Contents

Setting up a project environment 3
Introduction to Inertia.js 3
What is Inertia.js? 3
Comparison: Blade vs. SPA vs. Inertia 3
Setup (Lab Guide) 4
Prerequisites 4
Install Laravel 12 4
Server-Side Setup (Inertia) 4
Client-Side Setup (Vue 3) 5
Inertia Basics 7
Creating your first Vue page (Dashboard.vue) 7
Returning Inertia responses from Laravel routes 7
Passing props from Laravel controllers to Vue 8
Authentication Workflow 8
Setup Laravel Breeze with Inertia + Vue stack 8
User Registration, Login, and Logout 9
Protect routes with Laravel middleware 9
Display authenticated user in Vue 10
Vue 3 Core Concepts 10
Composition API (ref, reactive, computed) 10
Components & Props 11
Events (emit) 12
Lifecycle Hooks 12
Vue Router with Inertia 12
Navigating between pages 12
Shared Layouts (Persistent Layouts) 13
Route Guards 13
Hands-on Exercise: User Profile Update 14
CRUD Integration & Project Deployment 16

Forms and Validation 16
The useForm Helper 16
Validation with Laravel 17
Flash Messages 17
CRUD Project Implementation 17
Database Setup (MySQL) 17
Backend (Laravel) 18
Frontend (Vue + Inertia) 20
UI Enhancements 23
Deployment Guide 27
Server Prerequisites (LEMP Stack) 27
Application Setup 27
Building Assets 28
Nginx Configuration 28
Permissions Fix 29
SSL (HTTPS) 29
On demonstrating certbot without a domain 30
Recommended Project Architectural and Structure Setup 31
1. The High-Level Architecture 31
2. Recommended Directory Structure 31
3. Key Architectural Patterns 32
Dockerizing the app and deploying to docker hub (Windows) 34

Setting up a project environment

1. Create a GitHub Account and Prepare GitHub Personal Access Token
    a. Go to Github→click on the Avatar Icon→settings→developer settings→personal access token→token
    b. Set token permissions then take note of the token, it may be used when prompted for a GitHub password

2. Download and install NodeJS
3. Download and install vscode
4. Add the following extensions:

Postman, PHP Intelephense, Laravel Snippets, Laravel Extra Intellisense, Vue, Vetur

5. In VS Code, click the Source Control tool from the left panel and configure the GitHub repository connection for the project.

Introduction to Inertia.js

What is Inertia.js?

Inertia is not a framework; it is a "glue" library. It allows you to build single-page apps (SPAs) using classic server-side routing and controllers.

Think of it as the "Modern Monolith."

- The Old Way (Blade): The server sends a full HTML page every time you click a link.
- The API Way (Standard SPA): You build a separate API (Laravel) and a separate Frontend (Vue). You have to manage tokens, loading states, and duplicate validation logic.
- The Inertia Way: You write Vue components, but you use Laravel routes and controllers just like you would with Blade. Inertia intercepts the links and swaps the components without a full page reload.[2]

Comparison: Blade vs. SPA vs. Inertia

Use this table to help students visualize the trade-offs:

Feature	Laravel + Blade	Laravel + API (SPA)	Laravel + Inertia
Experience	Multi-page (Full Reloads)	Single-page (Instant)	Single-page (Instant)
Routing	Laravel Routes (web.php)	Vue Router (Client-side)	Laravel Routes (web.php)
Data Fetching	Passed to View	API Calls (Axios/Fetch)	Passed to Component as Props
Complexity	Low	High (Authentication/State)	Medium (Best of both)
SEO	Excellent	Difficult (Needs SSR)	Good (can use SSR)

Setup (Lab Guide)

Prerequisites

- PHP 8.2+
- Composer
- Node.js & NPM

Install Laravel 12

```
composer create-project laravel/laravel my-app
cd my-app
```

Server-Side Setup (Inertia)

First, we install the PHP adapter for Inertia.
#1. Install the package:

```
composer require inertiajs/inertia-laravel
```

#2. Setup the Root Template:
Rename
resources/views/welcome.blade.php

to
resources/views/app.blade.php.

Replace its content with the Inertia root hook:

```
<!DOCTYPE html>
<html>
 <head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0, maximum-scale=1.0" />
  @vite('resources/js/app.js')
  @inertiaHead
 </head>
 <body>
  @inertia
 </body>
</html>
```

#3. Generate the Middleware:
Inertia needs middleware to handle requests.

```
php artisan inertia:middleware
```

Note: In Laravel 11/12, append this middleware to your bootstrap/app.php file.

Client-Side Setup (Vue 3)

a. Install Dependencies:

```
npm install @inertiajs/vue3 vue @vitejs/plugin-vue
```

b. Initialize the App:

Open resources/js/app.js and paste the standard initialization code:

```
import { createApp, h } from 'vue'
import { createInertiaApp } from '@inertiajs/vue3'

createInertiaApp({
 resolve: name => {
  const pages = import.meta.glob('./Pages/**/*.vue', { eager: true })
  return pages[`./Pages/${name}.vue`]
 },
 setup({ el, App, props, plugin }) {
  createApp({ render: () => h(App, props) })
   .use(plugin)
   .mount(el)
 },
})
```

2. Install TailwindCSS

We will use Tailwind for utility-first styling.

a. Install and Init:

```
npm install -D tailwindcss@3 postcss autoprefixer
npx tailwindcss init -p
```

b. Configure Paths:

Update tailwind.config.js to look for your Vue files:

```
/** @type {import('tailwindcss').Config} */
export default {
 content: [
  "./resources/**/*.blade.php",
  "./resources/**/*.js",
  "./resources/**/*.vue",
 ],
 theme: {
  extend: {},
 },
 plugins: [],
}
```

c. Add Directives:
Create a CSS file at resources/css/app.css and add:

```
@tailwind base;
@tailwind components;
@tailwind utilities;
```

(Ensure this CSS file is imported in your resources/js/app.js via import '../css/app.css';)

3. Configure Vite

Finally, wire everything together in 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/css/app.css', 'resources/js/app.js'],
      refresh: true,
    }),
    vue({
      template: {
        transformAssetUrls: {
          base: null,
          includeAbsolute: false,
        },
      },
    }),
  ],
});
```

4. Verification
   Run the development servers in two separate terminals:

   Terminal 1 (PHP):

```
php artisan serve
```

   Terminal 2 (Node/Vite):

```
npm run dev
```

   If the page loads without errors (it might show a blank screen or 404 until we create the first Page), the environment is ready!

Inertia Basics

Inertia allows you to build a single-page app (SPA) while keeping the routing and controller logic in Laravel. It acts as the glue connecting your backend to your frontend.

Creating your first Vue page (Dashboard.vue)

In a standard Laravel Inertia setup, your pages live in resources/js/Pages.

1. Create a new file: resources/js/Pages/Dashboard.vue.
2. Add a basic template. Note that you don't need a typical HTML shell (<html>, <body>) because the main app layout handles that.

```
<template>
  <div class="p-6">
    <h1 class="text-2xl font-bold mb-4">Dashboard</h1>
    <p>Welcome to your new Inertia app!</p>
  </div>
</template>

<script setup>
// Logic goes here
</script>
```

Returning Inertia responses from Laravel routes

Instead of returning a standard Blade view(), you return an Inertia::render() response.

Open routes/web.php and define the route:

```
use Illuminate\Support\Facades\Route;
use Inertia\Inertia;

Route::get('/dashboard', function () {
   // 'Dashboard' corresponds to resources/js/Pages/Dashboard.vue
   return Inertia::render('Dashboard');
});
```

Passing props from Laravel controllers to Vue

This is where Inertia shines. You can pass data from your database directly to your frontend component as if it were a standard Blade view variable.

1. Update the Route (or Controller):

```
Route::get('/dashboard', function () {
   return Inertia::render('Dashboard', [
      'userCount' => 150,
      'recentActivity' => 'Server restart at 10:00 AM',
   ]);
});
```

2. Accept the Props in Dashboard.vue:

```
<script setup>
// Define the props you expect from Laravel
defineProps({
   userCount: Number,
   recentActivity: String,
});
</script>

<template>
  <div class="p-6">
   <h1 class="text-2xl font-bold">Dashboard</h1>

   <div class="mt-4 bg-gray-100 p-4 rounded">
      <p><strong>Total Users:</strong> {{ userCount }}</p>
      <p><strong>Latest Log:</strong> {{ recentActivity }}</p>
   </div>
  </div>
</template>
```

Authentication Workflow

While you can build authentication manually, the Laravel Breeze starter kit is the industry standard for quickly scaffolding a robust authentication system specifically for Vue and Inertia.

Setup Laravel Breeze with Inertia + Vue stack

Run the following commands in your terminal to install Breeze and configure it for Vue:

```
composer require laravel/breeze --dev
```

Install Breeze for Vue with Server-Side Rendering (optional but recommended)

```
php artisan breeze:install vue
```

Run migrations to create the users table

```
php artisan migrate
```

Compile your assets

```
npm run dev
```

Note: This process automatically generates all the necessary login, registration, and password reset pages in resources/js/Pages/Auth.

User Registration, Login, and Logout

To use tailwind forms, install

```
npm install -D @tailwindcss/forms --legacy-peer-deps
```

Breeze sets up these routes automatically in routes/auth.php.

- Registration: Navigate to /register. The form submits to the backend, creates the user, and logs them in.
- Login: Navigate to /login. Upon success, Laravel regenerates the session.
- Logout: This usually requires a POST request for security.

Example Logout Link (Vue): Inertia uses the Link component to handle method spoofing for non-GET requests.

```
<script setup>
import { Link } from '@inertiajs/vue3';
</script>

<template>
  <Link href="/logout" method="post" as="button" type="button">
    Log Out
  </Link>
</template>
```

Protect routes with Laravel middleware

To prevent unauthenticated users from accessing the Dashboard, wrap the route in the auth middleware group in routes/web.php.

```
Route::middleware(['auth', 'verified'])->group(function () {
  Route::get('/dashboard', function () {
    return Inertia::render('Dashboard');
  })->name('dashboard');
});
```

If a user tries to hit /dashboard without being logged in, Laravel will automatically redirect them to the /login page.

Display authenticated user in Vue

Inertia shares the authenticated user globally via the $page object (usually configured in HandleInertiaRequests.php middleware). You don't need to pass the user manually from every controller.

Accessing the User in Template:

```
<template>
  <nav>
    Welcome back, {{ $page.props.auth.user.name }}!
  </nav>
</template>
```

Accessing the User in Script:

```
<script setup>
import { usePage } from '@inertiajs/vue3';
import { computed } from 'vue';

const page = usePage();
const user = computed(() => page.props.auth.user);

console.log(user.value.email);
</script>
```

Vue 3 Core Concepts

The Composition API is the modern standard for writing Vue components. It groups logical concerns together rather than splitting them by options (data, methods, mounted, etc.).

Composition API (ref, reactive, computed)

These are the building blocks of your component's state.

- ref() :Used for primitive values (strings, numbers, booleans). You must access the value using .value in the script, but just the name in the template.
- reactive() :Used for objects or arrays. You don't need .value.
- computed() :Derives a value based on other state; it auto-updates when dependencies change.

```
<script setup>
import { ref, reactive, computed } from 'vue';

// State
const count = ref(0);
const user = reactive({ name: 'Alice', role: 'Admin' });

// Computed Property
```

```
const doubleCount = computed(() => count.value * 2);

// Method
function increment() {
  count.value++; // Note the .value
}
</script>

<template>
  <button @click="increment">Count is: {{ count }}</button>
  <p>Double is: {{ doubleCount }}</p>
  <p>User: {{ user.name }}</p>
</template>
```

Components & Props

Props allow parent components to pass data down to children.

Child Component (Button.vue):

```
<script setup>
defineProps({
  label: String,
  type: {
    type: String,
    default: 'submit'
  }
});
</script>

<template>
  <button :type="type" class="bg-blue-500 text-white px-4 py-2 rounded">
    {{ label }}
  </button>
</template>
```

Parent Component:

```
<Button label="Save Changes" type="submit" />
```

Events (emit)

Events allow children to communicate up to parents.

```
<script setup>
const emit = defineEmits(['update']);

function handleClick() {
  emit('update', 'New Value');
}
</script>
```

Lifecycle Hooks

These allow you to run code at specific times. onMounted is the most common (e.g., for fetching data via an API if not provided by Inertia, or initializing third-party libraries like charts).

```
<script setup>
import { onMounted } from 'vue';

onMounted(() => {
  console.log('Component is now mounted to the DOM!');
});
</script>
```

Vue Router with Inertia

- In a standard Vue SPA, you use vue-router.
- In an Inertia app, Inertia is the router. You do not need a router.js file.
- You define routes in Laravel (routes/web.php), and Inertia handles the page switching on the frontend.

Navigating between pages

Use the <Link> component provided by Inertia. It intercepts the click, prevents a full page reload, and fetches the new page component via XHR (AJAX).

```
<script setup>
import { Link } from '@inertiajs/vue3';
</script>

<template>
  <nav>
    <Link href="/">Home</Link>
    <Link href="/about" class="ml-4 text-blue-500">About</Link>
  </nav>
</template>
```

Shared Layouts (Persistent Layouts)

If you have a sidebar or navbar that should not disappear/reload when navigating between pages, you use Persistent Layouts.

1. Create a Layout (resources/js/Layouts/MainLayout.vue):

```
<template>
 <div class="min-h-screen bg-gray-100">
  <nav class="bg-white p-4 shadow">My App</nav>
  <main class="p-6">
   <slot /> </main>
 </div>
</template>
```

2. Assign Layout in Page Component:

```
<script setup>
import MainLayout from '@/Layouts/MainLayout.vue';

defineOptions({ layout: MainLayout });
</script>

<template>
 <h1>Dashboard Content</h1>
</template>
```

Route Guards

Since routes are defined in Laravel, "guards" are primarily handled by Laravel Middleware.

- Server-side: Use Route::middleware('auth') in Laravel. If a user isn't logged in, Laravel redirects them before the Vue page even loads.
- Client-side: If you need to restrict access based on user role *inside* a template (e.g., hiding a button), check the user prop:

```
<button v-if="$page.props.auth.user.role === 'admin'">
  Delete User
</button>
```

Hands-on Exercise: User Profile Update

Let's build a feature where a user can update their name and email. We will use the Inertia Form Helper, which automatically handles loading states and validation errors.

1. The Laravel Route & Controller

Open routes/web.php:

```
use App\Http\Controllers\ProfileController;

Route::middleware('auth')->group(function () {
   Route::get('/profile', [ProfileController::class, 'edit'])->name('profile.edit');
   Route::patch('/profile', [ProfileController::class, 'update'])->name('profile.update');
});
```

(Assuming you have a ProfileController. If not, php artisan make:controller ProfileController)

2. The Vue Page

Create resources/js/Pages/Profile/Edit.vue.

```
<script setup>
import { useForm } from '@inertiajs/vue3';

// 1. Receive the user data from Laravel
const props = defineProps({
   user: Object
});

// 2. Initialize the form with existing data
const form = useForm({
   name: props.user.name,
   email: props.user.email,
});

// 3. Handle submission
function submit() {
   form.patch('/profile', {
      preserveScroll: true,
      onSuccess: () => alert('Profile updated!'),
   });
}
</script>

<template>
  <div class="max-w-md mx-auto p-6 bg-white rounded shadow mt-10">
   <h1 class="text-xl font-bold mb-4">Edit Profile</h1>
   <form @submit.prevent="submit">
    <div class="mb-4">
      <label class="block text-gray-700">Name</label>
      <input
       v-model="form.name"
       type="text"
       class="border p-2 w-full rounded"
      />
      <div v-if="form.errors.name" class="text-red-500 text-sm mt-1">
        {{ form.errors.name }}
      </div>
    </div>

    <div class="mb-4">
      <label class="block text-gray-700">Email</label>
      <input
       v-model="form.email"
       type="email"
       class="border p-2 w-full rounded"
      />
      <div v-if="form.errors.email" class="text-red-500 text-sm mt-1">
        {{ form.errors.email }}
      </div>
    </div>

    <button
      type="submit"
      :disabled="form.processing"
      class="bg-blue-600 text-white px-4 py-2 rounded hover:bg-blue-500 disabled:opacity-50"
    >
      {{ form.processing ? 'Saving...' : 'Update Profile' }}
    </button>
   </form>
  </div>
</template>
```

3. Handle the Update in Laravel

In your ProfileController.php:

```
public function edit(Request $request)
{
   return Inertia::render('Profile/Edit', [
      'user' => $request->user()
   ]);
}
```

```
public function update(Request $request)
{
   $validated = $request->validate([
      'name' => 'required|string|max:255',
      'email' => 'required|email|max:255|unique:users,email,' . $request->user()->id,
   ]);

   $request->user()->update($validated);

   // Inertia will automatically flash errors if validation fails.
   // If successful, we redirect back.
   return redirect()->route('profile.edit');
}
```

CRUD Integration & Project Deployment

Forms and Validation

Inertia wrappers standard HTML form submissions to provide a single-page app experience.

The useForm Helper

The useForm helper is the core component for handling data. It maintains the state of your fields, errors, and processing status.

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

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

// submitting
form.post('/posts');
Key Properties of form:

form.data(): The current data object.

form.processing: Boolean, true while the request is sending (use this to disable buttons).

form.errors: Object containing validation error messages from Laravel.

form.reset(): Resets fields to their original state.
```

Validation with Laravel

You don't need to write complex validation logic in Vue. Define it in Laravel, and if it fails, Laravel redirects back with errors that Inertia automatically catches.

Flash Messages

To show "Success" notifications, we pass data from the Laravel session to the Vue frontend via the HandleInertiaRequests middleware.

Middleware (app/Http/Middleware/HandleInertiaRequests.php):

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

CRUD Project Implementation

We will build a "Posts" management system.

Database Setup (MySQL)

Create the Database: Open your MySQL terminal or a tool like phpMyAdmin/HeidiSQL and create a new database (e.g., inertia_blog).

Configure .env: Update your Laravel .env file to connect to this database.

```
DB_CONNECTION=mysql
DB_HOST=127.0.0.1
DB_PORT=3306
DB_DATABASE=inertia_blog
DB_USERNAME=root
DB_PASSWORD=
```

Backend (Laravel)

1. Model & Migration Run the command to generate the Model, Migration, and Controller:

```
php artisan make:model Post -mcr
```

(The -mcr flag creates the Migration, Controller, and Resource)

Edit Migration (database/migrations/xxxx_create_posts_table.php):

```
Schema::create('posts', function (Blueprint $table) {
    $table->id();
    $table->foreignId('user_id')->constrained()->cascadeOnDelete();
    $table->string('title');
    $table->text('content');
    $table->timestamps();
});
```

Run php artisan migrate to apply changes.

```
php artisan migrate
```

Edit Model (app/Models/Post.php):

```
class Post extends Model
{
    protected $fillable = ['title', 'content', 'user_id'];

    public function user()
    {
        return $this->belongsTo(User::class);
    }
}
```

2. Policies (Authorization) Ensure users can only edit/delete their own posts.

```
php artisan make:policy PostPolicy --model=Post
```

Edit Policy (app/Policies/PostPolicy.php):

```
public function update(User $user, Post $post)
{
    return $user->id === $post->user_id;
}

public function delete(User $user, Post $post)
{
    return $user->id === $post->user_id;
}
```

3. Controller Implementation (PostController.php)

```
use App\Models\Post;
use Inertia\Inertia;
use Illuminate\Http\Request;

class PostController extends Controller
{
  public function index()
  {
    // Fetch posts with the associated user to display names
    return Inertia::render('Posts/Index', [
      'posts' => Post::with('user')->latest()->get()
    ]);
  }

  public function store(Request $request)
  {
    $validated = $request->validate([
      'title' => 'required|string|max:255',
      'content' => 'required|string',
    ]);

    $request->user()->posts()->create($validated);

    return redirect()->route('posts.index')->with('message', 'Post Created Successfully!');
  }

  public function update(Request $request, Post $post)
  {
    $this->authorize('update', $post);

    $validated = $request->validate([
      'title' => 'required|string|max:255',
      'content' => 'required|string',
    ]);

    $post->update($validated);

    return redirect()->route('posts.index')->with('message', 'Post Updated!');
  }

  public function destroy(Post $post)
  {
    $this->authorize('delete', $post);
    $post->delete();
```

```
        return redirect()->route('posts.index')->with('message', 'Post Deleted!');
    }
}
```

4. Routes (routes/web.php)

```
use App\Http\Controllers\PostController;

Route::middleware(['auth', 'verified'])->group(function () {
    Route::resource('posts', PostController::class);
});
```

Frontend (Vue + Inertia)

We will create a Single View (Index.vue) that handles listing and creating, and an Inline Edit feature for simplicity and speed.

resources/js/Pages/Posts/Index.vue

```
<script setup>
import { ref } from 'vue';
import { useForm, usePage, Head } from '@inertiajs/vue3';
import AuthenticatedLayout from '@/Layouts/AuthenticatedLayout.vue';

// 1. Props from Laravel
defineProps(['posts']);

// 2. Form for Creating a Post
const form = useForm({
    title: '',
    content: '',
});

// 3. State for Editing
const isEditing = ref(null); // Stores the ID of the post being edited
const editForm = useForm({
    title: '',
    content: '',
});

// 4. Methods
const submitCreate = () => {
    form.post(route('posts.store'), {
        onSuccess: () => form.reset(),
    });
};
```

```
const startEdit = (post) => {
  isEditing.value = post.id;
  editForm.title = post.title;
  editForm.content = post.content;
};

const submitUpdate = (post) => {
  editForm.put(route('posts.update', post.id), {
    onSuccess: () => isEditing.value = null,
  });
};

const deletePost = (post) => {
  if (confirm('Are you sure you want to delete this post?')) {
    form.delete(route('posts.destroy', post.id));
  }
};
</script>

<template>
  <Head title="Posts" />

  <AuthenticatedLayout>
    <template #header>
      <h2 class="font-semibold text-xl text-gray-800 leading-tight">Community Posts</h2>
    </template>

    <div class="max-w-2xl mx-auto p-4 sm:p-6 lg:p-8">

      <div v-if="$page.props.flash.message" class="p-4 mb-4 text-sm text-green-700 bg-green-100
rounded-lg">
        {{ $page.props.flash.message }}
      </div>

      <form @submit.prevent="submitCreate" class="bg-white p-6 rounded-lg shadow-md mb-6">
        <input
          v-model="form.title"
          placeholder="Post Title"
          class="block w-full border-gray-300 focus:border-indigo-300 focus:ring focus:ring-indigo-200
focus:ring-opacity-50 rounded-md shadow-sm mb-3"
        >
        <div v-if="form.errors.title" class="text-red-600 text-sm mb-2">{{ form.errors.title }}</div>

        <textarea
```

```
          v-model="form.content"
          placeholder="What's on your mind?"
          class="block w-full border-gray-300 focus:border-indigo-300 focus:ring focus:ring-indigo-200
focus:ring-opacity-50 rounded-md shadow-sm"
        ></textarea>
        <div v-if="form.errors.content" class="text-red-600 text-sm mt-2">{{ form.errors.content }}</div>

        <button
          class="mt-4 px-4 py-2 bg-gray-800 text-white rounded hover:bg-gray-700"
          :disabled="form.processing"
        >
          Post
        </button>
      </form>

      <div class="bg-white shadow-sm rounded-lg divide-y">
        <div v-for="post in posts" :key="post.id" class="p-6 flex space-x-2">
          <div class="flex-1">
            <div class="flex justify-between items-center">
              <div>
                <span class="text-gray-800 font-bold">{{ post.user.name }}</span>
                <span class="text-gray-600 text-sm ml-2">{{ new
Date(post.created_at).toLocaleDateString() }}</span>
              </div>

              <div v-if="post.user_id === $page.props.auth.user.id">
                <button v-if="isEditing !== post.id" @click="startEdit(post)" class="text-blue-600
hover:underline mr-3">Edit</button>
                <button v-if="isEditing !== post.id" @click="deletePost(post)" class="text-red-600
hover:underline">Delete</button>
              </div>
            </div>

            <div v-if="isEditing === post.id" class="mt-4">
              <form @submit.prevent="submitUpdate(post)">
                <input v-model="editForm.title" class="block w-full border-gray-300 rounded mb-2">
                <textarea v-model="editForm.content" class="block w-full border-gray-300
rounded"></textarea>
                <div class="mt-2 space-x-2">
                  <button type="submit" class="bg-blue-600 text-white px-3 py-1
rounded">Save</button>
                  <button @click="isEditing = null" type="button" class="bg-gray-300 text-gray-700 px-3
py-1 rounded">Cancel</button>
                </div>
              </form>
            </div>

            <div v-else class="mt-4">
              <h3 class="text-lg font-bold text-gray-900">{{ post.title }}</h3>
              <p class="mt-2 text-gray-800 whitespace-pre-wrap">{{ post.content }}</p>
            </div>
          </div>
        </div>
      </div>
    </div>
   </AuthenticatedLayout>
</template>
```

UI Enhancements

Upgrade your existing "Posts" project with professional UI tables, Pagination, and Real-time Search functionality.

Backend: Controller Updates
We need to update the PostController to handle search queries and return paginated data instead of a simple get() list.

Open app/Http/Controllers/PostController.php and update the index method:

```
public function index(Request $request)
{
  // 1. Start the query
  $query = Post::with('user')->latest();

  // 2. Apply Search Filter if present
  if ($request->input('search')) {
    $query->where('title', 'like', '%' . $request->input('search') . '%')
        ->orWhere('content', 'like', '%' . $request->input('search') . '%');
  }

  // 3. Return Paginated Resource (10 items per page)
  return Inertia::render('Posts/Index', [
    'posts' => $query->paginate(10)->withQueryString(), // Maintains search params in pagination links
    'filters' => $request->only(['search']), // Pass search term back to Vue to repopulate the input
  ]);
}
```

Frontend: Pagination Component
Inertia pagination links are slightly complex structures (containing url, label, and active states). It's best to create a reusable component.

Create: resources/js/Components/Pagination.vue

```
<script setup>
import { Link } from '@inertiajs/vue3';

defineProps({
  links: Array,
});
</script>

<template>
  <div v-if="links.length > 3" class="flex flex-wrap -mb-1">
    <template v-for="(link, key) in links" :key="key">
      <div
        v-if="link.url === null"
        class="mr-1 mb-1 px-4 py-3 text-sm leading-4 text-gray-400 border rounded"
        v-html="link.label"
      />

      <Link
        v-else
        :href="link.url"
        class="mr-1 mb-1 px-4 py-3 text-sm leading-4 border rounded hover:bg-white focus:border-
indigo-500 focus:text-indigo-500"
        :class="{ 'bg-blue-700 text-white': link.active, 'bg-white': !link.active }"
        v-html="link.label"
      />
    </template>
  </div>
</template>
```

Frontend: Index Page Overhaul
We will replace the simple list in resources/js/Pages/Posts/Index.vue with a data table, a search bar, and our new pagination.

Updates to Script Section: We need to watch the search input and trigger a reload.

```
<script setup>
import { ref, watch } from 'vue';
import { useForm, router } from '@inertiajs/vue3'; // Import router for manual visits
import AuthenticatedLayout from '@/Layouts/AuthenticatedLayout.vue';
import Pagination from '@/Components/Pagination.vue';
import debounce from 'lodash/debounce'; // Standard in Laravel scaffolding
```

```
// Props now include 'filters'
const props = defineProps({
  posts: Object,
  filters: Object
});

// Initialize search with value from backend
const search = ref(props.filters.search || '');

// Watch search and reload automatically
// Debounce waits 300ms after typing stops to prevent server overload
watch(search, debounce((value) => {
  router.get(route('posts.index'), { search: value }, {
    preserveState: true, // Keep the user's scroll position and component state
    replace: true,      // Don't clutter browser history
  });
}, 300));

// ... (Keep your existing delete/edit logic here) ...
</script>
```

Updates to Template Section (The Table UI):

```
<template>
  <AuthenticatedLayout>
    <div class="max-w-7xl mx-auto p-4 sm:p-6 lg:p-8">

      <div class="flex justify-between items-center mb-6">
        <h2 class="text-xl font-semibold">Manage Posts</h2>
        <input
          v-model="search"
          type="text"
          placeholder="Search posts..."
          class="border border-gray-300 rounded px-4 py-2 w-64"
        >
      </div>

      <div class="bg-white overflow-hidden shadow-sm sm:rounded-lg overflow-x-auto">
        <table class="min-w-full divide-y divide-gray-200">
          <thead class="bg-gray-50">
            <tr>
              <th class="px-6 py-3 text-left text-xs font-medium text-gray-500 uppercase tracking-
wider">Title</th>
              <th class="px-6 py-3 text-left text-xs font-medium text-gray-500 uppercase tracking-
wider">Author</th>
```

```
                    <th class="px-6 py-3 text-left text-xs font-medium text-gray-500 uppercase tracking-
wider">Date</th>
                    <th class="px-6 py-3 text-right text-xs font-medium text-gray-500 uppercase tracking-
wider">Actions</th>
                  </tr>
                </thead>
                <tbody class="bg-white divide-y divide-gray-200">
                  <tr v-for="post in posts.data" :key="post.id">
                    <td class="px-6 py-4 whitespace-nowrap">
                      <div class="text-sm font-medium text-gray-900">{{ post.title }}</div>
                      <div class="text-sm text-gray-500 truncate w-48">{{ post.content }}</div>
                    </td>
                    <td class="px-6 py-4 whitespace-nowrap text-sm text-gray-500">
                      {{ post.user.name }}
                    </td>
                    <td class="px-6 py-4 whitespace-nowrap text-sm text-gray-500">
                      {{ new Date(post.created_at).toLocaleDateString() }}
                    </td>
                    <td class="px-6 py-4 whitespace-nowrap text-right text-sm font-medium">
                      <button v-if="post.user_id === $page.props.auth.user.id"
                          @click="deletePost(post)"
                          class="text-red-600 hover:text-red-900 ml-4">
                        Delete
                      </button>
                      </td>
                  </tr>
                  <tr v-if="posts.data.length === 0">
                    <td colspan="4" class="px-6 py-4 text-center text-gray-500">
                      No posts found.
                    </td>
                  </tr>
                </tbody>
              </table>
            </div>

            <div class="mt-6">
              <Pagination :links="posts.links" />
            </div>

          </div>
  </AuthenticatedLayout>
</template>
```

Deployment Guide

In a production environment, your architecture changes slightly. You don't run npm run dev (the Vite development server). Instead, you build static assets, and Nginx serves them while passing API requests to PHP-FPM.

Server Prerequisites (LEMP Stack)

SSH into your server and run the following commands to install Nginx, MySQL, and PHP.

```
# 1. Update packages
sudo apt update && sudo apt upgrade -y

# 2. Install Nginx
sudo apt install nginx -y

# 3. Install MySQL
sudo apt install mysql-server -y
sudo mysql_secure_installation # Run this to secure your database

# 4. Install PHP 8.2 (or your version) and extensions
sudo apt install php8.2-fpm php8.2-mysql php8.2-mbstring php8.2-xml php8.2-bcmath php8.2-curl
unzip -y

# 5. Install Composer (PHP Dependency Manager)
curl -sS https://getcomposer.org/installer | php
sudo mv composer.phar /usr/local/bin/composer

# 6. Install Node.js & NPM (required to build Vue assets)
curl -fsSL https://deb.nodesource.com/setup_20.x | sudo -E bash -
sudo apt install -y nodejs
```

Application Setup

1. Clone your repository Navigate to the web root and clone your code.

```
cd /var/www
sudo git clone https://github.com/your-username/your-repo.git html
sudo chown -R www-data:www-data html          # Give Nginx ownership
cd html
```

2. Install Dependencies

```
# PHP Dependencies
composer install --optimize-autoloader --no-dev
```

```
# Javascript Dependencies
npm install
```

3. Configure Environment

```
cp .env.example .env
nano .env
```

- Set APP_ENV=production
- Set APP_DEBUG=false
- Enter your database credentials (DB_DATABASE, DB_USERNAME, DB_PASSWORD).

4. Generate Key & Migrate

```
php artisan key:generate
php artisan migrate --force                    # --force runs migrations in production
```

Building Assets

This is the most distinct step for Inertia apps compared to standard Laravel Blade apps. You must compile your Vue components into static JS/CSS files.

```
npm run build
```

This creates a public/build directory containing your compiled assets.

Nginx Configuration

Create a configuration file for your site.

```
sudo nano /etc/nginx/sites-available/my-app
```

Paste the following configuration. Crucially, note the try_files directive; this ensures that if a user refreshes the page on a route like /posts, Nginx doesn't look for a folder named "posts" but routes it back to Laravel index.php so Inertia can handle it.

```
server {
  listen 80;
  server_name your-domain.com; # OR your server IP
  root /var/www/html/public;

  add_header X-Frame-Options "SAMEORIGIN";
  add_header X-Content-Type-Options "nosniff";

  index index.php;

```

```
    charset utf-8;

    location / {
        try_files $uri $uri/ /index.php?$query_string;
    }

    location = /favicon.ico { access_log off; log_not_found off; }
    location = /robots.txt  { access_log off; log_not_found off; }

    error_page 404 /index.php;

    location ~ \.php$ {
        fastcgi_pass unix:/var/run/php/php8.2-fpm.sock; # Check your PHP version here
        fastcgi_param SCRIPT_FILENAME $realpath_root$fastcgi_script_name;
        include fastcgi_params;
    }

    location ~ /\.(?!well-known).* {
        deny all;
    }
}
```

Enable the site and restart Nginx:

```
sudo ln -s /etc/nginx/sites-available/my-app /etc/nginx/sites-enabled/
sudo nginx -t # Test config for errors
sudo systemctl restart nginx
```

Permissions Fix

Ensure the storage directory is writable by the server.

```
sudo chown -R www-data:www-data /var/www/html/storage
sudo chown -R www-data:www-data /var/www/html/bootstrap/cache
```

SSL (HTTPS)

Never run a production app on HTTP. Use Certbot to get a free certificate from Let's Encrypt.

```
sudo apt install certbot python3-certbot-nginx
sudo certbot --nginx -d your-domain.com
```

On demonstrating certbot without a domain

You cannot use Certbot (Let's Encrypt) for a bare IP address. Certbot validates that you own a specific domain name. It does not issue certificates for raw IP addresses (e.g., 192.168.1.1).

Workarounds:
Option 1: The "IT Admin" Way (Self-Signed Certificate)
This is the standard approach for internal servers or testing environments. You generate your own certificate locally.

- Pro: Fully encrypted traffic (HTTPS).
- Con: Browsers will show a big red "Not Secure" warning because they don't trust you as a Certificate Authority. You have to click "Advanced -> Proceed anyway."

1. Generate the Certificate Run this OpenSSL command. It creates a key and a certificate valid for 365 days.

```
sudo openssl req -x509 -nodes -days 365 -newkey rsa:2048 \
-keyout /etc/ssl/private/nginx-selfsigned.key \
-out /etc/ssl/certs/nginx-selfsigned.crt
```

(You can hit Enter through the questions it asks, or fill them in.)

2. Configure Nginx to use it Edit your site config:

```
sudo nano /etc/nginx/sites-available/my-app
```

Update the server block to listen on 443 and point to your new files:

```
server {
  listen 443 ssl;
  server_name _; # Accepts any IP/Name

  ssl_certificate /etc/ssl/certs/nginx-selfsigned.crt;
  ssl_certificate_key /etc/ssl/private/nginx-selfsigned.key;

  root /var/www/html/public;
  index index.php;

  # ... rest of your config (location blocks, etc) ...
}
```

```
# Optional: Redirect HTTP to HTTPS
server {
  listen 80;
  server_name _;
  return 301 https://$host$request_uri;
}
```

3. Restart Nginx

```
sudo systemctl restart nginx
```

Option 2: The "Magic DNS" Way (Using sslip.io)
If you really want to use Certbot and get a "Green Lock" without buying a domain, you can use a wildcard DNS service like sslip.io.

This service maps any subdomain containing an IP address back to that IP.

Example: If your server IP is 203.0.113.5, the domain 203.0.113.5.sslip.io automatically resolves to 203.0.113.5.

1. Update Nginx Config Open your config:

```
sudo nano /etc/nginx/sites-available/my-app
```

Change the server name to match the sslip format:

```
server_name 203.0.113.5.sslip.io; # REPLACE with your actual IP
```

2. Reload Nginx

```
sudo systemctl reload nginx
```

3. Run Certbot Now Certbot sees a "real" domain name and will issue the certificate.

```
sudo certbot --nginx -d 203.0.113.5.sslip.io
```

Recommended Project Architectural and Structure Setup

For a Laravel + Inertia + Vue application, the industry standard is often called the "Modern Monolith". Unlike a traditional API-driven architecture (where Laravel is just an API and Vue is a completely separate repo/server), Inertia allows you to keep everything in one repository. This architecture simplifies deployment, authentication, and state management significantly.
Here is the recommended project structure and architectural breakdown.

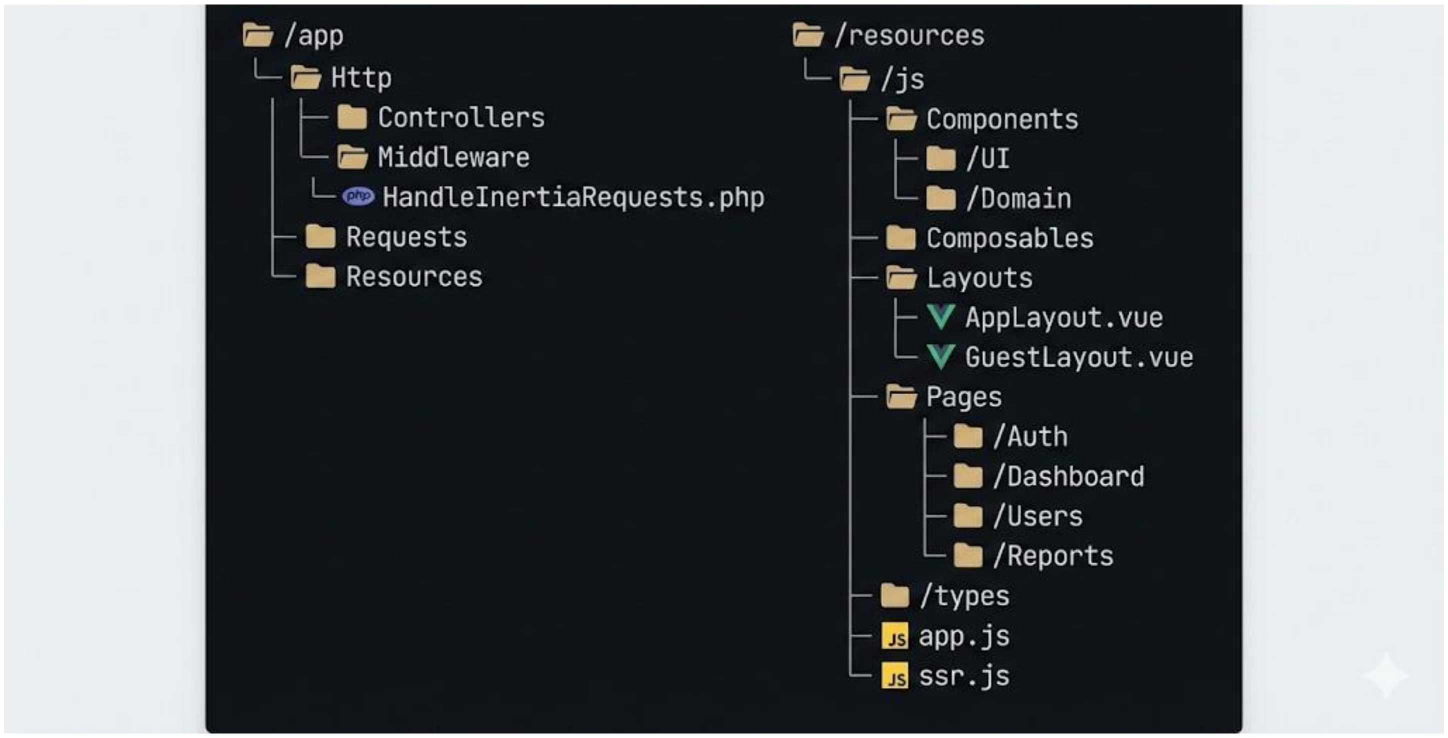
1. The High-Level Architecture

The core philosophy is: Laravel handles the routing and data; Vue handles the rendering.

- Routing: Defined in Laravel (routes/web.php). You do *not* use Vue Router.
- Controllers: Return Inertia::render() instead of view() or json().
- Authentication: Handled by Laravel Sessions (Sanctum/Passport are usually not needed unless you also have a mobile app).

2. Recommended Directory Structure

The most critical changes happen inside resources/js. Here is the robust structure used in enterprise Inertia projects:



```
/app
└─ Http
   ├─ Controllers
   └─ Middleware
      └─ HandleInertiaRequests.php
   ├─ Requests
   └─ Resources

/resources
└─ /js
   ├─ Components
   │  ├─ /UI
   │  └─ /Domain
   ├─ Composables
   ├─ Layouts
   │  ├─ AppLayout.vue
   │  └─ GuestLayout.vue
   ├─ Pages
   │  ├─ /Auth
   │  ├─ /Dashboard
   │  ├─ /Users
   │  └─ /Reports
   ├─ /types
   ├─ app.js
   └─ ssr.js
```

3. Key Architectural Patterns

A. The "Pages" Directory Strategy

Mirror your Controller structure inside resources/js/Pages. If you have a UserController, you should have a Pages/Users folder.

- Laravel: PostController@index
- Vue: resources/js/Pages/Posts/Index.vue

This makes navigating between backend logic and frontend views intuitive.

B. API Resources for Data Shaping

- ✓ Do not pass raw Eloquent models to Vue. This exposes database columns (like password or internal_flags) and makes the frontend payload heavy.
- ✓ Instead, use Laravel API Resources to define exactly what Vue receives.

Backend (UserResource.php):

```
public function toArray($request)
{
  return [
    'id' => $this->id,
    'name' => $this->name,
    'role' => $this->isAdmin() ? 'Administrator' : 'User',
    'edit_url' => route('users.edit', $this), // Pass URLs from backend!
    'can' => [
      'delete' => $request->user()->can('delete', $this),
    ]
  ];
}
```

Controller:

```
return Inertia::render('Users/Index', [
  'users' => UserResource::collection(User::paginate(10))
]);
```

C. Handling Global State (The "Shared" Prop)

In a traditional SPA, you might fetch the authenticated user in a mounted() hook. In Inertia, you pass global data via middleware so it is available instantly on every page load.

HandleInertiaRequests.php:

```
public function share(Request $request): array
{
  return array_merge(parent::share($request), [
    'auth' => [
      'user' => $request->user() ? new UserResource($request->user()) : null,
    ],
    'flash' => [
      'success' => $request->session()->get('success'),
      'error' => $request->session()->get('error'),
    ],
    'app_name' => config('app.name'),
  ]);
}
```

D. Component Organization (Atomic Design Lite)

Don't dump everything into one components folder.

1. Base (UI): Dumb components. They just look good. (Buttons, Inputs, Tables).
2. Partials: Complex chunks of a page (e.g., PostCommentList.vue).
3. Layouts: Wrappers that contain the Navbar/Sidebar.

Dockerizing the app and deploying to docker hub (Windows)

To Dockerize your Laravel + Inertia + Vue + Tailwind (LIVT) application and deploy it to Docker Hub from Windows, we will create a single "production-ready" image that includes your PHP backend and your compiled Vue/Tailwind assets.

Phase 1: Create Docker Configuration

We need to tell Docker how to build the app. We will use a Multi-Stage Build:

1. Node Stage: Compiles your Vue/Tailwind assets (npm run build).
2. PHP Stage: Installs Laravel dependencies and copies the compiled assets from the Node stage.

1. Create the Dockerfile
Create a file named Dockerfile (no extension) in your project root folder:

```
# --- Stage 1: Build Frontend Assets ---
FROM node:20 as frontend
WORKDIR /app
COPY package*.json ./
RUN npm install
COPY . .
RUN npm run build

# --- Stage 2: Build Backend & Serve ---
FROM php:8.2-fpm

# Install system dependencies (Nginx, Supervisor, libraries)
RUN apt-get update && apt-get install -y \
  git curl libpng-dev libonig-dev libxml2-dev zip unzip \
  nginx supervisor

# Clear cache
RUN apt-get clean && rm -rf /var/lib/apt/lists/*

# Install PHP extensions required by Laravel
RUN docker-php-ext-install pdo_mysql mbstring exif pcntl bcmath gd

# Get latest Composer
COPY --from=composer:latest /usr/bin/composer /usr/bin/composer

# Set working directory
WORKDIR /var/www

# Copy existing application directory
COPY . /var/www

# Copy compiled assets from Stage 1 (The "public/build" folder)
```

```
COPY --from=frontend /app/public/build /var/www/public/build
# Copy mix-manifest if you use Mix, otherwise ignore this line
COPY --from=frontend /app/public/mix-manifest.json /var/www/public/mix-manifest.json 2>/dev/null || :

# Install PHP dependencies
RUN composer install --optimize-autoloader --no-dev

# Set permissions for Laravel storage
RUN chown -R www-data:www-data /var/www/storage /var/www/bootstrap/cache

# Copy Configuration Files (We will create these next)
COPY docker/nginx.conf /etc/nginx/sites-available/default
COPY docker/supervisord.conf /etc/supervisor/conf.d/supervisord.conf

# Expose Port 80
EXPOSE 80

# Start Supervisor (Runs Nginx + PHP-FPM together)
CMD ["/usr/bin/supervisord"]
```

2. Create Configuration Files
Create a new folder named docker in your project root. Inside, create two files.

File A: docker/nginx.conf This tells Nginx to serve your public folder and pass PHP requests to the local PHP process.

```
server {
  listen 80;
  root /var/www/public;
  index index.php index.html;
  server_name localhost;
  error_log  /var/log/nginx/error.log;
  access_log /var/log/nginx/access.log;

  location / {
    try_files $uri $uri/ /index.php?$query_string;
  }

  location ~ \.php$ {
    try_files $uri =404;
    fastcgi_split_path_info ^(.+\.php)(/.+)$;
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_index index.php;
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    fastcgi_param PATH_INFO $fastcgi_path_info;
  }
}
```

File B: docker/supervisord.conf This tool allows us to run both Nginx and PHP-FPM in a single container (standard for simple deployments).

```
[supervisord]
nodaemon=true

[program:php-fpm]
command=docker-php-entrypoint php-fpm
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0

[program:nginx]
command=nginx -g "daemon off;"
stdout_logfile=/dev/stdout
stdout_logfile_maxbytes=0
stderr_logfile=/dev/stderr
stderr_logfile_maxbytes=0
```

Phase 2: Build and Tag the Image

1. Open PowerShell or CMD in your project root.
2. Log in to Docker Hub:

```
docker login
```

(Enter your Docker Hub username and password/token).

3. Build the Image: Replace yourusername with your actual Docker Hub username. We use --platform linux/amd64 to ensure it works on standard servers (even if you build on a different architecture).

```
docker build --platform linux/amd64 -t yourusername/livt-app:latest .
```

Phase 3: Push to Docker Hub

Once the build finishes successfully, push it to the cloud.

```
docker push yourusername/livt-app:latest
```

Phase 4: How to Deploy (Run it anywhere)

You don't need your code on the server anymore. You just need a docker-compose.yml file to pull your image and connect it to a database.

Create this file on your server (or locally to test):

```
version: '3.8'
services:
  app:
    image: yourusername/livt-app:latest # pulls your image
    ports:
      - "80:80"
    environment:
      APP_ENV: production
      APP_DEBUG: 'false'
      APP_KEY: "base64:..." # Put your Laravel Key here
      DB_CONNECTION: mysql
      DB_HOST: db
      DB_PORT: 3306
      DB_DATABASE: laravel
      DB_USERNAME: sail
      DB_PASSWORD: password
    depends_on:
      - db

  db:
    image: mysql:8.0
    environment:
      MYSQL_DATABASE: laravel
      MYSQL_USER: sail
      MYSQL_PASSWORD: password
      MYSQL_ROOT_PASSWORD: password
    volumes:
      - db-data:/var/lib/mysql

volumes:
  db-data:
```

To start the app:

```
docker-compose up -d
```

To run migrations:

```
docker-compose exec app php artisan migrate --force
```