NOVEMBER 16, 2023 / #ASYNCHRONOUS

How to Use React Suspense to Improve your React Projects

Okosa Leonard

React's ecosystem is always growing, and it seems like there are always new features and enhancements to make development more effective and user-friendly.

React Suspense is one such feature that has become quite popular among React devs.

In this guide, you'll learn all about React Suspense and examine its features, use cases, and potential to transform your web applications.

What is React Suspense?

React suspense is a new feature released in React.js version 16.6. With this new feature, components may pause rendering while they wait for an asynchronous process to finish, like separating code or retrieving data.

Suspense was created to make it easier for developers to create apps that have improved loading indications and a more cohesive user experience. It makes it possible to halt component tree rendering until specific criteria

are satisfied, which makes it easier for developers to work with asynchronous data.

A Problem That Loading States Face in React

Managing loading states in React was a little trickier prior to React Suspense. Developers had to implement the loading mechanism using third-party frameworks like Redux or Mobx, conditional rendering, or state management. This frequently resulted in complicated, error-prone code.

This problem was solved by React Suspense, which offers a more sophisticated and integrated method of managing asynchronous actions and loading states.

Key Concepts and Features of React Suspense

Let's talk about some concepts and features to help you understand React Suspense and how it works.

1) Suspense Component

An essential element of React Suspense is the Suspense component. It lets you declare how to handle fallback content while asynchronous actions are pending and encapsulate any portion of your component tree.

```
<Suspense fallback={<LeoFallback />}>
  <LeoComponent />
</Suspense>
```

In this use case, if the `LeoComponent` is not ready, React will display the `LeoFallback` component instead.

2) Using `React.lazy()` or `lazy()`

React has a dynamic import mechanism called `lazy()` that lets you load components in a lazy manner.

Basically, lazy loading refers to the requirement that a component or portion of code will load only when it's needed. This functionality, which was first included in React 16.6, is frequently used in conjunction with React Suspense to enhance web application speed by loading components only when needed.

This is very helpful for minimizing your application's loading speed and lowering the initial bundle size.

Now we'll investigate `lazy()` in further depth and learn how it works.

Basic Syntax of `lazy()`

To use `lazy()`, you have to follow these steps:

First, import the `Suspense` components from React, as well as any components you intend to load lazily.

```
import { Suspense } from 'react';
```

Then use `lazy()` to define a dynamic import. For the component you wish

to load slowly, this function accepts an argument that is a function that produces a dynamic import statement.

```
const LeoComponent = lazy(() => import('./LeoComponent'));
```

In this use case, the `LeoComponent` will be loaded lazily when it's needed. The dynamic `import()` statement specifies the path to the component you want to import.

Next, enclose the element you wish to load lazily in a Suspense element. You may designate a backup component to show while the lazy-loaded component is being retrieved by using the Suspense component.

```
function App() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <LeoComponent />
      </Suspense>
    </div>
  );
}
```

Here, while the `LeoComponent` is being fetched, the fallback component will be displayed, indicating that the content is loading.

Advantages of `React.lazy()`

1. Enhanced speed: By selectively loading the components needed for

the current view and not loading all of the components at once, lazy loading of components can enhance the speed of the application.

2. Improved User Experience: You may improve the user experience by informing users that the application is actively loading material by utilizing Suspense to display a loading indication.
3. Code Splitting: One of `lazy()`'s main advantages is that it makes code splitting possible. The process of code splitting involves breaking up the code of your application into smaller, on-demand bundles. This minimizes the initial bundle size and quickens your application's loading time.

With `lazy()`, you can do code splitting and lazy loading of components in your React apps. This is a great feature. It is a useful tool for streamlining your web applications' efficiency and loading times, improving user experience by loading components only when needed.

It has restrictions and certain concerns, but when utilized properly, it may be a useful tool in your toolset for React development.

3) Error Boundaries

React components known as error boundaries have the ability to detect and manage faults inside their subtree. Because they can gently manage any problems that arise while waiting for asynchronous data, they are essential for dealing with suspense.

```
class ErrorBoundary extends React.Component {
  componentDidCatch(error, info) {
    // Handle the error
  }
```

```
  render() {
    return this.props.children;
  }
}
```

4) What is Concurrent Rendering?

React Suspense was introduced as part of Concurrent Mode, which is an experimental set of features in React. Later on, concurrent rendering was introduced.

Concurrent rendering enables the execution of numerous tasks at once, improving the responsiveness and efficiency of React apps. It is a component of Concurrent Mode, a trial-and-error collection of features designed to overcome some of the drawbacks of conventional React rendering.

Ensuring that the user interface stays fluid and responsive even when React is handling complicated rendering or other asynchronous operations is the core objective of concurrent rendering.

How Does React Suspense Work?

When using React Suspense with asynchronous operations, it follows these steps:

1. After React loads, a component tree is rendered.
2. React looks to see whether any of its child components are in a

   suspended state when it comes across a Suspense component.

3. React will display the given fallback UI until the data is ready, if a child component is awaiting data (for example, as a result of a `lazy()` import or a data fetch).
4. React smoothly transitions to rendering the real content once the data is available.

Because this procedure is automated, handling asynchronous actions without coding sophisticated logic is significantly easier for developers.

Use Cases for React Suspense

React Suspense is a flexible tool for your toolbox that may be used in a variety of scenarios, which include:

1. Data Fetching

React Suspense's data fetching feature makes managing asynchronous data loading in your React apps easier. React Suspense allows you to postpone rendering until the data is available, enhancing user experience by offering fallback content or loading indications.

I'll give an example using a fake API to show how to retrieve data using React Suspense.

Here's how to use React Suspense to manage data loading, assuming you're using a framework like React Query or Relay for data fetching:

Establish Error Boundary and React Suspense

To capture any failures during data retrieving, you must first set up an error

boundary and a React Suspense component. Here's how to make a custom `ErrorBoundary` component in react for data fetching:

```
import React from 'react';

class ErrorBoundary extends React.Component {
  producer(props) {
    unique(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      return <div>Error: Something is wrong!</div>;
    }

    return this.props.children;
  }
}

export default ErrorBoundary;
```

Use React Query to Fetch Data

If you're using React Query to retrieve data, you can make a component that uses `useQuery` to retrieve data and wrap it in `Suspense`:

```
import React from 'react';
import { useQuery } from 'react-query';
import ErrorBoundary from './ErrorBoundary';
```

```
// Define your data-fetching function
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  if (!response.ok) {
    throw new Error('Network response was not ok');
  }
  return response.json();
}

function DataFetching() {
  const { data, isLoading, isError } = useQuery('data', fetchData);

  if (isLoading) {
    throw new Promise((resolve) => setTimeout(resolve, 2000)); // Simul
  }

  if (isError) {
    throw new Error('Error while fetching data');
  }

  return (
    <div>
      <h1>Fetching data with React Suspense</h1>
      <p>{data}</p>
    </div>
  );
}

function App() {
  return (
    <div>
      <h1>Leo's App</h1>
      <ErrorBoundary>
        <React.Suspense fallback={<div>Loading...</div>}>
          <DataFetchingComponent />
        </React.Suspense>
      </ErrorBoundary>
    </div>
  );
}

export default App;
```

In this example, we fetch data using React Query's `useQuery` hook. To display the loading indicator, we `throw in new Promise` to mimic a loading delay if the data is still loading (`isLoading is true`). We throw an error so that the error boundary can catch it if there is a problem.

You can provide a backup component to display while loading by enclosing the `DataFetching` component in `Suspense`. It's a straightforward "Loading..." message in this instance.

You'll also want to ensure that the error handling, loading indications, and data fetching function are all customized according to your unique data-fetching needs.

This example shows how React Suspense makes state management and data fetching easier, leading to a more organized and understandable codebase.

These are examples of Suspense being used in a data fetching scenario in React.

2. Lazy Loading in React Suspense

Loading components of your application only when needed (lazy loading, also known as code splitting) can lower the size of your initial bundle and speed up the loading of your React application.

You can use `React.lazy()` in conjunction with React `Suspense` to easily include lazy loading into your application.

Here are the steps to implement lazy loading with React suspense:

First, import React Suspense from React:

```
import { Suspense } from 'react';
```

Next, create a lazy loaded component in your React app. To construct a component that loads slowly, use the `React.lazy()` method. Provide an arrow function to dynamically import the component.

```
const LennyComponent = lazy(() => import('./LennyComponent'));
```

Then wrap your component with a `Suspense`. Wrap Suspense around the sluggish part. While the lazy component is loading, you may designate a loading indication or fallback component to be shown.

```
function App() {
  return (
    <div>
      <h1>Leo's App</h1>
      <Suspense fallback={<div>Loading...</div>}>
        <LennyComponent />
      </Suspense>
    </div>
  );
}
```

The component can be used like any other component in your React application.

```
function LennyComponent() {
  return <div>This component is lazily loaded.</div>;
}
```

To finish building your application, you must use a development server and a tool like Webpack to build and serve your application in order to ensure that code splitting functions as intended. Your code will be automatically divided into pieces using Webpack for lazy loading.

This setup will minimize the initial bundle size and speed up the loading of your React application by only loading the `LazyComponent` when needed. While the component is being retrieved, users will see the loading indication (in this example, "Loading..."). The component loads and is rendered in the application with ease.

3. Better User Experiences

React Suspense can be employed to ensure a smooth user experience by showing loading indicators or fallback content during data loading. This reduces the perceived loading time for people who are using your React application.

For example, let's say you have a component that fetches data from an API using `fetch` and you want to display a loading spinner while the data is being fetched. Here's a basic example using React Suspense:

```
import React, { Suspense } from 'react';

// A component that fetches data
const fetchApiData = () => {
  return new Promise((resolve) => {
    setTimeout(() => {
      resolve('Data loaded!');
    }, 2000); // Simulating a 2-second delay for data fetching
  });
};

// A component that uses Suspense to handle asynchronous data fetching
const DataComponent = () => {
  const apidata = fetchApiData(); // This can be any async function, li

  return <div>{apiData}</div>;
};

// Wrapping the component with Suspense
const App = () => {
  return (
    <Suspense fallback={<LoadingSpinner />}>
      <DataComponent />
    </Suspense>
  );
};

// A simple loading spinner component
const LoadingSpinner = () => {
  return <div>Loading...</div>;
};
```

In this instance:

The `fetchApiData` method is used by the `DataComponent` component to get data. Keep in mind that while `fetchApiData` is really a straightforward

function that returns a promise, it may actually be an API call in practical

applications.

The Suspense component, which requires a fallback argument, encapsulates the App component. One component that will be displayed while the asynchronous operation is running is the fallback prop. It's the `LoadingSpinner` component in this instance.

React Suspense will take care of the asynchronous data fetching automatically when the `DataComponent` is rendered. The `LoadingSpinner` component will render if the data is not yet accessible. After the data is retrieved, the user interface is updated.

This method eliminates the need for manual loading state management, making for a more seamless user experience. This code is straightforward and React Suspense works efficiently here.

Conclusion

React Suspense is a major addition to the React ecosystem. It provides a simpler, integrated method of managing asynchronous actions and loading states. You can use it to design apps that offer quicker loading times, more efficient data fetching, and better user experiences by utilizing `Suspense`, `React.lazy()`, and error boundaries.

It's critical to comprehend the capabilities, restrictions, and best practices of any strong instrument. Concurrent Mode and React Suspense have the power to revolutionize web application development and improve user experience even more. But as the React ecosystem continues to grow, it's imperative to keep up with the most recent advancements and industry best practices.

Okosa Leonard

A frontend developer that's ready to explore. To shake the tree of life is to bring down a fruit unheard of. It's always nice to learn, Learning is something that we must never stop.

If you read this far, thank the author to show them you care.

Say Thanks

You can make a tax-deductible donation here.

Learn Swift

Learn Golang

Learn Node.js

Learn CSS Grid

Learn Solidity

Learn Express.js

Learn JS Modules

Learn Apache Kafka

REST API Best Practices

Front-End JS Development

Learn to Build REST APIs

Intermediate TS and React

Command Line for Beginners

Intro to Operating Systems

Learn to Build GraphQL APIs

OSS Security Best Practices

Distributed Systems Patterns

Software Architecture Patterns

Mobile App

Our Charity