Async Components

Basic Usage

In large applications, we may need to divide the app into smaller chunks and only load a component from the server when it's needed. To make that possible, Vue has a `defineAsyncComponent` function:

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

const AsyncComp = defineAsyncComponent(() => {
  return new Promise((resolve, reject) => {
    // ...load component from server
    resolve(/* loaded component */)
  })
})
// ... use `AsyncComp` like a normal component
```

As you can see, `defineAsyncComponent` accepts a loader function that returns a Promise. The Promise's `resolve` callback should be called when you have retrieved your component definition from the server. You can also call `reject(reason)` to indicate the load has failed.

ES module dynamic import also returns a Promise, so most of the time we will use it in combination with `defineAsyncComponent`. Bundlers like Vite and webpack also support the syntax (and will use it as bundle split points), so we can use it to import Vue SFCs:

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

const AsyncComp = defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
)
```

The resulting `AsyncComp` is a wrapper component that only calls the loader function when it is actually rendered on the page. In addition, it will pass along any props and slots to the inner

As with normal components, async components can be registered globally using `app.component()`:

```js
app.component('MyComponent', defineAsyncComponent(() =>
  import('./components/MyComponent.vue')
))
```

They can also be defined directly inside their parent component:

```vue
<script setup>
import { defineAsyncComponent } from 'vue'

const AdminPage = defineAsyncComponent(() =>
  import('./components/AdminPageComponent.vue')
)
</script>

<template>
  <AdminPage />
</template>
```

Loading and Error States

Asynchronous operations inevitably involve loading and error states - `defineAsyncComponent()` supports handling these states via advanced options:

```js
const AsyncComp = defineAsyncComponent({
  // the loader function
  loader: () => import('./Foo.vue'),

  // A component to use while the async component is loading
  loadingComponent: LoadingComponent,
  // Delay before showing the loading component. Default: 200ms.
  delay: 200,

  // A component to use if the load fails
  errorComponent: ErrorComponent,
  // The error component will be displayed if a timeout is
  // provided and exceeded. Default: Infinity.
  timeout: 3000
})
```

because on fast networks, an instant loading state may get replaced too fast and end up looking like a flicker.

If an error component is provided, it will be displayed when the Promise returned by the loader function is rejected. You can also specify a timeout to show the error component when the request is taking too long.

Lazy Hydration 3.5+

> This section only applies if you are using Server-Side Rendering.

In Vue 3.5+, async components can control when they are hydrated by providing a hydration strategy.

- Vue provides a number of built-in hydration strategies. These built-in strategies need to be individually imported so they can be tree-shaken if not used.
- The design is intentionally low-level for flexibility. Compiler syntax sugar can potentially be built on top of this in the future either in core or in higher level solutions (e.g. Nuxt).

Hydrate on Idle

Hydrates via `requestIdleCallback`:

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

const AsyncComp = defineAsyncComponent({
  loader: () => import('./Comp.vue'),
  hydrate: hydrateOnIdle(/* optionally pass a max timeout */)
})
```

Hydrate on Visible

Hydrate when element(s) become visible via `IntersectionObserver`.

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

const AsyncComp = defineAsyncComponent({
  loader: () => import('./Comp.vue'),
  hydrate: hydrateOnVisible()
})
```

```js
hydrateOnVisible({ rootMargin: '100px' })
```

Hydrate on Media Query

Hydrates when the specified media query matches.

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

const AsyncComp = defineAsyncComponent({
  loader: () => import('./Comp.vue'),
  hydrate: hydrateOnMediaQuery('(max-width:500px)')
})
```

Hydrate on Interaction

Hydrates when specified event(s) are triggered on the component element(s). The event that triggered the hydration will also be replayed once hydration is complete.

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

const AsyncComp = defineAsyncComponent({
  loader: () => import('./Comp.vue'),
  hydrate: hydrateOnInteraction('click')
})
```

Can also be a list of multiple event types:

```js
hydrateOnInteraction(['wheel', 'mouseover'])
```

Custom Strategy

```ts
import { defineAsyncComponent, type HydrationStrategy } from 'vue'

const myStrategy: HydrationStrategy = (hydrate, forEachElement) => {
  // forEachElement is a helper to iterate through all the root elements
  // in the component's non-hydrated DOM, since the root can be a fragment
  // instead of a single element
  forEachElement(el => {
    // ...
  })
  // call `hydrate` when ready
  hydrate()
  return () => {
```

```
}

const AsyncComp = defineAsyncComponent({
  loader: () => import('./Comp.vue'),
  hydrate: myStrategy
})
```

Using with Suspense

Async components can be used with the `<Suspense>` built-in component. The interaction between `<Suspense>` and async components is documented in the dedicated chapter for `<Suspense>`.

Edit this page on GitHub

Previous
Provide / inject

Next
Composables