Integrating Laravel with Vue 3 for a Modern Frontend

Laravel 12 integrates seamlessly with Vue 3 using Vite. This stack gives you Laravel's expressive backend and Vue's reactive UI, ideal for SPAs, dashboards, and highly interactive pages. In this guide, you'll install Vue 3 via Vite, understand the folder structure, import and mount components correctly, call them from Blade, fetch data from controllers, react to broadcast events, and build a practical login form UI with validation.

Install Vue 3 in a Laravel 12 Project (Vite Setup)

Laravel 12 uses Vite by default. Add Vue using the official plugin and enable it in your Vite config.

```
npm install vue @vitejs/plugin-vueCode language: Bash (bash)
```

This installs Vue 3 and the official Vite Vue plugin.

```
// vite.config.js
import { defineConfig } from 'vite';
import laravel from 'laravel-vite-plugin';
import vue from '@vitejs/plugin-vue';

export default defineConfig({
    plugins: [
        laravel({
            input: ['resources/js/app.js'],
            refresh: true,
        }),
        vue(),
    ],
});Code language: JavaScript (javascript)
```

This configuration tells Vite to compile `resources/js/app.js`, refresh on Blade changes, and process Vue single-file components.

Understanding Vue Folder Structure in Laravel

Your Vue code lives under `resources/js`. Here's the default layout after enabling Vue:

```
resources/
 └── js/
      ├── app.js          // Vue entry, mounts the app and registers components
      ├── bootstrap.js    // Axios, CSRF, Echo helpers (optional but useful)
      └── components/
           └── ExampleComponent.vue
Code language: plaintext (plaintext)
```

`app.js` is your entry file. You can mount a single root Vue component, register many components for use inside Blade, or auto-register all components in the folder for convenience.

Working with `app.js`: Three Practical Patterns

1) Mount a Single Root Component (SPA-style)

```
// resources/js/app.js
import './bootstrap';
```

```
import { createApp } from 'vue';
import ExampleComponent from './components/ExampleComponent.vue';

createApp(ExampleComponent).mount('#app');Code language: JavaScript (javascript)
```

Use this when Vue controls the entire page/app (typical SPA approach).

2) Register Multiple Components (Blade + Vue hybrid)

```
// resources/js/app.js
import './bootstrap';
import { createApp } from 'vue';
import ExampleComponent from './components/ExampleComponent.vue';
import PostList from './components/PostList.vue';

const app = createApp({});
app.component('example-component', ExampleComponent);
app.component('post-list', PostList);
app.mount('#app');Code language: JavaScript (javascript)
```

This lets you drop Vue widgets into Blade without making the whole page a SPA.

3) Auto-register All Components (scale easily)

```
// resources/js/app.js
import './bootstrap';
import { createApp } from 'vue';

const app = createApp({});

// Auto-register ./components/*.vue as <ComponentName />
Object.entries(import.meta.glob('./components/*.vue', { eager: true
})).forEach(([path, def]) => {
    const name = path.split('/').pop().replace('.vue', '');
    app.component(name, def.default);
});

app.mount('#app');Code language: JavaScript (javascript)
```

Now any `.vue` you add under `components/` becomes available by its filename (e.g., `PostList.vue` → `<PostList></PostList>`).

Create and Call a Vue Component from Blade

```
// resources/js/components/ExampleComponent.vue
<template>
  <div class="p-4">
    <h2>Hello from Vue 3!</h2>
    <button @click="count++">Clicked {{ count }} times</button>
  </div>
</template>

<script setup>
import { ref } from 'vue';
const count = ref(0);
</script>Code language: JavaScript (javascript)
```

A simple counter using the Composition API. Next, call it from a Blade view and ensure Vite loads `app.js`.

```
// resources/views/welcome.blade.php
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Laravel + Vue 3</title>
  @vite('resources/js/app.js')
</head>
<body>
  <div id="app">
    <example-component></example-component>
  </div>
```

```
</body>
</html>Code language: PHP (php)
```

With pattern #2 or #3 in `app.js`, the component renders where you place `<example-component>` inside the `#app` container.

Fetch Data from a Controller and Render in Vue

```
// app/Http/Controllers/PostController.php
namespace App\Http\Controllers;

use App\Models\Post;

class PostController extends Controller
{
    public function index()
    {
        return response()->json(Post::latest()->get());
    }
}Code language: PHP (php)
```

A simple endpoint that returns posts as JSON for Vue to consume.

```
// resources/js/components/PostList.vue
<template>
  <div>
    <h2 class="mb-2">Posts</h2>
    <ul>
      <li v-for="post in posts" :key="post.id">{{ post.title }}</li>
    </ul>
  </div>
</template>
```

```
<script setup>
import { ref, onMounted } from 'vue';
const posts = ref([]);

onMounted(async () => {
  const res = await fetch('/posts');
  posts.value = await res.json();
});
</script>Code language: JavaScript (javascript)
```

The component fetches data on mount and renders a reactive list. Drop `<post-list>` into Blade to display it.

Realtime UX with Events (Broadcast + Vue)

Broadcast Laravel events and listen in Vue to update the UI instantly (e.g., when a new post is created).

```
// app/Events/NewPostCreated.php
namespace App\Events;

use App\Models\Post;
use Illuminate\Broadcasting\Channel;
use Illuminate\Contracts\Broadcasting\ShouldBroadcast;

class NewPostCreated implements ShouldBroadcast
{
    public $post;

    public function __construct(Post $post)
    {
        $this->post = $post;
    }
```

```
    public function broadcastOn()
    {
        return new Channel('posts');
    }
}Code language: PHP (php)
```

This event publishes to the public `posts` channel. Configure Laravel Echo (Pusher or Ably) in `bootstrap.js` to subscribe from Vue.

```
// resources/js/components/PostList.vue (add listener)
<script setup>
import { ref, onMounted } from 'vue';

const posts = ref([]);

onMounted(async () => {
  const res = await fetch('/posts');
  posts.value = await res.json();

  if (window.Echo) {
    window.Echo.channel('posts')
      .listen('NewPostCreated', (e) => {
        posts.value.unshift(e.post);
      });
  }
});
</script>Code language: JavaScript (javascript)
```

When `NewPostCreated` is fired on the server, the UI prepends the new post instantly for users connected to the channel.

Build a Vue Login Form UI (Controller + Validation)

Let’s create a practical login UI in Vue that talks to a Laravel controller, validates input, and shows server error messages cleanly.

```
// resources/js/components/LoginForm.vue
<template>
  <form @submit.prevent="submit" class="max-w-sm">
    <h2 class="mb-3">Login</h2>

    <div class="mb-2">
      <label for="email">Email</label>
      <input id="email" v-model="form.email" type="email" required />
      <p v-if="errors.email" class="text-danger">{{ errors.email
}}</p>
    </div>

    <div class="mb-3">
      <label for="password">Password</label>
      <input id="password" v-model="form.password" type="password"
required />
      <p v-if="errors.password" class="text-danger">{{ errors.password
}}</p>
    </div>

    <button type="submit">Sign in</button>
    <p v-if="errors.global" class="text-danger mt-2">{{ errors.global
}}</p>
  </form>
</template>

<script setup>
import { reactive } from 'vue';

const form = reactive({ email: '', password: '' });
const errors = reactive({ email: '', password: '', global: '' });

async function submit() {
```

```
  errors.email = errors.password = errors.global = '';

  const res = await fetch('/login', {
    method: 'POST',
    headers: {
      'Content-Type': 'application/json',
      'X-CSRF-TOKEN': document.querySelector('meta[name="csrf-
token"]').content
    },
    body: JSON.stringify(form)
  });

  if (res.ok) {
    window.location.href = '/dashboard';
    return;
  }

  if (res.status === 422) {
    const data = await res.json();
    errors.email = data.errors?.email?.[0] || '';
    errors.password = data.errors?.password?.[0] || '';
  } else {
    errors.global = 'Invalid credentials or server error.';
  }
}
</script>Code language: JavaScript (javascript)
```

The component posts JSON to `/login` with a CSRF token, handles validation (422) errors, and redirects on success.

```
// resources/views/auth/login.blade.php
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta name="csrf-token" content="{{ csrf_token() }}">
  <title>Login</title>
  @vite('resources/js/app.js')
```

```
</head>
<body>
  <div id="app">
    <LoginForm></LoginForm>
  </div>
</body>
</html>Code language: PHP (php)
```

The Blade view provides the CSRF token meta tag and mounts the `<LoginForm>` component inside `#app`.

```
// app/Http/Controllers/Auth/LoginController.php
namespace App\Http\Controllers\Auth;

use App\Http\Controllers\Controller;
use Illuminate\Http\Request;
use Illuminate\Support\Facades\Auth;
use Illuminate\Validation\ValidationException;

class LoginController extends Controller
{
    public function __invoke(Request $request)
    {
        $validated = $request->validate([
            'email' => 'required|email',
            'password' => 'required|string|min:6',
        ]);

        if (Auth::attempt($validated, true)) {
            $request->session()->regenerate();
            return response()->json(['ok' => true]);
        }

        throw ValidationException::withMessages([
            'email' => ['These credentials do not match our
records.'],
        ]);
    }
```

```
}Code language: PHP (php)
```

The single-action controller validates the request, attempts login, regenerates the session, and returns JSON. On failure it throws a 422 with field errors that Vue shows inline.

```
// routes/web.php
use App\Http\Controllers\Auth\LoginController;

Route::view('/login', 'auth.login')->name('login');
Route::post('/login', LoginController::class);
Route::get('/dashboard', fn () =>
view('dashboard'))->middleware('auth');Code language: PHP (php)
```

These routes render the login page, handle the POST login, and protect the dashboard behind `auth` middleware.

Related deep dives: Mastering Validation Rules in Laravel 12, How to Add Authentication in Laravel 12 (Without Fortify).

Wrapping Up

You integrated Vue 3 into Laravel with Vite, learned the folder structure, registered and mounted components, called them from Blade, fetched data from controllers, reacted to broadcast events, and built a production-ready login form UI with validation. This Laravel + Vue approach pairs a clean backend with a delightful, reactive frontend—perfect for modern apps.

What’s Next

Level up your stack with these related guides:

- Laravel with Docker & Sail: The Right Way
- Using Laravel Telescope to Debug Performance Issues
- Mastering Validation Rules in Laravel 12