9- Actualizando el estado

Aquí es donde mas disfrutaremos de la magia de **Vue**. Sin tocar el DOM ni modificar elementos html tendremos un sistema que nos permita tener la información actualizada todo el tiempo.

Como trabajaremos con 3 modelos diferentes dedicaremos un único controlador al proceso de actualización. Sobre todo para mantener separada esa lógica. Llamaremos al controlador **QueryController.**

```
php artisan make:controller QueryController
```

y por ahora solo crearemos una función dentro del controlador

```
public function allQuery(Request $request){
if (!$request->ajax()) return redirect('/');
return [
         'departures'=>Departure::all()
       ];
}
```

y agregaremos un use antes del class

```
use App\Departure;
```

y una ruta que al ser solicitud de datos usaremos el protocolo **GET**

```
Route::get('/allQuery','QueryController@allQuery')->name('allQuery');
```

Ahora revisemos el código nuevamente.

Al ser una ruta get el navegador aceptaría **/allQuery** como ruta aceptable pero como la idea es que no sea navegable hemos hecho la primer linea

```
if (!$request->ajax()) return redirect('/');
```

Esto indica que si no es una llamada **Ajax** desviara la llamada a la ruta **/** .

En cambio si es una llamada **Ajax** saltara el if y devolvera los datos en un array

```
return [
         'departures'=>Departure::all()
       ];
```

Lo siguiente sera agregar un array a nuestras variables de **Vue**. (recordar que esto es en data)

```
 departures:[]
```

Simple verdad?. Estamos creado un array vacío con el nombre **departures**.

Lo siguiente que cambiaremos sera mostrar los datos en el html.

Cambiaremos estas lineas

```
<div class="columns">
    <div class="column">
        Tabla de departamentos
    </div>
</div>
```

por estas otras

```
<div class="columns">
    <div class="column">
        <div v-if="!departures.length">
            No hay departamentos
        </div>
        <table v-else class="table">
            <thead>
                <th>#</th>
                <th>Titulo</th>
            </thead>
            <tbody>
                <tr v-for="departure in departures">
                    <td>@{{ departure.id }}</td>
                    <td>@{{ departure.title }}</td>
                </tr>
            </tbody>
        </table>
    </div>
</div>
```

Veamos el código que tenemos cosas nuevas.

```
<div v-if="!departures.length">
```

v-if ya lo hemos visto y es que se muestra el **div** y su contenido si la condición se cumple. En este caso se mostrara siempre que el tamaño del array **NO** sea positiva (positivo es cualquier numero superior a 0). O sea cuando el array este vacio mostrara ese mensaje.

```
<table v-else class="table">
```

Esto si es nuevo y una de las diferencias entre **v-show** y **v-if**. **v-else** se muestra cuando la condición del **v-if** no se cumple. En este caso cuando el array tenga 1 o mas valores.

Lo siguiente es lo ultimo para completar la lista

```
<tr v-for="departure in departures">
    <td>@{{ departure.id }}</td>
    <td>@{{ departure.title }}</td>
</tr>
```

v-for es una secuencia de repetición. Recorrera cada elemento del array.

Aqui haremos una observación importante con respecto a la programación de backend. En el backend dentro de un foreach el orden seria **array as variable** en cambio aqui el orden es **variable in array.**

Nuevamente vemos **@{{}}** recordando que esto es para evitar que **blade** procese la variable. Ahora de donde sale **.id** o **.title?** Eso son los campos del modelo que hemos creado en la migración en anteriores posts.

```
 Schema::create('departures', function (Blueprint $table) {
            $table->increments('id');
            $table->string('title');
});
```

Y ahi los tenemos. **id** y **title**.

Perfecto. Tenemos como obtener los valores desde la db. Como mostrarlo en el front pero nos falta conectar estas dos cosas.

Esto lo haremos con una función en la zona **methods** de nuestro **Vue**,

```
allQuery() {
    let me = this;
    axios.get('{{route('allQuery')}}')
        .then(function (response) {
            let answer = response.data;
            me.departures = answer.departures;
        })
        .catch(function (error) {
            console.log(error);
        });
}
```

Hemos llamado a la función allQuery. El código de **Axios** esta vez tiene que ser mas facil de entender que la primera vez pero lo miraremos de nuevo.

```
let me = this;
```

Guardamos el this de Vue para referenciarlo sin problemas dentro de **Axios**

```
axios.get('{{route('allQuery')}}')
```

llamada con el protocolo **GET** a la ruta definida mas arriba

```
.then(function (response) {
            let answer = response.data;
            me.departures = answer.departures;
 })
```

Recordemos que **.then** es cuando la llamada ha tenido éxito. Si ese es el caso trabajamos con la respuesta cosa que aun no habíamos hecho. Para recibir la información de respuesta estamos usando **response**. La respuesta trae muchos datos que aun no necesitaremos. Lo que nos interesa esta en **response.data** que nos trae lo que hemos enviado desde el controlador.

¿Porque el código **let answer=response.data**? Es una prevención a futuro ya que ese mismo **data** luego nos traerá la información de mas modelos y lo veremos mas claro en futuras actualizaciones.

Ahora la siguiente linea **me.departures=answer.departures** es muy sencilla y potente. Estamos guardado el array que enviamos desde el controlador en el array del front. Eso sera mas que suficiente para mantener el sistema actualizado. Sin borrar elementos html ni tocar otras cosas. Con eso es mas que suficiente.

El siguiente paso es de estrategia de programación. ¿Cuando usar la función?

Necesitamos actualizar al inicio de cargar la pagina, en el crear, en el update y en el eliminar.

El crear, el update y el eliminar tienen algo en común -> el modal.

Cuando hacemos una de esas 3 cosas abrimos el modal. Seleccionamos lo que necesitamos y el modal se cierra. Así que lo que haremos sera asociar los cambios en la variable que abre y cierra el modal con la función. Y para tenerlo al inicio lo pondremos en una función especial de **Vue**.

Cambiaremos este código en **Vue**

```
let elemento = new Vue({
            el: '.app',
            data: {
```

agregando una función especial de **Vue** llamada **mounted**

```
let elemento = new Vue({
            el: '.app',
            mounted: function () {
                this.allQuery();
            },
            data: {
```

Las funciones que usemos dentro de **mounted** se ejecutaran al inicio de la aplicación. Con tan poco código tendremos los últimos datos en el front al iniciar la carga de la pagina.

cambiaremos

```
data: {
        .
        .
    },
    methods: {
```

por

```
data: {
        .
        .
    },
watch: {
    modalGeneral: function (value) {
        if (!value) this.allQuery();
    }
},
    methods: {
```

Aqui usamos una particularidad de **Vue** que no habíamos usado. **watch** simplemente es decirle al sistema que este atento al cambio a una variable y haga algo cuando eso pasa.

En este caso en particular le estamos diciendo que este atento a los cambios de **modalGeneral**. La variable **value** es el valor actual de la variable.

En el **if** le indicamos que cuando sea 0 (que es cuando se cierra el modal) se ejecute la función que hemos creado para mantener la actualización. Ya no necesitamos nada mas para mantener los últimos datos en el front.

Este post ha sido bastante largo pero hemos logrado un montón de cosas de manera muy sencilla.

Código: Github

Proximo post: Eliminando departamentos