

PRIMEROS PASOS CON **INERTIA**

Laravel

ANDRÉS CRUZ YORIS
DESARROLLADOR WEB

Primeros pasos con Laravel Inertia

Aquí continúa tu camino en el desarrollo de aplicaciones web en Laravel 12
con Inertia 2

Andrés Cruz Yoris

Esta versión se publicó: 2025-03-18

¡Postea sobre el libro!

Por favor ayuda a promocionar este libro.

El post sugerido para este libro es:

¡Acabo de comprar el libro "Primeros pasos con Laravel Inertia" de @LibreDesarrollo!

Hazte con tu copia en:

<https://www.desarrollolibre.net/libros/primeros-pasos-laravel-inertia>

Sobre el autor

Este libro fue elaborado por Andrés Cruz Yoris, Licenciado en Computación, con más de 10 años de experiencia en el desarrollo de aplicaciones web en general; trabajo con PHP, Python y tecnologías del lado del cliente como HTML, JavaScript, CSS, Vue entre otras; y del lado del servidor como Laravel, Flask, Django y CodeIgniter. También soy desarrollador en Android Studio, xCode y Flutter para la creación de aplicaciones nativas para Android e IOS.

Pongo a tú disposición parte de mi aprendizaje, reflejado en cada una de las palabras que componen este libro, mi tercer libro sobre Laravel; en este caso más avanzado y viene siendo el desarrollo con Laravel Inertia.

Copyright

Ninguna parte de este libro puede ser reproducido o transmitido de ninguna forma; es decir, de manera electrónica o por fotocopias sin permiso del autor.

Prólogo

Laravel es un framework fascinante, inmenso y con una curva de aprendizaje algo elevada y con múltiples opciones; este libro no es para iniciales y da por hecho de que ya sabes programar en Laravel.

Laravel Inertia lleva el desarrollo de Laravel un poco más allá; automatizando procesos rutinarios en esquemas muy flexibles en base a componentes, los componentes en Vue, que vienen a reemplazar a las vistas blade de Laravel y que con los mismos, podemos comunicar cliente y servidor de una manera muy simple, eficiente y sencilla.

Laravel Inertia no es un framework, es solamente una capa o scaffolding que agrega ciertas características extras al framework que podemos utilizar para crear grandes aplicaciones, con menos esfuerzo y tiempo de desarrollo.

Para quién es este libro

Este libro está dirigido a cualquiera que quiera comenzar a desarrollar con Laravel Inertia cuya principal ventaja es que, podemos usar componentes de Vue, como si fueran vistas de blade y con la ventaja agregada de tener todo el poderío de Vue junto con Laravel.

Este libro no se recomienda a aquellas personas que no hayan trabajado con Laravel básico o sin ningún agregado, si es tu caso, te aconsejo que primero conozcas y practiques con Laravel antes de comenzar; en mi sitio web encontrarás cursos, libros, publicaciones, vídeos y en general más información sobre Laravel.

Para aquellas personas que quieran aprender algo nuevo, conocer sobre una herramienta con poca documentación y la mayoría de ella está en inglés.

Para las personas que quieran mejorar una habilidad en el desarrollo web, que quieran crecer como desarrollador.

Con que te identifiques al menos con alguno de los puntos señalados anteriormente, este libro es para ti.

Consideraciones

Recuerda que cuando veas el símbolo de \$ es para indicar comandos en la terminal; este símbolo no lo tienes que escribir en tu terminal, es una convención que ayuda a saber que estás ejecutando un comando.

Al final de cada capítulo, tienes el enlace al código fuente para que lo compares con tu código.

El uso de las **negritas** en los párrafos tiene dos funciones:

1. Si colocamos una letra en **negritas** es para resaltar algún código como nombre de variables, tablas o similares.
2. Resaltar partes o ideas importantes.

Para mostrar tips usamos el siguiente diseño:

Tips importantes

Para los fragmentos de código:

```
$ npm run dev
```

Al emplear los *** o //***

Significa que estamos indicando que en el código existen fragmentos que presentamos anteriormente.

La página oficial del framework viene siendo:

<https://laravel.com/>

De la estructura del proyecto:

<https://jetstream.laravel.com/2.x/stacks/inertia.html>

De lo que nos ofrece Laravel Inertia al momento de desarrollar:

<https://inertiajs.com/>

La cual es fundamental para seguir este libro y con esto conocer más aspectos del framework.

Fe de errata y comentarios

Si tienes alguna duda, recomendación o encontraste algún error, puedes hacerla saber en el siguiente enlace:

<https://www.desarrollolibre.net/contacto/create>

Por mi correo electrónico:

desarrollolibre.net@gmail.com

O por Discord en el canal de Laravel:

<https://discord.gg/sq85Zqwz96>

Como recomendación, antes de reportar un posible problema, verifica la versión a la que te refieres y lo agregas en tu comentario; la misma se encuentra en la segunda página del libro.

Introducción

Esta guía tiene la finalidad de dar los primeros pasos con Laravel Inertia; con esto, vamos a plantear dos cosas:

1. No es un libro que tenga por objetivo conocer al 100% Laravel Inertia, si no sus componentes principales en base a ejemplos.
2. Se da por hecho de que el lector tiene conocimientos al menos básicos sobre Laravel.

Laravel, al ser un framework más completo al cual te quieras enfrentar que en la práctica es que, tienen muchos más componentes con los cuales trabajar, se da por hecho que el lector tiene cierto conocimiento básico sobre cómo funciona el framework, como el uso o teoría de para qué funcionan las migraciones, el MVC, rutas, entre otras; no es necesario que sepas cómo manejarlas, pero sí que entiendas la lógica detrás de todo esto; si no los tienes, te recomiendo que veas mi primer libro de programación web en el cual damos los primeros pasos con CodeIgniter, el cual es un framework estupendo con muchas coincidencias con Laravel, y al ser un framework más pequeño y sencillo de manejar resulta más fácil de iniciar tu aprendizaje.

Finalmente; en comparación con otros libros, el enfoque será un poco más acelerado o general cuando aborde las explicaciones de los elementos que conforman el framework; y esto es así por dos aspectos principales:

1. Quiero abordar la mayor cantidad de características de Laravel Inertia sin alargar de más el libro.
2. Esto no es un libro recomendado si es el primer framework PHP de este tipo al cual te enfrentas, por lo tanto, siendo así, ya deberías de conocer estos aspectos de la estructura del framework.

Para seguir este libro necesitas tener una computadora con Windows, Linux o MacOS.

Recuerda que el libro actualmente se encuentra en desarrollo.

Mapa

Este libro tiene un total de 17 capítulos, se recomienda que leas en el orden en el cual están dispuestos y a medida que vayamos explicando los componentes del framework, vayas directamente a la práctica, repliques, pruebes y modifiques los códigos que mostramos en este libro.

Capítulo 1: En este capítulo vamos a conocer la herramienta de Laravel Inertia y que nos ofrece al momento del desarrollo de aplicaciones web.

Capítulo 2: En este capítulo vamos a crear un proyecto en Laravel Inertia.

Capítulo 3: En este capítulo vamos a conocer las características que cuenta un proyecto en Laravel Inertia, tanto la estructura que trae por defecto un proyecto, como su funcionamiento base.

Capítulo 4: En este capítulo vamos a crear el típico CRUD para conocer las bases de Laravel Inertia y su comunicación con componentes en Vue.

Capítulo 5: En este capítulo vamos a conocer el uso de las redirecciones y mensajes flash.

Capítulo 6: En este capítulo vamos a crear el proceso CRUD para los posts, tomando todos los temas tratados en los anteriores capítulos.

Capítulo 7: En este capítulo vamos a conocer el proceso de upload en Laravel Inertia y usando plugins de terceros.

Capítulo 8: En este capítulo vamos a instalar plugins de terceros usando Vue, específicamente, el plugin de CKEditor.

Capítulo 9: En este capítulo vamos a conocer los diálogos de confirmación y mensajes tipo toast de Laravel Inertia y usando plugins de terceros.

Capítulo 10: En este capítulo vamos a conocer la comunicación entre componentes de componentes usados directamente desde Laravel, para eso, crearemos un formulario paso por paso.

Capítulo 11: En este capítulo vamos a implementar filtros y un campo de búsqueda para un listado desde el módulo de administración.

Capítulo 12: En este capítulo vamos a implementar la ordenación de columnas de una tabla en un listado desde el módulo de administración.

Capítulo 13: En este capítulo vamos a crear el módulo de blog de vista al usuario final, un listado y página de detalle.

Capítulo 14: En este capítulo vamos a crear un carrito de compras, que incluye el típico CRUD y pantallas.

Capítulo 15: En este capítulo vamos a explicar algunas opciones imprescindibles que podemos usar al momento de enviar peticiones mediante el objeto de Inertia.

Capítulo 16: En este capítulo vamos a crear una aplicación tipo to do list tipo CRUD y con reordenación vía Drag and Drop.

Capítulo 17: Conoceremos cómo crear pruebas unitarias y de integración en la aplicación que creamos anteriormente mediante PHPUnit.

Tabla de Contenido

| | |
|---|-----------|
| Primeros pasos con Laravel Inertia | 2 |
| ¡Tuitea sobre el libro! | 3 |
| Sobre el autor | 4 |
| Copyright | 5 |
| Mapa | 11 |
| Capítulo 1: Sobre Laravel Inertia | 1 |
| Jetstream | 1 |
| Inertia | 1 |
| ¿Qué nos ofrece Inertia? | 1 |
| Ventajas y desventajas de Inertia | 3 |
| Capítulo 2: Crear un proyecto | 4 |
| Crear el proyecto en Laravel Inertia en MacOS o Linux | 4 |
| Crear el proyecto en Laravel Inertia en Windows | 8 |
| Capítulo 3: Características de un proyecto Inertia con Jetstream | 9 |
| Laravel Fortify | 9 |
| Opciones | 13 |
| Opciones en perfil | 16 |
| Opciones en API Tokens | 19 |
| Opciones de equipos | 22 |
| Estructura del proyecto | 23 |
| Tipos de rutas | 26 |
| Rutas y renderizado de vistas en Inertia | 26 |
| Render de Inertia y Laravel básico | 27 |
| Capítulo 4: Primeros pasos | 30 |
| Migraciones | 30 |
| Modelos | 33 |
| Controladores | 35 |
| Crear un registro | 35 |
| Helper para el formulario | 38 |
| Crear un registro en la base de datos | 39 |
| Aplicar validaciones y mostrar errores | 39 |
| Usar el layout de la aplicación | 41 |
| Usar componentes Vue de Inertia | 43 |
| Editar un registro | 47 |
| Actualizar un registro en la base de datos | 49 |
| Generar slug | 50 |
| Listado | 50 |
| Todos los registros | 50 |
| Enlaces de navegación | 52 |

| | |
|---|------------|
| Listado Paginado | 53 |
| Crear un componente de paginación en Vue | 55 |
| Progress bar | 59 |
| Eliminar un registro | 61 |
| Estilos y contenedores | 62 |
| Container | 62 |
| Carta | 63 |
| Enlaces de acción | 64 |
| Paginación | 65 |
| Capítulo 5: Redirecciones y mensajes flash | 66 |
| Redirecciones | 67 |
| Mensajes flash | 67 |
| Middleware de peticiones de Inertia | 68 |
| Objeto \$page | 69 |
| Caso práctico | 70 |
| Estilo para el contenedor del mensaje flash | 71 |
| Remover contenedor pasado un tiempo | 72 |
| Capítulo 6: CRUD para los posts | 75 |
| Controlador | 75 |
| Ruta | 76 |
| Validaciones | 76 |
| Listado | 79 |
| Creación | 80 |
| Edición | 84 |
| Fusionar creación y actualización en un solo componente de Vue | 88 |
| Capítulo 7: Upload | 90 |
| Carga de archivos manejada desde el formulario de crear o editar | 94 |
| Consideraciones finales | 96 |
| Usar plugins para manejar la carga de archivos | 97 |
| Upload via Drag and Drop | 99 |
| Ver imagen cargada | 101 |
| Eliminar una imagen | 102 |
| Descargar una imagen | 103 |
| Capítulo 8: CKEditor | 105 |
| Capítulo 9: Diálogos de confirmación y mensajes toast | 109 |
| Configurar el diálogo de confirmación | 109 |
| Diálogo de confirmación para eliminar | 110 |
| Replicar para los Post | 111 |
| Modal de Inertia | 113 |
| Capítulo 10: Comunicación en componentes, Formulario paso por paso | 117 |
| Crear migraciones | 118 |
| Migraciones | 118 |
| Modelos | 122 |
| Formulario, Paso 1 | 123 |

| | |
|---|------------|
| Formulario, Paso 2: Empresa | 129 |
| Opcional, Definir un layout personalizado | 135 |
| Formulario, Paso 2: Persona | 136 |
| Formulario, Paso 3: Detalle | 141 |
| Componentes de componentes | 145 |
| Lógica para el formulario paso por paso | 146 |
| Manejar la lógica del paso por paso desde GeneralController.php | 148 |
| Segundo paso, empresa | 150 |
| Segundo paso, persona | 152 |
| Tercer paso, detalle | 153 |
| Errores de validaciones | 154 |
| Crear una función de ayuda para indicar el paso actual | 156 |
| Compartir datos, propiedad de step, primeros pasos | 157 |
| Compartir datos en Inertia | 157 |
| Compartir datos, propiedad de step, actualizar | 161 |
| Implementar el paso por paso | 162 |
| Opción de paso anterior | 164 |
| Parámetro de contacto general para los pasos por pasos | 167 |
| Capítulo 11: Filtros y campos de búsqueda | 169 |
| Cambios en la tabla | 169 |
| Seeders para los posts | 170 |
| Filtros para categorías, posteados y tipos | 171 |
| Campo de búsqueda para id, título y descripción | 174 |
| Filtrar por rango de fecha | 176 |
| Valores aplicados en los filtros desde el componente en Vue | 178 |
| Aplicar filtros sin usar un botón | 179 |
| debounce, retardo en los eventos | 180 |
| Filtros con cláusulas condicionales, when | 181 |
| Limpiar filtros | 183 |
| Capítulo 12: Ordenación de las columnas | 185 |
| Detalles finales | 189 |
| Foco en el campo de texto | 189 |
| Definir un texto para la opción por defecto en los SELECT | 189 |
| Botón de filtro | 191 |
| Capítulo 13: Web simple de Blog | 192 |
| Rutas | 192 |
| Layout para la web | 193 |
| Listado de Post | 193 |
| Detalle del Post | 199 |
| ¿Agregar paso por paso a la vista de detalle? | 201 |
| Capítulo 14: Carrito de compras | 203 |
| Esquema general | 204 |
| Administrar ítem del carrito | 206 |
| Controlador | 206 |

| | |
|--|------------|
| Ruta | 208 |
| Compartir datos del carrito mediante la sesión a Vue | 208 |
| Componente de Vue para administrar un Item del carrito | 208 |
| Listado de Post en el carrito | 210 |
| Vista de detalle del carrito | 213 |
| Replicar el carrito de compra en la base de datos | 214 |
| Crear migración y tabla | 214 |
| Modelo | 215 |
| Registrar cambios en la base de datos | 216 |
| Replicar carrito de la base de datos a sesión al momento del login | 218 |
| Mostrar un mensaje toast | 221 |
| Botón para mostrar la cantidad total de ítems | 222 |
| Mostrar el total y enlace para el listado de ítems del carrito | 224 |
| Capítulo 15: Opciones de los métodos HTTP de Inertia | 227 |
| Historial del navegador | 227 |
| Estado del componente | 227 |
| Preservar el scroll | 227 |
| Headers personalizados | 228 |
| Callbacks personalizados | 228 |
| Capítulo 16: Aplicación de to do list | 229 |
| Crear componente en Vue y controlador | 229 |
| Crear migración y modelo para los Todos | 232 |
| Crear un To Do | 233 |
| Popular listado | 235 |
| Editar un To Do | 236 |
| Aplicar validaciones | 238 |
| Remover un to do | 241 |
| Marcar completado un to do | 244 |
| Eliminar todos los to do | 246 |
| Ordenación | 247 |
| Ordenar el listado de to do | 249 |
| Problemas con la sincronización de las operaciones | 251 |
| Validación | 253 |
| Otras soluciones | 255 |
| Capítulo 17: Pruebas | 258 |
| Primeros pasos | 258 |
| De Laravel a Inertia | 258 |
| Assert Inertia | 259 |
| Métodos has y where | 265 |
| Blog | 267 |
| Prueba para el listado | 267 |
| Prueba para el detalle | 268 |
| Prueba para el filtro | 268 |
| Configurar base de datos para pruebas | 269 |

| | |
|--|-----|
| Autenticación | 273 |
| Dashboard | 274 |
| Categoría | 274 |
| Listado | 274 |
| Creación | 276 |
| Actualización | 277 |
| Eliminar | 278 |
| Post | 279 |
| Formulario paso por paso | 284 |
| Primer paso: General | 286 |
| Segundo paso: Company | 288 |
| Segundo paso: Person | 289 |
| Tercer paso: Detail | 290 |
| Editar | 291 |
| Primer paso: General | 291 |
| Segundo paso: Compañía | 292 |
| Segundo paso: Persona | 293 |
| Tercer paso: Detalle | 294 |
| Problemas revisando los datos compartidos/shared-data de Inertia | 295 |
| Errores de validaciones | 297 |
| Primer paso: General | 297 |
| Segundo paso: Compañía | 298 |
| Segundo paso: Persona | 300 |
| Tercer paso: Detalle | 301 |
| Carrito | 302 |
| To Do | 306 |
| Upload | 311 |
| Conclusiones | 314 |

Capítulo 1: Sobre Laravel Inertia

Inertia es un scaffolding para Laravel al igual que Livewire, es el otro scaffolding que podemos seleccionar al momento crear un proyecto en Laravel.

Inertia

Además de esto, un proyecto en Jetstream con Inertia ya nos trae configurados Tailwind.css y Vue ya listos para usar; así que, en definitiva, si quieres usar Vue junto con Laravel, Inertia es tu camino.

Inertia no es un framework, ni es un reemplazo a Laravel o Vue si no, está diseñado para funcionar con ellos, es un puente que permite conectar los dos.

En este libro emplearemos el término de “Laravel básico” para indicar el framework Laravel sin ningún añadido o scaffolding como es el caso de Laravel Inertia.

¿Qué nos ofrece Inertia?

Crear aplicaciones web modernas es difícil, herramientas como Vue y React son extremadamente poderosas, pero la complejidad que agregan al flujo de trabajo de un desarrollador de pila completa es una locura.

Agregar Vue en un proyecto en Laravel no es una tarea muy difícil, y lo podríamos hacer mediante la Node o la CDN y luego comunicar un proyecto en Vue con Laravel de múltiples formas aunque usualmente se usa una Rest Api; el “problema” con este enfoque es que es necesario trabajar en muchas capas y tecnologías y luego interconectar las mismas; trabajar con Laravel, con todo lo que nos permite, una rest api, luego las vistas con blade, para pasar a crear el proyecto de Vue con Node (o la CDN) y consumir la Rest Api con peticiones axios o fetch es un camino un poco laborioso que con Inertia podemos simplificar y optimizar bastante.

Pensando en lo anterior, Laravel Inertia es un marco completo para Laravel que simplifica la creación de interfaces dinámicas, sin dejar la comodidad de Laravel; en pocas palabras nos permite comunicar fácilmente Laravel con Vue y viceversa.

Si vemos un proyecto en Laravel Inertia, veremos que en el package.json:

```
"dependencies": {  
    "@headlessui/vue": "^1.7.23",  
    "@inertiajs/vue3": "^2.0.0-beta.3",  
    "@vitejs/plugin-vue": "^5.2.1",  
    "@vueuse/core": "^12.0.0",  
    "autoprefixer": "^10.4.20",  
    "class-variance-authority": "^0.7.1",  
    "clsx": "^2.1.1",  
    "concurrently": "^9.0.1",  
    "laravel-vite-plugin": "^1.0",
```

```
"lucide": "^0.468.0",
"lucide-vue-next": "^0.468.0",
"radix-vue": "^1.9.11",
"tailwind-merge": "^2.5.5",
"tailwindcss": "^3.4.1",
"tailwindcss-animate": "^1.0.7",
"typescript": "^5.2.2",
"vite": "^6.2.0",
"vue": "^3.5.13",
"ziggy-js": "^2.4.2"
},
```

Tenemos a Vue y a Inertia; si revisamos el **composer.json**:

```
"require": {
    "php": "^8.2",
    "inertiajs/inertia-laravel": "^2.0",
    "laravel/framework": "^12.0",
    "laravel/tinker": "^2.10.1",
    "tightenco/ziggy": "^2.4"
},
```

Tenemos a Laravel e Inertia; nuevamente, Inertia está presente en ambas capas y **es el encargado de comunicar el lado del servidor con el cliente de una manera mucho más directa que la mencionada anteriormente.**

Aparte de que, dependiendo como decidias instalar a Laravel Inertia, puedes habilitar opciones que ya vienen de gratis como:

1. Sistema de autenticación con registro, recuperación de credenciales.
2. Vista de perfil con carga de usuario.
3. Manejo de roles mediante equipos.
4. Manejo de la API Tokens mediante Laravel Breeze con una interfaz administrable.

En definitiva, Inertia no es un framework, lo puedes ver como un paquete más que agrega funcionalidades extras a algunos elementos de Laravel que en definitiva lo convierten en un scaffolding o esqueleto para nuestras aplicaciones.

Para esta tecnología, tenemos que emplear una documentación aparte (sin contar con la de Laravel).

La que nos ofrece Laravel Inertia al momento de desarrollar:

<https://inertiajs.com/>

Ventajas y desventajas de Inertia

Entre las ventajas que ofrece Inertia están:

- Las rutas de la aplicación están contenidas en un solo archivo, **web.php** y con esto, no hay necesidad de conectarlo con Vue Router en el cliente.
- Ahorra mucho trabajo y configurar capas extras para comunicar Laravel y Vue.

Entre las desventajas, presenta:

- El desarrollador que vaya a utilizar Inertia debe tener conocimientos tanto de PHP como de Vue, lo que abre una complejidad mayor en el uso de estas tecnologías por separado, en equipos de desarrolladores distintos.
- Agrega capas de complejidad extras y bifurcaciones que no a todos los desarrolladores pudiera gustar.

Capítulo 2: Crear un proyecto

Se da por hecho de que el lector tiene los conocimientos necesarios para desarrollar en Laravel básico; siendo así, se hará para una mera mención al entorno que se va a usar para desarrollar en Laravel que será:

- Laravel Sail con Docker para Linux y MacOS.
- Laragon para Windows.

Aunque tu eres libre de emplear otro de los tantos disponibles para Laravel.

Crear el proyecto en Laravel Inertia en MacOS o Linux

En este apartado, vamos a crear el proyecto con el cual vamos a trabajar a lo largo del libro para MacOS o Linux.

En versiones recientes, el instalador te pregunta antes si quieres crear un proyecto con Vue o React:

```
$ laravel new inertiatore
```

Te aparecerá una pantalla como la siguiente:



Which starter kit would you like to install?

- None
- React
- Vue
- Livewire

Seleccione:

```
Vue
```

A continuación, te preguntará cuál sistema de autenticación quieras emplear:

```
Which authentication provider do you prefer? [Laravel's built-in authentication]:  
[laravel] Laravel's built-in authentication  
[workos ] WorkOS (Requires WorkOS account)
```

Puedes colocar cualquiera pero en el libro usaremos:

```
|laravel
```

A continuación, te preguntara cual paquete quieres utilizar para las pruebas unitarias:

```
|Which testing framework do you prefer? [Pest]:
```

- [0] Pest
- [1] PHPUnit

Puedes seleccionar cualquiera, en el libro seleccionamos la opción "1".

Si quieres preparar el ecosistema de Node:

```
|Would you like to run npm install and npm run build? (yes/no) [yes]:
```

```
>
```

Coloca:

```
|yes
```

Una vez terminado el proceso, para que pueda funcionar con Laravel Sail (Si vas a emplear Sails y no Laragon/Herd); cuyo paquete viene instalado por defecto:

```
|$ php artisan sail:install
```

Levantamos el proceso de Sail; para que genera la imagen en Docker, configure nuestro proyecto, y en pocas palabras, tengamos todo listo:

```
|$ ./vendor/bin/sail up
```

Con esto, veremos en Docker:

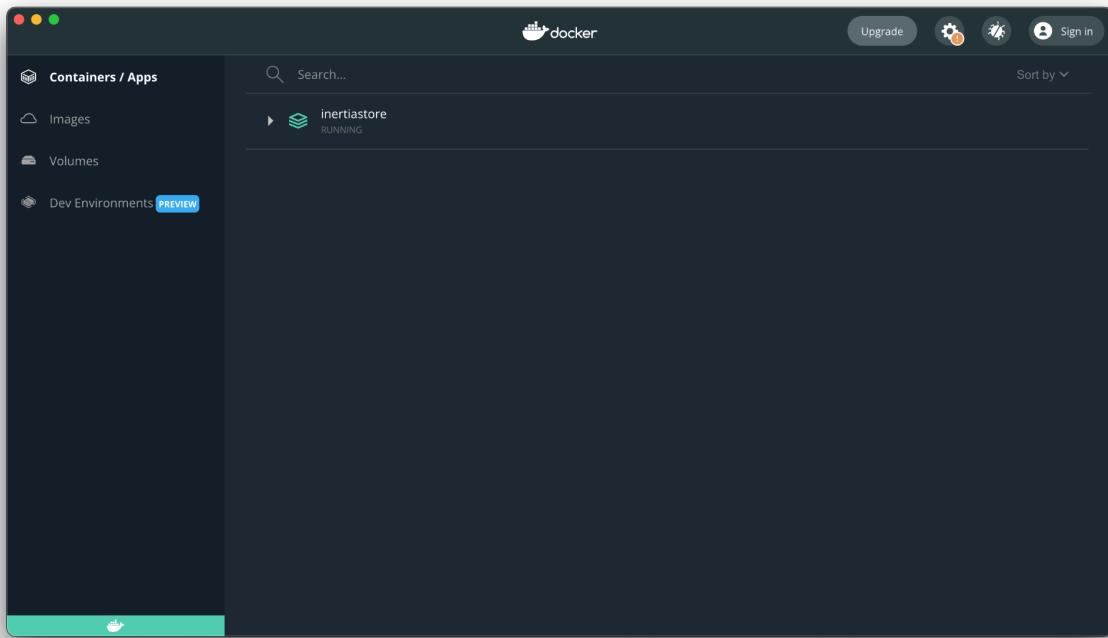


Figura 2-1: Laravel Inertia en Docker

Recuerda que no debes de tener ningún otro proceso iniciado en el puerto 80; por ejemplo Apache; daría un error como el siguiente:

```
Error response from daemon: Ports are not available: exposing port TCP 0.0.0.0:80 ->
0.0.0.0:0: listen tcp 0.0.0.0:80: bind: address already in use
```



Figura 2-2: Localhost

Para solucionarlo, debes de detener el proceso que tiene ocupado dicho puerto; en el caso de Apache, sería algo como:

```
$ sudo apachectl stop
```

Ahora notaremos que no tenemos tablas generadas en nuestro proyecto; para eso, recuerda que tienes que ejecutar el **migrate** desde **sail**, para que pueda generar las tablas en el contenedor de MySQL de Docker:

```
$ php artisan migrate
```

Con esto, si vamos a:

<http://localhost>

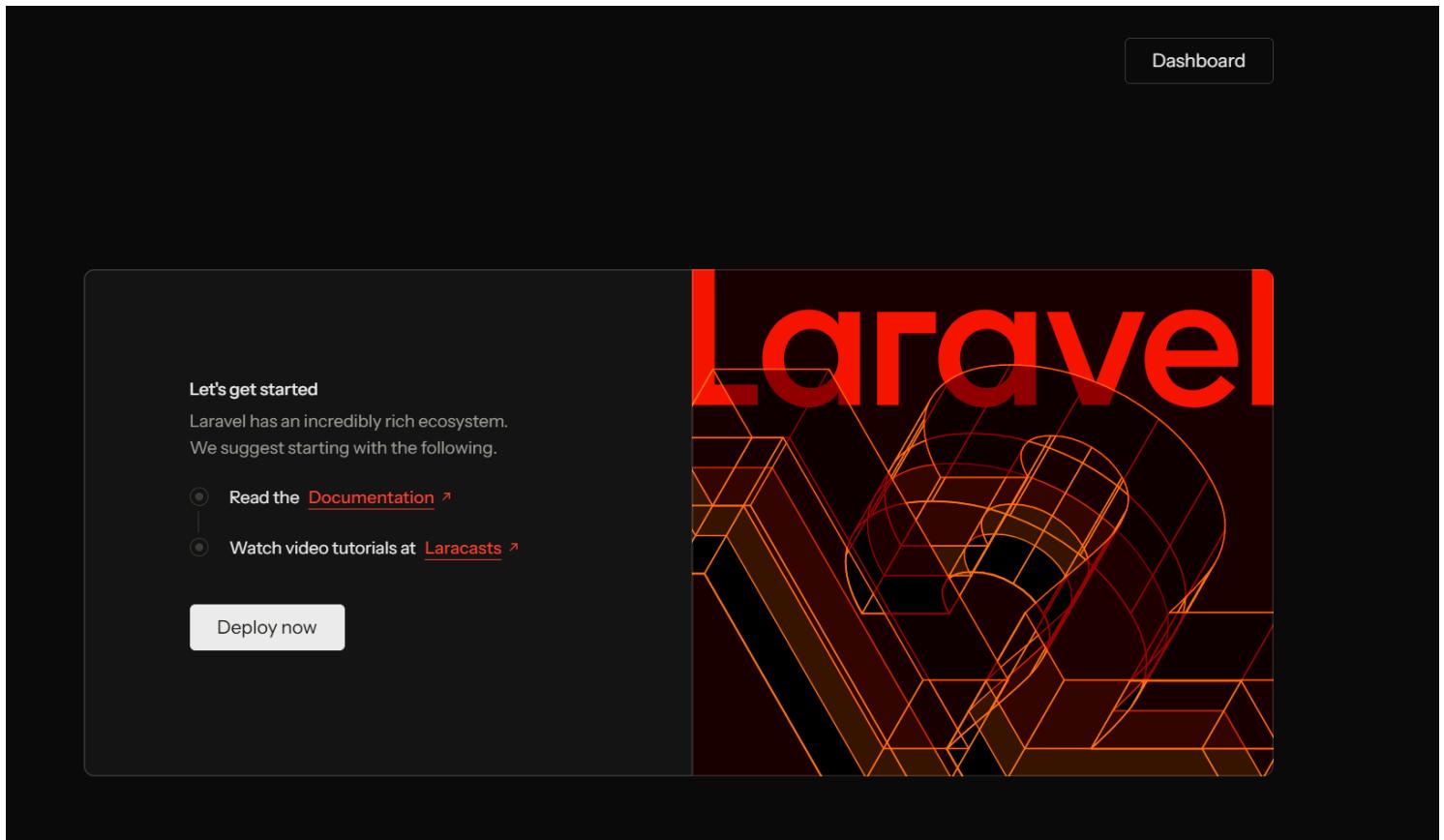


Figura 2-3: Inertia ejecutándose

Con esto tenemos un proyecto en Laravel Inertia completamente funcional y listo para usar.

A partir del proyecto generado en Docker, podrás detener/iniciar el proyecto desde la interfaz de Docker:

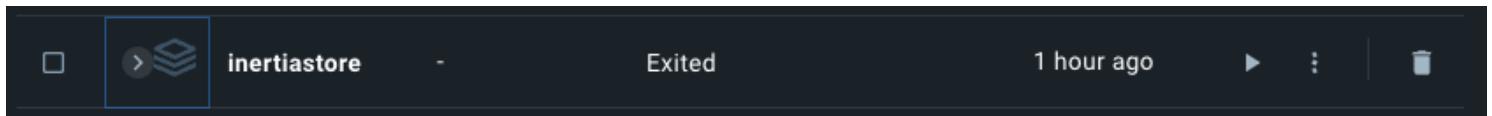


Figure 2-4: Opciones en Docker

Crear el proyecto en Laravel Inertia en Windows

En Windows con Laragon, de igual manera, crearemos el proyecto con el instalador de Laravel:

```
$ laravel new inertiastore
```

Y reiniciamos Laragon desde el botón de "Recargar" (probablemente tengas que crear la base de datos y ejecutar las migraciones como hicimos antes).

Ya con esto, si vamos a:

<http://inertiastore.test/>

Con esto tenemos un proyecto en Laravel Inertia completamente funcional y listo para usar.

Capítulo 3: Características de un proyecto Inertia

En este capítulo, vamos a hablar sobre los aspectos más importantes que tenemos al momento de crear un proyecto en Laravel Livewire a partir de la versión 12 de Laravel, que como comentamos antes, a diferencia de versiones anteriores, ya no trae incluidas las funcionalidades que tenemos en versiones anteriores con Jetstream.

Lo primero que vemos, es un par de enlaces para registrarse y login, en este punto, si ya no los has hecho, crea un usuario; una vez registrado o iniciar la sesión, tendrás acceso a:

<http://inertiastore.test/dashboard>

Cuya vista corresponde a:

resources\js\pages\auth\Login.vue

```
<script setup lang="ts">
import InputError from '@/components/InputError.vue';
```

```

import TextLink from '@/components/TextLink.vue';
import { Button } from '@/components/ui/button';
import { Checkbox } from '@/components/ui/checkbox';
import { Input } from '@/components/ui/input';
import { Label } from '@/components/ui/label';
import AuthBase from '@/layouts/AuthLayout.vue';
import { Head, useForm } from '@inertiajs/vue3';
import { LoaderCircle } from 'lucide-vue-next';

defineProps<{
    status?: string;
    canResetPassword: boolean;
}>();

const form = useForm({
    email: '',
    password: '',
    remember: false,
});

const submit = () => {
    form.post(route('login'), {
        onFinish: () => form.reset('password'),
    });
};

</script>

<template>
    <AuthBase title="Log in to your account" description="Enter your email and password below to log in">
        <Head title="Log in" />

        <div v-if="status" class="mb-4 text-center text-sm font-medium text-green-600">
            {{ status }}
        </div>

        <form @submit.prevent="submit" class="flex flex-col gap-6">
            <div class="grid gap-6">
                <div class="grid gap-2">
                    <Label for="email">Email address</Label>
                    <Input
                        id="email"
                        type="email"
                        required
                        autofocus
                        :tabindex="1"
                    >
                </div>
            </div>
        </form>
    </AuthBase>
</template>

```

```

        autocomplete="email"
        v-model="form.email"
        placeholder="email@example.com"
    />
    <InputError :message="form.errors.email" />
</div>

<div class="grid gap-2">
    <div class="flex items-center justify-between">
        <Label for="password">Password</Label>
        <TextLink v-if="canResetPassword" :href="route('password.request')"
class="text-sm" :tabindex="5">
            Forgot password?
        </TextLink>
    </div>
    <Input
        id="password"
        type="password"
        required
        :tabindex="2"
        autocomplete="current-password"
        v-model="form.password"
        placeholder="Password"
    />
    <InputError :message="form.errors.password" />
</div>

<div class="flex items-center justify-between" :tabindex="3">
    <Label for="remember" class="flex items-center space-x-3">
        <Checkbox id="remember" v-model:checked="form.remember"
:tabindex="4" />
        <span>Remember me</span>
    </Label>
</div>

<Button type="submit" class="mt-4 w-full" :tabindex="4"
:disabled="form.processing">
    <LoaderCircle v-if="form.processing" class="h-4 w-4 animate-spin" />
    Log in
</Button>
</div>

<div class="text-center text-sm text-muted-foreground">
    Don't have an account?
    <TextLink :href="route('register')" :tabindex="5">Sign up</TextLink>
</div>

```

```
</form>
</AuthBase>
</template>
```

Cómo puedes apreciar, hay muchos componentes preexistentes listos para utilizar, por ejemplo:

- resources\js\components\ui\input
- resources\js\components\ui\button
- resources\js\components\ui\label
- resources\js\components\ui\sidebar

Cuyos nombres son muy descriptivos y que utilizaremos a lo largo del libro para la creación de nuestra aplicación, muchos de estos componentes tienen variantes por lo tanto, desde una sola importación, podemos utilizar variantes:

```
interface Props extends PrimitiveProps {
    variant?: ButtonVariants['variant'];
    size?: ButtonVariants['size'];
    class?: HTMLAttributes['class'];
}
```

El resto de las acciones disponibles creadas por el instalador de Laravel tenemos acciones como cambiar la contraseña y otros datos de usuario, cerrar la cuenta, modo oscuro y poco más, esto lo tienes en el sidebar listado antes en la parte inferior:

<http://inertiastore.test/settings/profile>

Lamentablemente, ya no hay disponible opciones predefinidas antes con Jetstream como carga de avatar, sesión, roles/equipos.

Capítulo 4: Primeros pasos

En este capítulo, vamos a trabajar con los componentes en Vue junto con los controladores en Laravel de manera directa usando Inertia, para eso, vamos a tener que realizar las configuraciones típicas que hacemos en los proyectos Laravel, que van desde crear y ejecutar las migraciones, hasta sus modelos, para que finalmente, podamos crear los controladores y hacer las primeras pruebas.

Migraciones

Para el proyecto vamos a usar un conjunto de migraciones para crear las bases de nuestra aplicación; la aplicación que vamos a crear va a ser de tipo CRUD, por tal motivo, vamos a crear las relaciones típicas de categorías, posts y etiquetas, en donde un post tiene una categoría asignada y un post puede tener de cero a muchas etiquetas.

```
$ php artisan make:migration createCategoriesTable
```

Con el siguiente cuerpo:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up(): void
    {
        Schema::create('categories', function (Blueprint $table) {
            $table->id();
            $table->string('title', 255);
            $table->string('slug', 255);
            $table->string('image', 260)->nullable();
            $table->text('text')->nullable();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down(): void
    {
        Schema::dropIfExists('categories');
    }
}
```

```
|};
```

Para las migraciones de categorías y etiquetas, incluimos una columna para colocar contenido al igual que definir una imagen; por lo demás, cuenta con los campos típicos de título y la url limpia.

```
$ php artisan make:migration createTagsTable
```

Con el siguiente cuerpo:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up(): void
    {
        Schema::create('tags', function (Blueprint $table) {
            $table->id();
            $table->string('title', 255);
            $table->string('slug', 255);
            $table->string('image', 260)->nullable();
            $table->text('text')->nullable();
            $table->timestamps();
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down(): void
    {
        Schema::dropIfExists('tags');
    }
};
```

Y la de posts:

```
$ php artisan make:migration createPostsTable
```

Con el siguiente cuerpo:

```
***  
use Illuminate\Support\Carbon;
```

```

***  

return new class extends Migration  

{  

    /**  

     * Run the migrations.  

     *  

     * @return void  

     */  

    public function up(): void  

    {  

        Schema::create('posts', function (Blueprint $table) {  

            $table->id();  

            $table->string('title', 255);  

            $table->string('slug', 255);  

            $table->string('date')->default(Carbon::now());  

            $table->string('image', 260)->nullable();  

            $table->text('description');  

            $table->text('text')->nullable();  

            $table->enum('posted', ['yes', 'not'])->default('not');  

            $table->enum('type', ['advert', 'post', 'course', 'movie'])->default('post');  

            $table->foreignId('category_id')->constrained()->onDelete('cascade');  

            $table->timestamps();  

        });  

    }  

    /**  

     * Reverse the migrations.  

     *  

     * @return void  

     */  

    public function down(): void  

    {  

        Schema::dropIfExists('posts');  

    }  

};
```

Aquí tenemos las columnas para manejar el título, url limpia, descripción, contenido e imagen; también colocamos un par de tipos enumerados, el primero es para saber el estado de la publicación y otro para indicar el tipo de contenido; esto es útil ya que, con un mismo esquema puedes crear distintos tipos de recursos; en este ejemplo, post, publicidad, cursos, películas...

La tabla pivot:

```
$ php artisan make:migration createTaggablesTable
```

Con el siguiente cuerpo:

```
return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up(): void
    {
        Schema::create('taggables', function (Blueprint $table) {
            $table->bigInteger('tag_id')->unsigned();
            $table->bigInteger('taggable_id')->unsigned();
            $table->string('taggable_type');
        });
    }

    /**
     * Reverse the migrations.
     *
     * @return void
     */
    public function down(): void
    {
        Schema::dropIfExists('taggables');
    }
};
```

Al trabajar con una relación de muchos a muchos, necesitamos una tabla pivote, vamos a crear la de **taggables**, con la misma, podemos definir las etiquetas de una manera mórficas; lo que significa que puedes usar esta tabla para otras relaciones que no sean la de post; por eso vez que tenemos el tipo, en el cual colocamos la relación.

Finalmente, ejecutamos las migraciones con:

```
$php artisan migrate
```

Modelos

Ya con nuestras migraciones y tablas listas, lo siguiente que necesitamos son sus modelos junto con las relaciones que vamos a usar en cada uno de esos modelos.

Para las categorías:

```
$ php artisan make:model Category
```

```
class Category extends Model
{
    use HasFactory;

    protected $fillable=['title','slug','image','text'];

    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

Para las etiquetas:

```
$ php artisan make:model Tag
```

```
class Tag extends Model
{
    use HasFactory;
    protected $fillable=['title','slug','image','text'];

    public function posts()
    {
        return $this->hasMany(Post::class);
    }
}
```

Para los posts:

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

```
class Post extends Model
{
    use HasFactory;

    protected $fillable=['title',
'slug','date','image','text','description','posted','type', 'category_id'];

    public function category()
    {
        return $this->belongsTo(Category::class);
    }

    public function tags()
```

```

{
    return $this->morphToMany(Tag::class, 'taggables');
}

public function getImageUrl()
{
    return URL::asset("images/post/".$this->image);
}
}

```

Controladores

Ahora, ya tenemos las estructuras en la base de datos, es decir, las tablas, y el cómo nos conectamos a las mismas; es decir, los modelos, lo siguiente que necesitamos son los controladores; que es el mecanismo empleado por excelencia para devolver las vistas, en este caso, las vistas son componentes en Vue tal cual vimos anteriormente.

Para crear los componentes de Laravel, usamos artisan al igual que siempre:

```

$ php artisan make:controller Dashboard/PostController -r

$ php artisan make:controller Dashboard/CategoryController -r

$ php artisan make:controller Dashboard/TagController -r

```

Debes de ejecutar cada uno de estos 3 comandos de manera independiente.

Crear un registro

El siguiente punto en la lista, es la de crear un formulario con el cual podamos crear nuestros primeros registros y aprender cual es el funcionamiento básico; para ello, vamos a comenzar por el componente, el componente de Vue que permite implementar el formulario.

Como veremos en la implementación, en cuanto al template, es exactamente el mismo cuerpo que tendríamos en un formulario con Vue.

- Un formulario con el evento de submit que es procesado mediante una función.
- Los **v-model** para cada uno de los campos.

En la siguiente implementación se va a crear el formulario para crear categorías dado el título y el slug.

La implementación queda como:

resources/js/pages/dashboard/category/Create.vue

```

<template>
    <form @submit.prevent="submit">

```

```

<label for="">Title</label>
<input type="text" v-model="form.title">

<label for="">Slug</label>
<input type="text" v-model="form.slug">

    <button type="submit">Send</button>
</form>
</template>

<script>

import { router, useForm } from '@inertiajs/vue3';

export default {
    setup() {

        const form = useForm({
            title: '',
            slug: '',
        });

        function submit() {
            form.post(route('category.store', form))
            // router.post(route('category.store', form))
        }

        return {
            submit, form
        };
    }
}
</script>

```

Explicación del código anterior

En el JavaScript, que es donde existen algunos cambios, vemos que no tenemos la opción de data; en su lugar, para declarar las propiedades, se usa una función llamada **setup()** con la cual se declaran e inicializan las propiedades, así como las funciones que se necesites usar en el componente, en este caso, la función de **submit()**.

Para referenciar rutas con nombre, tenemos la función de **route()** la cual recibe como parámetro, el nombre de la ruta.

Para enviar peticiones, al servidor desde Inertia, tenemos:

```
form.submit(method, url, options)
```

```
form.get(url, options)
form.post(url, options)
form.put(url, options)
form.patch(url, options)
form.delete(url, options)
```

En el caso anterior, sería una petición de tipo post que, como parámetro, se pasa un objeto con los datos, en este caso, el formulario.

También podemos enviar peticiones mediante el objeto router:

```
router.post(route('category.store'), form))
```

Como puedes apreciar en el código anterior.

Para el controlador, es lo típico en Laravel, tenemos una función para devolver el formulario, y otra para procesarlo, de momento, solamente devuelve un mensaje mediante la función de **dd()** con la cual imprimimos los datos recibidos por el formulario:

app/Http/Controllers/Dashboard/CategoryController.php

```
<?php

namespace App\Http\Controllers\Dashboard;

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

class CategoryController extends Controller
{

    public function create()
    {
        return inertia("dashboard/category/Create");
    }

    public function store(Request $request)
    {
        dd($request->all());
    }
}
```

Se crea la ruta de tipo CRUD para las categorías.

routes/web.php

```

Route::middleware(
    ['auth', 'verified'],
)->prefix('dashboard')->group(function () {
    Route::resource('/category', App\Http\Controllers\Dashboard\CategoryController::class);

    Route::get('/', function () {
        return Inertia::render('Dashboard');
    })->name('dashboard');
});

```

Si damos a enviar, veremos una pantalla con los datos:

```

array:2 [
    "title" => "Test"
    "slug" => "test-slug"
]

```

Helper para el formulario

La función de **useForm()** ayuda a reducir la cantidad de repetitivo necesario para los formularios típicos indicando una cantidad de opciones muy comunes como indicar el estado del formulario y los errores.

Para usar la función anterior, la establecemos en la propiedad del formulario:

```

import { useForm } from '@inertiajs/vue3';

// ***
const form = useForm({
    title: null,
    slug: null,
});
// ***

```

Si damos a enviar:

```

array:10 [
    "title" => "Test"
    "slug" => "test-slug"
    "isDirty" => true
    "errors" => []
    "hasErrors" => false
    "processing" => false
    "progress" => null
    "wasSuccessful" => false
    "recentlySuccessful" => false
    "__rememberable" => true
]

```

Veremos que ahora, tenemos datos adicionales en el request:

- **isDirty**, Verifica si un formulario tiene algún cambio.
- **errors**, Lista de los errores provistos por las validaciones aplicadas en el servidor.
- **hasErrors**, Verifica si un formulario tiene algún error.
- **processing**, Rastrea si un formulario se está enviando actualmente.
- **progress**, En el caso de que esté cargando archivos, el evento de progreso actual está disponible a través de la propiedad de progress.
- **wasSuccessful**, Cuando un formulario se ha enviado con éxito, la propiedad **wasSuccessful** será verdadera
- **recentlySuccessful**, Esta propiedad se establecerá en verdadero durante dos segundos después de enviar un formulario correctamente. Esto es útil para mostrar mensajes de éxito temporales.

Crear un registro en la base de datos

Ya con el formulario, para crear un registro, en este caso una categoría, es exactamente la misma implementación que hacemos en Laravel básico, es decir, con el modelo, usamos la método de **create()** en Laravel, para crear el registro:

app/Http/Controllers/Dashboard/CategoryController.php

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

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

class CategoryController extends Controller
{

    // ***

    public function store(Request $request)
    {
        Category::create(
            [
                'title' => request('title'),
                'slug' => request('slug'),
            ]
        );
    }
}
```

Aplicar validaciones y mostrar errores

Ya tenemos un formulario para crear registros, aunque falta un paso fundamental, aplicar las validaciones en el servidor con Laravel y poder manejarlas desde el componente en Vue; su uso es muy transparente y sencillo y desde Laravel se realiza exactamente las mismas configuraciones que podemos hacer con Laravel básico; por lo tanto, esta es otra prueba de que Inertia se integra muy bien con Laravel sin necesidad de cambiar su estructura básica.

Vamos a usar las validaciones creando un archivo aparte, que son las más empleadas y recomendadas de usar:

```
$ php artisan make:request Category/Store
```

El cual tendrá la siguiente estructura:

app/Http/Requests/Category/Store.php

```
<?php

namespace App\Http\Requests\Category;

use Illuminate\Foundation\Http\FormRequest;

class Store extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
    {
        return true;
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string, mixed>
     */
    public function rules(): array
    {
        return [
            "title" => "required|min:5|max:255",
            "slug" => "required|min:5|max:255|unique:categories"
        ];
    }
}
```

Y se establecen en el método de **store()** del componente:

```
use App\Http\Requests\Category\Store;

// ***
public function store(Store $request)
{
    Category::create($request->validated());
}
```

Si desde el componente de creación de componentes en Vue, enviamos un formulario sin datos o invalido y revisamos la pestaña Network del navegador, veremos que los errores se devuelven en un props, automáticamente:

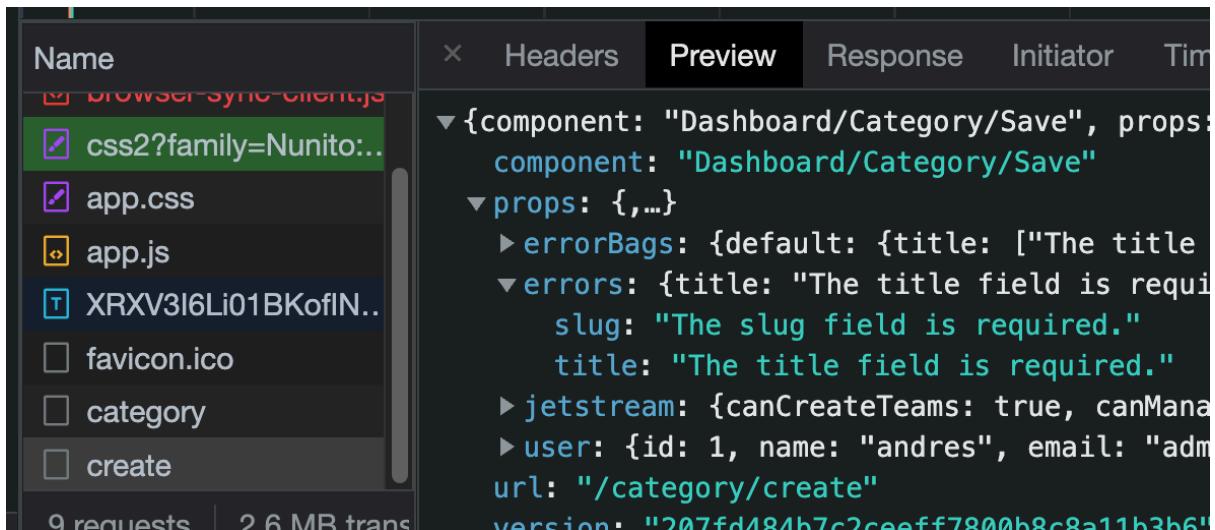


Figura 4-1: Parámetros en el prop del formulario

Para poder usar estos errores desde el componente en Vue; debemos de crear el apartado de **props** en Vue y declarar el de **errors**:

```
export default {
    // ***
    props: {
        errors: Object
    },
    // ***
}
```

Y desde el formulario:

```
<label for="">Title</label>
<input type="text" v-model="form.title" />
```

```
<div v-if="errors.title">{{errors.title}}</div>
<label for="">Slug</label>
<input type="text" v-model="form.slug" />
<div v-if="errors.slug">{{errors.slug}}</div>
<button type="submit">Send</button>
```

Si enviamos un formulario inválido nuevamente, veremos:

```
The title field is required.
The slug field is required.
```

Usar el layout de la aplicación

Hasta este momento, se tiene un formulario de creación con validaciones aplicadas completamente funcional, pero, en cuanto a interfaz, estamos usando cero CSS y con esto, nada del estilo que podemos usar de gratis al usar Inertia; si revisamos cualquiera de los componentes de nuestra aplicación, por ejemplo, el de perfil:

resources/js/pages/settings/Profile.vue

```
***  
<template>  
  <AppLayout :breadcrumbs="breadcrumbs">  
***
```

El cual se encuentra en:

resources/js/layouts/AppLayout.vue

Desde la ubicación que tenemos actualmente, para acceder al AppLayout que es el que define el CSS usado por la aplicación, al igual que el resto del HTML con sus componentes, tenemos:

```
import AppLayout from "../../Layouts/AppLayout"
```

Aunque en Vue, podemos usar el carácter de arroba “@“ con el cual podemos acceder a cualquier archivo que se encuentra partir de la ruta de **resources/js** como si fuera una importancia relativa, así que, con esto, la importación que vamos a usar, queda de la siguiente manera:

resources/js/pages/dashboard/category/Create.vue

```
import AppLayout from '@/layouts/AppLayout.vue';
components: {
  AppLayout
},
```

Y lo usamos desde el template del componente de creación, como hijo directo:

resources/js/Pages/Dashboard/Category/Create.vue

```
<template>
  <AppLayout>
    // ***
  </AppLayout>
</template>
```

Y tendremos:

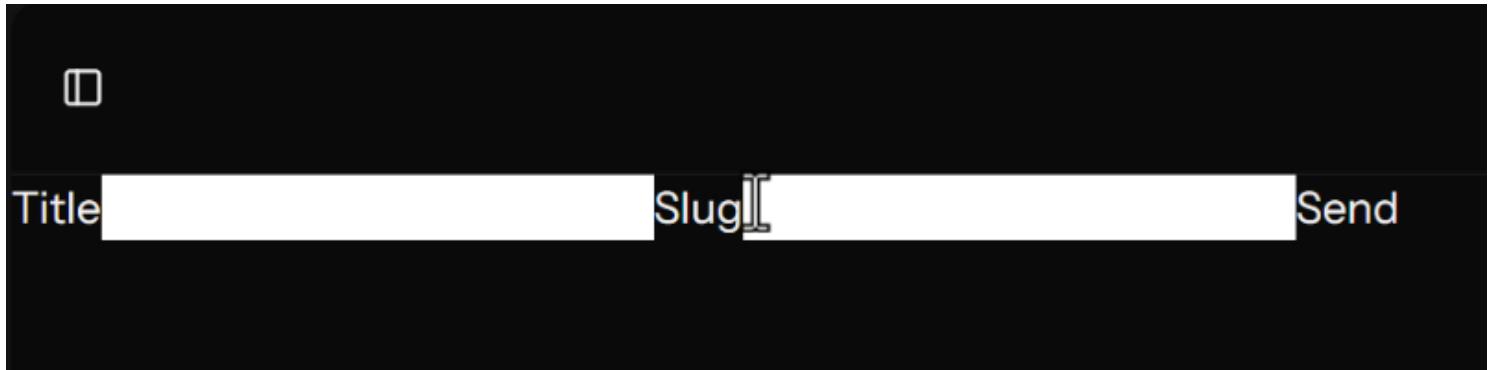


Figura 4-2: Formulario para crear categoría

Usar componentes Vue de Inertia

Apenas creamos un proyecto en Inertia, ya tenemos a nuestra disposición un conjunto de componentes en Vue listos para usar, que van desde elementos de formulario como inputs, labels, labels de errores entre otros, también tenemos botones, modal entre otros; todos estos componentes los puedes ver en:

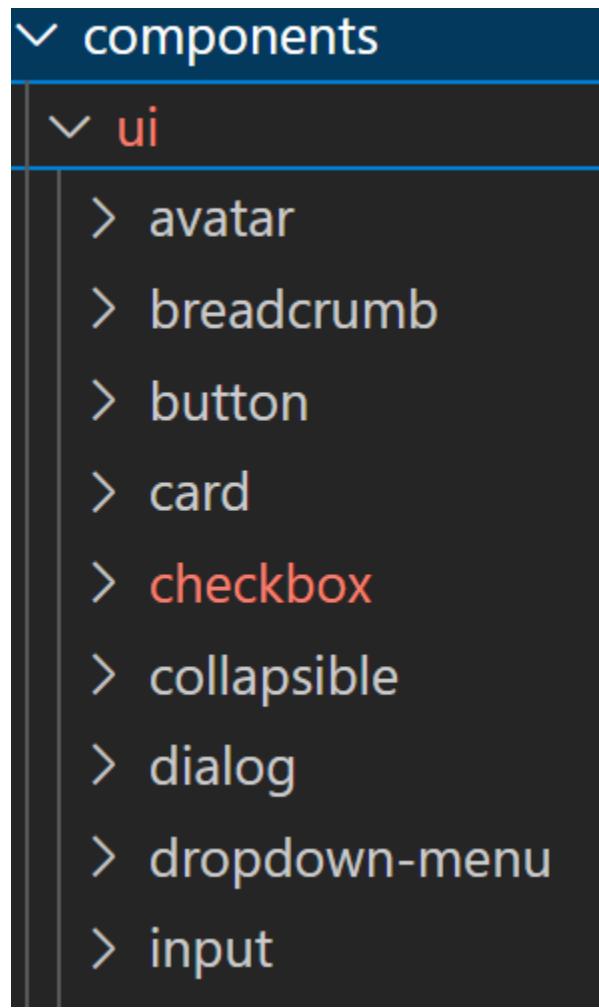


Figura 4-3: Componentes listo para emplear en Vue

Siguiendo adaptando el componente de creación de categorías en Vue al diseño que tenemos a nivel de la aplicación de Inertia, vamos a trabajar en el resto de los elementos HTML que son:

- Contenedor de formulario.
- Campo de formulario.
- Label.
- Label para los errores.
- Botón.

Todos estos elementos HTML tienen su correspondiente con un componente en Vue creado desde Inertia:

- Contenedor de formulario: FormSection.vue.
- Campo de formulario: TextInput.vue.
- Label: InputLabel.vue.
- Label para los errores: InputError.vue.
- Botón: PrimaryButton.vue.

Y en caso de que no exista, lo puedes crear y si existe y quieras otro diseño, puedes cambiarlo y crear uno nuevo.

FormSection.vue

En el caso de este componente, que es usado para trabajar con los formularios y funciona en base a slot:

- title, título del formulario.
- description, algún texto secundario que quieras definir.
- form, todo el cuerpo del formulario.
- actions, botones y demás acciones del formulario.

Aparte de esto, se debe de especificar un evento **submitted** el cual se encarga de procesar el formulario.

En el caso del resto de los componentes de Vue creados por Inertia que vamos a usar, basta con usarlos como si fueran su equivalente en HTML; finalmente, el código queda como:

resources\js\pages\dashboard\category\Create.vue

```
<template>
  <AppLayout :breadcrumbs="breadcrumbs">
    <div class="px-4 py-6 max-w-xl">
      <HeadingSmall title="Create Category" description="Form to create categories"
/>
      <form class="my-6" @submit.prevent="submit">
        <div class="grid grid-cols-2 gap-2">
          <Label>Title</Label>
          <!-- <input type="text" v-model="form.title"> -->
          <Input type="text" v-model="form.title" placeholder="Title" required />
          <!-- <div v-if="errors.title">{{ errors.title }}</div> -->
          <InputError :message="errors.title" />

          <Label>Slug</Label>
          <!-- <input type="text" v-model="form.slug"> -->
          <Input type="text" v-model="form.slug" placeholder="Slug" required />
          <!-- <div v-if="errors.slug">{{ errors.slug }}</div> -->
          <InputError :message="errors.slug" />

          <div>
            <Button :disabled="form.processing" type="submit">Send</Button>
          </div>
        </div>
      </form>
    </div>
  </AppLayout>
</template>

<script>
import { router, useForm } from '@inertiajs/vue3';

```

```

import AppLayout from '@/layouts/AppLayout.vue';

import Input from '@/components/ui/input/Input.vue';
import Label from '@/components/ui/label/Label.vue';
import InputError from '@/components/InputError.vue';
import HeadingSmall from '@/components/HeadingSmall.vue';

import { Button } from '@/components/ui/button';

export default {
  props: {
    errors: Object
  },
  components: {
    AppLayout,
    InputError,
    HeadingSmall,
    Input,
    Label,
    Button,
  },
  setup() {
    const form = useForm({
      title: '',
      slug: '',
    });

    const breadcrumbs = [
      {
        title: 'Categories',
        // href: '/dashboard/category/create',
      },
    ];

    function submit() {
      console.log(form);
      // form.post(route('category.store', form))
      router.post(route('category.store', form))
      // Your submit logic here
    }

    return {
      submit, form, breadcrumbs
    };
  }
}

```

```
}
```

Y tenemos:

Title

The title field is required.

Slug

cate-test2

SEND

Figura 4-4: Formulario para crear categoría

Breadcrumbs

Si revisamos otros componentes, como el de perfil:

resources\js\pages\settings\Profile.vue

```
const breadcrumbs: BreadcrumbItem[] = [
  {
    title: 'Profile settings',
    href: '/settings/profile',
  },
];
```

Veremos que existe un componente de breadcrumbs, cuyo propósito es ir dejando rastros en la navegación, es decir, las páginas previas que se encuentran vinculadas a la actual; en código anterior, es una sintaxis solamente válida si usas TypeScript, como lo usa el componente de perfil:

```
<script setup lang="ts">
```

Que no es utilizado en el libro, TypeScript da posibilidades adicionales en JavaScript como los tipos, interfaces:

```
export interface BreadcrumbItem {
    title: string;
    href: string;
}
```

Etc; el código anterior corresponde a la definición de la clase **BreadcrumbItem**; en donde define como obligatorio el título y el href, y en definitiva, con la interfaz ofrece una estructura que tenemos que seguir; si queremos usar las migajas de pan en nuestro componente sin emplear TypeScript, queda como:

resources\js\pages\dashboard\category\Create.vue

```
setup() {
    **

    const breadcrumbs = [
        {
            title: 'Categories',
            // href: '/dashboard/category/create',
        },
    ];
    **

    return {
        submit, form, breadcrumbs
    };
}
```

Editar un registro

Para editar un registro, se usa la misma estructura que en la fase de creación, pero con algunos agregados; desde el controlador, pasamos la referencia a la categoría, al igual que hacemos con Laravel básico:

app/Http/Controllers/Dashboard/CategoryController.php

```
public function edit(Category $category)
{
    return inertia("dashboard/category/Edit", compact('category'));
```

```

}

public function update(Request $request, Category $category)
{
    dd($request->all());
}

```

Y con el **nombre del argumento provisto en la función de inertia()** (category), en el cual, pasamos la categoría a editar mediante la función de **compact()**, lo especificamos en el **props** del componente en Vue:

resources/js/Pages/dashboard/category/Edit.vue

```

/***
props: {
    errors: Object,
    category: Object
},
/***

```

Luego, para usar el prop anterior, se pasa como argumento en la función de **setup()** y se inicializa los **v-model** del formulario. Además, se actualiza la ruta de la función de submit:

```

setup(props) {
    const form = useForm({
        title: props.category.title,
        slug: props.category.slug,
    });

    const breadcrumbs = [
        {
            title: 'Edit Category: ' + props.category.title,
            // href: '/dashboard/category/create',
        },
    ];

    function submit() {
        router.put(route("category.update", props.category.id), form);
    }

    return { form, submit, breadcrumbs };
},
/***

```

Y al enviar el formulario, tenemos:

```

array:11 [
    "id" => 1
]

```

```
"title" => "Test"
"slug" => "test-slug"
"isDirty" => false
"errors" => []
"hasErrors" => false
"processing" => false
"progress" => null
"wasSuccessful" => false
"recentlySuccessful" => false
"__rememberable" => true
]
```

Actualizar un registro en la base de datos

Ya tenemos el componente en Vue listo para actualizar, y el renderizado del mismo mediante el función de `edit()` en Laravel; el siguiente paso, es realizar la actualización con las validaciones del lado del servidor; creamos un archivo aparte para las validaciones:

```
$ php artisan make:request Category/Put
```

No usamos las validaciones de `app/Http/Requests/Category/Store.php` ya que, debemos de realizar una condición adicional para que el slug de la categoría que estamos editando, sea excluida de la condición de unicidad de la validación; para ello:

`app/Http/Requests/Category/Put.php`

```
<?php

namespace App\Http\Requests\Category;

use Illuminate\Foundation\Http\FormRequest;

class Put extends FormRequest
{
    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return [
            "title" => "required|min:5|max:255",
            "slug" =>
"required|min:5|max:255|unique:categories,id,".$this->route("category")->id
        ];
    }
}
```

```
|}
```

Y en el controlador para actualizar:

```
app/Http/Controllers/Dashboard/CategoryController.php
```

```
use App\Http\Requests\Category\Put;
// ***
public function update(Put $request, Category $category)
{
    $category->update($request->validated());
}
// ***
```

Con esto, tenemos ya listo el proceso de edición.

Generar slug

Usualmente en las aplicaciones que manejan un campo para el slug, es generado internamente por otro campo, en este caso, el del título; en Laravel, podemos hacer este proceso de generar el slug automáticamente fácilmente mediante el método de **prepareForValidation()** en el cual hacemos el merge con el slug generado a partir del título usando el helper de ayuda **slug()** provisto por Laravel; recuerda que el método de **prepareForValidation()** se ejecuta antes de realizar las validaciones; así que, colocamos en ambos archivos de validaciones:

```
app/Http/Requests/Category/Store.php
app/Http/Requests/Category/Put.php
```

```
// ***
public function prepareForValidation()
{
    if(str($this->slug)->trim() == "")
        $this->merge([
            'slug' => str($this->title)->slug()
        ]);
}
// ***
```

Y con esto, si desde el formulario no es especificado en campo slug, entonces se genera a partir del título, si está especificado desde el formulario, entonces, no se genera uno nuevo a partir del título.

Listado

Para el listado, es igualmente sencillo; desde el controlador, obtendremos los registros que queremos mostrar; vamos a ver varias variantes, desde la más sencilla, con todos los registros, hasta la versión paginada, en la cual es necesario crear un componente de paginación.

Todos los registros

Desde el controlador, obtenemos todos los registros y los pasamos al componente en Vue:

app/Http/Controllers/Dashboard/CategoryController.php

```
// ***
public function index()
{
    $categories = Category::all();
    return inertia("dashboard/category/Index", compact("categories"));
}
// ***
```

Creamos un componente para el listado:

resources\js\pages\dashboard\category\Index.vue

En el cual, cargamos el layout general, y pasamos el **props** para las categorías, que recuerda que corresponde al listado de categorías generado desde el controlador:

```
<script>
import AppLayout from "@/Layouts/AppLayout.vue"

export default {
    components:{
        AppLayout
    },
    props:{
        categories: Array
    }
}
</script>
```

En el template, iteramos el listado mediante un **for**, y aplicamos un estilo sencillo basado en un color de fondo claro para la cabecera de la tabla, un bordeado y espaciado para las celdas de la tabla:

```
<template>
<app-layout>
    <div class="mx-4">
        <table class="w-full border">
            <thead class="dark:bg-gray-800 bg-gray-100">
                <tr>
                    <th class="p-3">Id</th>
                    <th class="p-3">Title</th>
                    <th class="p-3">Slug</th>
                    <th class="p-3">Actions</th>
```

```

        </tr>
    </thead>
    <tbody>
        <tr v-for="c in categories" :key="c.id">
            <td class="p-2">{{ c.id }}</td>
            <td class="p-2">{{ c.title }}</td>
            <td class="p-2">{{ c.slug }}</td>
            <td class="p-2">--</td>
        </tr>
    </tbody>
</table>
</div>

</app-layout>
</template>

```

Y obtenemos:

| Id | Name | Slug | Acciones |
|-----------|-------------|-------------|-----------------|
| 1 | Test2 | test-slug | -- |
| 6 | Titulo cate | slug-cate | -- |

Figura 4-5: Listado de categorías

Enlaces de navegación

Para completar más el componente de listado, vamos a colocar los enlaces de navegación; para ello, se usará el componente de Inertia llamado **Link**:

```
import { Link } from "@inertiajs/vue3"
```

Que permite navegar la web como si fuera una web de tipo SPA; el componente de **Link**, viene siendo el equivalente a los **router-view** de Vue Router; los componentes de **Link** reciben varios parámetros como puedes ver en la documentación oficial; entre los más importantes, tenemos:

- **href**, para indicar el enlace.
- **method**, para indicar el tipo de petición, GET (por defecto), POST, PUT, PATCH o DELETE.
- **data**, para indicar data extra para pasar.
- **as**, para indicar que sea de tipo button.

Para el caso de la aplicación, necesitamos dos enlaces, uno para crear y otro para editar:

```
***  
<Link :href="route('category.create')">Create</Link>  
  
<table class="w-full">  
  <thead>  
    <th class="p-3">Id</th>  
    <th class="p-3">Title</th>  
    <th class="p-3">Slug</th>  
    <th class="p-3">Actions</th>  
  </thead>  
  <tbody>  
    <tr v-for="c in categories.data" :key="c.id">  
      <td class="p-2">{{ c.id }}</td>  
      <td class="p-2">{{ c.title }}</td>  
      <td class="p-2">{{ c.slug }}</td>  
      <td class="p-2">  
        <Link :href="route('category.edit',c.id)">Edit</Link>  
      </td>  
    </tr>  
  </tbody>  
</table>  
***  
<script>  
import { Link } from "@inertiajs/vue3"  
// ***  
export default {  
  components:{  
    AppLayout,  
    Link,  
  },  
// ***  
</script>
```

Listado Paginado

Para obtener los registros paginados, desde el controlador, usamos el método provisto de paginación en Laravel básico:

```
public function index()  
{  
  $categories = Category::paginate(2);  
  return inertia("Dashboard/Category/Index",compact("categories"));  
}
```

A nivel del componente en Vue, ahora tenemos un objeto y no un **array**; recuerda que el componente de paginación luce como:

```
Illuminate\Pagination\LengthAwarePaginator {  
    #items: Illuminate\Database\Eloquent\Collection {  
        #items: array:2 [  
            0 => App\Models\Category  
            1 => App\Models\Category  
        ]  
    }  
    #perPage: 2  
    #currentPage: 1  
    #path: "http://localhost/category"  
    #query: []  
    #fragment: null  
    #pageName: "page"  
    +onEachSide: 3  
    #options: array: 2  
    #total: 5  
    #lastPage: 3  
}
```

Y el provisto por el método de **all()** como:

```
Illuminate\Database\Eloquent\Collection {  
    #items: array:5 [  
        0 => App\Models\Category  
        1 => App\Models\Category  
        2 => App\Models\Category  
        3 => App\Models\Category  
        4 => App\Models\Category  
    ]  
}
```

Por lo tanto, cambiamos la firma del props:

```
resources\js\pages\dashboard\category\Index.vue
```

```
<script>  
export default {  
    // ***  
    props:{  
        categories: Object  
    }  
}</script>
```

Así que, con esto en mente, actualizamos la firma en el **v-for**:

```
|<tr v-for="c in categories.data" :key="c.id">
```

Y tendremos el mismo diseño que teníamos anteriormente, pero paginado; sin embargo, no podemos paginar ya que, no tenemos los enlaces o componente de paginación; si usamos la extensión de Vue Devtool para el navegador:

<https://devtools.vuejs.org/>

E inspeccionamos el componente de Index, veremos el detalle de la categoría que se está recibiendo:

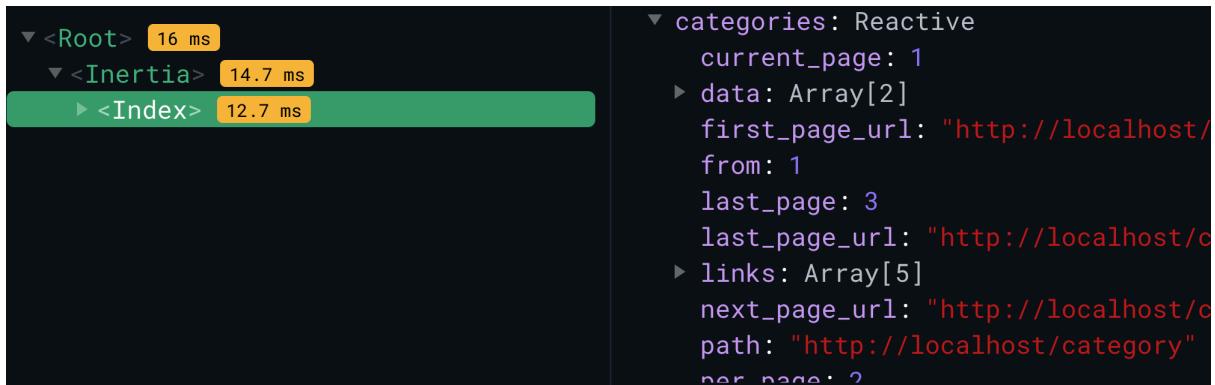


Figura 4-6: prop paginado

Crear un componente de paginación en Vue

En Inertia, no tenemos un componente de paginación para usar; así que, tenemos que crear uno.

Desde el listado paginado, debemos el apartado de **links** en los cuales se encuentran todos los enlaces para la paginación en un **array**:

```

▼ links: Array[5]
  ▼ 0: Object
    active: false
    label: "&laquo; Previous"
    url: null
  ▼ 1: Object
    active: true
    label: "1"
    url: "http://localhost/cate
  ► 2: Object
  ► 3: Object
  ► 4: Object

```

Figura 4-7: Estructura del prop paginado

Así que, los iteramos:

```
<Link v-for="l in categories.links" :key="l" v-if="!l.active" class="px-2 py-1"
:href="l.url" v-html="l.label"/>
```

Para desactivar la página actual y evitar dejar un enlace que navegue a sí mismo, podemos hacer un condicional y colocar un SPAN cuando se utilice la página actual:

```
<template v-for="l in categories.links" :key="l">
  <Link v-if="!l.active" class="px-2 py-1" :href="l.url" v-html="l.label"/>
  <span v-else class="px-2 py-1" v-html="l.label" />
</template>
```

Y agregar algo de estilo para indicar que es un enlace desactivado, aunque realmente es una etiqueta SPAN:

```
<template v-for="l in categories.links" :key="l">
```

```

<Link v-if="!l.active" class="px-2 py-1" :href="l.url" v-html="l.label"/>
<span v-else class="px-2 py-1 cursor-pointer text-gray-500" v-html="l.label" />
</template>
<script>
```

Desde Vue, tenemos una etiqueta llamada "Component" con la cual podemos renderizar componentes de manera dinámica en base a una condición evaluada en con el prop de **is**:

```

import Foo from './Foo.vue'
import Bar from './Bar.vue'

export default {
  components: { Foo, Bar },
  data() {
    return {
      view: 'Foo'
    }
  }
}
</script>

<template>
  <component :is="view" />
</template>
```

Con este componente en Vue, podemos renderizar ya sea componente de Vue o HTML directamente desde la definición del mismo:

```
<Component v-for="l in categories.links" :key="l" :is="!l.active ? 'Link' : 'span'" 
class="px-2 py-1" :class="!l.active ? '' : 'text-gray-500 cursor-pointer'" :href="l.url"
v-html="l.label" />
```

Y con esto, tenemos el mismo resultado que antes, pero queda más limpia la sintaxis.

Ahora, para poder reutilizar el componente de manera global, creamos un nuevo componente que reciba como parámetro el listado paginado mediante un prop:

resources/js/Shared/Pagination.vue

```

<template>
  <!-- <Link class="px-2 py-1" v-if="!l.active" :href="l.url">{{ l.label }}</Link>
    <span class="px-2 py-1 cursor-pointer text-gray-500" v-else>{{ l.label }}</span>
-->
  <template v-for="l in links" v-bind:key="l">

    <Component v-html=`${l.label}` class="px-2 py-1" :is="!l.active ? 'Link' :
'span'"
```

```

        :href="l.url == null ? '#' : l.url" :class="!l.active ? '' : 'cursor-pointer
text-gray-500'"
      />
    </template>
</template>
<script>
import { Link } from '@inertiajs/vue3';
export default {
  components: {
    Link
  },
  props: {
    links: Array
  }
}
</script>

```

Desde el listado de categorías, usamos este componente:

resources\js\pages\dashboard\category\Index.vue

```

***          <pagination :links="categories.links" />
  </app-layout>
</template>
***
import Pagination from '@/Shared/Pagination.vue'

export default {
  components:{ AppLayout,
    Link,
    Pagination
  },
// ***

```

Y obtendremos:

<http://localhost/category?page=1>

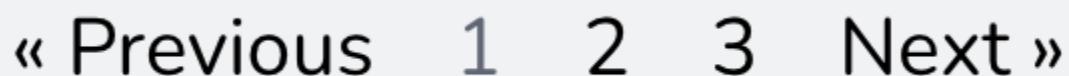


Figura 4-8: Paginación personalizada

Y con esto, tenemos un componente de paginación personalizado que podemos usar a lo largo de la aplicación.

Para evitar una advertencia al usar el v-html:

```
[vue/no-v-text-v-html-on-component]
Using v-html on component may break component's content.eslint-plugin-vu
```

Puedes usar:

```
<Component class="px-2 py-1" :is="!l.active ? 'Link' : 'span'"
    :href="l.url == null ? '#' : l.url" :class="!l.active ? '' : 'cursor-pointer
text-gray-500'"
    >{{l.label}}</Component>
```

Progress bar

Otro componente que no puede faltar en una web SPA es el de progress bar o barra de progreso; que es el que aparece cuando estamos realizando alguna operación que requiere cierto cálculo o tiempo en resolverse; usualmente se colocan al hacer una petición a Internet para obtener datos o para pasar de una página a otra.

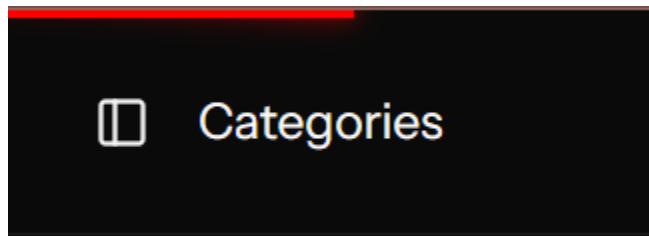


Figura 4-9: Barra de progreso

El uso de la barra de progreso ya viene configurado por defecto en el proyecto en Inertia:

resources/js/app.ts

```
/**
createInertiaApp({
    title: (title) => `${title} - ${appName}`,
    resolve: (name) => resolvePageComponent(`./pages/${name}.vue`,
import.meta.glob<DefineComponent>('./pages/**/*.{vue,js,ts,jsx,tsx}') ),
    setup({ el, App, props, plugin }) {
        createApp({ render: () => h(App, props) })
            .use(plugin)
            .use(ZiggyVue)
            .mount(el);
    },
    progress: {
```

```
        color: '#4B5563',
    },
});
```

En la cual, puedes personalizar, por ejemplo, el color; al desarrollar en un proyecto en un proyecto en local, es difícil ver la misma; para poder hacer algunas pruebas, vamos a diferir la carga de alguna página algunos segundos:

```
public function edit(Category $category)
{
    sleep(3);
    return inertia("Dashboard/Category/Edit", compact('category'));
}
```

Desde el listado anterior, si intentamos ingresar a la edición de una categoría, verás la barra de progreso.

Podemos personalizar la misma de la siguiente manera:

resources/js/app.ts

```
createInertiaApp({
    progress: {
        // The delay after which the progress bar will appear
        // during navigation, in milliseconds.
        delay: 250,

        // The color of the progress bar.
        color: '#29d',

        // Whether to include the default NProgress styles.
        includeCSS: true,

        // Whether the NProgress spinner will be shown.
        showSpinner: false,
    },
    // ...
})
```

- **delay**, para retrasar cuando aparece la barra de progreso.
- **color**, para cambiar el color de la barra de progreso y spinner en caso de que esté establecido.
- **includeCSS**, para indicar si quieres usar el CSS provisto por defecto, o vas a incluir uno propio, si lo colocas en falso y no incorporas un CSS, no aparecerá la barra.
- **showSpinner**, para mostrar paralelamente, un icono de carga.

Si utilizamos la configuración anterior, tendremos:

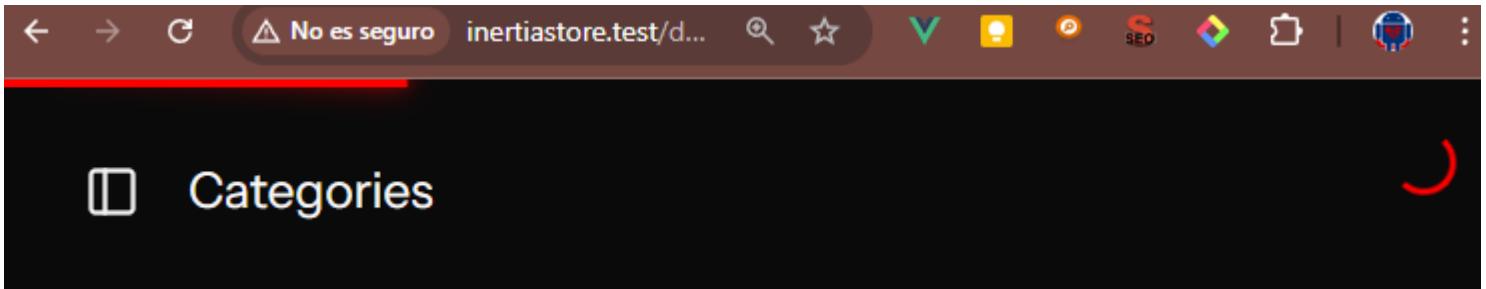


Figura 4-10: Barra de progreso con sprite personalizada

Eliminar un registro

Vamos a implementar la funcionalidad para eliminar una categoría; colocamos un nuevo enlace usando el componente de Link e indicamos el método de tipo DELETE en el TD de las acciones en el listado:

resources\js\pages\dashboard\category\Index.vue

```
***  
<td class="p-2">  
    <Link :href="route('category.edit', c.id)">Edit</Link>  
    <Link method="DELETE" :href="route('category.destroy', c.id)">Delete</Link>  
</td>  
***
```

Y desde el controlador, es exactamente el mismo código usado en Laravel básico, que es llamar al método de **delete()** de Eloquent:

app/Http/Controllers/Dashboard/CategoryController.php

```
public function destroy(Category $category)  
{  
    $category->delete();  
}
```

Si vemos en la consola del navegador desde el componente de **Index.vue** de las categorías, veremos una advertencia como la siguiente:

```
...links is discouraged as it causes "Open Link in New Tab/Window" accessibility issues.  
Instead, consider using a more appropriate element, such as a <button>.
```

Tal cual indica, renderizar un enlace para este tipo de operaciones no es lo recomendado ya que, el usuario puede abrir una pestaña en el navegador; para evitar este comportamiento, hay que convertir el enlace a un botón; para ello:

```
<Link as="button" type="button" method="DELETE" class="text-sm text-red-400  
hover:text-red-700 ml-2" :href="route('category.destroy', c.id)">Delete</Link>
```

Estilos y contenedores

En este apartado, se va a trabajar en mejorar la presentación del módulo CRUD para las categorías, tomando como referencia el estilo que existe actualmente en la aplicación generada en Inertia.

Enlaces de acción

También colocaremos un estilo a los enlaces de acción del listado:

1. Para el enlace de crear, como es usualmente el botón más visible, se usará las mismas reglas CSS para los botones que tenemos en la aplicación.
2. Para los enlaces de acción desde la tabla, un color de texto y tamaño basta para nuestro objetivo:

resources\js\pages\dashboard\category\Index.vue

```
<Link class="link-button-default my-3" :href="route('category.create')">Create</Link>
***
<td class="p-2">
    <Link class="text-sm text-purple-400 hover:text-purple-700"
:href="route('category.edit', c.id)">Edit</Link>
</td>
***
```

Definimos las reglas para la clase botón creada anteriormente:

resources/css/app.css

```
***
.link-button-default{
    @apply flex-none w-fit whitespace nowrap rounded-md text-sm font-medium
ring-offset-background transition-colors focus-visible:outline-none focus-visible:ring-2
focus-visible:ring-ring focus-visible:ring-offset-2 disabled:pointer-events-none
disabled:opacity-50 bg-primary text-primary-foreground shadow hover:bg-primary/90 h-9 px-4
py-2
}
```

Y tendremos:

| Category List | | | |
|---------------|--------|--------|---|
| ID | Title | Slug | Actions |
| 8 | Test 1 | test-1 | Edit Delete |
| 9 | Test 2 | test-2 | Edit Delete |

Figura 4-11: Enlaces en tabla con estilo

Para definir estos estilos, copiamos el actual de componentes como el de resources\js\components\ui\button\Button.vue

Paginación

Otro diseño que no podemos pasar por alto, viene siendo el de la paginación, al cual aplicaremos un diseño simple, basado en un bordeado redondeado como el siguiente:

resources/js/Shared/Pagination.vue

```
<template>
  <div class="flex mt-5">
    <Component v-for="l in links.links" :key="l" :is="!l.active ? 'Link' :'span'">
      <div class="px-4 py-2 border border-gray-300 text-sm font-medium rounded-md text-gray-700 bg-white hover:bg-gray-50 ml-2" :class="!l.active ? '' : 'text-gray-500 cursor-pointer'" href="l.url ?? ''" v-html="l.label" />
    </div>
  </div>
</template>
```

Y tendremos:



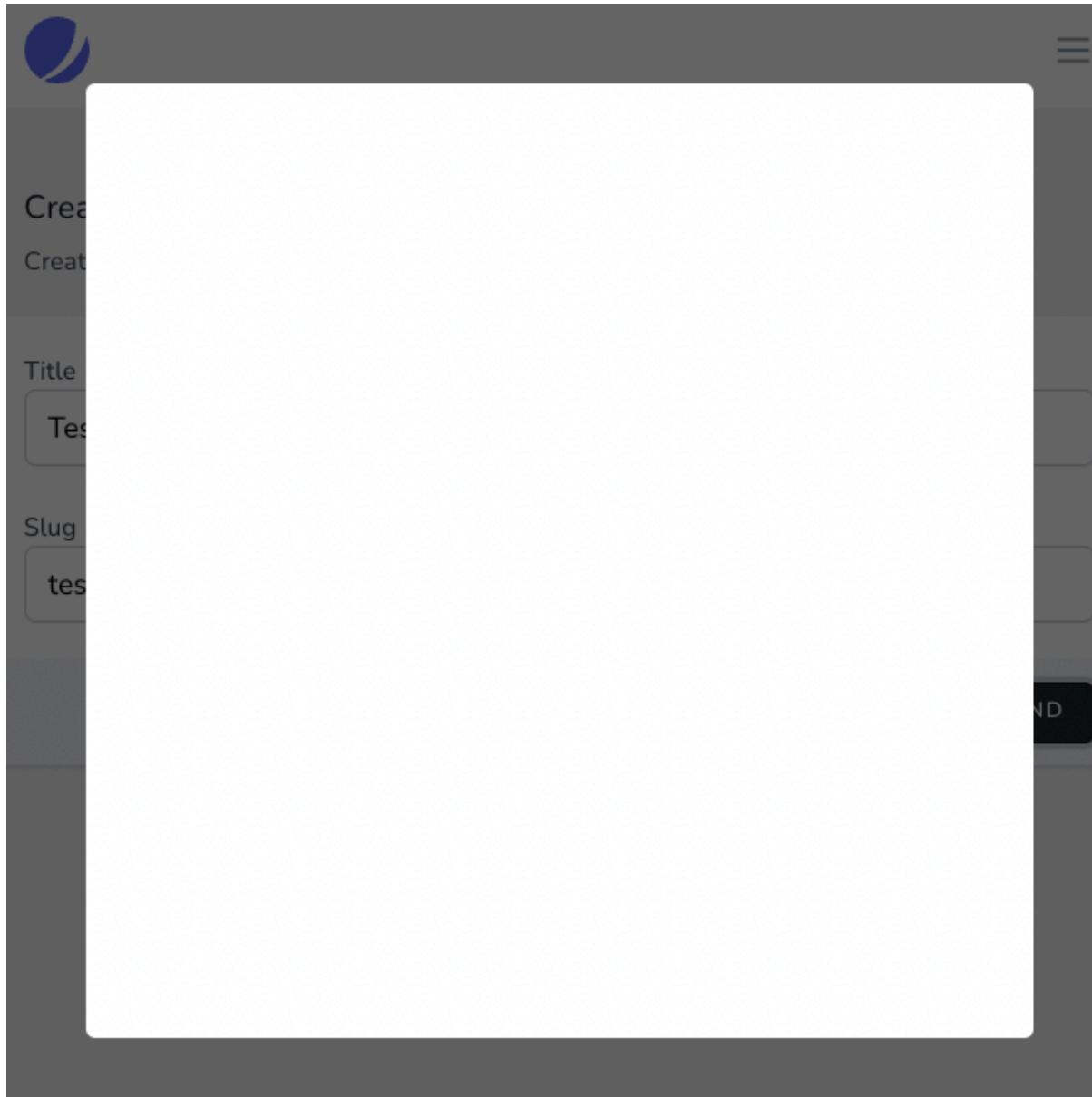
Figura 4-12: Paginación con estilo

Con esto terminamos el esquema básico de un CRUD para las categorías; puedes consultar el código fuente en:

https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.1_12

Capítulo 5: Redirecciones y mensajes flash

Un problema que existe actualmente en la aplicación es que cada vez que se completa la operación de creación, actualización o eliminación de un registro, queda presente la siguiente pantalla, una pantalla en blanco:



O directamente la ventana no muestra ninguna actualización o cambio. Esto es porque, en cada una de los controladores que resuelven las operaciones anteriores, no devuelve ninguna vista ni realiza alguna redirección; para estas operaciones finales, lo normal es hacer una redirección a otra página, generalmente la de index; también abordaremos el uso de los mensajes flash para confirmar operaciones.

Redirecciones

En Laravel Inertia, las redirecciones son exactamente las mismas que en Laravel básico; así que, para ello, podemos hacer uso de cualquiera de los mecanismos que existen en la redirección; ya sea mediante los Facades:

```
|Redirect::route('category.index')
```

O funciones de ayuda:

```
to_route('category.index')
redirect()->route('category.index')
```

Así que, colocamos las redirecciones en cada uno de los métodos que no devuelven una vista:

app/Http/Controllers/Dashboard/CategoryController.php

```
class CategoryController extends Controller
{
    public function store(Store $request)
    {
        Category::create($request->validated());
        return to_route('category.index');
    }

    public function update(Put $request, Category $category)
    {
        $category->update($request->validated());
        return redirect()->route('category.index');
    }

    public function destroy(Category $category)
    {
        $category->delete();
        return to_route('category.index');
    }
}
```

Mensajes flash

Para hacer el uso de los mensajes flash en Laravel Inertia, aunque desde el controlador, podemos usar las operaciones típicas para establecer los mensajes de sesión tipo flash:

```
$request->session()->flash('message', 'Message');
```

O mediante la redirección:

```
with('message', 'Message');
```

Pero, al usar componentes en Vue y no vistas de blade, tenemos que hacer un paso extra para pasar estos datos vía sesión flash.

Middleware de peticiones de Inertia

En Inertia, tenemos un middleware que se encarga de interceptar las peticiones del usuario; desde este middleware, podemos establecer datos globales a los componentes en Vue; por lo tanto, si quieras pasar datos globales a los componentes desde la base de datos, archivos, configuraciones, sesión, etc:

```
return [
    ...parent::share($request),
    'name' => config('app.name'),
    'quote' => ['message' => trim($message), 'author' => trim($author)],
    'auth' => [
        'user' => $request->user(),
    ],
    'ziggy' => [
        ...new Ziggy)->toArray(),
        'location' => $request->url(),
    ],
];
```

Debes de usar este middleware; para el caso de los mensajes flash; puedes usar un esquema como el siguiente:

app/Http/Middleware/HandleInertiaRequests.php

```
public function share(Request $request): array
{
    return [
        ***
        'flash' => [
            'message' => $request->session()->get('message')
        ]
    ];
}
```

En el ejemplo anterior, estamos suponiendo que vamos a usar una sola clasificación de mensajes flash, pero, puedes usar cuantos necesites; por ejemplo:

app/Http/Middleware/HandleInertiaRequests.php

```
public function share(Request $request): array
{
    return [
        ***
        'flash' => [
            'message' => $request->session()->get('message'),
            'status' => $request->session()->get('status')
            ***
            'other' => $request->session()->get('other')
        ]
    ];
}
```

```
        ],
    ];
}
```

En el ejemplo anterior, usamos un **array de array** para organizar los mensajes flash al considerar que es un esquema escalable y organizado para poder definir otros esquemas que quieras pasar de manera global.

Objeto \$page

El objeto **\$page** es una variable que se puede acceder de manera global en todos los componentes en Vue, tiene datos variados como:

```
{
  "component": "Dashboard/Post/Index",
  "props": {
    "jetstream": {
      "canCreateTeams": true,
      "canManageTwoFactorAuthentication": true,
      "canUpdatePassword": true,
      "canUpdateProfileInformation": true,
      "hasEmailVerification": false,
      "flash": [
        ],
      "hasAccountDeletionFeatures": true,
      "hasApiFeatures": true,
      "hasTeamFeatures": true,
      "hasTermsAndPrivacyPolicyFeature": true,
      "managesProfilePhotos": true
    },
    "user": {
      "id": 1,
      "name": "andres",
      "email": "admin@gmail.com",
      "email_verified_at": null,
      "two_factor_confirmed_at": null,
      "current_team_id": 1,
      "profile_photo_path": null,
      "created_at": "2022-05-31T15:10:22.000000Z",
      "updated_at": "2022-05-31T15:28:19.000000Z",
      "profile_photo_url":
      "https://ui-avatars.com/api/?name=a&color=7F9CF5&background=EBF4FF",
      "current_team": {
        "id": 1,
        "user_id": 1,
        "name": "andres's Team",
        "personal_team": true,
    }
  }
}
```

```

    "created_at": "2022-05-31T15:10:22.000000Z",
    "updated_at": "2022-05-31T15:10:22.000000Z"
},
"all_teams": [
{
    "id": 1,
    "user_id": 1,
    "name": "andres's Team",
    "personal_team": true,
    "created_at": "2022-05-31T15:10:22.000000Z",
    "updated_at": "2022-05-31T15:10:22.000000Z"
}
],
"two_factor_enabled": false
},
"errorBags": [
],
"errors": {

},
"data": ...
"url": "/post",
"version": "207fd484b7c2ceeff7800b8c8a11b3b6"
}

```

Y por supuesto, los datos que establezcamos en el middleware de **HandleInertiaRequests** se encuentran aquí también.

Caso práctico

Vamos a establecer los mensajes flash de confirmación en cada una de las redirecciones definidas anteriormente:

app/Http/Controllers/Dashboard/CategoryController.php

```

class CategoryController extends Controller
{

    public function store(Store $request)
    {
        Category::create($request->validated());
        return to_route('category.index')->with('message', "Created category successfully");
    }

    public function update(Put $request, Category $category)

```

```

    {
        $category->update($request->validated());
        return redirect()->route('category.index')->with('message', "Updated category
successfully");
    }

    public function destroy(Category $category)
    {
        $category->delete();
        return to_route('category.index')->with('message', "Deleted category successfully");
    }
}

```

Para usarlos desde los componentes en Vue, en vez definirlos manualmente en cada uno de los componentes que queramos que tengan el mensaje de confirmación, lo podemos colocar en un componente global como el de **AppLayout**:

resources/js/Layouts/AppLayout.vue

```

***  

<main>  

    {{ $page.props.flash.message }}  

    <slot />  

</main>  

***
```

Y al realizar cualquier operación de eliminación, creación o actualización en este caso, tendremos:

Updated category successfully

Estilo para el contenedor del mensaje flash

Vamos a dar un estilo para el mensaje de confirmación anterior; para eso:

resources/js/layouts/AppLayout.vue

```

<main>
    <div
        v-if="$page.props.flash.message"
        class="
            container my-2 bg-purple-300 text-purple-800 px-4 py-3 rounded shadow-sm"
        >
            {{ $page.props.flash.message }}
    </div>
</main>
```

Y tendremos:

Deleted category successfully

Remover contenedor pasado un tiempo

Desaparecer el contenedor de mensaje flash pasado unos segundos

Es común que queramos desaparecer el mensaje de confirmación pasados unos segundos después, es un comportamiento típico en este tipo de componentes; para ello, vamos a crear una propiedad y un método que se ejecute pasado un tiempo establecido que es el que se encarga de ocultar el componente del mensaje de confirmación; es importante notar que, se está usando la sintaxis de la API de composición con la opción de **<script setup>** de Vue:

<https://vuejs.org/api/sfc-script-setup.html>

En el cual, en pocas palabras el método de **setup()** viene implícito y podemos tener una organización más sencilla al momento de definir propiedades, métodos y cualquier otra de las características que quieras usar de Vue; en esencia, podemos definir todo en un solo bloque, tal cual tenemos en cada uno de los componentes de Vue definidos por Inertia por defecto.

Para crear la propiedad (variable en este esquema) que se encarga de ocultar o mostrar el contenedor del mensaje de confirmación y el método de **setTimeout()**, que se encarga de ocultar el contenedor del mensaje de confirmación mediante la variable anterior, tenemos:

resources/js/layouts/AppLayout.vue

```
<script setup>
import { ref } from 'vue';
/**/

// flash message
const visibleFlashContainer = ref(true);

const hideFlashMessage = () => {
  setTimeout(() => (visibleFlashContainer.value = false), 4000);
  return true;
};
// flash message
</script>
```

En el ejemplo anterior, ocultamos el método de confirmación pasado 4 segundos, pero lo puedes personalizar a gusto, como puedes ver, lo único que hace es establecer en falso la variable que se va a usar desde el template:

resources/js/layouts/AppLayout.vue

```
***  

<main>  

  <transition>  

    <div v-if="visibleFlashContainer">  

      <div  

        v-if="$page.props.flash.message && hideFlashMessage()"  

        class="  

          container  

          my-2  

          bg-purple-300  

          text-purple-800  

          px-4  

          py-3  

          rounded  

          shadow-sm  

        "  

      >  

        {{ $page.props.flash.message }}  

      </div>  

    </div>  

  </transition>  

  <slot />  

</main>  

***
```

Y el CSS para ocultar el componente mediante la opacidad:

resources\css\app.css

```
/* transition for vue */  

.fade-enter-active, .fade-leave-active {  

  transition: opacity .5s  

}  

.fade-enter,  

.fade-leave-to {  

  opacity: 0  

}  

/* transition for vue */
```

El uso de transition es opcional y solo se emplea para ocultar el componente de manera suave y no brusca.

Para poder invocar el método de timer anterior (**hideFlashMessage()**) se coloca desde el condicional que se encarga de mostrar el contenedor del mensaje de confirmación.

Puedes consultar el código fuente en:

https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.2_12

Capítulo 6: CRUD para los posts

Este capítulo es de reforzamiento y tiene como intención que el lector ponga en práctica lo evaluado en todos los capítulos anteriores construyendo un CRUD un poco más complejo que el de las categorías, ya que, la entidad de los posts cuenta con más campos a administrar.

En definitiva, es un capítulo en la cual debes de replicar y adaptar el CRUD de las categorías presentado anteriormente, para los POST; de igual manera, se deja a continuación el código generado para el CRUD de los POST para que lo tengas de referencia.

Controlador

El controlador de los POST es igual al de las categorías, pero en vez de administrar categorías, se administran post, por lo demás, en las funciones controladoras que se encargan de retornar la vista del formulario, obtienen el listado de las categorías para crear el listado de categorías usado en la relación de categoría con el post:

app/Http/Controllers/Dashboard/PostController.php

```
<?php

namespace App\Http\Controllers\Dashboard;

use App\Http\Controllers\Controller;
use App\Http\Requests\Post\Put;
use App\Http\Requests\Post\Store;
use App\Models\Category;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{

    public function index()
    {
        $posts = Post::paginate(2);
        return inertia("Dashboard/Post/Index", compact("posts"));
    }

    public function create()
    {
        $categories = Category::get();
        return inertia("Dashboard/Post/Create", compact('categories'));
    }

    public function store(Store $request)
    {
```

```

Post::create($request->validated());
return to_route('post.index')->with('message', "Created post successfully");
}

public function edit(Post $post)
{
    $categories = Category::get();
    return inertia("Dashboard/Post/Edit", compact('post', 'categories'));
}

public function update(Put $request, Post $post)
{
    $post->update($request->validated());
    return redirect()->route('post.index')->with('message', "Updated post
successfully");
}

public function destroy(Post $post)
{
    $post->delete();
    return to_route('post.index')->with('message', "Deleted post successfully");
}
}

```

Ruta

Se crea la ruta de tipo CRUD para los posts:

```

Route::group(['middleware' => [
    'auth:sanctum',
    config('jetstream.auth_session'),
    'verified',
]], function () {
    Route::resource('/category', App\Http\Controllers\Dashboard\CategoryController::class);
    Route::resource('/post', App\Http\Controllers\Dashboard\PostController::class);
});

```

Validaciones

Y se crean los nuevos campos de validación para el post, que tiene adicionales al título y al slug; como lo son, la fecha, descripción, texto de la publicación, si está posteado, el tipo y la categoría; puedes personalizar las validaciones como gustes.

Las validaciones usadas en la fase de creación:

app/Http/Requests/Post/Store.php

```

<?php

namespace App\Http\Requests\Post;

use Illuminate\Foundation\Http\FormRequest;

class Store extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize(): bool
    {
        return true;
    }

    public function prepareForValidation()
    {
        if(str($this->slug)->trim() == "")
            $this->merge([
                'slug' => str($this->title)->slug()
            ]);
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string, mixed>
     */
    public function rules(): array
    {
        return [
            "title" => "required|min:5|max:255",
            "slug" => "required|min:5|max:255|unique:categories",
            "date" => "required",
            "description" => "required",
            "text" => "required",

            "posted" => "required",
            "type" => "required",
            "category_id" => "required",
        ];
    }
}

```

}

Las validaciones usadas en la fase de actualización:

app/Http/Requests/Post/Put.php

```
<?php

namespace App\Http\Requests\Post;

use Illuminate\Foundation\Http\FormRequest;

class Put extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     *
     * @return bool
     */
    public function authorize():bool
    {
        return true;
    }

    public function prepareForValidation()
    {
        if(str($this->slug)->trim() == "")
            $this->merge([
                'slug' => str($this->title)->slug()
            ]);
    }

    /**
     * Get the validation rules that apply to the request.
     *
     * @return array<string, mixed>
     */
    public function rules():array
    {
        return [
            "title" => "required|min:5|max:255",
            "slug" => "required|min:5|max:255|unique:posts,slug,".$this->route("post")->id,
            "date" => "required",
            "description" => "required",
            "text" => "required",
        ];
    }
}
```

```

        "posted" => "required",
        "type" => "required",
        "category_id" => "required",
    ];
}
}

```

Listado

Para el listado, se usa el mismo que el empleado en las categorías, pero, iterando la relación de post:

resources/js/Pages/dashboard/post/Index.vue

```

<template>
  <AppLayout :breadcrumbs="breadcrumbs">

    <Link class="link-button-default mx-4 my-3"
:href="route('post.create')">Create</Link>

    <div class="mx-4">
      <table class="w-full border">
        <thead class="dark:bg-gray-800 bg-gray-100">
          <tr class="border-b">
            <th class="p-3">Id</th>
            <th class="p-3">Title</th>
            <th class="p-3">Slug</th>
            <th class="p-3">Actions</th>
          </tr>
        </thead>
        <tbody>
          <tr class="border-b" v-for="p in posts.data" :key="p.id">
            <td class="p-2">{{ p.id }}</td>
            <td class="p-2">{{ p.title }}</td>
            <td class="p-2">{{ p.slug }}</td>
            <td class="p-2">
              <Link class="text-sm text-purple-400 hover:text-purple-700"
                :href="route('post.edit', p.id)">Edit</Link>
              <Link as="button" type="button" method="DELETE"
                class="text-sm text-red-400 hover:text-red-700 ml-2"
                :href="route('post.destroy', p.id)">Delete</Link>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </AppLayout>

```

```

        <pagination class="my-4" :links="posts" />
    </AppLayout>
</template>

<script>

import { Link } from "@inertiajs/vue3"
import AppLayout from '@/layouts/AppLayout.vue';
import Pagination from '@/shared/Pagination.vue';

export default {
    props: {
        posts: Object
    },
    components: {
        Pagination,
        AppLayout,
        Link
    },
    setup() {

        const breadcrumbs = [
            {
                title: 'Posts',
                // href: '/dashboard/category',
            },
        ];
        return {
            breadcrumbs
        };
    }
}
</script>

```

Y tendremos:

| Listado de publicaciones | | | |
|---|----------|--------|---|
| Id | Title | Slug | Actions |
| 1 | Test 2.1 | test-2 | Edit Delete |
| « Previous 1 Next » | | | |

Figura 6-1: Listado de publicaciones

Creación

El formulario de creación usados nuevos campos para:

- Un input, usando el componente **TextInput** de tipo date, para la fecha.
- TEXTAREA, para la descripción y texto de la publicación.
- SELECT, para el listado de tipos y posted, que son valores fijos y de categorías para la relación con dicha tabla; para las categorías, como es una variable adicional que se pasa desde el controlador, se debe de registrar en los **props** de Vue.

resources/js/pages/dashboard/post/Create.vue

```
<template>
  <AppLayout :breadcrumbs="breadcrumbs">
    <div class="px-4 py-6 max-w-xl">
      <HeadingSmall title="Create Post" description="Form to create post" />
      <form class="my-6" @submit.prevent="submit">

        <Label>Title</Label>
        <Input id="title" v-model="form.title" type="text" class="block w-full mt-1" autofocus />
        <InputError :message="errors.title" class="mt-2" />

        <Label>Slug</Label>
        <Input id="slug" v-model="form.slug" type="text" class="block w-full mt-1" />
        <InputError :message="errors.slug" class="mt-2" />

        <Label>Date</Label>
        <Input id="date" v-model="form.date" type="date" class="block w-full mt-1" />
        <InputError :message="errors.date" class="mt-2" />

        <Label>Text</Label>
        <textarea id="text" v-model="form.text" class="block w-full mt-1 border-gray-300 rounded-md"></textarea>
        <InputError :message="errors.text" class="mt-2" />

        <Label>Description</Label>
        <textarea id="description" v-model="form.description" class="block w-full mt-1 border-gray-300 rounded-md"></textarea>
        <InputError :message="errors.description" class="mt-2" />

        <Label>Posted</Label>
        <select v-model="form.posted" class="rounded-md w-full border-gray-300 my-2">
          <option value="not">No</option>
```

```

        <option value="yes">Yes</option>
    </select>
    <InputError :message="errors.posted" class="mt-2" />

    <Label>Type</Label>
    <select v-model="form.type" class="rounded-md w-full border-gray-300">
        <option value="advert">Advert</option>
        <option value="post">Post</option>
        <option value="course">Course</option>
        <option value="movie">Movie</option>
    </select>

    <InputError :message="errors.type" class="mt-2" />

    <Label>Category</Label>
    <select v-model="form.category_id" class="rounded-md w-full
border-gray-300">
        <option v-for="c in categories" :value="c.id" :key="c.id">{{ c.title
}}</option>
    </select>

    <InputError :message="errors.category_id" class="mt-2" />

    <div>
        <Button class="mt-2" :disabled="form.processing"
type="submit">Send</Button>
    </div>

    </form>
</div>
</AppLayout>
</template>

<script>

import { router, useForm } from '@inertiajs/vue3';

import AppLayout from '@/layouts/AppLayout.vue';

import Input from '@/components/ui/input/Input.vue';
import Label from '@/components/ui/label/Label.vue';
import InputError from '@/components/InputError.vue';
import HeadingSmall from '@/components/HeadingSmall.vue';

import { Button } from '@/components/ui/button';

```

```
export default {
  props: {
    errors: Object,
    categories: Array
  },
  components: {
    AppLayout,
    InputError,
    HeadingSmall,
    Input,
    Label,
    Button,
  },
  setup() {
    const form = useForm({
      title: '',
      slug: '',
      date: '',
      description: '',
      text: '',
      posted: '',
      type: '',
      category_id: ''
    });

    const breadcrumbs = [
      {
        title: 'Post',
        // href: '/dashboard/post/create',
      },
    ];

    function submit() {
      router.post(route('post.store', form))
    }

    return {
      submit, form, breadcrumbs
    };
  }
}
</script>
```

Edición

Para el de edición, sigue el mismo lineamiento que para el formulario de creación de los posts:

resources/js/pages/dashboard/post/Edit.vue

```
<template>
  <AppLayout :breadcrumbs="breadcrumbs">
    <div class="px-4 py-6 max-w-xl">
      <HeadingSmall title="Create Post" description="Form to create post" />
      <form class="my-6" @submit.prevent="submit">

        <Label>Title</Label>
        <div>
          <Input id="title" v-model="form.title" type="text" class="block w-full mt-1" autofocus />
          <InputError :message="errors.title" class="mt-2" />
        </div>

        <Label>Slug</Label>
        <div>
          <Input id="slug" v-model="form.slug" type="text" class="block w-full mt-1" />
          <InputError :message="errors.slug" class="mt-2" />
        </div>
        <Label>Date</Label>
        <div>
          <Input id="date" v-model="form.date" type="date" class="block w-full mt-1" />
          <InputError :message="errors.date" class="mt-2" />
        </div>

        <Label>Text</Label>
        <div>
          <textarea id="text" v-model="form.text"
            class="block w-full mt-1 border-gray-300 rounded-md"></textarea>
          <InputError :message="errors.text" class="mt-2" />
        </div>

        <Label>Description</Label>
        <div>
          <textarea id="description" v-model="form.description"
            class="block w-full mt-1 border-gray-300 rounded-md"></textarea>
          <InputError :message="errors.description" class="mt-2" />
        </div>
```

```

<Label>Posted</Label>
<div>
    <select v-model="form.posted" class="rounded-md w-full border-gray-300
my-2">
        <option value="not">No</option>
        <option value="yes">Yes</option>
    </select>
    <InputError :message="errors.posted" class="mt-2" />
</div>

<Label>Type</Label>
<div>
    <select v-model="form.type" class="rounded-md w-full border-gray-300">
        <option value="advert">Advert</option>
        <option value="post">Post</option>
        <option value="course">Course</option>
        <option value="movie">Movie</option>
    </select>

    <InputError :message="errors.type" class="mt-2" />
</div>

<Label>Category</Label>
<div>
    <select v-model="form.category_id" class="rounded-md w-full
border-gray-300">
        <option v-for="c in categories" :value="c.id" :key="c.id">{{
c.title }}</option>
    </select>

    <InputError :message="errors.category_id" class="mt-2" />
</div>

<div>
    <Button class="mt-2" :disabled="form.processing"
type="submit">Send</Button>
</div>

</form>
</div>
</AppLayout>
</template>

<script>

import { router, useForm } from '@inertiajs/vue3';

```

```

import AppLayout from '@/layouts/AppLayout.vue';

import Input from '@/components/ui/input/Input.vue';
import Label from '@/components/ui/label/Label.vue';
import InputError from '@/components/InputError.vue';
import HeadingSmall from '@/components/HeadingSmall.vue';

import { Button } from '@/components/ui/button';
export default {
    props: {
        errors: Object,
        post: Object,
        category: Object,
        categories: Array
    },
    components: {
        AppLayout,
        InputError,
        HeadingSmall,
        Input,
        Label,
        Button,
    },
    setup(props) {
        const form = useForm({
            title: props.post.title,
            slug: props.post.slug,
            date: props.post.date,
            description: props.post.description,
            text: props.post.text,
            posted: props.post.posted,
            type: props.post.type,
            category_id: props.post.category_id,
        })
        const breadcrumbs = [
            {
                title: 'Edit Category: ' + props.post.title,
                // href: '/dashboard/post/create',
            },
        ];
        function submit() {
            router.put(route("post.update", props.post.id), form)
        }
    }
}

```

```
        return {
            submit, form, breadcrumbs
        };
    }
}
</script>
```

Y tendremos:

The form consists of several input fields:

- Title:** Test 2.1
- Slug:** test-2
- Date:** 16/06/2022
- Text:** Text
- Description:** Description
- Posted:** No
- Type:** Course
- Category:** asasasasas

At the bottom right of the form is a large, dark blue "SEND" button.

Figura 6-2: Formulario para crear y actualizar posts

Fusionar creación y actualización en un solo componente de Vue

La reutilización es un factor a tener en cuenta al momento de la creación de cualquier tipo de aplicaciones; poder reutilizar los formularios que usamos para crear y editar publicaciones que son prácticamente iguales, es un punto a favor y se puede realizar con pocos cambios; desde los controladores que devuelven las vistas de editar y crear, vamos a devolver una misma vista llamada Save:

app/Http/Controllers/Dashboard/PostController.php

```
public function create()
{
    $categories = Category::get();
    return inertia("Dashboard/Post/Save", compact('categories'));
}

public function edit(Post $post)
{
    $categories = Category::get();
    return inertia("Dashboard/Post/Save", compact('post', 'categories'));
}
```

Creamos la vista de Save a partir de la de **Edit.vue** para las publicaciones:

resources/js/Pages/Dashboard/Post/Save.vue

Para los props, vamos a cambiar de este esquema

```
props: {
    errors: Object,
    post: Object,
    categories: Object,
},
```

Al siguiente para el prop del post:

```
props: {
    errors: Object,
    post: {
        type: Object,
        default: {
            id: "",
            title: "",
            slug: "",
            date: "",
            description: "",
            text: ""
        }
    }
},
```

```

        type: '',
        posted: '',
        category_id: '',
    },
},
categories: Object,
},

```

Con esto, estamos dando valores por defecto a un post cuando el mismo no sea recibido para la operación de creación.

Ahora, tenemos que determinar qué ruta vamos a llamar según si vamos a crear o actualizar:

```

function submit() {
  if (props.post.id != '') {
    router.put(route("post.update", props.post.id), form)
  } else {
    router.post(route("post.store"), form)
  }
}

```

Con estos sencillos pasos, podemos usar una misma vista o componente en Vue tanto para crear como para actualizar.

Extra: Composition API: setup en Atributo

Hasta el momento, hemos usado el modo de composición para desarrollar los componentes de Vue mediante el **setup()** como métodos al ser más sencillo, pero, si vemos, los componentes que se encuentran implementados, veremos que se encuentran a nivel de atributos, así que, veremos la traducción de uno de los componentes para que sepas como es la traducción aunque es importante acotar que esto no es un curso sobre Vue; el componente de:

resources\js\pages\dashboard\category\Create.vue

Queda como:

```

<template>
  ***
</template>

<script setup>

import { router, useForm } from '@inertiajs/vue3';

import AppLayout from '@/layouts/AppLayout.vue';

```

```

import Input from '@/components/ui/input/Input.vue';
import Label from '@/components/ui/label/Label.vue';
import InputError from '@/components/InputError.vue';
import HeadingSmall from '@/components/HeadingSmall.vue';

import { Button } from '@/components/ui/button';

const form = useForm({
  title: '',
  slug: '',
});

const props = defineProps({
  errors: Object,
});

const breadcrumbs = [
  {
    title: 'Categories',
  },
];

```

function submit() {
 console.log(form);
 router.post(route('category.store', form))
}

```

</script>

```

Como puedes apreciar, quitamos toda la estructura teniendo todo a un mismo nivel, el método setup y los datos retornados, además de las opciones de **props** y **components** son eliminados por una estructura más limpia.

Extra: Componentes con TypeScript

Si vemos los componentes que se encuentran implementados, veremos que emplean TypeScript, esto también lo podemos habilitar fácilmente en nuestros componentes para poder emplear TypeScript y por ejemplo, poder definir el tipo, por ejemplo:

resources\js\pages\dashboard\category\Create.vue

```

<script setup lang="ts">

import { router, useForm } from '@inertiajs/vue3';

import { type BreadcrumbItem } from '@/types';

```

```
***  
  
const breadcrumbs : BreadcrumbItem[] = [  
  {  
    title: 'Categories',  
    href: '',  
    // href: '/dashboard/category/create',  
  },  
];  
***
```

Como puedes apreciar, es simplemente colocar el atributo de lang y el valor de ts en el script; puedes probar quitar el atributo mencionado y veras que al interpretar el componente, da un error por la definición del tipo que establecido laa las migajas de pan, indicando que es una característica de TypeScript y no de JavaScript, especificando una interfaz y de esta forma poder garantizar una misma estructura para el tipo de dato definido para las migajas de pan.

Puedes consultar el código fuente en:

https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.3_12

Capítulo 7: Upload

En este capítulo, vamos a conocer cómo funciona el upload o carga de archivos en Laravel Inertia; como puedes suponer, todo lo que tiene que ver con el backend en Laravel, se mantiene exactamente igual, y es en el cliente, que tenemos los cambios para poder usar la integración de la carga de archivos desde Inertia.

Inicialmente, vamos a manejar la carga de archivos, específicamente la carga de imágenes de las publicaciones, de manera independiente al formulario que ya tenemos para la edición; usaremos el de **Save.vue** colocando un condicional solamente para editar:

```
app/Http/Controllers/Dashboard/PostController.php
```

```
public function edit(Post $post)
{
    $categories = Category::get();
    return inertia("Dashboard/Post/Edit", compact('post', 'categories'));
}
```

Vamos a crear el disco donde se van a cargar las imágenes de las publicaciones, que serían en la carpeta public:

```
config/filesystems.php
```

```
'public_upload' => [
    'driver' => 'local',
    'root' => public_path(),
],
```

El controlador, para la carga de imágenes es lo normal, aplicar validaciones:

```
$request->validate(
[
    'image' => 'required|mimes:jpg,jpeg,png,gif|max:10240'
])
);
```

Eliminar la imagen anterior (en caso de que exista):

```
Storage::disk("public_upload")->delete("image/post/".$post->image);
```

Generar un nombre aleatorio de la imagen:

```
time()." ".$request['image']->extension();
```

Mover a la carpeta public:

```
$request->image->move(public_path("image/post"),$filename);
```

Y actualizar el post:

```
$post->update($data);
```

Finalmente, tenemos:

app/Http/Controllers/Dashboard/PostController.php

```
***  
use Illuminate\Support\Facades\Storage;  
***  
public function upload(Request $request, Post $post)  
{  
    $request->validate(  
        [  
            'image' => 'required|mimes:jpg,jpeg,png,gif|max:10240'  
        ]  
    );  
    Storage::disk("public_upload")->delete("image/post/" . $post->image);  
    $data['image'] = $filename = time() . "." . $request['image']->extension();  
    $request->image->move(public_path("image/post"), $filename);  
    $post->update($data);  
    return to_route('post.index')->with('message', "Upload image to post successfully");  
}
```

Creamos la ruta:

routes/web.php

```
Route::group(['middleware' => [  
    'auth:sanctum',  
    'prefix' => 'dashboard',  
    config('jetstream.auth_session'),  
    'verified',  
], function () {  
    Route::resource('/category', App\Http\Controllers\Dashboard\CategoryController::class);  
    Route::resource('/post', App\Http\Controllers\Dashboard\PostController::class);  
    Route::post('/post/upload/{post}',  
    [App\Http\Controllers\Dashboard\PostController::class, 'upload'])->name('post.upload');  
});
```

En el formulario de editar, definimos un prop para la imagen:

```
const form = useForm({  
    // ***  
    category_id: props.post.category_id,  
    image: "",  
});
```

Y ahora, vamos a colocar un bloque adicional, para manejar la carga de archivos, como se comentó antes, la carga de la imagen, se realizará de manera independiente del formulario de actualización:

resources/js/Pages/Dashboard/Post/Save.vue

```

<div v-if="post.id != ''">
  <div class="px-4 py-6 max-w-xl">
    <div class="card">
      <div class="card-body">
        <div class="grid grid-cols-2 gap-2">
          <div class="col-span-6">
            <Label>Image</Label>
            <Input type="file" @input="form.image = $event.target.files[0]" />
            <InputError :message="errors.image" class="mt-2" />
            <Button @click="upload">Upload</Button>
          </div>
        </div>
      </div>
    </div>
  </div>
</div>

</AppLayout>
***

setup(props) {
  const form = useForm({
    // ***
    image: "",
  });

  function submit() {
    router.put(route("post.update"), form.id), form);
  }

  function upload() {
    router.post(route("post.upload"), form.id), form);
  }

  return { form, submit, upload };
},

```

El código anterior, es sencillo, un campo de tipo **file** con su botón, que llama a un método llamado **upload()** en la cual enviamos el formulario, tal cual hacemos cuando se actualiza un registro; es importante notar que, para usar la carga de archivos, debe ser una petición de tipo POST; si quieras usar la carga de archivos en otro tipo de peticiones como DELETE, PATCH o PUT, debes de usar la petición de tipo POST y especificar el método en cuestión (esto lo veremos más adelante):

```

router.post(route(***) , {
  _method: "put",
  // tus datos
}

```

```
| },
```

Con esto, tendremos la carga de archivos completamente funcional:

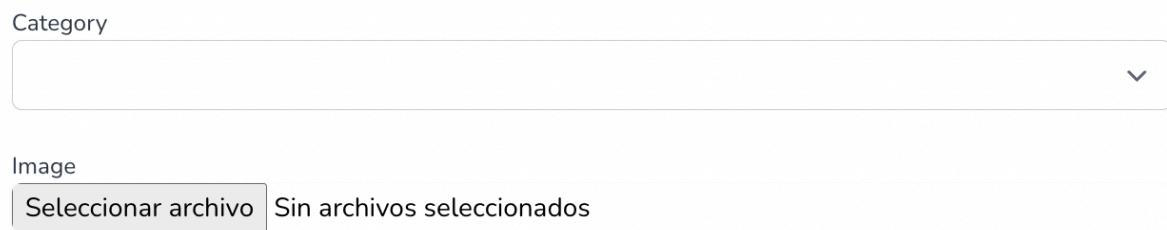


Figura 7-1: Seleccionar imagen

Y al cargar un archivo, veremos el resultado dentro del proyecto:

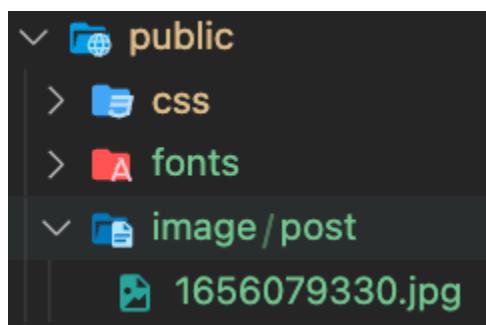


Figura 7-2: Imagen cargada

Un punto importante en el código anterior es la definición del input, que está implementada la asignación del valor mediante el evento de `@input`, obteniendo el archivo cargado por el usuario mediante `$event.target.files[0]`:

```
<TextInput class="w-full" type="file" @input="form.image = $event.target.files[0]" />
```

Específicamente:

- `@input`, un evento que se ejecuta cuando se selecciona un archivo desde el campo de tipo `file`.
- `$event.target.files[0]`, devuelve el último archivo seleccionado por el usuario, que en la selección simple de archivos mediante el campo de formulario de tipo `file`, viene siendo el único seleccionado por el usuario.

Si realizas un debug desde el controlador para saber que estamos recibiendo sobre el `$request['image']`, verás que, tenemos un archivo:

```
Illuminate\Http\UploadedFile {
    -test: false
    -originalName: "bird-g542c3699d_640.jpg"
    -mimeType: "image/jpeg"
    -error: 0
    #hashName: null
```

```

path: "/tmp"
filename: "php6RICEf"
basename: "php6RICEf"
pathname: "/tmp/php6RICEf"
extension: ""
realPath: "/tmp/php6RICEf"
***
}

```

Si lo colocaremos de la manera tradicional, con un **v-model** de Vue:

```
<TextInput class="w-full" type="file" v-model="form.image" />
```

Tendríamos:

```
^ "C:\fakepath\bird-g542c3699d_640.jpg"
```

Así que, en definitiva, tenemos que usar la opción del evento **@input** para la carga de archivos.

Carga de archivos manejada desde el formulario de crear o editar

Manejar la carga de archivos desde el formulario de creación o edición como un todo, viene siendo el proceso más común, por lo tanto, en el libro, veremos algunos lineamientos para que podamos administrar los posts, desde un solo formulario y petición; el primer cambio que haremos será mover solamente el campo para el formulario de editar/crear el post:

resources/js/Pages/Dashboard/Post/Save.vue

```

***  

<InputError :message="errors.category_id" class="mt-2" />  

</div>  

<div class="col-span-6">  

    <InputLabel value="Image" />  

    <TextInput type="file" @input="form.image = $event.target.files[0]" />  

    <InputError :message="errors.image" class="mt-2" />  

</div>  

***
```

Al colocar el campo de tipo archivo dentro del formulario en el componente de **Save.vue**, significa que, vamos a habilitar la carga de archivo en crear y actualizar post.

Los controladores, van a llamar al método de **upload()** definida anteriormente, importante notar que, el método de **upload()** no necesita modificación alguna:

app/Http/Controllers/Dashboard/PostController.php

```

public function create()
{
    $categories = Category::get();
    return inertia("Dashboard/Post/Save", compact('categories'));
}

public function store(Store $request)
{
    $post = Post::create($request->validated());
    if (request('image')) // img opcional
        $this->upload($request, $post);
    return to_route('post.index')->with('message', "Created post successfully");
}

public function edit(Post $post)
{
    $categories = Category::get();
    return inertia("Dashboard/Post/Save", compact('post', 'categories'));
}

public function update(Put $request, Post $post)
{
    $post->update($request->validated());
    if (request('image')) // img opcional
        $this->upload($request, $post);
    return redirect()->route('post.index')->with('message', "Updated post
successfully");
}

```

Para el envío de los datos al servidor, para la creación, no es necesario realizar ningún cambio, ya que, como se mencionó antes, el esquema por defecto usado para la carga de archivos viene siendo el POST:

```
router.post(route("post.store"), form);
```

Al ser la carga de la imagen un proceso opcional, verificamos si hay una imagen para iniciar el proceso de upload.

Para actualizar, al usar una petición de tipo PUT, no funciona la carga de archivos, así que, se debe de usar una petición de tipo POST, con la opción del método:

resources\js\Pages\Dashboard\Post\Save.vue

```

function submit() {
    if (form.id == "") router.post(route("post.store"), form);
    //else router.put(route("post.update", form.id), form);
    else
        router.post(route("post.update", form.id), {

```

```

        _method: "put",
        title: form.title,
        date: form.date,
        description: form.description,
        text: form.text,
        type: form.type,
        posted: form.posted,
        category_id: form.category_id,
        image: form.image,
    });
}

```

O podemos emplear el operador de desestructurar de JavaScript:

```

function submit() {
    if (props.post.id != '') {
        // router.put(route("post.update", props.post.id), form)
        router.post(route("post.update", props.post.id), {
            _method: 'put',
            ...form
        })
    } else {
        router.post(route("post.store"), form)
    }
}

```

Con esto, funcionará el proceso de carga de archivos tanto de manera independiente como combinado con el formulario con el cual se administra las publicaciones.

Consideraciones finales

Como puedes darte cuenta, la implementación mostrada anteriormente, está sujeta a adaptaciones, verás que, en el servidor, se están manejando el proceso del post y de la imagen en funciones diferentes, por lo tanto, las validaciones son independientes y por ende si:

1. **Existen errores en el formulario de crear**, no será procesada la imagen, esto puede ser útil cuando se quiere crear un post, ya que, si el post no existe, entonces no se puede asignar una imagen al post, pero cuando se está actualizando un post, puede que este no sea el comportamiento deseado.
2. **Existen solamente problemas en la carga de la imagen**, en este escenario, se crearía el post, o se actualizará, pero, la imagen no sería procesada, lo cual puede ser un problema en la fase de creación.

En definitiva, existen muchos caminos que puedes tomar para que la aplicación funcione como deseas, en este apartado solamente se explicó cómo puedes combinar la carga de archivos junto con el proceso del post, pero puedes adaptarlo a gusto para que tenga el comportamiento deseado.

Usar plugins para manejar la carga de archivos

Muchas veces, vas a querer instalar un plugin adicional para manejar la carga de archivos, en este caso, vamos a mostrar una posible implementación usando Oruga UI en un proyecto en Laravel con Inertia como mostramos en el libro básico:

<https://www.desarrollolibre.net/libros/primeros-pasos-laravel>

Recordemos que Oruga UI es una biblioteca liviana de componentes de interfaz de usuario para Vue.js sin dependencia de CSS.

Para instalar Oruga UI en un proyecto en Vue, se hace como un paquete de Node más:

```
$ npm install @oruga-ui/oruga-next
```

Y el tema, que es opcional pero recomendado para que los componentes de Oruga no aparezcan sin estilo:

```
$ npm install @oruga-ui/theme-oruga
```

También vamos a instalar la iconografía de Material Design Icons:

```
$ npm install @mdi/font
```

Configuramos a nivel de la aplicación:

resources/js/app.ts

```
/***
// ORUGA
import Oruga from '@oruga-ui/oruga-next'
import '@oruga-ui/theme-oruga/dist/oruga.css'

// Material Design
import '@mdi/font/css/materialdesignicons.min.css'

createInertiaApp({
    title: (title) => `${title} - ${appName}`,
    resolve: (name) => resolvePageComponent(`./Pages/${name}.vue`,
import.meta.glob('./Pages/**/*.{vue}`)),
    setup({ el, App, props, plugin }) {
        return createApp({ render: () => h(App, props) })
            .use(plugin)
            .use(Oruga)
            .use(ZiggyVue)
            .mount(el);
    },
    progress: {
```

```

        color: '#29d',
    },
});

});
```

Ya con esto, podemos usar cualquiera de los componentes de Vue en el proyecto de Inertia; no hay que realizar ningún proceso extra.

En el caso de la carga de archivos, tenemos el componente de upload:

```
<o-upload v-model="form.image" >
  <o-button tag="a" variant="primary" >
    <o-icon icon="upload"></o-icon>
    <span>Click to upload</span>
  </o-button>
</o-upload>
```

Y para usarlo:

resources/js/Pages/Dashboard/Post/Save.vue

```
***  

<div class="col-span-6">  

  <Label>Image Oruga</Label>  
  

  <o-upload v-model="form.image">  

    <o-button tag="upload-button" variant="primary">  

      <o-icon icon="upload"></o-icon>  

      <span>Click to Upload</span>  

    </o-button>  

  </o-upload>  
  

  <InputError :message="errors.image" class="mt-2" />  

</div>  
***
```

Con esto adaptamos la aplicación para que funcione con Oruga UI e Inertia; tendremos:

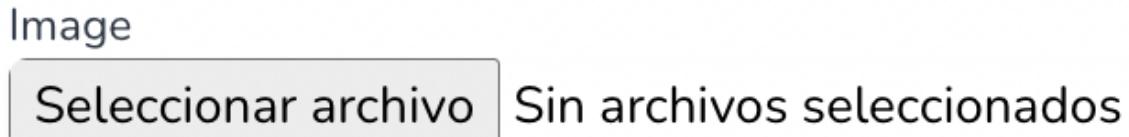


Figura 7-3: Upload con Oruga

Upload via Drag and Drop

En Oruga UI, tenemos un componente que permite realizar el upload de archivos vía Drag and Drop; para eso, puede ser de tipo múltiple:

```
<div class="col-span-6" v-if="post.id">
  <o-upload v-model="dropFiles" multiple drag-drop>
    <section class="ex-center">
      <p>
        <o-icon icon="upload" size="is-large"> </o-icon>
      </p>
      <p>Drop your files here or click to upload</p>
    </section>
  </o-upload>
</div>
<div>
  <Button class="mt-3" @click="upload">Send</Button>
</div>
***
```

O Individual:

```
<div class="col-span-6" v-if="post.id">
  <o-upload v-model="dropFiles" drag-drop>
    <section class="ex-center">
      <p>
        <o-icon icon="upload" size="is-large"> </o-icon>
      </p>
      <p>Drop your files here or click to upload</p>
    </section>
```

```

        </o-upload>
    </div>
    <div>
        <Button class="mt-3" @click="upload">Send</Button>
    </div>
***
```

En cualquier caso, debemos de especificar una propiedad, en este caso, llamada como **dropFiles**; y se declara una constante dentro de la función de **setup()** según los lineamientos del Composition Api en Vue; si es de tipo múltiple:

```

import { ref } from "vue";
***
setup(props) {
    const dropFiles = ref([]);
    ***
}
```

O si es de tipo individual:

```

import { ref } from "vue";
***
setup(props) {
    const dropFiles = ref("")
    ***
}
```

La función de **ref()** se usa para crear un objeto reactivo.

Creamos un watch u observable para determinar los cambios; si es de tipo múltiple, tendremos una respuesta de tipo **array**:

```

Array(1)
0: File {name: 'owl-g9ba965816_640.jpg', lastModified: 1655731428000, lastModifiedDate: Mon Jun 20 2024 09:23:48 GMT-0400 (hora de Venezuela), webkitRelativePath: '', size: 76140, ...}
```

O individual, sería un archivo:

```

File {name: 'owl-g9ba965816_640.jpg', lastModified: 1655731428000, lastModifiedDate: Mon Jun 20 2024 09:23:48 GMT-0400 (hora de Venezuela), webkitRelativePath: '', size: 76140}
```

Finalmente, enviamos la petición al servidor:

```

import { watch, ref } from "vue";
***
setup(props) {
```

```

const dropFiles: ref("") = ref("");
watch(() => dropFiles, (currentValue, oldValue) => {
    router.post(route("post.upload"), props.post.id), {
        "image": currentValue.value[currentValue.value.length - 1]
    });
}, { deep: true });
***
```

Usamos el watch, para observar cambios en la propiedad de **dropFiles** y cargar la imagen en el servidor; en este caso, al hacer uso de una carga múltiple, se usa la referencia de:

```
currentValue[currentValue.length - 1]
```

En vez de solamente:

```
currentValue
```

Aunque, a nivel funcional, se procesa siempre una única imagen; con el código anterior, al detectar cambios en la constante **dropFiles**, se dispara el **watch** y con esto, se hace una petición al servidor.

La opción de **deep** se usa para observar las mutaciones en arrays/objetos

Ver imagen cargada

Para poder visualizar la imagen cargada, podemos hacerlo como en un proyecto en Laravel, indicando la ruta; en el caso de este libro, la ruta usada es:

```
/image/post/
```

Por lo tanto, vamos a colocar la imagen cargada mediante la etiqueta IMG en el componente de **Save.vue**, como puede que exista la imagen en la publicación o puede que no, verificamos mediante una condición; finalmente, queda como:

```

<span>Drop your files here or click to upload</span>
</section>
</o-upload>

<div class="container mt-4" v-if="post.image">
    <div class="card">
        <div class="card-body">
            
            <button variant="danger" size="small" class="mt-2" @click="form.delete(route('post.image.delete', post.id))">
                Delete
            </button>
        </div>
    </div>
</div>
```

```

        </button>
        <a class="mt-2 ml-2 link-button-default" :href="'/image/post/' + post.image"
download>Download</a>
    </div>
</div>
</div>

```

Y tendremos:

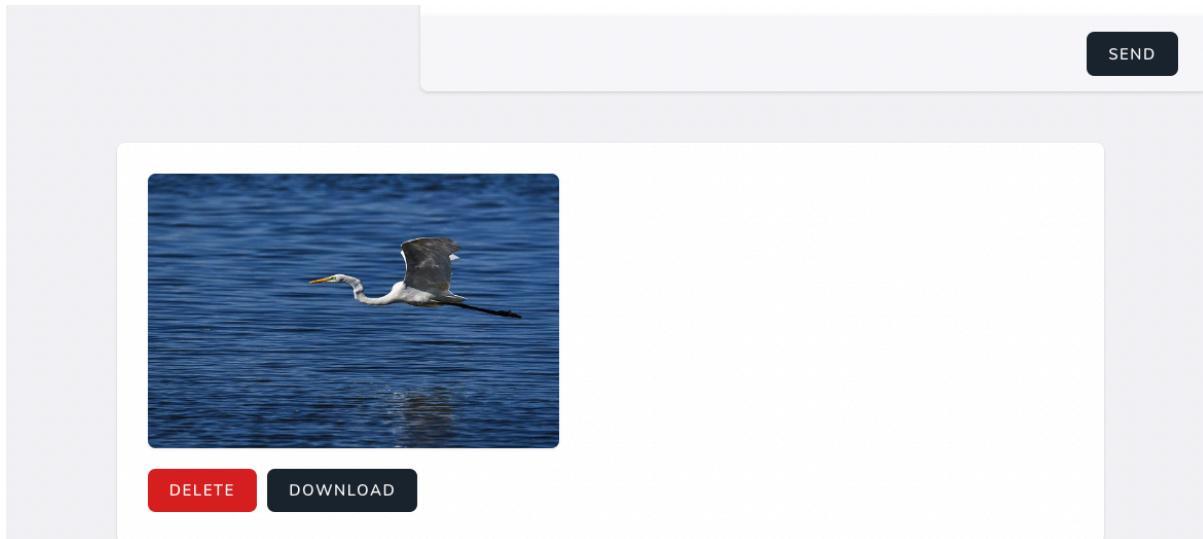


Figura 7-4: Imagen cargada

Eliminar una imagen

Ya que tenemos la imagen establecida y visible, podemos crear un sencillo panel de gestión para las mismas; comencemos indicando la opción de eliminación.

El método controlador, elimina la imagen de la carpeta y actualiza la publicación para remover la referencia a la imagen:

app/Http/Controllers/Dashboard/PostController.php

```

public function imageDelete(Post $post)
{
    Storage::disk("public_upload")->delete("image/post/" . $post->image);
    $post->update(['image' => '']);
    return to_route('post.edit', $post->id)->with('message', "image removed to post
successfully");
}

```

La ruta de tipo delete, ya que, vamos a eliminar:

routes/web.php

```
Route::delete('post/image/delete/{post}',  
[App\Http\Controllers\Dashboard\PostController::class,  
'imageDelete'])->name('post.image-delete');
```

En el componente de **Save.vue**, colocamos la petición de tipo delete; podemos usar el formulario, que es una instancia de **useForm()** para poder enviar peticiones, en este caso, una de tipo delete para eliminar la imagen:

resources/js/Pages/Dashboard/Post/Save.vue

```
  
<button variant="danger" size="small" class="mt-2"  
@click="form.delete(route('post.image.delete', post.id))">  
    Delete  
</button>
```

Con esto, tendremos la función de eliminación de la imagen desde la visualización de la imagen cargada.

Descargar una imagen

Para descargar la imagen, basta con usar un enlace con el atributo **download** de HTML y en el enlace, colocamos la ruta al archivo que queremos descargar, que, en este caso, sería la imagen:

resources/js/Pages/Dashboard/Post/Save.vue

```
<button variant="danger" size="small" class="mt-2"  
@click="form.delete(route('post.image.delete', post.id))">  
    Delete  
</button>  
<a class="mt-2 ml-2 link-button-default" :href="/image/post/'+post.image  
download>Download</a>  
</div>
```

Puedes consultar el código fuente en:

https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.4_12

Capítulo 8: CKEditor

Extender una aplicación con extensiones y plugins existentes, es una tarea cotidiana, así que, como se mostró en el anterior capítulo, que instalamos y configuramos Oruga UI en el proyecto, vamos a usar otro plugin en el proyecto de Laravel Inertia; en este caso, el de CKEditor para el contenido enriquecido de los posts.

CKEditor no es más que un plugin para el contenido enriquecido, si queremos agregar textos, imágenes, listas, tablas, enlaces, entre otros aspectos, este tipo de plugins son estupendos para tal fin.

Vamos a usar la integración nativa de CKEditor para Vue 3, la cual requiere, instalar dos paquetes, el CKEditor de Node, y la extensión para Vue:

```
$ npm install ckeditor5 @ckeditor/ckeditor5-vue
```

En la documentación oficial:

<https://ckeditor.com/docs/ckeditor5/latest/getting-started/installation/vuejs-v3.html>

Puedes ver que también utilizan otro:

```
$ npm install ckeditor5-premium-features
```

Pero es de pago:

<https://ckeditor.com/docs/trial/latest/guides/overview.html>

Ahora, se configura a nivel del proyecto en Vue, al igual que el resto de los plugins:

resources/js/app.ts

```
import { CkeditorPlugin } from '@ckeditor/ckeditor5-vue';
***  
createInertiaApp({  
    title: (title) => `${title} - ${appName}`,  
    resolve: (name) => resolvePageComponent(`./pages/${name}.vue`,  
import.meta.glob<DefineComponent>('./pages/**/*.{vue}')'),  
    setup({ el, App, props, plugin }) {  
        createApp({ render: () => h(App, props) })  
            .use(plugin)  
            .use(Oruga)  
            .use(CkeditorPlugin)  
            .use(ZiggyVue)  
            .mount(el);  
    },  
***
```

Y finalmente, ya estamos listos para usarlo; para ello, vamos a abrir el archivo de:

resources\js\pages\dashboard\post\Save.vue

```
<div class="col-span-6">  
    <!-- <textarea  
        class="rounded-md w-full border-gray-300"  
        v-model="form.text"></textarea> -->  
  
    <ckeditor v-model="form.text" :editor="editor" :config="editorConfig" />  
</div>  
***  
import { ClassicEditor, Bold, Essentials, Italic, Mention, Paragraph, Undo, Heading } from  
'ckeditor5';
```

```

import 'ckeditor5/ckeditor5.css';
export default {
  components: {
    /**
     * ClassicEditor
    */,
    data() {
      return {
        editor: ClassicEditor,
        editorConfig: {
          licenseKey: 'GPL',
          plugins: [Bold, Essentials, Italic, Mention, Paragraph, Undo, Heading ,],
          toolbar: ['undo', 'redo', '|', 'bold', 'italic', 'heading',],
        }
      };
    },
  }
}

```

O

```

setup(props) {
  /**
   * ClassicEditor
  */

  const editor = ClassicEditor
  const editorConfig = {
    licenseKey: 'GPL',
    plugins: [Bold, Essentials, Italic, Mention, Paragraph, Undo, Heading ,],
    toolbar: ['undo', 'redo', '|', 'bold', 'italic', 'heading',],
  }

  return {
    submit, upload, form, breadcrumbs, dropFiles, editor, editorConfig
  };
}

```

Se importa recursos como el CSS y la funcionalidad del plugin; puedes colocar más opciones en el toolbar cómo puedes consultar en la documentación oficial en el apartado de código fuente:

<https://ckeditor.com/docs/ckeditor5/latest/examples/builds/classic-editor.html>

En la cual, es una pareja entre el plugin y toolbar; por ejemplo, el de heading:

```

plugins: [*** Heading ],
toolbar: [*** 'heading'],

```

Luego, debemos de crear las configuraciones; CKEditor es un plugin con una enorme cantidad de personalizaciones; en el libro, usamos la personalización mínima.

<https://ckeditor.com/docs/ckeditor5/latest/installation/getting-started/frameworks/vuejs-v3.html>

Y tenemos:

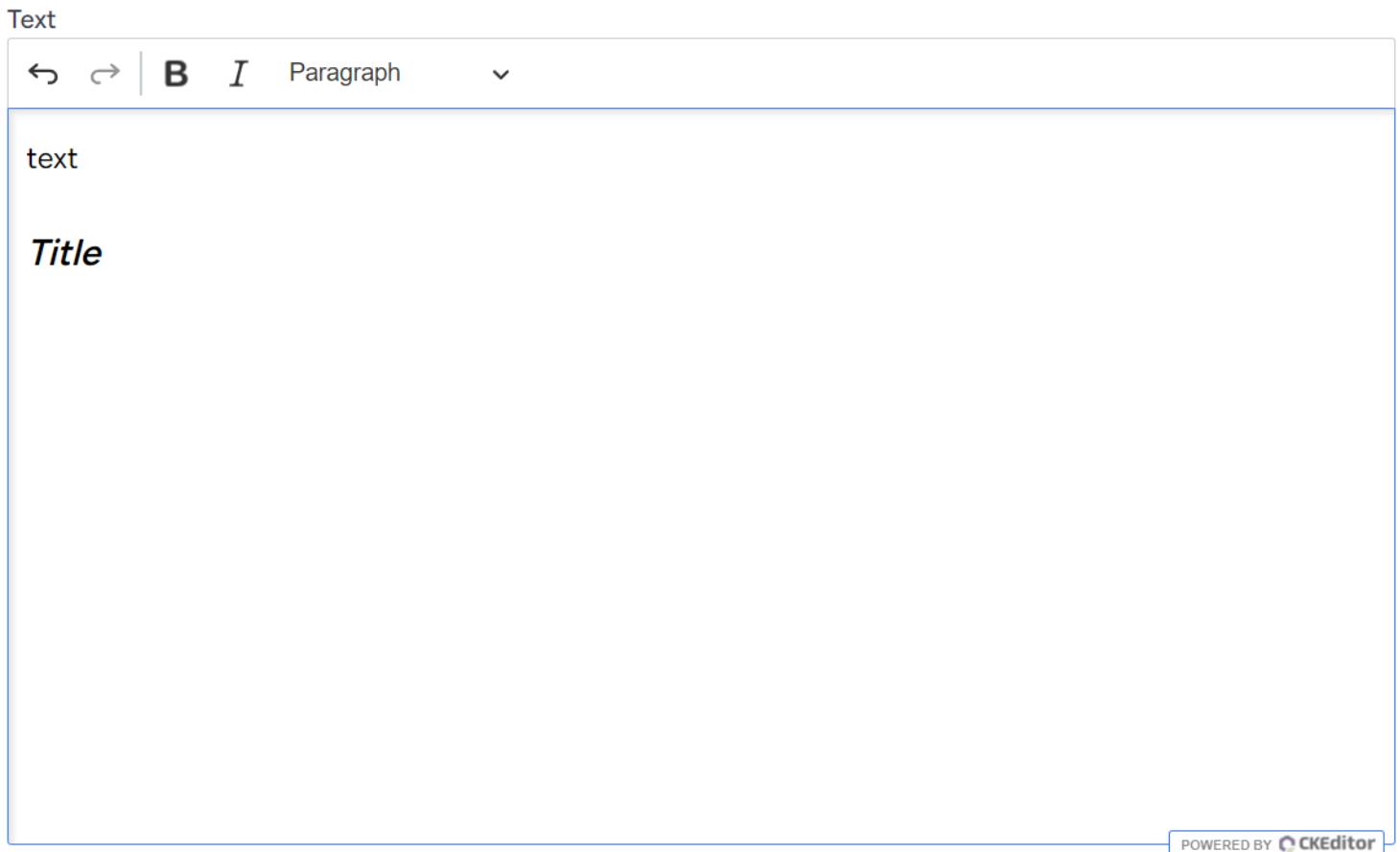


Figura 8-1: CKeditor

Importante notar:

- El uso del v-model empleado originalmente para manejar el texto o contenido.
- El uso de las configuraciones.

Al estar usando Tailwind en la aplicación, perdemos es estilo aplicado por defecto, como los tamaños de los títulos, listados, etc, por lo tanto, para recuperar este estilo, definimos un CSS como el siguiente:

resources/css/app.css

```
/* CKEDITOR */  
.ck-editor__main{  
    color: #000;  
}  
.ck-editor__editable_inline {
```

```
    min-height: 400px;
}

.ck-editor__main h1 {
    font-size: 40px;
}

.ck-editor__main h2 {
    font-size: 30px;
}

.ck-editor__main h3 {
    font-size: 25px;
}

.ck-editor__main h4 {
    font-size: 20px;
}

.ck-editor__main ul {
    list-style-type: circle;
    margin: 10px;
    padding: 10px;
}

.ck-editor__main ol {
    list-style-type: decimal;
    margin: 10px;
    padding: 10px;
}
/* CKEDITOR */
```

El cual puedes personalizar a gusto.

Puedes consultar el código fuente en:

https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.5_12

Capítulo 9: Diálogos de confirmación y mensajes toast

Inertia ofrece varios componentes interesantes, pero, como podrás darte cuenta, se echan de menos algunos otros como para manejar diálogos de confirmación y mensajes toast; sin embargo, el hecho que no exista es algo bueno, ya que, Inertia al usar Vue, ya contamos con una gran cantidad de plugins listos para usar como hemos demostrado en los anteriores capítulos y con esto, podemos personalizar nuestra aplicación y prescindir de lo más posible de integraciones que pudieran venir por defecto en Inertia.js

Para los diálogos y toasts, puedes usar cualquier plugin, pero, al tener instalado Oruga UI; vamos a aprovechar los componentes que ya existen en la misma y con esto, evitar instalar paquetes adicionales.

Configurar el diálogo de confirmación

Como vimos en el libro básico de Laravel, en Oruga UI, contamos con un componente de modal:

<https://oruga.io/components/modal.html>

Que luce como:



Figura 9-1: Diálogo en Oruga

Que en definitiva no es más que un contenedor vacío en el cual, podemos colocar textos, imágenes, formularios y, en definitiva, cualquier contenido HTML.

Diálogo de confirmación para eliminar

Comencemos adaptando el diálogo de confirmación para la eliminación de los posts; para ello, vamos a colocar el texto de confirmación y los botones de acción; uno para eliminar y otro para cancelar:

resources\js\pages\dashboard\category\Index.vue

```
<template>
  <o-modal v-model:active="confirmDeleteActive">
    <p class="p-4 text-black">Are you sure you want to delete the selected record?</p>

    <div class="flex flex-row-reverse gap-2 bg-gray-100 p-3">
      <o-button>Delete</o-button>
      <o-button>Cancel</o-button>
    </div>
  </o-modal>
***
```

Recuerda que todos estos cambios se aplican sobre:

resources\js\pages\dashboard\category\Index.vue

Para mostrar el modal, usamos una propiedad booleana y para manejar el registro que queremos eliminar, usamos otra propiedad que contiene el id del registro que queremos eliminar; así que, creamos las propiedades respectivas:

```
data() {
  return {
    confirmDeleteActive: false,
    deleteCategoryRow: "",
  };
},
```

Desde el apartado de acciones en el listado, ahora, la operación para eliminar, lo único que va a realizar es, registrar el ID del post que se quiere eliminar:

Creamos una función que elimine el post:

```
import { Link, router } from "@inertiajs/vue3"
***
methods: {
  deleteCategory() {
    router.delete(route("category.destroy", this.deleteCategoryRow));
  }
},
```

```
        this.confirmDeleteActive = false;  
    },  
},
```

Y finalmente, volviendo a los botones de acción, el de cancelar, lo único que hace es ocultar el modal con la propiedad que se encarga de indicar si el modal es visible o no:

```
<o-button @click="confirmDeleteActive = false">Cancel</o-button>
```

Y el de conformación, llama a la función definida anteriormente:

```
<o-button variant="danger" @click="deleteCategory">Delete</o-button>
```

Y desde el listado:

```
<!-- <Link as="button" type="button" method="DELETE" class="text-sm text-red-400  
hover:text-red-700 ml-2" :href="route('category.destroy', c.id)">Delete</Link> -->  
  
<o-button iconLeft="delete" rounded size="small" variant="danger" @click="=  
confirmDeleteActive = true; deleteCategoryRow = c.id;">Delete</o-button>
```

Con esto, al presionar sobre eliminar en el listado, tendremos:

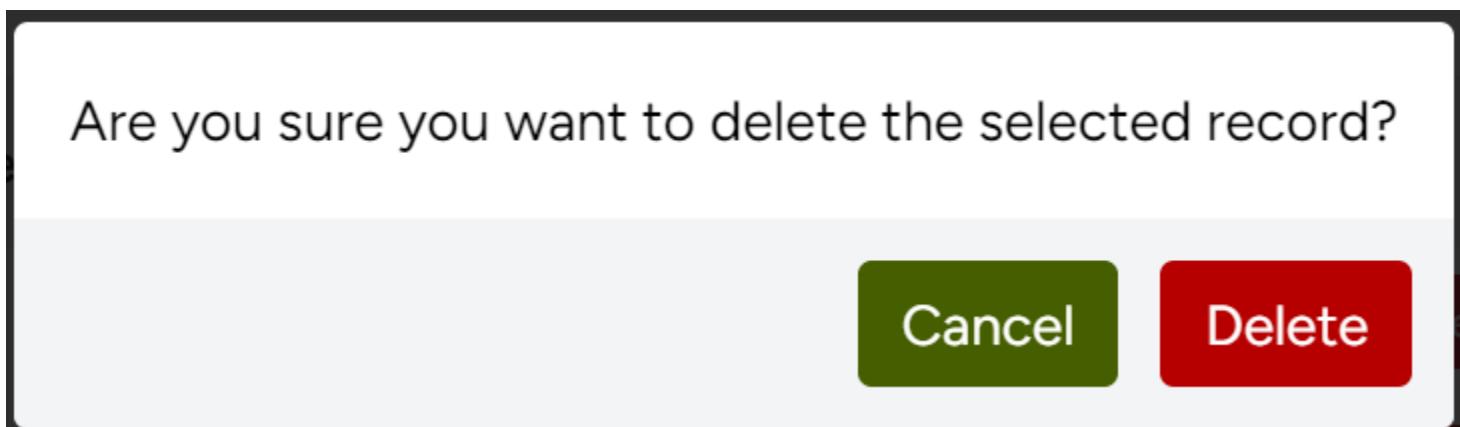


Figura 9-2: Diálogo de eliminar de Oruga

Replicar para los Post

La implementación realizada anteriormente, se debe hacer para el listado de los posts; el componente para los Post, quedaría:

resources/js/Pages/Dashboard/Post/Index.vue

```
<template>  
  <o-modal v-model:active="confirmDeleteActive">
```

```

<p class="p-4 text-black">Are you sure you want to delete the record?</p>

<div class="flex flex-row-reverse gap-2 bg-gray-100 p-3">
  <o-button variant="danger" @click="deletePost">Delete</o-button>
  <o-button @click="confirmDeleteActive = false">Cancel</o-button>
</div>
</o-modal>

<app-layout>
  <div class="container">
    <div class="card">
      <div class="card-body">
        <Link class="link-button-default my-3" :href="route('post.create')">Create</Link>

        <table class="w-full border">
          <thead class="dark:bg-gray-800 bg-gray-100">
            <tr class="border-b">
              <th class="p-3">Id</th>
              <th class="p-3">Title</th>
              <th class="p-3">Slug</th>
              <th class="p-3">Actions</th>
            </tr>
          </thead>
          <tbody>
            <tr class="border-b" v-for="p in posts.data" :key="p.id">
              <td class="p-2">{{ p.id }}</td>
              <td class="p-2">{{ p.title }}</td>
              <td class="p-2">{{ p.slug }}</td>
              <td class="p-2">
                <Link class="text-sm text-purple-400 hover:text-purple-700"
:href="route('post.edit', p.id)">Edit</Link>
                <!-- <Link as="button" type="button" method="DELETE" class="text-sm
text-red-400 hover:text-red-700 ml-2"
                  :href="route('post.destroy', p.id)">Delete</Link> -->
                <o-button iconLeft="delete" rounded size="small" variant="danger"
@click="confirmDeleteActive = true;
        deletePostRow = p.id;">Delete</o-button>
              </td>
            </tr>
          </tbody>
        </table>
        <pagination class="my-4" :links="posts" />
      </div>
    </div>
  </div>
</app-layout>

```

```

</template>

<script>
import { Link, router } from "@inertiajs/vue3";
import AppLayout from "@/Layouts/AppLayout.vue";
import Pagination from "@/Shared/Pagination.vue";

export default {
  data() {
    return {
      confirmDeleteActive: false,
      deletePostRow: "",
    };
  },
  methods: {
    deletePost() {
      router.delete(route("post.destroy", this.deletePostRow));
      this.confirmDeleteActive = false;
    },
  },
  components: {
    AppLayout,
    Link,
    Pagination,
  },
  props: {
    posts: Object,
  },
};
</script>

```

Modal de Inertia

En inertia, tenemos unos modals a nuestra disposición:

1. resources\js\Components\Modal.vue
2. resources\js\Components\DialogModal.vue
3. resources\js\Components\ConfirmationModal.vue

El Modal.vue, permite usar el componente de Modal de una manera más flexible, al implementar lo básico:



Figura 9-3: Modal de Laravel Inertia

En esencia, una caja donde colocar el contenido y el oscurecimiento del resto de la página.

Para usarlo:

```
<script setup lang="ts">
import { useForm } from '@inertiajs/vue3';
import { ref } from 'vue';

// Components
import HeadingSmall from '@/components/HeadingSmall.vue';
import InputError from '@/components/InputError.vue';
import { Button } from '@/components/ui/button';
import {
    Dialog,
    DialogClose,
   DialogContent,
    DialogDescription,
    DialogFooter,
    DialogHeader,
    DialogTitle,
    DialogTrigger,
} from '@/components/ui/dialog';
import { Input } from '@/components/ui/input';
import { Label } from '@/components/ui/label';

const passwordInput = ref<HTMLInputElement | null>(null);

const form = useForm({
    password: '',
});

const deleteUser = (e: Event) => {
    e.preventDefault();

    form.delete(route('profile.destroy'), {
        preserveScroll: true,
        onSuccess: () => closeModal(),
        onError: () => passwordInput.value?.focus(),
        onFinish: () => form.reset(),
    });
};

const closeModal = () => {
    form.clearErrors();
}
```

```

        form.reset();
    };
</script>

<template>
    <div class="space-y-6">
        <HeadingSmall title="Delete account" description="Delete your account and all of its resources" />
        <div class="space-y-4 rounded-lg border border-red-100 bg-red-50 p-4 dark:border-red-200/10 dark:bg-red-700/10">
            <div class="relative space-y-0.5 text-red-600 dark:text-red-100">
                <p class="font-medium">Warning</p>
                <p class="text-sm">Please proceed with caution, this cannot be undone.</p>
            </div>
            <Dialog>
                <DialogTrigger as-child>
                    <Button variant="destructive">Delete account</Button>
                </DialogTrigger>
                <DialogContent>
                    <form class="space-y-6" @submit="deleteUser">
                        <DialogHeader class="space-y-3">
                            <DialogTitle>Are you sure you want to delete your account?</DialogTitle>
                            <DialogDescription>
                                Once your account is deleted, all of its resources and data will also be permanently deleted. Please enter your password to confirm you would like to permanently delete your account.
                            </DialogDescription>
                        </DialogHeader>

                        <div class="grid gap-2">
                            <Label for="password" class="sr-only">Password</Label>
                            <Input id="password" type="password" name="password" ref="passwordInput" v-model="form.password" placeholder="Password" />
                            <InputError :message="form.errors.password" />
                        </div>

                        <DialogFooter class="gap-2">
                            <DialogClose as-child>
                                <Button variant="secondary" @click="closeModal"> Cancel </Button>
                            </DialogClose>

                            <Button variant="destructive" :disabled="form.processing">
                                <button type="submit">Delete account</button>
                            </Button>
                        </DialogFooter>
                    </form>
                </DialogContent>
            </Dialog>
        </div>
    </div>
</template>

```

```

                </Button>
            </DialogFooter>
        </form>
    <DialogContent>
        </Dialog>
    </div>
</div>
</template>

<modal :show="true">
    Hello Inertia
</modal>
```

Tenemos 3 props principales para usar:

```
const props = defineProps({
    show: {
        type: Boolean,
        default: false,
    },
    maxWidth: {
        type: String,
        default: '2xl',
    },
    closeable: {
        type: Boolean,
        default: true,
    },
});
```

1. El **show** indica si es visible o no mediante un boolean.
2. el tamaño máximo permitido, que usa clases de Tailwind:
 - a. 'sm': 'sm:max-w-sm',
 - b. 'md': 'sm:max-w-md',
 - c. 'lg': 'sm:max-w-lg',
 - d. 'xl': 'sm:max-w-xl',
 - e. '**2xl**
3. Un evento que podrías usar al momento de cerrar el modal: **closeable**.

El modal de confirmación **ConfirmationModal**, tal cual indica su nombre, cuenta con una estructura más elaborada, si nos fijamos, utiliza el componente de Modal.vue anterior más algunos agregados para usar tres secciones:

1. El título, **title**.
2. Cuerpo, **content**.
3. Acciones o **footer**.

Por lo demás, su funcionamiento es el mismo que el del **Modal.vue**.

Así que, para usar este modal en vez del de Oruga, podemos definirlo como:

resources/js/Pages/Dashboard/Post/Index.vue

```
<confirmation-modal :show="confirmDeleteActive">
  <template v-slot:title> Confirmation </template>

  <template v-slot:content>
    <p class="p-4">Are you sure you want to delete the record?</p>
  </template>

  <template v-slot:footer>
    <o-button variant="danger" @click="deletePost">Delete</o-button>
    <div class="mr-3"></div>
    <o-button @click="confirmDeleteActive = false">Cancel</o-button>
  </template>
</confirmation-modal>
```

Recuerda importar:

```
import ConfirmationModal from "@/Components/ConfirmationModal.vue";

export default {
  data() {
    return {
      confirmDeleteActive: false,
      deletePostRow: "",
    };
  },
  components: {
    ConfirmationModal,
  },
}/***
```

Y tendremos:

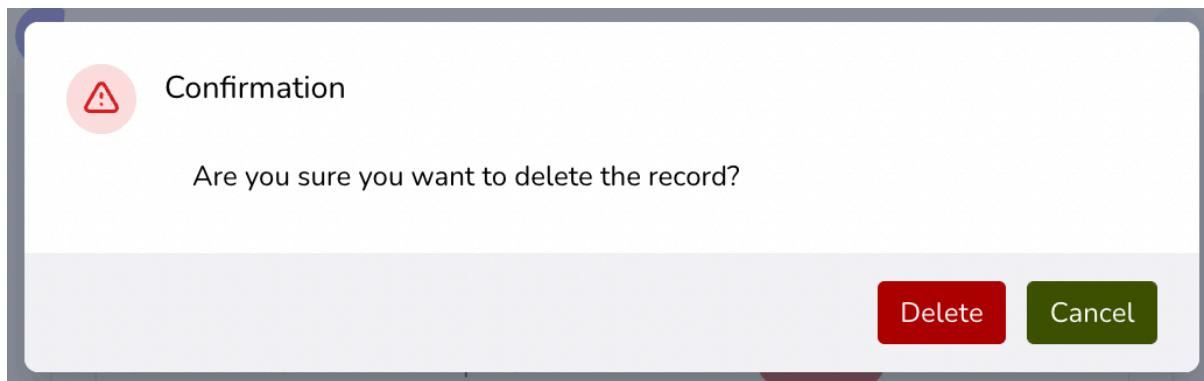


Figura 9-4: Diálogo de confirmación

Puedes consultar el código fuente en:

https://github.com/libredesarrollo/curso-libro-laravel-11-inertia-store/releases/tag/v0.6_12

Capítulo 10: Comunicación en componentes, Formulario paso por paso

Hasta ahora, hemos usado los componentes en Vue como un elemento más para administrar recursos, como si se tratase de una vista en blade, pero con Vue; pero, los componentes tienen la particularidad de poder ser fácilmente incrustables dentro de otros componentes; con esto, ganamos un gran nivel de reusabilidad. Esto es algo que seguramente no es una sorpresa ya que, hemos reutilizado muchos componentes de Laravel Inertia para conseguir la aplicación tipo CRUD anterior.

En este capítulo, vamos a trabajar con la reusabilidad de los componentes y con esto, poder colocar componentes dentro de otros componentes y comunicar los mismos; para ello, vamos a usar un formulario, un formulario por pasos en los cuales vamos a tener una página central (componente padre) y cada uno de estos pasos, van a ser un componente independiente (componente hijo) con su respectivo formulario.

Lo explicado anteriormente, viene siendo la idea básica, pero vamos a explotar este modelo para poder comunicar componentes, poder actualizar estados del componente padre según el estado del componente hijo, así como poder embeber más de dos niveles de componentes:

Componente Padre

Componente hijo 1

Componente hijo 2

⋮

Componente hijo N

Figura 10-1: Componentes para el formulario paso por paso

Y tener una idea clara de cómo podemos aprovechar al máximo el uso de los componentes de Vue.

Crear migraciones

Como mencionamos antes, vamos a usar un formulario paso por paso, en las cuales, tendremos un componente padre que contiene a los formularios que son sus componentes hijos; a su vez, el componente padre, también será un formulario (formulario padre):

```
// *** PADRE
<form id="ComponentePadre"/>
<div>
  <form id="hijo1"/> - HIJO 1
  <form id="hijo2"/> - HIJO 2
  <form id="hijo3"/> - HIJO 3
</div>
```

Teniendo esto en mente, vamos a crear el formulario padre y sus hijos:

```
$ php artisan make:migration create_contact_generals_table
$ php artisan make:migration create_contact_persons_table
$ php artisan make:migration create_contact_companies_table
$ php artisan make:migration create_contact_optionals_table
```

Explicación del código anterior

- El componente principal o padre, va a ser el de "general".
- Las migraciones para las personas y las compañías o empresas, van a ser cada uno de ellos solo paso; por lo tanto, según el tipo de requerimiento que se determinará desde el formulario general, usaremos uno o el otro.
- El resto de los componentes serán componentes hijos, una migración para cada paso, por lo tanto, tendremos 3 pasos.

Migraciones

Para la migración principal (componente padre):

```
return new class extends Migration
{
    public function up(): void
    {
        Schema::create('contact_generals', function (Blueprint $table) {
            $table->id();
            $table->string('subject', 255);
            $table->text('message');
            $table->enum('type', ['company', 'person']);
        });
    }
}
```

```

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

public function down(): void
{
    Schema::dropIfExists('contact_generals');
}
};
```

Explicación del código anterior

Para este formulario principal, nos interesa registrar el asunto y contenido del mensaje; también nos interesa saber que ente está escribiendo, si es una empresa o particular (persona) y según la selección del usuario, usaremos uno o el otro; para el formulario de las empresas:

The form consists of five input fields and two buttons. The fields are labeled 'Name', 'Identification', 'Email', and 'Extra'. Below these is a section labeled 'Choices' containing a dropdown menu with the placeholder 'Seleccione' and a dark blue 'ENVIAR' button.

| | |
|----------------|--|
| Name | <input type="text"/> |
| Identification | <input type="text"/> |
| Email | <input type="text"/> |
| Extra | <input type="text"/> |
| Choices | <input style="border: 1px solid #ccc; padding: 5px; width: 150px; height: 30px;" type="button" value="Seleccione"/> ▼ |
| | <input style="background-color: #002B36; color: white; border: none; padding: 10px; width: 150px; height: 30px;" type="button" value="ENVIAR"/> |

Figura 10-2: Formulario paso 2, empresa

O personas:

The form consists of four fields:

- Name**: A text input field.
- Surname**: A text input field.
- Choices**: A dropdown menu with the placeholder "Seleccione" and a dropdown arrow.
- Other**: A text area for additional information.

At the bottom right is a large, dark, rounded rectangular button with the word "ENVIAR" in white capital letters.

Figura 10-3: Formulario paso 2, persona

La migración para las empresas solicitará información base sobre la empresa como su nombre, identificador, email y un campo extra para cualquier otra información pertinente; aparte de la relación foránea para poder vincular los futuros formularios:

```
return new class extends Migration
{
    public function up(): void
    {
        Schema::create('contact_companies', function (Blueprint $table) {
            $table->id();
            $table->string('name', 255);
            $table->string('identification', 50);
            $table->string('email', 80);
            $table->string('extra', 255);
            $table->foreignId('contact_general_id')->onDelete('cascade');
            $table->enum('choices', ['advertis', 'post', 'course', 'movie', 'other']);
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('contact_companies');
    }
};
```

Similar al anterior, pero solicitamos información de una persona en su lugar:

```

return new class extends Migration
{
    public function up(): void
    {
        Schema::create('contact_persons', function (Blueprint $table) {
            $table->id();
            $table->string('name', 255);
            $table->string('surname', 80);
            $table->foreignId('contact_general_id')->onDelete('cascade');
            $table->enum('choices', ['advert', 'post','course','movie','other']);
            $table->string('other', 255)->nullable();
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('contact_persons');
    }
};

```

Esta migración corresponderá al último paso del formulario, en la cual agregaremos más información variada:

```

return new class extends Migration
{

    public function up(): void
    {
Schema::create('contact_details', function (Blueprint $table) {
            $table->id();
            $table->string('extra', 500);
            $table->foreignId('contact_general_id')->onDelete('cascade');
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('contact_details');
    }
};

```

Importante notar que, puedes personalizar los campos de estos formularios para registrar datos de interés; en este capítulo se intenta trabajar con migraciones con pocos campos para evitar perder el enfoque, el cual es, reutilizar fácilmente componentes y la comunicación entre los mismos.

Finalmente, ejecutamos las migraciones con:

```
$ php artisan migrate
```

Modelos

Y para los modelos:

```
$ php artisan make:model ContactGeneral
$ php artisan make:model ContactPerson
$ php artisan make:model ContactCompany
$ php artisan make:model ContactDetail
```

Que tendrán la siguiente definición:

```
class ContactGeneral extends Model
{
    use HasFactory;

    protected $fillable = ['subject', 'message', 'type'];

    public function person(){
        return $this->hasOne(ContactPerson::class);
    }
    public function company(){
        return $this->hasOne(ContactCompany::class);
    }
    public function detail(){
        return $this->hasOne(ContactDetail::class);
    }
}
```

Para el de personas y su relación foránea:

```
class ContactPerson extends Model
{
    use HasFactory;

    public $timestamps = false;
    protected $table = "contact_persons";

    protected $fillable = ['name', 'surname', 'contact_general_id', 'choices', 'other'];

    public function general()
    {
        return $this->belongsTo(ContactGeneral::class, 'contact_general_id');
```

```
| }  
| }
```

Para el de la empresa y su relación foránea:

```
class ContactCompany extends Model  
{  
    use HasFactory;  
  
    public $timestamps = false;  
  
    protected $fillable = ['name', 'identification', 'email', 'extra',  
'contact_general_id', 'choices'];  
  
    public function general(){  
        return $this->belongsTo(ContactGeneral::class, 'contact_general_id');  
    }  
}
```

Para el último paso y su relación foránea:

```
class ContactDetail extends Model  
{  
    use HasFactory;  
  
    public $timestamps = false;  
  
    protected $fillable = ['extra', 'contact_general_id'];  
  
    public function general(){  
        return $this->belongsTo(ContactGeneral::class, 'contact_general_id');  
    }  
}
```

Formulario, Paso 1

En este apartado, vamos a crear el proceso para crear y editar contactos generales que corresponden al primer paso del formulario paso por paso; vamos a crear el controlador de tipo recurso:

```
$ php artisan make:controller Contact/GeneralController -r
```

Con su tipo recurso, solamente vamos a usar los que manejan un formulario, es decir, para crear y editar:

app/Http/Controllers/Contact/GeneralController.php

```
| <?php
```

```

namespace App\Http\Controllers>Contact;

use App\Http\Controllers\Controller;
use App\Http\Requests>Contact\GeneralRequest ;
use App\Models>ContactGeneral;
use Illuminate\Http\Request;

class GeneralController extends Controller
{

    public function create()
    {
        return inertia("Contact/General/Form");
    }

    public function store(GeneralRequest $request)
    {
        ContactGeneral::create($request->validated());
    }

    public function edit(ContactGeneral $contactGeneral)
    {
        return inertia("Contact/General/Form", compact('contactGeneral'));
    }

    public function update(GeneralRequest $request, ContactGeneral $contactGeneral)
    {
        $contactGeneral->update($request->validated());
    }
}

```

Creamos el request:

```
$ php artisan make:request Contact/GeneralRequest
```

Con las validaciones típicas:

app/Http/Requests/Contact/General.php

```

<?php

namespace App\Http\Requests>Contact;

use Illuminate\Foundation\Http\FormRequest;

```

```

class GeneralRequest extends FormRequest
{
    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return [
            'subject' => 'required|min:2|max:255',
            'type' => 'required',
            'message' => 'required|min:2',
        ];
    }
}

```

Creamos la ruta:

```

use App\Http\Controllers\Contact\GeneralController;
*** 
Route::group([
    'prefix' => 'contact',
    'middleware' => 'auth:sanctum',
    config('jetstream.auth_session'),
    'verified'
], function () {
    Route::resource('contact-general', GeneralController::class)->only(['create', 'edit',
'store', 'update']);
});

```

Y para la vista de formulario, usaremos los siguientes campos:

```

field: {
    subject: "",
    message: "",
    type: "",
},

```

Por lo demás, usaremos la misma estructura y lógica que la empleada para el componente de:

resources/js/Pages/Dashboard/Post/Save.vue

Usaremos:

1. subject, campo de texto.
2. message, textarea, puede usar también el CKEditor.

3. type, un campo de tipo selección, el mismo usado para administrar los posts, pero, en este caso sustituyendo los OPTIONS con:
 - a. Company
 - b. Person

A la final, quedaría:

resources/js/Pages/Contact/General/Form.vue

```
<template>
  <AppLayout title="Create Category">
    <template #header>
      <h2 class="font-semibold text-xl text-gray-800 leading-tight">
        Create Contact
      </h2>
    </template>

    <div class="max-w-7xl mx-auto py-10 sm:px-6 lg:px-8">
      <FormSection @submitted="submit">
        <template #title>
          Create Contact
        </template>

        <template #description>
          Create Contact
        </template>

        <template #form>
          <div class="col-span-6">
            <InputLabel for="subject" value="Subject" />
            <TextInput id="subject" v-model="form.subject" type="text"
class="block w-full mt-1"
              autofocus />
            <InputError :message="errors.subject" class="mt-2" />
          </div>
          <div class="col-span-6">
            <InputLabel for="message" value="Message" />
            <textarea v-model="form.message"
              class="block w-full mt-1 border-gray-300
rounded-md"></textarea>
            <InputError :message="errors.message" class="mt-2" />
          </div>
          <div class="col-span-6">
            <InputLabel for="type" value="Type" />
          </div>
        </template>
      </FormSection>
    </div>
  </AppLayout>
</template>
```

```

        <select v-model="form.type" class="block w-full mt-1
border-gray-300 rounded-md">
            <option value="company">Company</option>
            <option value="person">Person</option>
        </select>

        <InputError :message="errors.type" class="mt-2" />
    </div>
</template>

<template #actions>
    <PrimaryButton :class="{ 'opacity-25': form.processing }"
:disabled="form.processing">
        Save
    </PrimaryButton>
</template>
</FormSection>
</div>
</AppLayout>
</template>

<script>

import { router, useForm } from "@inertiajs/vue3"

import AppLayout from "@/Layouts/AppLayout.vue";

import FormSection from '@/Components/FormSection.vue';
import InputError from '@/Components/InputError.vue';
importInputLabel from '@/Components/InputLabel.vue';
import PrimaryButton from '@/Components/PrimaryButton.vue';
import TextInput from '@/Components/TextInput.vue';

export default {
    props: {
        errors: Object,
        contactGeneral: {
            default: {
                id: '',
                subject: '',
                type: '',
                message: ''
            }
        }
    },
    components: {

```

```
AppLayout,
FormSection,
InputError,
InputLabel,
PrimaryButton,
TextInput
},
setup(props) {
    const form = useForm({
        id: props.contactGeneral.id,
        subject: props.contactGeneral.subject,
        type: props.contactGeneral.type,
        message: props.contactGeneral.message,
    })

    function submit() {
        if(form.id == '') {
            router.post(route("contact-general.store"), form)
        } else{
            router.put(route("contact-general.update", form.id), form)
        }
    }

    return { form, submit }
}
</script>
```

Y tendremos:

The screenshot shows a 'Create Contact' form with three required fields: Subject, Message, and Type. Each field has a red error message below it indicating it is required. A large 'SEND' button is at the bottom right.

| Field | Error Message |
|---------|--------------------------------|
| Subject | The subject field is required. |
| Message | The message field is required. |
| Type | The type field is required. |

Figura 10-4: Formulario paso 1, para contacto general

Ya con el proceso anterior, es posible crear y actualizar contactos de tipo general.

Formulario, Paso 2: Empresa

Ahora, vamos a crear el paso dos tal cual se realizó para el paso 1; seguiremos la misma estructura de crear el controlador, validaciones y vista.

Creamos el controlador:

```
$ php artisan make:controller Contact/CompanyController -r
```

El cual tendrá la misma estructura que el controlador de **GeneralController**, pero adaptada a los campos de dicha entidad:

```
<?php

namespace App\Http\Controllers\Contact;

use App\Http\Controllers\Controller;
use App\Http\Requests>Contact\CompanyRequest;
use App\Models>ContactCompany;
use Illuminate\Http\Request;

class CompanyController extends Controller
```

```
{
    public function create()
    {
        return inertia("Contact/Company/Form");
    }

    public function store(CompanyRequest $request)
    {
        ContactCompany::create($request->validated());
    }

    public function edit(ContactCompany $contactCompany)
    {
        return inertia("Contact/Company/Form", compact('contactCompany'));
    }

    public function update(CompanyRequest $request, ContactCompany $contactCompany)
    {
        $contactCompany->update($request->validated());
    }
}
```

La ruta:

```
Route::group([
    'prefix' => 'contact',
    'middleware' => [
        'auth:sanctum',
        config('jetstream.auth_session'),
        'verified',
    ],
    function () {
        Route::resource('contact-general',
App\Http\Controllers>Contact\GeneralController::class)->only(['create','edit','store','update']);
        Route::resource('contact-company',
App\Http\Controllers>Contact\CompanyController::class)->only(['create','edit','store','update']);
    });
});
```

Las validaciones:

```
$ php artisan make:request Contact/CompanyRequest
```

```
<?php
```

```

namespace App\Http\Requests\Contact;

use Illuminate\Foundation\Http\FormRequest;

class Company extends FormRequest
{

    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return [
            'name' => 'required|min:2|max:255',
            'identification' => 'required|min:2|max:50',
            'email' => 'required|min:2|max:80|email',
            'extra' => 'required|min:2|max:255',
            'choices' => 'required',
            'contact_general_id' => 'required',
        ];
    }
}

```

Y su vista o componente en Vue, seguirá la misma estructura que el creado anteriormente:

resources/js/Pages/Contact/Company/Form.vue

```

<template>
    <AppLayout title="Create Category">
        <template #header>
            <h2 class="font-semibold text-xl text-gray-800 leading-tight">
                Create Contact
            </h2>
        </template>

        <div class="max-w-7xl mx-auto py-10 sm:px-6 lg:px-8">
            <FormSection @submitted="submit">
                <template #title>
                    Create Contact
                </template>

                <template #description>
                    Create Contact
                </template>
            </FormSection>
        </div>
    </AppLayout>

```

```

        </template>

        <template #form>
            <div class="col-span-6">
                <InputLabel for="name" value="Name" />
                <TextInput id="name" v-model="form.name" type="text" class="block w-full mt-1"
                           autofocus />
                <InputError :message="errors.name" class="mt-2" />
            </div>
            <div class="col-span-6">
                <InputLabel for="identification" value="Identification" />
                <TextInput id="identification" v-model="form.identification" type="text" class="block w-full mt-1"
                           autofocus />
                <InputError :message="errors.identification" class="mt-2" />
            </div>
            <div class="col-span-6">
                <InputLabel for="email" value="Email" />
                <TextInput id="email" v-model="form.email" type="email" class="block w-full mt-1"
                           autofocus />
                <InputError :message="errors.email" class="mt-2" />
            </div>
            <div class="col-span-6">
                <InputLabel for="extra" value="Extra" />
                <textarea v-model="form.extra" class="block w-full mt-1 border-gray-300 rounded-md"></textarea>
                <InputError :extra="errors.extra" class="mt-2" />
            </div>
            <div class="col-span-6">
                <InputLabel for="choices" value="Choices" />

                <select v-model="form.choices" class="block w-full mt-1 border-gray-300 rounded-md">
                    <option value="Advert">Advert</option>
                    <option value="post">Post</option>
                    <option value="course">Course</option>
                    <option value="movie">Movie</option>
                    <option value="other">Other</option>
                </select>

                <InputError :message="errors.type" class="mt-2" />
            </div>
        </template>

```

```

        <template #actions>
            <PrimaryButton :class="{ 'opacity-25': form.processing }"
:disabled="form.processing">
                Save
            </PrimaryButton>
        </template>
    </FormSection>
</div>
</AppLayout>
</template>

<script>

import { router, useForm } from "@inertiajs/vue3"

import AppLayout from "@/Layouts/AppLayout.vue";

import FormSection from '@/Components/FormSection.vue';
import InputError from '@/Components/InputError.vue';
importInputLabel from '@/Components/InputLabel.vue';
import PrimaryButton from '@/Components/PrimaryButton.vue';
import TextInput from '@/Components/TextInput.vue';

export default {
    props: {
        errors: Object,
        contactCompany: {
            default: {
                id: '',
                name: '',
                identification: '',
                email: '',
                choices: '',
                extra: '',
                contact_general_id: 1
            }
        }
    },
    components: {
        AppLayout,
        FormSection,
        InputError,
       InputLabel,
        PrimaryButton,
        TextInput
    }
}

```

```

},
setup(props) {
  const form = useForm({
    id: props.contactCompany.id,
    name: props.contactCompany.name,
    identification: props.contactCompany.identification,
    email: props.contactCompany.email,
    choices: props.contactCompany.choices,
    extra: props.contactCompany.extra,
    contact_general_id: props.contactGeneralId,
  })

  function submit() {
    if(form.id == '') {
      router.post(route("contact-company.store"), form)
    } else{
      router.put(route("contact-company.update", form.id), form)
    }
  }

  return { form, submit }
}
</script>

```

Y tendremos:

The screenshot shows a 'Create Contact' form with the following fields:

- Name: An input field.
- ID: An input field.
- Email: An input field.
- Extra: An input field.
- Choice: A dropdown menu.

At the bottom right of the form is a 'SEND' button.

Figura 10-5: Formulario paso 2, para contacto empresa

Opcional, Definir un layout personalizado

Usaremos otro layout para manejar el formulario; un layout mucho más sencillo y que se puede usar sin necesidad de estar autenticado:

resources/js/Layouts/ContactLayout.vue

```
<template>
  <header>
    <div v-if="$slots.header" class="mx-auto px-3">
      <slot name="header" />
    </div>
  </header>

  <div class="container">
    <slot />
  </div>
</template>
```

Si el layout anterior funciona para tus necesidades, puedes conservarlo, pero, para el enfoque a cubrir en el libro, no sería necesario.

En cada uno de los archivos **Form.vue** anteriores, actualizamos las referencias al nuevo layout:

Form.vue

```
<template>
  <contact-layout>
    <div class="card">
      <div class="card-body">
        <form @submit.prevent="submit">
          ***
          <PrimaryButton class="mt-5" type="submit">Send</PrimaryButton>
        </form>
      </div>
    </div>
  </contact-layout>
</template>
***

import ContactLayout from "@/Layouts/ContactLayout.vue";
export default {
  components: {
    ContactLayout,
  }
}***
```

Con este layout, no es necesario que el usuario esté autenticado, por lo tanto, se pueden remover los middlewares para proteger con la autenticación y las rutas quedarían como:

routes/web.php

```
Route::group([
    'prefix' => 'contact',
], function () {
    Route::resource('contact-general',
App\Http\Controllers>Contact\GeneralController::class)->only(['create','edit','store','update']);
    Route::resource('contact-company',
App\Http\Controllers>Contact\CompanyController::class)->only(['create','edit','store','update']);
});
```

Formulario, Paso 2: Persona

Los siguientes pasos corresponden al formulario del paso 2 para las personas.

Creamos el controlador:

```
$ php artisan make:controller Contact/PersonController -r
```

El cual tendrá la misma estructura que el controlador de **GeneralController**:

```
<?php

namespace App\Http\Controllers\Contact;

use App\Http\Controllers\Controller;
use App\Http\Requests>Contact\PersonRequest;
use App\Models>ContactPerson;
use Illuminate\Http\Request;

class PersonController extends Controller
{

    public function create()
    {
        return inertia("Contact/Person/Form");
    }

    public function store(PersonRequest $request)
    {
        ContactPerson::create($request->validated());
    }
}
```

```

    }

    public function edit(ContactPerson $contactPerson)
    {
        return inertia("Contact/Person/Form", compact('contactPerson'));
    }

    public function update(PersonRequest $request, ContactPerson $contactPerson)
    {
        $contactPerson->update($request->validated());
    }
}

```

La ruta:

```

Route::group([
    'prefix' => 'contact',
], function () {
    Route::resource('contact-general',
App\Http\Controllers\Contact\GeneralController::class)->only(['create','edit','store','update']);
    Route::resource('contact-company',
App\Http\Controllers\Contact\CompanyController::class)->only(['create','edit','store','update']);
    Route::resource('contact-person',
App\Http\Controllers\Contact\PersonController::class)->only(['create','edit','store','update']);
});

```

Las validaciones:

```

$ php artisan make:request Contact/PersonRequest

<?php

namespace App\Http\Requests\Contact;

use Illuminate\Foundation\Http\FormRequest;

class PersonRequest extends FormRequest
{
    /**
     * Determine if the user is authorized to make this request.
     */
    public function authorize(): bool
    {

```

```

        return true;
    }

/**
 * Get the validation rules that apply to the request.
 *
 * @return array<string, \Illuminate\Contracts\Validation\Rule|array|string>
 */
public function rules(): array
{
    return [
        'name' => 'required|min:2|max:255',
        'surname' => 'required|min:2|max:80',
        'other' => 'required|min:2|max:255',
        'choices' => 'required',
        'contact_general_id' => 'required',
    ];
}
}

```

Y su vista o componente en Vue, seguirá la misma estructura que el creado anteriormente:

resources/js/Pages/Contact/Person/Form.vue

```

<template>
    <ContactLayout>
        <template #header>
            <h2 class="font-semibold text-xl text-gray-800 leading-tight">
                Create Contact
            </h2>
        </template>

        <div class="max-w-7xl mx-auto py-10 sm:px-6 lg:px-8">
            <FormSection @submitted="submit">
                <template #title>
                    Create Contact
                </template>

                <template #description>
                    Create Contact
                </template>

                <template #form>
                    <div class="col-span-6">
                        <InputLabel for="name" value="Name" />

```

```

        <TextInput id="name" v-model="form.name" type="text" class="block
w-full mt-1"
            autofocus />
        <InputError :message="errors.name" class="mt-2" />
    </div>
    <div class="col-span-6">
        <InputLabel for="surname" value="Surname" />
        <TextInput id="surname" v-model="form.surname" type="text"
class="block w-full mt-1"
            autofocus />
        <InputError :message="errors.surname" class="mt-2" />
    </div>

    <div class="col-span-6">
        <InputLabel for="other" value="Other" />
        <textarea v-model="form.other"
            class="block w-full mt-1 border-gray-300
rounded-md"></textarea>
        <InputError :other="errors.other" class="mt-2" />
    </div>
    <div class="col-span-6">
        <InputLabel for="choices" value="Choices" />

        <select v-model="form.choices" class="block w-full mt-1
border-gray-300 rounded-md">
            <option value="Advert">Advert</option>
            <option value="post">Post</option>
            <option value="course">Course</option>
            <option value="movie">Movie</option>
            <option value="other">Other</option>
        </select>

        <InputError :message="errors.type" class="mt-2" />
    </div>
</template>

<template #actions>
    <PrimaryButton :class="{ 'opacity-25': form.processing }"
:disabled="form.processing">
        Save
    </PrimaryButton>
</template>
</FormSection>
</div>
</ContactLayout>
</template>

```

```
<script>

import { router, useForm } from "@inertiajs/vue3"

import ContactLayout from "@/Layouts/ContactLayout.vue";

import FormSection from '@/Components/FormSection.vue';
import InputError from '@/Components/InputError.vue';
importInputLabel from '@/Components/InputLabel.vue';
import PrimaryButton from '@/Components/PrimaryButton.vue';
import TextInput from '@/Components/TextInput.vue';

export default {
    props: {
        errors: Object,
        contactPerson: {
            default: {
                id: '',
                name: '',
                surname: '',
                choices: '',
                other: '',
                contact_general_id: 1
            }
        }
    },
    components: {
        ContactLayout,
        FormSection,
        InputError,
        InputLabel,
        PrimaryButton,
        TextInput
    },
    setup(props) {
        const form = useForm({
            id: props.contactPerson.id,
            name: props.contactPerson.name,
            surname: props.contactPerson.surname,
            choices: props.contactPerson.choices,
            other: props.contactPerson.other,
            contact_general_id: props.contactGeneralId,
        })
    }
}

function submit() {
```

```

        if(form.id == '') {
            router.post(route("contact-person.store"), form)
        } else{
            router.put(route("contact-person.update", form.id), form)
        }
    }

    return { form, submit }
}
</script>

```

Y tendremos:

The form consists of four input fields:

- Name:** Pepe
- Surname:** Cruz
- Other:** 2 No se
- Choice:** Course (with a dropdown arrow)

Below the form is a dark blue button labeled "SEND".

Figura 10-6: Formulario paso 2, para contacto persona

Formulario, Paso 3: Detalle

En este apartado, vamos a crear el proceso para el último paso, que es el de detalle.

Creamos el controlador:

```
$ php artisan make:controller Contact/DetailController -r
```

El cual tendrá la misma estructura que el controlador de **GeneralController**:

```

<?php

namespace App\Http\Controllers\Contact;

use App\Http\Controllers\Controller;
use App\Http\Requests>Contact\DetailRequest;
use App\Models>ContactDetail;
use Illuminate\Http\Request;

class DetailController extends Controller
{

    public function create()
    {
        return inertia("Contact/Detail/Form");
    }

    public function store(DetailRequest $request)
    {
        ContactDetail::create($request->validated());
    }

    public function edit(ContactDetail $contactDetail)
    {
        return inertia("Contact/Detail/Form", compact('contactDetail'));
    }

    public function update(DetailRequest $request, ContactDetail $contactDetail)
    {
        $contactDetail->update($request->validated());
    }
}

```

La ruta:

```

Route::resource('contact-detail',
App\Http\Controllers>Contact\DetailController::class)->only(['create','edit','store','update']);

```

Las validaciones:

```

$ php artisan make:request Contact\DetailRequest
<?php

namespace App\Http\Requests\Contact;

```

```

use Illuminate\Foundation\Http\FormRequest;

class DetailRequest extends FormRequest
{

    public function authorize(): bool
    {
        return true;
    }

    public function rules(): array
    {
        return [
            'extra' => 'required|min:2|max:500',
            'contact_general_id' => 'required',
        ];
    }
}

```

Y su vista o componente en Vue, seguirá la misma estructura que el creado anteriormente:

resources/js/Pages/Contact/Detail.vue

```

<template>
  <ContactLayout>
    <template #header>
      <h2 class="font-semibold text-xl text-gray-800 leading-tight">
        Create Contact
      </h2>
    </template>

    <div class="max-w-7xl mx-auto py-10 sm:px-6 lg:px-8">
      <FormSection @submitted="submit">
        <template #title>
          Create Contact
        </template>

        <template #description>
          Create Contact
        </template>

        <template #form>
          <div class="col-span-6">
            <InputLabel for="extra" value="Extra" />

```

```

        <textarea v-model="form.extra" class="block w-full
mt-1"></textarea>
        <InputError :message="errors.extra" class="mt-2" />
    </div>

</template>

<template #actions>
    <PrimaryButton :class="{ 'opacity-25': form.processing }"
:disabled="form.processing">
        Save
    </PrimaryButton>
</template>
</FormSection>
</div>
</ContactLayout>
</template>

<script>

import { router, useForm } from "@inertiajs/vue3"

import ContactLayout from "@/Layouts/ContactLayout.vue";

import FormSection from '@/Components/FormSection.vue';
import InputError from '@/Components/InputError.vue';
importInputLabel from '@/Components/InputLabel.vue';
import PrimaryButton from '@/Components/PrimaryButton.vue';

export default {
    props: {
        errors: Object,
        contactDetail: {
            default: {
                id: '',
                extra: '',
                contact_general_id: 1
            }
        }
    },
    components: {
        ContactLayout,
        FormSection,
        InputError,
       InputLabel,
        PrimaryButton
    }
}

```

```

},
setup(props) {
    const form = useForm({
        id: props.contactDetail.id,
        extra: props.contactDetail.extra,
        contact_general_id: props.contactGeneralId,
    })

    function submit() {
        if(form.id == '') {
            router.post(route("contact-detail.store"), form)
        } else{
            router.put(route("contact-detail.update", form.id), form)
        }
    }

    return { form, submit }
}
</script>

```

Y tendremos:

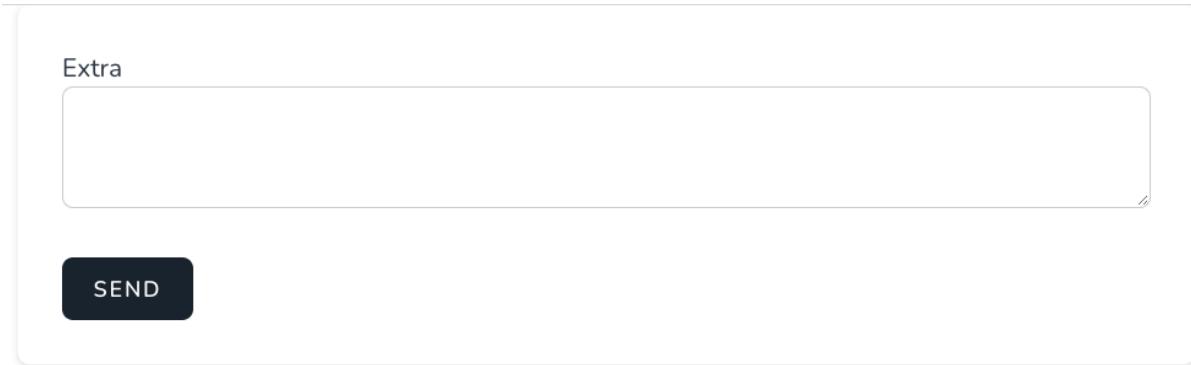


Figura 10-7: Formulario paso 3, para contacto adicional

Componentes de componentes

En este apartado, aprenderemos a reutilizar componentes en Vue dentro de otros componentes; esto es un paso que ya hemos realizado hasta ahora, por ejemplo, cuando usamos los botones, layout de formulario, entre otros, estamos usando componentes dentro de otros componentes; por ejemplo:

resources/js/Pages/Contact/Person/Form.vue

```
import FormSection from '@/Components/FormSection.vue';
```

Lo que no hemos probado hasta ahora es, usar componentes que forman parte de un controlador, es decir, que son devueltos **de manera directa** desde el controlador y por lo tanto, tienen una mayor integración con Laravel; por ejemplo, los llamados **Form.vue**:

```
class CompanyController extends Controller
{
    public function create()
    {
        return inertia("Contact/Company/Form");
    }
}
```

O:

```
class GeneralController extends Controller
{
    public function create()
    {
        return inertia("Contact/General/Form");
    }
}
```

Son componentes más especiales que tienen una integración más fuerte con el controlador y con esto con Laravel y que son usadas en conjunto mediante Inertia como vimos anteriormente, por lo tanto, es lógico pensar que la integración no es tan directa como la que usábamos para componentes de Vue que no se integran directamente desde un controlador o ruta en Laravel; en definitiva, tenemos dos tipos de componentes de Vue en Inertia:

1. Componentes en Vue sin integrar de forma directa desde Laravel, estos son, por ejemplo, los layouts y botones: **DangerButton** y **FormSection**.
2. Componentes en Vue que son usados de manera directa mediante Laravel con la función de **inertia()**, estos son por ejemplo, los de formulario que mostramos en los anteriores bloques de código.

Lo más importante de estos segundos tipos, es que, reciben datos desde un controlador de Laravel, específicamente, el modelo y los errores del formulario, por lo tanto, es un punto que se debe de tener en cuenta cuando se quiere usar este tipo de componentes de manera separada del controlador que lo gobierna.

Lógica para el formulario paso por paso

Todo esto es importante ya que, al trabajar con un formulario paso por paso, es lógico pensar que cada uno de los pasos van a tener que ser integrados en otro componente y no usar de manera directa; vamos a ver dos formas de hacer esto y se muestran en las siguientes imágenes.

Crear un nuevo controlador en la cual se incluyan cada uno de los pasos y sea este el que maneje cual es el paso activo:

Extra Component

Step 1

Step 2

Step n

Figura 10-8: Formulario padre, componente adicional

Usar un controlador existente del formulario paso por paso (por ejemplo, el formulario del primer paso) en la cual se incluya el resto de los pasos y sea este el que maneje cual es el paso activo; en este escenario que es el que vamos a seguir en el libro, usaremos el primer paso (contacto general) para incluir al resto de los pasos y sea este el paso que maneje cual paso está activo en un momento dado:

Step 1 Component

Step 2

Step 3

Step n

Figura 10-9: Formulario padre, componente adicional sin controlador

Usaremos el segundo enfoque sobre el primero al ser el más directo y sencillo, lo cual es ideal para conocer cómo funcionan este tipo de integraciones; aunque, lo más recomendable sería usar el primer enfoque al ser más organizado y modular.

Manejar la lógica del paso por paso desde GeneralController.php

Comencemos creando una nueva vista que se va a utilizar desde el controlador de **GeneralController.php** y en donde, colocaremos uno de los pasos, el de general:

resources/js/Pages/Contact/General/Step.vue

```
<template>
  <div>
    <contact-layout>
      <general-form/>
    </contact-layout>
  </div>
</template>
<script>
import ContactLayout from "@/Layouts/ContactLayout.vue";
import GeneralForm from "@/Pages/Contact/General/Form.vue";

export default {
  components:{
    ContactLayout,
    GeneralForm
  }
}
</script>
```

Este es el componente que devolveremos por defecto desde el controlador en crear y actualizar:

```
class GeneralController extends Controller
{
  public function create()
  {
    return inertia("Contact/General/Step");
  }

  public function store(General $request)
  {
    ContactGeneral::create($request->validated());
  }

  public function edit(ContactGeneral $contactGeneral)
  {
    return inertia("Contact/General/Step", compact('contactGeneral'));
  }

  public function update(General $request, ContactGeneral $contactGeneral)
```

```
{  
    $contactGeneral->update($request->validated());  
}  
}
```

Desde el componente anterior, vamos a empezar a integrar el formulario del primer paso; veremos qué ocurre un error, que, si analizamos desde la consola del navegador, verás que es por el **props de errors**:

```
Cannot read properties of undefined (reading 'subject')  
*** $props.errors.subject
```

El error es bastante claro, nos dice que errors es indefinido y por ende, cualquier cosa que se consulte desde la variable **errors**, daría un error; y es precisamente porque no se está pasando los errores; así que, se los pasamos:

resources/js/Pages/Contact/General/Step.vue

```
<contact-layout>  
    <general-form :errors="errors" />  
</contact-layout>  
***  
props:{  
    errors:Object,  
},
```

Ya por aquí puedes empezar a entender los inconvenientes que tenemos al trabajar con componentes de Vue usados directamente desde Laravel, y es que, hay que inyectar los parámetros necesarios; si intentamos editar un contacto general:

<http://localhost/contact/contact-general/1/edit>

Veremos que no aparece nada en el formulario, y es debido a que, también hay que inyectar el contacto general tal cual hicimos con los errores:

```
<contact-layout>  
    <general-form :errors="errors" :contactGeneral="contactGeneral"/>  
</contact-layout>  
props:{  
    errors:Object,  
    contactGeneral:Object  
,
```

Finalmente, veríamos:

Figura 10-10: Contacto general incrustado en Step.vue

Segundo paso, empresa

Vamos con el paso de las empresas, al cual tenemos que definir la misma lógica que implementamos antes, pasar los errores y el contacto, aunque, en esta ocasión, el contacto no es el general si no el de compañía.

Recordemos que todos los pasos, guardan una relación foránea con el contacto general, según definimos en el modelo:

```
class ContactGeneral extends Model
{
    use HasFactory;

    public $timestamps = false;

    protected $fillable=['subject','message','type'];

    public function person(){
        return $this->hasOne(ContactPerson::class);
    }

    public function company(){
        return $this->hasOne(ContactCompany::class);
    }

    public function detail(){
        return $this->hasOne(ContactDetail::class);
    }
}
```

Por lo tanto, desde el contacto general, que es donde renderizamos el resto de los contactos (en caso de que existan), podemos utilizar el contacto general para traer la referencia al resto de los contactos; recordemos que el

tipo de relación en Laravel es de tipo lazy, lo que significa es que, solamente se traen los datos relacionales cuando se consultan; pensando en esto, vamos a inyectar la misma desde la relación de contacto general; para eso, en el controlador:

app/Http/Controllers/Contact/GeneralController.php

```
public function edit(ContactGeneral $contactGeneral)
{
    $contactGeneral->company;

    if($contactGeneral->company == null)
        unset($contactGeneral->company);

    return inertia("Contact/General/Step", compact('contactGeneral'));
}
```

También se verifica si la misma existe y es distinta de nula, si es así, se eliminan del objeto para evitar causar conflictos desde los componentes de Vue:

```
if($contactGeneral->company == null)
    unset($contactGeneral->company);
```

Y con esto, podemos usarla en el componente en Vue:

resources/js/Pages/Contact/General/Step.vue

```
<template>
    <div>
        <contact-layout>
            <general-form :errors="errors" :contactGeneral="contactGeneral"/>
            <company-form :contactCompany="contactGeneral.company"/>
        </contact-layout>
    </div>
</template>
<script>
import ContactLayout from "@/Layouts/ContactLayout.vue";
import GeneralForm from "@/Pages/Contact/General/Form.vue";
import CompanyForm from "@/Pages/Contact/Company/Form.vue";

export default {
    props:{
        errors:Object,
        contactGeneral:Object
    },
    components:{
        ContactLayout,
        GeneralForm,
```

```

        CompanyForm
    }
}
</script>

```

Al colocarlo e ir al navegador, veremos que sucede los siguientes errores que antes:

```

TypeError: Cannot read properties of undefined (reading 'name')
***
$props.errors.name

```

Así que, colocamos el **prop** de los errores:

```
<company-form :errors="errors" :contactCompany="contactGeneral.company"/>
```

Con esto, debería de funcionar correctamente tanto para crear, como para editar.

Segundo paso, persona

Ya que conocemos cual es la lógica que tenemos que seguir para el paso de empresa, lo podemos replicar para el resto de los pasos; en definitiva, tenemos que colocar el props de los errores y la relación, vamos ahora con el de personas:

```

<template>
  <div>
    <contact-layout>
      <general-form :errors="errors" :contactGeneral="contactGeneral"/>
      <company-form :errors="errors" :contactCompany="contactGeneral.company"/>
      <person-form :errors="errors" :contactPerson="contactGeneral.person"/>
    </contact-layout>
  </div>
</template>
<script>
import ContactLayout from "@/Layouts/ContactLayout.vue";
import GeneralForm from "@/Pages/Contact/General/Form.vue";
import CompanyForm from "@/Pages/Contact/Company/Form.vue";
import PersonForm from "@/Pages/Contact/Person/Form.vue";

export default {
  props:{
    errors:Object,
    contactGeneral:Object
  },
  components:{
    ContactLayout,
    GeneralForm,
    CompanyForm,

```

```

        PersonForm,
    }
}
</script>

```

Y para la entidad, inyectamos desde la relación de contacto general:

app/Http/Controllers/Contact/GeneralController.php

```

public function edit(ContactGeneral $contactGeneral)
{
    $contactGeneral->company;
    $contactGeneral->person;

    if($contactGeneral->person == null)
        unset($contactGeneral->person);
    if($contactGeneral->company == null)
        unset($contactGeneral->company);

    return inertia("Contact/General/Step", compact('contactGeneral'));
}

```

Tercer paso, detalle

Aquí, aplicamos la misma lógica que la mencionada antes; en la vista:

```

<template>
    <div>
        <contact-layout>
            <general-form :errors="errors" :contactGeneral="contactGeneral"/>
            <company-form :errors="errors" :contactCompany="contactGeneral.company"/>
            <person-form :errors="errors" :contactPerson="contactGeneral.person"/>
            <detail-form :errors="errors" :contactDetail="contactGeneral.detail"/>
        </contact-layout>
    </div>
</template>
<script>
import ContactLayout from "@/Layouts/ContactLayout.vue";
import GeneralForm from "@/Pages/Contact/General/Form.vue";
import CompanyForm from "@/Pages/Contact/Company/Form.vue";
import PersonForm from "@/Pages/Contact/Person/Form.vue";
import DetailForm from "@/Pages/Contact/Detail/Form.vue";
import DetailForm from "@/Pages/Contact/Detail/Form.vue";

export default {
    props:{
        errors:Object,

```

```

        contactGeneral:Object
    },
    components:{
        ContactLayout,
        GeneralForm,
        CompanyForm,
        PersonForm,
        DetailForm
    }
}
</script>

```

Y en el controlador:

app/Http/Controllers/Contact/GeneralController.php

```

public function edit(ContactGeneral $contactGeneral)
{
    $contactGeneral->company;
    $contactGeneral->person;
    $contactGeneral->detail;

    if($contactGeneral->person == null)
        unset($contactGeneral->person);
    if($contactGeneral->company == null)
        unset($contactGeneral->company);
    if($contactGeneral->detail == null)
        unset($contactGeneral->detail);

    return inertia("Contact/General/Step", compact('contactGeneral'));
}

```

Ya con esto, tenemos listos y funcionales todos los pasos, ahora resolver pequeños detalles e implementar la lógica para mostrar un paso por vez.

Errores de validaciones

Como un detalle interesante, puedes ver que, en ciertos formularios que manejen el mismo nombrado para los campos, al enviar un formulario invalido, veremos que los errores se dispensan para el resto de los formularios y se muestran los errores para los campos que mantengan el mismo nombrando; como mostramos en la siguiente imagen, que al enviar el formulario de empresa, sin un nombre, muestra el error tanto para la empresa, como para la persona; esto se debe a que, se inyectan los mismos errores para todos los formularios:

Name

The name field is required.

ID

Email

Extra

Choice

Name

The name field is required.

Surname

Figura 10-11: Errores de validaciones en campos del mismo nombre

Esto no tiene importancia ya que, al final, vamos a renderizar un paso por vez, pero es importante conocer el funcionamiento de la aplicación que estamos llevando a cabo.

Crear una función de ayuda para indicar el paso actual

En este apartado, vamos a conocer un mecanismo con el cual nos permitirá mostrar el paso que queremos visualizar en un momento dado, es decir, la idea de todo esto no es que se muestren todos los formularios al mismo tiempo, que es como está actualmente la aplicación, si no un paso por vez, por lo tanto, para esto, vamos a manejar el siguiente esquema:

- Paso 1, mostramos el paso de contacto general.
- Paso 2, mostramos el paso de contacto empresa.
- Paso 2, mostramos el paso de contacto personal.
- Paso 3, mostramos el paso de contacto extra.

Como puedes ver, tenemos dos pasos dos, así que, para mantener esto de manera funcional y saber cuál formulario mostramos según la selección del usuario, podemos usar un valor numérico representado en una variable step o paso:

- Paso 1, el valor de step es 1.
- Paso 2, el valor de step es 2.
- Paso 2, el valor de step es 2.5.
- Paso 3, el valor de step es 3.

Como puedes darte cuenta, el esquema anterior es muy sencillo de escalar para otros pasos adicionales, inclusive si se encuentran en el mismo paso, como sucede con el paso dos, el cual puede ser o empresa o persona.

Pensando en esto, vamos a crear una función de ayuda como la siguiente:

app/Helpers/contact.php

```
<?php

use App\Models>ContactGeneral;

function getStep(ContactGeneral $contactGeneral = null): float
{
    //step 1
    if ($contactGeneral == null)
        return 1;

    //step 2
    if ($contactGeneral->company == null && $contactGeneral->type == 'company')
        return 2.5;
    if ($contactGeneral->person == null && $contactGeneral->type == 'person')
        return 2;

    //step 3
    if ($contactGeneral->detail == null)
        return 3;
```

```
// finish  
return 4;  
}
```

Y se consume desde el controlador padre del paso por paso: **GeneralController.php** en cada una de las funciones controladores que retornan el componente de Vue del formulario (la vista) como veremos un poco más adelante.

Registraremos la función de ayuda para poder usarlo a lo largo del proyecto:

composer.json

```
"autoload": {  
    "psr-4": {  
        "App\\": "app/",  
        "Database\\Factories\\": "database/factories/",  
        "Database\\Seeders\\": "database/seeders/"  
    },  
    "files": [  
        "app/Helpers/contact.php"  
    ]  
},
```

Y lo cargamos en el proyecto:

```
$ composer dump-autoload
```

Con esto, podemos usar esa función de ayuda en cualquier parte de la aplicación, pero, debemos de conocer cómo podemos manejar la misma en el paso por paso, para mostrar el formulario correspondiente según el paso.

Compartir datos, propiedad de step, primeros pasos

Estos valores mostrados anteriormente, se manejan desde una variable o propiedad que debemos de manejar desde **GeneralController.php** y **Step.vue** para mostrar el paso correspondiente; adicional a esto, cada vez que se llene de manera exitosa cualquiera de los formularios que forman parte del paso por paso, se debe de actualizar la referencia a la variable step; es decir, si estamos en el paso uno, significa que se muestra el de contacto general, llegado este formulario y procesado de manera exitosa, la variable de step se actualiza al valor de 2 o 2.5 según la selección del usuario, y así para el resto de los pasos. Como puedes darte cuenta, al usar esta variable de step tanto en el componente padre, el llamado step y en sus hijos que son los pasos, y son los hijos los cuales se encargan de actualizarla, tenemos que escoger un mecanismo que permita manejar esta variable o propiedad global que llamamos step de una manera eficiente.

Compartir datos en Inertia

En Inertia, tenemos una mejor solución al problema anterior y es precisamente la que usamos antes para establecer los mensajes por sesión usando el middleware de **HandleInertiaRequests**; anteriormente vimos cómo usarla usando el middleware, debemos de configurar una nueva variable:

```
app\Http\Middleware\HandleInertiaRequests.php
```

```
public function share(Request $request): array
{
    return array_merge(parent::share($request), [
        'flash' => [
            'message' => $request->session()->get('message')
        ],
        'step' => 1
    ]);
}
```

Y ahora, debemos de establecer los datos del paso por paso de manera manual y con esto, podemos usar desde el controlador padre de los formularios, es decir, el de **GeneralController.php**; la variable (o variables) que queremos tengan alcance global, esto es estupendo ya que, con esto, no hay necesidad de pasar eventos entre componentes ni nada de la lógica especificada antes; la forma manual sería algo como:

```
Inertia::share('appName', config('app.name'))
```

Aunque, en nuestro caso, sería la variable step; esta variable, la usaremos en el componente padre, ya que, cada vez que se envía una petición y redireccionamos, se actualiza toda la aplicación de Vue y con esto, la variable **step**:

```
Inertia::share('step', <VALUE>);
```

Importante colocar las redirecciones en los procesos de guardados de todos los formularios paso por paso, ya que, gracias a las redirecciones, podemos recargar en toda la aplicación en Vue, la referencia a la variable global definida anteriormente:

```
/*
class GeneralController extends Controller
{
    public function store(General $request)
    {
        $contactGeneral = ContactGeneral::create($request->validated());
        return to_route('contact-general.edit', ['contact_general' =>
$contactGeneral->id]);
    }

    public function update(General $request, ContactGeneral $contactGeneral)
    {
        $contactGeneral->update($request->validated());
        return to_route('contact-general.edit', ['contact_general' => $contactGeneral->id]);
    }
}
/*
class PersonController extends Controller
```

```

{
    public function store(Person $request)
    {
        $contactPerson = ContactPerson::create($request->validated());
        return to_route('contact-general.edit', ['contact_general' =>
$contactPerson->general->id]);
    }

    public function update(Person $request, ContactPerson $contactPerson)
    {
        $contactPerson->update($request->validated());
        return to_route('contact-general.edit', ['contact_general' =>
$contactPerson->general->id]);
    }
}

//*****
class CompanyController extends Controller
{

    public function store(Company $request)
    {
        $contactCompany = ContactCompany::create($request->validated());
        return to_route('contact-general.edit', ['contact_general' =>
$contactCompany->general->id]);
    }

    public function update(Company $request, ContactCompany $contactCompany)
    {
        $contactCompany->update($request->validated());
        return to_route('contact-general.edit', ['contact_general' =>
$contactCompany->general->id]);
    }
}

//*****
class DetailController extends Controller
{
    public function store(Detail $request)
    {
        $contactDetail = ContactDetail::create($request->validated());
        return to_route('contact-general.edit', ['contact_general' =>
$contactDetail->general->id]);
    }

    public function update(Detail $request, ContactDetail $contactDetail)
    {
        $contactDetail->update($request->validated());
    }
}

```

```

        return to_route('contact-general.edit', ['contact_general' =>
$contactDetail->general->id]);
    }
}

```

El llamado al método de **Inertia::share('step', <VALUE>)**, vamos a realizarlo desde el método de **edit()** de **GeneralController.php** que a la final, es donde todas las redirecciones de los formularios hijos que forman parte del paso por paso, retornan, por lo tanto, es el sitio ideal que tenemos para realizar la actualización del paso al cual vamos a mostrar, que de momento sería, solamente una prueba con una función de **time()**:

app/Http/Controllers/Contact/GeneralController.php

```

public function edit(ContactGeneral $contactGeneral)
{
    Inertia::share('step', time());
    /**
}

```

En el template, colocamos la impresión de la variable para probar su funcionamiento:

resources/js/Pages/Contact/General/Step.vue

```

<template>
    <div>
        <contact-layout>

            {{ $page.props.step }}

            <general-form :errors="errors" :contactGeneral="contactGeneral"/>
            <company-form :errors="errors" :contactCompany="contactGeneral.company"/>
            <person-form :errors="errors" :contactPerson="contactGeneral.person"/>
                <detail-form :errors="errors" :contactDetail="contactGeneral.detail"/>
            </contact-layout>
        </div>
    </template>

```

La prueba que debes de realizar es, enviar los formularios para crear o actualizar los pasos y veras que el resultado siempre varía, lo que significa es que, la actualización de la variable step se realiza de manera exitosa y podemos usar este esquema para siempre tener el paso actual en el paso por paso.

Si enviamos una vez cualquiera de los pasos por pasos, veremos una fecha:

1657738179

Subject

Test 1

Figura 10-12: Valores de prueba con step

Si enviamos nuevamente, veremos otra fecha:

1657738220

Subject

Test 1

Figura 10-13: Valores de prueba con step, segunda prueba

Compartir datos, propiedad de step, actualizar

Finalmente, podemos devolver desde el controlador el paso que debe de procesar el usuario, así que, en vez de devolver una función de **time()**, devolvemos el paso correspondiente:

app/Http/Controllers/Contact/GeneralController.php

```
public function create()
{
    Inertia::share('step', getStep());
    return inertia("Contact/General/Step");
}

public function edit(ContactGeneral $contactGeneral)
{
```

```

    Inertia::share('step', getStep($contactGeneral));
    /**
}

```

Implementar el paso por paso

Ahora, vamos por el diseño del paso por paso; usualmente cuando estamos en un formulario de paso por paso, se muestra la cantidad de pasos; por supuesto, diseños hay muchos, pero, para nuestro módulo de contacto, usaremos algo como esto:

resources\js\Pages>Contact\General\Step.vue

```

<template>
<div>
<contact-layout>
{{ $page.props.step }}

<div class="flex" v-if="$page.props.step != 4">
<div class="flex mx-auto flex-col sm:flex-row">
<div class="step" :class="{ active: $page.props.step == 1 }">
    STEP 1
</div>
<div
    class="step"
    :class="{ active: parseInt($page.props.step) == 2 }"
>
    STEP 2
</div>
<div class="step" :class="{ active: $page.props.step == 3 }">
    STEP 3
</div>
</div>
</div>

<general-form
    v-if="$page.props.step == 1"
    :errors="errors"
    :contactGeneral="contactGeneral"
/>
<company-form
    v-if="$page.props.step == 2.5"
    :errors="errors"
    :contactCompany="contactGeneral.company"
/>
<person-form
    v-if="$page.props.step == 2"

```

```

:errors="errors"
:contactPerson="contactGeneral.person"
/>
<detail-form
  v-if="$page.props.step == 3"
  :errors="errors"
  :contactDetail="contactGeneral.detail"
/>
<div v-if="$page.props.step == 4">
  <h3 class="text-center"> </h3>
</div>
</contact-layout>
</div>
</template>
<script>
***
```

Y definimos sus clases en el archivo de estilos:

resources/css/app.css

```

.step{
  @apply font-medium px-6 border-b-2 tracking-wider py-2
}

.active{
  @apply border-indigo-500 rounded-t bg-gray-100
}
```

Y obtendremos un paso por paso completamente funcional:



Figura 10-14: Diseño del paso por paso

Es importante notar que, ya implementamos los condicionales para movernos en el paso por paso para los formularios y para indicar que paso esta activo; el casteo a enteros en el paso dos usando la función de `parseInt()` se debe a que el paso dos, puede ser o una compañía, o una persona, representados por el 2.5 y 2 respectivamente, casteando el resultado a un entero, siempre podemos garantizar que vamos a dejar seleccionado el paso dos en cualquiera de los escenarios anteriores.

Opción de paso anterior

Vamos a implementar el botón para regresar al paso anterior, es decir, si estamos en el paso 2, al presionar el mismo, volvemos al paso 1 y si estamos en el paso 3, volvemos al paso 2 al presionar este botón, el botón para volver hacia atrás, va a estar localmente en cada uno de los formularios del paso por paso (menos en el de general, ya que, es el primer paso).

Un punto importante con esto es que, al estar la opción al lado del botón de "Enviar":



Figura 10-15: Botón atrás

El botón, tendría que estar dentro de los formularios:

```
***  
  <PrimaryButton :class="{ 'opacity-25': form.processing }" :disabled="form.processing">  
    Save  
  </PrimaryButton>  
  <span class="mt-1 ml-3 cursor-pointer" @click="$emit('backStepEvent', 2)">Back</span>  
</template>  
</FormSection>
```

Y si colocamos el botón, automáticamente el formulario se enviará al presionar el mismo; si colocamos un enlace, entonces la página se recargaría al darle click y se reiniciaría el estado, cosa que no queremos, por lo tanto, podemos usar un SPAN como el siguiente:

```
<span class="mt-1 ml-3 cursor-pointer">Back</span>
```

El problema que tenemos es que, la gestión de la variable **step** se realiza en el controlador, pero, ahora es necesario variar el valor de la misma desde los componentes hijos (los formularios paso por paso) que son los que implementan dicho botón; para solucionar este problema, podemos usar los eventos personalizados en Vue, en el cual, notificamos al componente padre el paso al que hay que regresar y desde el componente de **Step.vue**, se establece el formulario correspondiente.

Así que, colocamos un evento personalizado indicando el paso al cual queremos regresar:

```
resources\js\Pages\Contact\Company\Form.vue  
resources\js\Pages\Contact\Person\Form.vue
```

```
<template #actions>
```

```

<PrimaryButton :class="{ 'opacity-25': form.processing }" :disabled="form.processing">
    Save
</PrimaryButton>
<span class="mt-1 ml-3 cursor-pointer" @click="$emit('backStepEvent', 1)">Back</span>
</template>

export default {
/**
  emits:['backStepEvent'],
**/
};


```

Y en el de detalle:

resources\js\Pages\Contact\Detail\Form.vue

```

<template #actions>
    <PrimaryButton :class="{ 'opacity-25': form.processing }" :disabled="form.processing">
        Save
    </PrimaryButton>
    <span class="mt-1 ml-3 cursor-pointer" @click="$emit('backStepEvent', 2)">Back</span>
</template>

export default {
/**
  emits:['backStepEvent'],
**/
};


```

En el componente padre, nos suscribimos al evento de los componentes hijos, y en el caso del paso dos, hacemos una verificación en base al tipo para saber a cuál regresar:

resources\js\Pages\Contact\General\Step.vue

```

<general-form
  v-if="$page.props.step == 1"
  :errors="errors"
  :contactGeneral="contactGeneral"
/>
<company-form
  v-if="$page.props.step == 2.5"
  :errors="errors"
  :contactCompany="contactGeneral.company"
  @back-step-event="backStep"
/>
<person-form>
```

```

v-if="$page.props.step == 2"
:errors="errors"
:contactPerson="contactGeneral.person"
@back-step-event="backStep"
/>
<detail-form
  v-if="$page.props.step == 3"
  :errors="errors"
  :contactDetail="contactGeneral.detail"
  @back-step-event="backStep"
/>
/**
*** 
export default {
  methods: {
    backStep(value) {
      console.log(value)
      if (value == 2 && this.contactGeneral && this.contactGeneral.type) {
        // paso 2
        if (this.contactGeneral.type == 'person') {
          this.$page.props.step = 2.5
        } else {
          this.$page.props.step = 2
        }
      } else {
        // paso 1
        this.$page.props.step = value
      }
    }
  }
};

```

Si te fijas en la implementación anterior, al recibir el evento desde con el valor desde los hijos, en el padre, modificamos la propiedad compartida de **step**; podríamos usar otra propiedad adicional, algo como:

```

data() {
  return {
    backStepProps:""
  }
},
/**/
backStep(value){
  this.backStepProps=value
}

```

Pero, usando la propiedad compartida de **step**, nos queda muy limpia la implementación y evitamos colocar múltiples condiciones por distintas propiedades que a la final lo que permitiría es indicar que pasó queremos mostrar.

Parámetro de contacto general para los pasos por pasos

El paso por paso está prácticamente listo, pero, falta resolver un punto muy importante y es que, en los componentes hijos, estamos definiendo a fuerza el identificador del contacto general:

resources/js/Pages/Contact/General/Step.vue

```
router.post(route("****", form.id), {  
    _method: "put",  
    <DATA>  
    contact_general_id: 1,  
});
```

Cosa que no queremos claro está, por lo tanto, para solventar este problema, vamos a pasar como referencia, el identificador del contacto general:

resources/js/Pages/Contact/General/Step.vue

```
<company-form  
  v-if="$page.props.step == 2.5"  
  :errors="errors"  
  :contactCompany="contactGeneral.company"  
  :contactGeneralId="contactGeneral.id"  
  @back-step-event="backStep"  
/>  
<person-form  
  v-if="$page.props.step == 2"  
  :errors="errors"  
  :contactPerson="contactGeneral.person"  
  :contactGeneralId="contactGeneral.id"  
  @back-step-event="backStep"  
/>  
<detail-form  
  v-if="$page.props.step == 3"  
  :errors="errors"  
  :contactDetail="contactGeneral.detail"  
  :contactGeneralId="contactGeneral.id"  
  @back-step-event="backStep"  
/>  
***
```

Y lo usaremos en cada uno de los hijos, si empleas la API de composición:

```
props: {  
  ***  
  contactGeneralId: {
```

```

        type: Number,
    },
},
***  

const form = useForm({
    ***  

    contact_general_id: props.contactGeneralId,
});  

***  

router.post(route("****", form.id), {
    _method: "put",
    <DATA>
    contact_general_id: 1,
});

```

O la API de opciones:

```

props: {
    errors: Object,
    contactGeneralId: {
        type: Number,
    },
    contactPerson: {
        default(props) {
            return {
                id: '',
                name: '',
                surname: '',
                choices: '',
                other: '',
                contact_general_id: props.contactGeneralId
            }
        },
    }
},

```

Ya con esto, completamos el paso por paso para nuestra aplicación; puedes consultar el código fuente en:

<https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.7>

Capítulo 11: Filtros y campos de búsqueda

En este capítulo vamos a tratar como crear filtros y campos de búsquedas mediante campos de selección y de textos empleando por supuesto, los componentes en Vue, con solicitudes al servidor para indicar la ordenación.

Estos filtros los puedes usar para trabajar en cualquier tipo de listado, en este caso enfocado a las tablas paginadas, pero puedes usar la misma estructura para cualquier otro tipo de listado.

Cambios en la tabla

El primer cambio que vamos a realizar es para que la tabla no se rompa cuando generamos los datos de pruebas; específicamente vamos a colocar un límite para el texto y encerrar la descripción en un TEXTAREA, con esto, podremos tener textos tan largos como queramos sin preocuparnos de romper la tabla:

resources/js/Pages/Dashboard/Post/Index.vue

```
<table class="w-full border">
  <thead class="dark:bg-gray-800 bg-gray-100">
    <tr>
      <th class="p-3">Id</th>
      <th class="p-3">Title</th>
      <th class="p-3">Date</th>
      <th class="p-3">Posted</th>
      <th class="p-3">Category</th>
      <th class="p-3">Description</th>
      <th class="p-3">Actions</th>
    </tr>
  </thead>
  <tbody>
    <tr v-for="p in posts.data" :key="p.id">
      <td class="p-2 text-center">{{ p.id }}</td>
      <td class="p-2 text-center">{{ p.title.substring(0, 15) }}</td>
      <td class="p-2 text-center">{{ p.date }}</td>
      <td class="p-2 text-center">{{ p.posted }}</td>
      <td class="p-2 text-center">{{ p.category.title }}</td>
      <td class="p-2 text-center">
        <textarea class="w-48 block m-auto">
          {{ p.description }}
        </textarea>
      </td>
      <td class="p-2">
        <Link class="mr-2 text-sm text-purple-400 hover:text-purple-700"
              :href="route('post.edit', p)">Edit</Link>
        <o-button variant="danger" size="small"
                  @click="confirmDeleteActive = true; deletePostRow =
p.id">Delete</o-button>
      </td>
    </tr>
  </tbody>
</table>
```

```
        </td>
    </tr>
</tbody>
</table>
```

Seeders para los posts

Ahora, para poder trabajar con filtros, vamos a necesitar datos de prueba y para no crearlos de manera manual, podemos usar un seeder el cual como puedes ver la definición, nos enfocamos de que sea lo más aleatorio posible; creamos el seeder:

```
$ php artisan make:seeder PostSeeder
```

Y como puntos importantes, todos los tipos enums y foráneos, son aleatorios; para todas las operaciones, como puedes ver, usamos la función de **rand()** de PHP que nos permite generar números aleatorios, los cuales aprovechamos para generar string de tamaño aleatorio, así como un rango de fechas y los seleccionables:

database/seeders/PostSeeder.php

```
<?php

namespace Database\Seeders;

use App\Models\Post;
use App\Models\Category;

use Carbon\Carbon;
use Illuminate\Database\Seeder;
use Illuminate\Support\Str;

class PostSeeder extends Seeder
{
    /**
     * Run the database seeds.
     */
    public function run(): void
    {
        for ($i=0; $i < 700; $i++) {
            $title = Str::random(rand(2,50));
            Post::create(
                [
                    'title' => $title,
                    'slug' => str($title)->slug(),
                    'text' => Str::random(rand(5,500)),
                    'description' => Str::random(rand(2,150)),
                    'date' => Carbon::today()->subDays(rand(0,400)),
                ]
            );
        }
    }
}
```

```

        'posted' => ['yes', 'not'][rand(0,1)],
        'type' => ['advert', 'post','course','movie'][rand(0,4)],
        'category_id' => Category::inRandomOrder()->first()->id,
    ]
);
}

}
}

```

Finalmente, registramos el seeder:

database/seeders/DatabaseSeeder.php

```

/**
public function run()
{
    // \App\Models\User::factory(10)->create();

    $this->call(PostSeeder::class);
}

```

Y lo ejecutamos:

```
$ php artisan db:seed
```

Filtros para categorías, posteados y tipos

Vamos a comenzar creando nuestros primeros filtros; para ello, vamos a crear la lógica de condicionales necesarios para realizar los filtros en el método de **index()**:

app/Http/Controllers/Dashboard/PostController.php

```

/**
public function index()
{
    $posts = Post::where("id", ">=", 1);
    $categories = Category::get();

    if(request('type')){
        $posts->where('type', request("type"));
    }

    if(request('category_id')){
        $posts->where('category_id', request("category_id"));
    }
}

```

```

if(request('posted')){
    $posts->where('posted', request("posted"));
}

$posts = $posts->with('category')->paginate(10);

return inertia("Dashboard/Post/Index", compact("posts","categories"));
}
/**/

```

Explicación del código anterior

Como puedes darte cuenta, dividimos la consulta para los registros paginados en tres bloques:

1. Lo primero que hacemos es registrar la consulta inicial, como necesitamos alguna referencia al post, **hacemos una condición la cual posibilite seguir utilizando funciones de Eloquent**, en este caso, un **where()** (`where("id", ">=", 1)`) con esto, podemos seguir anexando los filtros y los campos de búsqueda que veremos luego; importante notar que, esta consulta es “muerta”, lo que quiere decir es que todos los registros siempre cumplen esta condición.
2. El apartado de los condicionales son los filtros; para cada filtro un condicional con su respectivo **where()** para filtrar la data.
3. Finalmente, hacemos la consulta final, que en este caso es para obtener los datos paginados.
4. También puedes ver que se buscan todas las categorías en la base de datos.

También, podemos aplicar crear una instancia limpia del Post:

```
$posts = new Post();
```

Pero en cada filtro, debemos de colocar una igualdad, por ejemplo:

```

if (request('type')) {
    $posts = $posts->where('type', request('type'));
}

```

Esto se debe al tipo de dato devuelto, en este caso:

```
$posts = Post::where("id", ">=", 1);
```

Es de tipo Database/Eloquent/Builder, y en el:

```
$posts = new Post();
```

Es de tipo App/Models/Post.

En la vista o componente en Vue, usamos exactamente los mismos campos de selección que usamos para crear/actualizar registros:

`resources/js/Pages/Dashboard/Post/Index.vue`

```

***

<div class="grid grid-cols-2 gap-2 mb-2">
    <select class="rounded-md w-full border-gray-300" v-model="posted">
        <option :value="null">Posted</option>
        <option value="not">No</option>
        <option value="yes">Yes</option>
    </select>

    <select class="rounded-md w-full border-gray-300" v-model="type">
        <option :value="null">Type</option>
        <option value="advert">Advert</option>
        <option value="post">Post</option>
        <option value="course">Course</option>
        <option value="movie">Movie</option>
    </select>

    <select class="rounded-md w-full border-gray-300" v-model="category_id">
        <option :value="null">Category</option>
        <option v-for="c in categories" :value="c.id" :key="c.id">
            {{ c.title }}
        </option>
    </select>
    <PrimaryButton @click="customSearch">
        Filter
    </PrimaryButton>
</div>
<table class="table w-full border">
***

<script>
//***
props: {
  posts: Object,
  categories: Object,
},
data() {
  return {
    confirmDeleteActive: false,
    deletePostRow: "",
    type: "",
    category_id: "",
    posted: "",
  };
},
methods: {
  deletePost() {
    router.delete(route('post.destroy', this.deletePostRow))
  }
}
</script>

```

```

        this.confirmDeleteActive = false
    },
    customSearch() {
        router.get(route('post.index', {
            category_id: this.category_id,
            type: this.type,
            posted: this.posted,
        }))
    }
},
/***/
<script>

```

Y tendremos:

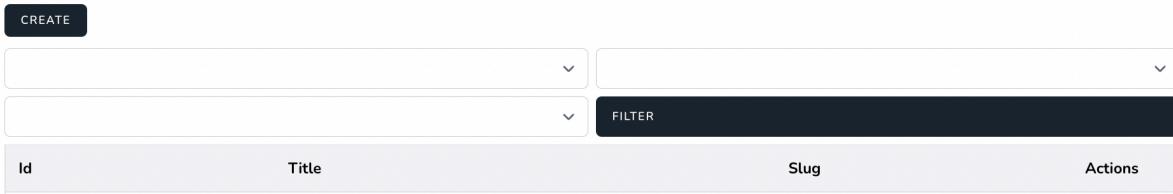


Figura 11-1: Filtros iniciales

Campo de búsqueda para id, título y descripción

Siguiendo la misma lógica y estructura anterior, ahora vamos con los campos de búsqueda; como puedes ver en el presente código, usamos un **where()** agrupado para buscar coincidencias parciales mediante los LIKE; aquí puedes colocar otros campos por los que te gustaría buscar:

resources/js/Pages/Dashboard/Post/Index.vue

```

public function index()
{
    $posts = Post::where("id", ">=", 1);
    $categories = Category::get();

    if(request('search')){
        $posts->where(function($query){
            $query->orWhere("id", "like", "%".request("search")."%")
                ->orWhere("title", "like", "%".request("search")."%")
                ->orWhere("description", "like", "%".request("search")."%")
            ;
        });
    }

    if(request('type')){

```

```

        $posts->where('type', request("type"));
    }
    if(request('category_id')){
        $posts->where('category_id', request("category_id"));
    }
    if(request('posted')){
        $posts->where('posted', request("posted"));
    }

    $posts = $posts->paginate(10);

    return inertia("Dashboard/Post/Index", compact("posts","categories"));
}

```

Al final, hacemos condicionales para verificar si estamos recibiendo el filtro mediante el **request** y si es así, aplicamos el filtro.

Para la vista, adaptamos otro contenedor y colocamos el campo de tipo texto y modificamos la petición:

resources/js/Pages/Dashboard/Post/Index.vue

```

<div class="grid grid-cols-2 mb-2 gap-2">
    <TextInput class="w-full" type="text" placeholder="Search..." v-model="search" />
//***
<script>
  ***
  import TextInput from "@/Components/TextInput.vue";
  ***
  components: {
    ***
    TextInput
  },
  ***
  data() {
    return {
      confirmDeleteActive: false,
      deletePostRow: "",
      type: "",
      category_id: "",
      posted: "",
      search: "",
    };
  },
  ***
  customSearch() {
    router.get(

```

```

        route("post.index", {
            category_id: this.category_id,
            type: this.type,
            posted: this.posted,
            search: this.search,
            from: this.from,
            to: this.to,
        })
    );
},
router.get(
    route("post.index", {
        category_id: this.category_id,
        type: this.type,
        posted: this.posted,
        search: this.search,
    })
);
***  

<script>

```

Y tendremos:

The screenshot shows a top navigation bar with a 'CREATE' button. Below it is a search bar containing the placeholder 'Search...'. Underneath the search bar are two dropdown menus. A 'FILTER' button is visible above a table header. The table has columns for 'Id', 'Title', 'Slug', and 'Actions'.

| Id | Title | Slug | Actions |
|----|-------|------|---------|
|----|-------|------|---------|

Figura 11-2: Filtros campo de búsqueda

Filtrar por rango de fecha

Otro filtro interesante es el que permite filtrar por rangos de fecha; para esto, vamos a usar dos campos para definir la cota inicial y final para el rango, y recuerda que es necesario convertir al tipo de dato correspondiente (fecha en este caso) para poder realizar las comparaciones correctamente desde el `where()`:

app/Http/Controllers/Dashboard/PostController.php

```

//***
if (request('from') && request('to')) {
    $posts->whereBetween('date', [date(request("from")), date(request("to"))]);
}
//***

```

Desde el componente en Vue, creamos un par de campos de tipo fecha más para dichos rangos:

resources/js/Pages/Dashboard/Post/Index.vue

```
<TextInput class="w-full" type="date" placeholder="Date From" v-model="from" />
<TextInput class="w-full" type="date" placeholder="Date To" v-model="to" />
***
<script>
  data() {
    return {
      confirmDeleteActive: false,
      deletePostRow: "",
      type: "",
      category_id: "",
      posted: "",
      search: "",
      from: "",
      to: ""
    };
  },
***
  router.get(
    route("post.index", {
      category_id: this.category_id,
      type: this.type,
      posted: this.posted,
      search: this.search,
      from: this.from,
      to: this.to,
    })
  );
***
</script>
```

Y tendremos:

The screenshot shows a user interface for filtering posts. At the top left is a 'CREATE' button. Below it is a search bar with placeholder 'Buscar por id, título o descripción'. To its right are two date input fields: one showing '07/07/2022' and another showing '09/07/2022'. Below these are two dropdown menus. A 'FILTER' button is located above a table header. The table has columns for 'Id', 'Title', 'Slug', and 'Actions'. The 'Actions' column contains a small icon.

Figura 11-3: Filtros rango de fecha

Valores aplicados en los filtros desde el componente en Vue

Para poder conservar los valores anteriores al momento de enviar una petición desde los filtros en el componente de Vue, debemos de pasar el **request** con el campo específico desde el controlador para poder restablecer los valores del filtro:

app\Http\Controllers\Dashboard\PostController.php

```
public function index()
{
    $posts = Post::where("id", ">=", 1);
    $categories = Category::get();
    $search = request('search');
    $from = request('from');
    $to = request('to');
    $type = request('type');
    $category_id = request('category_id');
    $posted = request('posted');

    if (request('search')) {
        $posts->where(function ($query) {
            $query->orWhere("id", "like", "%" . request("search") . "%")
                ->orWhere("title", "like", "%" . request("search") . "%")
                ->orWhere("description", "like", "%" . request("search") . "%");
        });
    }

    if (request('from') && request('to')) {
        $posts->whereBetween('date', [date(request("from")), date(request("to"))]);
    }

    if (request('type')) {
        $posts->where('type', request("type"));
    }
    if (request('category_id')) {
        $posts->where('category_id', request("category_id"));
    }
    if (request('posted')) {
        $posts->where('posted', request("posted"));
    }

    $posts = $posts->paginate(10);

    return inertia("Dashboard/Post/Index", ["posts" => $posts, "categories" => $categories,
    "prop_posted" => $posted, "prop_category_id" => $category_id, "prop_type" =>
    $type, "prop_from" => $from, "prop_to" => $to, "prop_search" => $search]);
}
```

```
}
```

Desde el componente en Vue se declaran los props adicionales para inicializar los campos de formulario y mediante estos props, se inicializan los **v-model** de los formularios desde la función de **data()**:

resources/js/Pages/Dashboard/Post/Index.vue

```
<script>
  props: {
    posts: Object,
    categories: "",
    prop_type: String,
    prop_category_id: String,
    prop_posted: String,
    prop_search: String,
    prop_from: String,
    prop_to: String,
  },
  data() {
    return {
      confirmDeleteActive: false,
      deletePostRow: "",
      type: this.prop_type,
      category_id: this.prop_category_id,
      posted: this.prop_posted,
      search: this.prop_search,
      from: this.prop_from,
      to: this.prop_to,
    };
  },
</script>
```

Con esto, al enviar el formulario del filtro, se deben de conservar las selecciones y valores de los campos de formulario.

Aplicar filtros sin usar un botón

Otra característica deseada en la aplicación, es que los filtros podamos usarlos sin necesidad de presionar un botón para aplicar los cambios, dicho botón, a la final lo único que hace es enviar los datos al servidor mediante una petición de Inertia; por lo tanto, podemos sustituir este botón en la mayoría de los casos, por eventos nativos en los propios campos de formulario:

- Para el campo de búsqueda, usamos un evento de teclado, por ejemplo, en **@keyup**.
- Para los campos de selección, usamos el evento **@change**; a excepción de los filtros por rango de fecha, que ambos tienen que estar presente, y en este caso, se usa un botón tal cual tenemos implementado o

implementamos también el evento **@change** en los campos de fecha, solamente en el último campo para evitar enviar el filtro por fechas solamente seleccionando la primera fecha (el **from**).

Finalmente, tenemos:

```
<TextInput class="w-full" type="text" sq v-model="search" placeholder="Search by id, title or description"></TextInput>

<select @change="customSearch" class="rounded-md w-full border-gray-300" v-model="posted">
  <option value="not">No</option>
  <option value="yes">Yes</option>
</select>

<select @change="customSearch" class="rounded-md w-full border-gray-300" v-model="type">
  <option value="advert">Advert</option>
  <option value="post">Post</option>
  <option value="course">Course</option>
  <option value="movie">Movie</option>
</select>

<select @change="customSearch"
  class="rounded-md w-full border-gray-300"
  v-model="category_id">
>
  <option value=""></option>
  <option v-for="c in categories" :value="c.id" :key="c.id">
    {{ c.title }}
  </option>
</select>

<TextInput class="w-full" type="date" placeholder="Date From" v-model="from" />
<TextInput class="w-full" @change="customSearch" type="date" placeholder="Date To" v-model="to" />
```

debounce, retardo en los eventos

Un problema que tendríamos cuando pasemos la aplicación a producción, es que, cada vez que usamos el campo de búsqueda, veremos que automáticamente se dispara la petición al servidor, es decir, cada vez que el usuario escribe sobre dicho campo, se va enviando una petición, lo cual es muy ineficiente ya que, usualmente el usuario quiere colocar al menos una palabra para realizar la búsqueda, por lo tanto, lo ideal es, esperar al menos unos pocos milisegundos antes de enviar la petición, para eso, vamos a aplicar un retardo o debounce a dicho campo; Vue, no soporta de manera nativa esta característica, por lo tanto, tenemos que usar un plugin como el siguiente:

<https://www.npmjs.com/package/vue-debounce>

El cual instalamos con:

```
$ npm i vue-debounce
```

Y configuramos:

```
resources/js/app.ts
```

```
import { vueDebounce } from 'vue-debounce'  
//***  
myApp  
  .directive('debounce', vueDebounce ({ lock: true }))  
  .mount(el);
```

Para usarlo, tenemos varias maneras como puedes revisar en la documentación oficial, pero, al interesarnos solamente aplicar en un solo campo, podemos usar la siguiente implementación:

```
resources/js/Pages/Dashboard/Post/Index.vue
```

```
<!-- <TextInput autofocus @keyup="customSearch" class="w-full" type="text"  
placeholder="Search..." v-model="search" /> -->  
<TextInput autofocus v-debounce.500ms="customSearch" :debounce-events="[ 'keyup' ]"  
class="w-full" type="text" placeholder="Search..." v-model="search" />
```

En la cual, indicamos a cuál función queremos llamar pasado X tiempo:

```
v-debounce.500ms="customSearch"
```

Aunque puede ser aplicada a por ejemplo segundos:

```
v-debounce.1s="customSearch"
```

Y cuáles son los eventos que queremos escuchar:

```
:debounce-events="[ 'keyup' ]"
```

Ahora, cada vez que escribamos algo, tarda medio segundo antes de enviar la petición, dando un tiempo prudencial para que el usuario termine de escribir.

Ya con esto, completamos los filtros y campo de búsqueda aplicado a nuestra aplicación.

Filtros con cláusulas condicionales, when

Vamos a crear los filtros que implementamos antes mediante condicionales, con "modo Laravel" que es el recomendado por el equipo de Laravel y ya queda de parte del lector cuál prefiere emplear.

Para esto, emplearemos el **when()** en el cual, el primer parámetro corresponde a la propiedad a evaluar, que en caso de que el valor este definido, ejecuta el segundo parámetro que corresponde al clausure:

```
|Model::when(request('search'), function (Builder $query, string $search) ...
```

El closure que corresponde al segundo parámetro recibe por parámetros el **\$query** para aplicar cualquier tipo de operación con Eloquent y el parámetro evaluado:

```
|Model::when(request('search'), function (Builder $query, string $search) {
    $query->where(function ($query) use ($search) {
        $query-><WHERE>
    })
})
```

Quedando como:

```
app\Http\Controllers\Dashboard\PostController.php
```

```
public function index()
{
    $categories = Category::get();

    //data
    $type = request('type');
    $category_id = request('category_id');
    $posted = request('posted');
    $from = request('from');
    $to = request('to');
    $search = request('search');
    //data

    $posts = Post::when(request('search'), function (Builder $query, string $search) {
        $query->where(function ($query) use ($search) {
            $query->orWhere('id', 'like', "%" . $search . "%")
                ->orWhere('title', 'like', "%" . $search . "%")
                ->orWhere('description', 'like', "%" . $search . "%");
        });
    })
        ->when(request('type'), function (Builder $query, string $type) {

            $query->where('type', $type);
        })
        ->when(request('category_id'), function (Builder $query, string $category_id) {
            $query->where('category_id', $category_id);
        })
        ->when(request('posted'), function (Builder $query, string $posted) {
            $query->where('posted', $posted);
        })
        ->when(request('to'), function (Builder $query, string $to) {
```

```

        $query->whereBetween('date', [
            date(request("from")),
            date($to)
        ]);
    })
->with('category')
->orderBy($sortColumn, $sortDirection)
->paginate(15);

return inertia(
    'Dashboard/Post/Index',
    [
        'posts' => $posts,
        'categories' => $categories,
        'prop_type' => $type,
        'prop_posted' => $posted,
        'prop_category_id' => $category_id,
        'prop_from' => $from,
        'prop_to' => $to,
        'prop_search' => $search,
    ]
);
}
}

```

Limpiar filtros

Para limpiar los filtros, usaremos un nuevo botón junto con una petición sin argumentos al controlador que construye este listado:

```

<PrimaryButton @click="customSearch">
    Filter
</PrimaryButton>
<SecondaryButton class="ml-3" @click="cleanFilter">
    Clear
</SecondaryButton>
*** 
cleanSearch() {
    router.get(route("post.index"));
},

```

Y quedaría:

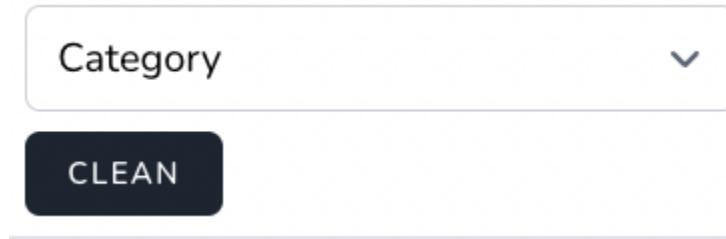


Figura 11-4: Filtros rango de fecha

Código fuente:

<https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.8>

Capítulo 12: Ordenación de las columnas

En este capítulo, vamos a trabajar en ordenación de columnas de manera individual y ascendente/descendente en el listado.

El esquema que vamos a seguir, para poder ordenar fácilmente todas las columnas de un listado, es definiendo las columnas que vamos a mostrar en el listado a nivel de variable declarada en nuestro controlador y que pasaremos por referencia a la vista o componente en Vue.

Usaremos estas variables desde la vista a nivel de **props**, para inicializar las propiedades a nivel de la función **data()** para poder cambiar los valores según la selección de ordenación del usuario:

resources/js/Pages/Dashboard/Post/Index.vue

```
props: {
  ***
  prop_sortDirection: String,
  prop_sortColumn: String,
},
data() {
  return {
    sortColumn: this.prop_sortColumn,
    sortDirection: this.prop_sortDirection,
    ***
  };
},
```

Además de esto, pasaremos un tercer argumento, que corresponde a las columnas ordenables:

resources/js/Pages/Dashboard/Post/Index.vue

```
props: {
  //***
  sortColumn: "",
  sortDirection: "",
  columns: "",
},
```

Desde el componente en Vue, construimos el encabezado de las tablas usando el **prop** anterior:

```
<thead class="dark:bg-gray-800 bg-gray-100">
  <tr class="border-b">
    <th v-for="c in columns" class="p-3" :key="c">
      {{ c }}
    </th>
    <th class="p-3">Actions</th>
```

```
</tr>
</thead>
```

E indicamos aquí el evento de click para forzar la ordenación usando una nueva función cuya definición mostraremos más adelante, a la final la ordenación será otro dato manejado desde el controlador, de manera similar a los filtros:

```
<thead class="dark:bg-gray-800 bg-gray-100">
  <tr class="border-b">
    <th v-for="(c, k) in columns" class="p-3" :key="c">
      <button @click="sort(k)">
        {{ c }}
        <template v-if="k == sortColumn">
          <template v-if="'asc' == sortDirection">
            &uarr;
          </template>
          <template v-else>
            &darr;
          </template>
        </template>
      </button>
    </th>

    <th class="p-3">Actions</th>
  </tr>
</thead>
```

En el código anterior usamos dos condicionales; el primer condicional permite verificar por cual columna se está ordenando, y el segundo argumento si es ascendente o descendente, en este caso aplicado para mostrar un icono de una flecha.

Desde el controlador, usamos los filtros para ordenar, es importante notar que, siempre estamos aplicando una ordenación, por lo tanto, en caso de que no estén definidos, damos un valor por defecto, ya con esto, cada vez que demos un click sobre alguno de los títulos en el encabezado, se actualizan las referencias y con esto, la columna a la cual se va a realizar el orden:

app/Http/Controllers/Dashboard/PostController.php

```
$categories = Category::get();
$search = request('search');
$from = request('from');
$to = request('to');
$type = request('type');
$category_id = request('category_id');
$posted = request('posted');

$sortColumn = request('sortColumn') ?? 'id';
```

```

$sortDirection = request('sortDirection') ?? 'desc';

$posts = Post::with('category')->orderBy($sortColumn, $sortDirection);
/***

```

Para las columnas, la puedes definir como una variable en la función o como una propiedad, en la misma pasamos las columnas y sus labels a usar en la construcción de los encabezados de la tabla:

app/Http/Controllers/Dashboard/PostController.php

```

class PostController extends Controller
{

    public $columns = [
        'id' => 'Id',
        'title' => 'Title',
        'date' => 'Date',
        'posted' => 'Posted',
        'category_id' => 'Category',
        'description' => 'Description',
        'type' => 'Type'
    ];

    public function index()
    {
        /**
    }
}

```

Desde la función de búsqueda que teníamos originalmente, hacemos los cambios para que puedan ordenar también:

```

data() {
    return {
        confirmDeleteActive: false,
        deletePostRow: "",
        column: "id",
    };
},
/**
customSearch() {
    router.get(
        route("post.index", {
            category_id: this.category_id,
            type: this.type,
            posted: this.posted,

```

```

        search: this.search,
        from: this.from,
        to: this.to,
        sortColumn: this.column,
        sortDirection: this.sortDirection == 'asc' ? 'desc' : 'asc',
    })
);
},
sort(column) {
    this.sortColumn = column
    this.customSearch()
}

```

Importante notar:

- Indicamos una propiedad extra que usaremos como pivote para saber por cual columna vamos a ordenar y anexamos las columnas de ordenación al método de **customSearch()** y damos un valor por defecto; recuerda que este método también es empleada por los filtros y por lo tanto, no se pasaría el argumento de la columna.
- Para variar la dirección, usamos el típico toggle: **this.sortDirection == 'asc' ? 'desc' : 'asc'** por lo tanto, si la columna es ascendente y le damos un click, ahora es descendente y viceversa.

Y finalmente, pasamos todos los argumentos a la vista:

```

return inertia("Dashboard/Post/Index", [
    "columns" => $this->columns,
    'posts' => $posts,
    'categories' => $categories,
    'prop_type' => $type,
    'prop_search' => $search,
    'prop_category_id' => $category_id,
    'prop_posted' => $posted,
    'prop_from' => $from,
    'prop_to' => $to,
    "prop_sortDirection" => $sortDirection,
    "prop_sortColumn" => $sortColumn
]);

```

Ya con esto, completamos la ordenación de las columnas del listado de posts; basta con darle un click a alguna de las columnas para realizar la ordenación:

The screenshot shows a browser window with the URL `localhost/dashboard/post?sortColumn=date&sortDirection=asc` in the address bar. Below the address bar is a table with four columns: `Id`, `Date ↑`, `Title`, and a fourth column represented by a downward arrow icon. The `Date ↑` column is currently highlighted, indicating it is the active sort column.

Figura 12-1: Ordenación de las columnas

Puedes consultar el código fuente en:

<https://github.com/libredesarrollo/curso-libro-laravel-10-inertia-store/releases/tag/v0.9>

Detalles finales

En este apartado, vamos a resolver pequeños detalles que quedaron pendiente al momento de crear el filtro para el listado anterior.

Foco en el campo de texto

Cuando escribimos en el campo de texto:



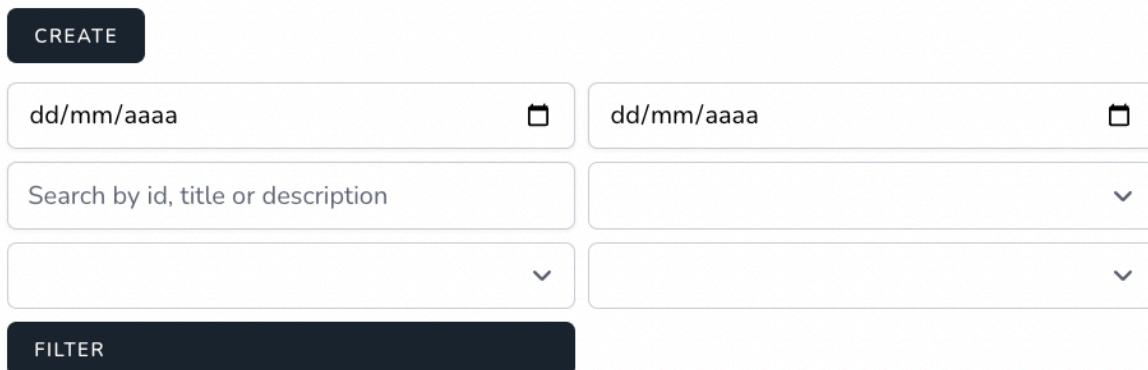
Figura 12-2: Campo de texto con foco

Veremos que al enviarse la petición al servidor, el foco al campo se pierde y por ende, si el usuario está escribiendo, tiene que volver a dar un click sobre dicho campo e intentar nuevamente; para evitar este comportamiento, vamos a dejar el foco activo para dicho campo:

```
<TextInput class="w-full" type="text" autofocus v-debounce.500ms="customSearch"
:debounce-events="['keyup']" v-model="search" placeholder="Search by id, title or
description" ></TextInput>
```

Definir un texto para la opción por defecto en los SELECT

Actualmente el filtro luce como:



The form consists of several input fields and buttons. At the top left is a dark button labeled 'CREATE'. Below it are two date input fields, each with a placeholder 'dd/mm/aaaa' and a small calendar icon. To the right of these are two dropdown menus, both with a placeholder 'Search by id, title or description' and a downward arrow icon. At the bottom is a large, dark, rectangular button labeled 'FILTER' in white capital letters.

Figura 12-3: Placeholder para los campos de texto

Con lo cual, no se sabe exactamente qué hace cada campo; en estos casos, podríamos colocar un LABEL, cosa que haremos para el rango de la fecha:

```
<div class="grid grid-cols-2 gap-2 mb-2">
  <div class="col-span-2 border grid grid-cols-2 gap-2 p-3">
    <InputLabel value="Date From" />
    <InputLabel value="Date To" />
    <TextInput class="w-full" type="date" placeholder="Date From" v-model="from" />
    <TextInput class="w-full" @change="customSearch" type="date" placeholder="Date To" />
  </div>
<script>
import TextInput from "@/Components/TextInput.vue";
components: {
  ***
  TextInput
},
</script>
```

Pero, para simplificar la interfaz en los campos de selección, estableceremos una opción por defecto:

```
<select
  @change="customSearch"
  class="rounded-md w-full border-gray-300"
  v-model="posted"
>
  <option :value="null">Posted</option>
  ***
</select>

<select
  @change="customSearch"
  class="rounded-md w-full border-gray-300"
  v-model="type"
>
  <option :value="null">Type</option>
  ***
</select>

<select
  @change="customSearch"
  class="rounded-md w-full border-gray-300"
  v-model="category_id"
>
  <option :value="null">Category</option>
  ***
```

```
|</select>
```

Y quedaría como:

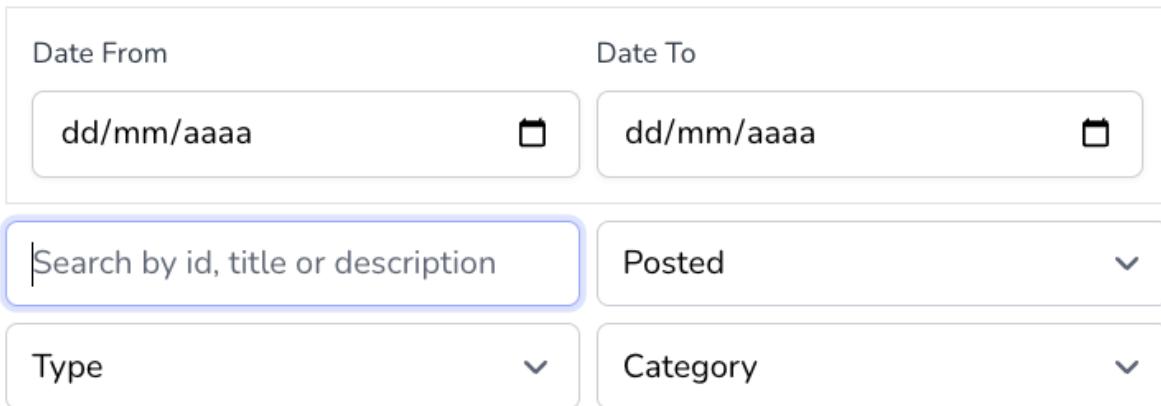


Figura 12-4: Placeholder para los campos de texto

Botón de filtro

Para el botón de filtrar, lo colocaremos dentro de un DIV para que no ocupe todo el largo:

```
<div>
  <PrimaryButton @click="customSearch">
    Filter
  </PrimaryButton>
</div>
```

Y quedaría:

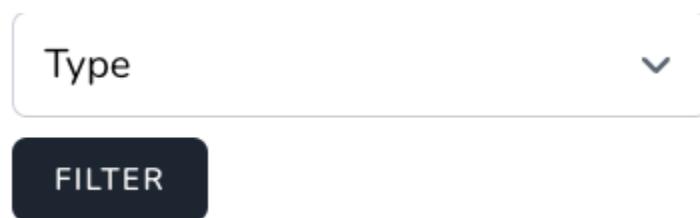


Figura 12-5: Botón para filtrar

Puedes consultar el código fuente en:

<https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.9>

Capítulo 13: Web simple de Blog

Esta sección sirve como reforzamiento de parte de los temas tratados en las anteriores secciones (y aparte de que, vamos a necesitar de este desarrollo para los próximos capítulos), vamos a crear un módulo para un sencillo blog de cara al usuario final, el cual consta de un listado y una página de detalle, empleando por supuesto, Laravel Inertia.

Para la creación del controlador:

```
$ php artisan make:controller Blog\PostController
```

Rutas

Y la definición de las rutas:

```
Route::group([
    'prefix' => 'blog',
], function () {
    Route::get('/', [App\Http\Controllers\Blog\PostController::class,
    'index'])->name("web.index");
```

```
Route::get('/{post:slug}', [App\Http\Controllers\Blog\PostController::class,
    'show'])->name("web.show");
});
```

Importante nota que, son rutas sin login requerido ya que, el propósito es que puedan ser consumidas por un usuario final, entiéndase de publicaciones realizadas y compartidas para todos por redes sociales, indexadas en Google, etc.

Layout para la web

Al ser un módulo aparte al de dashboard, es necesario crear un layout extra en donde el login no sea necesario y que podamos ajustar a nuestras necesidades; para ello, vamos a clonar el de contacto:

```
resources/js/Layouts/ContactLayout.vue
```

Y lo llamaremos como:

```
resources/js/Layouts/WebLayout.vue
```

Listado de Post

El código para el componente de Post, para el módulo del blog, es exactamente el mismo que el usado para el proceso de gestión del post (su listado) pero:

- Removiendo la ordenación por columnas.
- Removiendo el filtro para filtrar si están posteados.
- Obteniendo los posts que se encuentran publicados.
- Usando el nuevo layout.

```
app/Http/Controllers/Blog/PostController.php
```

```
<?php

namespace App\Http\Controllers\Blog;

use App\Http\Controllers\Controller;
use App\Models\Category;
use App\Models\Post;
use Illuminate\Http\Request;

class PostController extends Controller
{
    public function index()
    {

        $categories = Category::get();
        $search = request('search');
```

```

$from = request('from');
$to = request('to');
$type = request('type');
$category_id = request('category_id');

$posts = Post::with('category')->where('posted', 'yes');

if (request('search')) {
    $posts->where(function ($query) {
        $query->orWhere("id", "like", "%" . request("search") . "%")
            ->orWhere("title", "like", "%" . request("search") . "%")
            ->orWhere("description", "like", "%" . request("search") . "%");
    });
}

if (request('from') && request('to')) {
    $posts->whereBetween('date', [date(request("from")), date(request("to"))]);
}

if (request('type')) {
    $posts->where('type', request("type"));
}
if (request('category_id')) {
    $posts->where('category_id', request("category_id"));
}

$posts = $posts->paginate(10);

return inertia("Blog/Index", ["posts" => $posts, "categories" => $categories,
"prop_category_id" => $category_id, "prop_type" => $type, "prop_from" => $from, "prop_to"
=> $to, "prop_search" => $search, ]);
}
}

```

La vista nuevamente es similar al listado empleado en el módulo de dashboard, pero variando la tabla por un listado en base a bloques en DIVs, alineadas con Flex y cambiando la estructura de cada publicación:

resources/js/Pages/Blog/Index.vue

```

<template>
  <web-layout>
    <div class="container">
      <div class="card">
        <div class="card-body">

          <div class="grid grid-cols-2 gap-2 mb-3">

```

```

        <!-- <TextInput autofocus @keyup="customSearch" class="w-full"
type="text" placeholder="Search..." v-model="search" /> -->
            <TextInput autofocus v-debounce.500ms="customSearch"
:debounce-events="['keyup']" class="w-full"
                type="text" placeholder="Search..." v-model="search" />

            <select @change="customSearch" class="rounded w-full border-gray-300"
v-model="type">
                <option :value="null">Type</option>
                <option value="advert">Advert</option>
                <option value="post">Post</option>
                <option value="course">Course</option>
                <option value="movie">Movie</option>
            </select>
            <select @change="customSearch" class="rounded w-full border-gray-300"
v-model="category_id">
                <option :value="null">Category</option>
                <option v-for="c in categories" :value="c.id" :key="c">{{ c.title
}}</option>
            </select>

            <TextInput class="w-full" type="date" placeholder="Date From"
v-model="from" />
            <TextInput class="w-full" @change="customSearch" type="date"
placeholder="Date To"
                v-model="to" />

            <div>
                <PrimaryButton @click="customSearch">
                    Filter
                </PrimaryButton>

                <SecondaryButton class="ml-3" @click="cleanFilter">
                    Clear
                </SecondaryButton>
            </div>
        </div>
    </div>

    <div class="flex flex-col items-center mt-5">
        <div class="w-full sm:max-w-4xl overflow-hidden">
            <div v-for="p in posts.data" class="p-3" :key="p">

```

```

        <h4 class="text-center text-4xl mb-3">{{ p.title }}</h4>
        <p class="
            text-center text-sm text-gray-500
            italic
            font-bold
            uppercase
            tracking-widest
        ">
            {{ p.date }}
        </p>

        <p class="mx-4">{{ p.description }}</p>

        <div class="flex flex-col items-center mt-7">
            <a class="btn-primary" :href="route('web.show', p.slug)">Read
more!</a>
        </div>

        <hr class="my-16">

        </div>
    </div>
</div>

        <pagination class="my-4" :links="posts" />
    </div>
</web-layout>
</template>

<script>
import { Link } from "@inertiajs/vue3";
import { router } from "@inertiajs/vue3";

import WebLayout from "@/Layouts/WebLayout.vue";
importInputLabel from "@/Components/InputLabel.vue";
import TextInput from "@/Components/TextInput.vue";

```

```
import PrimaryButton from "@/Components/PrimaryButton.vue";
import Pagination from "@/Shared/Pagination.vue";

export default {
    data() {
        return {};
    },
    components: {
        WebLayout,
        Link,
        Pagination,
        PrimaryButton,
        InputLabel,
        InputLabel,
    },
    props: {
        posts: Object,
        categories: Object,
        prop_type: "",
        prop_category_id: "",
        prop_search: "",
        prop_from: "",
        prop_to: "",
    },
    data() {
        return {
            type: this.prop_type,
            category_id: this.prop_category_id,
            search: this.prop_search,
            from: this.prop_from,
            to: this.prop_to,
        }
    },
    methods: {
        customSearch() {
            router.get(
                route("web.index", {
                    category_id: this.category_id,
                    type: this.type,
                    search: this.search,
                    from: this.from,
                    to: this.to,
                })
            );
        },
        cleanSearch() {

```

```
        router.get(route("web.index"));
    },
},
};

</script>
```

Es importante notar la condición de la imagen:

```

```

Además del CSS adicional para el botón:

resources/css/app.css

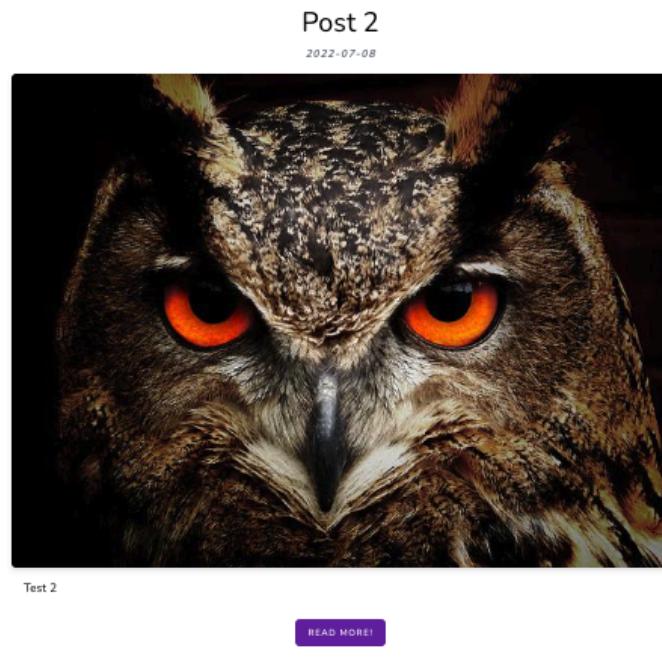
```
.btn-primary{
  @apply inline-flex items-center px-4 py-2 bg-purple-800 border border-transparent
  rounded-md font-semibold text-xs text-white uppercase tracking-widest hover:bg-gray-700
  active:bg-purple-900 focus:outline-none focus:border-gray-900 focus:ring
  focus:ring-gray-300 disabled:opacity-25 transition
}
```

Ya que, la misma es opcional y si no está definida, es cargada una imagen establecida por defecto en la aplicación.

Con esto, tendremos:

<http://localhost/blog>

| | |
|---|---|
| Date From | Date To |
| <input type="text" value="dd/mm/aaaa"/> | <input type="text" value="dd/mm/aaaa"/> |
| Search by id, title or description | |
| Type | |
| Category | FILTER |
| <input type="button" value="CLEAN"/> | |



Post 3

2022-07-08

Figura 13-1: Listado

Detalle del Post

Para el componente de detalle, obtenemos el detalle del post en base al slug:

app/Http/Controllers/Blog/PostController.php

```
public function show(Post $post) {
    $post->category;
    return inertia("Blog/Show", compact('post'));
}
```

Que luego mostramos en el componente de Vue:

resources/js/Pages/Blog/Show.vue

```
<template>
```

```

<web-layout>
  <div class="container">
    <div class="card">
      <div class="card-body">
        

        <h4 class="text-center text-5xl mb-3">{{ post.title }}</h4>

        <p class="my-4 ml-2">
          <span
            class="
              text-sm text-gray-500
              italic
              font-bold
              uppercase
              tracking-widest
            "
            >{{ post.date }}</span>
          <span class="ml-4 rounded-md bg-purple-500 py-1 px-2 text-gray-800">{{ post.category.title }}</span>
          <span class="ml-4 rounded-md bg-purple-500 py-1 px-2 text-gray-800">{{ post.type }}</span>
        </p>

        <div v-html="post.text"></div>

        <hr class="my-8" />
      </div>
    </div>
  </web-layout>
</template>

<script>
import WebLayout from "@/Layouts/WebLayout.vue";

```

```
export default {
  data() {
    return {};
  },
  components: {
    WebLayout,
  },
  props: {
    post: Object,
  },
};
</script>
```

Obteniendo como resultado:



Figura 13-2: Detalle

¿Aregar paso por paso a la vista de detalle?

Para hacer el experimento más interesante y comparar esta tecnología con Livewire, en el cual, realizamos el mismo tipo de integración, vamos a intentar realizar la integración del paso por paso en la vista de detalle de la publicación; para ello:

```
***  

    <hr class="my-8" />  

</div>  

<GeneralStep :errors="errors"/>  

</div>  

***  

<script>  

import WebLayout from "@/Layouts/WebLayout.vue";  

import GeneralStep from "@/Pages/Contact/General/Step.vue";  

export default {  

  data() {  

    return {};  

  },  

  components: {  

    WebLayout,  

    GeneralStep  

  },  

  props: {  

    post: Object,  

    errors: Object,  

  },  

};  

</script>
```

Con esto, tendríamos el paso por paso junto con la vista de detalle; el problema ocurre al procesar y dar un submit que la página es enviada al segundo paso; por ejemplo:

<http://localhost/contact/contact-general/X/edit>

El problema que se quiere evidenciar e que, en el caso de Livewire, los componentes tienen una vinculación más débil con la aplicación como un todo, dando la posibilidad de incluirlos en cualquier parte de una manera sencilla, además de que, en Livewire, usamos vistas de blade en vez de componentes de Vue, con lo cual, podemos usar todo el poderío de manera directa de Laravel; en Inertia no tenemos ninguna de esas posibilidades, por lo tanto, como presentamos al momento de hacer el paso por paso, su integración no es tan limpia, al depender los componentes de Vue, que usamos como vistas, de los controladores en Laravel para poder existir; en esta oportunidad, al estar incluyendo dos capas (el paso por paso como tal y sus hijos) es imposible tener una integración limpia, y en caso de que queramos adaptarlo completamente, tendríamos que crear una ruta adicional a la cual regresar desde el paso por paso, ruta, que es necesaria para tener el ID del contacto general así como el resto de los datos que queramos presentar, en nuestro caso, la publicación.

Es importante saber estos aspectos, ya que, son los que tienes que tener en cuenta al momento de decidir si vas a trabajar con Livewire o con Inertia.

Puedes consultar el código fuente en:

<https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.10>

Capítulo 14: Carrito de compras

En este capítulo, vamos a construir un carrito de compras, para explorar un poco más el uso de componentes en Vue junto con Laravel, conocer otros posibles usos y en general evidenciar su funcionamiento.

El shopping cart implementará las siguientes funcionalidades:

- Opción de agregar post de tipo publicidad (advert) al carrito.
- Variar las cantidades desde un listado.
- Indicar desde el post si la publicación está incluida en el carrito y en qué cantidades.
- Eliminar una publicación del carrito.

Esquema general

Al ser el uso del carrito de compras de alcance global en la aplicación, en la cual varias pantallas o páginas van a necesitar acceder a los datos o productos del carrito de compras, por ejemplo, una página con la cantidad de elementos en el carrito, mostrar el ícono en toda la web de la cantidad de elementos en el carrito, otra función anexa a la página de detalle de la publicación para ver si el producto está o no en el carrito, entre otras; vamo a necesitar manejar estos datos de manera global y compartida, para eso, usaremos la opción compartir datos de Inertia, tal cual usamos anteriormente para los mensajes flash:

app/Http/Middleware/HandleInertiaRequests.php

```
public function share(Request $request): array
{
    return array_merge(parent::share($request), [
        'flash' => [
            'message' => $request->session()->get('message')
        ],
        'cart' => [
            Post::find(10),
            Post::find(11),
            Post::find(12),
        ],
    ]);
}
```

De momento, serán unos datos de post por defecto, que luego variarán con el típico CRUD asociado a un controlador.

Vamos a crear un nuevo componente en Vue para mostrar la cantidad de elementos:

resources/js/Fragment/Cart.vue

```
<template>
<div class="mb-3" v-for="c in $page.props.cart" :key="c">
    <input type="number" class="w-20" />
    {{ c.title }}
</div>
</template>
```

Como puedes ver, en la misma no se incluye en layout, solamente el HTML del carrito y listo, por lo tanto, vamos a incrustar este componente en otros componentes, en este caso, la de detalle de una publicación de tipo publicidad:

resources/js/Pages/Blog>Show.vue

```
<div v-html="post.text"></div>

<div v-if="post.type == 'advert'">
  <cart />
</div>

<hr class="my-8" />
***

<script>
/***
import Cart from "@/Fragment/Cart.vue";

export default {
  components: {
    WebLayout,
    GeneralStep,
    Cart,
  }
}
</script>
```

Si nos vamos a una publicación de tipo publicidad, veremos:

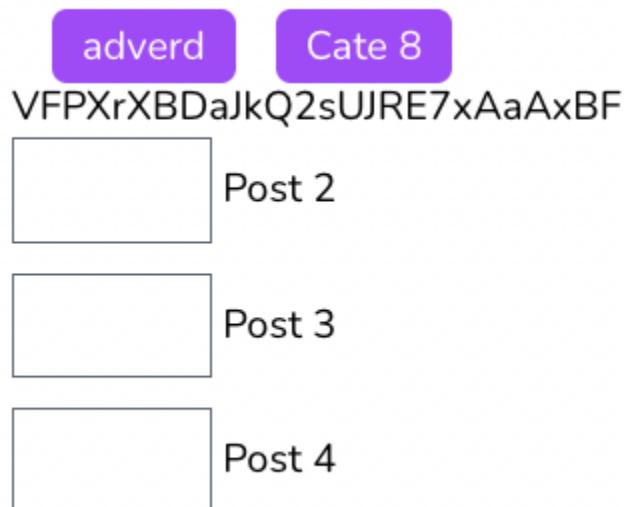


Figura 14-1: Listado de productos

Administrar ítem del carrito

En este apartado se explica el proceso que vamos a seguir para poder administrar ítems de un carrito de compras, recordemos que para los ítems del carrito de compras es necesario que se manejen de alcance global a la aplicación y que no sean usados de manera local que es el enfoque típico entre un controlador y su vista o componente en Vue, por lo tanto, vamos a adaptar el flujo básico que presentamos anteriormente para poder lograr el propósito de usar datos compartidos desde cualquier lugar de la aplicación.

Controlador

Este controlador, va a servir para realizar la gestión de nuestro carrito, por lo tanto, tendremos los métodos necesarios para manipular el mismo como serían la opción de eliminar y agregar items al carrito:

```
$ php artisan make:controller Shop/CartController
```

Y, vamos a definirlo de la siguiente manera:

app/Http/Controllers/Shop/CartController.php

```
<?php

namespace App\Http\Controllers\Shop;

use App\Http\Controllers\Controller;
use App\Models\Post;
use Illuminate\Http\Request;
use Illuminate\Support\Arr;

class CartController extends Controller
{
    public function add(Post $post, $count = 1)
    {

        $cart = session('cart', []);

        // [
        //     18 =>[$post, 5],
        //     19 =>[$post2, 8],
        // ]
        // [
        //     [post, 5],
        //     [post2, 8],
        // ]

        // delete
        if ($count <= 0) {
            if (Arr::exists($cart, $post->id)) {
```

```

        unset($cart[$post->id]);
        session(['cart' => $cart]);
    }
    return redirect()->back();
}

// add
if (Arr::exists($cart, $post->id)) {
    $cart[$post->id][1] = $count;
} else {
    $cart[$post->id] = [$post, $count];
}

// set en la sesion
session(['cart' => $cart]);
return redirect()->back();
// dd($cart);

}
}

```

Explicación del código anterior

El código del método **add()**, la empleamos para agregar un post por vez (puedes fijarte en la firma del método en la cual recibimos el post y las cantidades), luego, establecemos el post en la sesión junto con las cantidades.

El método de **add()** también la usamos para realizar el proceso de eliminación de los ítems en el carrito; cuando se pasa el valor de cero o un número negativo, se elimina del carrito.

Como puedes ver, es ahora en este componente en el cual hacemos la implementación de la sesión en caso de que la misma no exista; aparte de que, se tiene implementado la opción de agregar ítems en el carrito, hacer las verificaciones, eliminar el ítem, y poco más.

Es importante notar que, se establece en el índice del **array** la PK del post, con esto, sabemos de manera directa si el post/item está agregado en el carrito y actuar en consecuencia, es decir, agregar el post si no existe:

```
|$cart[$post->id] = [$post, $count];
```

Modificar la cantidad:

```
|$cart[$post->id][1] = $count;
```

Eliminar el post:

```
|unset($cart[$post->id]);
```

Ruta

Creamos la ruta para el método anterior:

routes/web.php

```
Route::group([
    'prefix' => 'shop',
], function () {
    Route::post('/add/{post}/{count}', [App\Http\Controllers\Shop\CartController::class,
'add'])->name('shop.add');
});
```

Compartir datos del carrito mediante la sesión a Vue

Para poder usar este controlador desde un componente en Vue, vamos a usar el **HandleInertiaRequests**:

app/Http/Middleware/HandleInertiaRequests.php

```
public function share(Request $request): array
{
    return array_merge(parent::share($request), [
        'flash' => [
            'message' => $request->session()->get('message')
        ],
        'cart' =>
            session('cart', [])
    ]);
}
```

Esto es crucial ya que, el controlador no lo vamos a conectar de manera directa con el componente de Vue, si no, de manera indirecta mediante el **HandleInertiaRequests**; en pocas palabras, desde el carrito de compras, vamos a enviar peticiones al controlador del carrito, que modificará la sesión que luego es usada mediante el **HandleInertiaRequests** para poder consumir estos datos en cualquier parte.

Componente de Vue para administrar un Item del carrito

Este componente es el que se usará para mostrar la opción de agregar un ítem al carrito si no existe el mismo o poder eliminar o cambiar las cantidades del mismo si el ítem existe ya en el carrito:

resources/js/Fragment/CartItem.vue

```
<template>
    <TextInput v-model="count" type="number" class="w-20" min="1" />
    <PrimaryButton type="submit" @click="submit"> Send </PrimaryButton>
</template>

<script>
```

```

import { router } from "@inertiajs/vue3";
import TextInput from "@/Components/TextInput.vue";
import PrimaryButton from "@/Components/PrimaryButton.vue";
export default {
    components: {
        TextInput,
        PrimaryButton,
    },
    data() {
        return {
            count: "1",
        };
    },
    props: {
        post: {
            type: Object,
            required: true,
        },
    },
    methods: {
        submit() {
            router.post(
                route("shop.add", {
                    post: this.post.id,
                    count: this.count,
                })
            );
        },
    },
},
</script>

```

Desde este componente, se realizará la gestión del ítem, por lo tanto, necesitamos tanto el post, como la cantidad (la cantidad es usada mediante un **v-model**), cuyos parámetros son enviados al controlador para poder administrar el mismo.

Y lo usamos, desde la vista de detalle del post, la cual, es el lugar ideal para poder pasar el **prop** de post que es requerido para poder agregar el ítem al carrito:

resources\js\Pages\Blog>Show.vue

```

<div v-if="post.type == 'advert'">
    <Cart/>
    <CartItem :post="post" />
</div>
<script>

```

```

import WebLayout from "@/Layouts/WebLayout.vue";
import GeneralStep from "@/Pages/Contact/General/Step.vue";

import CartItem from "@/Fragment/CartItem.vue";
import Cart from "@/Fragment/Cart.vue";

export default {
  data() {
    return {};
  },
  components: {
    WebLayout,
    GeneralStep,
    CartItem,
    Cart,
  },
},

```

Listado de Post en el carrito

Al tener un listado de ítems (post) en el carrito que es diferente al que teníamos en la fase de pruebas, debemos de variar la implementación que se tenía inicialmente y preguntar por las posiciones:

1. La posición cero, si queremos acceder al post.
2. La posición uno, si queremos acceder a la cantidad.

resources/js/Pages/Shop/Cart.vue

```

<template>
<!-- {{$page.props.cart}} -->
<div class="mb-3" v-for="c in $page.props.cart" :key="c">
  <input type="number" class="w-20" :value="c[1]" />
  {{ c[0].title }}
</div>
</template>

```

Vamos a aprovechar que, el ítem que vamos a colocar en el detalle de post para agregar el post al carrito y que el detalle que estamos construyendo en el componente de **Shop/Cart.vue** son prácticamente los mismos, para fusionarlos:

resources/js/Fragment/CartItem.vue

```

<template>
  <TextInput v-model="count" type="number"/>
  <PrimaryButton @click="submit">Send</PrimaryButton>
</template>
<script>

```

```

import { router } from "@inertiajs/vue3"

import PrimaryButton from '@/Components/PrimaryButton.vue';
import TextInput from '@/Components/TextInput.vue';

import { toRaw } from 'vue'

export default {
    components: {
        PrimaryButton,
        TextInput
    },
    mounted() {
        // console.log(toRaw(this.$page.props.cart))
        // console.log(Object.keys(this.$page.props.cart))
        let n = 0
        Object.keys(this.$page.props.cart).forEach((k) => {
            // console.log(this.$page.props.cart[k][1])
            n += this.$page.props.cart[k][1]
        })
        console.log(n)
    },
    data() {
        return {
            count: this.pcount,
        }
    },
    props: {
        post: {
            required: true,
            type: Object
        },
        pcount: {
            type: String,
            default: "1",
        },
    },
    methods: {
        submit() {
            router.post(route("shop.add"), {
                post: this.post.id,
                count: this.count
            )))
        }
    }
}

```

```
}
```

```
</script>
```

Es importante notar que, creamos un prop para las cantidades:

1. La cantidad es pasada si existe en la sesión del carrito
2. La cantidad no es pasada y toma el valor de 1 si el producto no existe en la sesión.

En cualquier caso, si el producto existe o no en el carrito, se invoca al mismo controlador según la implementación que creamos:

```
router.post(
  route("shop.add", {
    post: this.post.id,
    count: this.count,
  })
);
```

En el carrito, usamos el componente anterior:

resources/js/Fragment/Cart.vue

```
<template>
  <!-- {{ $page.props.cart }} -->
  <div class="mb-3" v-for="c in $page.props.cart" :key="c">
    <CartItem :post="c[0]" :pcount="c[1].toString()" />
  </div>
</template>

<script>
import CartItem from "@/Fragment/CartItem.vue";
export default {
  components: {
    CartItem,
  },
};
</script>
```

Desde el detalle del post, puedes probar la implementación:

resources/js/Pages/Blog>Show.vue

```
<div v-if="post.type == 'advert'">
  <cart-item :post="post"/>
  <cart />
</div>
```

Vista de detalle del carrito

Vamos a mostrar el detalle del carrito aparte en una vista aparte; es decir, el componente ya existente llamado como **Fragment/Cart.vue**, vamos a usarlo para mostrar el mismo en una página aparte; para eso, creamos el componente en Vue:

resources/js/Pages/Shop/Index.vue

```
<template>
<web-layout>
  <div class="card">
    <div class="card-body">
      <h3>Carrito de compras</h3>
      <cart />
    </div>
  </div>
</web-layout>
</template>

<script>
import WebLayout from "@/Layouts/WebLayout.vue";
import Cart from "@/Fragment/Cart.vue";

export default {
  components: {
    WebLayout,
    Cart,
  },
};
</script>
```

Y el controlador:

```
class CartController extends Controller
{
    public function index()
    {
        return inertia('Shop/Index');
    }
}/**/
}
```

Y la ruta:

```
Route::group([
    'prefix' => 'shop',
```

```
], function () {
    Route::get('/', [App\Http\Controllers\Shop\CartController::class,
'index'])->name('shop.index');
    Route::post('/add/{post}/{count}', [App\Http\Controllers\Shop\CartController::class,
'add'])->name('shop.add');
});
```

Ya con esto, tenemos acceso al detalle del carrito en exclusiva en una página aparte:

<http://localhost/shop>

<http://inertiastore.test/shop>

Replicar el carrito de compra en la base de datos

En el apartado anterior logramos la construcción de un carrito de compras y almacenar el mismo en la sesión; lógicamente, la implementación del carrito de compras incluye la gestión del mismo, para remover, agregar y cambiar cantidades; pero, de momento todo esto está vinculado a la sesión.

En este apartado, se va a realizar la implementación del carrito de compras en la base de datos; por lo tanto, el esquema que se tiene en la sesión, se va a replicar a la base de datos, la cual se aprovechará cuando el usuario inicie sesión y se puede replicar nuevamente a la sesión, para mantener siempre los datos del carrito de compras en un entorno persistente.

Crear migración y tabla

Para poder replicar el carrito de compras de la sesión a la base de datos, lógicamente es necesario una tabla, y para generar la tabla, se emplean las migraciones; vamos a generar una migración para el carrito:

```
$ php artisan make:migration create_shopping_carts_table
```

La cual tendrá la siguiente estructura:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     *
     * @return void
     */
    public function up(): void
```

```

{
    Schema::create('shopping_carts', function (Blueprint $table) {
        $table->id();
        $table->foreignId('user_id');
        $table->foreignId('post_id');
        $table->integer('count')->unsigned();
        $table->integer('control')->unsigned();
        $table->timestamps();
    });
}

/**
 * Reverse the migrations.
 *
 * @return void
 */
public function down(): void
{
    Schema::dropIfExists('shopping_carts');
}
};

```

Los campos definidos anteriormente son para:

- user_id, El usuario dueño del carrito de compras.
- post_id, El post (ítem).
- count, Cantidades.
- control, Un campo de control que usaremos para saber qué ítems del carrito se van actualizando y cuyo propósito explicaremos un poco más adelante.

Ejecutamos la migración con:

```
$ php artisan migrate
```

Y con esto, ya tenemos la tabla en la base de datos.

Modelo

Para el modelo del carrito de compras, va a lucir de la siguiente manera, en la cual implementamos también los métodos relacionales:

```
<?php

namespace App\Models;

use Illuminate\Database\Eloquent\Factories\HasFactory;
use Illuminate\Database\Eloquent\Model;
```

```

class ShoppingCart extends Model
{
    use HasFactory;

    protected $fillable = ['user_id', 'post_id', 'count', 'control'];

    public function post()
    {
        return $this->belongsTo(Post::class);
    }

    public function user()
    {
        return $this->belongsTo(Post::class);
    }
}

```

Registrar cambios en la base de datos

A nivel del controlador del carrito de compras, implementamos un nuevo método para replicar los cambios en la base de datos, y el llamado del mismo en el método de **add()**:

app/Http/Controllers/Shop/CartController.php

```

<?php

namespace App\Http\Controllers\Shop;

use App\Http\Controllers\Controller;
use App\Models\Post;
use App\Models\ShoppingCart;
use Illuminate\Http\Request;
use Illuminate\Support\Arr;

class CartController extends Controller
{
    public function add(Post $post, $count = 1)
    {

        $cart = session('cart', []);

        // delete
        if ($count <= 0) {
            if (Arr::exists($cart, $post->id)) {
                unset($cart[$post->id]);
                session(['cart' => $cart]);
                $this->saveDB($cart);
            }
        }
    }
}

```

```

        }

        return redirect()->back();
    }

    // add
    if (Arr::exists($cart, $post->id)) {
        $cart[$post->id][1] = $count;
    } else {
        $cart[$post->id] = [$post, $count];
    }

    // set en la sesion
    session(['cart' => $cart]);

    $this->saveDB($cart);
    return redirect()->back();
    // dd($cart);

}

public function saveDB($cart)
{
    if (auth()->check()) {
        $control = time();

        foreach ($cart as $c) {

            ShoppingCart::updateOrCreate(
                [
                    'post_id' => $c[0]->id,
                    'user_id' => auth()->id(),
                ],
                [
                    'post_id' => $c[0]->id,
                    'count' => $c[1],
                    'user_id' => auth()->id(),
                    'control' => $control,
                ]
            );
        }
    }

    //*** BORRAR */
}

```

```
    ShoppingCart::whereNot('control', $control)->where('user_id',
auth()->id())->delete();
}
}
```

Explicación del código anterior

El método de **saveDB()**, recibe el objeto de carrito que manipulamos desde la sesión; con este objeto, se replican los cambios en la base de datos.

Primero, se realizan las actualizaciones o inserciones de los ítems en la base de datos, es decir, si para la condición de:

```
'post_id' => $c[0]['id'],
'user_id' => auth()->id(),
```

Existe un registro, entonces actualiza, caso contrario, inserta el registro.

El campo de control se utiliza para saber qué registros se están actualizando o insertando, se usa una clave única, que en este caso es el tiempo (**time()**).

El último paso, es para eliminar los ítems que no se actualizaron anteriormente, y para esto es usado el campo de control.

Todo este proceso se realiza si y sólo si el usuario inició sesión; lo que quiere decir, es que, si el usuario no inicia sesión, entonces el proceso de sincronización en la base de datos no es utilizado.

Replicar carrito de la base de datos a sesión al momento del login

El objetivo de hacer persistente el carrito de compras mediante la base de datos, es que, al momento del login, podamos obtener el mismo; para ello, al trabajar con un login generado por defecto por Laravel Inertia mediante Fortify, el cual no podemos manipular directamente al formar parte del Inertia, vamos a usar los eventos.

Como sucede con varias operaciones que se realizan en el framework, como el registro de usuarios, verificación de usuarios y el mismo login, estos generan eventos que se pueden escuchar mediante un listener de Laravel, en este caso, el evento de Login que es emitido mediante Fortify internamente; creamos el listener con:

```
$ php artisan make:listener LoginSuccessful --event=Illuminate\Auth\Events\Login
```

La opción de **event** se utiliza para indicar el evento que quieras escuchar; en este caso, el evento de login; esto lo emplea Laravel internamente para definir la estructura básica, ya el evento de autoregistra ya que es monitoreado internamente por laravel, es decir, no hace falta registrarlo.

Ahora, lo único que falta es poder hacer la réplica; para esto, vamos a tomar el método de **add()** definida para el componente de ítem del carrito, y lo usamos desde el listener.

El listener, busca los ítems del carrito en la base de datos, los itera uno a uno y va armando la sesión para poder usarla.

Ya con esto, si todo está correctamente configurado, al iniciar sesión, deberías de tener en la sesión, los ítems del carrito de compras (claro está, si el usuario tenía ítems agregado en dicha sesión y replicados en la base de datos).

app/Listeners/LoginSuccessful.php

```
<?php

namespace App\Listeners;

use App\Models\Post;
use App\Models\ShoppingCart;
use Illuminate\Auth\Events\Login;
use Illuminate\Contracts\Queue\ShouldQueue;
use Illuminate\Queue\InteractsWithQueue;
use Illuminate\Support\Arr;

class LoginSuccessful
{
    public function __construct()
    {
        //
    }

    public function handle(Login $event): void
    {
        $cart = ShoppingCart::where('user_id', auth()->id())->get();
        foreach($cart as $c){
            $this->add($c->post,$c->count);
        }
    }

    public function add(Post $post, $count = 1)
    {
        $cart = session('cart', []);
        // add
        if (Arr::exists($cart, $post->id)) {
            $cart[$post->id][1] = $count;
        } else {
            $cart[$post->id] = [$post, $count];
        }
    }
}
```

```

    // set en la sesion
    session(['cart' => $cart]);
}

}

```

Otra forma que tenemos disponible, es empleando un callback directamente en el **AppServiceProvider**:

app\Providers\AppServiceProvider.php

```

<?php

namespace App\Providers;

use App\Models\ShoppingCart;
use Illuminate\Auth\Events\Login;
use Illuminate\Support\Arr;
use Illuminate\Support\Facades\Event;

use Illuminate\Support\ServiceProvider;

class AppServiceProvider extends ServiceProvider
{
    public function boot(): void
    {
        Event::listen(function (Login $event) {
            $this->setShoppindCartSession();
        });
    }

    private function setShoppindCartSession()
    {
        $cartDB = ShoppingCart::where('user_id', auth()->id())->get();
        $cartSession = session('cart', []);

        foreach ($cartDB as $c) {
            if (Arr::exists($cartSession, $c->post->id)) {
                // update
                $cartSession[$c->post->id][1] = $c->count;
            } else {
                // add
                $cartSession[$c->post->id] = [$c->post, $c->count];
            }
        }

        session(['cart' => $cartSession]);
    }
}

```

```
}
```

Mostrar un mensaje toast

Mostrar un mensaje de confirmación cada vez que realicemos una acción sobre el carrito, es una medida que usualmente se realizan en este tipo aplicaciones; al tener Oruga UI instalado en el proyecto, ya contamos con un componente:

```
this.$oruga.notification.open({<OPCIONES>});
```

El cual, podemos personalizar a nivel de:

- Posición mediante la opción de **position**: top-right, top, top-left, bottom-right, bottom, bottom-left.
- Color, mediante la opción de **variant**.
- Duración en segundos o milisegundos mediante la opción de **duration**.
- Si es cerrable mediante la opción de **closable** en true.

Por nombrar algunas opciones; usaremos la siguiente:

resources/js/Fragment/CartItem.vue

```
submit() {
  router.post(
    route("shop.add", {
      post: this.post.id,
      count: this.count,
    })
  );
}

this.$oruga.notification.open({
  message: "Applied changes",
  position: "top-right",
  variant: "success",
  duration: 2000,
  closable: true,
});
},
```

Y con esto, tendremos el siguiente resultado:

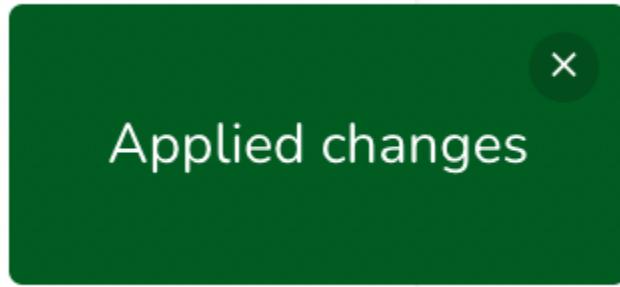


Figura 14-2: Toast de confirmación de ítem agregado

Este mensaje aparecerá ya sea que agreguemos un ítem, cambiemos la cantidad o eliminemos el ítem del carrito.

Botón para mostrar la cantidad total de ítems

Ahora, se va a configurar un botón el cual servirá tanto como enlace para ir al listado de ítems en el carrito, como para mostrar cuántos ítems tenemos en el carrito.

Esta es una operación que realizaremos desde otro componente en Vue, es decir, desde el cliente, usando la variable global de:

```
this.$page.props.cart
```

Para esto, lo primero que se va a realizar, es calcular el total de items que tenemos en el carrito, desde el cliente; pero, antes de esto, tenemos que conocer cuál es la estructura que tenemos en **this.\$page.props.cart**:

Veremos que es un objeto reactivo de Vue:

```
Proxy {15: Array(2), 27: Array(2)}
[[Handler]]: Object
[[Target]]: Object
[[IsRevoked]]: false
```

Vamos a usar la función de **toRaw()** para conocer exactamente qué es lo que tenemos:

```
/*import { toRaw } from "vue"
const cart = toRaw(this.$page.props.cart);
console.log(cart)
*/
{15: Array(2), 27: Array(2)}
15: (2) [...], '5']
27: (2) [...], '9']
[[Prototype]]: Object
```

Y veremos que tenemos es un objeto y no un **array**; al ser procesado como un objeto desde el cliente, no podemos usar un ciclo para iterar cada uno de estos elementos; necesitamos conocer cuál es la key de cada uno de las opciones de nuestro objeto, en el ejemplo anterior, las keys sería 15 y 27; para obtener las mismas, de manera programática:

```
const cart = this.$page.props.cart //toRaw()  
Object.keys(cart) // [15,27]
```

Es importante notar que, no hay necesidad de usar la función de **toRaw()**, podemos trabajar con el objeto reactivo sin problemas; con el método anterior, tenemos acceso a las keys del objeto que es devuelto en un **array, array** que podemos iterar:

```
const cart = this.$page.props.cart  
Object.keys(cart).forEach((k)=> {  
    // OP  
})
```

Esto es importante, ya que, calcularemos el total de ítems iterando el mismo y sumando las cantidades que podemos acceder mediante las keys; esto lo haremos en un nuevo componente, tomando como referencia el código explicado anteriormente:

resources/js/Fragment/CartCount.vue

```
<template>  
  {{ total }}  
</template>  
  
<script>  
  
//import { toRaw } from "vue"  
  
export default {  
  computed:{  
    total:function(){  
      const cart = this.$page.props.cart //toRaw()  
      let n=0  
      Object.keys(cart).forEach((k)=> {  
        n += parseInt(cart[k][1])  
      })  
      return n  
    }  
  }  
</script>
```

En el código anterior, usamos un método computado, pero puedes usar también un método normal de Vue; con el script anterior, deberías de ver el total de productos en el carrito de compras, sumando las cantidades.

Mostrar el total y enlace para el listado de ítems del carrito

Para mostrar el total calculado anteriormente, se realizará mediante un botón flotante en el cual se colocará tanto el enlace, como el total calculado anteriormente:

resources/js/Fragment/CartCount.vue

```
<template>
  <a :href="route('shop.index')" class="fixed bottom-0 right-0 m-5">
    <div
      class="
        bg-red-500
        rounded-full
        w-5
        h-5
        text-white text-sm text-center
        shadow
        absolute
        -top-2
        -right-1
      "
    >
      {{ total }}
    </div>

    <div
      class="
        p-2
        bg-purple-500
        rounded-full
        w-10
        h-10
        shadow
        border-purple-800 border-2
      "
    >
      <svg
        xmlns="http://www.w3.org/2000/svg"
        fill="#FFF"
        width="24"
        height="24"
        viewBox="0 0 24 24"
      >
        <path
```

```

        d="M10 19.5c0 .829-.672 1.5-1.5 1.5s-1.5-.671-1.5-1.5c0-.828.672-1.5
1.5-1.5s1.5.672 1.5 1.5zm3.5-1.5c-.828 0-1.5.671-1.5 1.5s.672 1.5 1.5 1.5 1.5-.671
1.5-1.5c0-.828-.672-1.5-1.5-1.5zm1.336-511.977-7h-16.813l2.938 7h11.898zm4.969-101-3.432
12h-12.597l1.839 2h13.239l3.474-12h1.929l1.743-2h-4.195z"
    />
</svg>
</div>
</a>
</template>

```

El HTML SVG es provisto de la siguiente página:

<https://iconmonstr.com/shopping-cart-3-svg/>

Este componente lo puedes incluir en cualquier otro componente en Vue, en nuestro caso, lo incluiremos en el layout de la aplicación:

resources/js/Layouts/WebLayout.vue

```

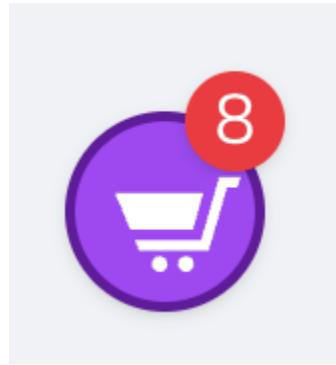
<template>
<header>
<div v-if="$slots.header" class="mx-auto px-3">
  <slot name="header" />
</div>
</header>

<div class="container">
  <slot />
  <cart-count/>
</div>
</template>

<script>
import CartCount from "@/Fragment/CartCount.vue"
export default {
  components:{
    CartCount
  }
}
</script>

```

Finalmente, se obtiene en las vistas del componente de carrito:



Con esto, se tiene un carrito de compras en la sesión y la base de datos completamente funcional, que puedes extender para incluir el precio y usarlo en compras reales.

Puedes consultar el código fuente en:

<https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v.10>

Capítulo 15: Opciones de los métodos HTTP de Inertia

Muchas veces cuando enviamos peticiones HTTP mediante el objeto de Inertia, tenemos que configurar varios aspectos sobre la misma; para ello, tenemos el último parámetro de dicho método que corresponde a las opciones, y esto aplica para cada uno de los métodos HTTP que podemos usar con Inertia:

```
this.$router.get(url, data, options)
this.$router.post(url, data, options)
this.$router.put(url, data, options)
this.$router.patch(url, data, options)
this.$router.delete(url, options)
```

Tenemos un apartado de opciones con el cual podemos personalizar la petición HTTP, opciones como persistir el estado actual, el scroll y métodos tipo callback que se ejecutan según el estado de la petición realizada; veamos algunos.

Historial del navegador

Al realizar peticiones, Inertia agrega automáticamente una nueva entrada en el historial del navegador. Sin embargo, también es posible remover la petición de la entrada del historial actual:

```
router.get('/post', <data>, { replace: true })
```

Estado del componente

De forma predeterminada, las visitas a la misma página fuerzan una instancia de componente de página nueva, que borra cualquier estado local, como entradas de formulario, posiciones de desplazamiento y estados de enfoque.

En ciertas situaciones, es necesario preservar el estado del componente de la página. Por ejemplo, al enviar un formulario, debe conservar los datos del formulario en caso de que vuelvan a aparecer errores de validación.

```
router.get('/post', <data>, { preserveState: true })
```

Preservar el scroll

Al navegar entre páginas, Inertia imita el comportamiento predeterminado del navegador restableciendo automáticamente la posición de desplazamiento del cuerpo del documento (es decir, al enviar una petición, el scroll se restablece al inicio), de vuelta a la parte superior; para evitar esto, se une la opción `preserveScroll`:

```
router.put('/update', <data>, { preserveScroll: true });
```

Headers personalizados

La opción de headers permite agregar headers personalizados; por ejemplo:

```
router.post('/post', <data>, {
  headers: {
    'Custom-Header': 'value',
  },
})
```

Callbacks personalizados

En este apartado, vamos a hablar sobre los métodos tipo callback que se invocan según el estado de la petición, por ejemplo:

- **onBefore**, cuando se inicia la petición.
- **onProgress**, cuando la petición está en proceso.
- **onSuccess**, cuando existe una respuesta.
- **onError**, cuando ocurre algún problema.

Específicamente, tenemos:

```
router.<method>('/post', <data>, {
  onBefore: visit => {},
  onStart: visit => {},
  onProgress: progress => {},
  onSuccess: page => {},
  onError: errors => {},
  onFinish: visit => {},
})
```

Puedes ver la referencia completa en:

<https://inertiajs.com/manual-visits>

Capítulo 16: Aplicación de to do list

En este capítulo, vamos a construir una aplicación tipo to do list, en las cuales, tendremos un listado de tareas asociadas a usuarios con las típicas operaciones tipo CRUD, cambio de estado y reordenación.

Crear componente en Vue y controlador

Comencemos creando los elementos necesarios para construir cualquier aplicación, que serían su controlador y vista (componente en Vue) asociado.

Comencemos creando un componente en Vue para manejar toda la aplicación de to do:

resources/js/Pages/Todo/Index.vue

Crearemos la estructura típica importante un layout que hemos usado en anteriores aplicaciones junto con la estructura de cartas:

```
<template>
  <web-layout>
    <div class="card">
      <div class="card-body">
        <h3>App To do</h3>
      </div>
    </div>
  </web-layout>
</template>
<script>
import WebLayout from "@/Layouts/WebLayout.vue";

export default {
  components: {
    WebLayout,
  },
};
</script>
```

Ya con esto, vamos a crear un listado sencillo en base a UL en la cual, cada LI corresponde a una tarea y un formulario para crear los to do:

```
<template>
  <web-layout>
    <div class="card">
      <div class="card-body">
        <h3>App To Do</h3>

        <form @submit.prevent="create" class="flex gap-2 mt-2">
```

```

        <TextInput v-model="form.name" placeholder="Create To Do"></TextInput>
        <PrimaryButton>Send</PrimaryButton>
    </form>

    <ul>
        <li class="border py-3 px-4 mt-2">
            <span>To do 1</span>
            <button class="float-right" @click="remove">
                <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24"
stroke-width="1.5"
stroke="currentColor" class="h-6 w-6" fill="#F00">
                    <path stroke-linecap="round" stroke-linejoin="round"
d="m14.74 9-.346 9m-4.788 0L9.26
9m9.968-3.21c.342.052.682.107 1.022.166m-1.022-.165L18.16 19.673a2.25 2.25 0 0 1-2.244
2.077H8.084a2.25 2.25 0 0 1-2.244-2.077L4.772 5.79m14.456 0a48.108 48.108 0 0
0-3.478-.397m-12 .562c.34-.059.68-.114 1.022-.165m0 0a48.11 48.11 0 0 1 3.478-.397m7.5
0v-.916c0-1.18-.91-2.164-2.09-2.201a51.964 51.964 0 0 0-3.32 0c-1.18.037-2.09 1.022-2.09
2.201v.916m7.5 0a48.667 48.667 0 0 0-7.5 0" />
                </svg>
            </button>
        </li>
    </ul>

    </div>
</div>

</web-layout>
</template>
<script>
import PrimaryButton from '@/Components/PrimaryButton.vue';
import TextInput from '@/Components/TextInput.vue';
import WebLayout from '@/Layouts/WebLayout.vue';

export default {
    components: {
        WebLayout,
        PrimaryButton,
        TextInput
    },
    data() {
        return {
            form: {
                name: ''

```

```

        }
    }
},
methods: {
    create(){
        console.log('create')
    },
    remove(){
        console.log('remove')
    }
},
}
</script>

```

Creamos el controlador para devolver el componente anterior:

app/Http/Controllers/TodoController.php

```

<?php

namespace App\Http\Controllers;

use Illuminate\Http\Request;

class TodoController extends Controller
{
    public function index()
    {
        return inertia('Todo/Index');
    }
}

```

Y su ruta asociada:

routes/web.php

```

Route::middleware(
    [
        'auth:sanctum',
        config('jetstream.auth_session'),
        'verified',
    ]
)->prefix('todo')->group(function () {
    Route::get('/', [App\Http\Controllers\TodoController::class,
    'index'])->name('todo.index');
});

```

Y tendremos:

App To do

Create To Do

CREATE

To do 1



Figura 16-1: Item de ejemplo de To Do

El icono de bote de basura fue tomado de la web de <https://heroicons.com/>

Crear migración y modelo para los Todos

Para poder registrar los todos de manera persisten, vamos a crear la migración y el modelo:

```
$ php artisan make:migration create.todos_table
```

Y lucirá como:

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    public function up(): void
    {
        Schema::create('todos', function (Blueprint $table) {
            $table->id();
            $table->foreignId('user_id');
            $table->string('name', 255);
            $table->boolean('status')->default(0);
            $table->integer('count')->unsigned();
            $table->timestamps();
        });
    }

    public function down(): void
    {
    }
}
```

```
{  
    Schema::dropIfExists('todos');  
}  
};
```

Explicación de los campos anteriores

- **name**: Campo para registrar el to do.
- **user_id**: Campo para registrar el usuario dueño del to do.
- **count**: Campo para indicar el orden de los to do.
- **status**: Campo para manejar si la tarea fue completada (1) o no (0) por defecto, la tarea es marcada como no completada.

El único campo que va a usar de manera directa el usuario, es el de **name**.

Y ejecutamos la migración:

```
$ php artisan migrate
```

Creamos el modelo:

```
$ php artisan make:model Todo
```

El modelo lucirá:

app/Models/Todo.php

```
<?php  
  
namespace App\Models;  
  
use Illuminate\Database\Eloquent\Factories\HasFactory;  
use Illuminate\Database\Eloquent\Model;  
  
class Todo extends Model  
{  
    use HasFactory;  
  
    public $fillable = ['user_id', 'name', 'count', 'status'];  
}
```

Crear un To Do

Vamos a crear un controlador para crear un to do, en el mismo, colocaremos también la validación para el nombre del to do y regresar a la página anterior:

app/Http/Controllers/TodoController.php

```

public function store(Request $request)
{
    $data = $request->validate(
        [
            'name' => "required|min:2|max:255",
        ]
    );
    Todo::create([
        'name' => $data['name'],
        'user_id' => auth()->id(),
        'count' => Todo::where('user_id', auth()->id())->count()
    ]);

    return back();
}

```

Creamos la ruta:

routes/web.php

```

Route::middleware(
    [
        'auth:sanctum',
        config('jetstream.auth_session'),
        'verified',
    ]
)->prefix('todo')->group(function () {
    Route::get('/', [App\Http\Controllers\TodoController::class,
    'index'])->name('todo.index');
    Route::post('store', [App\Http\Controllers\TodoController::class,
    'store'])->name('todo.store');
});

```

Y desde el componente en Vue, realizamos la petición:

```

import { router } from "@inertiajs/vue3";
*** 
create() {
    router.post(
        route("todo.store", {
            name: this.form.name,
        })
    );
}

```

```
| },
```

Para mostrar los errores de validación:

```
<form @submit.prevent="create" class="flex gap-2 mt-2">
  ***
</form>

<InputError :message="errors.name" />
  ***
import InputError from '@/Components/InputError.vue';
  ***
props: {
  errors: Object,
},
```

E inclusive, de manera opcional, vamos a cambiar la clase de campo de texto si tenemos errores:

```
<TextInput :class="{ 'text-red-400 bg-red-200': errors.name }" placeholder="Create To Do"
v-model="form.name" />
```

Y tendremos:



Figura 16-2: Error de validación

Popular listado

Para el listado, se va a crear una consulta a la base de datos de todos los to do del usuario y ordenados por el campo "count":

```
app/Http/Controllers/TodoController.php
```

```
public function index()
{
    $todos = Todo::where('user_id', auth()->id())->orderBy('count')->get();
    return inertia('Todo/Index', compact('todos'));
}
```

Con el **array** de todos los to do del usuario, los iteramos y construimos el listado:

```

<li v-for="t in todos" class="border py-3 px-4 mt-2" :key="t">
  <span>
    {{ t.name }}
  </span>
  <button @click="remove" class="float-right">
    ***
  </button>
</li>
***

props: {
  todos: Object,
  errors: Object,
},

```

Y tendremos un listado de los to do provisto por la base de datos.

Editar un To Do

Para la opción de edición de un to do, vamos a hacerlo desde el mismo listado sin usar un componente de modal; lo que realizaremos es que, cuando demos un click sobre el SPAN del nombre del to do, colocaremos un INPUT en su lugar con el nombre del to do que vamos a editar:

resources/js/Pages/Todo/Index.vue

```

<li v-for="t in todos" class="border py-3 px-4 mt-2" :key="t">
  <span v-show="!t.editMode" @click="t.editMode = true">
    {{ t.name }}
  </span>
  <TextInput v-show="t.editMode == true" v-model="t.name" @keyup.enter="update(t)" />
  <button @click="remove" class="float-right">
    ***
  </button>
</li>
***

update(todo) {
  console.log(todo.name);
  todo.editMode = false;
},

```

Como puedes ver en el código anterior, para eso, se emplea una opción que no viene en el listado por defecto, llamada "editMode", y dependiendo del valor de la misma:

- **true**, entramos en modo de edición y por lo tanto, ocultamos el SPAN y mostramos el campo de texto.
- **false**, entramos en modo de lectura y por lo tanto, mostramos el SPAN y ocultamos el campo de texto.

Gracias a la reactividad de Vue, al cambiar cualquier opción del to do, como el nombre o el "editMode" automáticamente los cambios se verán reflejados en el template, es importante notar que, estamos creando **v-models** a partir del **array** de todos los to do.

El método de update del to do se llama al detectar la tecla de enter: `@keyup.enter="update(t)"`.

El proceso de actualización del to do:

app/Http/Controllers/TodoController.php

```
public function update(Todo $todo, Request $request)
{
    Todo::where("id", $todo['id'])->where('user_id', auth()->id())->update(['name' =>
$data['name']]);
    return back();
}
```

Creamos la ruta:

routes/web.php

```
Route::middleware(
[
    'auth:sanctum',
    config('jetstream.auth_session'),
    'verified',
])
->prefix('todo')->group(function () {
    Route::get('/', [App\Http\Controllers\TodoController::class,
'index'])->name('todo.index');
    Route::post('store', [App\Http\Controllers\TodoController::class,
'store'])->name('todo.store');
    Route::put('update/{todo}', [App\Http\Controllers\TodoController::class,
'update'])->name('todo.update');
});
```

Finalmente, creamos el proceso para actualizar el to do desde el cliente:

resources/js/Pages/Todo/Index.vue

```
update(todo) {
    todo.editMode = false;

    router.put(route("todo.update", todo.id), {
        name: todo.name,
    });
},
```

Con esto, al dar un click sobre algun to do del listado, entramos a modo edición:



Todo Test

Figura 16-3: Edición del todo: Modo Edición activada

Y al dar un enter, entramos nuevamente en modo lectura:



Todo Test

Figura 16-4: Edición del todo: Modo Edición desactivada

Con este mecanismo, podemos editar fácilmente los to do.

Aplicar validaciones

Para las validaciones, usaremos el mismo esquema para crear, y las validaciones se mostrarán a nivel del listado, del to do que se esté editando:

app/Http/Controllers/TodoController.php

```
public function update(Todo $todo, Request $request)
{
    $data = $request->validate(
        [
            'name' => "required|min:2|max:255",
        ]
    );

    Todo::where("id", $todo['id'])->where('user_id', auth()->id())->update(['name' =>
    $data['name']]);
    return back();
}
```

Lo que hará que los errores se inyecten automáticamente al **prop de errors** desde la vista, el cual usaremos a nivel de cada listado:

resources/js/Pages/Todo/Index.vue

```
<span v-show="!t.editMode" @click="t.editMode = true">
    {{ t.name }}
</span>
<InputError v-if="todoSelected == t.id" :message="errors.name" />
```

Lo que hará, que cada vez que tengamos un error, se muestra en todas las partes que se encuentre definido el componente de **InputError**:

App To do

Create To Do CREATE

The name field is required.

- 898988989998
The name field is required.

- 9889898
The name field is required.

- asas
The name field is required.

- asas
The name field is required.

- asas
The name field is required.

Figura 16-5: Edición del todo: Error de validación mostrándose en todo los campos

Que no es lo que queremos; para evitar esto, usaremos un pivote con el cual aplicaremos un condicional para que solamente se muestre el error donde queremos:

```
data() {
  return {
    form: {
      name: "",
    },
    todoSelected: 0, // 0 create, >0 update
  };
},
```

Si el pivote es cero, significa que estamos en modo creación, un número positivo, en modo de actualización, el número positivo representa el ID del to do, así que, para gestionar el mismo, tenemos:

```
create() {
  this.todoSelected = 0;
  router.post(
    route("todo.store", {
      name: this.form.name,
    })
  );
},
update(todo) {
  this.todoSelected = todo.id;
  todo.editMode = false;

  router.put(
    route("todo.update", todo.id),
    {
      name: todo.name,
    },
  );
},
```

En pocas palabras, al momento de enviar una petición, si es crear, establecemos el cero el pivote, si es para actualizar, registramos el ID del to do.

Y a nivel del template, colocamos los condicionales para mostrar los errores según la operación que esté realizando:

resources/js/Pages/Todo/Index.vue

```
<form @submit.prevent="create" class="flex gap-2 mt-2">
  <TextInput :class="{'text-red-400 bg-red-200': (errors.name && todoSelected == 0) }"
    v-model="form.name" placeholder="Create To Do"></TextInput>
  <PrimaryButton>Send</PrimaryButton>
```

```

</form>
<InputError v-if="todoSelected == 0" :message="errors.name" />

<ul>
  <li class="border py-3 px-4 mt-2" v-for="t in todos" :key="t">
    <span v-show="!t.editMode" @click="t.editMode = true">
      {{ t.name }}
    </span>
    ***
    <InputError
      v-if="todoSelected == t.id"
      :message="errors.name"
    />
  ***

```

Y esto sería todo, con estos sencillos pasos, podemos mostrar el mensaje de error justamente donde lo queremos.

Remover un to do

Vamos a implementar la eliminación de un to do desde el listado; para esto, vamos a usar el mismo mecanismo que presentamos anteriormente en el CRUD de los POST, es decir, un diálogo de confirmación, en el cual registramos el id del todo que queremos eliminar mediante un pivote, para luego, desde el diálogo de confirmación, realizar la operación de eliminación:

resources/js/Pages/Todo/Index.vue

```

<template>
  <web-layout>
    <confirmation-modal :show="confirmDeleteActive">
      <template v-slot:title> Confirmation </template>

      <template v-slot:content>
        <p class="p-4">Are you sure you want to delete the record?</p>
      </template>

      <template v-slot:footer>
        <o-button variant="danger" @click="destroy">Delete</o-button>
        <div class="mr-3"></div>
        <o-button @click="confirmDeleteActive = false">Cancel</o-button>
      </template>
    </confirmation-modal>

    <div class="card">
      <div class="card-body">
        <h3>App To do</h3>

```

```

<form @submit.prevent="create" class="flex gap-2 mt-2">
  ***
<ul>
  <li v-for="t in todos" class="border py-3 px-4 mt-2" :key="t">
    {{ t.editMode }} -
    ***
    <button @click="confirmDeleteActive = true; deleteTodoRow=t.id"
class="float-right">
      <svg
        xmlns="http://www.w3.org/2000/svg"
        class="h-6 w-6"
        fill="#F00"
        viewBox="0 0 24 24"
        stroke="currentColor"
        stroke-width="2"
      >
        <path
          stroke-linecap="round"
          stroke-linejoin="round"
          d="M19 71-.867 12.142A2 2 0 0116.138 21H7.862a2 2 0 01-1.995-1.858L5 7m5
4v6m4-6v6m1-10V4a1 1 0 00-1-1h-4a1 1 0 00-1 1v3M4 7h16"
        />
      </svg>
    </button>
  </li>
</ul>
</div>
</div>
</web-layout>
</template>

<script>
import WebLayout from "@/Layouts/WebLayout.vue";
***
import ConfirmationModal from "@/Components/ConfirmationModal.vue";

import { Inertia } from "@inertiajs/inertia";

export default {
  components: {
    WebLayout,
    **|,
    ConfirmationModal,
  },
  props: {

```

```

    todos: Object,
    errors: Object,
},
data() {
  return {
    form: {
      name: "",
    },
    todoSelected: 0, // 0 create, >0 update
    confirmDeleteActive: false,
    deleteTodoRow: "",
  };
},
methods: {
  destroy() {
    this.confirmDeleteActive = false;
    router.delete(route("todo.destroy", this.deleteTodoRow));
  },
  ***
},
};
</script>

```

El controlador:

app/Http/Controllers/TodoController.php

```

public function destroy(Todo $todo)
{
  Todo::where("id", $todo->id)->where('user_id', auth()->id())->delete();
  return back();
}

```

Y su ruta:

routes\web.php

```

Route::middleware(
  [
    'auth:sanctum',
    config('jetstream.auth_session'),
    'verified',
  ]
)->prefix('todo')->group(function () {
  ***

```

```

Route::delete('destroy/{todo}', [App\Http\Controllers\TodoController::class,
'destroy'])->name('todo.destroy');
});

```

Con esto, al presionar sobre el ícono de papelera en el listado de to do para eliminar veremos el diálogo de confirmación, y al presionar el botón de confirmación, eliminaremos el registro; y con esto, completamos nuestro CRUD.

Marcar completado un to do

Para la función de marcar un to do como completado o no, se realizará un proceso similar al proceso de actualizar; para esta funcionalidad, usaremos un SVG para determinar si está activo o no:

resources/js/Pages/Todo/Index.vue

```

***  

<svg  

  xmlns="http://www.w3.org/2000/svg"  

  class="h-4 w-4 inline mr-1"  

  fill="none"  

  viewBox="0 0 24 24"  

  :stroke="t.status == '1' ? '#0F0' : '#000'"  

  stroke-width="1"  

  @click="status(t)"  

>  

  <path  

    v-if="t.status == '1'"  

    stroke-linecap="round"  

    stroke-linejoin="round"  

    d="M9 12l2 2 4-4m6 2a9 9 0 11-18 0 9 9 0 0118 0z"  

  />  

  <path  

    v-else  

    fill-rule="evenodd"  

    d="M13.477 14.89A6 6 0 015.11 6.52418.367 8.368zm1.414-1.414L6.524 5.11a6 6 0 018.367  

8.367zM18 10a8 8 0 11-16 0 8 8 0 0116 0z"  

    clip-rule="evenodd"  

  />  

</svg>  

<span v-show="!t.editMode" @click="t.editMode = true">  

  {{ t.name }}  

</span>
***
```

Como puedes ver, según el estado, se usa un tipo de SVG u otro.

Desde el apartado de script, se crea una función para cambiar el estado:

resources/js/Pages/Todo/Index.vue

```
status(todo) {
  todo.status = !todo.status;
  router.post(route("todo.status", todo.id), {
    status: todo.status,
  });
},
```

Como puedes ver, al no usar un checkbox, el toggle del estado lo hacemos de manera manual:

```
todo.status = !todo.status;
```

Definimos el controlador:

app/Http/Controllers/TodoController.php

```
public function status(Todo $todo)
{
    Todo::where("id", $todo->id)->where('user_id', auth()->id())->update(['status' =>
request('status') == '1']);
    return back();
}
```

Y su ruta:

routes\web.php

```
Route::middleware(
  [
    'auth:sanctum',
    config('jetstream.auth_session'),
    'verified',
  ]
)->prefix('todo')->group(function () {
  ***
  Route::post('status/{todo}', [App\Http\Controllers\TodoController::class,
  'status'])->name('todo.status');
});
```

Y tendremos:

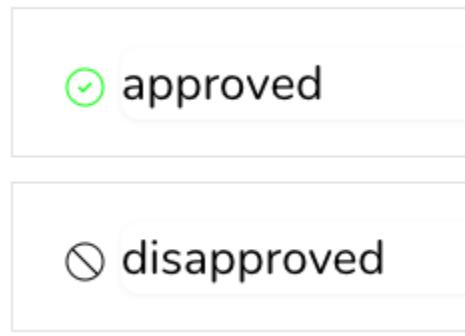


Figura 16-6: Icono del status del todo

Los iconos SVG fueron tomados de <https://heroicons.com/>

Eliminar todos los to do

Para eliminar todos los to do, se va a usar el mismo método de `delete()` implementada para eliminar un solo todo, pero colocando el parámetro de todo como opcional:

app/Http/Controllers/TodoController.php

```
public function destroy(Todo $todo = null)
{
    if ($todo == null)
        Todo::where('user_id', auth()->id())->delete();
    else
        Todo::where('user_id', auth()->id())->where('id', $todo->id)->delete();
    return back();
}
```

Y lo mismo hacemos en su ruta:

routes\web.php

```
Route::delete('destroy/{todo?}', [App\Http\Controllers\TodoController::class,
'destroy'])->name('todo.destroy');
```

En el componente en Vue, creamos el botón que llama a un nuevo método, la cual es idéntica a la de eliminar un todo, salvo que no recibe el parámetro:

resources\js\Pages\Todo\Index.vue

```
<div class="flex flex-row-reverse">
    <div>
        <PrimaryButton @click="destroyAll">
```

```

        <svg xmlns="http://www.w3.org/2000/svg" fill="none" viewBox="0 0 24 24"
stroke-width="1.5"
          stroke="#F00" class="size-6">
          <path stroke-linecap="round" stroke-linejoin="round"
            d="m14.74 9-.346 9m-4.788 0L9.26 9m9.968-3.21c.342.052.682.107
1.022.166m-1.022-.165L18.16 19.673a2.25 2.25 0 0 1-2.244 2.077H8.084a2.25 2.25 0 0
1-2.244-2.077L4.772 5.79m14.456 0a48.108 48.108 0 0 0-3.478-.397m-12 .562c.34-.059.68-.114
1.022-.165m0 0a48.11 48.11 0 0 1 3.478-.397m7.5 0v-.916c0-1.18-.91-2.164-2.09-2.201a51.964
51.964 0 0 0-3.32 0c-1.18.037-2.09 1.022-2.09 2.201v.916m7.5 0a48.667 48.667 0 0 0-7.5 0"
/>
      </svg>
    Delete All
  </PrimaryButton>
</div>
</div>
</div>
</div>
</web-layout>
</template>
***
```

destroyAll() {
 router.delete(route("todo.destroy"));
},

Ya con esto, podemos eliminar todos los to do con un botón.

Ordenación

La siguiente funcionalidad que vamos a implementar es la de ordenación de los to do; ordenación aplicada mediante el Drag and Drop en Vue; para ello, vamos a instalar un plugin que permita dicha aplicar la ordenación de los to do via Drag and Drop:

<https://github.com/SortableJS/vue.draggable.next>

Lo instalamos:

```
$ npm i -S vuedraggable@next
```

Y a diferencia de otros plugins, no hay necesidad de configurarlo en la instancia principal de Vue; basta con importarlo como un componente cualquiera:

```
<script>
import draggable from "vuedraggable";

export default {
  components: {
```

```

    draggable,
},
}
</script>

```

Y usarlo; para usarlo, existen varias configuraciones y cuyo detalle lo puedes ver en la documentación oficial, pero, el uso básico, que consiste en crear una lista que se pueda reordenar via Drag and Drop, tenemos:

```

<draggable v-model="myArray">
  <template #item="{ element }">
    <div>
      {{ element }}
    </div>
  </template>
</draggable>
*** 
data() {
  return {
    myArray: ["Test", "Test 2", "Test 3", "Test 4"],
  };
},

```

Mediante el componente **draggable**, indicamos el **v-model** indicamos la propiedad, que debe ser un **array**, ya sea de String, como en el ejemplo anterior, o de objetos; no hay necesidad de emplear un **v-for** para iterar los datos, en su lugar, podemos usar el:

```
#item="{ element }"
```

Para representar un elemento del **array** especificado en el **v-model**.

Con esto, tendremos:

```

Test
Test 2
Test 3
Test 4

```

Figura 16-7: Listado ordenable

A su vez, tenemos eventos que se ejecutan en algún momento del Drag and Drop; entre los más importante, tenemos:

- **start**: Se ejecuta al inicio del Drag and Drop.
- **end**: Se ejecuta después del Drag and Drop.

Ordenar el listado de to do

Ahora, ya aclarado el funcionamiento, vamos a adaptar el listado de to do:

```
<ul id="listToDo">
  <draggable v-model="dtodos" item-key="id" @end="order">
    <template #item="{ element }">
      <li :data-id="element.id" class="border py-3 px-4 mt-2">
        <!-- <li v-for="t in todos" class="border py-3 px-4 mt-2" :key="t">-->

        <svg @click="status(element)" xmlns="http://www.w3.org/2000/svg"
fill="none"
          :stroke="element.status == '1' ? '#0F0' : '#000'" viewBox="0 0 24 24"
          stroke-width="1.5" class="size-4 inline mr-1">

          <path v-if="element.status == '1'" stroke-linecap="round"
stroke-linejoin="round"
            d="M9 12.75 11.25 15 15 9.75M21 12a9 9 0 1 1-18 0 9 9 0 0 1 18 0Z"
/>

          <path v-else stroke-linecap="round" stroke-linejoin="round"
            d="M15 12H9m12 0a9 9 0 1 1-18 0 9 9 0 0 1 18 0Z" />

        </svg>

        <span v-show="!element.editMode" @click="element.editMode = true">{{

element.name
          }}</span>
        <InputError v-if="todoSelected == element.id" :message="errors.name" />
        <TextInput v-show="element.editMode == true" v-model="element.name"
          @keyup.enter="update(element)" />
        <button class="float-right"
          @click="confirmDeleteActive = true; deleteTodoRow = element.id">

          <svg xmlns="http://www.w3.org/2000/svg" viewBox="0 0 24 24"
stroke-width="1.5"
            stroke="currentColor" class="h-6 w-6" fill="#F00">
            <path stroke-linecap="round" stroke-linejoin="round"
              d="m14.74 9-.346 9m-4.788 0L9.26 9m9.968-3.21c.342.052.682.107
1.022.166m-1.022-.165L18.16 19.673a2.25 2.25 0 0 1-2.244 2.077H8.084a2.25 2.25 0 0
1-2.244-2.077L4.772 5.79m14.456 0a48.108 48.108 0 0 0-3.478-.397m-12 .562c.34-.059.68-.114
1.022-.165m0 0a48.11 48.11 0 0 1 3.478-.397m7.5 0v-.916c0-1.18-.91-2.164-2.09-2.201a51.964
51.964 0 0 0-3.32 0c-1.18.037-2.09 1.022-2.09 2.201v.916m7.5 0a48.667 48.667 0 0 0-7.5 0"
/>
          </svg>

```

```

        </button>
    </li>
</template>
</draggable>
</ul>
data() {
    return {
        ***
        deleteTodoRow: '',
        dtodos: this.todos
    }
},
*** 
order() {
    let ids = "";
    document.querySelectorAll("#list-to-do li").forEach((li) => {
        // console.log(li.getAttribute("data-id"));
        ids += li.getAttribute("data-id") + ",";
    });
    console.log(ids.slice(0,-1))
},

```

Los cambios realizados son muy puntuales, y son los siguientes:

- Adaptamos el listado de UL de los to do, para funcionar con el plugin del Drag and Drop anterior y definimos el evento de **end**, para cuando termina el Drag and Drop.
- Colocamos un ID al listado de UL para poder referenciar los LIs (to do) y poder generar el listado de IDs ordenados; como puedes ver, componemos un **array** de IDs de to do.
- Creamos un listado auxiliar llamado **dtodos** ya que no es posible usar el prop de **todos** como v-model.

Este enfoque de ordenación, es muy sencillo de aplicar ya que, siempre vamos a tener en orden los IDs reflejados en el **array** anterior desde el componente en Vue; así que, lo único que debemos de realizar es, iterar este listado de IDs desde el servidor y aplicar los cambios en la ordenación de los to do según se encuentran posicionados cada uno de los ID de los to do en el **array**.

Desde el controlador, también podemos recibir un array con los IDs en vez de un String y con esto, facilitar el procesamiento:

resources/js/Pages/Todo/Index.vue

```

order() {
    let ids = [];
    document.querySelectorAll("#list-to-do li").forEach((li) => {
        ids.push(li.getAttribute("data-id"));
    });
    router.post(route("todo.order"), {
        ids: ids,
    })
}

```

```
|     });
| },
```

Creamos el controlador, que va a recibir el **array** anterior y se encarga de iterar uno a uno sus componentes y actualizar las posiciones:

app/Http/Controllers/TodoController.php

```
public function order()
{
    foreach (request('ids') as $count => $id) {
        Todo::where('user_id', auth()->id())->where("id", $id)->update(['count' =>
$count]);
    }
}
```

Creamos la ruta:

routes/web.php

```
Route::middleware(
    [
        'auth:sanctum',
        config('jetstream.auth_session'),
        'verified',
    ]
)->prefix('todo')->group(function () {
    ***
    Route::post('order', [App\Http\Controllers\TodoController::class,
    'order'])->name('todo.order');
});
```

Con esto, completamos la aplicación de to do, aplicado a un usuario autenticado, con el típico CRUD y los extras de la eliminación de todo los to do y la reordenación via Drag and Drop; con esta aplicación, mostramos otro enfoque en el uso de los componentes en Vue en relación de los controladores, y mostramos que podemos tener múltiples controladores y un solo componente que utilice los controladores para la gestión de la data.

Problemas con la sincronización de las operaciones

Actualmente la aplicación no logra sincronizar correctamente cuando se hacen las operaciones CRUD sobre los to do; es decir, es necesario recargar la pagina cada vez que se crean, actualizan o eliminan los to do; para evitar este comportamiento, podemos ir actualizando el **array** de to do cada vez que se realizan estas operaciones:

resources/js/Pages/Todo/Index.vue

```
|create() {
```

```

this.todoSelected = 0;
//this.todos.push();
// router.post(
//   route("todo.store", {
//     name: this.form.name,
//   })
// )
// );

axios.post(route("todo.store", {
  name: this.form.name,
})).then((todo)=> {
  console.log(todo)
  this.todos.push(todo.data)
})

this.form.name = ''
},
destroy() {
  this.confirmDeleteActive = false;
  this.todos = this.todos.filter((t) => t.id != this.deleteTodoRow) // devolvemos un
array de todos los elementos que NO queremos eliminar
  router.delete(route("todo.destroy", this.deleteTodoRow));
},
destroyAll() {
  this.todos = [] // elimina todos los to do
  router.delete(route("todo.destroy"));
},

```

En el caso de la creación, es necesario usar otro esquema, ya que, tal cual comenta el equipo de Inertia:

"Using Inertia to submit forms works great for the vast majority of situations; however, in the event that you need more control over the form submission, you're free to make plain XHR or fetch requests instead using the library of your choice."

<https://inertiajs.com/forms#xhr-fetch-submissions>

En este caso, es necesario tener más control ya que, es necesario ingresar el to do creado mediante un **push()** en el array de to do; por lo tanto, se realiza una petición mediante **axios**, para que con la respuesta provista por el controlador:

app/Http/Controllers/TodoController.php

```

public function store(Request $request)
{
  $data = $request->validate(
    [

```

```

        'name' => "required|min:2|max:255",
    ]
);
$todo = Todo::create([
    'name' => $data['name'],
    'user_id' => auth()->id(),
    'count' => Todo::where('user_id', auth()->id())->count()
]);
return response()->json($todo);

//return back();
}

```

Se almacena en el array de los to do:

```

create() {
    this.todoSelected = 0;
    //this.todos.push();
    // router.post(
    //     route("todo.store", {
    //         name: this.form.name,
    //     })
    //     )
    // );

    axios.post(route("todo.store", {
        name: this.form.name,
    })).then((todo)=> {
        console.log(todo)
        this.todos.push(todo.data)
    })
}

```

Validación

Al usar ahora peticiones por axios, se pierden algunas funcionalidades como lo es mostrar los errores cuando ocurran los errores de validación; así que, para corregir esto, tendremos que aplicar las validaciones de manera manual y enviar una respuesta con los errores:

```

use Response;
use Validator;
*** 
public function store(Request $request)
{
    $validator = Validator::make($request->all(), [

```

```

        'name' => "required|min:2|max:255",
    ]);

if ($validator->fails()) {
    return Response::json([
        'error' => $validator->errors()->get('name')[0]
    ], 400);
}

// $data = $request->validate(
//     [
//         'name' => "required|min:2|max:255",
//     ]
// );

$todo = Todo::create([
    'name' => $request->name,
    'user_id' => auth()->id(),
    'count' => Todo::where('user_id', auth()->id())->count()
]);

return response()->json($todo);
}

```

Al hacer este cambio de enviar las peticiones mediante axios y no mediante el **router** de Inertia, perdemos la integración de mostrar los errores de manera automática cuando existe un error de validación mediante los **props**; así que, hay que realizarla de manera manual, pero solo para el caso del input de creación de las tareas, es decir, los to do para editar queda igual:

```

<form @submit.prevent="create" class="flex gap-2 mt-2">
    <jet-input :class="{ 'text-red-400 bg-red-200': form.error }" placeholder="Create To
Do"
        v-model="form.name" />
    <jet-button>Create</jet-button>
</form>
***
data() {
    return {
        ***
        form: {
            name: '',
            error: ''
        },
    }
},
***
```

```

create() {
    this.todoSelected = 0
    this.form.error = ''

    axios.post(route("todo.store", {
        name: this.form.name,
    })).then((todo)=> {
        console.log(todo)
        this.todos.push(todo.data)
    }).catch((error) => {
        console.log(error.response.data.error)
        this.form.error = error.response.data.error
    })

    this.form.name = ''
},

```

Se usa el **catch()** ya que, es el método que se ejecuta para las respuestas con código HTTP 400; con esto completamos la aplicación de to do.

Otras soluciones

Otra posible solución podría ser hacer una redirección de la página para recargar los To Do en cada operación que sea necesaria como la de creación y eliminación individual; algo como:

resources\js\Pages\Todo\Index.vue

```

create() {
    this.todoSelected = 0
    router.post(route('todo.store', {
        name: this.form.name
    }))
    setTimeout(() => window.location.reload(), 500)
},

```

Con las 500 milésimas de segundo daría tiempo de enviar la petición post antes de recargar.

Otra forma podría ser empleando las opciones de las peticiones mediante el router:

```

import { router } from '@inertiajs/vue3'

router.<method>('<route>', data, {
    onBefore: (visit) => {},
    onStart: (visit) => {},
    onProgress: (progress) => {},
    onSuccess: (page) => {},
}

```

```

onError: (errors) => {},
onCancel: () => {},
onFinish: visit => {},
})

```

Para recargar la página o mediante el objeto **\$page** obtener todos los to do:

resources\js\Pages\Todo\Index.vue

```

create() {
    this.todoSelected = 0
    router.post(route('todo.store'), {
        name: this.form.name
    }, {
        onSuccess: (page) => {
            // console.log(page)
            console.log(page.props.todos)
            this.dtodos = page.props.todos
            // setTimeout(() => window.location.reload(), 500)

        },
    })
},
***  

router.delete(route('todo.destroy', this.deleteTodoRow), {
    preserveScroll: true,
    onSuccess: (page) => {
        // console.log(page)
        console.log(page.props.todos)
        this.dtodos = page.props.todos
        // setTimeout(() => window.location.reload(), 500)

    },
})

```

También pudieras habilitar la opción preservar el scroll al momento de eliminar los To Do como mostramos en el código anterior:

```
|router.delete(route('todo.destroy', this.deleteTodoRow), { preserveScroll: true })
```

Con esto, evitamos que si eliminamos To Do que solamente se pueden visualizar mediante el scroll, que al momento de eliminar se mueva el scroll al inicio.

Código fuente del capítulo en:

<https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.11>

Capítulo 17: Pruebas

En este capítulo, vamos a realizar las pruebas para el proyecto creado anteriormente, para cada uno de los módulos, crearemos las pruebas no en el mismo orden en el cual fueron desarrollados los módulos en el libro, si no, crearemos las pruebas comenzando con los módulos más sencillos como sería el del blog.

Usaremos el framework de pruebas de PHPUnit para crear cada una de las pruebas, pero, puedes emplear Pest si así lo prefieres ya que, hay casi que una relación de uno a uno con los métodos de aserción entre PHPUnit y Pest.

Primeros pasos

Comenzaremos creando las pruebas para el módulo de blog, es decir, para el listado y para la página de detalle.

Crearemos la prueba para el blog:

```
$ php artisan make:test Blog/PostTest
```

De Laravel a Inertia

Definiremos la siguiente prueba para el listado, la cual, nos permitirá ejemplificar los cambios entre las pruebas de Laravel o Inertia:

tests\Feature\BlogTest.php

```
<?php

namespace Tests\Unit\Blog;

// use PHPUnit\Framework\TestCase;

use Tests\TestCase;

class PostTest extends TestCase
{
    public function test_index(): void
    {
        $this->get(route('web.index'))
            ->assertViewIs('blog.show');
            ->assertStatus(200);
    }
}
```

La prueba anterior permite verificar que el listado de posts en el index devuelve un código de estado 200.

Ejecutamos:

```
$ php artisan test
```

Y veremos un resultado como:

```
Unable to locate file in Vite manifest: resources/js/Pages/Blog/Index.vue. (View:  
C:\Users\andre\Herd\inertiastore\resources\views\app.blade.php)
```

En el cual, nos indica que debemos de tener habilitado los archivos generados por vite, así que, o habilitas el modo desarrollo:

```
$ npm run dev
```

O puedes generar los archivos a producción:

```
$ npm run prod
```

Si ejecutas las pruebas otra vez, verás que ahora tenemos un error como el siguiente:

```
+++ Actual  
@@ @@  
- 'blog.show'  
+ 'app'
```

Esto se debe a que el método de **assertViewIs()** es empleado para verificar vistas de blade y no componentes en Vue, en base al error anterior, puedes ver que inertia internamente al emplear el método de **inertia()** para retornar un componente, emplea una vista llamada app:

```
->assertViewIs('app')
```

Si ejecutas otra vez, verás que ya no sucede el error anterior, pero, dejar definido una vista app que nosotros no estamos empleando, ya que nosotros. Estamos usando los componentes en Vue no tiene sentido, es ahora donde entran las aserciones creadas específicamente para inertia que veremos en el siguiente apartado.

Assert Inertia

En Inertia, tenemos aserciones específicas para trabajar con los componentes de Vue, para ello, debemos de importar a nivel del archivo de pruebas:

```
tests\Unit\Blog\PostTest.php
```

```
// use PHPUnit\Framework\TestCase;  
use Tests\TestCase;  
  
use Inertia\Testing\AssertableInertia as Assert;
```

```

class YourTest extends TestCase

    public function test_test(): void
    {
        $this->get(route('web.index'))
            ->assertInertia(fn (Assert $page) => dd($page)));
    }
}

```

En la prueba de listado, en cuyo controlador empleamos el componente de **Assert**, debemos de configurarlo de la siguiente manera a nivel de la prueba:

tests\Unit\Blog\PostTest.php

```

<?php

namespace Tests\Unit\Blog;

// use PHPUnit\Framework\TestCase;

use Tests\TestCase;

use Inertia\Testing\AssertableInertia as Assert;

class PostTest extends TestCase
{
    public function test_index(): void
    {
        $this->get(route('web.index'))
            // ->assertViewIs('app')
            ->assertStatus(200);

        $this->get(route('web.index'))
            ->assertInertia(fn (Assert $page) => dd($page)
                ->component('Blog/Index')
                ->has('posts', fn (Assert $page) => $page
                    )
            );
    }
}

```

Y veremos que, si ejecutamos las prueba, la misma pasa; hay muchas verificaciones que se pueden realizar a nivel de las aserciones de Inertia como puedes ver en la documentación oficial:

<https://inertiajs.com/testing>

Aunque la sintaxis es algo extraña, vamos a ir paso a paso para saber exactamente que como está estructurada, evaluemos el objeto llamado **\$page** a ver que provee:

```
| ->assertInertia(fn (Assert $page) => dd($page))
```

Al ejecutar la prueba, veremos una salida cómo la siguiente:

```
Inertia\Testing\AssertableInertia {#2397
    -props: array:14 [
        "errors" => []
        "jetstream" => array:11 [
            "canCreateTeams" => false
            "canManageTwoFactorAuthentication" => true
            "canUpdatePassword" => true
            "canUpdateProfileInformation" => true
            "hasEmailVerification" => false
            "flash" => []
            "hasAccountDeletionFeatures" => true
            "hasApiFeatures" => true
            "hasTeamFeatures" => true
            "hasTermsAndPrivacyPolicyFeature" => true
            "managesProfilePhotos" => true
        ]
        "auth" => array:1 [
            "user" => null
        ]
        "errorBags" => []
        "flash" => array:1 [
            "message" => null
        ]
        "step" => 1
        "cart" => []
        "posts" => array:13 [
            "current_page" => 1
            "data" => array:15 [
                0 => array:13 [
                    "id" => 2
                    "title" => "Post 5111"
                    "slug" => "post-4"
                    "date" => "2024-08-22"
                    "image" => "1729333215.png"
                    "text" => "asasasasas"
                    "description" => "asasasas"
                    "posted" => "not"
                    "type" => "course"
                    "category_id" => 1
                ]
            ]
        ]
    ]
}
```

```

    "created_at" => "2024-08-18T09:54:24.000000Z"
    ***
    "category_id" => 2
    "created_at" => "2024-09-21T09:43:12.000000Z"
    "updated_at" => "2024-09-21T09:43:12.000000Z"
    "category" => array:7 [
        "id" => 2
        "title" => "Cate 2"
        "slug" => "cate-2"
        "image" => null
        "text" => null
        "created_at" => "2024-08-15T10:08:19.000000Z"
        "updated_at" => "2024-08-15T10:08:19.000000Z"
    ]
]
]
"first_page_url" => "http://inertiastore.test/blog?page=1"
"from" => 1
"last_page" => 47
"last_page_url" => "http://inertiastore.test/blog?page=47"
"links" => array:15 [
    0 => array:3 [
        "url" => null
        "label" => "&lquo; Previous"
        "active" => false
    ***
        "active" => false
    ]
]
]
"next_page_url" => "http://inertiastore.test/blog?page=2"
"path" => "http://inertiastore.test/blog"
"per_page" => 15
"prev_page_url" => null
"to" => 15
"total" => 702
]
"categories" => array:2 [
    0 => array:7 [
        "id" => 1
        "title" => "Cate 1"
        "slug" => "category-1"
        "image" => null
        "text" => null
        "created_at" => "2024-08-15T10:08:12.000000Z"
        "updated_at" => "2024-08-15T10:08:12.000000Z"
    ]
]
```

```

1 => array:7 [
    "id" => 2
    "title" => "Cate 2"
    "slug" => "cate-2"
    "image" => null
    "text" => null
    "created_at" => "2024-08-15T10:08:19.000000Z"
    "updated_at" => "2024-08-15T10:08:19.000000Z"
]
]
"prop_type" => null
"prop_category_id" => null
"prop_from" => null
"prop_to" => null
"prop_search" => null
]
-path: null
#interacted: []
-component: "Blog/Index"
-url: "/blog"
-version: "b05311e78830e9fb34e382b9802ceab2"

```

La salida anterior fue recordaba para evitar llenar 4 páginas con datos que nosotros no vamos a evaluar en esta guía, pero, se recomienda al lector que haga la prueba y evalúe el resultado.

Veremos que la salida anterior corresponde al objeto global llamado **\$page** que es el que empleamos anteriormente para obtener datos sobre la página, como el componente de Vue empleado, **props**, datos compartidos, de usuario, y más:

resources\js\Pages\Blog\Index.vue

```
 {{ $page }}
```

Los datos suministrados dependen del recurso que queramos evaluar, ya que al igual que las pruebas en Laravel básico, todo depende de cómo esté formado el recurso o controlador a evaluar.

Podemos ir segmentando, por ejemplo, objetos complejos como son el de posts para la paginación:

```

$this->get(route('web.index'))
    ->assertInertia(fn (Assert $page) => $page
        ->component('Blog/Index')
        ->has('posts', fn (Assert $page) => dd($page))

```

Y el detalle que obtendremos, será solo de la paginación:

```

"errors" => []
"jetstream" => array:11 [

```

```

***  

Inertia\Testing\AssertableInertia {#2305
-props: array:13 [
    "current_page" => 1
    "data" => array:15 [
        ***
        14 => array:13 [
            "id" => 18
            "title" => "NSf"
            "slug" => "nsf"
            "date" => "2023-11-07 00:00:00"
            "image" => null
            "text" =>
"59nZkTBFh1v6Nt42Xz4Gsx1YbrujpK0ykmX0koIZ1uv5o4HVHKpvuciBQbqukXwUPrnGxirZkvspZPK1zQnHu0Dvp0
L98CyoqM053CKmi2KKmIiBkKSzTgKt38jwGk5YUkKd0c06YwuyzwQ0AFmxzP1fEnAvl1dpb1f038Fj6y3YeIk4gqB7
cTcMkpaq5zyhQY2guGYL8vB8Et1l0605kqL8HvpHGymZR0OaabEKBbIjBsA687NZwNFwqEEoVQdkekOM2tv0xFXC2H
M514t5ystQMR7oflo8wuDFyDyraGI5H4sz0ZSIT8HIt7gDt26RMzhYARx9aaw1vY0U2cPiLPCU9MNf4aoIUvSshcfLi
U50xSVbbCxZHlo0eT5zJFq7K59IcStqipbDFV0HjuYZmorXkz3hJA9Nb6ZwmjdY0zhCgt66R6AFxq0QmguUvi6rertu
PP"
            "description" => null
            "posted" => "yes"
            "type" => "movie"
            "category_id" => 2
            "created_at" => "2024-09-21T09:43:12.000000Z"
            "updated_at" => "2024-09-21T09:43:12.000000Z"
            "category" => array:7 [
                "id" => 2
                "title" => "Cate 2"
                "slug" => "cate-2"
                "image" => null
                "text" => null
                "created_at" => "2024-08-15T10:08:19.000000Z"
                "updated_at" => "2024-08-15T10:08:19.000000Z"
            ]
        ]
    ]
    "first_page_url" => "http://inertiastore.test/blog?page=1"
    "from" => 1
    "last_page" => 47
    "last_page_url" => "http://inertiastore.test/blog?page=47"
    "links" => array:15 [
        ***
        14 => array:3 [
            "url" => "http://inertiastore.test/blog?page=2"
            "label" => "Next &raquo;"
            "active" => false
        ]
    ]
]

```

```

        ]
    ]
    "next_page_url" => "http://inertiastore.test/blog?page=2"
    "path" => "http://inertiastore.test/blog"
    "per_page" => 15
    "prev_page_url" => null
    "to" => 15
    "total" => 702
]
***
```

Como puedes ver, el detalle devuelto NO es el objeto de **\$page**, que tiene muchísima data, es simplemente del objeto que se está observando que en este ejemplo es el prop de **posts**, por lo tanto, podemos darle un mejor nombrado acorde a la respuesta:

```

$this->get(route('web.index'))
    ->assertInertia(fn (Assert $page) => $page
        ->component('Blog/Index')
        ->has('posts', fn (Assert $posts) => dd($posts))
```

Métodos has y where

Existen un par de métodos claves que debemos de tener en cuenta al momento de crear las pruebas empleando el Assert de Inertia, el método de **has()** y **where()** que exemplificamos en el siguiente apartado.

Desde el componente de listado, tenemos varios props que podemos verificar su integridad y si están presentes, qué datos deben de manejar:

resources\js\Pages\Blog\Index.vue

```

props: {
    posts: Object,
    categories: Array,
    prop_category_id: String,
    prop_type: String,
    prop_from: String,
    prop_to: String,
    prop_search: String,
},
```

El listado paginado de los posts, es algo complejo, ya que tiene los filtros de tipo **when** que implementamos antes, pero, al hacer la petición desde la prueba, no estamos inyectando valores al filtro:

app\Http\Controllers\Blog\PostController.php

```

$posts = Post::
    when(***)
```

```
})->with('category')
    ->paginate(15);
```

Por lo tanto, al crear el query en la prueba, podemos prescindir de los filtros quedando la consulta con lo señalado en negritas en el fragmento de código anterior.

Recordemos que en el controlador, exponemos las siguientes variables/props al componente de Vue:

```
$this->get(route('web.index'))
    ->assertInertia(fn (Assert $page) => $page
        ->component('Blog/Index')
        ->where('categories', Category::get())
        ->where('prop_from', null)
        ->where('posts', Post::with('category')->paginate(15))
    );
```

Mediante el método de **where()**, podemos comprobar por cada uno de los props por el valor que deben de tener.

Mediante el método **has()** podemos verificar si el prop existe, pasando solamente la key:

```
->has(<KEY>)
->has('prop_from')
```

O si la longitud del mismo, pasando el segundo parámetro que corresponde a la longitud:

```
->has(<KEY>,<LENGTH>)
->has('posts',15)
```

Por ejemplo:

```
$this->get(route('web.index'))
    ->assertInertia(fn (Assert $page) => $page
        ->component('Blog/Index')
        ->where('categories', Category::get())
        ->where('prop_from', null)
        ->where('posts', Post::with('category')->paginate(15))
        ->has('prop_from')
        ->has('posts', 15)
        ->has('posts', fn (Assert $page) => $page
            ->where('last_page', 47))
```

También podemos emplear el método de **has()** para navegar sobre el objeto observado y aplicar condiciones, por ejemplo:

```
->has('categories', fn(Assert $page) => dd($page))
```

De igual manera, podemos inspeccionar o navegar por el objeto:

```
->has('categories.0.title')
->where('categories.0.title', 'Cate 1')
```

Blog

En este apartado, vamos a crear las pruebas para el blog, que son dos controladores, el de listado y el de detalle.

Prueba para el listado

La prueba del listado fue completada en base a los experimentos que hicimos anteriormente para conocer los métodos de aspiración provistos por Inertia:

tests\Feature\BlogTest.php

```
class BlogTest extends TestCase
{
    public function test_index(): void
    {
        Category::factory(3)->create();
        Post::factory(100)->create();

        $category = Category::first();

        $this->get(route('web.index'))
            // ->assertViewIs('app')
            ->assertStatus(200);

        $this->get(route('web.index'))->assertInertia(
            // fn(Assert $page) => dd($page)
            fn(Assert $page) => $page
                ->component('Blog/Index')
                ->has('posts')
                ->has('posts', 13)
                ->has('categories.0.title')
                ->where('categories.0.title', $category->title)
                ->where('categories', Category::get())
                ->where('prop_from', null)
                ->where('posts', Post::with('category')->paginate(15))
        );
    }
}
```

Es importante mencionar que no existe un método que evalúe directamente lo que tenemos en el componente, como tenemos en Laravel, el método de aserción de **assertSee()** para comprobar si los props u otros fueron empleados a nivel del componente/vista, en estos casos, tendrías que crear pruebas directamente para Vue.

Prueba para el detalle

La prueba de detalle queda como:

tests\Feature\BlogTest.php

```
public function test_show(): void
{
    $post = Post::with('category')->first();
    // $post->category;

    $this->get(route('web.show', ['post' => $post]))
        ->assertStatus(200)
        ->assertInertia(
            fn(Assert $page) => $page
                ->component('Blog/Show')
                ->has('post')
                ->has('post.category')
                ->has('post.category.title')
                ->where('post.title', $post->title)
                ->where('post', $post)

        );
}
```

Probando campos claves como lo es, los posts con su categoría.

Prueba para el filtro

Para el listado, podemos crear varias pruebas, para saber si se están filtrando datos, por ejemplo, creamos la siguiente y a partir de esta, puedes crear adicionales con el resto de los filtros o variantes:

tests\Feature\BlogTest.php

```
/**
class BlogTest extends TestCase
{
    /**
     * @test
     */
    public function test_index_filter(): void
    {
        Category::factory(3)->create();
        Post::factory(100)->create();
        $category_id = 1;

        // dd(Post::with('category')->toSQL());
    }
}
```

```
// dd(route('web.index', ['type' => 'post']));
$this->get(route('web.index', ['type' => 'post', 'category_id' =>
$category_id]))->assertInertia(
    fn(Assert $page) => $page
        ->component('Blog/Index')
        ->where('posts', Post::with('category')->where('category_id',
$category_id)->paginate(15))
    );
}
```

También puedes ejecutar tus pruebas individuales:

```
$ php artisan test --filter BlogTest
```

O indicando la prueba en especifica:

```
$ php artisan test --filter BlogTest::test_index_filter
```

Lo cual es particularmente útil cuando queremos corregir algún error y estamos haciendo debug con la función de **dd()** en los controladores y pruebas ya que, cuando implementamos varias pruebas que evalúan el mismo recurso, y queremos realizar debug, puede que las pruebas implementadas antes se ejecuten antes de la que estamos implementando, dificultando las correcciones.

En la prueba anterior, definimos en la URL, los filtros que queremos probar:

```
route('web.index', ['type' => 'post', 'category_id' => $category_id]
```

Y colocamos where en base a los mismos:

```
->where('posts', Post::with('category')->where('category_id', $category_id)->paginate(15))
```

No colocamos un where para el tipo, ya que en el Factory implementado no indicamos el tipo, por lo tanto, se toma el tipo por defecto definido en la migración de los posts que es el de 'post'.

Configurar base de datos para pruebas

Usualmente las pruebas unitarias se deben de realizar en una base de datos de prueba, que no sea la de desarrollo y mucho menos la de producción, de momento, hemos estado empleando la base de datos que empleamos en desarrollo, entonces, todas las operaciones realizadas por las pruebas persisten en la misma y con esto, no tenemos un entorno controlado para hacer las pruebas, para establecer una base de datos paralela para hacer las pruebas debemos de realizar una configuración desde el siguiente archivo:

phpunit.xml

```
***  
<php>
```

```

<env name="APP_ENV" value="testing"/>
<env name="APP_MAINTENANCE_DRIVER" value="file"/>
<env name="BCRYPT_ROUNDS" value="4"/>
<env name="CACHE_STORE" value="array"/>

<env name="DB_CONNECTION" value="sqlite"/>
<env name="DB_DATABASE" value=":memory:"/>

<env name="MAIL_MAILER" value="array"/>
<env name="PULSE_ENABLED" value="false"/>
<env name="QUEUE_CONNECTION" value="sync"/>
<env name="SESSION_DRIVER" value="array"/>
<env name="TELESCOPE_ENABLED" value="false"/>
</php>
***
```

Aquí puedes personalizar la base de datos a emplear, en este ejemplo, SQLite (**DB_DATABASE**) y que sea en memoria (**DB_CONNECTION**), lo que significa que las operaciones a la base de datos se van a realizar en una base de datos en memoria y no haciendo operaciones de lectura/escritura sobre la base de datos.

La versión en memoria (**DB_DATABASE**) de la base de datos SQLite (**DB_CONNECTION**) funcionará sólo cuando estamos en entorno de pruebas. Se creará y almacenará en la memoria y luego dejará de existir tan pronto como se cierre la conexión a la base de datos (Si configuramos los **trait** que veremos en el siguiente apartado).

Cuando tenemos muchos casos de prueba, es posible que la prueba se realice más lentamente cuando usamos la base de datos que lee y escribe en el disco. El beneficio de la base de datos en memoria es la velocidad porque la memoria puede funcionar más rápido que el disco.

Como recomendación, se debe de emplear una base de datos independiente para cada ambiente, al igual que no usamos la base de datos de desarrollo (entiéndase los registros) en producción, no deberíamos de emplear la base de datos de desarrollo al momento de hacer las pruebas para siempre trabajar en ambientes controlados y evitar comparar listados vacíos o con formatos NO controlados desde las pruebas.

Esto significa que, al momento de crear las pruebas unitarias, debemos de ejecutar las migraciones, para ello, podemos usar algún trait como:

```
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\RefreshDatabase;
```

Que permiten ejecutar las migraciones:

```
$ php artisan migrate
```

O

```
$ php artisan migrate:refresh
```

De manera programática (puedes consultar la definición de estas clases para más detalle).

Y alguna de estas, debemos de anexar a la prueba unitaria, por ejemplo:

```
// use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\RefreshDatabase;
***
class CategoryTest extends TestCase
{
    use RefreshDatabase;
    // use DatabaseMigrations;
}
```

Con este cambio, debemos de adaptar las pruebas anteriores para generar las tablas y los datos de prueba quedando como:

tests\Feature\Dashboard\CategoryTest.php

```
// use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\RefreshDatabase;
***
class CategoryTest extends TestCase
{
    use DatabaseMigrations;

    public User $user;
    protected function setUp(): void
    {
        parent::setUp();

        User::factory(1)->create();
        $this->user = User::first();
        $this->actingAs($this->user);
    }

    public function test_index(): void
    {
        Category::factory(2)->create();

        $categories = Category::paginate(2);
        $category = Category::first();
        ***
    }
***
```

```
|}
```

tests\Feature\BlogTest.php

```
use Illuminate\Foundation\Testing\DatabaseMigrations;
***
class BlogTest extends TestCase
{
    use DatabaseMigrations;

    public function test_index(): void
    {

        Category::factory(2)->create();
        $category = Category::first();
        ***
    }
    public function test_show(): void
    {

        Category::factory(3)->create();
        Post::factory(10)->create();
        ***
    }
}
```

El Factory de los posts:

database\factories\PostFactory.php

```
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class PostFactory extends Factory
{

    public function definition(): array
    {
        $name = $this->faker->name();
        return [
            'title' => $name,
            'slug' => str($name)->slug(),
            'description' => $this->faker->paragraph(1),
        ];
    }
}
```

```

        'text' => $this->faker->paragraph(4),
        'category_id' => $this->faker->randomElement([1,2,3]),
        'posted' => $this->faker->randomElement(['yes','not']),
        'image' => $this->faker->imageUrl(),
    ];
}
}

```

Y las categorías:

database\factories\CategoryFactory.php

```

<?php

namespace Database\Factories;
use Illuminate\Database\Eloquent\Factories\Factory;

class CategoryFactory extends Factory
{

    public function definition(): array
    {
        // $name = $this->faker->name();
        $name = $this->faker->sentence;
        return [
            'title' => $name,
            'slug' => str($name)->slug(),
        ];
    }
}

```

Autenticación

Las rutas del dashboard requieren de autenticación requerida, por tal motivo, debemos de autenticar el usuario para evitar que, al momento de implementar la prueba, de un error de redirección al momento de intentar ingresar al dashboard desde la prueba como:

```
| Failed asserting that 302 is identical to 200.
```

También, puedes comentar los middlewares desde las rutas que son las que bloquean las rutas con autenticación requerida de la siguiente forma:

routes\web.php

```

Route::middleware(
    [
        // 'auth:sanctum',
        // config('jetstream.auth_session'),
        // 'verified',
    ]
)
->prefix('dashboard')->group(function () {

```

Para hacerlo de la manera correcta, en cada prueba del dashboard, utilizamos el método **setUp()** que se ejecuta antes de cada prueba para inicializar la autenticación:

tests\Feature\Dashboard\CategoryTest.php

```

/**
class CategoryTest extends TestCase
{
    use DatabaseMigrations;

    public User $user;
    protected function setUp(): void
    {
        parent::setUp();

        User::factory(1)->create();
        $this->user = User::first();
        $this->actingAs($this->user);
    }
}

```

De esta forma, generamos un usuario aleatorio mediante el Factory de usuario que viene provisto al crear un proyecto en Laravel y lo autenticamos mediante **actingAs()**.

Dashboard

En este apartado, vamos a crear las pruebas para el dashboard comenzando por el de la categoría:

Categoría

Creamos la prueba:

```
$ php artisan make:test Dashboard\CategoryTest
```

Listado

La prueba del listado de las categorías queda como:

tests\Feature\Dashboard\CategoryTest.php

```
<?php

namespace Tests\Feature\Dashboard;

use App\Models\Category;
use App\Models\User;
use Illuminate\Foundation\Testing\DatabaseMigrations;
// use Illuminate\Foundation\Testing\RefreshDatabase;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

use Inertia\Testing\AssertableInertia as Assert;

class CategoryTest extends TestCase
{

    use DatabaseMigrations;

    public User $user;
    protected function setUp(): void
    {
        parent::setUp();

        User::factory(1)->create();
        $this->user = User::first();
        $this->actingAs($this->user);
    }

    /**
     * A basic feature test example.
     */
    public function test_index(): void
    {

        Category::factory(2)->create();

        $categories = Category::paginate(2);
        $category = Category::first();

        $this->get(route('category.index'))
            ->assertStatus(200)
            ->assertInertia(

```

```

        fn(Assert $page) => $page
            ->component('Dashboard/Category/Index')
            ->has('categories')
            ->has('categories.data.0.title')
            // ->where('categories.data.0.title', 'Cate 1')
            ->where('categories.data.0.title', $category->title)
            ->where('categories', Category::paginate(2))
    );
}
public function test_create_get(): void
{

    $this->get(route('category.create'))
        ->assertOk()
        ->assertInertia(
            fn(Assert $page) => $page
                ->component('Dashboard/Category/Create')
            );
}
}

```

Como punto importante, colocamos el método de:

```
Category::paginate(2)
```

Dentro de la asercción al momento de definir la ruta, para que Laravel pueda configurar las rutas internas de la paginación y evitar errores como los siguientes al momento de ejecutar la prueba:

```

--- Expected
+++ Actual
@@ @@
    'updated_at' => '2024-08-15T10:08:19.000000Z',
],
],
-
'first_page_url' => 'http://inertiastore.test?page=1',
+
'first_page_url' => 'http://inertiastore.test/dashboard/category?page=1',
'from' => 1,
'last_page' => 1,
-
'last_page_url' => 'http://inertiastore.test?page=1',
+
'last_page_url' => 'http://inertiastore.test/dashboard/category?page=1',

```

Cosa que sucedería si se construye la paginación antes de configurar la ruta.

Creación

El proceso de creación queda como:

```
tests\Feature\Dashboard\CategoryTest.php
```

```
class CategoryTest extends TestCase
{
    /**
     * @test
     */
    public function test_create_get(): void
    {
        $this->get(route('category.create'))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => $page
                    ->component('Dashboard/Category/Create')
            );
    }
}
```

Variamos el método de aserción **assertStatus(200)** a **assertOk()** para mostrar variantes entre los métodos de aserción, por lo demás, para probar la petición GET solamente verificamos que el componente sea el de crear; para la petición de tipo POST:

```
tests\Feature\Dashboard\CategoryTest.php
```

```
class CategoryTest extends TestCase
{
    /**
     * @test
     */
    public function test_create_post(): void
    {
        $data = [
            'title' => 'Title',
            'slug' => 'title',
        ];

        $this->post(route('category.store'), $data)
            ->assertRedirect(route('category.index'));

        $this->assertDatabaseHas('categories', $data);
    }
}
```

Actualización

El proceso de actualización es similar al anterior pero, empleando una categoría existente que es la que vamos a editar, así que, la generamos mediante el facade y componemos la URL y datos en base a la misma:

```
tests\Feature\Dashboard\CategoryTest.php
```

```

class CategoryTest extends TestCase
{
/**
 * @group category
 */
 public function test_edit_get(): void
 {
     $category = Category::factory(1)->create();
     $category = Category::first();

     $this->get(route('category.edit', $category))
         ->assertOk()
         ->assertInertia(
             fn(Assert $page) => $page
                 ->component('Dashboard/Category/Edit')
                 ->has('category')
                 ->where('category', $category)
             );
 }

 public function test_edit_post(): void
 {
     $category = Category::factory(1)->create();
     $category = Category::first();

     $data = [
         'title' => 'Title',
         'slug' => 'title',
     ];

     $this->put(route('category.update', $category), $data)
         ->assertRedirect(route('category.index'));

     $this->assertDatabaseHas('categories', $data);
     $this->assertDatabaseMissing('categories', $category->toArray());
 }
}

```

Eliminar

La prueba de eliminar es similar a la de PUT, pero, llamando al método de destroy en su lugar:

tests\Feature\Dashboard\CategoryTest.php

```

class CategoryTest extends TestCase
{
/**
 * @group category
 */
 public function test_delete_post(): void
 {
}
}

```

```

    {
        Category::factory(1)->create();
        $category = Category::first();

        $this->delete(route('category.destroy', $category))
            ->assertRedirect(route('category.index'));

        $this->assertDatabaseMissing('categories', $category->toArray());
    }
}

```

Post

Las pruebas iniciales para el módulo de dashboard para los posts, es similar a las creadas en las categorias, así que, inicialmente, partiremos de la estructura realizada anteriormente pero adaptándola para los posts:

tests\Feature\Dashboard\PostTest.php

```

<?php

namespace Tests\Feature\Dashboard;

use App\Models\Category;
use App\Models\Post;
use App\Models\User;
use Carbon\Carbon;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Illuminate\Foundation\Testing\WithFaker;
use Tests\TestCase;

use Inertia\Testing\AssertableInertia as Assert;

class PostTest extends TestCase
{
    use DatabaseMigrations;

    public User $user;
    protected function setUp(): void
    {
        parent::setUp();

        User::factory(1)->create();
        $this->user = User::first();
        $this->actingAs($this->user);
    }
}

```

```

public function test_index(): void
{
    Category::factory(3)->create();
    Post::factory(15)->create();

    $post = Post::orderBy('id','desc')->first();

    $this->get(route('post.index'))
        ->assertStatus(200)
        ->assertInertia(
            fn(Assert $page) => $page
                ->component('Dashboard/Post/Index')
                ->has('posts')
                ->has('posts.data.0.title')
                // ->where('posts.data.0.title', 'Cate 1')
                ->where('posts.data.0.title', $post->title)
                ->where('posts.data.0.posted', $post->posted)
                ->where('posts',
Post::with('category')->orderBy('id','desc')->paginate(15))
        );
}

public function test_create_get(): void
{

    $this->get(route('post.create'))
        ->assertOk()
        ->assertInertia(
            fn(Assert $page) => $page
                ->component('Dashboard/Post/Save')
        );
}

public function test_create_post(): void
{
    Category::factory(1)->create();

    $data = [
        'title' => 'Title',
        'slug' => 'title',
        'description' => 'Description',
        'posted' => 'yes',
        'text' => 'Content',
        'type' => 'post',
        'category_id' => 1,
        'date' => Carbon::now()
    ];
}

```

```

    $this->post(route('post.store'), $data)
        ->assertRedirect(route('post.index')));

    $this->assertDatabaseHas('posts', $data);
}

public function test_edit_get(): void
{
    Category::factory(3)->create();
    Post::factory(1)->create();
    $post = Post::first();

    $this->get(route('post.edit', $post))
        ->assertOk()
        ->assertInertia(
            fn(Assert $page) => $page
                ->component('Dashboard/Post/Save')
                ->has('post')
                ->where('post', $post)
        );
}

public function test_edit_post(): void
{
    Category::factory(3)->create();
    Post::factory(1)->create();

    $post = Post::first();

    $data = [
        'title' => 'Title',
        'slug' => 'title',
        'description' => 'Description',
        'posted' => 'yes',
        'text' => 'Content',
        'type' => 'post',
        'category_id' => 1,
        'date' => Carbon::now()
    ];

    $this->put(route('post.update', $post), $data)
        ->assertRedirect(route('post.index'));

    $this->assertDatabaseHas('posts', $data);
    $this->assertDatabaseMissing('posts', $post->toArray());
}

public function test_delete_post(): void

```

```

{
    Category::factory(3)->create();
    Post::factory(1)->create();
    $post = Post::first();

    $this->delete(route('post.destroy', $post))
        ->assertRedirect(route('post.index'));

    $this->assertDatabaseMissing('posts', $post->toArray());
}
}

```

Es importante acotar que la ordenación en el listado de post por defecto como lo tenemos en el controlador es:

app\Http\Controllers\Dashboard\PostController.php

```

$sortColumn = request('sortColumn') ?? 'id';
$sortDirection = request('sortDirection') ?? 'desc';

```

Por lo tanto, debemos de componer la consulta de la prueba para probar el index en base a esta ordenación como hicimos antes.

También es importante que en el controlador para crear y editar los posts el proceso del upload de la imagen que hacemos desde un método aparte se maneje de manera opcional, para evitar un error de validación y con esto, que fallen las pruebas:

app\Http\Controllers\Dashboard\PostController.php

```

public function store(Store $request)
{
    // dd($request['image']);
    $post = Post::create($request->validated());
    if (request('image')) // img opcional
        $this->upload($request, $post);
    return to_route('post.index')->with('message', 'Record Created!');
}

```

```

public function update(Put $request, Post $post)
{
    $post->update($request->validated());
    if (request('image')) // img opcional
        $this->upload($request, $post);
    return to_route('post.index')->with('message', 'Record Updated!');
}

```

Finalmente, al ser la entidad de post más compleja que la de categorías, podemos agregar algunos pocos métodos de asercción por ejemplo en el index y en los formularios de crear y editar para preguntar por el listado de categorías:

```
***  
class PostTest extends TestCase  
{  
  
    ***  
    public function test_index(): void  
    {  
        ***  
        $this->get(route('post.index'))  
            ->assertStatus(200)  
            ->assertInertia(  
                fn(Assert $page) => $page  
                    ***  
                    ->where('posts.data.0.title', $post->title)  
                    ->where('posts.data.0.slug', $post->slug)  
                    ->where('posts.data.0.posted', $post->posted)  
                    ->where('posts.data.0.type', $post->type)  
                    ->where('posts', Post::with('category')->orderBy('id',  
'desc')->paginate(15))  
                );  
    }  
    public function test_create_get(): void  
{  
  
        $categories = Category::get();  
  
        $this->get(route('post.create'))  
            ->assertOk()  
            ->assertInertia(  
                fn(Assert $page) => $page  
                    ->component('Dashboard/Post/Save')  
                    ->has('categories')  
                    ->where('categories', $categories)  
            );  
    }  
  
    public function test_edit_get(): void  
{  
  
        Category::factory(3)->create();  
        Post::factory(1)->create();
```

```

$categorias = Category::get();
$post = Post::first();

$this->get(route('post.edit', $post))
    ->assertOk()
    ->assertInertia(
        fn(Assert $page) => $page
            ->component('Dashboard/Post/Save')
            ->has('post')
            ->has('categorias')
            ->where('categorias', $categorias)
            ->where('post', $post)
    );
}

***
```

}

Formulario paso por paso

En este apartado, crearemos las pruebas para el formulario paso por paso, evaluando las peticiones de tipo GET y POST para los pasos y propiedades como la de step.

Los Factories que vamos a emplear en las pruebas:

database\factories>ContactGeneralFactory.php

```

<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class ContactGeneralFactory extends Factory
{

    public function definition(): array
    {
        return [
            'subject' => $this->faker->sentence,
            'message' => $this->faker->paragraph(1),
            'type' => $this->faker->randomElement(['company', 'person']),
        ];
    }
}
```

}

database\factories>ContactCompanyFactory.php

```
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class ContactCompanyFactory extends Factory
{

    public function definition(): array
    {
        return [
            'name' => $this->faker->sentence,
            'identification' => $this->faker->sentence,
            'email' => $this->faker->email,
            'extra' => $this->faker->sentence,
            'choices' => $this->faker->randomElement(['advert', 'post', 'course', 'movie',
'other']),
            'contact_general_id' => 1
        ];
    }
}
```

database\factories>ContactPersonFactory.php

```
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class ContactPersonFactory extends Factory
{

    public function definition(): array
    {
        return [
            'name' => $this->faker->sentence,
            'surname' => $this->faker->sentence,
```

```
        'other' => $this->faker->sentence,
        'choices' => $this->faker->randomElement(['advert', 'post', 'course', 'movie',
'other']),
        'contact_general_id' => 1,
    ];
}
}
```

database\factories>ContactDetailFactory.php

```
<?php

namespace Database\Factories;

use Illuminate\Database\Eloquent\Factories\Factory;

class ContactDetailFactory extends Factory
{

    public function definition(): array
    {
        return [
            'extra' => $this->faker->sentence,
            'contact_general_id' => 1,
        ];
    }
}
```

Primer paso: General

Comencemos con el primer paso:

```
$ php artisan make:test Contact/GeneralTest
```

Cuyas pruebas son similares a las realizadas anteriormente para el post o categoría:

tests\Feature\Dashboard\PostTest.php

```
<?php

namespace Tests\Feature>Contact;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Tests\TestCase;
```

```

use Inertia\Testing\AssertableInertia as Assert;

class GeneralTest extends TestCase
{

    use DatabaseMigrations;

    /**
     * A basic feature test example.
     */
    public function test_create_get(): void
    {
        $this->get(route('contact-general.create'))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => $page
                    ->where('step', 1)
                    ->component('Contact/General/Step')
                );
    }

    public function test_create_post(): void
    {

        $data = [
            'subject' => 'Subject',
            'message' => 'Message',
            'type' => 'person'
        ];

        $this->post(route('contact-general.store'), $data)
            ->assertRedirect(route('contact-general.edit', ['contact_general' => 1]));

        $this->assertDatabaseHas('contact_generals', $data);
    }
}

```

Puntos importantes

- Como comprobación adicional, tenemos la de la propiedad de step, que en el primer paso tiene que ser 1.
- También, es la primera prueba que al crear el recurso se redirecciona a la vista de editar en base al registro creado, como no tenemos una referencia directa al contacto creado y no poder buscarlo en la base de datos a menos que queramos evaluar el proceso en dos partes (uno para crear el contacto y otro para evaluar la redirección) colocamos un 1 como la PK del contacto creado; recordemos que cada prueba se ejecuta con una base de datos de prueba completamente limpia (**DatabaseMigrations**), por lo tanto, siempre el contacto a crear, va a tener un identificador de 1, así que, de esta forma, evitamos complicar el proceso de evaluación dividiéndolo en dos procesos apartes.

Segundo paso: Company

Creamos la prueba:

```
$ php artisan make:test Contact/CompanyTest
```

Las pruebas son similares a la anterior:

```
<?php

namespace Tests\Feature>Contact;

use App\Models>ContactGeneral;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Tests\TestCase;

use Inertia\Testing\AssertableInertia as Assert;

class CompanyTest extends TestCase
{
    use DatabaseMigrations;

    public function test_create_get(): void
    {
        $this->get(route('contact-company.create'))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => $page
                    ->component('Contact/Company/Form')
            );
    }

    public function test_create_post(): void
    {

        ContactGeneral::factory(1)->create();
        // dd(ContactGeneral::find(1)->id);
        $data = [
            'name' => 'Name',
            'identification' => 'Identification',
            'email' => 'person@gmail.com',
            'extra' => 'Extra',
            'choices' => 'post',
            'contact_general_id' => 1,
        ];
        $this->post(route('contact-company.store'), $data)
    }
}
```

```

        ->assertRedirect(route('contact-general.edit', ['contact_general' => 1]));
        $this->assertDatabaseHas('contact_companies', $data);
    }
}

```

Segundo paso: Person

Creamos la prueba:

```
$ php artisan make:test Contact/PersonTest
```

Las pruebas son similares a la anterior:

```

<?php

namespace Tests\Feature>Contact;

use App\Models>ContactGeneral;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Tests\TestCase;

use Inertia\Testing\AssertableInertia as Assert;

class PersonTest extends TestCase
{
    use DatabaseMigrations;

    public function test_create_get(): void
    {
        $this->get(route('contact-person.create'))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => $page
                // ->where('step', 2.5)
                ->component('Contact/Person/Form')
            );
    }

    public function test_create_post(): void
    {

        ContactGeneral::factory(1)->create();
        // dd(ContactGeneral::find(1)->id);
        $data = [
            'name' => 'Name',
            'surname' => 'Surname',
            'other' => 'Extra',
        ];
    }
}

```

```

        'choices' => 'post',
        'contact_general_id' => 1,
    ];

    $this->post(route('contact-person.store'), $data)
        ->assertRedirect(route('contact-general.edit', ['contact_general' => 1]));
    $this->assertDatabaseHas('contact_persons', $data);
}
}

```

Tercer paso: Detail

Creamos la prueba:

```
$ php artisan make:test Contact/DetailTest
```

Las pruebas son similares a la anterior:

```

<?php

namespace Tests\Feature>Contact;

use App\Models>ContactGeneral;
use Illuminate\Foundation\Testing\DatabaseMigrations;
use Tests\TestCase;

use Inertia\Testing\AssertableInertia as Assert;

class DetailTest extends TestCase
{
    use DatabaseMigrations;

    public function test_create_get(): void
    {
        $this->get(route('contact-detail.create'))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => $page
                // ->where('step', 3)
                ->component('Contact/Detail/Form')
            );
    }

    public function test_create_post(): void
    {
        ContactGeneral::factory(1)->create();
    }
}

```

```

    $data = [
        'extra' => 'Extra',
        'contact_general_id' => 1,
    ];

    $this->post(route('contact-detail.store'), $data)
        ->assertRedirect(route('contact-general.edit', ['contact_general' => 1]));
    $this->assertDatabaseHas('contact_optionals', $data);
}
}

```

Editar

En este apartado, vamos a completar las pruebas realizadas anteriormente en cada uno de los pasos realizando las pruebas para la edición de cada uno de los pasos.

Muchas veces al implementar las pruebas, es necesario desde las pruebas similar de la mejor forma posible el funcionamiento del módulo que se quiere probar, el formulario paso por paso a diferencia del resto de los módulos, es un componente (paso uno) complejo, que consta de otros componentes (pasos dos y tres) que a su vez consta de otros componentes (componentes como Input, Button) y es este anidamiento una parte importante de las pruebas; de momento, hemos probado cada componente como una pieza independiente, pero, es necesario también probarlo como un todo y esto lo empezaremos a implementar (las pruebas) desde este apartado.

Primer paso: General

Comencemos con las pruebas para el contacto general.

Las pruebas para la petición de tipo GET para editar es similar a las anteriores:

```

<?php
/**
 * @group general
 */
class GeneralTest extends TestCase
{
    /**
     * @test
     */
    public function test_edit_get(): void
    {
        $this->test_create_post();

        $contactGeneral = ContactGeneral::first();

        $this->get(route('contact-general.edit', ['contact_general' => 1]))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => $page
                    ->component('Contact/General/Step')
            );
    }
}

```

```

        ->has('contactGeneral')
        ->where('contactGeneral', $contactGeneral)
    );
}

public function test_edit_post(): void
{
    $this->test_create_post();

    $contactGeneral = ContactGeneral::first();

    $data = [
        'subject' => 'Subject new',
        'message' => 'Message new',
        'type' => 'person'
    ];

    $this->put(route('contact-general.update', $contactGeneral), $data)
        ->assertRedirect(route('contact-general.edit', $contactGeneral));

    $this->assertDatabaseHas('contact_generals', $data);
    $this->assertDatabaseMissing('contact_generals', $contactGeneral->toArray());
}
}

```

Aquí la diferencia fundamental es que, llamamos al método de `$this->test_create_post()` para crear un contacto general, que es fundamental para poder editar el mismo, pero, no lo hacemos mediante el Factory, si no, mediante una prueba, y esto es fundamental para probar la integración de componentes en base a lo comentado antes y poder probar de manera efectiva la página anterior guardada en el historial; lamentablemente, no es posible verificar props como el de **step** que es controlado internamente desde el componente de paso por paso ya que el mismo no funciona igual que el historial, como veremos más adelante.

Segundo paso: Compañía

La prueba para el paso dos, para las compañías es similar al anterior:

```

***  

class CompanyTest extends TestCase  

{  

    ***  

    public function test_edit_get(): void  

    {  

        ContactGeneral::factory(1)->create();  

        ContactCompany::factory(1)->create();  

        $contactCompany = ContactCompany::first();  


```

```

$this->get(route('contact-company.edit', ['contact_company' => 1]))
    ->assertOk()
    ->assertInertia(
        fn(Assert $page) => dd($page)
            ->component('Contact/Company/Form')
            // ->where('step', $contactGeneral->type == 'person' ? 2.5 : 2)
            ->has('contactCompany')
            ->where('contactCompany', $contactCompany)
    );
}

public function test_edit_put(): void
{
    ContactGeneral::factory(1)->create();
    ContactCompany::factory(1)->create();

    $contactCompany = ContactCompany::first();

    $data = [
        'name' => 'Name new',
        'identification' => 'Identification new',
        'email' => 'person@gmail.com',
        'extra' => 'Extra',
        'choices' => 'post',
        'contact_general_id' => 1,
    ];

    $this->put(route('contact-company.update', $contactCompany), $data)
        ->assertRedirect(route('contact-general.edit', $contactCompany));

    $this->assertDatabaseHas('contact_companies', $data);
    $this->assertDatabaseMissing('contact_companies', $contactCompany->toArray());
}
}

```

Segundo paso: Persona

La prueba para el paso dos, para las personas es similar al anterior:

tests\Feature>Contact\PersonTest.php

```

/**
class PersonTest extends TestCase
{
    /**
     * @test
     */
    public function test_edit_get(): void
    {

```

```

ContactGeneral::factory(1)->create();
ContactPerson::factory(1)->create();

$contactPerson = ContactPerson::first();

$this->get(route('contact-person.edit', ['contact_person' => 1]))
    ->assertOk()
    ->assertInertia(
        fn(Assert $page) => $page
            ->component('Contact/Person/Form')
            ->has('contactPerson')
            ->where('contactPerson', $contactPerson)
    );
}

public function test_edit_put(): void
{
    ContactGeneral::factory(1)->create();
    ContactPerson::factory(1)->create();

    $contactPerson = ContactPerson::first();

    $data = [
        'name' => 'Name New',
        'surname' => 'Surname New',
        'other' => 'Extra New',
        'choices' => 'post',
        'contact_general_id' => 1,
    ];

    $this->put(route('contact-person.update', $contactPerson), $data)
        ->assertRedirect(route('contact-general.edit', $contactPerson));

    $this->assertDatabaseHas('contact_persons', $data);
    $this->assertDatabaseMissing('contact_persons', $contactPerson->toArray());
}
}

```

Tercer paso: Detalle

La prueba para el paso tres, es similar al anterior:

tests\Feature>Contact\DetailTest.php

```
/**
 * @group contact
 */
class DetailTest extends TestCase
```

```

{
    /**
     * @test
     */
    public function test_edit_get(): void
    {
        ContactGeneral::factory(1)->create();
        ContactDetail::factory(1)->create();

        $contactDetail = ContactDetail::first();

        $this->get(route('contact-detail.edit', ['contact_detail' => 1]))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => $page
                    ->component('Contact/Detail/Form')
                    ->has('contactDetail')
                    ->where('contactDetail', $contactDetail)
            );
    }

    public function test_edit_put(): void
    {
        ContactGeneral::factory(1)->create();
        ContactDetail::factory(1)->create();

        $contactDetail = ContactDetail::first();

        $data = [
            'extra' => 'Extra New',
            'contact_general_id' => 1,
        ];

        $this->put(route('contact-detail.update', $contactDetail), $data)
            ->assertRedirect(route('contact-general.edit', $contactDetail));

        $this->assertDatabaseHas('contact_optionals', $data);
        $this->assertDatabaseMissing('contact_optionals', $contactDetail->toArray());
    }
}

```

Problemas revisando los datos compartidos/shared-data de Inertia

Una de las propiedades compartidas que son claves en el paso por paso, es la de **step**, la cual variamos según el paso en donde nos encontramos en el formulario y es un buen candidato para implementar pruebas; al igual que hacemos con Laravel básico, deberíamos de realizar peticiones a la ruta que se encarga de actualizar el step, que en nuestro caso es la de edición del paso uno:

```
|$this->get(route('contact-general.edit', ['contact_general' => 1]));
```

En Laravel básico, hacemos un proceso similar cuando validamos los errores de validación, que si antes no enviamos una petición a la página que pinta el formulario:

```
$this->get(route('post.edit', $post));
$data = [***];
$this->put(route('post.update', $post), $data)
    ->assertRedirect(route('post.edit', $post))
```

La ruta a la cual regresa sería la anterior, que por defecto sería la ruta /:

```
$this->get(route('post.edit', $post));
$data = [
    ***
];
$this->post(route('post.store', $data))
->assertRedirect('/')
```

Lamentablemente **no** podemos hacer lo mismo con el Shared Data de Inertia:

```
/**
class CompanyTest extends TestCase
{
    /**
     * @test
     */
    public function test_edit_get(): void
    {
        ContactGeneral::factory(1)->create();
        ContactCompany::factory(1)->create();

        $contactCompany = ContactCompany::first();

        $this->get(route('contact-general.edit', ['contact_general' => 1]));

        $this->get(route('contact-company.edit', ['contact_company' => 1]))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => dd($page)
                    ->component('Contact/Company/Form')
                    ->where('step', $contactGeneral->type == 'person' ? 2.5 : 2)
                    ***
            );
    }
}
```

```
| ->where('step', $contactGeneral->type == 'person' ? 2.5 : 2)
```

Al ejecutar la prueba anterior, la misma fallaría en la condición de where indicando que step tiene el valor de uno.

Errores de validaciones

De momento, hemos probado los casos que devuelven la respuesta esperada, es decir, que se crea o edita el contacto, pero, en las pruebas debemos de cubrir otros casos como lo son los errores de validaciones que son los que se van a cubrir en esta sección.

Es importante aclarar que las pruebas a realizar por controlador son las mínimas necesarias y el lector puede crear más pruebas para cubrir los errores de validaciones que considere necesarias.

Primer paso: General

Las pruebas para el contacto general son iguales a las cubiertas anteriormente pero, pasando data invalida como lo es, datos vacíos, mediante el método de aserción **assertSessionHasErrors()** verifica que los errores de validación se encuentren en la sesión flash:

```
***  
class GeneralTest extends TestCase  
{  
  
    ***  
    public function test_create_post_invalid(): void  
    {  
  
        // para tener una respuesta en el back del historial  
        $this->get(route('contact-general.create'));  
  
        $dataInvalid = [  
            'subject' => '',  
            'message' => '',  
            'type' => ''  
        ];  
  
        $this->post(route('contact-general.store'), $dataInvalid)  
            ->assertSessionHasErrors([  
                'subject' => 'The subject field is required.',  
                'message' => 'The message field is required.',  
                'type' => 'The type field is required.',  
            ])  
            ->assertRedirect(route('contact-general.create'));  
    }  
  
    public function test_edit_put_invalid(): void  
    {  
        ContactGeneral::factory(1)->create();
```

```

$contactGeneral = ContactGeneral::first();

$dataInvalid = [
    'subject' => '',
    'message' => '',
    'type' => ''
];

$this->put(route('contact-general.update', $contactGeneral), $dataInvalid)
    ->assertSessionHasErrors([
        'subject' => 'The subject field is required.',
        'message' => 'The message field is required.',
        'type' => 'The type field is required.',
    ])
    ->assertRedirect('/');

$this->assertDatabaseMissing('contact_generals', $dataInvalid);
}

}

```

Adicionalmente puedes ver que al ocurrir un error de validación, Laravel hace un **back()** para la página anterior, que usualmente es la página del formulario (la que obtenemos mediante una petición Get), pero, en las pruebas como son independientes, no existe una página anterior a la cual regresar, por lo tanto, si no hacemos una petición de manera manual a la página anterior, regresa a la raíz:

```
->assertRedirect('/');
```

O hacemos una petición antes de hacer la la petición de POST/PUT para tener una página anterior en el historial a la cual regresar:

```

$this->get(route('contact-general.create'));
***  

->assertRedirect(route('contact-general.create'));

```

Segundo paso: Compañía

Las pruebas para la compañía quedan como:

tests\Feature>Contact\CompanyTest.php

```

***  

class CompanyTest extends TestCase
{
    ***  

    // *** invalid form  

    public function test_create_post_invalid(): void

```

```

{

ContactGeneral::factory(1)->create();

$dataInvalid = [
    'name' => '',
    'identification' => '',
    'email' => '',
    'extra' => '',
    // 'choices' => '',
    // 'contact_general_id' => '',
];
}

$this->post(route('contact-company.store'), $dataInvalid)
->assertSessionHasErrors([
    'name' => 'The name field is required.',
    'identification' => 'The identification field is required.',
    'email' => 'The email field is required.',
    'choices' => 'The choices field is required.',
    'extra' => 'The extra field is required.',
    'contact_general_id' => 'The contact general id field is required.',
])
->assertRedirect('/');

}

public function test_edit_put_invalid(): void
{
    ContactGeneral::factory(1)->create();
    ContactCompany::factory(1)->create();

    $contactCompany = ContactCompany::first();

    $dataInvalid = [
        'name' => '',
        'identification' => '',
        'email' => 'andres',
        'extra' => '',
        // 'choices' => '',
    ];
}

```

```

        // 'contact_general_id' => '',
    ];

$this->put(route('contact-company.update', $contactCompany), $dataInvalid)
->assertSessionHasErrors([
    'name' => 'The name field is required.',
    'identification' => 'The identification field is required.',
    'email' => 'The email field must be a valid email address.',
    'choices' => 'The choices field is required.',
    'extra' => 'The extra field is required.',
    'contact_general_id' => 'The contact general id field is required.',
])
->assertRedirect('/');

$this->assertDatabaseMissing('contact_companies', $dataInvalid);
}
}

```

Segundo paso: Persona

Las pruebas para la persona quedan como:

tests\Feature>Contact\PersonTest.php

```

***  

class PersonTest extends TestCase  

{  

    ***  

    // invalid  

    public function test_create_post_invalid(): void  

    {  

        ContactGeneral::factory(1)->create();  

        $dataInvalid = [  

            'name' => '',  

            'surname' => '',  

            'other' => 'a',  

            // 'choices' => '',  

            // 'contact_general_id' => '',  

        ];  

        $this->post(route('contact-person.store'), $dataInvalid)
        ->assertSessionHasErrors([
            'name' => 'The name field is required.',  

            'surname' => 'The surname field is required.',  

        ]);
    }
}

```

```

        'other' => 'The other field must be at least 2 characters.',
        'choices' => 'The choices field is required.',
        'contact_general_id' => 'The contact general id field is required.',
    ])
->assertRedirect('/');
}

public function test_edit_put_invalid(): void
{
    ContactGeneral::factory(1)->create();
    ContactPerson::factory(1)->create();

    $contactPerson = ContactPerson::first();

    $dataInvalid = [
        'name' => '',
        'surname' => '',
        'other' => '',
        // 'choices' => '',
        // 'contact_general_id' => '',
    ];
}

$this->put(route('contact-person.update', $contactPerson), $dataInvalid)
->assertSessionHasErrors([
    'name' => 'The name field is required.',
    'surname' => 'The surname field is required.',
    'other' => 'The other field is required.',
    'choices' => 'The choices field is required.',
    'contact_general_id' => 'The contact general id field is required.',
])
->assertRedirect('/');
}
}

```

Tercer paso: Detalle

Las pruebas para la de detalle quedan como:

tests\Feature>Contact\DetailTest.php

```
/**
class DetailTest extends TestCase
{
    /**
     * @dataProvider invalidProvider
     */
    public function test_create_post_invalid(): void
    {
        $invalidData = [
            'name' => '',
            'surname' => '',
            'other' => '',
            'choices' => '',
            'contact_general_id' => ''
        ];
        $this->post(route('contact-person.create'), $invalidData)
            ->assertSessionHasErrors([
                'name' => 'The name field is required.',
                'surname' => 'The surname field is required.',
                'other' => 'The other field is required.',
                'choices' => 'The choices field is required.',
                'contact_general_id' => 'The contact general id field is required.'
            ])
            ->assertRedirect('/');
    }
}
```

```

ContactGeneral::factory(1)->create();

$dataInvalid = [
    'extra' => '',
    // 'contact_general_id' => 1,
];
}

$this->post(route('contact-detail.store'), $dataInvalid)
->assertSessionHasErrors([
    'extra' => 'The extra field is required.',
    'contact_general_id' => 'The contact general id field is required.',
]);
}

public function test_edit_put_invalid(): void
{
    ContactGeneral::factory(1)->create();
    ContactDetail::factory(1)->create();

    $contactDetail = ContactDetail::first();

    $dataInvalid = [
        'extra' => 'a',
        // 'contact_general_id' => 1,
    ];
}

$this->put(route('contact-detail.update', $contactDetail), $dataInvalid)
->assertSessionHasErrors([
    'extra' => 'The extra field must be at least 2 characters.',
    'contact_general_id' => 'The contact general id field is required.',
]);
}
}

```

Carrito

En este apartado, vamos a crear las pruebas CRUD para el carrito de compras:

```
$ php artisan make:test Shop/CartTest
```

Y las pruebas para el listado, agregar, modificar y eliminar ítems del carrito en la sesión y base de datos quedan como:

tests\Feature\Shop\CartTest.php

```

<?php

namespace Tests\Feature\Shop;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Inertia\Testing\AssertableInertia as Assert;

use Tests\TestCase;

use App\Models\Category;
use App\Models\Post;
use App\Models\ShoppingCart;
use App\Models\User;

class CartTest extends TestCase
{

    use DatabaseMigrations;

    protected function setUp(): void
    {
        parent::setUp();

        User::factory(1)->create();
        $this->actingAs(User::first());
    }

    /**
     * A basic feature test example.
     */
    public function test_index_get(): void
    {
        $this->get(route('shop.index'))
            ->assertOk()
            ->assertInertia(
                fn(Assert $page) => $page
                    ->component('Shop/Index')
            );
    }

    public function test_create_db_session_post(): void
    {
        Category::factory(3)->create();
        Post::factory(1)->create();

        $this->post(route('shop.add', ['post' => 1, 'count' => 1]))
    }
}

```

```

->assertRedirect('/');

// session('cart')[<PK>] = [<POST>, <COUNT>]
$this->assertEquals(1, session('cart')[1][0]->id);
$this->assertEquals(1, session('cart')[1][1]);

$this->assertDatabaseHas('shopping_carts', [
    'user_id' => 1,
    'post_id' => 1,
    'count' => 1,
]);
}

public function test_update_count_db_session_post(): void
{

    $this->test_create_db_session_post();

    $this->post(route('shop.add', ['post' => 1, 'count' => 2]))
        ->assertRedirect('/');

    // session('cart')[<PK>] = [<POST>, <COUNT>]
    $this->assertEquals(1, session('cart')[1][0]->id);
    $this->assertEquals(2, session('cart')[1][1]);

    $this->assertDatabaseMissing('shopping_carts', [
        'user_id' => 1,
        'post_id' => 1,
        'count' => 1,
    ]);
}

$this->assertDatabaseHas('shopping_carts', [
    'user_id' => 1,
    'post_id' => 1,
    'count' => 2,
]);
}

}

public function test_delete_count_db_session_post(): void
{
    $this->test_create_db_session_post();
    sleep(1);
    $this->post(route('shop.add', ['post' => 1, 'count' => 0]))
        ->assertRedirect('/');

    // dd(ShoppingCart::get());
}

```

```

    $this->assertEquals([], session('cart'));

    $this->assertDatabaseMissing('shopping_carts', [
        'user_id' => 1,
        'post_id' => 1,
        // 'count' => 1,
    ]);
}

public function test_delete_count_db_session_item_not_exist_post(): void
{
    // $this->test_create_db_session_post();

    Category::factory(3)->create();
    Post::factory(1)->create();
    $this->post(route('shop.add', ['post' => 1, 'count' => 0]))->assertRedirect('/');

    $this->assertNull(session('cart'));
}
}

```

Como puntos importantes tenemos, desde el método de **test_update_count_db_session_post()** que verifica si se actualizo el ítem del carrito en la base de datos y sesión, para generar el ítem no empleamos un Factory, que sería un esquema válido, si no, el proceso previo de agregar un ítem al carrito **test_create_db_session_post()**.

Para verificar la sesión, accedemos a la sesión como si fuera desde un controlador en Laravel:

```
session('cart')[1][0]->id
```

Y verificamos en base a nuestra lógica:

```
// session('cart')[<PK>] = [<POST>, <COUNT>]
$this->assertEquals(1, session('cart')[1][0]->id);
$this->assertEquals(2, session('cart')[1][1]);
```

Para la eliminación se hacen dos pruebas:

1. function **test_delete_count_db_session_item_not_exist_post()**
2. function **test_delete_count_db_session_post()**

Que verifican sobre un ítem existente y que no existe en el carrito, en ambos casos la sesión debe ser nula o simplemente un array vacío.

En la prueba de function **test_delete_count_db_session_post()** que primero agrega un ítem al carrito y luego se elimina, se emplea la función de **sleep()** para no hacer la operación tan rápido y que luego exista problemas al momento de eliminar los ítems no actualizados desde el controlador:

app\Http\Controllers\Shop\CartController.php

```
| ShoppingCart::whereNot('control', $control)->where('user_id', auth()->id())->delete();
```

Otro punto importante es que, el carrito de compra se puede emplear con autenticación y sin ella, en el caso de que el usuario no esté autenticado, solamente se emplea la sesión; se pueden crear pruebas específicas para cuando el usuario no está autenticado y que no fueron implementadas en el libro al igual que muchas otras para evitar colocar mucho código repetido y aburrir al lector, pero es importante aclarar que estas pruebas son solamente algunas pocas de las que puedes crear para la aplicación.

To Do

En este apartado, crearemos las pruebas para los To Do que recordemos que el usuario tiene que estar autenticado para poder gestionar los To Do:

```
| $ php artisan make:test Todo/TodoTest
```

Y las pruebas para el listado, agregar, modificar, eliminar, reordenar y validar errores de validaciones del To Do quedan como:

tests\Feature\Todo\TodoTest.php

```
<?php

namespace Tests\Feature\Todo;

use Illuminate\Foundation\Testing\DatabaseMigrations;
use Tests\TestCase;

use App\Models\User;
use App\Models\Todo;

use Inertia\Testing\AssertableInertia as Assert;

class TodoTest extends TestCase
{
    use DatabaseMigrations;

    public User $user;
    protected function setUp(): void
    {
        parent::setUp();

        User::factory(1)->create();
        $this->user = User::first();
        $this->actingAs($this->user);
    }
}
```

```

public function test_index(): void
{
    Todo::factory(5)->create();

    $todos = Todo::where('user_id', $this->user->id)->orderBy('count')->get();

    $this->get(route('todo.index'))
        ->assertStatus(200)
        ->assertInertia(
            fn(Assert $page) => $page
                ->component('Todo/Index')
                ->has('todos')
                ->has('todos.0.name')
                ->where('todos.0.name', $todos[0]->name)
                ->where('todos', $todos)
        );
}

public function test_create_post(): void
{
    $data = [
        'name' => 'Name',
    ];

    $this->post(route('todo.store'), $data)
        ->assertRedirect(route('todo.index'));

    $this->assertDatabaseHas('todos', $data);
}

public function test_edit_post(): void
{
    Todo::factory(1)->create();
    $todo = Todo::first();

    $data = [
        'name' => 'Name',
    ];

    $this->put(route('todo.update', $todo), $data)
        ->assertRedirect(route('todo.index'));

    $this->assertDatabaseHas('todos', $data);
    $this->assertDatabaseMissing('todos', $todo->toArray());
}

public function test_delete_post(): void

```

```

{
    Todo::factory(1)->create();
    $todo = Todo::first();

    $this->delete(route('todo.destroy', $todo))
        ->assertRedirect(route('todo.index'));

    $this->assertDatabaseMissing('todos', $todo->toArray());
}

public function test_status_completed_post(): void
{
    Todo::factory(1)->create();
    $todo = Todo::first();

    $data = [
        'status' => 1,
    ];

    $this->post(route('todo.status', $todo), $data)
        ->assertRedirect(route('todo.index'));

    $this->assertDatabaseHas('todos', $data);
    $this->assertDatabaseMissing('todos', $todo->toArray());
}

public function test_status_uncompleted_post(): void
{
    $this->test_status_completed_post();
    $todo = Todo::first();

    $data = [
        'status' => 0,
    ];

    $this->post(route('todo.status', $todo), $data)
        ->assertRedirect(route('todo.index'));

    // $todoNew = Todo::select('user_id', 'status', 'count', 'name')->first();

    // $this->assertDatabaseHas('todos', $todoNew->toArray());
    $this->assertDatabaseHas('todos', $data);
    $this->assertDatabaseMissing('todos', $todo->toArray());
}

public function test_reorder(): void

```

```

{
    Todo::factory(5)->create();
    // $todosNoOrder = Todo::pluck('id');
    $todosReOrder = [2, 4, 1, 3, 5];

    $this->post(route('todo.order'), ['ids' => $todosReOrder]);

    $todosOrdered = Todo::orderBy('count')->pluck('id');

    foreach ($todosOrdered as $key => $id) {
        $this->assertTrue($id == $todosReOrder[$key]);
    }
}

public function test_create_error_validation_post(): void
{

    $dataInvalid = [
        'name' => '',
    ];

    $this->post(route('todo.store'), $dataInvalid)->assertSessionHasErrors(
        [
            'name' => 'The name field is required.'
        ]
    );

    $this->assertDatabaseMissing('todos', $dataInvalid);
}
}

```

Es importante acotar que no tenemos vistas GET para el proceso de crear y actualizar los todo, por lo tanto, no hay pruebas para estos procesos, también, en el código anterior, vemos pruebas para otros casos como errores de validación, reordenar un todo y el estatus.

En las pruebas, al momento de hacer las operaciones anteriores, establecimos en el controlador que al momento de hacer las operaciones redireccione a la raíz:

app\Http\Controllers\TodoController.php

```
// return back();
return redirect(route('todo.index'));
```

En vez de hacer un **back()**:

app\Http\Controllers\TodoController.php

```
return back();
```

Ya que, si dejamos el back al momento de hacer las pruebas no existe la página anterior y redirecciona a la raíz.

Para la prueba del status del To Do, creamos dos pruebas, una para completar un To Do, y otra para pasar de un To Do completado a no completado (**test_status_uncompleted_post()**) esta última, utilizamos la prueba anterior (la que verifica el To Do como completado) para emplear un To Do con estatus completado (en el factory todos los To Do generados tiene estatus no completado) para luego pasarlo a no completado.

Para la prueba de **test_reorder()**, generamos un listado aleatorio que queremos que tengan los To Do:

```
$todosReOrder = [2, 4, 1, 3, 5];
```

Estos son las PKs, que en nuestro ejemplo van de 1 a 5, y luego es enviado al controlador mediante una petición tipo POST, obtenemos nuevamente el listado por PKs ordenamos en base al array anterior:

```
Todo::orderBy('count')->pluck('id');
```

Y los comparamos:

```
foreach ($todosOrdered as $key => $id) {  
    $this->assertTrue($id == $todosReOrder[$key]);  
}
```

El Factory queda como:

database\factories\TodoFactory.php

```
<?php  
  
namespace Database\Factories;  
  
use Illuminate\Database\Eloquent\Factories\Factory;  
  
class TodoFactory extends Factory  
{  
  
    public function definition(): array  
    {  
        $name = $this->faker->name();  
        return [  
            'name' => $name,  
            'count' => 1,  
            'user_id' => 1,  
        ];  
    }  
}
```

Upload

En este apartado, veremos cómo realizar pruebas para el upload, para ello, usaremos el controlador de Post en la cual, implementamos un proceso de upload; los discos, que son usados para aplicar el upload, tienen sus propios métodos se aserción y fakers:

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

Storage::disk(<DISK>)->assertExists('other.png');
UploadedFile::fake()->image('image.png');
```

También puedes personalizar la imagen cargada:

```
UploadedFile::fake()->image('image.jpg', $width, $height)->size(100);
```

Método para preguntar si una imagen no está definida:

```
Storage::disk(<DISK>)->assertMissing('other.png');
```

Algunos métodos interesantes que puedes emplear para ir armando el path a la imagen y ver si es correcta:

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

dd(Storage::disk(<DISK>)->path(''));
dd(Storage::disk(<DISK>)->path($post->image));
```

Quedando las pruebas como:

```
tests\Feature\Dashboard\PostTest.php
```

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

public function test_edit_upload_put(): void
{
    Category::factory(3)->create();
    Post::factory(1)->create();

    $post = Post::first();

    $data = $post->toArray();
    $data['image'] = UploadedFile::fake()->image('image.png');

    $this->put(route('post.update', $post), $data)
        ->assertRedirect(route('post.index'));
```

```

$post = Post::first();

Storage::disk('public_upload')->assertExists('image\post\\' . $post->image);
// Storage::disk('public_upload')->assertMissing('other.png');

}

public function test_edit_delete_old_image_upload_put(): void
{
    sleep(1);

    Category::factory(3)->create();
    Post::factory(1)->create();

    //*** init image
    $postOldImg = Post::first();

    $data = $postOldImg->toArray();
    $data['image'] = UploadedFile::fake()->image('imageOld.png');

    $this->put(route('post.update', $postOldImg), $data)
        ->assertRedirect(route('post.index'));

    $postOldImg = Post::first();
    Storage::disk('public_upload')->assertExists('image\post\\' . $postOldImg->image);

    /** update image
    // $post = Post::first();

    sleep(1);

    $data['image'] = UploadedFile::fake()->image('imageNew.png');
    $this->put(route('post.update', $postOldImg), $data)
        ->assertRedirect(route('post.index'));

    // old image
    Storage::disk('public_upload')->assertMissing('image\post\\' . $postOldImg->image);

    // new image
    $postNewImg = Post::first();
    Storage::disk('public_upload')->assertExists('image\post\\' . $postNewImg->image);
}

public function test_create_upload_post(): void
{
    sleep(1);
}

```

```

Category::factory(1)->create();

$data = [
    'title' => 'Title',
    'slug' => 'title',
    'description' => 'Description',
    'posted' => 'yes',
    'text' => 'Content',
    'type' => 'post',
    'category_id' => 1,
    'date' => Carbon::now(),
    'image' => UploadedFile::fake()->image('image.png')
];

$this->post(route('post.store'), $data)
    ->assertRedirect(route('post.index'));

// actualizamos nombre de la img
$post = Post::first();
$data['image'] = $post->image;

$this->assertDatabaseHas('posts', $data);

Storage::disk('public_upload')->assertExists('image\post\\' . $post->image);
}

```

En estas prueba, la primera, se realiza el **test_edit_upload_put()** en la prueba de **test_edit_delete_old_image_upload_put()** verificamos que se se elimine la imagen anterior al cargar una nueva imagen, en la prueba de **test_create_upload_put()** probamos el upload en el proceso de crear un post.

Colocamos el **sleep()** para evitar que en las pruebas de upload se generan imágenes con el mismo nombre.

Código fuente del capítulo:

<https://github.com/libredesarrollo/book-course-laravel-inertia/releases/tag/v0.12>

Capítulo 18: Legacy - Características de un proyecto Inertia con Jetstream

Antes de comenzar, como recomendación, usa la siguiente documentación:

<https://jetstream.laravel.com/stacks/inertia.html>

Aquí encontrarás el detalle de todo lo que vamos a resumir en este capítulo y es la documentación que tienes que emplear para conocer qué es lo que trae el proyecto por defecto, su estructura y cómo puedes escalar el mismo.

Como puedes darte cuenta, aquí tenemos el nombre de "Jetstream":

Laravel Jetstream es un kit de inicio de aplicaciones diseñado para Laravel y proporciona el punto de partida perfecto para tu próxima aplicación de Laravel.

Jetstream proporciona la implementación para:

- El inicio de sesión.
- El registro.
- La verificación de correo electrónico.
- La autenticación de dos factores.
- La gestión de sesiones.
- La API a través de Laravel Sanctum.
- Las funciones opcionales de gestión de equipos de su aplicación.

Jetstream está diseñado con Tailwind CSS y ofrece la elección de Livewire o Inertia.

Entonces, puedes ver que Jetstream es el núcleo de Livewire o Inertia; en este libro, recordemos que nos interesa es aprender a desarrollar con Inertia, pero, si quieras obtener más información sobre Livewire:

<https://laravel-livewire.com/>

O mi libro en Laravel Livewire:

<https://www.desarrollolibre.net/libros/primeros-pasos-laravel-livewire>

Con los pasos realizados en el capítulo anterior, tenemos a nuestra disposición un proyecto completo en Laravel con el scaffolding de Inertia; Inertia nos trae una serie de módulos ya listos que vamos a conocer a continuación.

Laravel Fortify

Laravel Fortify es una implementación de backend de autenticación independiente de frontend para Laravel. Fortify registra las rutas y los controladores necesarios para implementar todas las funciones de autenticación de Laravel, incluido el inicio de sesión, el registro, el restablecimiento de contraseña, la verificación de correo electrónico y más.

Laravel Fortify esencialmente toma las rutas y los controladores de Laravel Breeze y los ofrece como un paquete que no incluye una interfaz de usuario.

El módulo de Laravel Fortify para la autenticación:

<http://localhost/login>

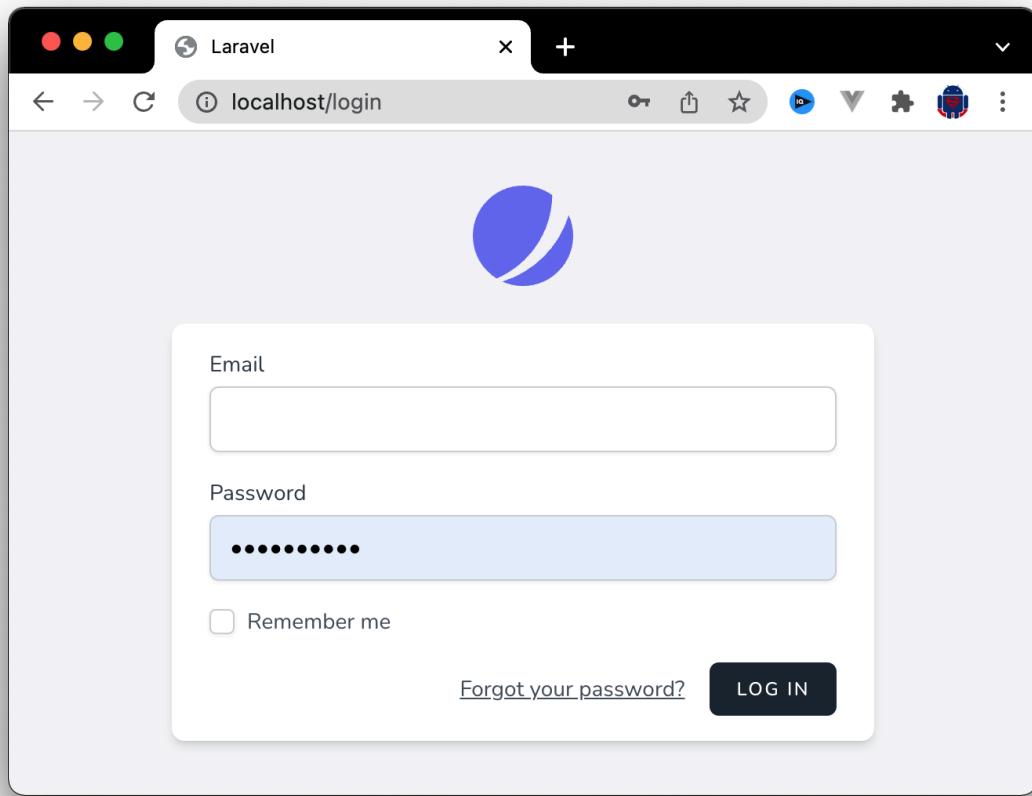


Figura 3-1: Ventana de login

Así como de registrarse; en el cual te debes de crear un usuario:

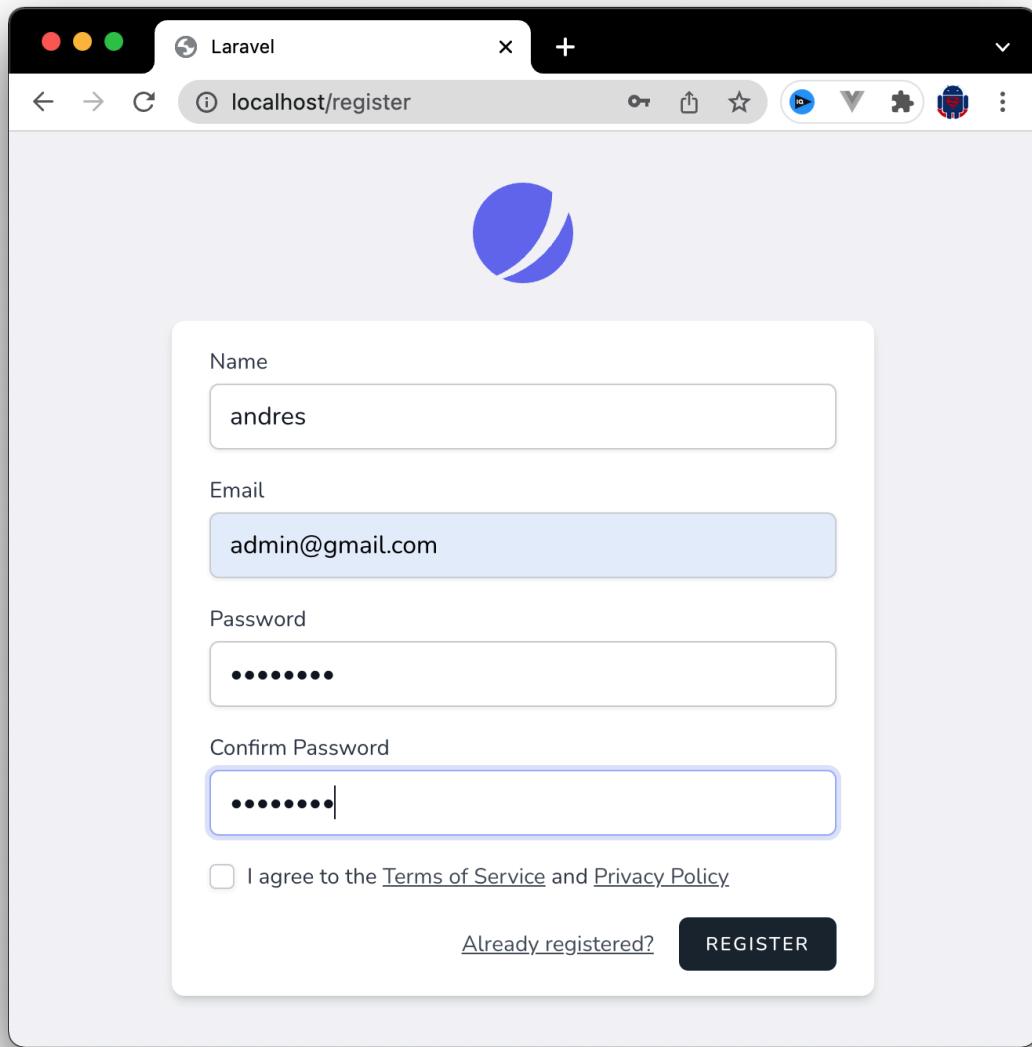


Figura 3-2:Registrar

Una vez autenticados; veremos la siguiente pantalla:

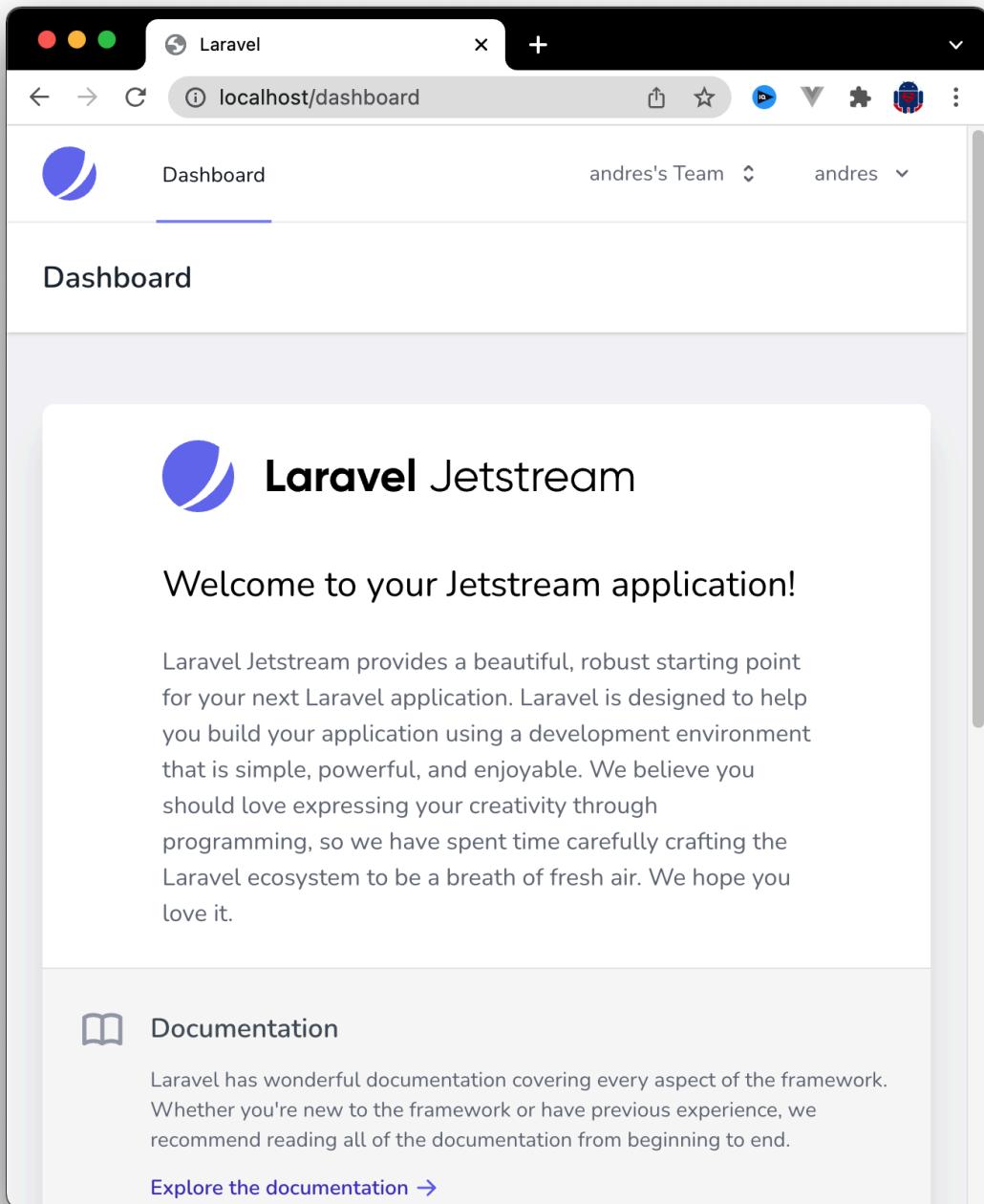


Figura 3-3: Dashboard

Si verificamos el archivo de las rutas:

routes/web.php

```
// ***
Route::middleware([
    'auth:sanctum',
    config('jetstream.auth_session'),
```

```

    'verified',
])->group(function () {
    Route::get('/dashboard', function () {
        return Inertia::render('Dashboard');
    })->name('dashboard');
});
// ***

```

Veremos que es un recurso protegido por la autenticación; y también, que no tenemos rutas adicionales salvo la de **welcome**; sin embargo; tenemos definidas unas series de opciones; como puedes ver el en menú:

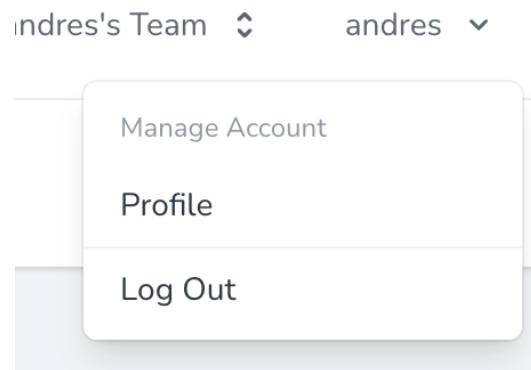


Figura 3-4: Opciones del usuario

Y

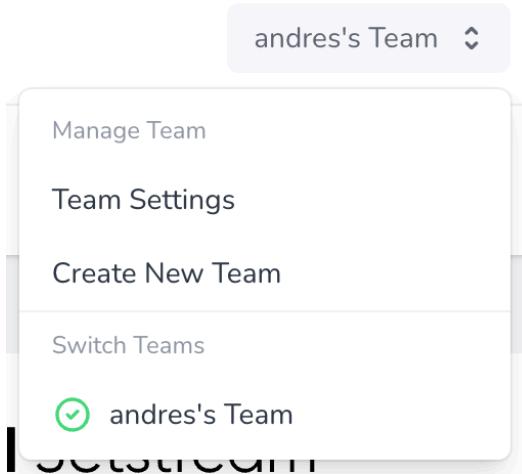


Figura 3-5: Teams

Opciones

Estas opciones nos permiten administrar el perfil del usuario y el manejo de los equipos; estos son parte de las funcionalidades que tenemos en Inertia por defecto; pero podemos activar más; si nos vamos a:

config/jetstream.php

```
'features' => [
    // Features::termsAndPrivacyPolicy(),
    // Features::profilePhotos(),
    // Features::api(),
    Features::teams(['invitations' => true]),
    Features::accountDeletion(),
],
```

Tenemos características para:

1. Términos y políticas.
2. Subir imagen al perfil.
3. Api Tokens con Laravel Sanctum.
4. Manejo de los equipos.
5. Borrar la cuenta.

Activa todas estas opciones:

```
'features' => [
    Features::termsAndPrivacyPolicy(),
    Features::profilePhotos(),
    Features::api(),
    Features::teams(['invitations' => true]),
    Features::accountDeletion(),
],
```

Y verás que automáticamente, tenemos algunos cambios; en la imagen de perfil, aparece un avatar o imagen de perfil:

Profile Information

Update your account's profile information and email address.

Photo



SELECT A NEW PHOTO

Name

Email

SAVE

Figura 3-6: Avatar

Y una opción para las Api Tokens con Sanctum:

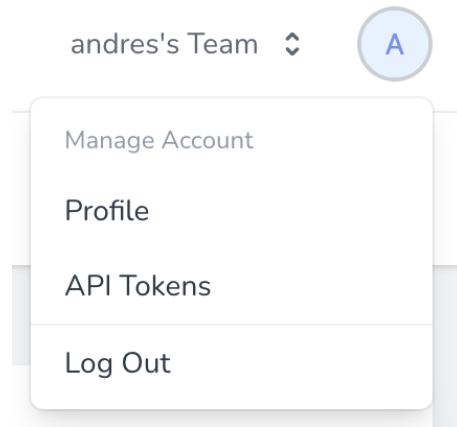
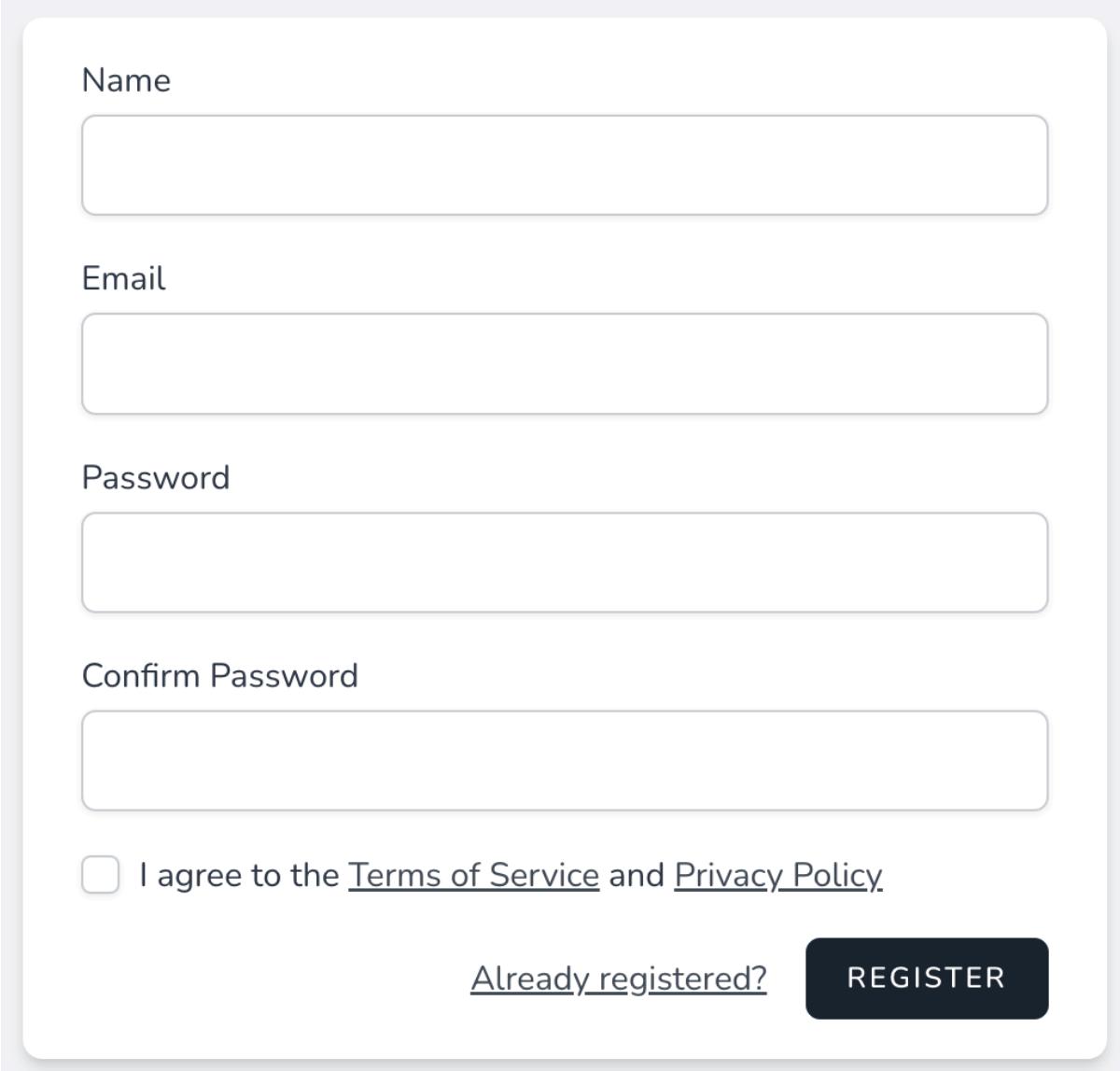


Figura 3-7: Api Tokens

El apartado de términos y servicios solamente aparece cuando vamos a registrar un usuario:



The image shows a registration form with the following fields and options:

- Name**: An input field for entering a name.
- Email**: An input field for entering an email address.
- Password**: An input field for entering a password.
- Confirm Password**: An input field for confirming the entered password.
- I agree to the Terms of Service and Privacy Policy.**: A checkbox followed by a descriptive text.
- Already registered?**: A link for users who have already registered.
- REGISTER**: A dark button with white text for submitting the form.

Figura 3-8: Registrar

Opciones en perfil

Si vamos a la página de perfil, verás que tenemos opciones para:

Cargar el avatar, gracias a la activación de la característica anterior y cambiar datos de usuarios:

Profile Information

Update your account's profile information and email address.

Photo



SELECT A NEW PHOTO

Name

Email

SAVE

Figura 3-9: Información del usuario

Actualizar la contraseña:

Update Password

Ensure your account is using a long, random password to stay secure.

Current Password

New Password

Confirm Password

SAVE

Figura 3-10: Actualizar contraseña

Activar la autenticación con doble factor:

Two Factor Authentication

Add additional security to your account using two factor authentication.

You have not enabled two factor authentication.

When two factor authentication is enabled, you will be prompted for a secure, random token during authentication. You may retrieve this token from your phone's Google Authenticator application.

ENABLE

Figura 3-11: Doble autenticación

Cerrar todas las sesiones abiertas (menos la actual):

The screenshot shows a user interface for managing browser sessions. At the top, it says "Browser Sessions" and "Manage and log out your active sessions on other browsers and devices." Below this, there is a message: "If necessary, you may log out of all of your other browser sessions across all of your devices. Some of your recent sessions are listed below; however, this list may not be exhaustive. If you feel your account has been compromised, you should also update your password." A session entry is shown: "OS X - Chrome" with the IP address "172.21.0.1, This device". At the bottom, there is a large red button labeled "LOG OUT OTHER BROWSER SESSIONS".

Figura 3-12: Sesiones abiertas

Destruir la cuenta, gracias a la característica que activamos anteriormente:

The screenshot shows a user interface for deleting an account. At the top, it says "Delete Account" and "Permanently delete your account." Below this, there is a message: "Once your account is deleted, all of its resources and data will be permanently deleted. Before deleting your account, please download any data or information that you wish to retain." At the bottom, there is a large red button labeled "DELETE ACCOUNT".

Figura 3-13: Borrar cuenta

Opciones en API Tokens

Si vamos a la opción de API Tokens en el menú anterior (cuya opción tenemos habilitada, gracias a la activación de la característica anterior):

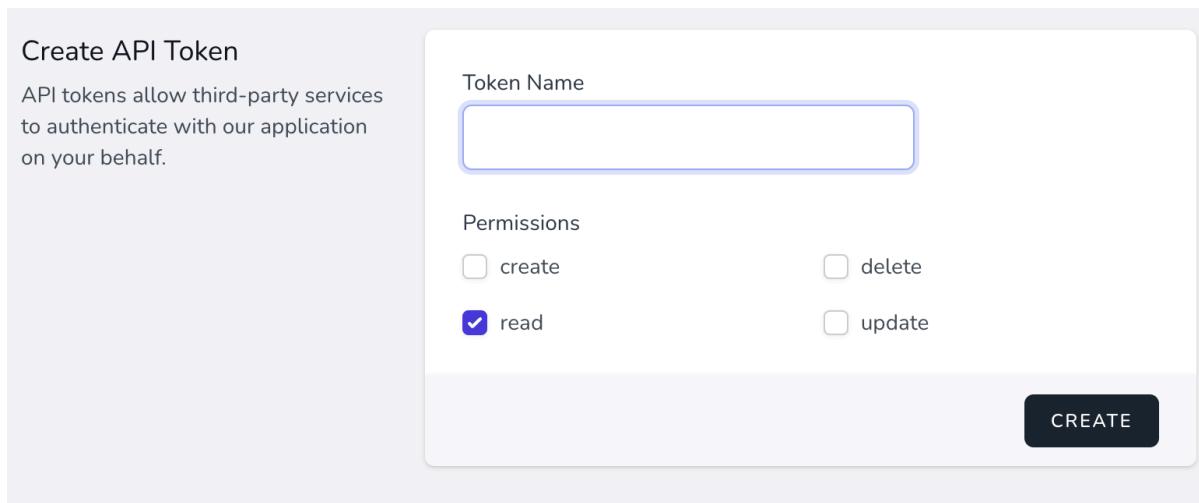


Figura 3-14: Crear Api Token

Desde esta opción, puedes crear permisos con habilidades a tus usuarios; esto es muy útil si vas a crear una Rest Api y deseas agregar protección vía API Tokens de Sanctum junto con las habilidades.

Estos permisos o habilidades, las puedes personalizar desde aquí:

App/Providers/JetstreamServiceProvider.php

```
Jetstream::role('admin', 'Administrator', [
    'create',
    'read',
    'update',
    'delete',
])->description('Administrator users can perform any action.');

Jetstream::role('editor', 'Editor', [
    'read',
    'create',
    'update',
])->description('Editor users have the ability to read, create, and update.');
```

Justamente, desde ese apartado, puedes editar las reglas, no importa si es el renglón de "editor" o "admin" o creas otro; Inertia va a hacer el **merge** y mostrar solo las opciones únicas; por ejemplo:

```
Jetstream::role('admin', 'Administrator', [
    'create',
    'read',
    'update',
    'matar a thanos',
    'delete',
])->description('Administrator users can perform any action.');
```

```

Jetstream::role('editor', 'Editor', [
    'read',
    'create',
    'otro',
    'update',
])->description('Editor users have the ability to read, create, and update.');

Jetstream::role('regular', 'Regular', [
    'read',
])->description('Regular users have the ability to read.');

```

Create API Token

API tokens allow third-party services to authenticate with our application on your behalf.

Token Name

Permissions

| | |
|--|---------------------------------|
| <input type="checkbox"/> create | <input type="checkbox"/> delete |
| <input type="checkbox"/> matar a thanos | <input type="checkbox"/> otro |
| <input checked="" type="checkbox"/> read | <input type="checkbox"/> update |

CREATE

Figura 3-15: Crear Api Token, permisos

Como puedes ver, creamos un nuevo tipo, llamado rol, y a los otros agregamos nuevos permisos y aparece es el **merge** de los mismos; vamos a guardar esta configuración para los roles y permisos, ya que, esta estructura se emplea en otro apartado que ya vamos a ver; finalmente, si creamos alguno, veremos que en la parte inferior aparece dicho permiso:

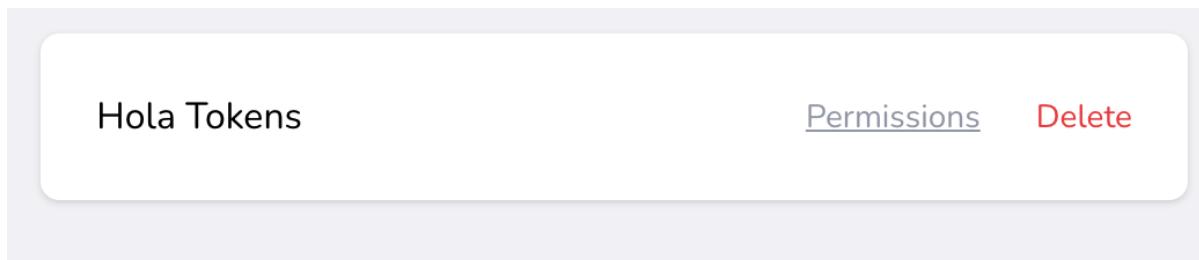


Figura 3-16: Token de ejemplo

Y el mismo se almacena en la base de datos, en la tabla que usa Sanctum para controlar los tokens y habilidades:

| id | tokenable_type | tokenable_id | name | token | abilities | last_used_at |
|----|-----------------|--------------|-------------|------------|-------------------------------------|--------------|
| 1 | App\Models\U... | | Hola Tok... | 58f4bc5... | ["read","update","delete","create"] | NULL |

Figura 3-17: Token creado en la base de datos

Opciones de equipos

Para los equipos o teams, su propósito, es similar al de los permisos por Sanctum, pero en vez de ser vistos desde recursos rest, lo usamos vía la aplicación que consumimos vía un navegador o similares; es decir, las rutas de web; con esto, podemos crear grupos de usuarios en base a un equipo o equipos; como puedes ver, ya por defecto, tenemos un equipo, pero, podemos crear más para un usuario y administrar permisos del mismo; en pocas palabras, las habilidades que van a tener:

Add Team Member

Add a new team member to your team, allowing them to collaborate with you.

Please provide the email address of the person you would like to add to this team.

Email

Role

- Administrator

Administrator users can perform any action.
- Editor

Editor users have the ability to read, create, and update.
- Regular

Regular users have the ability to read.

ADD

Figura 3-17: Crear equipo

Desde la pantalla anterior, tal cual puedes ver, puedes asignar usuarios y los permisos en base a los grupos que te presentaba en la opción de Provider; con esto, de manera programática, puedes realizar las siguientes opciones:

```
// Acceder al equipo actualmente seleccionado de un usuario...
$user->currentTeam : Laravel\Jetstream\Team
```

```

// Acceda a todos los equipos (incluidos los equipos propios) a los que pertenece un
usuario...
$user->allTeams() : Illuminate\Support\Collection

// Acceder a todos los equipos propiedad de un usuario...
$user->ownedTeams : Illuminate\Database\Eloquent\Collection

// Acceda a todos los equipos a los que pertenece un usuario pero que no son de su
propiedad...
$user->teams : Illuminate\Database\Eloquent\Collection

// Acceder al equipo "personal" de un usuario...
$user->personalTeam() : Laravel\Jetstream\Team

// Determinar si una usuario posee un equipo determinado
$user->ownsTeam($team) : bool

// Determinar si una usuario pertenece a un equipo determinado ...
$user->belongsToTeam($team) : bool

// Obtener el rol que tiene asignado el usuario en el equipo...
$user->teamRole($team) : \Laravel\Jetstream\Role

// Determine si el usuario tiene el rol dado en el equipo dado...
$user->hasTeamRole($team, 'admin') : bool

// Acceda a un array de todos los permisos que tiene un usuario para un equipo
determinado...
$user->teamPermissions($team) : array

// Determinar si un usuario tiene un permiso de equipo determinado...
$user->hasTeamPermission($team, 'server:create') : bool

```

Como puedes ver, son características muy útiles, que nos permiten manejar un sencillo esquema de roles y permisos a un grupo de usuarios.

Estructura del proyecto

El proyecto en sí, es un proyecto en Laravel con algunos agregados, como lo es Fortify para la autenticación y opciones presentadas anteriormente; te pudieras preguntar, donde se encuentran definidas las rutas y componentes que permiten las pantallas anteriores; a nivel de proyecto verás que tenemos archivos Vue y nada de vistas .blade (menos el resources/views/app.blade.php):

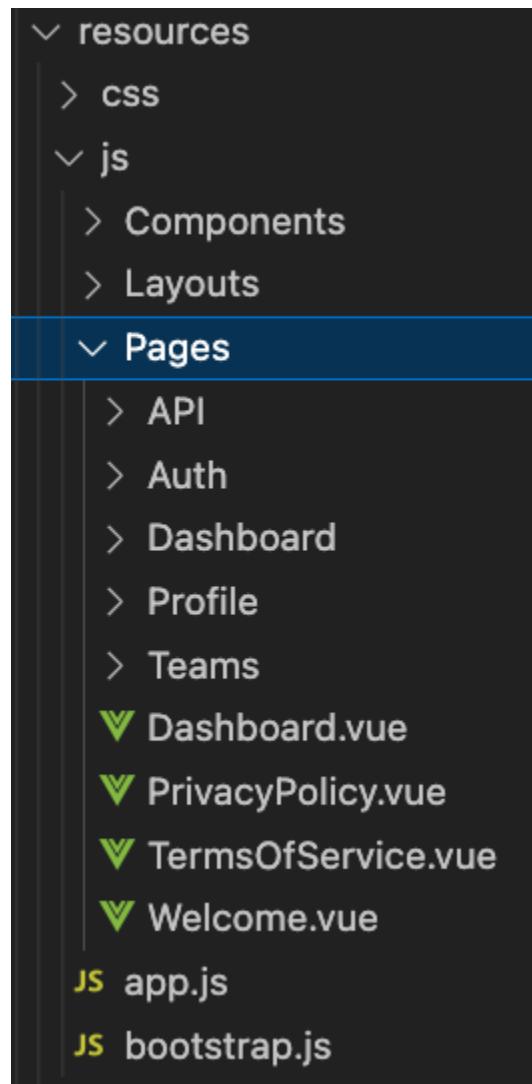


Figura 3-18: Vistas/Componentes en Vue

Toda la lógica de las vistas se maneja mediante Vue; sin embargo, existen procesos y rutas que se manejan desde el controlador, es decir, cuando se conecta con la base de datos, para administrar equipos, Api Tokens y demás es lógico pensar que debe de existir un proceso en el backend, es decir en Laravel, pero, si inspeccionamos el proyecto, no veremos nada de esto; al igual que las rutas:

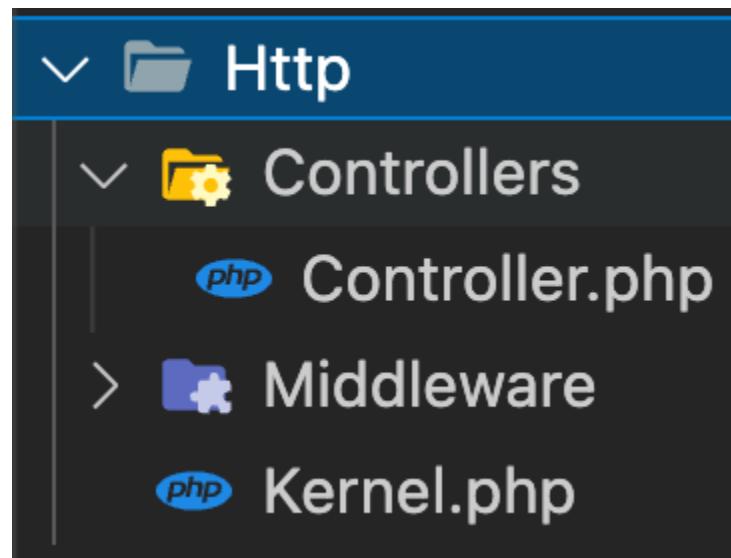


Figura 3-19: Carpeta de controladores vacía

Estas funciones se encuentran en el **vendor** del proyecto; para determinar esto, puedes hacer una búsqueda a nivel del proyecto y encontrar los controladores:

```
vendor/laravel/jetstream/src/Http/Controllers/Inertia
```

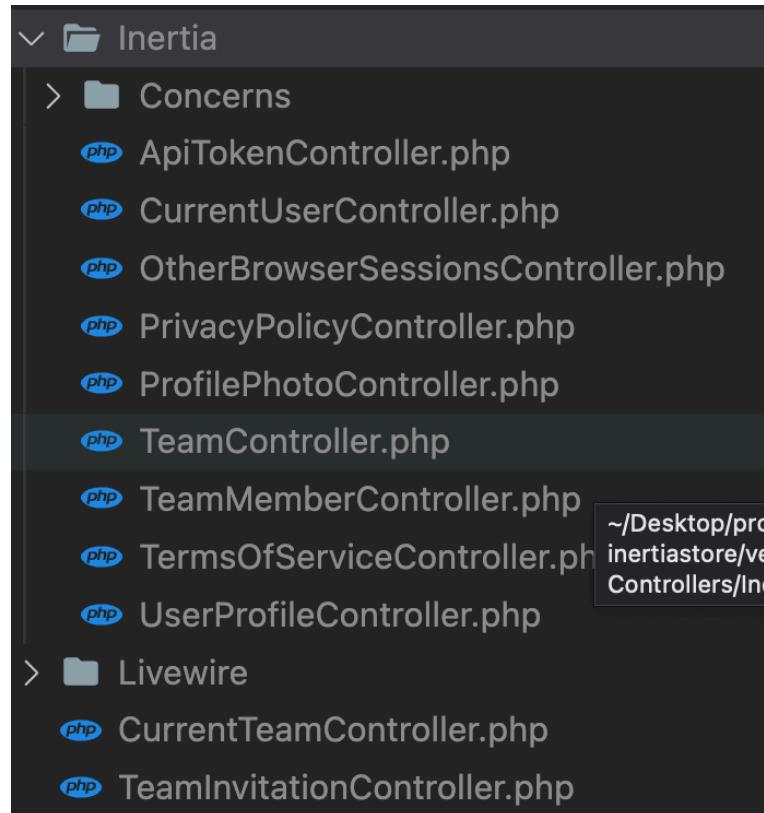


Figura 3-20: Controladores de Inertia en carpeta vendor

Estas referencias hacia la carpeta **vendor**, se hace por un tema netamente educativo, para que conozcas cómo funciona Inertia, pero recuerda que no debes de modificar ningún archivo que se encuentre dentro de la carpeta **vendor**.

Finalmente; aquí también se encuentran las rutas internas que emplea Inertia:

/vendor/laravel/jetstream/routes/inertia.php

```
// ***
$authSessionMiddleware = config('jetstream.auth_session', false)
    ? config('jetstream.auth_session')
    : null;

Route::group(['middleware' => array_values(array_filter([$authMiddleware,
$authSessionMiddleware]))], function () {
    // User & Profile...
    Route::get('/user/profile', [UserProfileController::class, 'show'])
        ->name('profile.show');

    Route::delete('/user/other-browser-sessions',
[OtherBrowserSessionsController::class, 'destroy'])
        ->name('other-browser-sessions.destroy');
// ***
```

Es el mismo esquema de rutas que se usa en Laravel básico.

Tipos de rutas

Al usar componentes de Vue para las vistas, es lógico pensar que debe de existir nuevas funciones que se emplean para especificar un archivo vue en lugar de las vistas blade de Laravel.

Es importante notar que, a diferencia del esquema clásico, vue es una tecnología del lado del cliente, mientras que las vistas usadas en Laravel básico son del servidor así que, con Inertia podemos lograr este tipo de comunicación.

Rutas y renderizado de vistas en Inertia

Para poder usar un componente en Vue en lugar de una vista de Laravel, tenemos el método de:

```
Route::inertia()
```

Que recibe dos parámetros, la URI y el componente; básicamente lo mismo que se hace con los métodos de ruta de Laravel básico; por ejemplo, una ruta de tipo GET:

```
Route::get()
```

Para probar el renderizado de componentes en Vue con rutas definidas en el servidor, vamos a crear un archivo Vue:

resources/js/Pages/Dashboard/Post/Index.vue

```
<template>
  <div>
    <h1>Listado de Post</h1>
  </div>
</template>
```

Y vamos a crear la siguiente ruta:

```
|Route::inertia('/indexconinertia', 'Dashboard\Post\Index');
```

Ejecutamos un:

```
|$ npm run dev
```

Para que tome los cambios automáticamente; si vamos a la ruta anterior, veremos el siguiente mensaje:

http://localhost/indexconinertia

```
|Listado de Post
```

Si ves una pantalla en blanco, seguramente tienes un error al momento de definir la ruta hacia el componente de Vue:

```
|Uncaught (in promise) Error: Cannot find module './Dashboard\Post\index.vue'
```

Render de Inertia y Laravel básico

Claro está, con definir en una ruta desde el servidor la referencia a un componente a Vue, que es una tecnología del cliente, no hacemos mucho, ya que, en la mayoría de los casos, vamos a necesitar definir el un controlador, para poder realizar las operaciones típicas en el mismo, como crear registros, obtener detalles, etc; para estos casos, tenemos que hacer una combinación entre rutas de Laravel básico, y el renderizado de componentes de Vue de Inertia.

Definir la ruta

Supongamos que tenemos una ruta de tipo GET, la cual referencia a un controlador en Laravel:

```
|Route::get('/', [App\Http\Controllers\Dashboard\PostController::class, 'index']);
```

De momento, el código anterior es 100% código Laravel básico, ahora, desde el controlador de Laravel, para devolver un componente en Vue mediante Inertia, hacemos uso del método de:

```
|Inertia::render(<RutaComponenteVue>)
```

O la función de ayuda:

```
|inertia(<RutaComponenteVue>)
```

Para indicar la ruta hacia el componente de Vue; es importante notar que, es equivalente al uso de la función `view()` en Laravel básico, pero, en vez de devolver un vista en Laravel como un blade, se devuelve un archivo en Vue; archivo en Vue que debe de estar definido en la carpeta de:

resources/js/Pages

Que es la ruta que emplea Inertia para buscar los componentes de Vue; esto lo puedes ver, si revisas el archivo de:

resources/js/app.ts

```
createInertiaApp({
    title: (title) => `${title} - ${appName}`,
    resolve: (name) => require(`./Pages/${name}.vue`),
    setup({ el, app, props, plugin }) {
        return createApp({ render: () => h(app, props) })
            .use(plugin)
            .mixin({ methods: { route } })
            .mount(el);
    },
});
```

Verás que Inertia procesa los componentes de Vue ubicados en dicha ubicación.

Finalmente, si definimos una ruta clásica en Laravel:

routes/web.php

```
|Route::get('/', [App\Http\Controllers\Dashboard\PostController::class, 'index']);
```

Y el controlador:

app/Http/Controllers/Dashboard/PostController.php

```
use Inertia\Inertia;
***

class PostController extends Controller
{
    public function index()
    {
        return Inertia::render('Dashboard/Post/Index');
        //return inertia('Dashboard/Post/Index');
```

```
}
```

Colocamos en nuestro navegador, la ruta configurada anteriormente, veremos nuevamente el mismo mensaje:

<http://localhost>

```
Listado de Post
```

Es decir, desde el controlador, fue retornado un componente de Vue; es decir, desde un proceso en el servidor, se devolvió directamente una vista que se encuentra en el cliente.

El uso del método de renderizado de Inertia, también la puedes ver aplicada por defecto si revisas el archivo de rutas de la aplicación:

`routes/web.php`

```
Route::get('/', function () {
    return Inertia::render('Welcome', [
        'canLogin' => Route::has('login'),
        'canRegister' => Route::has('register'),
        'laravelVersion' => Application::VERSION,
        'phpVersion' => PHP_VERSION,
    ]);
});

Route::middleware([
    'auth:sanctum',
    config('jetstream.auth_session'),
    'verified',
])->group(function () {
    Route::get('/dashboard', function () {
        return Inertia::render('Dashboard');
    })->name('dashboard');
});
```

Claro, en este caso, es aplicado directamente sobre las rutas y no usando un controlador.

Donde seguir desde aquí:

Sin mucho más que decir, espero que este libro fuese más de lo que esperas del mismo; si te quedaron algunas dudas, recuerda que puedes releer parte o todo el libro para afianzar tus ideas.

Recuerda que, para poder dominar este framework al igual que cualquier otro framework, tienes que realizar varios proyectos, plasmar tus propias ideas en tus proyectos, y modificar las presentadas en este libro.

Si te quedaste con ganas de más, en mi canal de YouTube cuento con más recursos, aparte de que cuento con un curso de Laravel Inertia, Livewire y Laravel básico en la cual vemos lo presentado en este libro y mucho más; este curso lo puedes conseguir desde mi blog en desarrollolibre.net/cursos el cual se encuentra disponible en desarrollolibre.net/academia y en otras plataformas.

Esto no será lo último de este libro, ya que lo pienso mantener por mucho tiempo; a medida que vaya recibiendo retroalimentación de ustedes, los lectores, iré haciendo correcciones y agregando nuevos capítulos; recuerda que el mantenimiento de este libro depende de ti. Que por favor compartas el enlace de donde comprar este libro, por tus redes sociales para que más personas puedan interesarse en este escrito y con esto, mientras a más personas llegue el libro, más contenido iré aportando.

Conclusiones

En definitiva, ya con esto conoces la estructura de un proyecto en Laravel Inertia, y puedes ver importantes agregados con respecto a Laravel básico como la definición de módulo como el de equipos, API Tokens, autenticación y llegando a la parte más interesante que son esas vistas en base a componentes que podemos reutilizar junto con la posibilidad de llamar a métodos en el servidor desde una vista.

También conoces cómo crear tus componentes, como comunicar los mismos, los diversos métodos y acciones que existen para crear aplicaciones complejas, de una manera simple y limpia.

El libro está actualmente en desarrollo...