What Are State Management Libraries, and When Should You Use Them?

As your app grows, managing how data flows between components can become complex.

When starting out, React's `useState` hook might be sufficient, but as you add features, you might encounter issues with:

- Passing props through components that don't need them, also known as prop drilling
- Keeping data in sync across different parts of your app
- Handling complex updates that affect multiple components simultaneously

These and other challenges may arise, which can lead to a codebase that's harder to maintain, debug, and test. That's where state management libraries come in – they provide a centralized place where components can get or update the data they need.

Let's take a look at a few different state management options you have, and when to use them.

The Context API is a state manager built into React that lets you share state across components without using a third-party library. It's a well-established upgrade over the `useState` hook, so it is perfect for cases like theme toggling or user authentication status.

However, the Context API does not handle frequent updates well, and can cause unnecessary re-renders, making it less suitable for complex state needs in applications like eCommerce and social media platforms.

Here's a counter component that demonstrates the basic usage of the Context API:

```
import { useState, createContext } from 'react';

const CounterContext = createContext();

const CounterProvider = ({ children }) => {
  const [count, setCount] = useState(0);

  return (
    <CounterContext.Provider value={{ count, setCount }}>
      {children}
    </CounterContext.Provider>
  );
};

export { CounterContext, CounterProvider };
```

This code creates a context and a provider to share a `count` state across the application.

`CounterProvider` uses the `useState` hook to initialize and manage the `count` state and its setter. Both are then passed into child components through the `Provider`.

So, when you wrap your whole app with the `CounterProvider`, the `count` state is available everywhere in your application.

Here's how you can wrap `CounterProvider` around your application:

```
import { CounterProvider } from './context/CounterContext';

function App() {
  return (
    <CounterProvider>
        {/* App components */}
    </CounterProvider>
  );
}

export default App;
```

And here's how you can use the `count` state:

```
import React, { useContext } from 'react';
import { CounterContext } from '../context/CounterContext';

const Counter = () => {
  const { count, setCount } = useContext(CounterContext);

  return (
    <>
      <div style={{ textAlign: 'center' }}>
        <h1>Context API Counter</h1>
        <button style={{ marginRight: '5px' }} onClick={() => setCount(count - 1)}>
          Decrease
        </button>
        <span>{count}</span>
        <button style={{ marginLeft: '5px' }} onClick={() => setCount(count + 1)}>
          Increase
        </button>
      </div>
    </>
  );
};

export default Counter;
```

As you can see, the `count` and its setter function, `setCount`, are initialized through the `useContext` function.

The current `count` state is then displayed, and `setCount` is used to increase and decrease the `count` state when the user clicks the decrement and increment buttons respectively.

Another popular state management library is Redux, which is one of the most popular state management libraries to use with React. It's been around for a long time, and is ideal for larger applications like eCommerce and social media platforms, forums, and so on.

Redux handles state management by providing a central store and strict control over state updates. It uses a predictable pattern with actions, reducers, and middleware.

Actions are payloads of information that send data from your application to the Redux store, often triggered by user interactions.

Reducers are functions that specify how the state should change in response to those actions, ensuring the state is updated in an immutable way.

Middleware, on the other hand, acts as a bridge between the action dispatching and the reducer, allowing you to extend Redux's functionality (for example, logging, handling async operations) without modifying the core flow.

The most common complaint about Redux is with all the boilerplate code you need to get started. In response, the Redux team introduced "Redux Toolkit" and "RTK Query", which simplify the setup process quite a bit.

You typically define both actions and reducers in a single file using the `createSlice()` function. It's common to name the file so it ends with the word `Slice`, for example, `productSlice`, `userSlice`, `counterSlice`, and so on.

Here's a `counterSlice` file to show you the basics:

```
import { createSlice } from '@reduxjs/toolkit';

const counterSlice = createSlice({
  name: 'counter',

  initialState: { count: 0 },

  reducers: {
    increment: (state) => {
      state.count += 1;
    },
    decrement: (state) => {
      state.count -= 1;
    },
  },
});

export const { increment, decrement } = counterSlice.actions;

export default counterSlice.reducer;
```

From here, you then need to wrap the entire app with the `Provider`, select a piece of state from the slice with `useSelector()`, then use `useDispatch()` to make the state active.

Another option to consider is Zustand.

Zustand is a lightweight state management library with a simple API. It is based on hooks, so there's less boilerplate compared to Redux, making it easier and quicker to set up.

Zustand is ideal for small to medium-scale applications. It works by using a `useStore` hook to access state directly in components and pages. This lets you modify and access data without needing actions, reducers, or a provider.

Here's a `useCounterStore` that implements another counter functionality:

```
import { create } from 'zustand';

const useCounterStore = create((set) => ({
  count: 0,
  increment: () => set((state) => ({ count: state.count + 1 })),
  decrement: () => set((state) => ({ count: state.count - 1 })),
}));

export default useCounterStore;
```

And here's how to initialize and use the states in your app:

```
// Import the useCounterStore (it's just a hook)
import useCounterStore from '../useCounterStore';

const Counter = () => {
  // Initialize the states with the useCounterStore hook
  const { count, increment, decrement } = useCounterStore();

  return (
    <>
      <div style={{ textAlign: 'center' }}>
        <h1>Zustand Counter</h1>
        <button style={{ marginRight: '5px' }} onClick={() => decrement()}>
          Decrease
        </button>
        <span>{count}</span>
        <button style={{ marginLeft: '5px' }} onClick={() => increment()}>
          Increase
        </button>
      </div>
    </>
  );
};

export default Counter;
```

Even though the frontend ecosystem is constantly evolving and new state management libraries regularly emerge, the ones we've discussed are widely used in the industry.

Questions

Which of these is a reason to use a state management library?

To avoid writing any state logic in your application.

To automatically update the UI without re-rendering.

To create a consistent and predictable data flow.

To eliminate the need for props entirely.

Which of these is NOT a state management library?

Context API

Redux

Zustand

Axios

What was a common complaint about Redux, and how was it addressed?

It had limited browser support, which was addressed by creating polyfills.

It had performance issues, which were addressed by optimizing its middleware.

It required a lot of complex boilerplate code, which was addressed by Redux Toolkit and RTK Query.

There was a lack of documentation, which was addressed by adding more examples.