



DIRECCIÓN DE POSGRADO – CEFORPI

**CENTRO DE ESTUDIOS Y FORMACIÓN DE
POSGRADO E INVESTIGACIÓN**

TRABAJO FINAL MÓDULO 3

DESARROLLO DE UN SISTEMA CRUD CON LOGIN Y COMPOSER

Diplomado en Diseño, Desarrollo y Mantenimiento de Sitios Web

PARTICIPANTE: ROLANDO JAHNSEN TOLEDO YUPANQUI

DOCENTE : ING.: JUAN CARLOS CONDORI ZAPANA

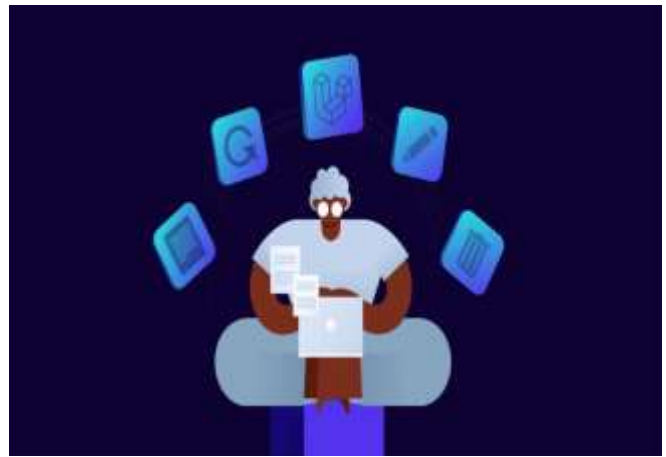
**EL ALTO, LA PAZ – BOLIVIA
Enero – 2025**

TRABAJO FINAL - MODULO 3

Desarrollo de un Sistema CRUD con Login y Composer: elegir el tema diferente desarrollado en la clase. En esta tarea final, deberás desarrollar un sistema web que permita la gestión de usuarios utilizando las funcionalidades de **Login** y un **CRUD** completo (Crear, Leer, Actualizar y Eliminar). El proyecto debe hacer uso de **Composer**, implementando dependencias externas, autoloading y buenas prácticas en PHP.

CÓMO HACER CRUD (CREATE, READ, UPDATE, AND DELETE. CREAR, LEER, ACTUALIZAR Y ELIMINAR) CON LARAVEL

[Laravel](#) es un popular [framework PHP](#) para crear aplicaciones web modernas y dinámicas en el panorama actual de desarrollo web, que evoluciona rápidamente. Una de sus características principales es [Laravel Eloquent](#), un [mapeador objeto-relacional](#) (ORM, Object-relational mapping) que permite a los desarrolladores realizar eficazmente operaciones de creación, lectura, actualización y eliminación (CRUD, create, read, update, and delete) en una base de datos.



Este tutorial muestra cómo realizar estas operaciones en tu aplicación Laravel utilizando el ORM Eloquent de Laravel y cómo desplegar tu aplicación Laravel CRUD utilizando [MyKinsta](#).

Funcionalidad CRUD en Laravel

Las operaciones CRUD son la columna vertebral de cualquier aplicación basada en bases de datos. Te permiten realizar las operaciones de base de datos más básicas y esenciales, como crear nuevos registros, leer los existentes, actualizarlos y eliminarlos. Estas operaciones son cruciales para la funcionalidad de cualquier [aplicación Laravel](#) que interactúe con una base de datos.

Eloquent proporciona una forma sencilla e intuitiva de interactuar con la base de datos, disminuyendo las complejidades de la gestión de bases de datos para que puedas centrarte en construir tu aplicación. Sus métodos y clases incorporados te permiten hacer CRUD fácilmente en los registros en la base de datos.

Requisitos Previos

Para seguir este tutorial, asegúrate de que tienes lo siguiente:

- [XAMPP](#)
- [Composer](#)
- [Una cuenta en MyKinsta](#)
- Una cuenta en [GitHub](#), [GitLab](#) o [Bitbucket](#) para enviar tu código
- [Bootstrap versión 5](#)

Pasos

1. Instala Laravel y crea una nueva aplicación
2. Crea una base de datos
3. Crea una tabla
4. Crea un controlador
5. Configura el modelo
6. Añade una ruta
7. Generar archivos Blade
8. Despliega y prueba tu aplicación CRUD

Para guiarte por el camino, consulta el [código completo](#) del tutorial.

Instalar Laravel y Crear una Nueva Aplicación

Abre tu terminal donde quieras crear tu aplicación Laravel y sigue los pasos que se indican a continuación.

1. Para instalar Laravel, ejecuta

```
composer global require laravel/installer
```

2. Para crear una nueva aplicación Laravel, ejecuta:

```
laravel new crudposts
```

Crear una Base de Datos

Para crear una nueva base de datos para tu aplicación

1. Inicia los servidores Apache y MySQL en el panel de control de XAMPP y visita <http://localhost/phpmyadmin> en tu navegador.
2. Haz clic en **Nueva** en la barra lateral izquierda. Deberías ver lo siguiente:

El formulario de creación de base de datos.

Databases



3. Añade un nombre de base de datos y haz clic en **Crear**.
4. Edita el archivo **.env** de tu aplicación en el root de tu aplicación Laravel. Contiene todas las variables de entorno utilizadas por la aplicación. Localiza las variables prefijadas con **DB_** y edítalas con las credenciales de tu base de datos:

```
DB_CONNECTION=  
DB_HOST=  
DB_PORT=  
DB_DATABASE=  
DB_USERNAME=  
DB_PASSWORD=
```

Crear una Tabla

Las filas de datos de tu aplicación se almacenan en tablas. Sólo necesitas una tabla para esta aplicación, creada utilizando las [migraciones de Laravel](#).

1. Para crear una tabla y generar un archivo de migración utilizando la interfaz de línea de comandos de Laravel, Artisan, ejecuta:

```
php artisan make:migration create_posts_table
```

El comando anterior crea un nuevo archivo,

yyyy_mm_dd_hhmmss_create_posts_table.php, en **base de datos/migraciones**.

2. Abre **yyyy_mm_dd_hhmmss_create_posts_table.php** para definir las columnas que quieres dentro de tu tabla de base de datos en la función up:

```
public function up()  
{  
    Schema::create('posts', function (Blueprint $table) {  
        $table->id();  
        $table->string('title');  
        $table->text('body');  
    });  
}
```

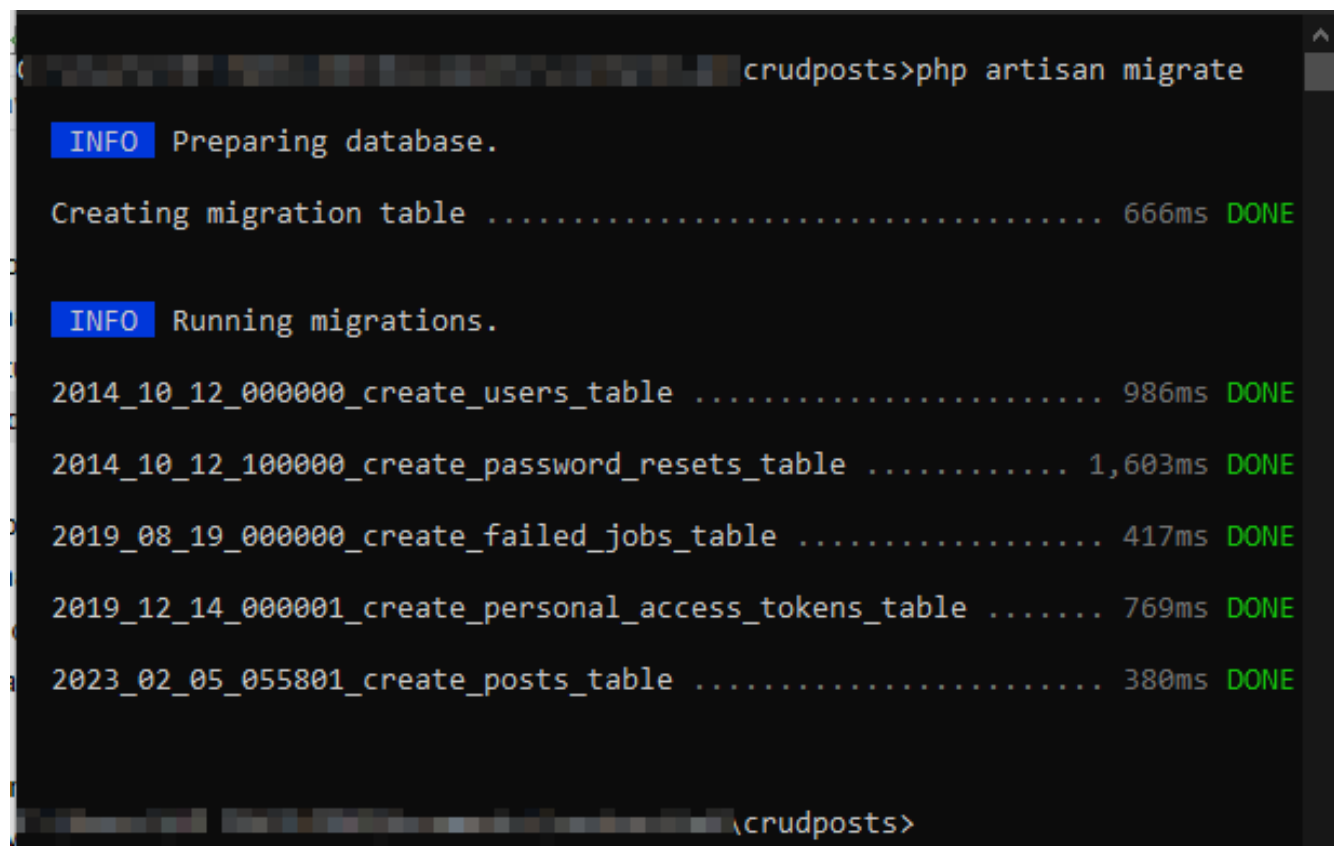
```
$table->timestamps();  
});
```

Este código define lo que contiene la tabla posts. Tiene cuatro columnas:

id, **title**, **body**, y **timestamps**.

3. Ejecuta los archivos de migraciones de la carpeta **database/migrations** para crear tablas en la base de datos:

```
php artisan migrate
```



```
crudposts>php artisan migrate  
  
INFO Preparing database.  
Creating migration table ..... 666ms DONE  
  
INFO Running migrations.  
2014_10_12_000000_create_users_table ..... 986ms DONE  
2014_10_12_100000_create_password_resets_table ..... 1,603ms DONE  
2019_08_19_000000_create_failed_jobs_table ..... 417ms DONE  
2019_12_14_000001_create_personal_access_tokens_table ..... 769ms DONE  
2023_02_05_055801_create_posts_table ..... 380ms DONE  
  
crudposts>
```

El resultado es el siguiente:

Ejecutando las migraciones.

4. Ve a la base de datos que creaste antes para confirmar que has creado las tablas:

Table	Action	Rows	Type	Collation	Size	Overhead
<input type="checkbox"/> failed_jobs	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	32.0 KiB	-
<input type="checkbox"/> migrations	★ Browse Structure Search Insert Empty Drop	5	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> password_resets	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> personal_access_tokens	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	48.0 KiB	-
<input type="checkbox"/> posts	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	16.0 KiB	-
<input type="checkbox"/> users	★ Browse Structure Search Insert Empty Drop	0	InnoDB	utf8mb4_unicode_ci	32.0 KiB	-
6 tables	Sum	5	InnoDB	utf8mb4_general_ci	160.0 KiB	0 B

Tablas creadas.

Crear un Controlador

El controlador contiene todas las funciones para hacer CRUD a las entradas desde la base de datos.

Genera un archivo controlador dentro de tu aplicación Laravel utilizando Artisan:

```
php artisan make:controller PostController --api
```

Ejecutando este comando se crea un archivo **PostController.php** en **app/Http/Controllers**, con código boilerplate y declaraciones de funciones vacías **index**, **store**, **show**, **update**, y **destroy**.

Crear Funciones

A continuación, crea las funciones que almacenan, indexan, actualizan, destruyen, crean, muestran y editan los datos.

Puedes añadirlas al archivo [app/Http/Controller/PostController.php](#) que se muestra a continuación.

La Función **store**

La función **store** añade una entrada a la base de datos.

Desplázate hasta la función `store` y añade el siguiente código dentro de las llaves vacías:

```
$request->validate([
    'title' => 'required|max:255',
    'body' => 'required',
]);
Post::create($request->all());
return redirect()->route('posts.index')
    ->with('success','Post created successfully.');
```

Este código toma un objeto que contiene el título y el cuerpo de la entrada, valida los datos, añade una nueva entrada a la base de datos si los datos son válidos y redirige al usuario a la página de inicio con un mensaje de éxito.

La Función **index**

La función **index** recupera todas las entradas de la base de datos y envía los datos a la página [posts.index](#).

La Función **update**

La función **update** contiene el **id** del post a actualizar, el nuevo post **title**, y el **body**. Tras validar los datos, busca el post con el mismo **id**. Si la encuentra, la función **update** actualiza el post en la base de datos con los nuevos **title** y **body**. A continuación, redirige al usuario a la página de inicio con un mensaje de éxito.

La Función **destroy**

La función **destroy** encuentra una entrada con el **id** dado y la borra de la base de datos, luego redirige al usuario a la página de inicio con un mensaje de éxito.

Las funciones anteriores son las que se utilizan para hacer CRUD a las entradas de la base de datos. Sin embargo, debes definir más funciones dentro del controlador para mostrar las páginas necesarias en **resources/views/posts/**.

La Función **create**

La función **create** renderiza la página [resources/views/posts/crear.hoja.php](#) que contiene el formulario para añadir entradas a la base de datos.

La Función **show**

La función **show** encuentra una entrada con el nombre **id** en la base de datos y muestra la página [resources/views/posts/mostrar.blade.php](#) con la entrada.

La Función edit

La función `edit` busca una entrada con el nombre `id` en la base de datos y muestra el archivo <resources/views/posts/edit.blade.php> con los detalles de la entrada dentro de un formulario.

Configurar el Modelo

El modelo `Post` interactúa con la tabla `posts` de la base de datos.

1. Para crear un modelo con Artisan, ejecuta

```
php artisan make:model Post
```

Este código crea un archivo `Post.php` dentro de la carpeta `App/Models`.

2. Crea un array `<fillable>`. Añade el siguiente código dentro de la clase `Post` y debajo de la línea `use HasFactory;`

```
protected $fillable = [  
    'title',  
    'body',  
];
```

Este código crea un array `fillable` que te permite añadir elementos a la base de datos desde tu aplicación Laravel.

3. Conecta el modelo `Post` al archivo `PostController.php`. Abre `PostController.php` y añade la siguiente línea en `use IlluminateHttpRequest;`. Tiene el siguiente aspecto

```
use IlluminateHttpRequest;  
use AppModelsPost;
```

El archivo `PostController.php` ahora debería tener este aspecto:

```
<?php  
namespace AppHttpControllers;  
use IlluminateHttpRequest;  
use AppModelsPost;  
class PostController extends Controller  
{  
    /**  
     * Display a listing of the resource.  
     *  
     * @return IlluminateHttpResponse
```



```

*/
public function index()
{
    $posts = Post::all();
    return view('posts.index', compact('posts'));
}
/**
 * Store a newly created resource in storage.
 *
 * @param IlluminateHttpRequest $request
 * @return IlluminateHttpResponse
 */
public function store(Request $request)
{
    $request->validate([
        'title' => 'required|max:255',
        'body' => 'required',
    ]);
    Post::create($request->all());
    return redirect()->route('posts.index')
        ->with('success', 'Post created successfully.');
```

```

    }
    /**
     * Update the specified resource in storage.
     *
     * @param IlluminateHttpRequest $request
     * @param int $id
     * @return IlluminateHttpResponse
     */
    public function update(Request $request, $id)
    {
        $request->validate([
            'title' => 'required|max:255',
            'body' => 'required',
        ]);
        $post = Post::find($id);
        $post->update($request->all());
        return redirect()->route('posts.index')
            ->with('success', 'Post updated successfully.');
```

```

    }
    /**
     * Remove the specified resource from storage.
     *
     * @param int $id
     * @return IlluminateHttpResponse
     */
    public function destroy($id)

```

```

{
$post = Post::find($id);
$post->delete();
return redirect()->route('posts.index')
->with('success', 'Post deleted successfully');
}
// routes functions
/**
 * Show the form for creating a new post.
 *
 * @return IlluminateHttpResponse
 */
public function create()
{
return view('posts.create');
}
/**
 * Display the specified resource.
 *
 * @param int $id
 * @return IlluminateHttpResponse
 */
public function show($id)
{
$post = Post::find($id);
return view('posts.show', compact('post'));
}
/**
 * Show the form for editing the specified post.
 *
 * @param int $id
 * @return IlluminateHttpResponse
 */
public function edit($id)
{
$post = Post::find($id);
return view('posts.edit', compact('post'));
}
}

```

Añadir Rutas

Después de crear las funciones del controlador y el modelo **Post**, debes añadir rutas para las funciones de tu controlador.

1. Abre **routes/web.php** y elimina la ruta boilerplate que generó la aplicación. Sustitúyela por el código que aparece a continuación para conectar las funciones del controlador a sus respectivas rutas:

```
// returns the home page with all posts
Route::get('/', PostController::class . '@index')->name('posts.index');
// returns the form for adding a post
Route::get('/posts/create', PostController::class . '@create')->name('posts.create');
// adds a post to the database
Route::post('/posts', PostController::class . '@store')->name('posts.store');
// returns a page that shows a full post
Route::get('/posts/{post}', PostController::class . '@show')->name('posts.show');
// returns the form for editing a post
Route::get('/posts/{post}/edit', PostController::class . '@edit')->name('posts.edit');
// updates a post
Route::put('/posts/{post}', PostController::class . '@update')->name('posts.update');
// deletes a post
Route::delete('/posts/{post}', PostController::class . '@destroy')->name('posts.destroy');
```

2. Para conectar las rutas, abre **app/Http/Controllers/PostController.php** y añade la línea siguiente debajo de la línea **use IlluminateSupportFacadesRoute;**

```
use IlluminateSupportFacadesRoute;
use AppHttpControllersPostController;
```

El archivo **routes/web.php** debería tener ahora este aspecto:

```
<?php
use IlluminateSupportFacadesRoute;
use AppHttpControllersPostController;
// returns the home page with all posts
Route::get('/', PostController::class . '@index')->name('posts.index');
// returns the form for adding a post
Route::get('/posts/create', PostController::class . '@create')->name('posts.create');
// adds a post to the database
Route::post('/posts', PostController::class . '@store')->name('posts.store');
// returns a page that shows a full post
Route::get('/posts/{post}', PostController::class . '@show')->name('posts.show');
// returns the form for editing a post
Route::get('/posts/{post}/edit', PostController::class . '@edit')->name('posts.edit');
// updates a post
```

```
Route::put('/posts/{post}', PostController::class . '@update')-
>name('posts.update');
// deletes a post
Route::delete('/posts/{post}', PostController::class . '@destroy')-
>name('posts.destroy');
```

Generar Archivos Blade

Ahora que tienes las rutas, puedes crear los archivos [Blade de Laravel](#). Antes de utilizar Artisan para generar los archivos Blade, crea el comando **make:view**, con el que podrás generar los archivos **blade.php**.

1. Ejecuta el siguiente código en la CLI para crear un archivo de comando **MakeViewCommand** dentro de la carpeta **app/Console/Commands** :

```
php artisan make:command MakeViewCommand
```

2. Crea un comando para generar archivos **.blade.php** desde la CLI sustituyendo el código del archivo **MakeViewCommand** por lo siguiente:

```
<?php
namespace App\ConsoleCommands;
use Illuminate\ConsoleCommand;
use File;
class MakeViewCommand extends Command
{
    /**
     * The name and signature of the console command.
     *
     * @var string
     */
    protected $signature = 'make:view {view}';
    /**
     * The console command description.
     *
     * @var string
     */
    protected $description = 'Create a new blade template.';
    /**
     * Execute the console command.
     *
     * @return mixed
     */
    public function handle()
    {
        $view = $this->argument('view');
        $path = $this->viewPath($view);
```

```

        $this->createDir($path);
        if (File::exists($path))
        {
            $this->error("File {$path} already exists!");
            return;
        }
        File::put($path, $path);
        $this->info("File {$path} created.");
    }
    /**
     * Get the view full path.
     *
     * @param string $view
     *
     * @return string
     */
    public function viewPath($view)
    {
        $view = str_replace('.', '/', $view) . '.blade.php';
        $path = "resources/views/{$view}";
        return $path;
    }
    /**
     * Create a view directory if it does not exist.
     *
     * @param $path
     */
    public function createDir($path)
    {
        $dir = dirname($path);
        if (!file_exists($dir))
        {
            mkdir($dir, 0777, true);
        }
    }
}

```

Crear una Página de Inicio

A continuación, crea tu página de inicio. La página de inicio es el archivo **index.blade.php**, que enumera todas las entradas.

1. Para crear la página de inicio, ejecuta

```
php artisan make:view posts.index
```

Esto crea una carpeta **posts** dentro de la carpeta **/resources/views** y un archivo **index.blade.php** debajo de ella. La ruta resultante es **/resources/views/posts/index.blade.php**.

2. Añade el siguiente código dentro del archivo **index.blade.php**:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-
alpha1/dist/css/bootstrap.min.css" rel="stylesheet"
integrity="sha384-
GLh1TQ8iRABdZLL1603oVMWSktQOp6b7In1Zl3/Jr59b6EGGoI1aFkw7cmDA6j6gD"
crossorigin="anonymous">
  <title>Posts</title>
</head>
<body>
  <nav class="navbar navbar-expand-lg navbar-light bg-warning">
    <div class="container-fluid">
      <a class="navbar-brand h1" href={{ route('posts.index')
}}>CRUDPosts</a>
      <div class="justify-end ">
        <div class="col ">
          <a class="btn btn-sm btn-success" href={{ route('posts.create')
}}>Add Post</a>
        </div>
      </div>
    </div>
  </nav>
  <div class="container mt-5">
    <div class="row">
      @foreach ($posts as $post)
        <div class="col-sm">
          <div class="card">
            <div class="card-header">
              <h5 class="card-title">{{ $post->title }}</h5>
            </div>
            <div class="card-body">
              <p class="card-text">{{ $post->body }}</p>
            </div>
            <div class="card-footer">
              <div class="row">
                <div class="col-sm">
                  <a href="{{ route('posts.edit', $post->id) }}"
```

```

        class="btn btn-primary btn-sm">Edit</a>
    </div>
    <div class="col-sm">
        <form action="{{ route('posts.destroy', $post->id) }}"
method="post">
            @csrf
            @method('DELETE')
            <button type="submit" class="btn btn-danger btn-
sm">Delete</button>
        </form>
    </div>
</div>
</div>
</div>
</div>
</div>
@endforeach
</div>
</div>
</body>
</html>

```

El código anterior crea una página HTML sencilla que utiliza [Bootstrap](#) para el estilo. Establece una barra de navegación y una plantilla de cuadrícula que enumera todas las entradas de la base de datos con detalles y dos botones de acción — editar y eliminar — utilizando el ayudante `@foreach` Blade.

El botón **Edit** lleva al usuario a la página **Edit post**, donde puede editar la entrada. El botón **Delete** borra la entrada de la base de datos utilizando `{{ route('posts.destroy', $post->id) }}` con un método `DELETE`.

Nota: El código de la barra de navegación de todos los archivos es el mismo que el del archivo anterior.

3. Crea la página **create.blade.php**. El archivo Blade llamado **create** añade entradas a la base de datos. Utiliza el siguiente comando para generar el archivo:

```
php artisan make:view posts.create
```

Esto genera un archivo **create.blade.php** dentro de la carpeta `/resources/views/posts`.

4. Añade el siguiente código al archivo **create.blade.php**:

```

// same as the previous file. Add the following after the nav tag and
before the closing body tag.
<div class="container h-100 mt-5">

```

```

<div class="row h-100 justify-content-center align-items-center">
  <div class="col-10 col-md-8 col-lg-6">
    <h3>Add a Post</h3>
    <form action="{{ route('posts.store') }}" method="post">
      @csrf
      <div class="form-group">
        <label for="title">Title</label>
        <input type="text" class="form-control" id="title" name="title"
required>
      </div>
      <div class="form-group">
        <label for="body">Body</label>
        <textarea class="form-control" id="body" name="body" rows="3"
required></textarea>
      </div>
      <br>
      <button type="submit" class="btn btn-primary">Create Post</button>
    </form>
  </div>
</div>
</div>

```

El código anterior crea un formulario con los campos **title** y **body** y un botón **submit** para añadir una entrada a la base de datos a través de la acción `{{ route('posts.store') }}` con un método POST.

5. Crea la página **Edit post** para editar entradas en la base de datos. Utiliza el siguiente comando para generar el archivo:

```
php artisan make:view posts.edit
```

Esto crea un archivo **edit.blade.php** dentro de la carpeta `/resources/views/posts`.

6. Añade el siguiente código al archivo **edit.blade.php**:

```

<div class="container h-100 mt-5">
  <div class="row h-100 justify-content-center align-items-center">
    <div class="col-10 col-md-8 col-lg-6">
      <h3>Update Post</h3>
      <form action="{{ route('posts.update', $post->id) }}" method="post">
        @csrf
        @method('PUT')
        <div class="form-group">
          <label for="title">Title</label>
          <input type="text" class="form-control" id="title" name="title"
value="{{ $post->title }}" required>
        </div>
      </form>
    </div>
  </div>
</div>

```



```

    </div>
    <div class="form-group">
        <label for="body">Body</label>
        <textarea class="form-control" id="body" name="body" rows="3"
required>{{ $post->body }}</textarea>
    </div>
    <button type="submit" class="btn mt-3 btn-primary">Update
Post</button>
    </form>
</div>
</div>
</div>

```

El código anterior crea un formulario con los campos **title** y **body** y un botón de envío para editar una entrada con el **id** especificado en la base de datos a través de la acción **{{ route('posts.update') }}** con un método **PUT**.

7. A continuación, reinicia tu servidor de aplicaciones utilizando el código siguiente:

```
php artisan serve
```

Visita **http://127.0.0.1:8000** en tu navegador para ver tu nuevo blog. Haz clic en el botón **Add post** para añadir nuevas entradas.

Despliega y Prueba tu Aplicación CRUD

Prepara tu aplicación para el despliegue como se indica a continuación.

1. Haz que el despliegue sea fácil y sencillo declarando la carpeta pública. Añade un archivo **.htaccess** a el root de la carpeta de la aplicación con el siguiente código:

```

<IfModule mod_rewrite.c >
    RewriteEngine On
    RewriteRule ^(.*)$ public/$1 [L]
</IfModule >

```

2. Obliga a tu aplicación a utilizar HTTPS añadiendo el siguiente código encima de tus rutas dentro del archivo **routes/web.php**:

```

use Illuminate\Support\Facades\URL;
URL::forceScheme('https');

```

3. Envía tu código a un repositorio Git. Kinsta soporta despliegues desde GitHub, GitLab o Bitbucket.

Configurar un Proyecto en MyKinsta

1. Crea una cuenta [MyKinsta](#) si aún no tienes una.
2. Accede a tu cuenta y haz clic en el botón **Añadir Servicio** en el Panel de Control para crear una nueva aplicación.
3. Si eres nuevo en la aplicación, conéctate a tu cuenta de GitHub, GitLab o Bitbucket y concede permisos específicos.
4. Rellena el formulario, y añade el **APP_KEY**. Puedes encontrar su valor correspondiente en tu archivo **.env**.
5. Selecciona tus recursos de construcción y si quieres utilizar la ruta de construcción de tu aplicación o construir tu aplicación con Dockerfile. Para esta demostración, deja que MyKinsta construya la aplicación basándose en tu repositorio.
6. Especifica los diferentes procesos que quieres ejecutar durante el despliegue. Puedes dejarlo en blanco en este punto.
7. Por último, añade tu método de pago.

Después de confirmar tu método de pago, MyKinsta despliega tu aplicación y te asigna una URL como se muestra a continuación:

Deployment progress



Build process completed


Mar 5, 2023, 5:01 PM



Rollout process completed

Mar 5, 2023, 5:02 PM



Deployment available at crudposts-94z41.kinsta.app 

[View runtime logs](#)

Mar 5, 2023,
5:02 PM



Puedes visitar el enlace, pero obtendrás una página 500 | Server Error porque la aplicación necesita una conexión de base de datos válida para funcionar. La siguiente sección resuelve este problema.

Crear una Base de Datos a través de MyKinsta

1. Para crear una base de datos, ve a tu panel de MyKinsta y haz clic en **Añadir Servicio**.
2. Selecciona **Base de Datos** y rellena el formulario con el nombre, tipo, nombre de usuario y contraseña que prefieras para la base de datos. Añade una ubicación de centro de datos y un tamaño de base de datos que se ajuste a tu aplicación.
3. La página siguiente muestra el resumen de costes y tu forma de pago. Haz clic en **Crear Base de Datos** para completar el proceso.
4. Tras crear la base de datos, MyKinsta te redirige a tu lista de servicios. Haz clic en la base de datos que acabas de crear y desplázate hasta **Conexiones Externas**. Copia las credenciales de la base de datos.
5. Vuelve a la página de **Despliegue** de la aplicación y haz clic en **Configuración**. A continuación, desplázate hasta **Variables de Entorno** y haz clic en **Añadir Variables de Entorno**. Añade las credenciales de la base de datos como variables de entorno en el siguiente orden:

```
DB_CONNECTION=mysql
DB_HOST=External hostname
DB_PORT=External port
DB_DATABASE=Database name
DB_USERNAME=Username
DB_PASSWORD=Password
```

La lista de variables de entorno de la aplicación debería tener ahora este aspecto:

<input type="checkbox"/>	Key ↑	Value	Available	
<input type="checkbox"/>	APP_KEY	Build & Runtime	
<input type="checkbox"/>	APP_URL	Build & Runtime	
<input type="checkbox"/>	DB_CONNECTION	Build & Runtime	
<input type="checkbox"/>	DB_DATABASE	Build & Runtime	
<input type="checkbox"/>	DB_HOST	Build & Runtime	
<input type="checkbox"/>	DB_PASSWORD	Build & Runtime	
<input type="checkbox"/>	DB_PORT	Build & Runtime	
<input type="checkbox"/>	DB_USERNAME	Build & Runtime	

6. Ve a la página de **Despliegue** de tu aplicación y despliega manualmente tu aplicación haciendo clic en **Desplegar Ahora** para aplicar estos cambios. Hasta ahora, has creado una base de datos y la has conectado a tu aplicación.
7. Por último, para crear las tablas de la base de datos en tu base de datos MyKinsta, conecta la base de datos a tu aplicación local actualizando tu archivo **.env** con las mismas credenciales que introdujiste en tu aplicación en MyKinsta y ejecuta el siguiente comando:

```
php artisan migrate
```

Este comando ejecuta todos los archivos de migración. Crea todas las tablas definidas en tu aplicación MyKinsta.

Ahora, se puede probar la aplicación con la URL asignada tras el primer despliegue.

Resumen

Laravel es un framework completo para crear aplicaciones web robustas y escalables que requieren funcionalidad CRUD. Con su sintaxis intuitiva y sus potentes características, Laravel facilita la construcción de operaciones CRUD en tu aplicación.

En este artículo se han tratado los conceptos fundamentales de las operaciones CRUD y cómo implementarlas utilizando las funciones integradas de Laravel. También explica:

- Cómo crear una base de datos en MyKinsta y conectarla a tu aplicación
- Cómo utilizar las migraciones de Laravel para definir la tabla de la base de datos, crear el archivo controlador y sus funciones
- Definir un modelo y conectarlo al controlador. El enrutamiento de Laravel genera archivos Blade para crear las páginas y formularios correspondientes y desplegar y probar la aplicación utilizando MyKinsta

Ahora que has visto lo fácil que es realizar operaciones CRUD en Laravel, echa un [vistazo a MyKinsta](#) para el desarrollo y [alojamiento](#) de aplicaciones web.

Potencia tu sitio con el alojamiento Administrado de WordPress de Kinsta, diseñado para ofrecer velocidad, seguridad y simplicidad. Con Kinsta, obtienes:

- Control simplificado a través del panel MyKinsta
- Migraciones gratuitas ilimitadas, gestionadas por nuestro equipo de expertos en migraciones
- Soporte 24/7/365 de expertos en WordPress
- Infraestructura premium de Google Cloud
- Seguridad de nivel empresarial mediante la integración con Cloudflare
- Alcance mundial con 37 centros de datos

FUENTES CONSULTADOS:

- <https://kinsta.com/es/blog/laravel-crud/>
- <https://www.youtube.com/watch?v=4EHTzl2oEqM>
- <https://lab.wallarm.com/what/crud-crear-leer-actualizar-y-eliminar/?lang=es>
- <https://www.cloudways.com/blog/laravel-crud/>

