Performance

Overview

Vue is designed to be performant for most common use cases without much need for manual optimizations. However, there are always challenging scenarios where extra fine-tuning is needed. In this section, we will discuss what you should pay attention to when it comes to performance in a Vue application.

First, let's discuss the two major aspects of web performance:

- **Page Load Performance**: how fast the application shows content and becomes interactive on the initial visit. This is usually measured using web vital metrics like Largest Contentful Paint (LCP) and Interaction to Next Paint.
- **Update Performance**: how fast the application updates in response to user input. For example, how fast a list updates when the user types in a search box, or how fast the page switches when the user clicks a navigation link in a Single-Page Application (SPA).

While it would be ideal to maximize both, different frontend architectures tend to affect how easy it is to attain desired performance in these aspects. In addition, the type of application you are building greatly influences what you should prioritize in terms of performance. Therefore, the first step of ensuring optimal performance is picking the right architecture for the type of application you are building:

- Consult Ways of Using Vue to see how you can leverage Vue in different ways.
- Jason Miller discusses the types of web applications and their respective ideal implementation / delivery in Application Holotypes.

Profiling Options

For profiling load performance of production deployments:

- PageSpeed Insights
- WebPageTest

For profiling performance during local development:

- Chrome DevTools Performance Panel
  - `app.config.performance` enables Vue-specific performance markers in Chrome DevTools' performance timeline.
- Vue DevTools Extension also provides a performance profiling feature.

Page Load Optimizations

There are many framework-agnostic aspects for optimizing page load performance - check out this web.dev guide for a comprehensive round up. Here, we will primarily focus on techniques that are specific to Vue.

Choosing the Right Architecture

If your use case is sensitive to page load performance, avoid shipping it as a pure client-side SPA. You want your server to be directly sending HTML containing the content the users want to see. Pure client-side rendering suffers from slow time-to-content. This can be mitigated with Server-Side Rendering (SSR) or Static Site Generation (SSG). Check out the SSR Guide to learn about performing SSR with Vue. If your app doesn't have rich interactivity requirements, you can also use a traditional backend server to render the HTML and enhance it with Vue on the client.

If your main application has to be an SPA, but has marketing pages (landing, about, blog), ship them separately! Your marketing pages should ideally be deployed as static HTML with minimal JS, by using SSG.

Bundle Size and Tree-shaking

One of the most effective ways to improve page load performance is shipping smaller JavaScript bundles. Here are a few ways to reduce bundle size when using Vue:

- Use a build step if possible.

in the final production bundle. Tree-shaking can also remove other unused modules in your source code.

  - When using a build step, templates are pre-compiled so we don't need to ship the Vue compiler to the browser. This saves **14kb** min+gzipped JavaScript and avoids the runtime compilation cost.

- Be cautious of size when introducing new dependencies! In real-world applications, bloated bundles are most often a result of introducing heavy dependencies without realizing it.

  - If using a build step, prefer dependencies that offer ES module formats and are tree-shaking friendly. For example, prefer `lodash-es` over `lodash`.

  - Check a dependency's size and evaluate whether it is worth the functionality it provides. Note if the dependency is tree-shaking friendly, the actual size increase will depend on the APIs you actually import from it. Tools like bundlejs.com can be used for quick checks, but measuring with your actual build setup will always be the most accurate.

- If you are using Vue primarily for progressive enhancement and prefer to avoid a build step, consider using petite-vue (only **6kb**) instead.

Code Splitting

Code splitting is where a build tool splits the application bundle into multiple smaller chunks, which can then be loaded on demand or in parallel. With proper code splitting, features required at page load can be downloaded immediately, with additional chunks being lazy loaded only when needed, thus improving performance.

Bundlers like Rollup (which Vite is based upon) or webpack can automatically create split chunks by detecting the ESM dynamic import syntax:

```js
// lazy.js and its dependencies will be split into a separate chunk
// and only loaded when `loadLazy()` is called.
function loadLazy() {
  return import('./lazy.js')
}
```

Lazy loading is best used on features that are not immediately needed after initial page load. In Vue applications, this can be used in combination with Vue's Async Component feature to create split chunks for component trees:

```js
import { defineAsyncComponent } from 'vue'
```

```
// rendered on the page.
const Foo = defineAsyncComponent(() => import('./Foo.vue'))
```

For applications using Vue Router, it is strongly recommended to use lazy loading for route components. Vue Router has explicit support for lazy loading, separate from `defineAsyncComponent`. See Lazy Loading Routes for more details.

Update Optimizations

Props Stability

In Vue, a child component only updates when at least one of its received props has changed. Consider the following example:

```template
<ListItem
  v-for="item in list"
  :id="item.id"
  :active-id="activeId" />
```

Inside the `<ListItem>` component, it uses its `id` and `activeId` props to determine whether it is the currently active item. While this works, the problem is that whenever `activeId` changes, **every** `<ListItem>` in the list has to update!

Ideally, only the items whose active status changed should update. We can achieve that by moving the active status computation into the parent, and make `<ListItem>` directly accept an `active` prop instead:

```template
<ListItem
  v-for="item in list"
  :id="item.id"
  :active="item.id === activeId" />
```

Now, for most components the `active` prop will remain the same when `activeId` changes, so they no longer need to update. In general, the idea is keeping the props passed to child components as stable as possible.

`v-once`

`v-once` is a built-in directive that can be used to render content that relies on runtime data but never needs to update. The entire sub-tree it is used on will be skipped for all future

`v-memo`

`v-memo` is a built-in directive that can be used to conditionally skip the update of large sub-trees or `v-for` lists. Consult its API reference for more details.

Computed Stability

In Vue 3.4 and above, a computed property will only trigger effects when its computed value has changed from the previous one. For example, the following `isEven` computed only triggers effects if the returned value has changed from `true` to `false`, or vice-versa:

```js
const count = ref(0)
const isEven = computed(() => count.value % 2 === 0)

watchEffect(() => console.log(isEven.value)) // true

// will not trigger new logs because the computed value stays `true`
count.value = 2
count.value = 4
```

This reduces unnecessary effect triggers, but unfortunately doesn't work if the computed creates a new object on each compute:

```js
const computedObj = computed(() => {
  return {
    isEven: count.value % 2 === 0
  }
})
```

Because a new object is created each time, the new value is technically always different from the old value. Even if the `isEven` property remains the same, Vue won't be able to know unless it performs a deep comparison of the old value and the new value. Such comparison could be expensive and likely not worth it.

Instead, we can optimize this by manually comparing the new value with the old value, and conditionally returning the old value if we know nothing has changed:

```js
const computedObj = computed((oldValue) => {
  const newValue = {
    isEven: count.value % 2 === 0
  }
  if (oldValue && oldValue.isEven === newValue.isEven) {
    return oldValue
  }
```

Try it in the playground

Note that you should always perform the full computation before comparing and returning the old value, so that the same dependencies can be collected on every run.

General Optimizations

> The following tips affect both page load and update performance.

Virtualize Large Lists

One of the most common performance issues in all frontend applications is rendering large lists. No matter how performant a framework is, rendering a list with thousands of items **will** be slow due to the sheer number of DOM nodes that the browser needs to handle.

However, we don't necessarily have to render all these nodes upfront. In most cases, the user's screen size can display only a small subset of our large list. We can greatly improve the performance with **list virtualization**, the technique of only rendering the items that are currently in or close to the viewport in a large list.

Implementing list virtualization isn't easy, luckily there are existing community libraries that you can directly use:

- vue-virtual-scroller
- vue-virtual-scroll-grid
- vueuc/VVirtualList

Reduce Reactivity Overhead for Large Immutable Structures

Vue's reactivity system is deep by default. While this makes state management intuitive, it does create a certain level of overhead when the data size is large, because every property access triggers proxy traps that perform dependency tracking. This typically becomes noticeable when dealing with large arrays of deeply nested objects, where a single render needs to access 100,000+ properties, so it should only affect very specific use cases.

Vue does provide an escape hatch to opt-out of deep reactivity by using `shallowRef()` and `shallowReactive()`. Shallow APIs create state that is reactive only at the root level, and exposes all nested objects untouched. This keeps nested property access fast, with the trade-off being that we must now treat all nested objects as immutable, and updates can only be triggered by replacing the root state:

```
])

// this won't trigger updates...
shallowArray.value.push(newObject)
// this does:
shallowArray.value = [...shallowArray.value, newObject]

// this won't trigger updates...
shallowArray.value[0].foo = 1
// this does:
shallowArray.value = [
  {
    ...shallowArray.value[0],
    foo: 1
  },
  ...shallowArray.value.slice(1)
]
```

Avoid Unnecessary Component Abstractions

Sometimes we may create renderless components or higher-order components (i.e. components that render other components with extra props) for better abstraction or code organization. While there is nothing wrong with this, do keep in mind that component instances are much more expensive than plain DOM nodes, and creating too many of them due to abstraction patterns will incur performance costs.

Note that reducing only a few instances won't have noticeable effect, so don't sweat it if the component is rendered only a few times in the app. The best scenario to consider this optimization is again in large lists. Imagine a list of 100 items where each item component contains many child components. Removing one unnecessary component abstraction here could result in a reduction of hundreds of component instances.

Edit this page on GitHub

‹ Previous
Production Deployment

Next ›
Accessibility