White Paper

The Art of Front-End Performance on WeWeb

Introduction

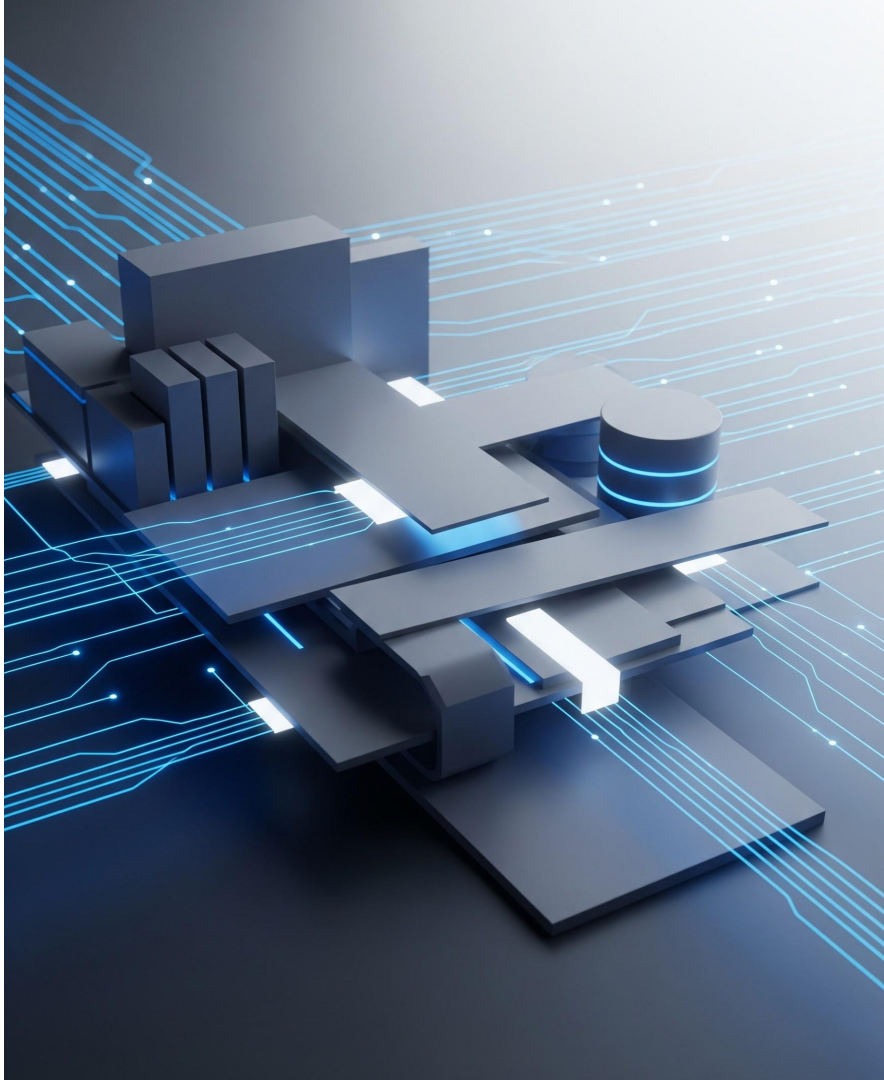
No-code tools like **WeWeb.io** are changing how applications are built.

But that ease of use can hide a major technical challenge: **the illusion that every page is naturally lightweight.**

Because WeWeb apps are based on **Vue.js**, every request, asset, and interface element still has an impact on the browser. This guide shares best practices to help you achieve:

- Faster loading times
- Smoother interactions
- Better SEO performance through Core Web Vitals

METHODOLOGY GUIDE



Chapter 1: Data Architecture

The Core Performance Lever

In more than 80% of cases, a slow WeWeb app comes from poor request management and inefficient data synchronization.

Fetch on Page Load (Enabled by default)

The app waits for the request to return data before displaying the page.

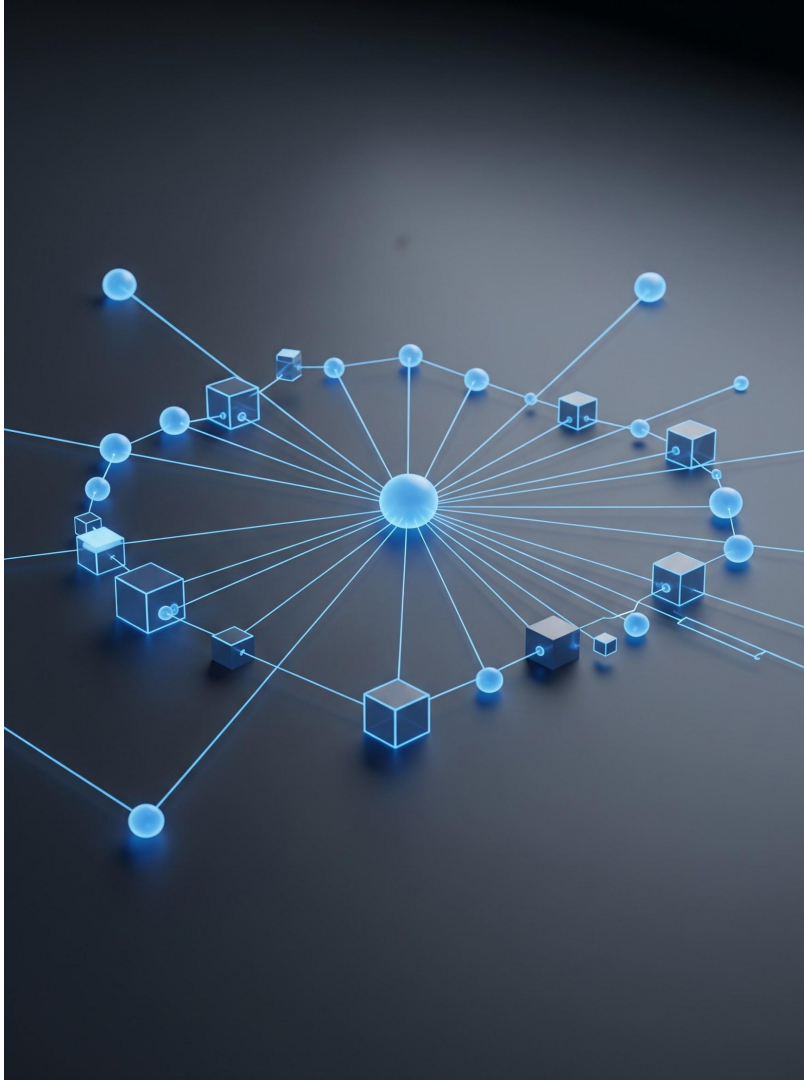
Best practice: Use this only for essential above-the-fold data.

Fetch on Demand (Triggered by a workflow)

Data is fetched only when the user takes an action, such as clicking or scrolling.

Best practice: Use this for modals, hidden sections, admin panels, and data that does not need to appear immediately.

DATA STRATEGY



Chapter 1: Data Architecture

2. The Front-End Filtering Trap

A common mistake is loading an entire database table into WeWeb, then using visual filters to display only part of it.

This overloads the browser's memory and makes the front end do work that should happen on the back end.

The Golden Rule: Delegate to the Back End

Filtering, sorting, and pagination should be handled by the back end, such as Xano, Supabase, or another API, **using dynamic query parameters.**

Limit the Response to What You Actually Need

If you are displaying a list of users, do not load their full transaction history in the same request.

Keep API responses focused and lightweight.

DATA STRATEGY

Chapter 1: Data Architecture

3. Calculated Formulas, Bindings, and Infinite Loops

WeWeb lets you write JavaScript formulas or visual formulas directly in bindings to connect data to interface elements.

The Problem: Interface Freezes

Even though Vue.js caches formulas, complex calculations inside bindings can still freeze the interface whenever the source data changes.

The Solution: Precalculate in a Workflow

Avoid complex filters, .map(), or .reduce() calls directly inside the binding panel.

Instead, calculate the result earlier in a workflow, store it in a simple variable, and bind your component to that variable.

DATA STRATEGY

Chapter 2: Render Optimization

The DOM and Performance

The browser has to analyze, position, and paint every HTML element on the page. If the DOM is too heavy or too deeply nested, interactions can become noticeably slower.

The Risk: Too Many Divs and Excessive Nesting

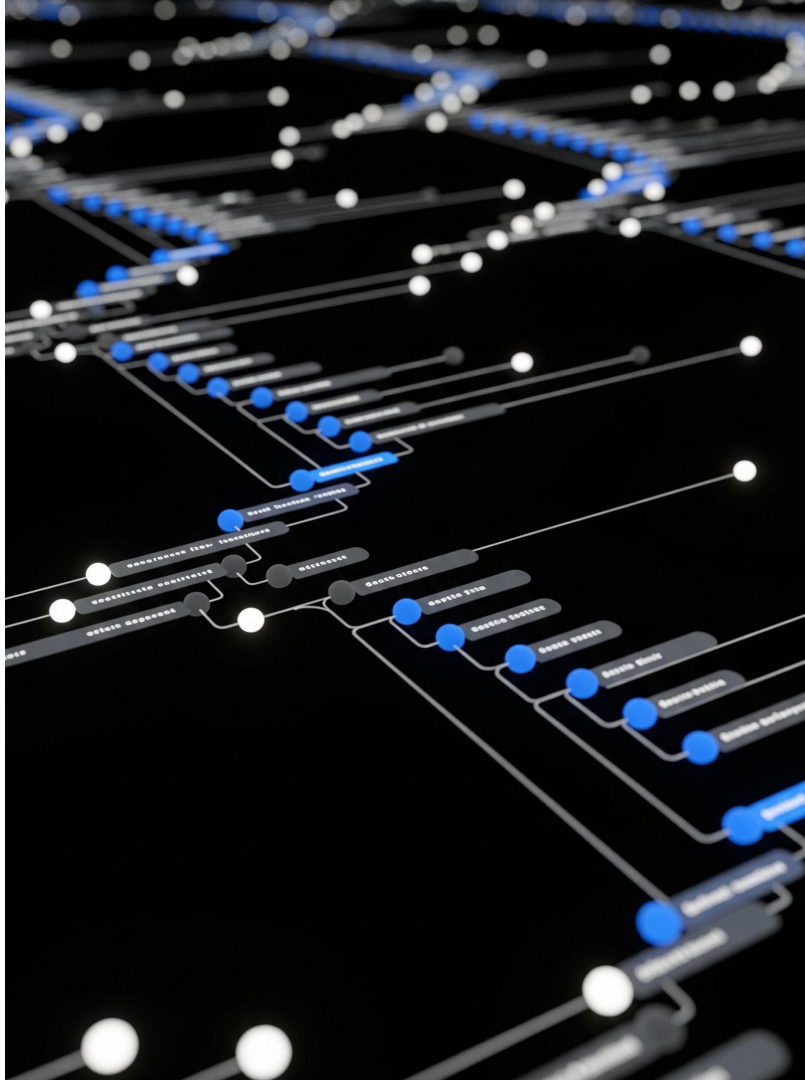
Adding unnecessary containers increases the cost of layout and paint calculations. A DOM depth above 15 levels is a warning sign for interface smoothness.

The Solution: Keep the Structure Semantic and Flat

Use WeWeb's native components and avoid nesting groups unless there is a real functional need.

Goal: Keep the page tree lightweight so the browser can recalculate layouts quickly.

DOM STRUCTURE



Chapter 2: Render Optimization

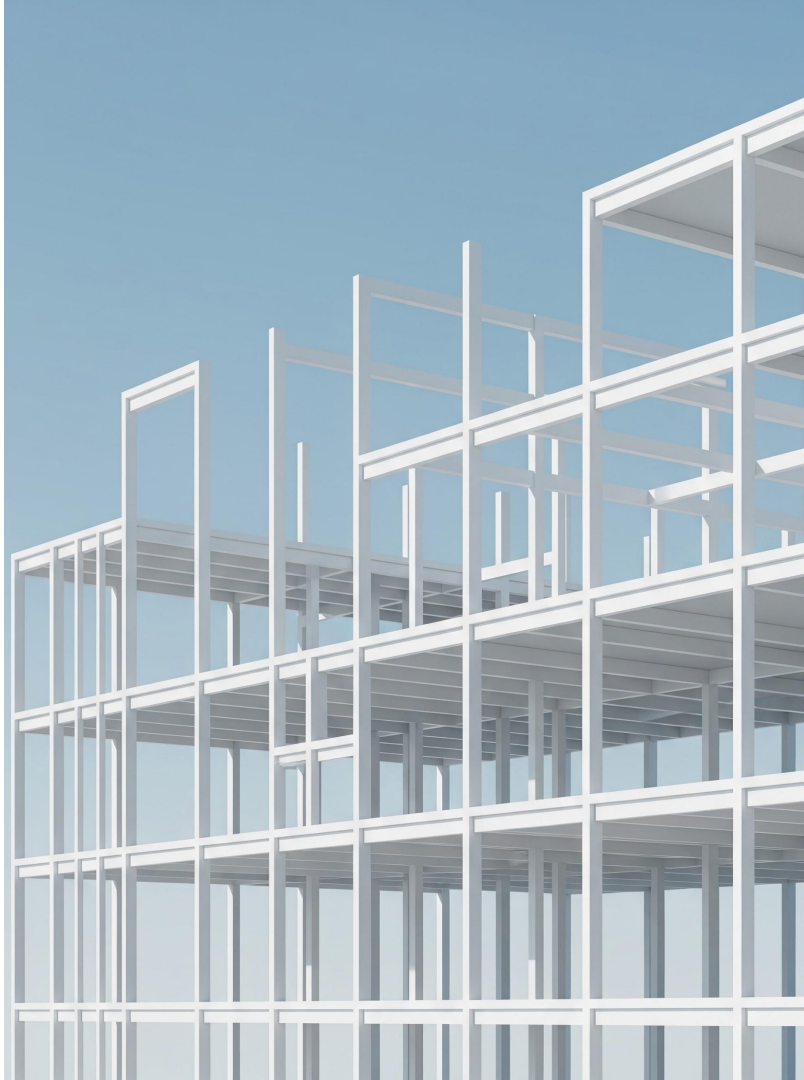
1. Remove Excessive Nesting

WeWeb's visual editor makes it easy to nest containers. Over time, a simple component can end up wrapped in several unnecessary divs.

Action Plan: Clean and Simplify

- Use WeWeb's structure panel, or tree view, to identify redundant layers.
- Simplify the hierarchy by using margin and padding on existing elements.
- Avoid adding empty containers only to create visual spacing.

DOM STRUCTURE



Chapter 2: Render Optimization

2. Conditional Display: "Render Only If" vs "Display"

WeWeb offers two main ways to hide or show an element in the interface.

Render Only If (v-if)

The element is added to the HTML only if the condition is true. If false, the browser does not render it at all.

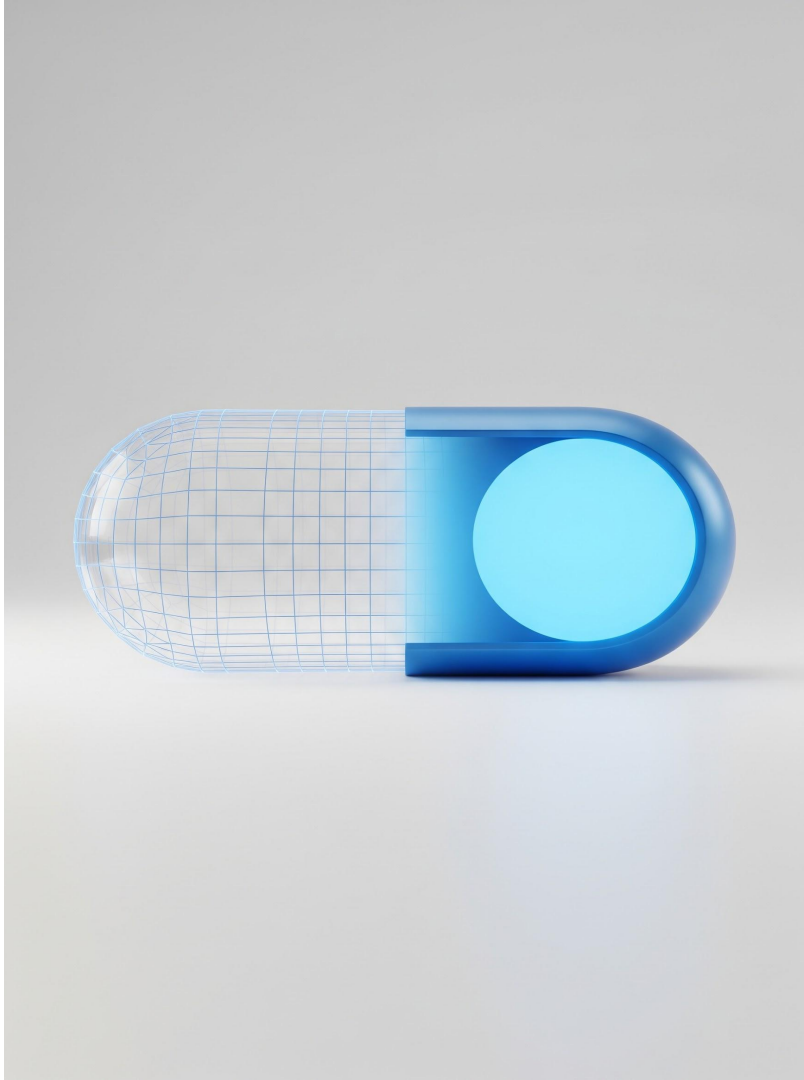
- Recommended for heavy elements (tables, charts, modals) to reduce initial load.

Display or Visibility (v-show)

The element remains in the DOM but is hidden via CSS properties.

- Recommended for lightweight elements that need instant toggling (e.g., dropdowns).

DOM STRUCTURE



Chapter 2: Render Optimization

3. Auto-Sizing, Dimensions, and CLS

Leaving containers in automatic size mode is one of the main causes of visual instability during page load.

The Problem: Layout Shift

Without defined dimensions, the browser recalculates the layout as images or data load.

This can create visible movement on the page and hurt SEO performance through Cumulative Layout Shift.

Solutions in WeWeb

- Reserve space: Set a min-height, for example 200px, on lists or variable content blocks to prevent layout jumps.
- Skeleton loaders: Use WeWeb's native skeleton loader component to mimic the content structure while data is loading.
- Define explicit ratios: Use aspect-ratio, such as 16/9 or 4/3, for images instead of relying on automatic sizing.

DOM STRUCTURE

Chapter 2: Render Optimization

4. Dynamic Lists and Pagination

Displaying too many complex elements at the same time can reduce animation quality and make scrolling feel less smooth.

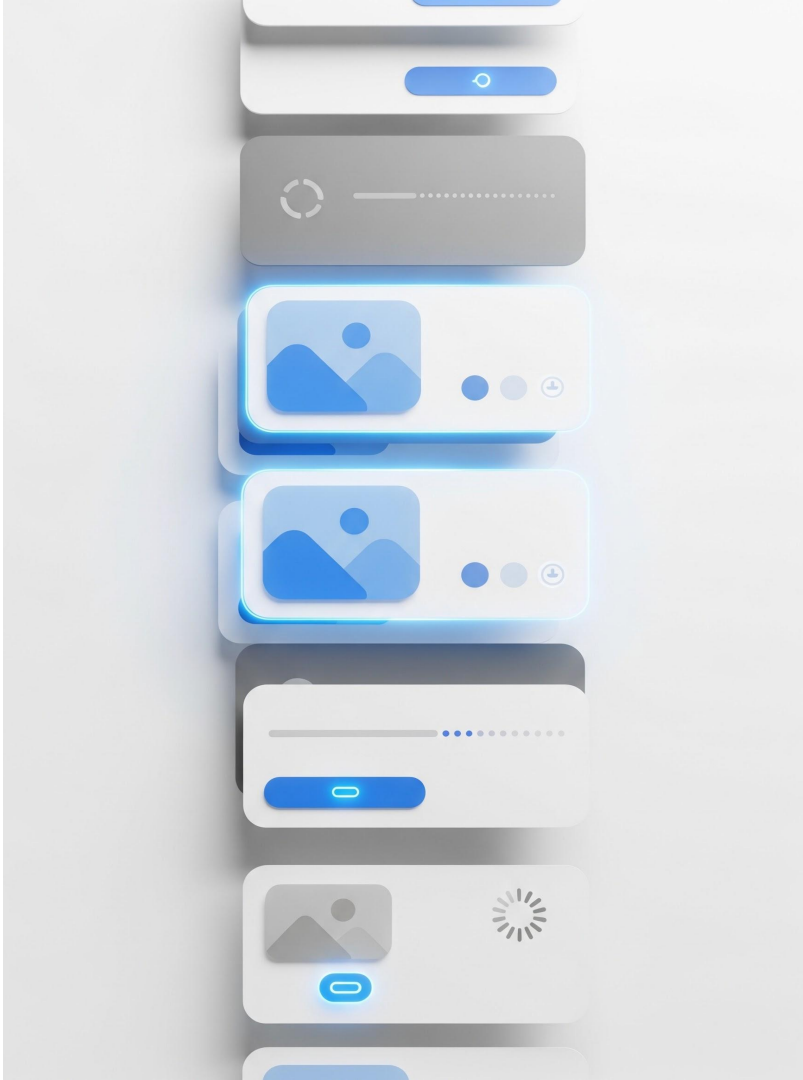
Solution 1: Classic Pagination

Use back-end controlled pagination so the app loads only the data needed for the current view.

Solution 2: Infinite Scroll or "Load More"

Use a controlled loading system and limit the initial render to 10 or 20 items to protect performance.

DOM STRUCTURE



Chapter 3: Assets and Fonts

Payload Optimization: The Weight of Static Elements

Static elements often represent most of the total weight of a web page.

Example of a typical unoptimized page, total size > 5.6 MB

Uncompressed images	75%	4.2 MB
Google Fonts, 6 styles	14%	800 KB
JavaScript and tracking	9%	500 KB
CSS and HTML	2%	100 KB

ASSETS & FONTS

Chapter 3: Assets and Fonts

1. Image Optimization

The most common mistake is importing a raw PNG or JPEG image, often 2 to 5 MB, and resizing it visually in the editor. The browser still downloads the full original file.

Processing Rules

- **Format:** Convert images to WebP or AVIF to reduce file weight by 30% to 80% without noticeable quality loss.
- **Dimensions:** Resize images to the largest real display size before hosting them.
- **Compression:** Use tools like TinyPNG or Squoosh.
- **Lazy loading:** Enable lazy loading for images below the fold.

2. Strict Font Management

The myth of local imports: uploading a font to WeWeb does not mean the visitor avoids downloading it. The font still loads through an HTTP request from the CDN.

Comparing Font Methods

1. Google Fonts: External requests, slower DNS resolution, and potential GDPR considerations.

2. Custom fonts via CDN: No external font domain, faster and easier to control.

3. System fonts: Zero font requests. Best for performance (e.g., Arial or UI fonts).

Performance Best Practices

- **Limit families:** Use two families max (one for headings, one for body).
- **Limit weights:** Only use weights you need, usually 400, 500, and 700.
- **Reduce FOIT:** Use system fonts where possible to avoid invisible text.

ASSETS & FONTS

Chapter 3: Assets and Fonts

3. Icons & SVGs

WeWeb offers many icon libraries (Font Awesome, Lucide, Phosphor). Avoid file inflation by managing assets efficiently.

Recommended Practice

Avoid mixing icon packs across the same project. Choose one library and use it consistently to minimize bundle size.

SVG Optimization

- Prefer inline SVGs for specific icons or brand visuals.
- Use raw HTML code for SVGs—it loads instantly and avoids HTTP requests.

Note: *WeWeb's native system handles icons well, but inline SVG is best for complex visuals.*

Chapter 4: Third-Party Scripts and Workflows

1. Control the Impact of Marketing Tools

The accumulation of third-party tools, such as GTM, HubSpot, Hotjar, or Crisp, is one of the main performance issues in mature web apps.

Loading Strategies

- **Asynchronous loading:** Configure scripts to load asynchronously using async or defer.
- **Delayed loading:** A major technique in WeWeb. Avoid initializing scripts during the initial page load.
- **Global workflow:** Trigger scripts through On App Load, then Delay, then Custom JS. This lets the user interact with the page before third-party scripts run.

SCRIPTS & WORKFLOWS

2. Design Workflows Carefully

WeWeb workflows translate logical action chains into the app experience. Poorly designed workflows can block the interface or create unnecessary load.

Optimization Principles

Avoid UI Locking:
When using a request or collection action, avoid blocking the UI with endless loaders unless it is essential for data integrity.

Debounce Inputs:
For filters triggered on change, apply a debounce (e.g., 300ms) to avoid firing an API request for every letter the user types.

SCRIPTS & WORKFLOWS

Conclusion: Express Audit

Pre-Production Checklist

01. Assets
Use WebP or AVIF. Keep images under 200 KB.

02. Ratios
Set aspect ratios to avoid layout shift.

03. Mobile
Use dvh and dvw units for full-screen sections.

04. Fonts
Remove unused weights from the project.

05. Stability
Use min-height or skeleton loaders for lists.

06. Data
Use server-side pagination (Xano, Supabase).

07. Rendering
Use "Render Only If" for modals and panels.

08. DOM
Remove unnecessary nested div elements.

09. Scripts
Delay third-party scripts by 5 seconds.

Apply these rules consistently to improve PageSpeed scores and create a smoother user experience.

AUDIT & PERFORMANCE

Chapter Closing

Thank you for reading!

This guide was presented by Raphael Scotto di Perta, with review and recommendations from the WeWeb team.

END OF MODULE