How to Use Laravel Livewire for Interactive UIs

Laravel Livewire allows you to build dynamic, reactive interfaces without writing JavaScript. Instead, Livewire components are written in PHP and Blade, and Livewire handles the DOM updates via AJAX under the hood. This makes it perfect for developers who prefer Laravel's syntax but still need interactivity. In this article, we'll set up Livewire, create components, connect them with controllers, trigger events, and build a practical login form with validation.

I remember the first time I decided to drop Livewire into a Laravel 12 project to build an interactive admin panel — I wanted the speed of Blade and the reactivity of a small SPA without introducing a full JavaScript framework. I scaffolded a basic CRUD for products and thought it would be straightforward: create component, wire up model, done. The reality turned out to be a lot richer and taught me several hard lessons.

At the beginning I created a `ProductTable` Livewire component that held public properties for filters, sort direction, and the current page. I mounted the component with `mount()` and loaded the first page of products. Initially everything worked: filtering, sorting, and pagination responded and the UI felt snappy. But as I added features, problems started showing.

First problem: state bloat. I had bound a collection directly to a public property for convenience. After a few users started testing, requests slowed — every interaction serialized a huge payload. The fix was immediate: I replaced the collection with a `query` and only stored primitive values (search string, category id, sort key). When I needed models I fetched them inside methods (e.g., `render()` or action methods) so the serialized state stayed tiny.

Next was the search input — I bound it `wire:model="search"` initially and we got network requests on every keystroke. For the developer demo it was annoying, and on slower connections it felt broken. I switched to `wire:model.debounce.500ms="search"` and later to `wire:model.defer="search"` for forms where the user would explicitly

submit. That single change reduced the number of requests by an order of magnitude and smoothed the UX.

I also learned to pair Alpine.js and Livewire properly. I had a modal that contained a Livewire form. At first I controlled the modal purely with Alpine, and Livewire updates sometimes re-rendered the modal markup which closed it unexpectedly. The pattern that fixed this was: keep the modal open/close state in Alpine and use Livewire events to notify Alpine when to open or close. Example: after saving, Livewire did `$this->emit('saved')`, and in JS `window.livewire.on('saved', () => Alpine.store('modal').close())`. This kept responsibilities clear: Alpine for local UI state, Livewire for server interactions.

File uploads were another headache. I implemented quick image uploads using Livewire's temporary upload feature. With small files it was fine, but a client tried to upload 30MB images and the upload failed on our server limits. I added client-side validations, limited maximum file size, and introduced direct S3 uploads for very large media. For UX I hooked into Livewire upload events (`livewire-upload-start`, `progress`, `finish`) and used Alpine to render a progress bar. Users loved the visible progress; ops loved that big files no longer hit the app server.

Performance in listing pages pushed me to adopt server-side pagination and careful querying. Initially I was using Eloquent relationships with `->with('heavyRelation')` everywhere. Under load the pages got slow. The answer was to eager load judiciously and cache counts with `remember()` for non-critical stats. I also used Livewire's `WithPagination` trait and made sure to reset the page when filters changed (`$this->resetPage()`), otherwise users were sometimes left on an empty page after filtering.

Testing saved me from regressions. I wrote Livewire tests with `Livewire::test(ProductTable::class)` and asserted `->set('search','foo')->call('apply')->assertSee('Foo Product')`. Those tests were fast and reliable compared to full browser E2E. When a bug slipped through in a JS/Alpine interaction I added a small integration test and a manual QA checklist for modal flows.

Deployment taught me two operational lessons: enable proper session affinity for the load balancer (Livewire round trips rely on sessions) and make sure your front-end build artifacts and Livewire assets are deployed atomically. One night we had a mismatch between blade markup and the compiled assets after a rolling deploy, and Livewire updates misbehaved —

we fixed it by aligning deploy steps and clearing caches in the right order.

Finally, the maintainability payoff was real. When a new product manager asked for inline editing of product names, I built a tiny `InlineEdit` Livewire component, plugged it into every row, and shipped the feature in an afternoon. The team loved that we stayed in Blade, kept Laravel validation rules centralized, and avoided a bigger JS rewrite.

If I summarize the takeaways from that project: keep Livewire state small, debounce or defer inputs, pair Alpine for purely local UI, avoid serializing large models, use direct uploads for big files, test Livewire interactions, and ensure deploys preserve session/asset consistency. Livewire gave us fast developer velocity and a clean, maintainable UX — but only after wrestling with these practical pitfalls.

Installing Laravel Livewire

```
composer require livewire/livewireCode language: Bash (bash)
```

After installing, include Livewire's scripts and styles in your Blade layout:

```
<!-- resources/views/layouts/app.blade.php -->
<html lang="en">
<head>
  @livewireStyles
</head>
<body>
  <div class="container">
    @yield('content')
  </div>
  @livewireScripts
</body>
</html>Code language: PHP (php)
```

Now Livewire is available throughout your Laravel project.

Creating Your First Livewire Component

```
php artisan make:livewire CounterCode language: Bash (bash)
```

This generates two files:

```
app/Http/Livewire/Counter.php     // Component logic (PHP)
resources/views/livewire/counter.blade.php // Component view (Blade)
Code language: plaintext (plaintext)
```

```
// app/Http/Livewire/Counter.php
namespace App\Http\Livewire;

use Livewire\Component;

class Counter extends Component
{
    public $count = 0;

    public function increment()
    {
        $this->count++;
    }

    public function render()
    {
        return view('livewire.counter');
    }
}Code language: PHP (php)
```

```
<!-- resources/views/livewire/counter.blade.php -->
<div>
  <h2>Count: {{ $count }}</h2>
```

```
  <button wire:click="increment">Increment</button>
</div>Code language: PHP (php)
```

The PHP property `$count` syncs with Blade automatically. Clicking the button calls the `increment` method, Livewire updates the DOM, and no JavaScript is required.

Embedding Livewire in Blade

```
<!-- resources/views/welcome.blade.php -->
@extends('layouts.app')

@section('content')
  <h1>Welcome</h1>
  @livewire('counter')
@endsectionCode language: PHP (php)
```

Use the `@livewire('name')` directive to embed Livewire components into Blade views.

Using Livewire with Controllers

You can fetch data inside Livewire components the same way you would in a controller. For example, listing posts:

```
// app/Http/Livewire/PostList.php
namespace App\Http\Livewire;

use Livewire\Component;
```

```
use App\Models\Post;

class PostList extends Component
{
    public $posts = [];

    public function mount()
    {
        $this->posts = Post::latest()->get();
    }

    public function render()
    {
        return view('livewire.post-list');
    }
}Code language: PHP (php)
```

```
<!-- resources/views/livewire/post-list.blade.php -->
<div>
  <h2>Posts</h2>
  <ul>
    @foreach($posts as $post)
      <li>{{ $post->title }}</li>
    @endforeach
  </ul>
</div>Code language: PHP (php)
```

The `mount()` method acts like a controller constructor, loading data before rendering. Livewire keeps it reactive when the component updates.

Triggering Events with Livewire

Livewire supports emitting and listening to events between components, just like Laravel events. This enables parent-child communication without JavaScript.

```
// app/Http/Livewire/ChildComponent.php
namespace App\Http\Livewire;

use Livewire\Component;

class ChildComponent extends Component
{
    public function notifyParent()
    {
        $this->emit('childNotified', 'Child says hello!');
    }

    public function render()
    {
        return view('livewire.child-component');
    }
}Code language: PHP (php)
```

```
<!-- resources/views/livewire/child-component.blade.php -->
<button wire:click="notifyParent">Notify Parent</button>Code language: PHP (php)
```

```
// app/Http/Livewire/ParentComponent.php
namespace App\Http\Livewire;

use Livewire\Component;

class ParentComponent extends Component
{
    protected $listeners = ['childNotified'];

    public function childNotified($message)
    {
        session()->flash('info', $message);
```

```
    }

    public function render()
    {
        return view('livewire.parent-component');
    }
}Code language: PHP (php)
```

The parent listens to the `childNotified` event and reacts by flashing a session message. This mimics event-driven UIs with only PHP and Blade.

Building a Livewire Login Form

Let's build a login form UI with Livewire that handles validation and authentication directly.

```
// app/Http/Livewire/LoginForm.php
namespace App\Http\Livewire;

use Livewire\Component;
use Illuminate\Support\Facades\Auth;

class LoginForm extends Component
{
    public $email = '';
    public $password = '';

    protected $rules = [
        'email' => 'required|email',
        'password' => 'required|min:6',
    ];

    public function login()
    {
```

```
        $this->validate();

        if (Auth::attempt(['email' => $this->email, 'password' =>
$this->password])) {
            session()->regenerate();
            return redirect()->intended('/dashboard');
        }

        $this->addError('email', 'Invalid credentials.');
    }

    public function render()
    {
        return view('livewire.login-form');
    }
}Code language: PHP (php)
```

```
<!-- resources/views/livewire/login-form.blade.php -->
<form wire:submit.prevent="login">
  <div>
    <label for="email">Email</label>
    <input id="email" type="email" wire:model="email">
    @error('email') <span class="text-danger">{{ $message }}</span>
@enderror
  </div>

  <div>
    <label for="password">Password</label>
    <input id="password" type="password" wire:model="password">
    @error('password') <span class="text-danger">{{ $message }}</span>
@enderror
  </div>

  <button type="submit">Login</button>
</form>Code language: PHP (php)
```

The component handles input binding, validation, and authentication logic in PHP. No JavaScript is needed, but the UI feels dynamic thanks to Livewire.

Wrapping Up

Livewire bridges the gap between Laravel Blade and modern JavaScript frameworks. You can build interactive components, handle events, fetch data, and even implement authentication UIs without leaving PHP. It’s ideal for developers who want interactivity while staying in the Laravel ecosystem.

Livewire vs Vue: Which Should You Choose?

Laravel supports both Livewire and Vue 3. Which one you use depends on your project’s needs. Here’s a quick comparison:

Feature	Livewire	Vue 3
Learning Curve	Low — stays in PHP and Blade.	Higher — requires JS and Vue ecosystem knowledge.
Setup	Composer install, Blade directives.	Install via NPM + Vite config.
Use Cases	Forms, dashboards, CRUD, admin panels.	SPAs, real-time UIs, highly interactive apps.
Performance	Good, but frequent AJAX calls.	Excellent, reactive updates in the browser.
Flexibility	Bound to Laravel ecosystem.	Can integrate with any backend API.
Developer Audience	Laravel devs who prefer PHP-only solutions.	Teams with frontend specialists.

Recommendation: Use Livewire if you want Laravel-only productivity without diving deep into JavaScript. Use Vue 3 if your app needs SPA features, heavy interactivity, or has a dedicated frontend team.

What's Next

Want to go further with Livewire and interactive UIs? Try these guides:

- Creating a Role-Specific Dashboard in Laravel 12
- Creating a User-Friendly Roles & Permissions UI in Laravel
- Mastering Validation Rules in Laravel 12