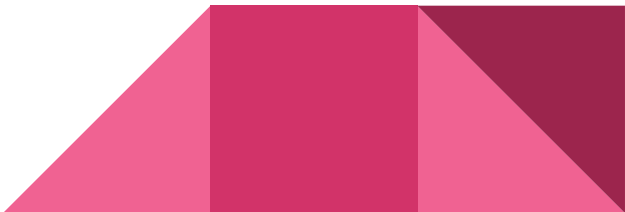


Framework Laravel

Instalación, blade & views

Requisitos


PHP 8 con las siguientes extensiones habilitadas:

- Ctype
 - cURL
 - DOM
 - Fileinfo
 - Filter
 - Hash
 - Mbstring
 - OpenSSL
 - PCRE
 - PDO
 - Session
 - Tokenizer
 - XML
- 

Entorno de desarrollo

- IDE (PHPStorm, Visual Studio Code)
- Servidor web (Apache o Nginx)
- Base de datos (MySQL)
- Gestor de BBDD (PHPMyAdmin)
- Node.js
- Composer
- Artisan (para CLI)

Para trabajar con este framework:

- Xampp (requiere instalación de componentes adicionales)
 - VirtualBox con ubuntu, Apache, MySQL y PHP (instalar todo manualmente)
 - Laravel Herd ← **RECOMENDADO - PARCIALMENTE GRATUITO**
 - Laragon ← **RECOMENDADO - GRATUITO**
- 

Herd

Settings

- General
- Sites
- PHP
- Node
- Expose
- Shortcuts
- Services**
- Mail
- Dumps
- Herd Pro
- About

Services

Add Service

Search

Meilisearch
1.6.2 Port: 7700

Settings Stop

Storage

MinIO
RELEASE.2024-03-05 Port: 9000

Settings Stop

Database

MySQL
8.0.36 Port: 3306

Settings Stop

Cache & Queue

Redis
7.0.0 Port: 6138

Settings Stop

MySQL

Documentation Open

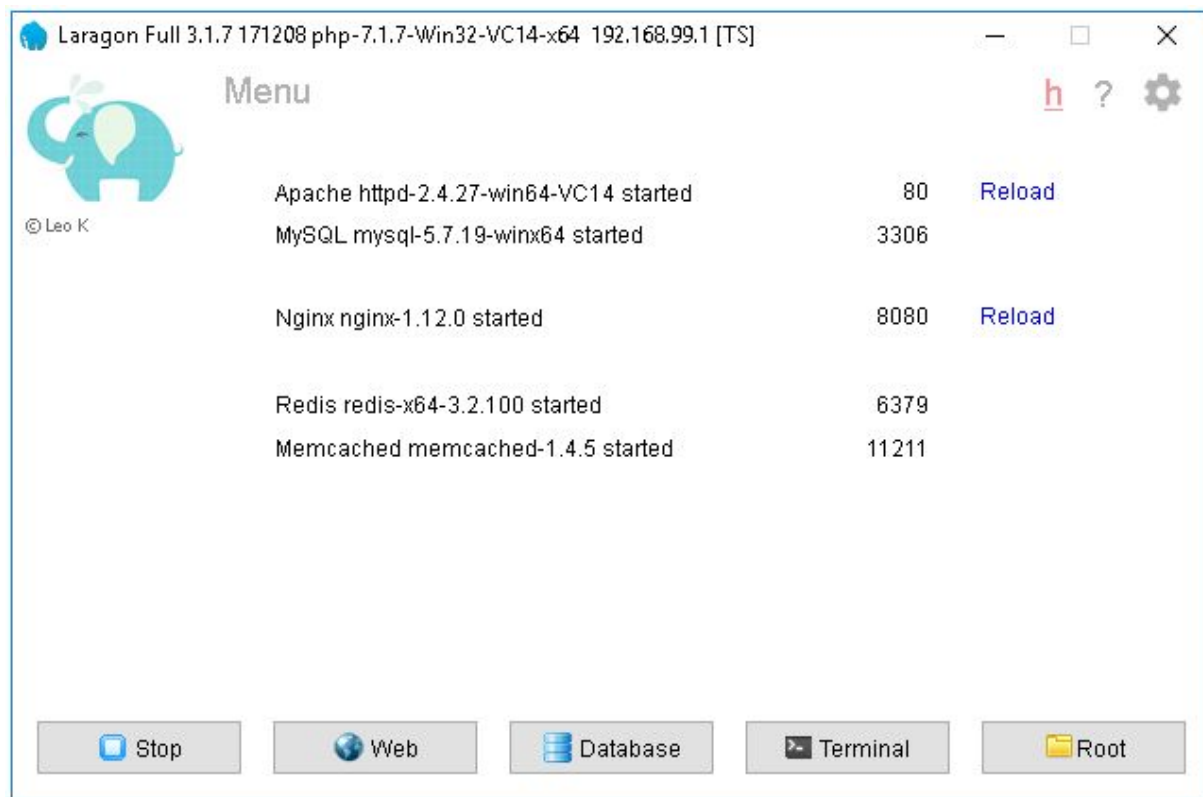
Environment Variables

DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=laravel
DB_USERNAME=root
DB_PASSWORD=

Logs Open

2024-03-25T13:29:31.807200Z 0 [System] [MY-
2024-03-25T13:29:31.824891Z 1 [System] [MY-

Laragon



Crear nuevo proyecto

Via powershell o cmd:

laravel new [Nombre_web]

Nos preguntará si deseamos instalar un **paquete inicial**:

- none (ningún paquete)
- **breeze**: autenticación básica
- **jetstream**: autenticación avanzada (comprobación de email, doble factor...)

Si seleccionamos un paquete también habrá que indicar su stack.

Blade es el motor de plantillas de Laravel.

stack: pila o colección de elementos.




Crear nuevo proyecto

También se debe indicar si se desea soporte para modo oscuro.

Después se debe elegir un **framework para tests**: PHPUnit o **Pest**

Por último, se debe elegir una **base de datos**:

- MySQL ← **RECOMENDADO**
 - PostgreSQL
 - SQLite ← por defecto
 - SQL Server
- 



Q Search

VERSION



Documentation

Laravel [framework](#) has wonderful documentation covering every aspect of the framework. Whether you are a newcomer or have prior experience with Laravel, we recommend reading our documentation from beginning to end. →



Laracasts

Laracasts offers thousands of video tutorials on Laravel, PHP, and JavaScript development. Check them out, see for yourself, and massively level up your development skills in the process. →



Laravel News

Laravel News is a community driven portal and newsletter aggregating all of the latest and most important news in the Laravel ecosystem, including new package releases and tutorials. →



Vibrant Ecosystem

Laravel's robust library of first-party tools and libraries, such as [Forge](#), [Vapor](#), [Nova](#), [Envoyer](#), and [Herd](#) help you take your projects to the next level. Pair them with powerful open source libraries like [Cashier](#), [Dusk](#), [Echo](#), [Horizon](#), [Sanctum](#), [Telescope](#), and more.

Cambiar la Base de Datos por defecto

La BBDD por defecto es SQLite aunque indiquemos que el proyecto trabajará con otra (como MySQL). Para cambiar la BBDD se debe:

1. Abrir el archivo **.env** que se ha generado para el proyecto.
2. Buscar los parámetros de “DB”, como “DB_CONNECTION” y cambiarlos según corresponda. Si trabajamos en local, la configuración para MySQL debería ser:

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=[Nombre_Web]
DB_USERNAME=root
DB_PASSWORD=[Contraseña_root]
```

3. en CMD, ejecutar: *php artisan migrate*
Crear la base de datos si no existe. ¡Ojo! Esto NO migra datos de usuario.

Funcionamiento

1. Cliente realiza petición vía navegador.
2. Laravel enruta la petición entrante según sus características.
3. La petición llega a un controlador.
4. El controlador carga, según corresponda, una vista para ser devuelta al cliente como respuesta.



Rutas en Laravel - Route

Las rutas son lo primero que se carga al acceder a una aplicación web de Laravel.

Estas rutas se definen en la carpeta routes, que contendrá los ficheros de php:

- auth
- console
- web

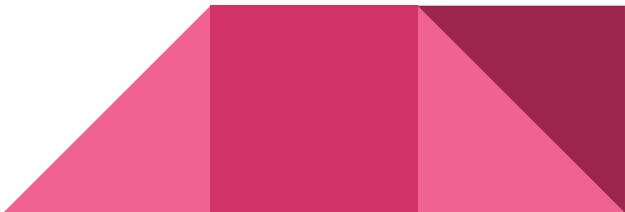
Nos centramos en web.php ya que define cómo se va a desplegar la web al acceder a la misma.



Rutas en Laravel - Route - web.php

Mediante el fichero web.php se define la página web a cargar. Para ello lo primero que se realiza es una llamada a la **clase Route**.

Esta clase de rutas, "Route" ofrece respuestas a peticiones de usuarios mediante métodos propios. Algunos de estos métodos son:


- **get()**
 - **view()**
 - **post()**
 - **patch()**
 - **put()**
 - **delete()**
 - **options()**
 - **match()** - para responder a varias peticiones simultáneamente.
 - **any()** - para responder a todo tipo de peticiones.
- 

Vistas en Laravel

Las respuestas resultantes que se darán a los clientes tras procesar sus peticiones entrantes serán **vistas** (views): **páginas web construidas según la interacción realizada**.

Para construir estas vistas se emplea el motor de plantillas blade (de aquí la extensión ***.blade.php**).

Las vistas generadas por defecto están en la carpeta profile (en el ejemplo; *welcome.blade.php*). Esta carpeta a su vez está en **resources**→**views**. Todas estas vistas se pueden cargar mediante las funciones *get()*, *view()*, *post()*...



Rutas en Laravel - Route - web.php - get()

```
use App\Http\Controllers\ProfileController;
```

```
use Illuminate\Support\Facades\Route;
```

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

El método **get()** tendrá dos parámetros de URL y una página web (view, código HTTP). Sirve para indicar qué vista (view) se debe mostrar según la ruta a la que accede el usuario.



Rutas en Laravel - Route - web.php - view()

Este método también responde a las peticiones de get y head. Esto significa que también se puede escribir:

```
Route::view('/', 'welcome');
```

Útil para casos en los que no se requiere aplicar una lógica de negocio, solo mostrar una web.



Rutas en Laravel - Route - web.php - Nombrar rutas

Es posible dar nombre a las rutas de web.php para trabajar solo con esta nomenclatura al crear enlaces (sin necesidad de indicar la URL específica a la web).

Route::view('/', 'welcome')->name('landing');

Para acceder a esta ruta se puede emplear la referencia: *route('landing')*. Ejemplo:

```
<a href=" <?php echo route('landing') ?> ">Welcome</a>
```

Recomendable siempre dar nombre a las rutas.

Si se intenta llamar a una ruta sin nombre definido, dará error.



Rutas en Laravel - Route - web.php - Orden (avanzado)

El orden con el que se definen las rutas dentro de web.php es **importante** (sobretudo cuando se trabaja con datos traídos de BD y/o variables).

Ejemplo de rutas ordenadas (para /chat/):

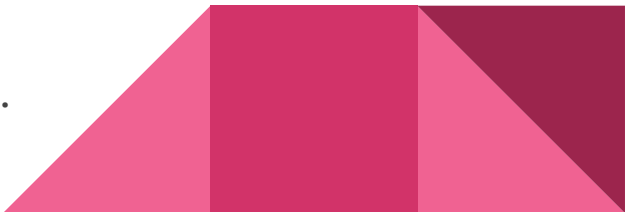
```
Route::get('/', function() { return view('welcome'); });
```

```
Route::get('chat', [ MensajeController::class, 'index' ] )->name('chat.index');
```

```
Route::get('/chat/create', [ MensajeController::class, 'create' ] )->name('chat.create');
```

```
Route::get('/chat/{mensaje}', [ MensajeController::class, 'show' ] )->name('chat.show');
```

{mensaje} contendrá Id's de mensajes. Por lo general,
poner al final las rutas que reciban parámetros variables.



Rutas en Laravel - Route - ver rutas con artisan

Es posible ver un listado de las rutas disponibles por comandos empleando artisan (ubicado en la carpeta de proyecto):

php artisan route:list



Motor de plantillas de Laravel - Blade

Blade Template Engine nos permite, entre otras funcionalidades, escribir código PHP dentro de HTML de forma más clara y descriptiva:

```
<a href="{{ route('landing') }}" >Welcome</a>
```

`{{ y }}` en lugar de `<?php y ?>`.

Blade traduce a PHP automáticamente (compilando las vistas). Se puede ver el resultado en la carpeta `storage/framework/views`.

Laravel identifica archivos de blade cuando su extensión es:
`[NOMBRE_FICHERO].blade.php`



Motor de plantillas de Laravel - Blade - {{ y }}

{{ y }} en lugar de <?php y ?>. ¡OJO! No son exactamente lo mismo.

Blade convierte a:

```
<a href=" <?php echo e(route('landing')) ?> ">Welcome</a>
```

La función **e()** propia de Laravel realiza escape de etiquetas HTML para evitar inyección de código (*htmlspecialchars()*).

Es posible evitar este escape mediante {!! y !!}. La conversión con estos es literal:

```
<a href=" <?php echo route('landing') ?> ">Welcome</a>
```



Motor de plantillas de Laravel - Blade - comentarios

`{{-- para iniciar un comentario y --}}` para cerrar.

Ejemplo:

`{{-- este texto esta comentado --}}`



header.blade.php

```
<ul class="nav">
  <li>Home</li>
  <li>Contact us</li>
  <li>About</li>
</ul>
```

sidebar.blade.php

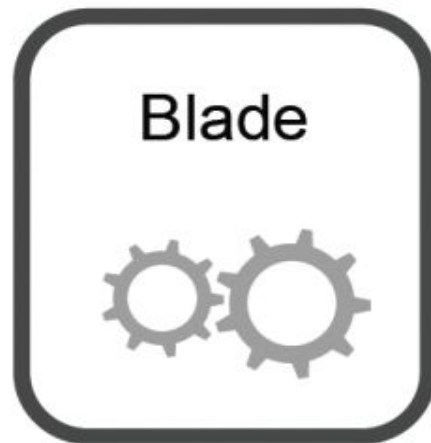
```
<ul class="sidebar">
  <li>Links</li>
  <li>Archive</li>
  <li>Search</li>
</ul>
```

content.blade.php

```
<div class="container">
  <div class="row">
    <div class="col-md-12">
      @foreach($posts as $post)
        <h1>{{ $post->title }}</h1>
        <p>{{ $post->content }}</p>
      @endforeach
    </div>
  </div>
</div>
```

footer.blade.php

```
<div id="footer">
  Copyright. {{ date('Y') }}
</div>
```



```
<ul class="nav">
  <li>Home</li>
  <li>Contact us</li>
  <li>About</li>
</ul>

<div class="container">
  <div class="row">
    <div class="col-md-12">
      <h1>Laravel and templates</h1>

      <p>
        Laravel allows the use of
        templates to separate
        application's views into
        different parts
      </p>

      <h1>Blade layouts</h1>

      <p>
        Layouts make it possible
        to use a single layout
        for all application's
        views
      </p>
    </div>
  </div>

  <ul class="sidebar">
    <li>Links</li>
    <li>Archive</li>
    <li>Search</li>
  </ul>

  <div id="footer">
    Copyright. {{ date('Y') }}
  </div>
```

Motor de plantillas de Laravel - Blade - vistas parciales

Blade permite combinar múltiples plantillas (o código parcial) para convertirlo en un único documento HTML.

Dentro del directorio resources/views, crear carpeta “partials” o emplear “profile.partials” (carpeta existente).

En esta carpeta tendremos **código de PHP que queramos reutilizar**, en ficheros con el formato [nombre].blade.php.

A modo de ejemplo, definimos nav.blade.php que contenga el siguiente código:

```
<a href="{{ route('landing') }}">Welcome</a>
```



Motor de plantillas de Laravel - Blade - vistas parciales

En la vista donde queramos incluir estas “vistas parciales” (código a reutilizar) añadimos un **include de Laravel**:

@include('partials.nav') Este include se puede probar en la vista Welcome.

Este include no requiere tags de `<?php ?>` o similar; Laravel interpreta y añade.

IMPORTANTE: No es necesario indicar la extensión del fichero, pero sí se ha indicado la carpeta “partials” mediante un punto (.)

Si se ha usado la carpeta profile/partials, debe ser

@include('profile.partials.nav')

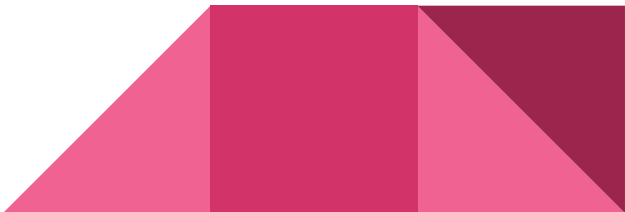


Motor de plantillas de Laravel - Blade - componentes

Los componentes de Blade estan en resources/views/components.

Ejemplo de componente para layout básico de una web, denominado *layout.blade.php*:

```
<!doctype html>
<html>
<head>
    <title>MiWeb</title>
</head>
<body>
    {{ $slot }} ← Variable reservada donde se va a imprimir el contenido variable.
</body>
</html>
```



Motor de plantillas de Laravel - Blade - componentes

Dos opciones para emplear componentes:

1. Usar directiva **@component** (y su cierre):

```
@component('components.layout')  
    echo "Hola mundo";    // todo lo que esté dentro se añadirá a $slot en layout  
@endcomponent
```

2. Usar etiqueta de componentes **<x-layout>** (y su cierre **</x-layout>**):

```
<x-layout>  
    echo "Hola mundo";    // también se añadirá a $slot en layout  
</x-layout>
```

La "x" hace referencia a la carpeta components de Laravel

Motor de plantillas de Laravel - Blade - componentes

Añadir **variables adicionales** a **\$slot** (en layout.blade.php):

```
(..)<body>
{{ $slot }}
<div id="sidebar">
    <div>{{ $sidebar }}</div>
</div>
</body>(..)
```

En este ejemplo, **\$sidebar** es obligatorio.




Motor de plantillas de Laravel - Blade - componentes

Trabajar con **variables adicionales a \$slot** (en la vista) con etiqueta de componentes `<x-layout>`:

```
<x-layout>
    echo "Hola mundo";
    <x-slot:sidebar>    también se puede escribir como <x-slot name="sidebar">
        echo "Sidebar";
    </x-slot:sidebar>
</x-layout>
```

En lugar de escribir las estructuras html, se escribe slots con un nombre de referencia.



Motor de plantillas de Laravel - Blade - componentes

Añadir **variables adicionales a \$slot** (en layout.blade.php), con directiva if para que sean **OPCIONALES**:

```
(..)<body>
{{ $slot }}
@if (isset($sidebar))
    <div id="sidebar">
        <div>{{ $sidebar }}</div>
    </div>
@endif
</body>(..)
```



Motor de plantillas de Laravel - Blade - componentes

Añadir **variables adicionales a \$slot** (en layout.blade.php), con directiva **isset** para que sean **OPCIONALES**:

```
(..)<body>
{{ $slot }}
@isset($sidebar)
    <div id="sidebar">
        <div>{{ $sidebar }}</div>
    </div>
@endisset
</body>(..)
```



Motor de plantillas de Laravel - Blade - componentes

Dar **valor por defecto** a las variables adicionales:

```
<head> (..)
<title>{{ $webTitle ?? "título por defecto" }}</title>
</head>
```

```
(..)<body>
{{ $slot }}
</body>(..)
```

\$webTitle emplea la nomenclatura camelCase.



Motor de plantillas de Laravel - Blade - componentes

Dar **valor** a las variables adicionales mediante x-slot como ya hemos visto:

```
<x-layout>  
    <x-slot:webTitle>  
        Home title  
    </x-slot:webTitle>  
</x-layout>
```



Motor de plantillas de Laravel - Blade - componentes


Dar **valor** a las variables adicionales como si fueran atributos html (útil para cuando son valor sencillos como una cadena de caracteres):

```
<x-layout web-title="Home title">
```

```
</x-layout>
```

web-title ahora está con nomenclatura kebab-case.

La conversión de nombres entre camelCase y kebab-case en este caso se realiza de forma automática internamente.



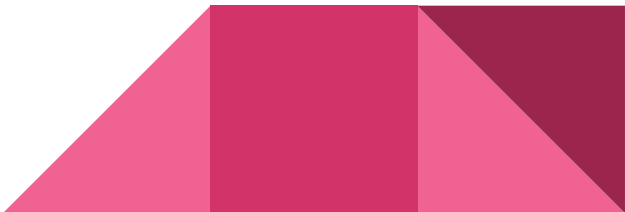
Motor de plantillas de Laravel - Blade - componentes

Dar **valor** a las variables adicionales como si fueran atributos html pero procesar como variables de PHP (mediante " : "):

```
<x-layout :web-title="Home title" :sum="1 + 1">  
</x-layout>
```

en layout:

```
<p>{{ $sum }}</p>      // imprime 2  
(..) {{ $slot }} (..)
```



Motor de plantillas de Laravel - Blade - componentes

Se pueden crear componentes nuevos y llamarlos directamente sin necesidad de emplear la directiva `@include(partial.nav)`. Sí “welcome-link.blade.php” está dentro de la carpeta components, en “layout.blade.php” (componente con la estructura básica de una web) se puede llamar mediante:

```
<x-welcome-link />
```

welcome-link.blade.php puede contener:

```
<p><a href="{{ route('landing') }}"> Welcome </a></p>
```



Rutas en Laravel - Route - Controladores

Para pasar variables mediante Route se puede emplear el tercer parámetro opcional del método view (ejemplo partiendo de web.php):

```
$mensajes = [ ['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje'] ];  
Route::view('chat', 'chat', ['mensajes' => $mensajes ])->name('chat');
```



Rutas en Laravel - Route - Controladores

Para pasar variables mediante Route se puede emplear el segundo parámetro del método get, que será un invocable o una función (ejemplo partiendo de web.php):

```
Route::get('chat', function () {  
    $mensajes = [ ['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje'] ];  
    return view('chat', ['mensajes' => $mensajes ]);  
} )->name('chat');
```



Rutas en Laravel - Route - Controladores

Para inspeccionar estos datos en chat.blade.php:

@dump(\$mensajes)

Equivalente a var_dump() de PHP.



Rutas en Laravel - Route - Controladores

Para iterar sobre los datos pasados a chat.blade.php mediante la directiva *foreach* de Laravel:

```
@foreach( $mensajes as $thisMensaje )  
    <h2> {{ $thisMensaje['titulo'] }} </h2>  
@endforeach
```

También se puede emplear un foreach de PHP.



Rutas en Laravel - Controladores


Para mantener limpio el archivo de rutas web.php, es recomendable utilizar controladores.

Siempre que necesitemos realizar alguna acción entre la petición y la respuesta, es preferible utilizar un controlador.

Un controlador es una clase de PHP que se encarga de recibir la petición y gestionar la respuesta. El método `view()` és un controlador.

Ver función `view()` en la definición de la clase ruta, en `Route.php`: delega sobre “`ViewController`” de Laravel (`ViewController.php`). Solo tiene el método `invoke`.

Los controladores se definen en **app/Http/Controllers**. Por convención la primera letra de cada palabra se escribe en mayúsculas, y siempre termina con “`Controller.php`”.



Rutas en Laravel - Controladores

Ejemplo de Controlador *MensajeController.php*:

```
namespace App\Http\Controllers;  
class MensajeController{  
}
```

En *web.php*:

```
use App\Http\Controllers\MensajeController;  
Route::get('chat', MensajeController::class )->name('chat');
```




Rutas en Laravel - Controladores - `__invoke()`

Ejemplo de Controlador *MensajeController.php* con método `invoke` para hacerlo "invocable":

```
namespace App\Http\Controllers;
class MensajeController{

    public function __invoke(){
        $mensajes = [ ['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje'] ];
        return view('chat', ['mensajes' => $mensajes ]);
    }
}
```



Rutas en Laravel - Controladores - métodos propios

Ejemplo de Controlador *MensajeController.php* con método propio:

```
namespace App\Http\Controllers;
class MensajeController{
    public function index() {
        $mensajes = [ ['titulo' => 'primer mensaje'], ['titulo' => 'segundo mensaje'] ];
        return view('chat', ['mensajes' => $mensajes ]);
    }
}
```

En *web.php*:

```
Route::get('chat', [ MensajeController::class, 'index' ]->name('chat');
```


Los métodos necesarios se pasan con la clase como parámetros en array.



Rutas en Laravel - Controladores - crear con Artisan

Ejemplo de generación de controlador *MensajeController.php* por comandos empleando artisan (ubicados en la carpeta de proyecto):

php artisan make:controller MensajeController


- Añadir **-i** (invoke) para que sea invocable (añade método `__invoke()`)
 - Añadir **-r** (resource) para que incluya los métodos `index()`, `create()`, `store()`, `show()`, `edit()`, `update()`, `destroy()`
 - Opción **--api** para que incluya los métodos `index()`, `store()`, `show()`, `update()`, `destroy()`
- 

Rutas en Laravel - Controladores y Formularios

Continuando con el ejemplo *MensajeController.php*, definir método `create()`:

```
namespace App\Http\Controllers;  
class MensajeController{  
    public function create() {  
        return view('chat.create');  
    }  
}
```

Esto requiere una vista *create.blade.php* creada en una subcarpeta llamada “chat”.



Rutas en Laravel - Controladores y Formularios

El contenido de `create.blade.php` es:

```
<x-layout>
  <h1>Nuevo mensaje</h1>
  <form method="POST" action="{{ route('chat.store') }}">
    @csrf
    <label>Titulo: <input type="text" name="title" /></label>
    <label>Mensaje: <br><textarea name="message"></textarea></label><br>
    <button type="submit">Enviar</button>
  </form>
  <a href="{{ echo route('chat.index') }}">Atras</a>
</x-layout>
```

IMPORTANTE: se está empleando un componente creado anteriormente para reutilizar el código. Su uso no es obligatorio.

Rutas en Laravel - Controladores y Formularios

Definir las rutas para estas vistas en web.php es:

Para mostrar el formulario (con get):

```
Route::get('/chat/create', [ MensajeController::class, 'create' ] )->name('chat.create');
```

Para recoger los datos del formulario (via POST):

```
Route::post('/chat', [ MensajeController::class, 'store' ] )->name('chat.store');
```



Rutas en Laravel - Controladores y Formularios

Añadir el método `store()` a *MensajeController.php*:

```
namespace App\Http\Controllers;  
class MensajeController{  
    public function store(){  
        return request(); // Convierte los datos entrantes automáticamente a json  
    }  
}
```

Nota: no requiere crear la vista `store.blade.php`.



Rutas en Laravel - Controladores y Formularios

Alternativa (buena práctica) para el método `store()` de *MensajeController.php*:

```
namespace App\Http\Controllers;  
namespace Illuminate\Http\Request;  
class MensajeController{  
    public function store(Request $request){  
        return $request;  
    }  
}
```



Rutas en Laravel - Controladores y Formularios

Para acceder a los valores del formulario individualmente, en concreto, al campo de título (title), dentro del método store() de *MensajeController.php*:

```
namespace App\Http\Controllers;
namespace Illuminate\Http\Request;
class MensajeController{
    public function store(Request $request){
        return $request->input('title');
    }
}
```




Rutas en Laravel - Controladores y Formularios - CSRF

La directiva **@csrf** es para prevenir de ataques CSRF. En Laravel, con formulario de método POST se debe añadir siempre.

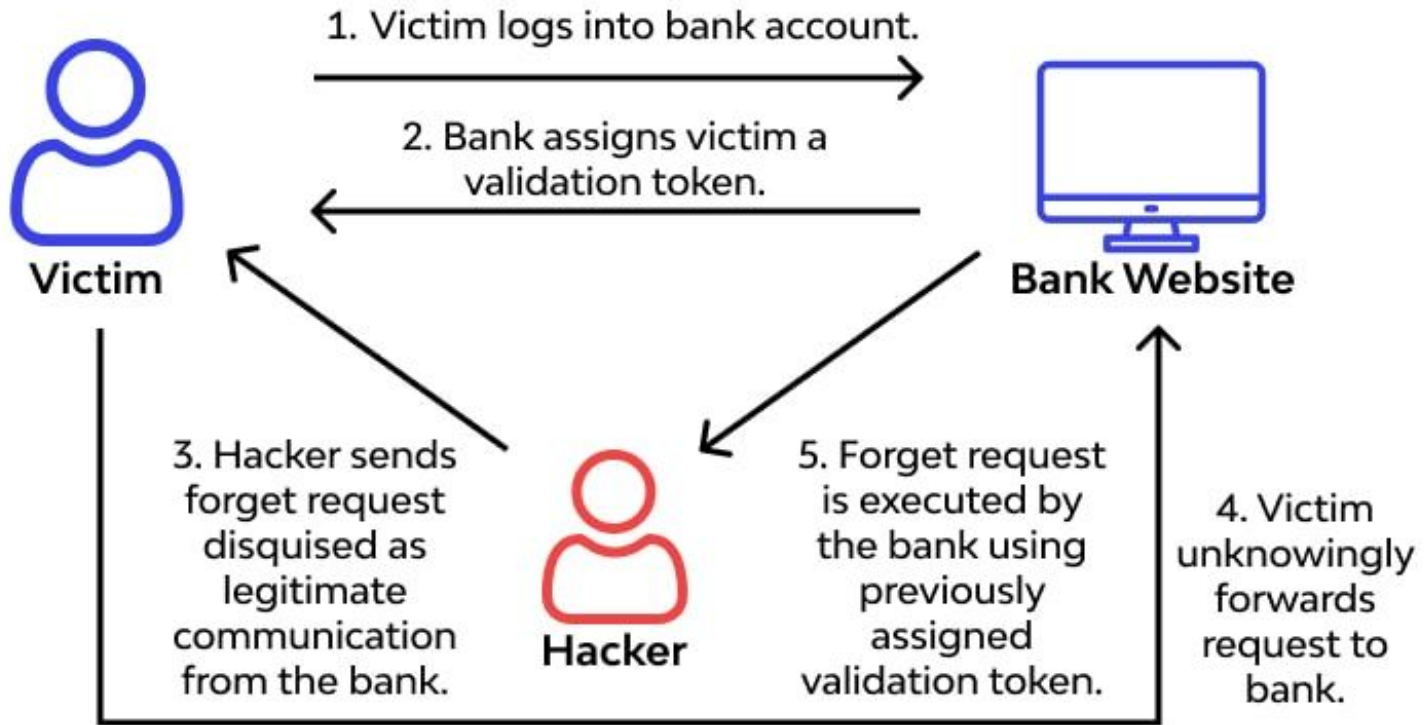
Cross-Site Request Forgery (CSRF) es un tipo de ataque que **engaña a un usuario para que realice acciones en una aplicación web sin su consentimiento o conocimiento**.

Esto puede llevar a acciones no autorizadas como transferir fondos, cambiar contraseñas u otras operaciones no deseadas por el cliente/usuario.

Laravel añade token oculto para verificar origen del POST. Este token es valido durante 2 horas (valor por defecto, configurable vía “.env”). Es posible visualizarlo desde el navegador (inspeccionar).



Cross-Site Request Forgery



Laravel y el MVC (Modelo-Vista-Controlador)

