p>The length of a string instance is defined as the minimum number of its characters.</p>
14	<p><b>pattern</b></p> <p>A string instance is considered valid if the regular expression matches the instance successfully.</p>

```

{
  "$schema": "http://json-schema.org/draft-04/schema#",
  "title": "Cat",
  "properties": {
    "name": {
      "type": "string"
    },
    "age": {
      "type": "number",
      "description": "Your cat's age in years."
    },
    "declawed": {
      "type": "boolean"
    },
    "description": {
      "type": "string"
    }
  },
  "required": [
    "name",
    "age",
    "declawed"
  ]
}

```

#### Example 4-6. Valid JSON

```

{
  "name": "Fluffy",
  "age": 2,
  "declawed": false,
  "description" : "Fluffy loves to sleep all day."
}

```

---

#### Example 4-7. Valid JSON without the description field

```

{
  "name": "Fluffy",
  "age": 2,

```

---

```

    "declawed": false
  }

```

# Manipulating JSON data with PHP

- PHP also includes built-in support for serializing and deserializing JSON.
- PHP refers to this as encoding and decoding JSON.
- When we encode something, we convert it to a coded (unreadable) form.
- When we decode something, we convert it back into a readable form. From the perspective of PHP, JSON is in a coded format.
- Therefore, to serialize JSON, the “json\_encode” function is called, and to deserialize JSON, the “json\_decode” function is called.

- The `json_encode()` function is used to encode a value to JSON format.
- The `json_decode()` function is used to decode or convert a JSON object to a PHP object

# Creating and serializing an address

```
<?php
class Account {
    public $firstName;
    public $lastName;
    public $phone;
    public $gender;
    public $addresses;
    public $famous;
}

class Address {
    public $street;
    public $city;
    public $state;
    public $zip;
}

$address1 = new Address();

$address1->street = "123 fakey st";
$address1->city = "Somewhere";
$address1->state = "CA";
$address1->zip = 96027;

$address2 = new Address();
$address2->street = "456 fake dr";
$address2->city = "Some Place";
$address2->state = "CA";
$address2->zip = 96345;

$account = new Account();
$account->firstName = "Bob";
$account->lastName = "Barker";
$account->gender = "male";
$account->phone = "555-555-5555";
$account->famous = true;
$account->addresses = array ($address1, $address2);

$json = json_encode($account);


?>
```



*Example 9-12. The JSON result from `json_encode($account)`*

```
{
  "firstName": "Bob",
  "lastName": "Barker",
  "phone": "555-555-5555",
  "gender": "male",
  "addresses": [
    {
      "street": "123 fakey st",
      "city": "Somewhere",
      "state": "CA",
      "zip": 96027
    },
    {
      "street": "456 fake dr",
      "city": "Some Place",
      "state": "CA",
      "zip": 96345
    }
  ],
  "famous": true
}
```

# Deserializing JSON

- To serialize JSON in PHP, we used the built-in `json_encode` function. To deserialize JSON, we use the `json_decode` function.
- Unfortunately, this does not have built-in support for deserializing the JSON to a specified PHP object, such as the `Account` class.
- So, we must do a little processing to reshape our data back into the PHP object.
- Let's add a new function to the account object, to load up its properties from a JSON string.
-  “loadFromJSON” function accepts a JSON string for a parameter, calls the built in “`json_decode`” function to deserialize to a generic PHP object, and maps the name-value pairs to the `Account` properties by name in the `foreach` loop

```

class Account {
    public $firstName;
    public $lastName;
    public $phone;
    public $gender;
    public $addresses;
    public $famous;

    public function loadFromJSON($json)
    {
        $object = json_decode($json);
        foreach ($object AS $name => $value)
        {
            $this->{$name} = $value;
        }
    }
}

```

Next, we can create a new Address object, and call the new “loadFromJSON” function.

*Example 9-14. Calling our new “loadFromJSON” function to deserialize the account JSON back into the Address object. The last line will display “Bob Barker.”*

```

$json = json_encode($account);

$deserializedAccount = new Account();
$deserializedAccount->loadFromJSON($json);

echo $deserializedAccount->firstName . " " . $deserializedAccount->lastName;

```

**Deserializing JSON**

# Requesting JSON

- To make an HTTP request for a resource with PHP, I can use the built in function “file\_get\_contents.”
- This function returns the resource body as a string. We can then deserialize the string to a PHP object.

*Example 9-15. Resource at URL <http://localhost:5984/accounts/ddc14efcf71396463f53c0f8800019ea> from my local CouchDB API.*

```
{
  "_id": "ddc14efcf71396463f53c0f8800019ea",
  "_rev": "6-69fd853972074668f99b88a86aa6a083",
  "address": {
    "street": "123 fakey ln",
    "city": "Some Place",
    "state": "CA",
    "zip": "96037"
  },
  "gender": "female",
  "famous": false,
  "age": 28,
  "firstName": "Mary",
  "lastName": "Thomas"
}
```

Calling the built in “file\_get\_contents” function to get the account JSON resource from the CouchDB API. Next, a new account object is created and our “loadFromJSON” function is called to deserialize. The last line will display “Mary @mas.”

```
$url = "http://localhost:5984/accounts/3636fa3c716f9dd4f7407bd6f700076c";  
$json = file_get_contents($url);  
  
$deserializedAccount = new Account();  
$deserializedAccount->loadFromJSON($json);  
  
echo $deserializedAccount->firstName . " " . $deserializedAccount->lastName;
```

# Web Development Frameworks: Laravel

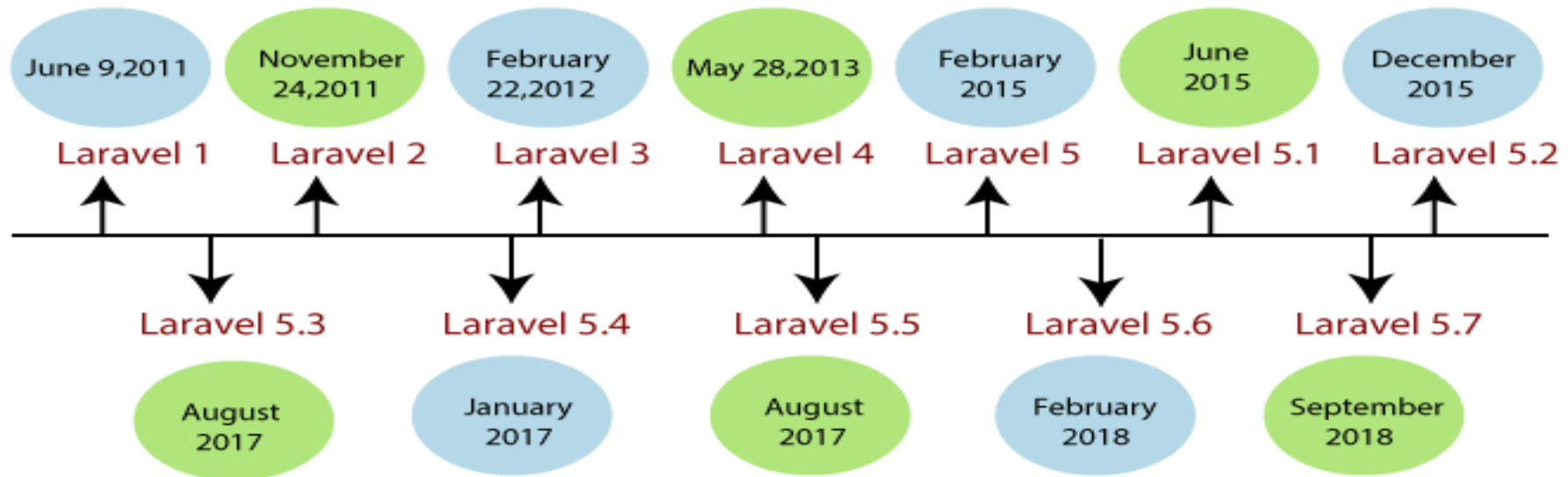
## Overview

- Laravel is an open-source PHP framework, which is robust and easy to understand.
- It follows a model-view-controller design pattern.
- Laravel reuses the existing components of different frameworks which helps in creating a web application.
- The web application thus designed is more structured and pragmatic.
- Laravel offers a rich set of functionalities which incorporates the basic features of PHP frameworks like CodeIgniter, Yii and other programming languages like Ruby on Rails.
- Laravel has a very rich set of features which will boost the speed of web development.

- Moreover, a website built in Laravel is secure and prevents several web attacks.
- The web application becomes more scalable, owing to the Laravel framework.
- It includes namespaces and interfaces, thus helps to organize and manage resources.



## History of Laravel



- Composer

Composer is a tool which includes all the dependencies and libraries. It allows a user to create a project with respect to the mentioned framework (for example, those used in Laravel installation). Third party libraries can be installed easily with help of composer.

All the dependencies are noted in **composer.json** file which is placed in the source folder.

- Artisan

Command line interface used in Laravel is called **Artisan**. It includes a set of commands which assists in building a web application. These commands are incorporated from Symfony framework, resulting in add-on features in Laravel 5.1 (latest version of Laravel).

# Features of Laravel

- Modularity

Laravel provides 20 built in libraries and modules which helps in enhancement of the application. Every module is integrated with Composer dependency manager which eases updates.

- Testability

Laravel includes features and helpers which helps in testing through various test cases. This feature helps in maintaining the code as per the requirements.

- Configuration Management

A web application designed in Laravel will be running on different environments, which means that there will be a constant change in its configuration. Laravel provides a consistent approach to handle the configuration in an efficient way.

- Query Builder and ORM

Laravel incorporates a query builder which helps in querying databases using various simple chain methods. It provides **ORM** (Object Relational Mapper) and **ActiveRecord** implementation called Eloquent.

- Schema Builder

Schema Builder maintains the database definitions and schema in PHP code. It also maintains a track of changes with respect to database migrations.

- Template Engine

Laravel uses the **Blade Template** engine, a lightweight template language used to design hierarchical blocks and layouts with predefined blocks that include dynamic content.

- E-mail

Laravel includes a **mail** class which helps in sending mail with rich content and attachments from the web application.

- Authentication

User authentication is a common feature in web applications. Laravel eases designing authentication as it includes features such as **register**, **forgot password** and **send password reminders**.

- Redis

Laravel uses **Redis** to connect to an existing session and general-purpose cache. Redis interacts with session directly.

- Queues

Laravel includes queue services like emailing large number of users or a specified **Cron** job. These queues help in completing tasks in an easier manner without waiting for the previous task to be completed.

- Event and Command Bus

Laravel 5.1 includes **Command Bus** which helps in executing commands and dispatch events in a simple way. The commands in Laravel act as per the application's lifecycle.

# Setting up a Laravel Development Environment

- The Laravel framework has a few system requirements. All of these requirements are satisfied by the Laravel Homestead virtual machine, so it's highly recommended that you use Homestead as your local Laravel development environment.
- However, if you are not using Homestead, you will need to make sure your server meets the following requirements:
- PHP  $\geq 7.2.5$
- BCMath PHP Extension
- Ctype PHP Extension
- Fileinfo PHP extension
- JSON PHP Extension
- Mbstring PHP Extension
- OpenSSL PHP Extension
- PDO PHP Extension
- Tokenizer PHP Extension
- XML PHP Extension

- For managing dependencies, Laravel uses **composer**. Make sure you have a Composer installed on your system before you install Laravel. In this chapter, you will see the installation process of Laravel.
- You will have to follow the steps given below for installing Laravel onto your system –
- **Step 1** – Visit the following URL and download composer to install it on your system.
- <https://getcomposer.org/download/>
- **Step 2** – After the Composer is installed, check the installation by typing the Composer command in the command prompt as shown in the following screenshot.



```
Administrator: C:\Windows\System32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\wamp\www\laravel>php artisan --version
Laravel Framework version 5.1.23 <LTS>

C:\wamp\www\laravel>cd\

C:\>composer

Composer version 1.0-dev (c7ed232ef42c2bd63cdba057b6c7c8043b37cd5a) 2015-10-29 09:52:59

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
  -d, --working-dir=WORKING-DIR If specified, use the given directory as working directory.
  -vvvv, --verbose         Increase the verbosity of messages: 1 for normal output, 2 for more verbose output and 3 for debug
```

- Installing Laravel with the Laravel Installer Tool If you have Composer installed globally, installing the Laravel installer tool is as simple as running the following command:

```
composer global require "laravel/installer=~1.1"
```

Once you have the Laravel installer tool installed, spinning up a new Laravel project is simple.

Just run this command from your command line:

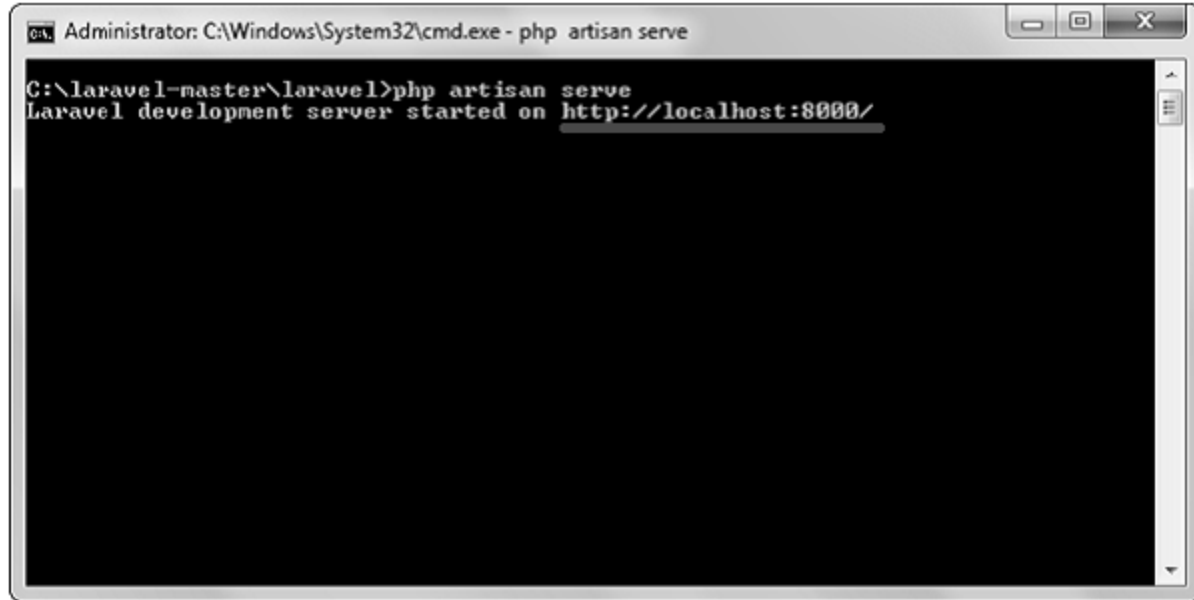
```
laravel new projectName
```

This will create a new subdirectory of your current directory named projectName and install a bare Laravel project in it.

- Installing Laravel with Composer's create-project Feature
- Composer also offers a feature called create-project for creating new projects with a particular skeleton. To use this tool to create a new Laravel project, issue the following command:
- `composer create-project laravel/laravel projectName --prefer-dist`
- Just like the installer tool, this will create a subdirectory of your current directory named projectName that contains a skeleton Laravel install, ready for you to develop

Start the Laravel service by executing the following command.  
`php artisan serve`

--prefer-dist would try to **download and unzip archives of the dependencies using GitHub or another API when available**. This is used for faster downloading of dependencies in most cases.



A screenshot of a Windows command prompt window. The title bar at the top reads "Administrator: C:\Windows\System32\cmd.exe - php artisan serve". The command prompt shows the following text:

```
C:\laravel-master\laravel>php artisan serve
Laravel development server started on http://localhost:8000/
```

The text "http://localhost:8000/" is underlined. The rest of the window is black.

- Copy the URL underlined in gray in the above screenshot and open that URL in the browser. If you see the following screen, it implies Laravel has been installed successfully.

Laravel 5

# Laravel's Directory Structure

When you open up a directory that contains a skeleton Laravel application, you'll see the following files and directories:

```
app/  
bootstrap/  
config/  
database/  
public/  
resources/  
routes/  
storage/  
tests/  
vendor/  
.env  
.env.example  
.gitattributes  
.gitignore  
artisan  
composer.json  
composer.lock  
gulpfile.js  
package.json  
phpunit.xml  
readme.md  
server.php
```



The root directory contains the following folders by default:

- app is where the bulk of your actual application will go. Models, controllers, route definitions, commands, and your PHP domain code all go in here.
- bootstrap contains the files that the Laravel framework uses to boot every time it runs.

- config where all the configuration files live.
- database is where database migrations and seeds live.
- public is the directory the server points to when it's serving the website. This contains index.php, which is the front controller that kicks off the bootstrapping process and routes all requests appropriately. It's also where any public-facing files like images, stylesheets, scripts, or downloads go.
- resources is where non-PHP files that are needed for other scripts live. Views, language files, and (optionally) Sass/LESS and source JavaScript files live here.

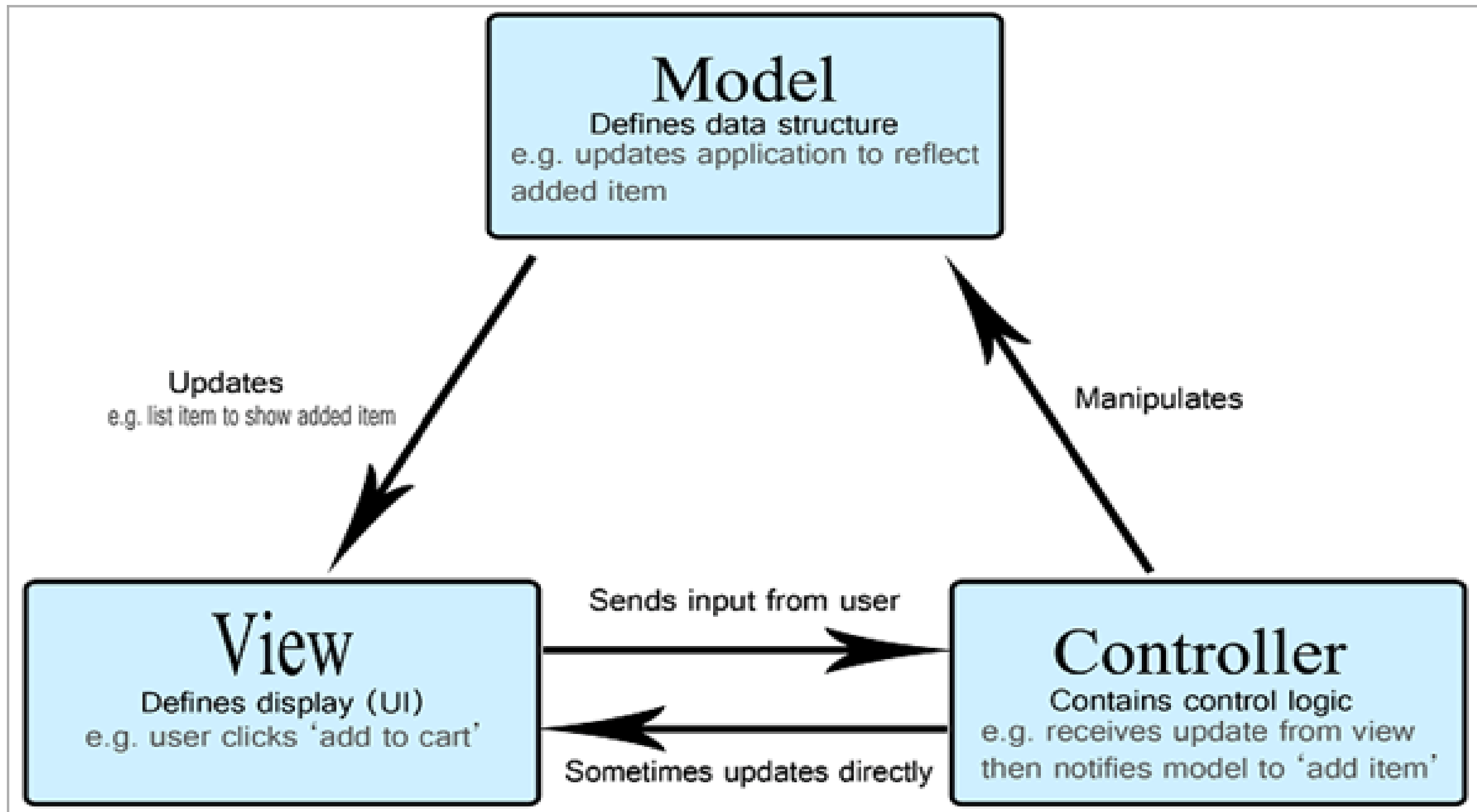
- routes is where all of the route definitions live, both for HTTP routes and “console routes,” or Artisan commands.
- storage is where caches, logs, and compiled system files live.
- tests is where unit and integration tests live.
- vendor is where Composer installs its dependencies. It’s Git-ignored (marked to be excluded from your version control system), as Composer is expected to run as a part of your deploy process on any remote servers.

- *.env* and *.env.example* are the files that dictate the environment variables (variables that are expected to be different in each environment and are therefore not committed to version control). *.env.example* is a template that each environment should duplicate to create its own *.env* file, which is Git-ignored.
- *artisan* is the file that allows you to run Artisan commands (see [Chapter 7](#)) from the command line.
- *.gitignore* and *.gitattributes* are Git configuration files.
- *composer.json* and *composer.lock* are the configuration files for Composer; *composer.json* is user-editable and *composer.lock* is not. These files share some basic information about this project and also define its PHP dependencies.
- *gulpfile.js* is the (optional) configuration file for Elixir and Gulp. This is for compiling and processing your frontend assets.
- *package.json* is like *composer.json* but for frontend assets.
- *phpunit.xml* is a configuration file for PHPUnit, the tool Laravel uses for testing out of the box.
- *readme.md* is a Markdown file giving a basic introduction to Laravel.

- `server.php` is a backup server that tries to allow less-capable servers to still pre-view the Laravel application.

# MVC Architecture Of Laravel

- The Laravel Framework follows **MVC architecture**.
- MVC is an architectural design pattern that helps to develop web applications faster.
- **MVC** stands for **Model-View-Controller**.
- **Model (M)**—A model handles data used by the web application.
- **View (V)**—A view helps to display data to the user.
- **Controller (C)**—A controller interacts with the model to create data for the view.



# Routing

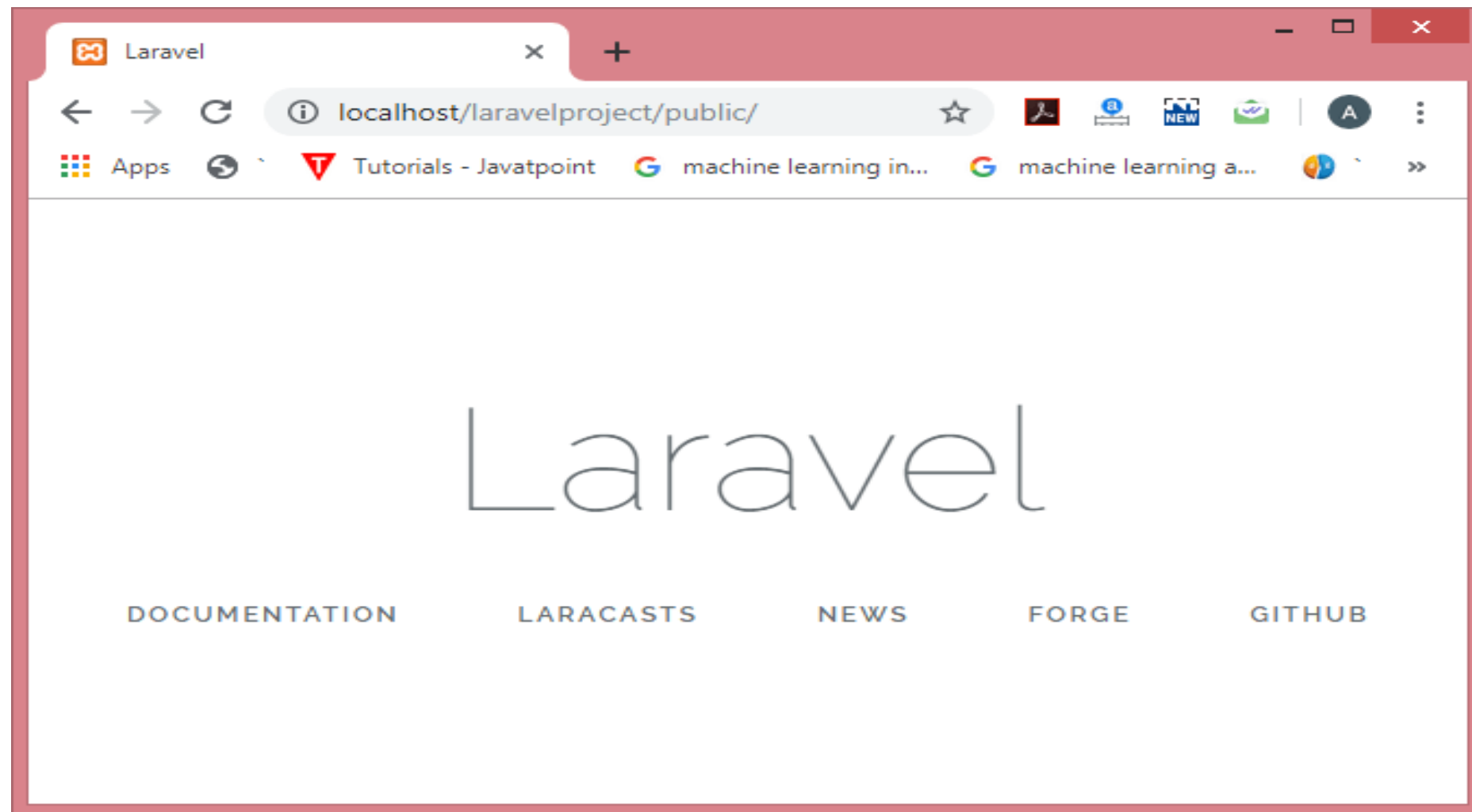
- Routing is one of the essential concepts in Laravel.
- The main functionality of the routes is to route all your application requests to the appropriate controller.
- All Laravel routes are defined inside the route files located in the **routes** directory.
- When we create a project, then a route directory is created inside the project. The **route/web.php** directory contains the definition of route files for your web interface.



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

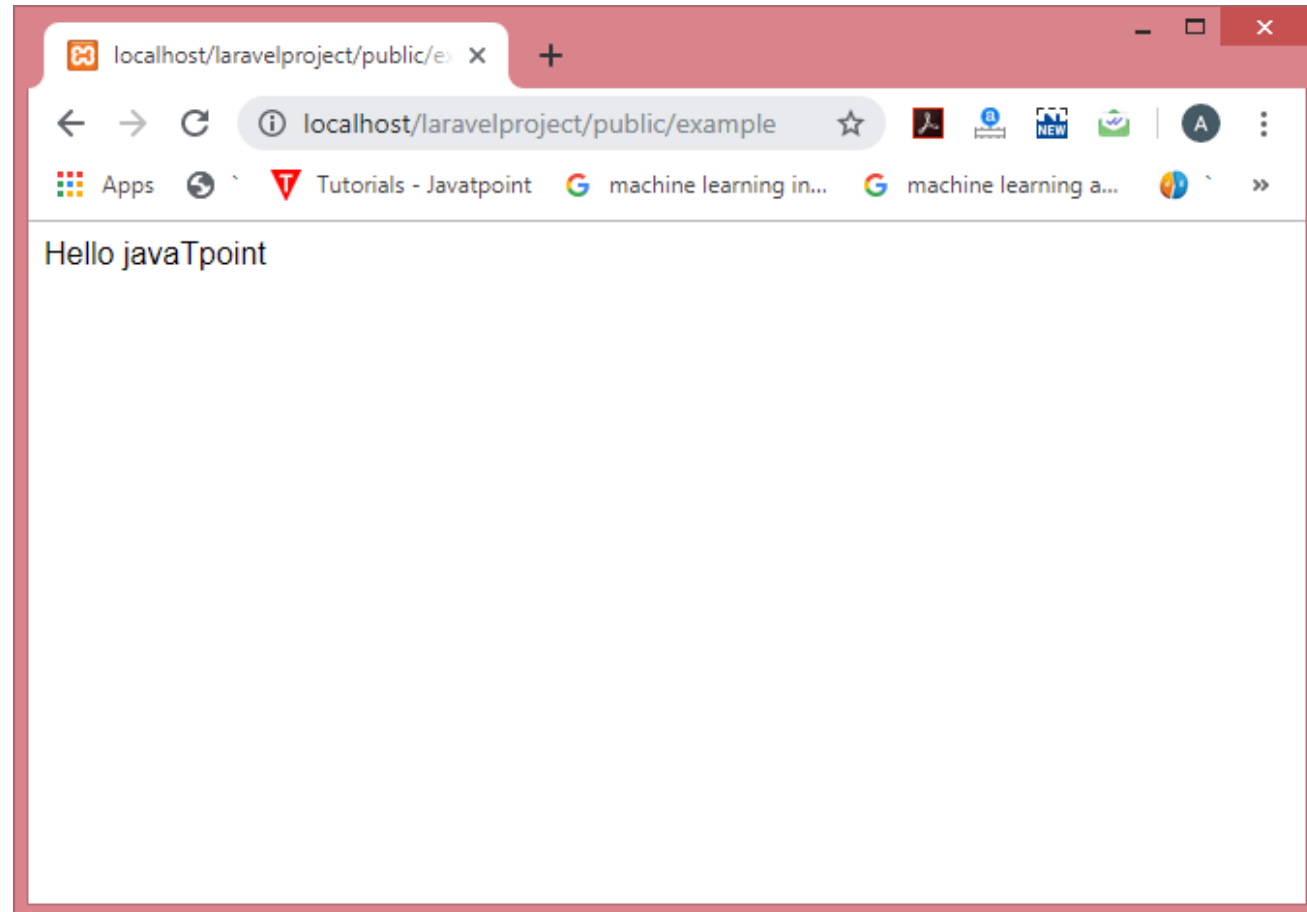
In the above case, Route is the class which defines the static method get(). The get() method contains the parameters '/' and function() closure. The '/' defines the root directory and function() defines the functionality of the get() method.

As the method returns the **view('welcome')**, so the above output shows the welcome view of the Laravel.



```
Route::get('/example', function ()  
{    return "Hello javaTpoint"; });
```

- In the above example, the route is defined in which URL is '/example', so we need to enter the URL **"localhost/laravelproject/public/example"** in the web browser



# Routing

- `Route::get('/', function () { return 'Hello, World!'; });`
- `Route::get('about', function () { return view('about'); });`
- `Route::get('services', function () { return view('services'); });`

# Route Verbs

- `Route::get('/', function () { return 'Hello, World!'; });`
- `Route::post('/', function () {});`
- `Route::put('/', function () {});`
- `Route::delete('/', function () {});`
- `Route::any('/', function () {});`
- `Route::match(['get', 'post'], '/', function () {})`

# Routes calling controller methods

- The other common option is to pass a controller name and method as a string in place of the closure
- `Route::get('/', 'WelcomeController@index');`
- This is telling Laravel to pass requests to that path to the `index()` method of the `App \Http\Controllers\WelcomeController` controller

# Route Parameters

- If the route you're defining has parameters—segments in the URL structure that are variable—it's simple to define them in your route and pass them to your closure

```
Route::get('users/{id}', function ($id) { // });
```

- You can also make your route parameters optional by including a question mark (?)

```
Route::get('users/{id?}', function ($id = 'fallbackId') { // });
```



# Regular expression route constraints

```
Route::get('users/{id}', function ($id) {  
    //  
})->where('id', '[0-9]+');
```

```
Route::get('users/{username}', function ($username) {  
    //  
})->where('username', '[A-Za-z]+');
```

```
Route::get('posts/{id}/{slug}', function ($id, $slug) {  
    //  
})->where(['id' => '[0-9]+', 'slug' => '[A-Za-z]+']);
```

# Route Names

- The simplest way to refer to these routes elsewhere in your application is just by their path. There's a `url()` helper to simplify that linking in your views, if you need it

*Example 3-8. URL helper*

```
<a href="<?php echo url('/'); ?>">  
// outputs <a href="http://myapp.com/">
```

Laravel also allows you to name each route, which enables you to refer to it without explicitly referencing the URL. This is helpful because it means you can give simple nicknames to complex routes.

*Example 3-9. Defining route names*

```
// Defining a route with name in routes/web.php:  
Route::get('members/{id}', 'MembersController@show')->name('members.show');  
  
// Link the route in a view using the route() helper  
<a href="<?php echo route('members.show', ['id' => 14]); ?>">
```

# Route Groups

- Route groups allow you to group several routes together, and apply any shared configuration settings once to the entire group, to reduce this duplication. Additionally, route groups are visual cues to future developers (and to your own brain) that these routes are grouped together

*Example 3-10. Defining a route group*

```
Route::group([], function () {  
    Route::get('hello', function () {  
        return 'Hello';  
    });  
    Route::get('world', function () {  
        return 'World';  
    });  
});
```

# Middleware

- Laravel uses for authenticating users and restricting guest users from using certain parts of a site.
- Middleware acts as a bridge between a request and a response. It is a type of filtering mechanism.
- Laravel includes a middleware that verifies whether the user of the application is authenticated or not. If the user is authenticated, it redirects to the home page otherwise, if not, it redirects to the login page.
- Most common use for route groups is to apply middleware to a group of routes

- Middleware can inspect a request and decorate it, or reject it, based on what it finds. That means middleware is great for something like rate limiting: it can inspect the IP address, check how many times it's accessed this resource in the last minute, and send back a 429 (Too Many Requests) status if a threshold is passed.
- Because middleware also gets access to the response on its way out of the application, it's great for decorating responses

# Creating Custom Middleware

There's an Artisan command to create custom middleware. Let's try it out:

```
php artisan make:middleware BanDeleteMethod
```

You can now open up the file at `app/Http/Middleware/BanDeleteMethod.php`. The default contents are shown in [Example 10-15](#).

*Example 10-15. Default middleware contents*

```
...  
class BanDeleteMethod  
{  
    public function handle($request, Closure $next)  
    {  
        return $next($request);  
    }  
}
```

- **handle** is the **method** that **handles** the Request in the given middleware
- To pass the request deeper into the application (allowing the middleware to "pass"), simply call the **\$next** callback with the **\$request**.

# Binding/Registering Middleware

- We need to register each and every middleware before using it. There are two types of Middleware in Laravel.
- Global Middleware
- Route Middleware
- The **Global Middleware** will run on every HTTP request of the application, whereas the **Route Middleware** will be assigned to a specific route.
- The middleware can be registered at **app/Http/Kernel.php**.
- This file contains two properties **\$middleware** and **\$routeMiddleware**.
- **\$middleware** property is used to register Global Middleware and **\$routeMiddleware** property is used to register route specific middleware.



```
// app/Http/Kernel.php  
protected $middleware = [  
    \Illuminate\Foundation\Http\Middleware\CheckForMaintenanceMode::class,  
    \App\Http\Middleware\BanDeleteMethod::class,  
];
```

# Binding route middleware

- Middleware intended for specific routes can be added as a route middleware or as part of a middleware group
- Route middleware are added to the `$routeMiddleware` array in `app/Http/Kernel.php`
- we have to give each a key that will be used when applying this middleware to a particular route

*Example 10-19. Binding route middleware*

```
// app/Http/Kernel.php
protected $routeMiddleware = [
    'auth' => \App\Http\Middleware\Authenticate::class,
    ...
    'nodelete' => \App\Http\Middleware\BanDeleteMethod::class,
];
```

- We can now use this middleware in our route definitions

```
// ...  
Route::group(['prefix' => 'api', 'middleware' => 'nodelete'], function () {  
    // All routes related to an API  
});
```

*Example 3-11. Restricting a group of routes to logged-in users only*

```
Route::group(['middleware' => 'auth'], function () {  
    Route::get('dashboard', function () {  
        return view('dashboard');  
    });  
    Route::get('account', function () {  
        return view('account');  
    });  
});
```

# Middleware Parameters

- We can also pass parameters with the Middleware. For example, if your application has different roles like user, admin, super admin etc. and you want to authenticate the action based on role, this can be achieved by passing parameters with middleware.
- The middleware that we create contains the following function and we can pass our custom argument after the **\$next** argument.

```
Route::get('company', function () {  
    return view('company.admin');  
})->middleware('auth:owner');
```

To make this work, you'll need to add one or more parameters to the middleware's `handle()` method, and update that method's logic accordingly:

```
public function handle($request, $next, $role)  
{  
    if (auth()->check() && auth()->user()->hasRole($role)) {  
        return $next($request);  
    }  
  
    return redirect('login');  
}
```

# Route Model Binding

- Laravel **route model binding** provides a convenient way to automatically inject the model instances directly into your routes
- This allows you to define that a particular parameter name (e.g., {conference}) will indicate to the route resolver that it should look up an Eloquent record with that ID and then pass it in as the parameter instead of just passing the ID.
- There are two kinds of route model binding: implicit and custom (or explicit)

# implicit

- The simplest way to use route model binding is to name your route parameter something unique to that model (e.g., name it \$conference instead of \$id), then typehint that parameter in the closure/controller method and use the same variable name there.

```
Route::get('conferences/{conference}', function (Conference $conference) {  
    return view('conferences.show')->with('conference', $conference);  
});
```



- Because the route parameter ({conference}) is the same as the method parameter (\$conference), and the method parameter is typehinted with a Conference model (Conference \$conference), Laravel sees this as a route model binding.
- Every time this route is visited, the application will assume that whatever is passed into the URL in place of {conference} is an ID that should be used to look up a Conference, and then that resulting model instance will be passed in to your closure or controller method.

# Custom Route Model Binding

To manually configure route model bindings, add a line like the one in [Example 3-29](#) to the `boot()` method in `App\Providers\RouteServiceProvider`.

*Example 3-29. Adding a route model binding*

```
public function boot(Router $router)
{
    // Just allows the parent's boot() method to still run
    parent::boot($router);

    // Perform the binding
    $router->model('event', Conference::class);
}
```

*Example 3-30. Using an explicit route model binding*

```
Route::get('events/{event}', function (Conference $event) {
    return view('events.show')->with('event', $event);
});
```

# views

- views (or templates) are files that describe what some particular output should look like.
- You might have views for JSON or XML or emails, but the most common views in a web framework output HTML.
- In Laravel, there are two formats of view you can use out of the box: plain PHP, or Blade templates.
- The difference is in the filename: `about.php` will be rendered with the PHP engine, and `about.blade.php` will be rendered with the Blade engine.
- There are three different ways to return a view. For now, just concern yourself with `view()`, but if you ever see `View::make()`, it's the same thing, and you could also inject the `Illuminate\View\ViewFactory` if you prefer

*Example 3-17. Simple view() usage*

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

*Example 3-18. Passing variables to views*

```
Route::get('tasks', function () {  
    return view('tasks.index')  
        ->with('tasks', Task::all());  
});
```

This closure loads the *resources/views/tasks/index.blade.php* or *resources/views/tasks/index.php* view and passes it a single variable named `tasks`, which contains the result of the `Task::all()` method. `Task::all()` is an Eloquent database query we'll learn

- It is simple to pass data to our views from the route definition

*Example 4-12. Reminder on how to pass data to views*

```
Route::get('passing-data-to-views', function () {  
    return view('dashboard')  
        ->with('key', 'value');  
});
```

Let's say you have a sidebar on every page, which is defined in a partial named `partials.sidebar` (`resources/views/partials/sidebar.blade.php`) and then included on every page. This sidebar shows a list of the last seven posts that were published on your site. If it's on every page, every route definition would normally have to grab that list and pass it in, like in [Example 4-13](#).

*Example 4-13. Passing sidebar data in from every route*

```
Route::get('home', function () {  
    return view('home')  
        ->with('posts', Post::recent());  
});  
  
Route::get('about', function () {  
    return view('about')  
        ->with('posts', Post::recent());  
});
```

# Using View Composers to Share Variables with Every View

- It allows you to define that any time a particular view loads, it should have certain data passed to it—without the route definition having to pass that data in explicitly.
- It's possible to share certain variables with every template or just certain templates, like in the following code:

```
view()->share('variableName', 'variableValue');
```

If you want to use `view()->share()`, the best place would be the `boot()` method of a service provider so that the binding runs on every page load

# Redirect

- Laravel offers an easy way to redirect a user to a specific page
- There are two common ways to generate a redirect; we'll use the redirect global helper here, but you may prefer the facade.
- Both create an instance of `Illuminate\Http\RedirectResponse`, perform some convenience methods on it, and then return it

# Redirect to URL

*Example 3-34. Different ways to return a redirect*

```
// Using the global helper to generate a redirect response
Route::get('redirect-with-helper', function () {
    return redirect()->to('login');
});

// Using the global helper shortcut
Route::get('redirect-with-helper-shortcut', function () {
    return redirect('login');
});

// Using the facade to generate a redirect response
Route::get('redirect-with-facade', function () {
    return Redirect::to('login');
});
```



# Redirect back to previous page

- we can redirect back to our previous page URL, so you can do it both way:

```
public function homez ()  
{  
    return redirect()->back();  
}
```

# Redirect to Named Routes

- The `route()` method is the same as the `to()` method, but rather than pointing to a particular path, it points to a particular route name

*Example 3-36. `redirect()->route()`*

```
Route::get('redirect', function () {  
    return redirect()->route('conferences.index');  
});
```

# redirect()->with()

- When you're redirecting users to different pages, you often want to pass certain data along with them.
- Most commonly, you can pass along either an array of keys and values or a single key and value using with()

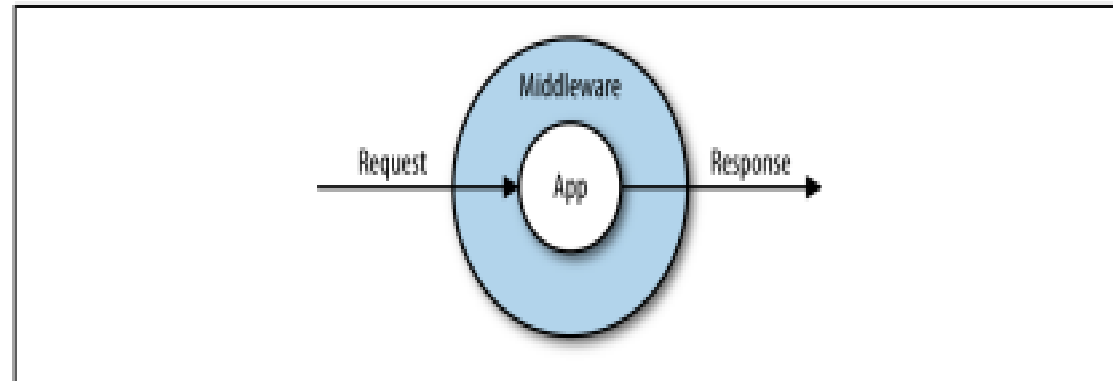
## Other Redirect Methods

The redirect service provides other methods that are less commonly used, but still available:

- `home()` redirects to a route named `home`.
- `refresh()` redirects to the same page the user is currently on.
- `away()` allows for redirecting to an external URL without the default URL validation.
- `secure()` is like `to()` with the `secure` parameter set to `"true"`.
- `action()` allows you to link to a controller and method like this: `redirect()->action('MyController@myMethod')`.
- `guest()` is used internally by the auth system (discussed in [Chapter 9](#)); when a user visits a route he's not authenticated for, this captures the "intended" route and then redirects the user (usually to a login page).
- `intended()` is also used internally by the auth system; after a successful authentication, this grabs the "intended" URL stored by the `guest()` method and redirects the user there.

# Request & Response

- Laravel's Request Lifecycle
- Every request coming into a Laravel application, whether generated by an HTTP request or a command-line interaction, is immediately converted into an Illuminate Request object.
- Then crosses many layers and ends up being parsed by the application itself.
- The application then generates an Illuminate Response object, which is sent back out across those layers and finally returned to the end user



*Figure 10-1. Request/response lifecycle*

- Every Laravel application has some form of configuration set up at the web server level, in an .htaccess file or an Nginx configuration setting or something similar, that captures every web request regardless of URL and routes it to public/index.php in the Laravel application directory (app).

# Request

- Laravel's `Illuminate\Http\Request` class provides an object-oriented way to interact with the current HTTP request being handled by your application as well as retrieve the input, cookies, and files that were submitted with the request.
- 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 route closure or controller method. The incoming request instance will automatically be injected by the Laravel service container:



```
<?php
```

```
namespace App\Http\Controllers;
```

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

```
class UserController extends Controller
```

```
{
```

```
    /**
```

```
     * Store a new user.
```

```
     *
```

```
     * @param  \Illuminate\Http\Request  $request
```

```
     * @return \Illuminate\Http\Response
```

```
     */
```

```
    public function store(Request $request)
```

```
    {
```

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

```
        //
```

```
    }
```

```
}
```

As mentioned, 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) {
    //
});
```

## # Dependency Injection & 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:

```
use App\Http\Controllers\UserController;

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

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

```
<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class UserController extends Controller
{
    /**
     * Update the specified user.
     *
     * @param \Illuminate\Http\Request $request
     * @param string $id
     * @return \Illuminate\Http\Response
     */
    public function update(Request $request, $id)
    {
        //
    }
}
```

# Basic user input

- `all()` returns an array of all user-provided input.
- `input(fieldName)` returns the value of a single user-provided input field.
- `only(fieldName | [array,of,field,names])` returns an array of all userprovided input for the specified field name(s).
  - `except(fieldName | [array,of,field,names])` returns an array of all userprovided input except for the specified field name(s).
- `exists(fieldName)` returns a boolean of whether or not the field exists in the input.

## # Retrieving Input

### # Retrieving All Input Data

You may retrieve all of the incoming request's input data as an `array` using the `all` method. This method may be used regardless of whether the incoming request is from an HTML form or is an XHR request:

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

Using the `collect` method, you may retrieve all of the incoming request's input data as a collection:

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

The `collect` method also allows you to retrieve a subset of the incoming request input as a collection:

```
$request->collect('users')->each(function ($user) {  
    // ...  
})
```

# User and request state

- `method()` returns the method (GET, POST, PATCH, etc.) used to access this route.
- `path()` returns the path (without the domain) used to access this page; e.g., for `http://www.myapp.com/abc/def` it would return `abc/def`.
- `url()` returns the URL (with the domain) used to access this page; e.g., for `http://www.myapp.com/abc` it would return `http://www.myapp.com/abc`.
- `is()` returns a boolean of whether or not the current page request fuzzy-matches a provided string (e.g., `/a/b/c` would be matched by `$request->is('*b*')`, where `*` stands for any characters). It uses a custom regex parser found in `Str::is`.
- `ip()` returns the user's IP address.
- `header()` returns an array of headers (e.g., `['accept-language' => ['enUS,en;q=0.8']]`), or, if passed a header name as a parameter, returns just that header.

- `file()` returns an array of all uploaded files, or, if a key is passed (the file upload field name), returns just the one file.
- `hasFile()` returns a boolean of whether a file was uploaded at the specified key.



# Response

- A web application responds to a user's request in many ways depending on many parameters.
- Laravel provides several different ways to return response. Response can be sent either from route or from controller. The basic response that can be sent is simple string as shown in the below sample code. This string will be automatically converted to appropriate HTTP response.

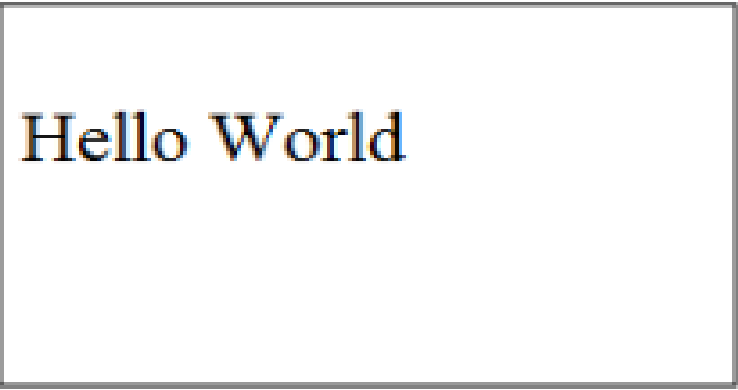
**app/Http/routes.php**

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

**Step 2 – Visit** the following URL to test the basic response.

```
http://localhost:8000/basic_response
```

**Step 3 –** The output will appear as shown in the following image.



Hello World

## Attaching Headers

The response can be attached to headers using the `header()` method. We can also attach the series of headers as shown in the below sample code.

```
return response($content,$status)
    ->header('Content-Type', $type)
    ->header('X-Header-One', 'Header Value')
    ->header('X-Header-Two', 'Header Value');
```

Observe the following example to understand more about Response –

**Step 1** – Add the following code to **app/Http/routes.php** file.

**app/Http/routes.php**

```
Route::get('/header',function() {  
    return response("Hello", 200)->header('Content-Type', 'text/html');  
});
```

**Step 2** – Visit the following URL to test the basic response.

```
http://localhost:8000/header
```

**Step 3** – The output will appear as shown in the following image.



Hello

# Attaching Cookies

- The **withcookie()** helper method is used to attach cookies.
- The cookie generated with this method can be attached by calling **withcookie()** method with response instance.
- By default, all cookies generated by Laravel are encrypted and signed so that they can't be modified or read by the client.

Observe the following example to understand more about attaching cookies –

**Step 1** – Add the following code to `app/Http/routes.php` file.

`app/Http/routes.php`

```
Route::get('/cookie',function() {  
    return response("Hello", 200)->header('Content-Type', 'text/html')  
        ->withcookie('name','Virat Gandhi');  
});
```

**Step 2** – Visit the following URL to test the basic response.

```
http://localhost:8000/cookie
```

**Step 3** – The output will appear as shown in the following image.



Hello

# Specialized Response Types

- The global `view()` helper to show how to return a template—for example, `view(view.name.here)` or something similar.
- But if you need to customize headers, HTTP status, or anything else when returning a view, you can use the `view()` response type

*Example 10-7. Using the `view()` response type*

```
Route::get('/', function (XmlGetterService $xml) {  
    $data = $xml->get();  
    return response()  
        ->view('xml-structure', $data)  
        ->header('Content-Type', 'text/xml');  
});
```

- Sometimes you want your application to force the user's browser to download a file
- The required first parameter is the path for the file you want the browser to download.
- If it's a generated file, you'll need to save it somewhere temporarily. The optional second parameter is the filename for the downloaded file (e.g., export.csv). If you don't pass a string here, it will be automatically generated.
- The optional third parameter allows you to pass an array of headers.



*Example 10-8. Using the download() response type*

```
public function export()  
{  
    return response()  
        ->download('file.csv', 'export.csv', ['header' => 'value']);  
}  
  
public function otherExport()  
{  
    return response()->download('file.pdf');  
}
```

- File responses The file response is similar to the download response, except it allows the browser to display the file instead of forcing a download. This is most common with images and PDFs. The required first parameter is the filename, and the optional second parameter can be an array of headers

*Example 10-9. Using the file() response type*

```
public function Invoice($id)
{
    return response()->file("./Invoices/{$id}.pdf", ['header' => 'value']);
}
```

.....

- JSON responses convert the passed data to JSON (with `json_encode()`) and set the Content-Type to `application/json`.

*Example 10-10. Using the `json()` response type*

```
public function contacts()  
{  
    return response()->json(Contact::all());  
}
```