

Installation

Via Composer Create-Project

composer create-project --prefer-dist laravel/laravel project_name

```
C:\Windows\system32\cmd.exe - composer create-project --prefer-dist laravel/laravel project_name
Microsoft Windows [Version 10.0.14393]
(c) 2016 Microsoft Corporation. All rights reserved.
C:\Users\Borey>cd C:\xampp\htdocs
C:\xampp\htdocs>composer create-project --prefer-dist laravel/laravel project_name
```

create laravel project

Local Development Server

php artisan serve

For port 8080:

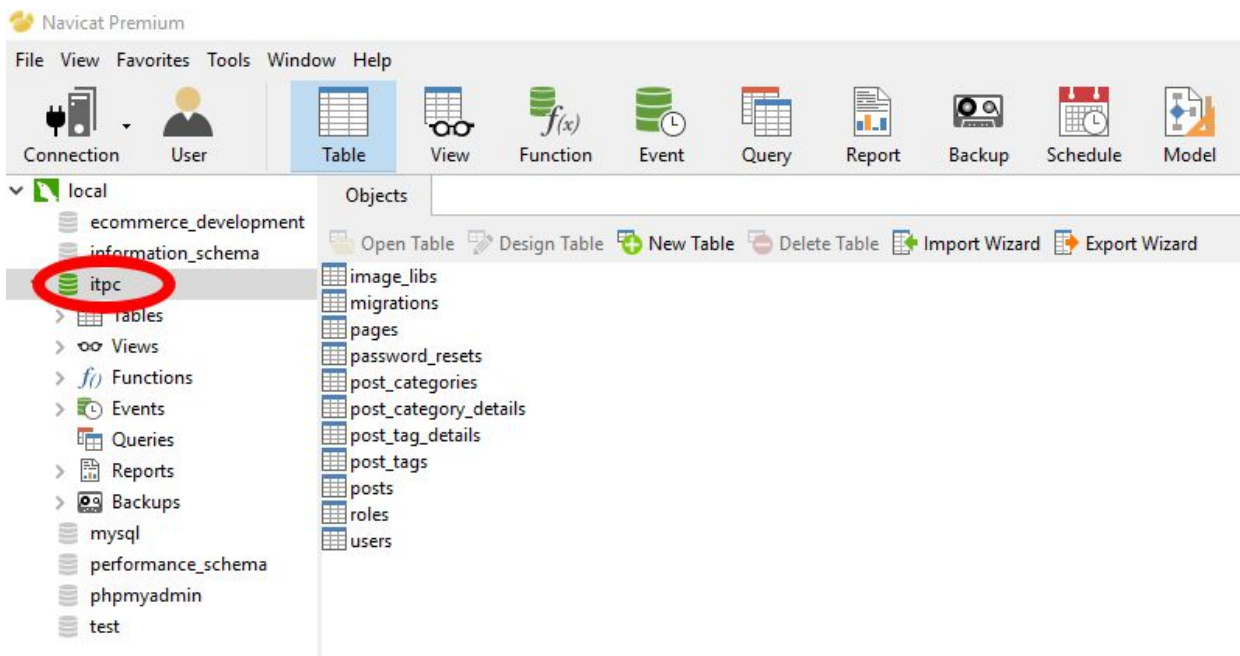
php artisan serve --port=8080

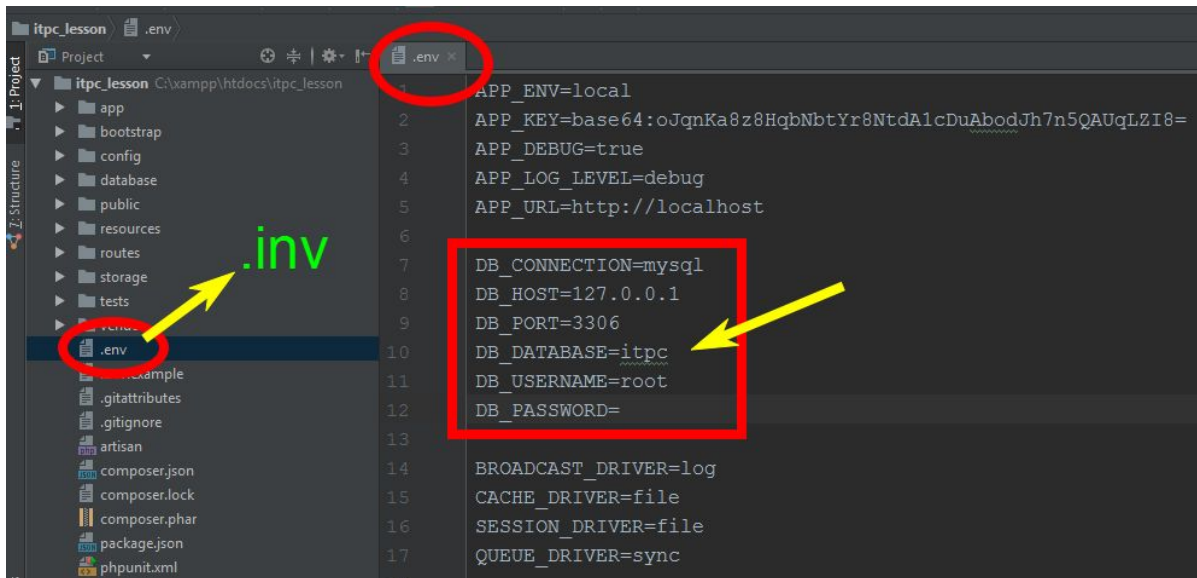
Application Key

php artisan key:generate.

Configuration

-install DB



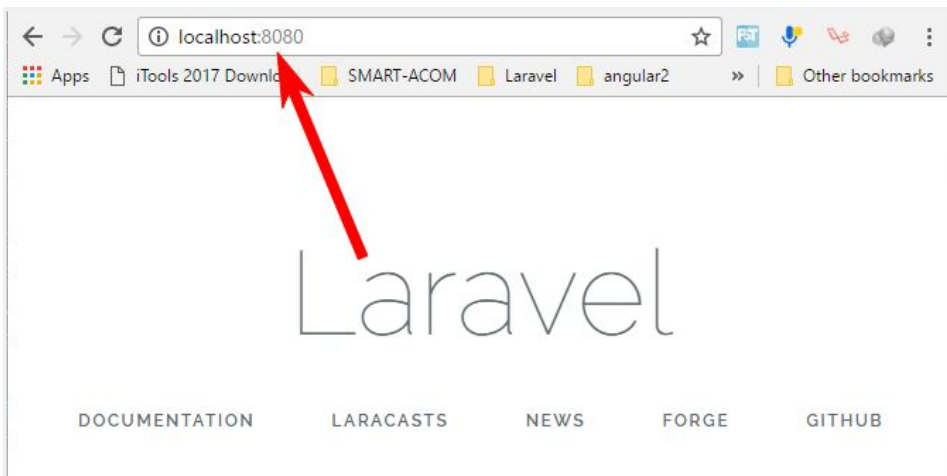
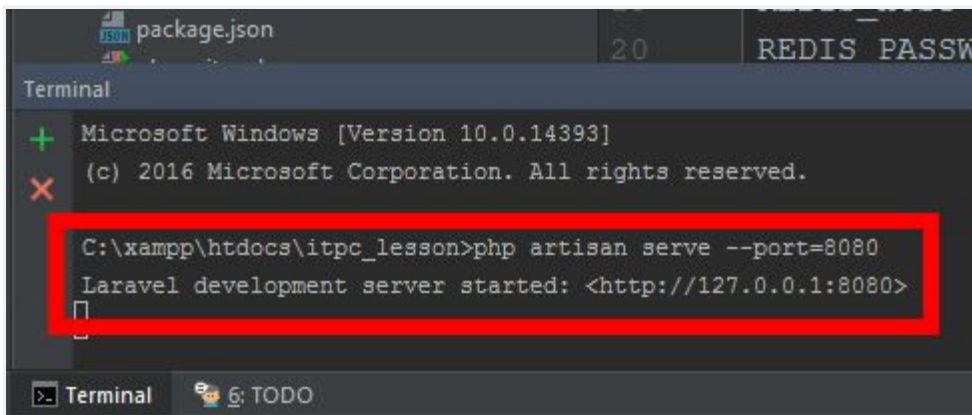


Local Development Server

`php artisan serve`

For port 8080:

`php artisan serve --port=8080`



Accessing Configuration Values

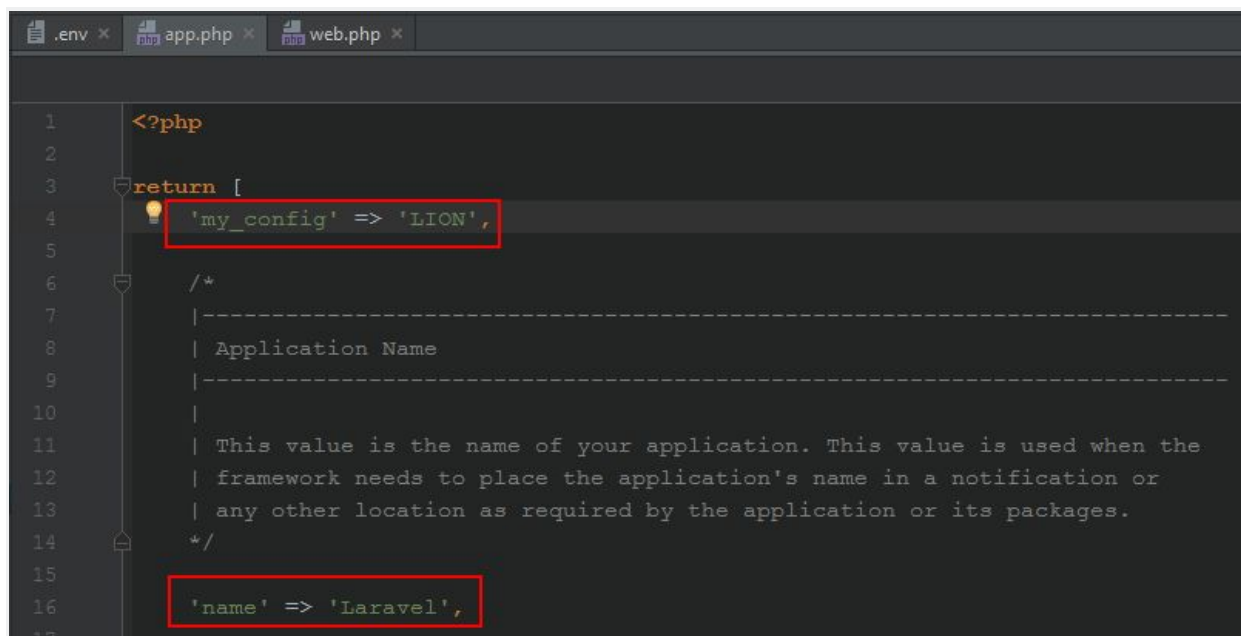
Go => **config/----**.php file. Ex: **config/app.php**

Get config helper in anywhere

```
$value = config('app.timezone');
```

To set configuration values at runtime, pass an array to the **config** helper:

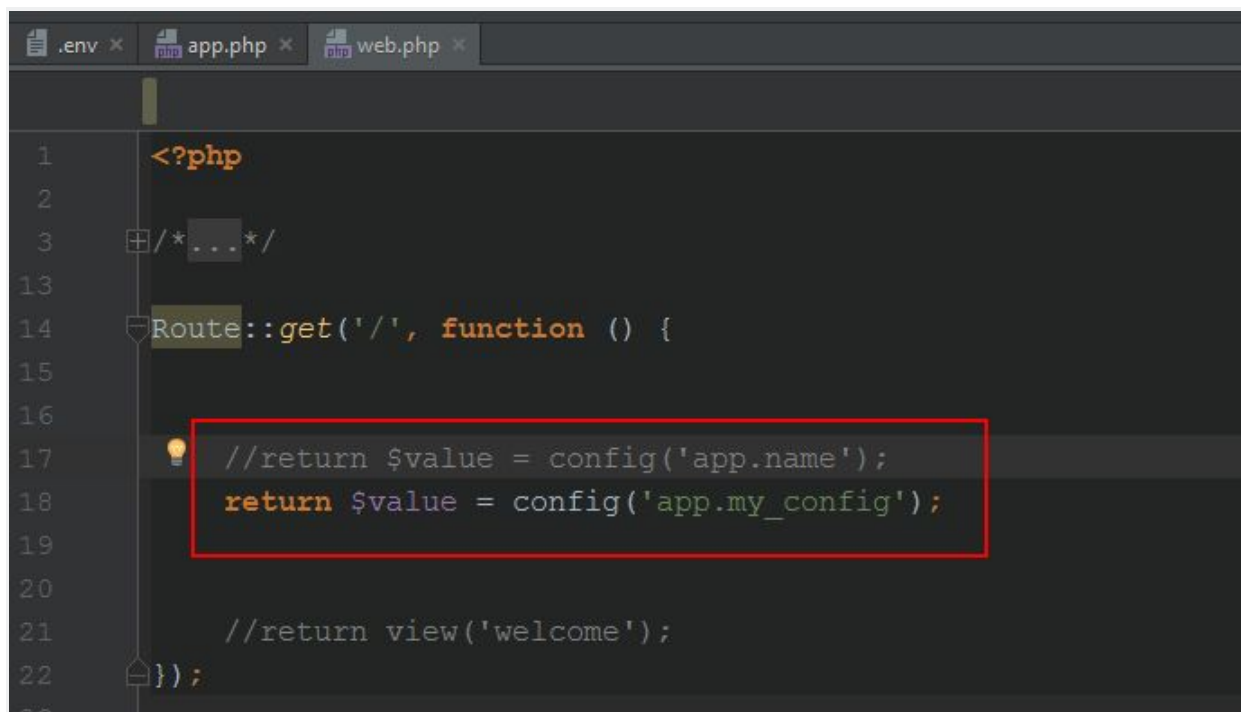
```
config(['app.timezone' => 'America/Chicago']);
```



```
<?php
return [
    'my_config' => 'LION',

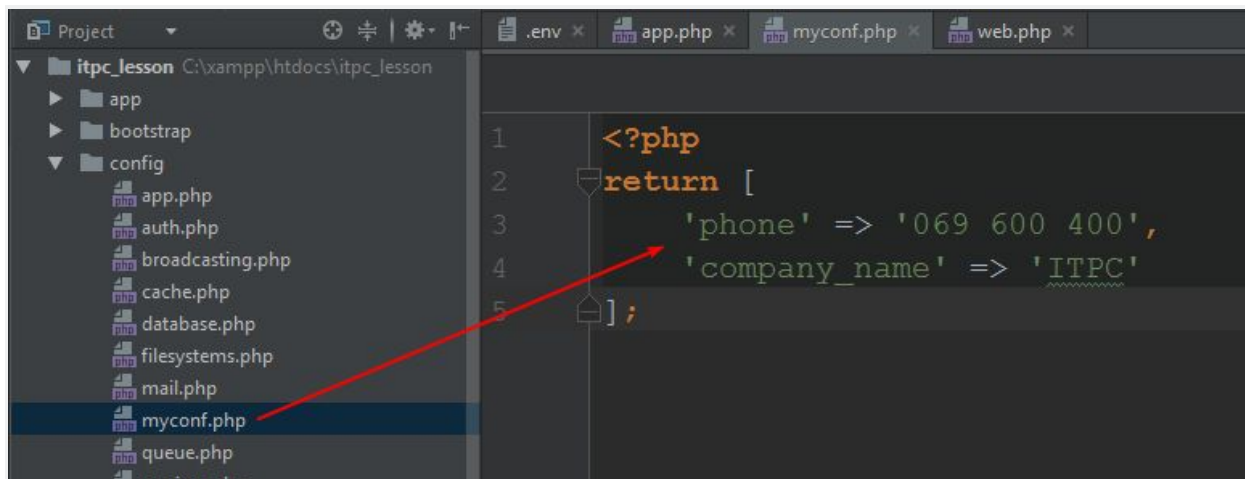
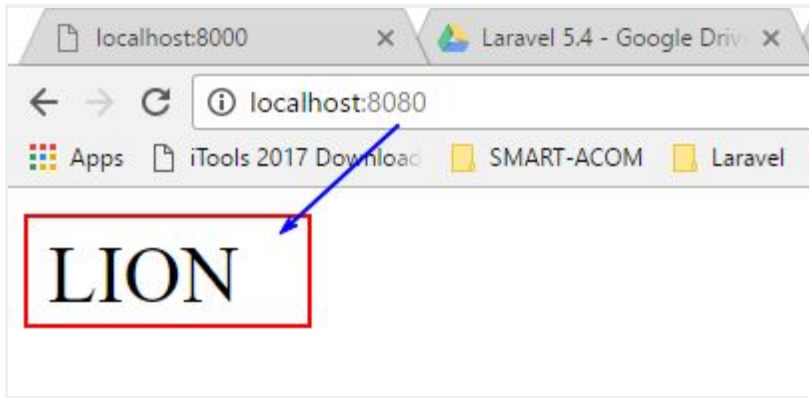
    /*
     * |-----|
     * | Application Name |
     * |-----|
     * | This value is the name of your application. This value is used when the
     * | framework needs to place the application's name in a notification or
     * | any other location as required by the application or its packages.
     */

    'name' => 'Laravel',
]
```



```
<?php
/*...*/
Route::get('/', function () {
    //return $value = config('app.name');
    return $value = config('app.my_config');

    //return view('welcome');
});
```





Maintenance Mode

To enable maintenance mode, simply execute the `down` Artisan command:

`php artisan down`

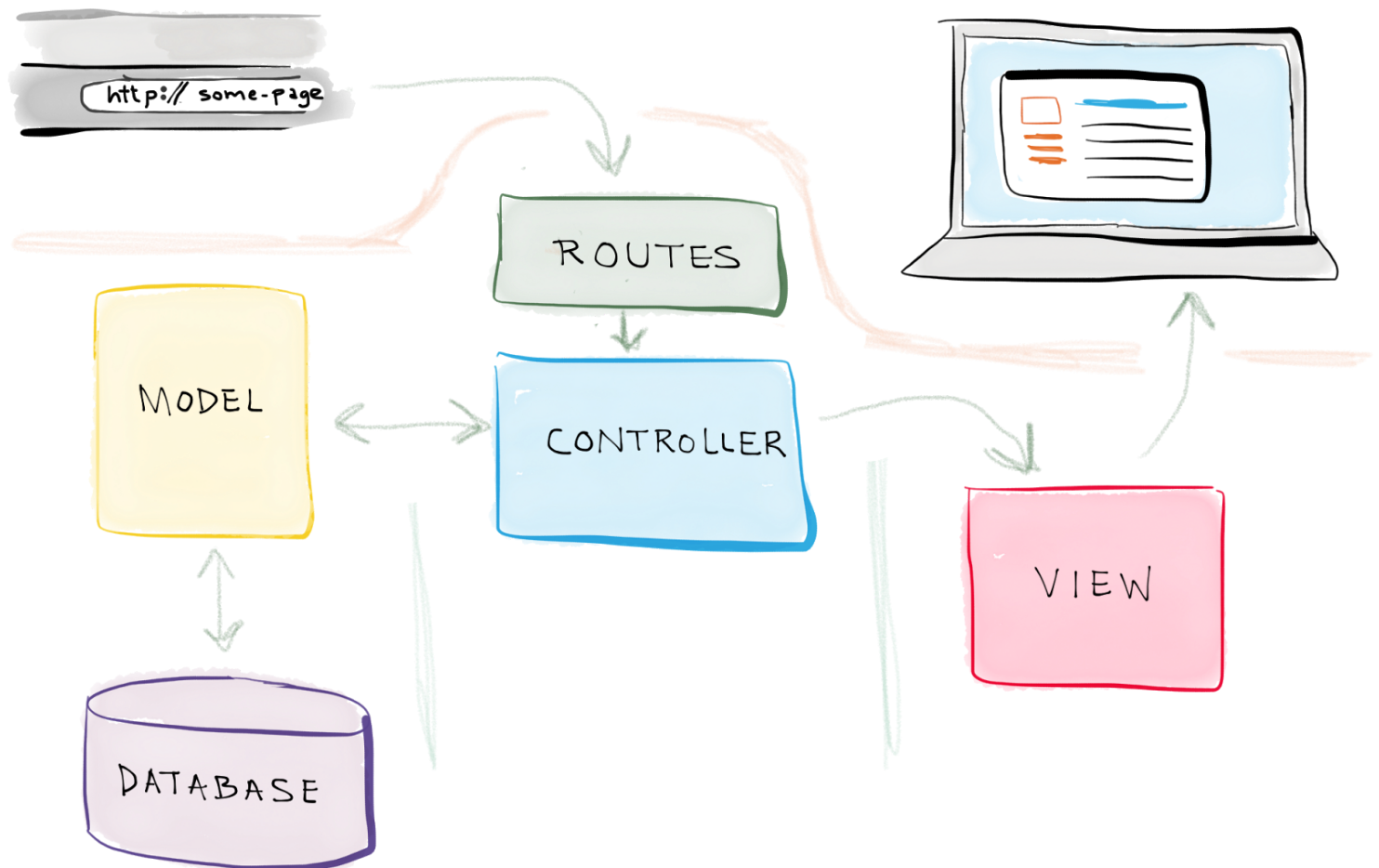
To disable maintenance mode, use the `up` command:

`php artisan up`

Maintenance Mode Response Template

The default template for maintenance mode responses is located in `resources/views/errors/503.blade.php`. You are free to modify this view as needed for your application.

MVC diagram with routes



Routing

Basic Routing

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

The Default Route Files

All Laravel routes are defined in your route files, which are located in the `routes` directory. These files are automatically loaded by the framework. The `routes/web.php` file defines routes that are for your web interface. These routes are assigned the `web` middleware group, which provides features like session state and CSRF protection. The routes in `routes/api.php` are stateless and are assigned the `api` middleware group.

Available Router Methods

The router allows you to register routes that respond to any HTTP verb:

```
Route::get($uri, $callback);  
Route::post($uri, $callback);  
Route::put($uri, $callback);  
Route::patch($uri, $callback);  
Route::delete($uri, $callback);
```

Sometimes you may need to register a route that responds to multiple HTTP verbs. You may do so using the `match` method. Or, you may even register a route that responds to all HTTP verbs using the `any` method:

```
Route::match(['get', 'post'], '/', function () {  
    //  
});  
  
Route::any('foo', function () {  
    //  
});
```

CSRF Protection

Any HTML forms pointing to `POST`, `PUT`, or `DELETE` routes that are defined in the `web` routes file should include a CSRF token field. Otherwise, the request will be rejected.

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


Laravel makes it easy to protect your application from [cross-site request forgery](#) (CSRF) attacks. Cross-site request forgeries are a type of malicious exploit whereby unauthorized commands are performed on behalf of an authenticated user.

In addition to checking for the CSRF token as a POST parameter, the `VerifyCsrfToken` middleware will also check for the `X-CSRF-TOKEN` request header. You could, for example, store the token in a HTML `meta` tag:

```
<meta name="csrf-token" content="{{ csrf_token() }}">
```

Then, once you have created the `meta` tag, you can instruct a library like jQuery to automatically add the token to all request headers. This provides simple, convenient CSRF protection for your AJAX based applications:

```
$.ajaxSetup({  
  headers: {  
    'X-CSRF-TOKEN': $('meta[name="csrf-token"]').attr('content')  
  }  
});
```

Laravel stores the current CSRF token in a `XSRF-TOKEN` cookie that is included with each response generated by the framework. You can use the cookie value to set the `X-XSRF-TOKEN` request header.

This cookie is primarily sent as a convenience since some JavaScript frameworks, like Angular, automatically place its value in the `X-XSRF-TOKEN` header.

Route Parameters

Required Parameters

Of course, sometimes you will need to capture segments of the URI within your route. For example, you may need to capture a user's ID from the URL. You may do so by defining route parameters:

```
Route::get('user/{id}', function ($id) {  
  return 'User '.$id;  
});
```

You may define as many route parameters as required by your route:

```
Route::get('posts/{post}/comments/{comment}', function ($postId, $commentId) {  
    //  
});
```

Optional Parameters

Occasionally you may need to specify a route parameter, but make the presence of that route parameter optional.

You may do so by placing a `?` mark after the parameter name. Make sure to give the route's corresponding variable a default value:

```
Route::get('user/{name?}', function ($name = null) {  
    return $name;  
});  
  
Route::get('user/{name?}', function ($name = 'John') {  
    return $name;  
});
```

Regular Expression Constraints

You may constrain the format of your route parameters using the `where` method on a route instance. The `where` method accepts the name of the parameter and a regular expression defining how the parameter should be constrained:

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

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

```
Route::get('user/{id}/{name}', function ($id, $name) {  
    //  
})->where(['id' => '[0-9]+', 'name' => '[a-z]+']);
```


Global Constraints

If you would like a route parameter to always be constrained by a given regular expression, you may use the `pattern` method. You should define these patterns in the `boot` method of your `RouteServiceProvider`:

`app\Providers\RouteServiceProvider.php`

```
public function boot()
{
    Route::pattern('id', '[0-9]+');

    parent::boot();
}
```

Once the pattern has been defined, it is automatically applied to all routes using that parameter name:

```
Route::get('user/{id}', function ($id) {
    // Only executed if {id} is numeric...
});
```

Named Routes

Named routes allow the convenient generation of URLs or redirects for specific routes. You may specify a name for a route by chaining the `name` method onto the route definition:

```
Route::get('user/profile', function () {
    //
})->name('profile');
```

You may also specify route names for controller actions:

```
Route::get('user/profile', 'UserController@showProfile')->name('profile');
```

Generating URLs To Named Routes

Once you have assigned a name to a given route, you may use the route's name when generating URLs or redirects via the global `route` function:

```
// Generating URLs...
```

```
$url = route('profile');
```

```
// Generating Redirects...
```

```
return redirect()->route('profile');
```

If the named route defines parameters, you may pass the parameters as the second argument to the `route` function. The given parameters will automatically be inserted into the URL in their correct positions:

```
Route::get('user/{id}/profile', function ($id) {
```

```
    //
```

```
}->name('profile');
```

```
$url = route('profile', ['id' => 1]);
```

Route Groups

Route groups allow you to share route attributes, such as middleware or namespaces, across a large number of routes without needing to define those attributes on each individual route.

Middleware

To assign middleware to all routes within a group, you may use the `middleware` key in the group attribute array.

Middleware are executed in the order they are listed in the array:

```
Route::group(['middleware' => 'auth'], function () {
```

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

```
        // Uses Auth Middleware
```

```
    });
```

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

```
        // Uses Auth Middleware
```

```
    });
```

```
});
```

Namespaces

Another common use-case for route groups is assigning the same PHP namespace to a group of controllers using the `namespace` parameter in the group array:

```
Route::group(['namespace' => 'Admin'], function () {  
    // Controllers Within The "App\Http\Controllers\Admin" Namespace  
});
```

Remember, by default, the `RouteServiceProvider` includes your route files within a namespace group, allowing you to register controller routes without specifying the full `App\Http\Controllers` namespace prefix. So, you only need to specify the portion of the namespace that comes after the base `App\Http\Controllers` namespace.

Sub-Domain Routing

Route groups may also be used to handle sub-domain routing. Sub-domains may be assigned route parameters just like route URIs, allowing you to capture a portion of the sub-domain for usage in your route or controller. The sub-domain may be specified using the `domain` key on the group attribute array:

```
Route::group(['domain' => '{account}.myapp.com'], function () {  
    Route::get('user/{id}', function ($account, $id) {  
        //  
    });  
});
```

Route Prefixes

The `prefix` group attribute may be used to prefix each route in the group with a given URI. For example, you may want to prefix all route URIs within the group with `admin:`

```
Route::group(['prefix' => 'admin'], function () {  
    Route::get('users', function () {  
        // Matches The "/admin/users" URL  
    });  
});
```

Form Method Spoofing

HTML forms do not support `PUT`, `PATCH` or `DELETE` actions. So, when defining `PUT`, `PATCH` or `DELETE` routes that are called from an HTML form, you will need to add a hidden `_method` field to the form. The value sent with the `_method` field will be used as the HTTP request method:

```
<form action="/foo/bar" method="POST">
  <input type="hidden" name="_method" value="PUT">
  <input type="hidden" name="_token" value="{{ csrf_token() }}">
</form>
```

You may use the `method_field` helper to generate the `_method` input:

```
{{ method_field('PUT') }}
```

Views

Creating Views

Views contain the HTML served by your application and separate your controller / application logic from your presentation logic. Views are stored in the `resources/views` directory. A simple view might look something like this:

```
<!-- View stored in resources/views/greeting.blade.php -->
<html>
  <body>
    <h1>Hello, {{ $name }}</h1>
  </body>
</html>
```

Since this view is stored at `resources/views/greeting.blade.php`, we may return it using the global `view` helper like so:

```
Route::get('/', function () {
    return view('greeting', ['name' => 'James']);
});
```

Of course, views may also be nested within sub-directories of the `resources/views` directory. "Dot" notation may be used to reference nested views. For example, if your view is stored at `resources/views/admin/profile.blade.php`, you may reference it like so:

```
return view('admin.profile', $data);
```

Passing Data To Views

As you saw in the previous examples, you may pass an array of data to views:

```
return view('greetings', ['name' => 'Victoria']);
```

When passing information in this manner, `$data` should be an array with key/value pairs. Inside your view, you can then access each value using its corresponding key, such as `<?php echo $key; ?>`. As an alternative to passing a complete array of data to the `view` helper function, you may use the `with` method to add individual pieces of data to the view:

```
return view('greeting')->with('name', 'Victoria');
```

Sharing Data With All Views

Occasionally, you may need to share a piece of data with all views that are rendered by your application. You may do so using the view facade's `share` method. Typically, you should place calls to `share` within a service provider's `boot` method. You are free to add them to the `AppServiceProvider` or generate a separate service provider to house them:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\View;

class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        View::share('key', 'value');
    }

    public function register()
    {
        //
    }
}
```

Blade Templates

Blade is the simple, yet powerful templating engine provided with Laravel. Unlike other popular PHP templating engines, Blade does not restrict you from using plain PHP code in your views. In fact, all Blade views are compiled into plain PHP code and cached until they are modified, meaning Blade adds essentially zero overhead to your application. Blade view files use the `.blade.php` file extension and are typically stored in the `resources/views` directory.

Template Inheritance

Defining A Layout

Two of the primary benefits of using Blade are *template inheritance* and *sections*. To get started, let's take a look at a simple example. First, we will examine a "master" page layout. Since most web applications maintain the same general layout across various pages, it's convenient to define this layout as a single Blade view:

```
<!-- Stored in resources/views/layouts/app.blade.php -->
```

```
<html>

  <head>

    <title>App Name - @yield('title')</title>

  </head>

  <body>

    @section('sidebar')

      This is the master sidebar.

    @show

    <div class="container">

      @yield('content')

    </div>

  </body>

</html>
```

As you can see, this file contains typical HTML mark-up. However, take note of the `@section` and `@yield` directives. The `@section` directive, as the name implies, defines a section of content, while the `@yield` directive is used to display the contents of a given section.

Now that we have defined a layout for our application, let's define a child page that inherits the layout.

Extending A Layout

When defining a child view, use the Blade `@extends` directive to specify which layout the child view should "inherit". Views which extend a Blade layout may inject content into the layout's sections using `@section` directives. Remember, as seen in the example above, the contents of these sections will be displayed in the layout using `@yield`:

```
<!-- Stored in resources/views/child.blade.php -->

@extends('layouts.app')

@section('title', 'Page Title')

@section('sidebar')

    @parent

    <p>This is appended to the master sidebar.</p>

@endsection

@section('content')

    <p>This is my body content.</p>

@endsection
```

In this example, the `sidebar` section is utilizing the `@parent` directive to append (rather than overwriting) content to the layout's sidebar. The `@parent` directive will be replaced by the content of the layout when the view is rendered.

Blade views may be returned from routes using the global `view` helper:

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

    return view('child');

});
```

Components & Slots

Components and slots provide similar benefits to sections and layouts; however, some may find the mental model of components and slots easier to understand. First, let's imagine a reusable "alert" component we would like to reuse throughout our application:

```
<!-- /resources/views/alert.blade.php -->

<div class="alert alert-danger">
```



```
{{ $slot }}
```

```
</div>
```

The `{{ $slot }}` variable will contain the content we wish to inject into the component. Now, to construct this component, we can use the `@component` Blade directive:

```
@component('alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

Sometimes it is helpful to define multiple slots for a component. Let's modify our alert component to allow for the injection of a "title". Named slots may be displayed by simply "echoing" the variable that matches their name:

```
<!-- /resources/views/alert.blade.php -->
<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>
    {{ $slot }}
</div>
```

Now, we can inject content into the named slot using the `@slot` directive. Any content not within a `@slot` directive will be passed to the component in the `$slot` variable:

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot
    You are not allowed to access this resource!
@endcomponent
```

Passing Additional Data To Components

Sometimes you may need to pass additional data to a component. For this reason, you can pass an array of data as the second argument to the `@component` directive. All of the data will be made available to the component template as variables:

```
@component('alert', ['foo' => 'bar'])  
...  
@endcomponent
```

Displaying Data

You may display data passed to your Blade views by wrapping the variable in curly braces. For example, given the following route:

```
Route::get('greeting', function () {  
    return view('welcome', ['name' => 'Samantha']);  
});
```

You may display the contents of the `name` variable like so:

```
Hello, {{ $name }}.
```

Of course, you are not limited to displaying the contents of the variables passed to the view. You may also echo the results of any PHP function. In fact, you can put any PHP code you wish inside of a Blade echo statement:

```
The current UNIX timestamp is {{ time() }}.
```

Blade `{{ }}` statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks.

Echoing Data If It Exists

Sometimes you may wish to echo a variable, but you aren't sure if the variable has been set. We can express this in verbose PHP code like so:

```
{{ isset($name) ? $name : 'Default' }}
```

However, instead of writing a ternary statement, Blade provides you with the following convenient shortcut, which will be compiled to the ternary statement above:

```
{{ $name or 'Default' }}
```

In this example, if the `$name` variable exists, its value will be displayed. However, if it does not exist, the word `Default` will be displayed.

Displaying Unescaped Data

By default, Blade `{{ }}` statements are automatically sent through PHP's `htmlspecialchars` function to prevent XSS attacks. If you do not want your data to be escaped, you may use the following syntax:

```
Hello, {!! $name !!}.
```

Be very careful when echoing content that is supplied by users of your application. Always use the escaped, double curly brace syntax to prevent XSS attacks when displaying user supplied data.

Blade & JavaScript Frameworks

Since many JavaScript frameworks also use "curly" braces to indicate a given expression should be displayed in the browser, you may use the `@` symbol to inform the Blade rendering engine an expression should remain untouched. For example:

```
<h1>Laravel</h1>
```

```
Hello, @{{ name }}.
```

In this example, the `@` symbol will be removed by Blade; however, `{{ name }}` expression will remain untouched by the Blade engine, allowing it to instead be rendered by your JavaScript framework.

The `@verbatim` Directive

If you are displaying JavaScript variables in a large portion of your template, you may wrap the HTML in the `@verbatim` directive so that you do not have to prefix each Blade echo statement with an `@` symbol:

```
@verbatim
```

```
<div class="container">
```

```
Hello, {{ name }}.
```

```
</div>
```

```
@endverbatim
```

Control Structures

In addition to template inheritance and displaying data, Blade also provides convenient shortcuts for common PHP control structures, such as conditional statements and loops. These shortcuts provide a very clean, terse way of working with PHP control structures, while also remaining familiar to their PHP counterparts.

If Statements

You may construct `if` statements using the `@if`, `@elseif`, `@else`, and `@endif` directives. These directives function identically to their PHP counterparts:

```
@if (count($records) === 1)
    I have one record!
@endif
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

For convenience, Blade also provides an `@unless` directive:

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

Loops

In addition to conditional statements, Blade provides simple directives for working with PHP's loop structures. Again, each of these directives functions identically to their PHP counterparts:

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach
```

@endforeach

@forelse (\$users as \$user)

```
<li>{{ $user->name }}</li>
```

@empty

```
<p>No users</p>
```

@endforelse

@while (true)

```
<p>I'm looping forever.</p>
```

@endwhile

When using loops you may also end the loop or skip the current iteration:

@foreach (\$users as \$user)

```
@if ($user->type == 1)
```

```
@continue
```

```
@endif
```

```
<li>{{ $user->name }}</li>
```

```
@if ($user->number == 5)
```

```
@break
```

```
@endif
```

@endforeach

You may also include the condition with the directive declaration in one line:

@foreach (\$users as \$user)

```
@continue($user->type == 1)
```

```
<li>{{ $user->name }}</li>
```

```
@break($user->number == 5)
```

@endforeach

The Loop Variable

When looping, a `$loop` variable will be available inside of your loop. This variable provides access to some useful bits of information such as the current loop index and whether this is the first or last iteration through the loop:

```
@foreach ($users as $user)
```

```
    @if ($loop->first)
```

```
        This is the first iteration.
```

```
    @endif
```

```
    @if ($loop->last)
```

```
        This is the last iteration.
```

```
    @endif
```

```
<p>This is user {{ $user->id }}</p>
```

```
@endforeach
```

If you are in a nested loop, you may access the parent loop's `$loop` variable via the `parent` property:

```
@foreach ($users as $user)
```

```
    @foreach ($user->posts as $post)
```

```
        @if ($loop->parent->first)
```

```
            This is first iteration of the parent loop.
```

```
        @endif
```

```
    @endforeach
```

```
@endforeach
```

The `$loop` variable also contains a variety of other useful properties:

| Property | Description |
|-----------------------------------|--|
| <code>\$loop->index</code> | The index of the current loop iteration (starts at 0). |
| <code>\$loop->iteration</code> | The current loop iteration (starts at 1). |

| | |
|-----------------------------------|--|
| <code>\$loop->remaining</code> | The iteration remaining in the loop. |
| <code>\$loop->count</code> | The total number of items in the array being iterated. |
| <code>\$loop->first</code> | Whether this is the first iteration through the loop. |
| <code>\$loop->last</code> | Whether this is the last iteration through the loop. |
| <code>\$loop->depth</code> | The nesting level of the current loop. |
| <code>\$loop->parent</code> | When in a nested loop, the parent's loop variable. |

Comments

Blade also allows you to define comments in your views. However, unlike HTML comments, Blade comments are not included in the HTML returned by your application:

```
{{-- This comment will not be present in the rendered HTML --}}
```

PHP

In some situations, it's useful to embed PHP code into your views. You can use the Blade `@php` directive to execute a block of plain PHP within your template:

```
@php
    //
@endphp
```

Including Sub-Views

Blade's `@include` directive allows you to include a Blade view from within another view. All variables that are available to the parent view will be made available to the included view:

```
<div>
    @include('shared.errors')
    <form>
        <!-- Form Contents -->
    </form>
</div>
```



```
</form>

</div>
```

Even though the included view will inherit all data available in the parent view, you may also pass an array of extra data to the included view:

```
@include('view.name', ['some' => 'data'])
```

Of course, if you attempt to `@include` a view which does not exist, Laravel will throw an error. If you would like to include a view that may or may not be present, you should use the `@includeIf` directive:

```
@includeIf('view.name', ['some' => 'data'])
```

Stacks

Blade allows you to push to named stacks which can be rendered somewhere else in another view or layout. This can be particularly useful for specifying any JavaScript libraries required by your child views:

```
@push('scripts')

    <script src="/example.js"></script>

@endpush
```

You may push to a stack as many times as needed. To render the complete stack contents, pass the name of the stack to the `@stack` directive:

```
<head>

    <!-- Head Contents -->

    @stack('scripts')

</head>
```

Extending Blade

Blade allows you to define your own custom directives using the `directive` method. When the Blade compiler encounters the custom directive, it will call the provided callback with the expression that the directive contains.

The following example creates a `@datetime($var)` directive which formats a given `$var`, which should be an instance of `DateTime`:

```
<?php

namespace App\Providers;

use Illuminate\Support\Facades\Blade;
use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    /**
     * Perform post-registration booting of services.
     *
     * @return void
     */
    public function boot()
    {
        Blade::directive('datetime', function ($expression) {
            return "<?php echo ($expression)->format('m/d/Y H:i'); ?>";
        });
    }

    /**
     * Register bindings in the container.
     *
     * @return void
     */
    public function register()
    {
        //
    }
}
```

As you can see, we will chain the `format` method onto whatever expression is passed into the directive. So, in this example, the final PHP generated by this directive will be:

```
<?php echo ($var)->format('m/d/Y H:i'); ?>
```

After updating the logic of a Blade directive, you will need to delete all of the cached Blade views. The cached Blade views may be removed using the **view:clear** Artisan command.