Suspense

> ⚠ **Experimental Feature**
>
> `<Suspense>` is an experimental feature. It is not guaranteed to reach stable status and the API may change before it does.

`<Suspense>` is a built-in component for orchestrating async dependencies in a component tree. It can render a loading state while waiting for multiple nested async dependencies down the component tree to be resolved.

Async Dependencies

To explain the problem `<Suspense>` is trying to solve and how it interacts with these async dependencies, let's imagine a component hierarchy like the following:

```
<Suspense>
└─ <Dashboard>
   ├─ <Profile>
   │  └─ <FriendStatus> (component with async setup())
   └─ <Content>
      ├─ <ActivityFeed> (async component)
      └─ <Stats> (async component)
```

In the component tree there are multiple nested components whose rendering depends on some async resource to be resolved first. Without `<Suspense>`, each of them will need to handle its own loading / error and loaded states. In the worst case scenario, we may see three loading spinners on the page, with content displayed at different times.

The `<Suspense>` component gives us the ability to display top-level loading / error states while we wait on these nested async dependencies to be resolved.

There are two types of async dependencies that `<Suspense>` can wait on:

2. Async Components.

`async setup()`

A Composition API component's `setup()` hook can be async:

```js
export default {
  async setup() {
    const res = await fetch(...)
    const posts = await res.json()
    return {
      posts
    }
  }
}
```

If using `<script setup>`, the presence of top-level `await` expressions automatically makes the component an async dependency:

```vue
<script setup>
const res = await fetch(...)
const posts = await res.json()
</script>

<template>
  {{ posts }}
</template>
```

Async Components

Async components are **"suspensible"** by default. This means that if it has a `<Suspense>` in the parent chain, it will be treated as an async dependency of that `<Suspense>`. In this case, the loading state will be controlled by the `<Suspense>`, and the component's own loading, error, delay and timeout options will be ignored.

The async component can opt-out of `Suspense` control and let the component always control its own loading state by specifying `suspensible: false` in its options.

Loading State

the node in the fallback slot will be shown instead.

```template
<Suspense>
  <!-- component with nested async dependencies -->
  <Dashboard />

  <!-- loading state via #fallback slot -->
  <template #fallback>
    Loading...
  </template>
</Suspense>
```

On initial render, `<Suspense>` will render its default slot content in memory. If any async dependencies are encountered during the process, it will enter a **pending** state. During the pending state, the fallback content will be displayed. When all encountered async dependencies have been resolved, `<Suspense>` enters a **resolved** state and the resolved default slot content is displayed.

If no async dependencies were encountered during the initial render, `<Suspense>` will directly go into a resolved state.

Once in a resolved state, `<Suspense>` will only revert to a pending state if the root node of the `#default` slot is replaced. New async dependencies nested deeper in the tree will **not** cause the `<Suspense>` to revert to a pending state.

When a revert happens, fallback content will not be immediately displayed. Instead, `<Suspense>` will display the previous `#default` content while waiting for the new content and its async dependencies to be resolved. This behavior can be configured with the `timeout` prop: `<Suspense>` will switch to fallback content if it takes longer than `timeout` milliseconds to render the new default content. A `timeout` value of `0` will cause the fallback content to be displayed immediately when default content is replaced.

Events

The `<Suspense>` component emits 3 events: `pending`, `resolve` and `fallback`. The `pending` event occurs when entering a pending state. The `resolve` event is emitted when new content has finished resolving in the `default` slot. The `fallback` event is fired when the contents of the `fallback` slot are shown.

The events could be used, for example, to show a loading indicator in front of the old DOM while new components are loading.

Error Handling

`<Suspense>` currently does not provide error handling via the component itself - however, you can use the `errorCaptured` option or the `onErrorCaptured()` hook to capture and handle async errors in the parent component of `<Suspense>`.

Combining with Other Components

It is common to want to use `<Suspense>` in combination with the `<Transition>` and `<KeepAlive>` components. The nesting order of these components is important to get them all working correctly.

In addition, these components are often used in conjunction with the `<RouterView>` component from Vue Router.

The following example shows how to nest these components so that they all behave as expected. For simpler combinations you can remove the components that you don't need:

```template
<RouterView v-slot="{ Component }">
  <template v-if="Component">
    <Transition mode="out-in">
      <KeepAlive>
        <Suspense>
          <!-- main content -->
          <component :is="Component"></component>

          <!-- loading state -->
          <template #fallback>
            Loading...
          </template>
        </Suspense>
      </KeepAlive>
    </Transition>
  </template>
</RouterView>
```

Vue Router has built-in support for lazily loading components using dynamic imports. These are distinct from async components and currently they will not trigger `<Suspense>`. However, they can still have async components as descendants and those can trigger `<Suspense>` in the usual way.

Nested Suspense

When we have multiple async components (common for nested or layout-based routes) like this:

```template
<Suspense>
  <component :is="DynamicAsyncOuter">
    <component :is="DynamicAsyncInner" />
  </component>
</Suspense>
```

`<Suspense>` creates a boundary that will resolve all the async components down the tree, as expected. However, when we change `DynamicAsyncOuter`, `<Suspense>` awaits it correctly, but when we change `DynamicAsyncInner`, the nested `DynamicAsyncInner` renders an empty node until it has been resolved (instead of the previous one or fallback slot).

In order to solve that, we could have a nested suspense to handle the patch for the nested component, like:

```template
<Suspense>
  <component :is="DynamicAsyncOuter">
    <Suspense suspensible> <!-- this -->
      <component :is="DynamicAsyncInner" />
    </Suspense>
  </component>
</Suspense>
```

If you don't set the `suspensible` prop, the inner `<Suspense>` will be treated like a sync component by the parent `<Suspense>`. That means that it has its own fallback slot and if both `Dynamic` components change at the same time, there might be empty nodes and multiple patching cycles while the child `<Suspense>` is loading its own dependency tree, which might not be desirable. When it's set, all the async dependency handling is given to the parent `<Suspense>` (including the events emitted) and the inner `<Suspense>` serves solely as another boundary for the dependency resolution and patching.

Related

- `<Suspense>` API reference

Edit this page on GitHub

Previous
Teleport

Next
Single-File Components