

Cours Laravel 10 – Livewire

 laravel.sillo.org/cours-laravel-10-livewire/

3 mars 2023

Livewire est finalement assez récent. Il se présentait comme un ovni et maintenant se banalise à tel point que je ressens la nécessité de l'intégrer dans mon cours sur Laravel. Ma première impression en abordant Livewire avait été assez négative. Parce que la proposition est la suivante : plutôt que d'utiliser un framework Javascript comme Vue ou React pourquoi ne pas quitter Laravel et de tout coder en PHP ? La proposition a de quoi surprendre parce que le PHP est loin du navigateur et qu'il faut assurer la liaison entre les deux. Mais finalement c'est un outil plaisant qui fait réellement gagner beaucoup de temps en évitant d'écrire pas mal de Javascript. Je vous en propose dans cet article une introduction.

Installation

On va partir d'un Laravel tout neuf :

```
composer create-project laravel/laravel livewire
```

Et on ajoute l'authentification avec Breeze :

```
composer require laravel/breeze --dev
php artisan breeze:install
```

Choisissez Breeze avec Blade (option 0) et dites non à tout le reste.

Précisez dans le fichier .env le nom de la base de données :

```
DB_DATABASE=livewire
```

Et lancez les migrations :

```
php artisan migrate
```

Livewire est un package pour Laravel qui s'installe classiquement avec composer :

```
composer require livewire/livewire
```

Et voilà on est prêts !

Créer un composant

Livewire fonctionne avec des composants. Pour en créer un c'est tout simple :

```
php artisan make:livewire ShowPosts
```

On se retrouve avec deux fichiers créés (et deux dossiers qui n'existaient pas auparavant) :

Le premier comporte l'intendance (la classe) et le second s'occupe de l'aspect (la vue).

Voyons le code généré dans la classe :

```
<?php

namespace App\Http\Livewire;

use Livewire\Component;

class ShowPosts extends Component
{
    public function render()
    {
        return view('livewire.show-posts');
    }
}
```

On a une méthode **render** et on retourne la vue générée.

Dans la vue on a :

```
<div>
    {{-- The best athlete wants his opponent at his
    best. --}}
</div>
```

La phrase est générée aléatoirement.

Donc on n'a pas grand chose pour le moment.

Utiliser un composant

Maintenant qu'on sait créer un composant voyons comment l'utiliser. Il y a deux façons de le faire :

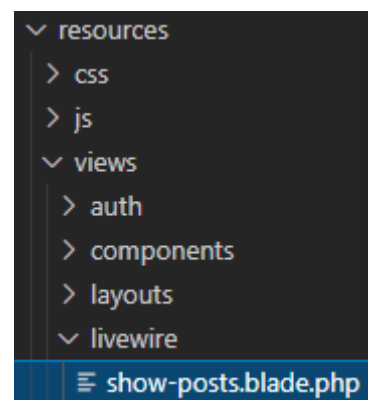
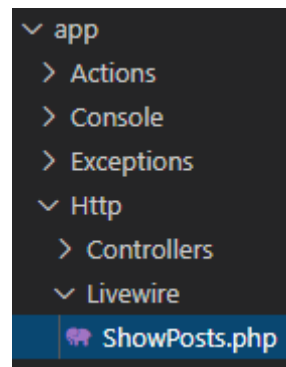
- dans une vue avec une nouvelle commande blade qui fonctionne donc comme une inclusion : **@livewire('show-posts')**
- dans une vue complète, dans ce cas le composant est la vue

C'est cette deuxième option qui est codée par défaut comme on l'a vu ci-dessus. Dans ce cas il nous faut aussi créer une route pour utiliser le composant :

```
use App\Http\Livewire\ShowPosts;

Route::get('posts', ShowPosts::class);
```

Si on fait ça avec une installation toute fraîche de Laravel on va tomber sur une erreur en utilisant l'url **.../posts** :



Le souci vient du fait que par défaut lorsqu'on utilise une page complète pour un composant il va automatiquement utiliser le layout

resources/views/layouts/app.blade.php.

Il y faut un emplacement **{{ \$slot }}**. Or ce layout est pour les utilisateurs authentifiés, ça va donc fonctionner si on a une authentification active. Mais pour le moment on va plutôt spécifier le layout à utiliser :

```
public function render()
{
    return view('livewire.show-posts')->layout('layouts.guest');
}
```

Pour que ça fonctionne il faut 3 choses dans le layout :

- une insertion des styles avec **@livewireStyles**
- un emplacement **{{ \$slot }}** dans lequel va se logger le composant
- une insertion des scripts de Livewire pour les capacités réactives qu'on va voir plus loin avec **@livewireScripts**

On va donc compléter et modifier notre code :

```
@livewireStyles
</head>
<body>
    ...
    @livewireScripts
</body>
```

Maintenant on n'a plus d'erreur mais évidemment une page avec juste le logo puisqu'on a rien ajouté dans notre vue. On va vérifier si ça fonctionne :

```
<div>
    <p>coucou</p>
</div>
```

Vous devriez avoir un simple coucou sur la page.

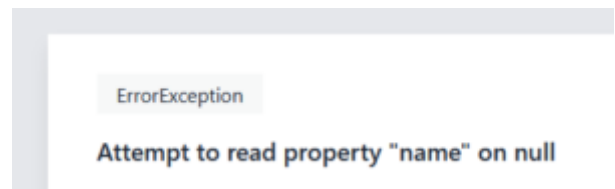
Les propriétés

Les composants de Livewire ont besoin de données. On peut créer des propriétés dans la classe :

```
class ShowPosts extends Component
{
    public $message = 'Coucou !';
}
```

Une propriété **publique** dans la classe est automatiquement disponible dans la vue :

```
<p>{{ $message }}</p>
```



Jusque là on n'a rien inventé de bien utile. Par contre ça va devenir plus intéressant avec la liaison de données. On connaît bien ça avec Vue.js par exemple on synchronise la valeur d'un élément sur la page web avec la propriété du composant.

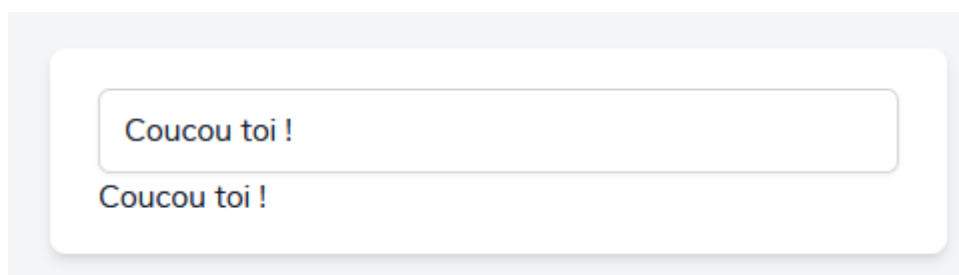
On va recoder notre vue :

```
<div>
  <input wire:model="message" class="block mt-1 w-full border-gray-300
focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm" type="text"/>
  <p>{{ $message }}</p>
</div>
```

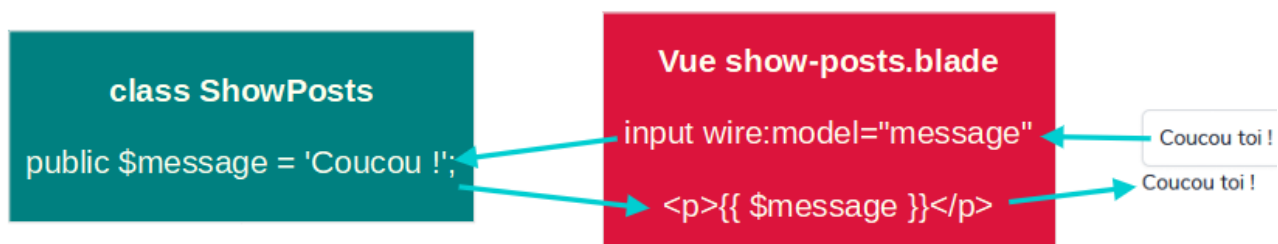
On ne va pas s'attarder sur les classes de Tailwind, ce n'est pas le sujet (qui a dit que c'était verbeux ?). J'ai d'ailleurs juste repris la mise en forme utilisée dans les formulaires de l'authentification pour garder le visuel. Les éléments importants ici sont :

```
<div>
  ...
  <input wire:model="message" ... />
  <p>{{ $message }}</p>
  ...
</div>
```

C'est l'attribut **wire:model** qui relie le champ de saisie à la propriété. Maintenant quand on change le texte dans le champ de saisie il s'actualise au-dessous :

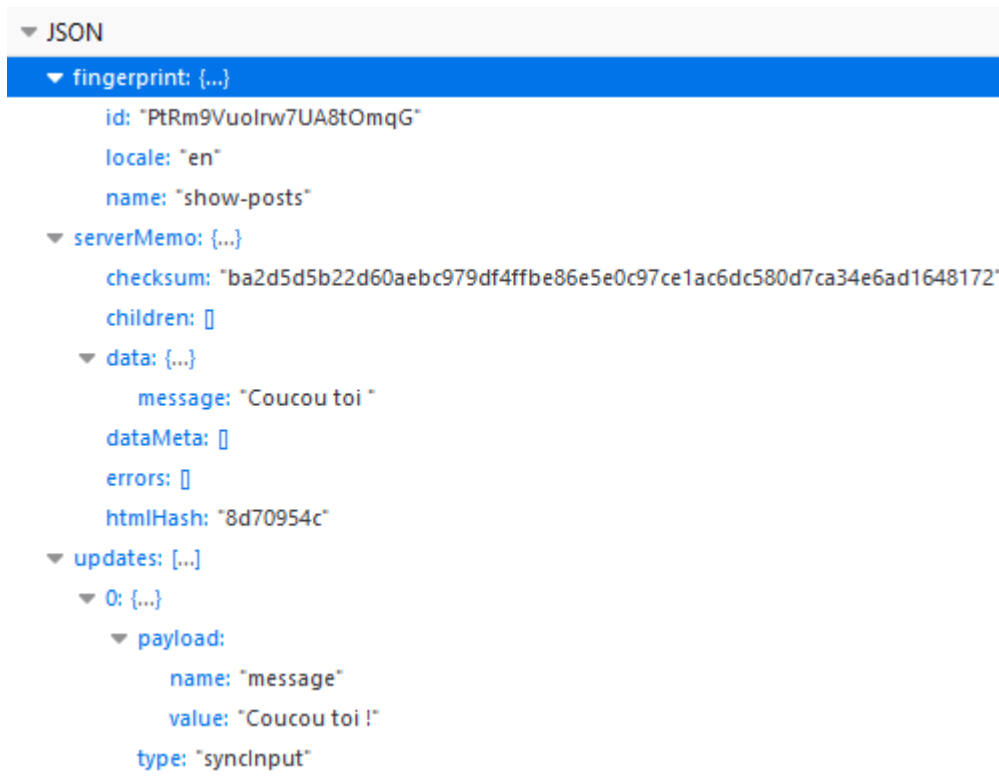


Un petit schéma pour bien visualiser ça :

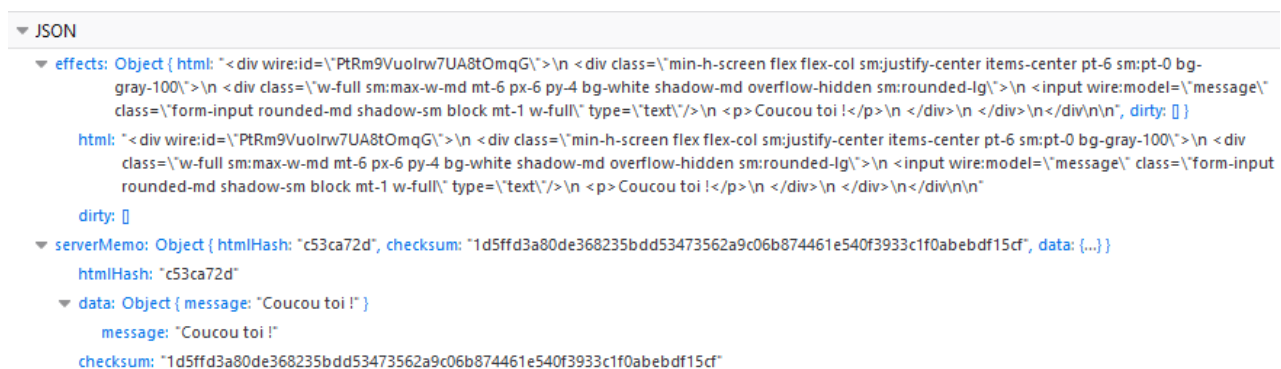


Maintenant la question qu'on peut se poser c'est : comment ça fonctionne ?

On peut remarquer qu'à chaque changement on a une requête, il part une requête POST de la forme **.../livewire/message/show-posts** avec ce corps :



On a au retour une réponse JSON :



Sous le capot Livewire utilise Alpine pour gérer le Javascript sur la page.

C'est lourd ce système, on peut faire la même chose avec quelques lignes de Javascript, mais ce n'est là que le principe de fonctionnement, et puis on peut améliorer les choses. Par défaut le rafraichissement s'effectue tous les 150 ms. On peut régler cette valeur comme on veut :

```
wire:model.debounce.1000ms="message"
```

Il est aussi possible de ne lancer l'actualisation qu'à la perte du focus :

```
wire:model.lazy="message"
```

On peut même reporter l'actualisation à la prochaine requête...

Les propriétés calculées

On peut aussi avoir des propriétés calculées (comme avec Vue.js), alors là évidemment ça devient plus intéressant parce qu'on a besoin d'information sur le serveur. Imaginons que nous ayons deux utilisateurs dans notre table **users** (il vous suffit d'utiliser le formulaire d'enregistrement). Alors on peut créer cette propriété calculée (computed) :

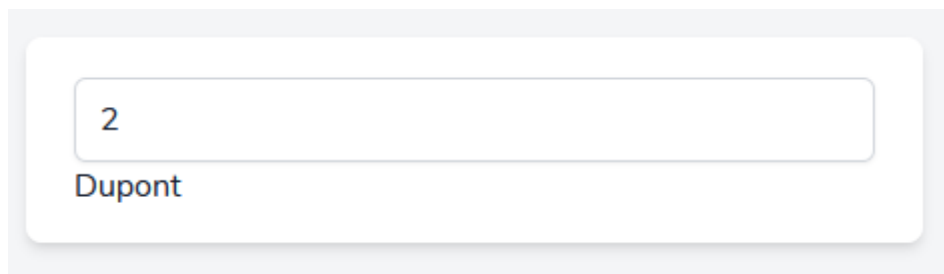
```
use App\Models\User;
use Livewire\Component;

class ShowPosts extends Component
{
    public $index = 1;
    public function getUserProperty()
    {
        return User::find($this->index);
    }
}
```

Et dans la vue :

```
<input wire:model="index" class="block mt-1 w-full border-gray-300 focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm" type="text"/>
<p>{{ $this->user->name }}</p>
```

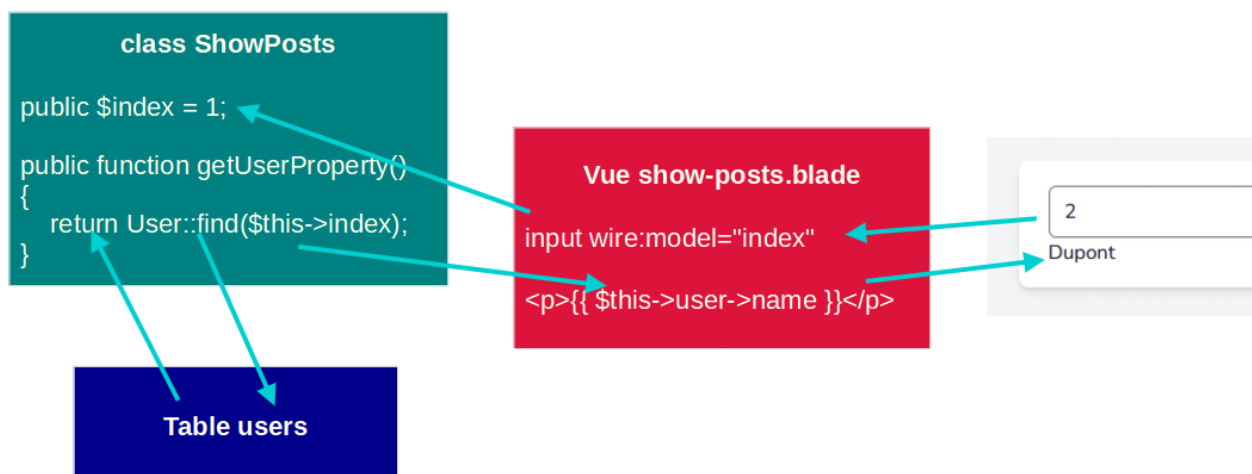
Maintenant quand on entre l'index dans la zone de texte le nom de l'utilisateur correspondant apparaît au-dessous :



A screenshot of a web form. It features a text input field with the number '2' entered. Below the input field, the name 'Dupont' is displayed in a blue font. The entire form is contained within a light gray rounded rectangle with a subtle shadow.

Là évidemment ça devient bien plus intéressant parce qu'on n'a pas à se soucier de toute l'intendance d'Ajax !

Un petit schéma à nouveau pour bien visualiser le fonctionnement :



Les actions

Une action dans Livewire c'est la capacité à écouter une action sur la page web et d'appeler une méthode dans le composant pour modifier la page. C'est donc un pas de plus dans l'interactivité par rapport aux propriétés calculées vues ci-dessus.

Préparation

Pour montrer ça on va ajouter une colonne dans notre table **users**. On crée donc une migration :

```
php artisan make:migration alter_users_table --table=users
```

On complète le code de la migration :

```
<?php

use Illuminate\Database\Migrations\Migration;
use Illuminate\Database\Schema\Blueprint;
use Illuminate\Support\Facades\Schema;

return new class extends Migration
{
    /**
     * Run the migrations.
     */
    public function up(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->integer('note');
        });
    }

    /**
     * Reverse the migrations.
     */
    public function down(): void
    {
        Schema::table('users', function (Blueprint $table) {
            $table->dropColumn('note');
        });
    }
};
```

Et on lance la migration :

```
php artisan migrate
```

Maintenant on dispose d'une colonne **note**.

Action simple

On va coder ainsi notre composant :

```

<?php

namespace App\Http\Livewire;

use App\Models\User;
use Livewire\Component;

class ShowPosts extends Component
{
    public $note = 0;

    public function noter()
    {
        $user = User::find(1);
        $user->note = $this->note;
        $user->save();
    }

    public function render()
    {
        return view('livewire.show-posts')->layout('layouts.guest');
    }
}

```

On a :

- une propriété **note**
- une action **noter**

Dans la vue :

```

<div>
    <input wire:model.defer="note" class="block mt-1 w-full border-gray-300
focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm" type="text"/>
    <button wire:click="noter" class="inline-flex items-center px-4 py-2 bg-gray-
800 border border-transparent rounded-md font-semibold text-xs text-white
uppercase tracking-widest hover:bg-gray-700 active:bg-gray-900 focus:outline-none
focus:border-gray-900 focus:ring ring-gray-300 disabled:opacity-25 transition
ease-in-out duration-150 mt-4">
        Noter
    </button>
</div>

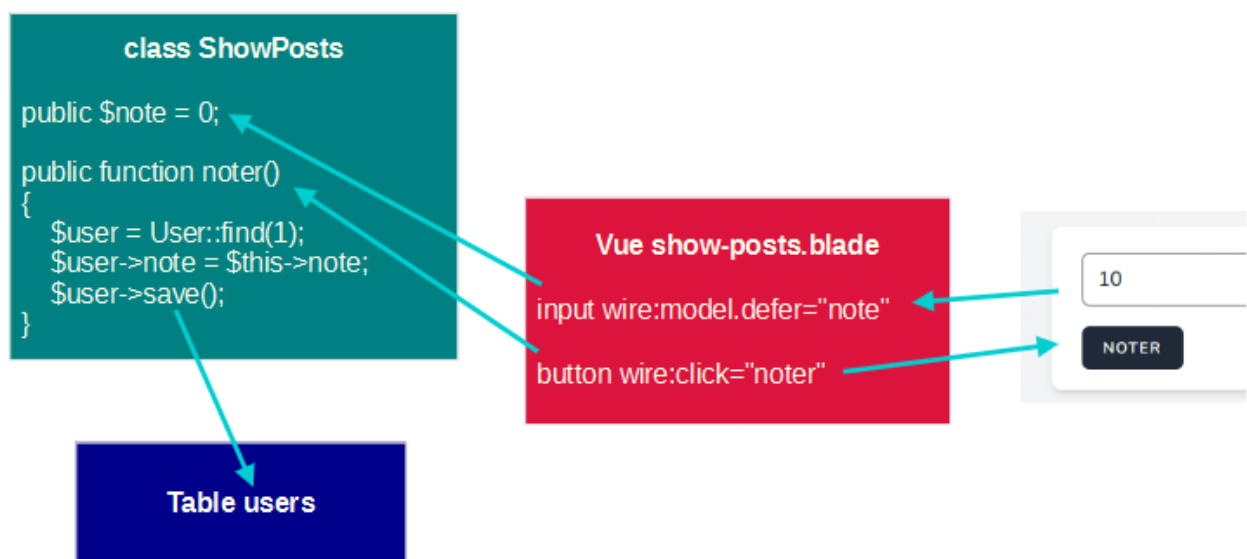
```

On a :

- une liaison de données avec la propriété note : **wire:model.defer= »note »** (on prévoit **defer** pour éviter les multiples requêtes)
- un bouton pour noter avec une action sur le click : **wire:click= »noter »**

Maintenant quand on met une note dans la zone de texte et qu'on clique sur le bouton on retrouve la valeur de la note dans la colonne note de l'utilisateur (ici on a pris simplement le premier, mais on pourrait évidemment rendre ça dynamique ou prendre l'utilisateur connecté) :

id	name	note	email
1	Durand	10	durand@chezlui.fr



Passage de paramètre

On peut passer un paramètre pour l'action. Poursuivons notre exemple en transmettant l'index de l'utilisateur. Dans la classe on ajoute une propriété **\$index** et on utilise le paramètre dans l'action :

```

public $note = 0;
public $index = 1;

public function noter($indexUser)
{
    $user = User::find($indexUser);
    $user->note = $this->note;
    $user->save();
}

```

Dans la vue on ajoute la saisie de l'index et la transmission du paramètre :

```

<div>
  <label class="block font-medium text-sm text-gray-700">Index de
l'utilisateur</label>
  <input wire:model.defer="index" class="block mt-1 w-full border-gray-300
focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm" type="text"/>
  <label class="mt-4 block font-medium text-sm text-gray-700">Note</label>
  <input wire:model.defer="note" class="block mt-1 w-full border-gray-300
focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm" type="text"/>
  <button wire:click="noter({{ $index }})" class="inline-flex items-center px-4
py-2 bg-gray-800 border border-transparent rounded-md font-semibold text-xs text-
white uppercase tracking-widest hover:bg-gray-700 active:bg-gray-900
focus:outline-none focus:border-gray-900 focus:ring ring-gray-300
disabled:opacity-25 transition ease-in-out duration-150 mt-4">
    Noter
  </button>
</div>

```

La partie intéressante est le passage du paramètre :

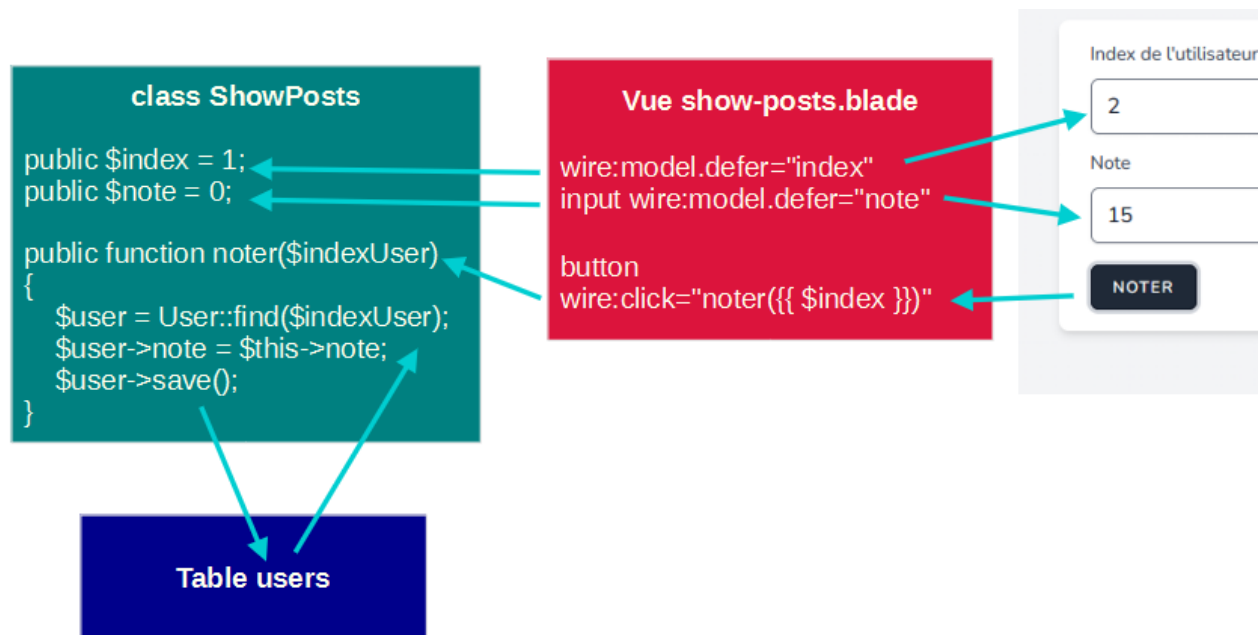
```
wire:click="noter({{ $index }})"
```

On a donc deux zones de texte :



The screenshot shows a light gray rounded rectangle containing a form. At the top, the text "Index de l'utilisateur" is displayed in a blue font. Below it is a text input field with the number "1" inside. Further down, the text "Note" is displayed in a blue font. Below that is another text input field with the number "0" inside. At the bottom of the form is a dark gray button with the word "NOTER" in white capital letters.

On peut maintenant choisir l'utilisateur à noter. Bon, c'est sommaire et pas du tout réaliste, mais ça donne le principe de fonctionnement. On pourrait par exemple remplir une liste de choix avec les noms des utilisateurs. Je ne vais pas le faire pour rester simplement dans les principes de fonctionnement.



La validation

Dans mon exemple précédent je n'ai prévu aucun contrôle quant aux valeurs transmises. Dans une approche réaliste il nous faut évidemment une validation des valeurs. La bonne nouvelle c'est que le système est le même que pour la validation dans Laravel.

Validation classique

On va changer le code pour adopter quelque chose de plus réaliste avec un formulaire :

```

<div>
  <form wire:submit.prevent="submit">
    <label class="block font-medium text-sm text-gray-700">Index de
l'utilisateur</label>
    <input wire:model.defer="index" class="block mt-1 w-full border-gray-300
focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm" type="text"/>
    <x-input-error :messages="$errors->get('index')" class="mt-2" />
    <label class="mt-4 block font-medium text-sm text-gray-700">Note</label>
    <input wire:model.defer="note" class="block mt-1 w-full border-gray-300
focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm" type="text"/>
    <x-input-error :messages="$errors->get('note')" class="mt-2" />
    <button type="submit" class="inline-flex items-center px-4 py-2 bg-gray-
800 border border-transparent rounded-md font-semibold text-xs text-white
uppercase tracking-widest hover:bg-gray-700 active:bg-gray-900 focus:outline-none
focus:border-gray-900 focus:ring ring-gray-300 disabled:opacity-25 transition
ease-in-out duration-150 mt-4">
      Noter
    </button>
  </form>
</div>

```

Pour l'affichage de erreurs de validation j'utilise le composant de Breeze **x-input-errors**.

Pour notre composant on a maintenant ce code :

```

<?php

namespace App\Http\Livewire;

use App\Models\User;
use Livewire\Component;

class ShowPosts extends Component
{
    public $note = 0;
    public $index = 1;

    protected $rules = [
        'note' => 'required|integer|between:0,20',
        'index' => 'required|exists:users,id',
    ];

    public function submit()
    {
        $this->validate();
        $user = User::find($this->index);
        $user->note = $this->note;
        $user->save();
    }

    public function render()
    {
        return view('livewire.show-posts')->layout('layouts.guest');
    }
}

```

Le nom du composant n'est pas trop approprié mais bon, on le garde.

On a donc deux propriétés :

- note
- index

On prévoit une validation des deux propriétés :

```

protected $rules = [
    'note' => 'required|integer|between:0,20',
    'index' => 'required|exists:users,id',
];

```

Et pour le reste c'est pratiquement pareil. Maintenant on a un contrôle de la validité des entrées :

Index de l'utilisateur

10

The selected index is invalid.

Note

23

The note field must be between 0 and 20.

NOTER

Bon je n'ai pas traduit en français mais c'est une autre histoire qu'on a déjà rencontrée plusieurs fois...

Mais on peut personnaliser les messages selon l'erreur :

```
protected $messages = [  
    'note.integer' => 'C'est quand même mieux un nombre pour une note !',  
];
```

Note

vingt

C'est quand même mieux un nombre pour une note !

Validation en temps réel

Livewire va plus loin en permettant une validation en temps réel lors de la saisie de la valeur sans attendre la soumission. Il suffit d'ajouter un hook :

```
public function updated($note)  
{  
    $this->validateOnly($note);  
}
```

On choisit la propriété qu'on veut valider avec **validateOnly** (sinon on validerait toutes les valeurs).

Mais pour que ça fonctionne il faut évidemment enlever le **defer** :

```
<input wire:model="note"
```

Maintenant quand on entre une valeur pour la note on a la validation instantanément sans attendre la soumission. Plutôt sympa !

Imbrication de composants

Les composants de Livewire peuvent être imbriqués (nested), autrement dit on peut mettre des composants dans des composants. On va poursuivre notre exemple avec cette fois deux composants :

- un composant parent qui va afficher un utilisateur
- un composant enfant qui permet de noter l'utilisateur

On commence par créer les deux composants :

```
php artisan make:livewire ShowUser
php artisan make:livewire NoteUser
```

On va prévoir une route avec un paramètre pour préciser l'index de l'utilisateur :

```
use App\Http\Livewire\ShowUser;

Route::get('user/{user}', ShowUser::class);
```

Comme Livewire est parfaitement intégré à Laravel on peut utiliser la liaison de données (Route Model Binding), donc là je précise le nom de paramètre **user** pour que Laravel comprenne ce que je veux.

Avec Livewire on n'utilise pas de contrôleur alors comment récupérer le paramètre ? C'est tout simple avec Livewire :

```
<?php

namespace App\Http\Livewire;

use App\Models\User;
use Livewire\Component;

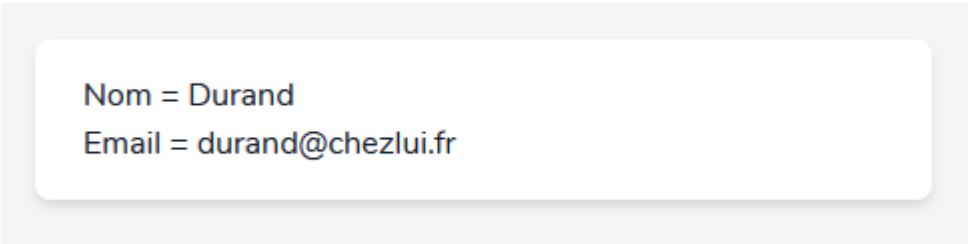
class ShowUser extends Component
{
    public User $user;

    public function render()
    {
        return view('livewire.show-user')->layout('layouts.guest');
    }
}
```

Là notre composant récupère bien le paramètre, va chercher dans la base les informations de l'utilisateur, et affecte la propriété **\$user** qui sera disponible dans la vue :

```
<div>
    <p>Nom = {{ $user->name }}</p>
    <p>Email = {{ $user->email }}</p>
</div>
```

Avec une url de la forme **.../user/1** on obtient les renseignements sur l'utilisateur :



Nom = Durand
Email = durand@chezlui.fr

On va maintenant inclure le composant pour noter l'utilisateur. On code la classe **NoteUser** :

```
<?php

namespace App\Http\Livewire;

use Livewire\Component;

class NoteUser extends Component
{
    public $note = 0;
    public $user;

    protected $rules = [
        'note' => 'required|integer|between:0,20',
    ];

    public function submit()
    {
        $this->validate();
        $this->user->note = $this->note;
        $this->user->save();
    }

    public function render()
    {
        return view('livewire.note-user');
    }
}
```

Et la vue **note-user** :

```

<div>
  <form wire:submit.prevent="submit">
    <label class="mt-4 block font-medium text-sm text-gray-700">Note</label>
    <input wire:model="note" class="block mt-1 w-full border-gray-300
focus:border-indigo-500 focus:ring-indigo-500 rounded-md shadow-sm" type="text"/>
    <x-input-error :messages="$errors->get('note')" class="mt-2" />
    <button type="submit" class="inline-flex items-center px-4 py-2 bg-gray-
800 border border-transparent rounded-md font-semibold text-xs text-white
uppercase tracking-widest hover:bg-gray-700 active:bg-gray-900 focus:outline-none
focus:border-gray-900 focus:ring ring-gray-300 disabled:opacity-25 transition
ease-in-out duration-150 mt-4">
      Noter
    </button>
  </form>
</div>

```

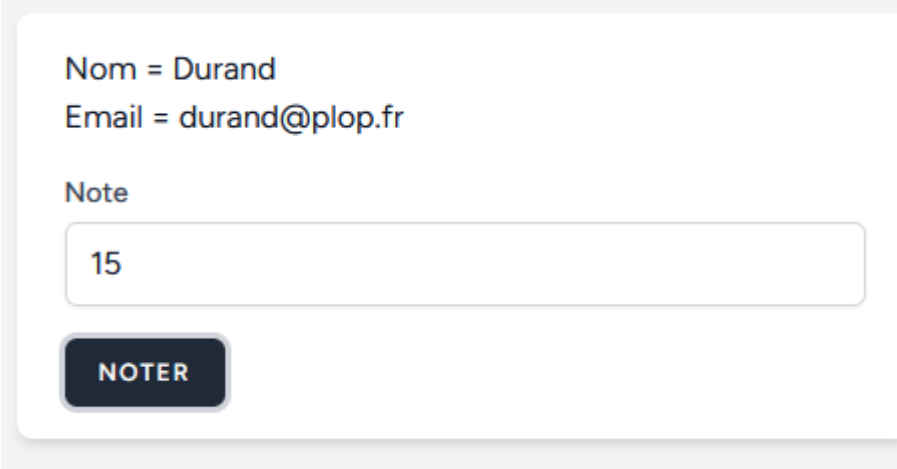
Il ne reste plus qu'à insérer ce composant dans l'autre, donc dans la vue **show-user** :

```

<div>
  <p>Nom = {{ $user->name }}</p>
  <p>Email = {{ $user->email }}</p>
  @livewire('note-user', ['user' => $user])
</div>

```

On prend la précaution d'envoyer l'information de l'utilisateur dans le composant enfant.



The screenshot shows a user profile card. At the top, it displays 'Nom = Durand' and 'Email = durand@plop.fr'. Below this, there is a section titled 'Note' which contains a text input field with the value '15'. At the bottom of the section is a dark blue button with the text 'NOTER' in white capital letters.

Et ça devrait fonctionner avec la validation !

Conclusion

Je n'ai fait dans cet article que montrer les fonctionnalités de base du système proposé. Je conseille vivement de consulter la documentation et surtout [les vidéos](#) qui sont très bien faites. C'est une autre façon d'envisager la gestion du frontend en allégeant le codage de ce côté. On fait de l'Ajax sans s'en rendre compte. Il faut évidemment bien mesurer la charge à assumer côté serveur parce que ça peut vite devenir lourd. D'autre part étant donné le délai qui peut se produire en attendant une réponse on peut se retrouver avec une réactivité dans les chaussettes. Pour ce cas Livewire prévoit [une gestion de l'attente](#).

Livewire connaît un certain succès qui me semble légitime.