React Context API

React Context API

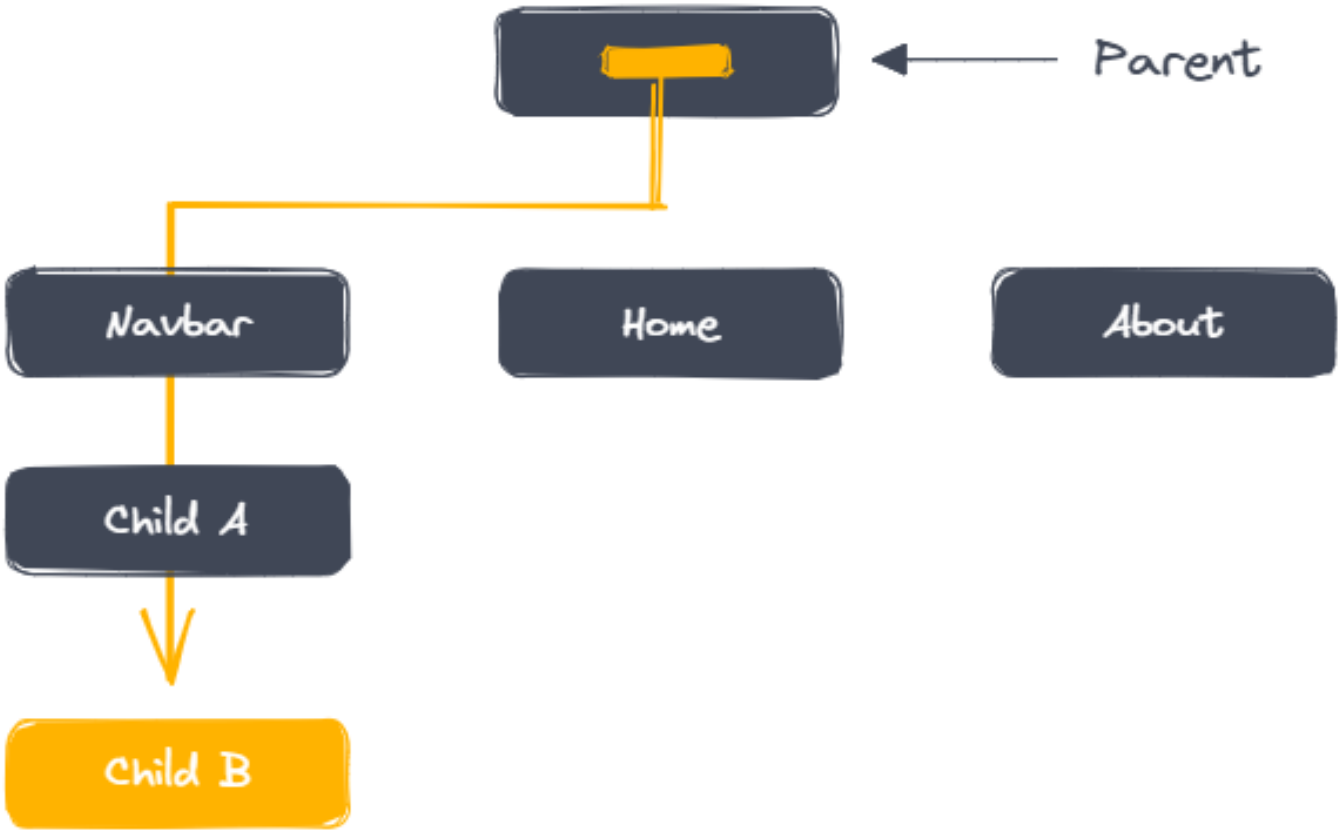
Managing state is an essential part of developing applications in React. A common way to manage state is by passing props. Passing props means sending data from one component to another. It's a good way to make sure that data gets to the right place in a React application.

But it can be annoying to pass props when you have to send the same data to lots of components or when components are far away from each other. This can make an application slower and harder to work with.

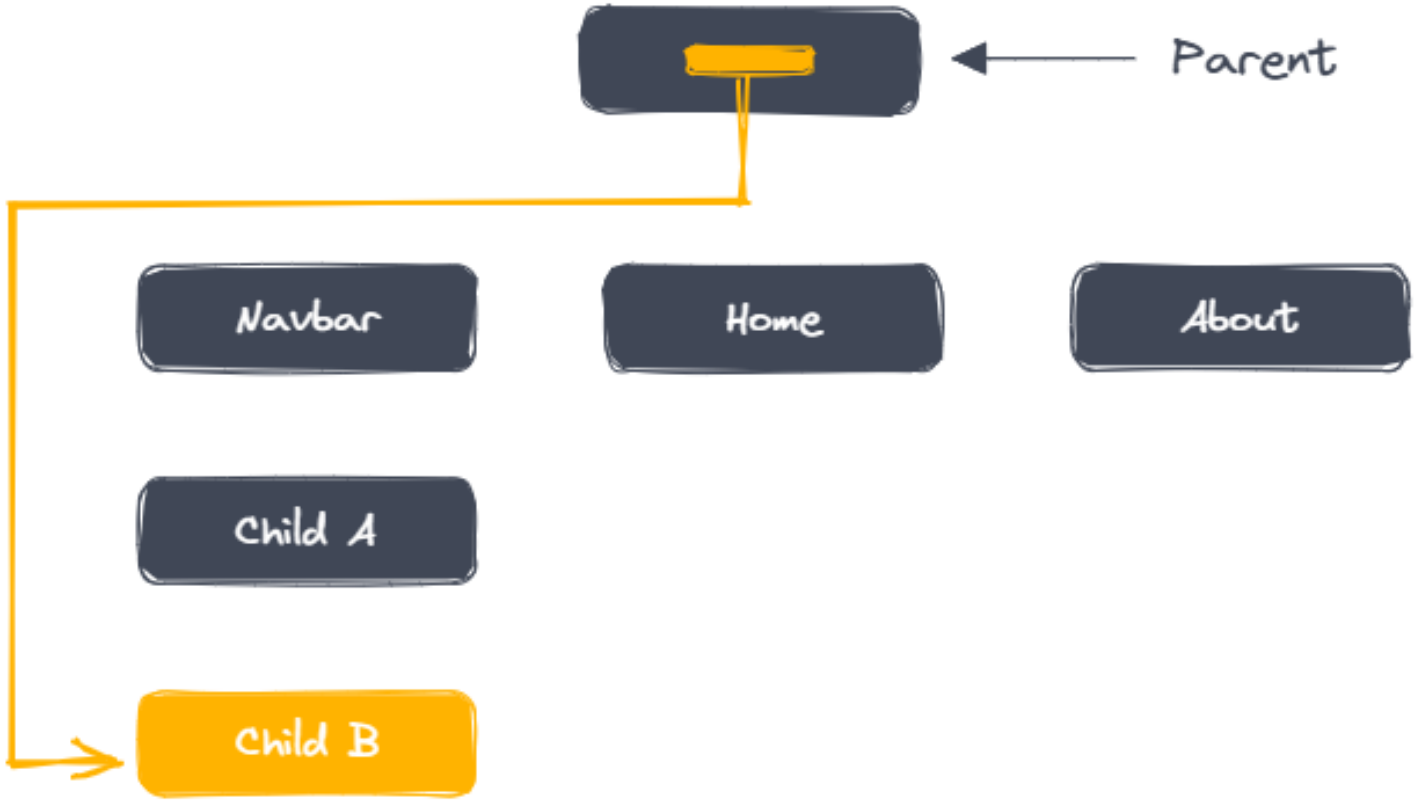
Fortunately, React provides a built-in feature known as the context API that helps “teleport” data to the components that need it without passing props.

While props are great for passing data from a parent component to a child component, there are certain situations where you may want to share data across multiple components without passing props down the component tree. This is where the **Context API** come into play.

The Problem with Passing Props



How the Context API Works



How the Context API Works

Context API consists of two main components:

- context provider
- context consumer

The provider is responsible for creating and managing the context, which holds the data to be shared between components.

On the other hand, the consumer is used to access the context and its data from within a component.

How to Get Started with the Context API

1. **Create a Context Object**

First, you need to create a context object using the **createContext()** function from the 'react' library. This context object will hold the data that you want to share across your application.

Create a new file named **MyContext.js** in the ***src*** folder and add the following code to create a context object:

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

export const MyContext = createContext("");
```

How to Get Started with the Context API

2. Wrap Components with a Provider

Once you've created a context object, you need to wrap the components that need access to the shared data with a Provider component.

The Provider component accepts a "value" prop that holds the shared data, and any component that is a child of the Provider component can access that shared data.

It's important to note that the Provider component should be wrapped around the top-level component
in an application to ensure that all child components have access to the shared data.

How to Get Started with the Context API

```
// Create a parent component that wraps child components with a Provider

import { useState, createContext, React } from "react";
import MyComponent from "./MyComponent";

const MyContext = createContext();

function App() {
  const [text, setText] = useState("");

  return (
    <div>
      <MyContext.Provider value={{ text, setText }}>
          <MyComponent />
      </MyContext.Provider>
    </div>
  );
}

export default App;
```

In this example, we have a parent component called App. This component has a state variable called "text", which is initially set to an empty string. We've also defined a function called setText that can be used to update the value of text.

Inside the return statement of the App component, we've wrapped the children of this component with the provider component ("MyContext.Provider"). Then we've passed an object to the value prop of the provider component that contains "text" and "setText" values.

How to Get Started with the Context API

3. Consume the Context

Now that we've created the provider component, we need to consume the context in other components.

To do this, we use the **"useContext" hook** from React.

```
import { useContext } from 'react';
import { MyContext } from './MyContext';

function MyComponent() {
  const { text, setText } = useContext(MyContext);

  return (
    <div>
      <h1>{text}</h1>
      <button onClick={() => setText('Hello, world!')}>
        Click me
      </button>
    </div>
  );
}

export default MyComponent;
```

In this example, we've used the useContext hook to access the "text" and "setText" variables
that were defined in the provider component.

Inside the return statement of "MyComponent", we've rendered a paragraph element that displays
the value of text. We've also rendered a button that, when clicked, will call the setText function to
update the value of text to "Hello, world!".

Use Cases of Context API

User Authentication:

You can also use Context API to store a user's authentication status and pass it down to all the components that need it.

This way, we can easily restrict access to
certain parts of your application based on the user's authentication status.

Accessing data from external sources:

Finally, you can use the Context API to store data retrieved
from external sources such as APIs or databases and make it available to all components.

This can simplify your code and make it easier to manage data across your application.

Best Practices for Context API

Keep Context API limited to global state management only:

It's best to use the Context API for managing state that needs to be accessed across multiple components in your application. Avoid using it for state that only needs to be accessed within a single component, as it can lead to unnecessary complexity and performance issues.

Use context providers sparingly:

While context providers can be a powerful tool for managing global state, it's generally a good idea to use them sparingly. Instead, consider using props to pass data down through
your component tree whenever possible.

What are children?

The **children,** in React, refer to the generic box whose contents are unknown until they're passed from the parent component.

What does this mean? It simply means that the component will display whatever is included in between the opening and closing tags while invoking the component.

What is children prop?

Children is a prop (**this.props.children**) that allow you to pass components as data to other components, just like any other prop you use. Component tree put between component's opening and closing tag will be passed to that component as **children** prop.

There are a number of methods available in the React API to work with this prop. These include **React.Children.map**, **React.Children.forEach**, **React.Children.count**, **React.Children.only**, **React.Children.toArray**.

children.prop

`Children` lets you manipulate and transform the JSX you received as the [`children` prop.]

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

function RowList({ children }) {
  return (
    <div className="RowList">
      {Children.map(children, child =>
        <div className="Row">
          {child}
        </div>
      )}
    </div>
  );
}
```

In this example , the RowList wraps every child it receives into a container. For example, let's say the parent component passes three <p> tags as the children prop to RowList:

The final rendered result will look like this:

```
<h1>Total rows: {Children.count(children)}</h1>
```

```
import RowList from './RowList.js';

export default function App() {
  return (
    <RowList>
      <p>This is the first item.</p>
      <p>This is the second item.</p>
      <p>This is the third item.</p>
    </RowList>
  );
}
```

```
<div className="RowList">
  <div className="Row">
    <p>This is the first item.</p>
  </div>
  <div className="Row">
    <p>This is the second item.</p>
  </div>
  <div className="Row">
    <p>This is the third item.</p>
  </div>
</div>
```

Thank You!