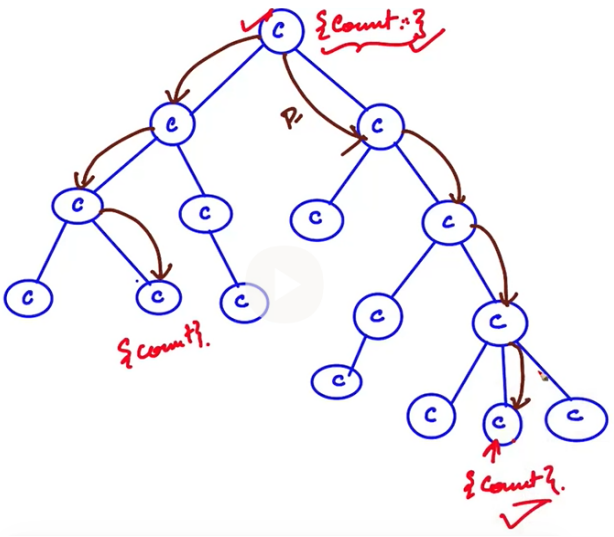
Context API

Passing Data From Parent to Child

Prop Drilling - Passing State from Parent to Child

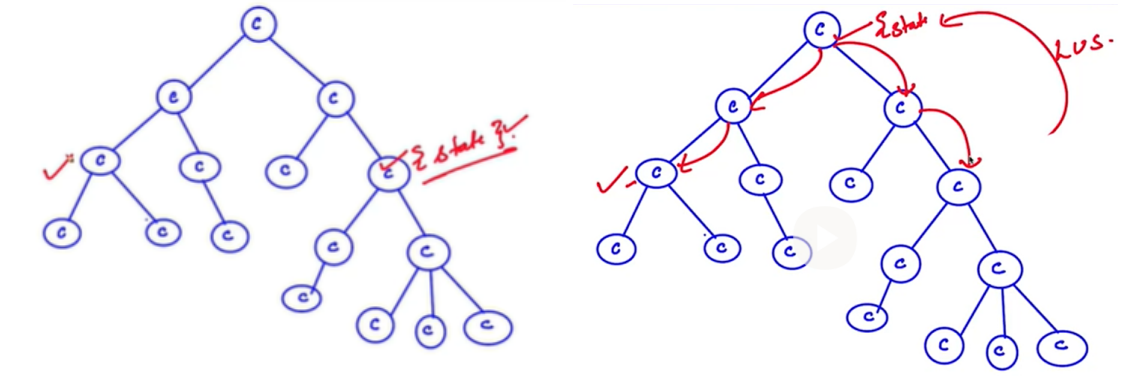
Your application might initially only have one component, but as it becomes more sophisticated, you must continue dividing it into smaller components. We can create a separation of concerns by isolating specific portions of a bigger application using components. Whenever anything in your program malfunctions, fault isolation makes it simple to pinpoint the problem area.

Props can be used to enable communication between our components in React. Prop drilling is a situation where data is passed from one component through multiple interdependent components until you get to the component where the data is needed. Prop drilling is not ideal as it quickly introduces complicated, hard-to-read code, re-rendering excessively, and slows down performance. Component re-rendering is especially damaging since passing data down multiple levels of components triggers the re-rendering of components unnecessarily.



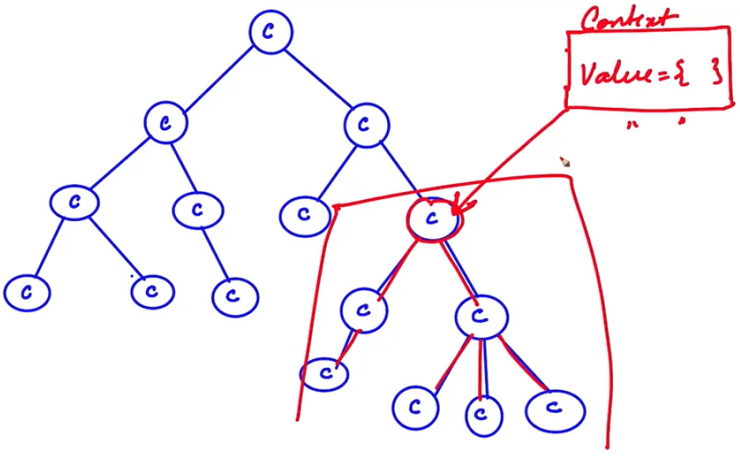
Lifting Up the State - Passing State to siblings

Lifting state up occurs when the state is placed in a common ancestor (or parent) of child components. Because each child component has access to the parent, they will then have access to the state (via prop drilling). If the state is updated inside the child component, it is lifted back to the parent container. However, as we are utilizing a poorly maintained pattern for the state, this approach can create issues in the future.



Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level. This is the alternative to "prop drilling" or moving props from grandparent to child to parent, and so on. Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.

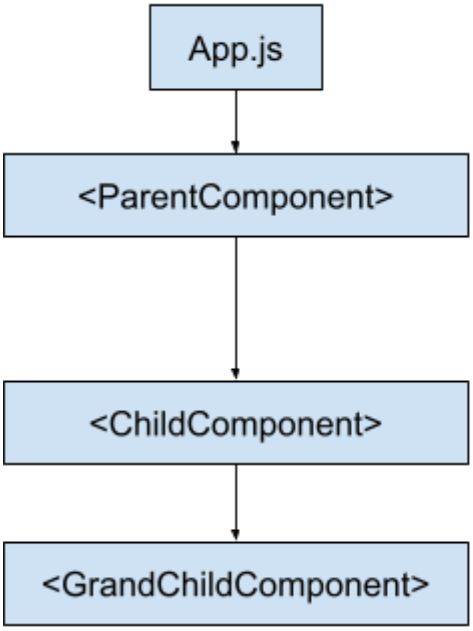


React Context API

The React Context API is a component structure that allows us to share data across all levels of the application. The main aim of Context API is to solve the problem of prop drilling (also called "Threading"). There are three steps to using React context:

1. **Create context -** using the `createContext` method.
2. **Provide Context -** Setup a `Context.provider` and define the data which you want to store.
3. **Consume the Context -** Use a `Context.consumer` or `useContext` hook whenever you need the data from the store.

Let's consider the following example:



1. Creating the Context

```
import { createContext } from "react";
export const colorContext= createContext();
```

2. Providing the Context

```
[color, setColor] = useState("#000000")
<colorContext.Provider value={{ color, setColor }}>
    <ChildComponent />
</colorContext.Provider>
```

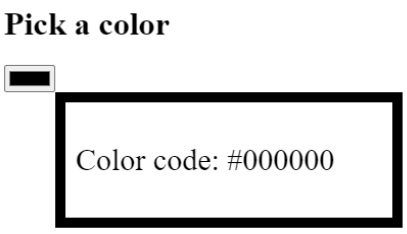
3. Consuming the Context (Way 1)

```
const {color} = useContext(colorContext);
```

3. Consuming the Context (Way 2)

```
<colorContext.Consumer>
    {(value) =>
        <p style={{ color: value.color }}>
            Color code: {value.color}
        </p>
    }
</colorContext.Consumer>
```

Output:



Creating the Context

React.createContext

`const MyContext = React.createContext(defaultValue);`

It is used for creating a Context object. When React renders a component subscribing to this Context object, it will read the current context value from the closest matching Provider above it in the tree.

```
import { createContext } from "react";
export const colorContext= createContext();
```

Providing the Context

Context.Provider

`<MyContext.Provider value={/* some value */}>`

Every Context object has a Provider React component which allows consuming components to subscribe to context changes. It acts as a delivery service. When a consumer component asks for something, it finds it in the context and provides it to where it is needed. The provider accepts a prop (value), and the data in this prop can be used in all the other child components. This value could be anything from the component state.

All consumers that are child components of a Provider will re-render whenever the Provider's value prop changes. Changes are determined by comparing the new and old values using the same algorithm as Object.is.

```
import { useState } from "react";
import ChildComponent from "./ChildComponent";
import { colorContext } from "../context";

const ParentComponent = (props) => {
  const [color, setColor] = useState("#000000");

  return (
```

```
    <>
      <h1>Pick a color</h1>
      <input type="color" onChange={(e) => { setColor(e.target.value);}}
value={color} />
      {/* Providing the context to the child component */}
      <colorContext.Provider value={{ color, setColor }}>
        <ChildComponent />
      </colorContext.Provider>
    </>
  );
};
```

Consuming the Context in Functional Components

useContext hook

`const value = useContext(SomeContext)`

useContext is a React Hook that lets you read and subscribe to context from your component.It can be used with the useState Hook to share the state between deeply nested components more easily.

```
import { useContext } from "react";
import { colorContext } from "../context";

const GrandChildComponent = () => {
  //Consuming the context
  const {value} = useContext(colorContext)
  return (
  <p style={{ color: value.color }}>Color code: {value.color}</p>
)
};
```

Consuming the Context in Class-Based Components

Context.Consumer

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
```

`</MyContext.Consumer>`

A React component that subscribes to context changes. Requires a function as a child. The function receives the current context value and returns a React node. The value argument passed to the function will equal the value prop of the closest Provider for this context in the component tree. If there is no Provider for this context, the value argument will be equal to the defaultValue, which was passed to createContext().

```
import React from 'react';
import { colorContext } from "../context";

class GrandChildComponent extends React.Component {
    render() {
        return (
            <colorContext.Consumer>
                {(value) => <p style={{ color: value.color }}>Color code:
{value.color}</p>}
            </colorContext.Consumer>
        );
    }
}
```

Using Multiple contexts

React also allows you to create multiple contexts. By providing multiple contexts in this way, components that require access to both context values can consume them both and be able to interact with their respective states. We should always try to separate contexts for different purposes to maintain the code structure and better readability.To keep context re-rendering fast, React needs to make each context consumer a separate node in the tree.

For Example, the Items component may need access to the item state from itemContext and the total state from totalContext, allowing it to display the total number of items in the shopping cart along with the total cost.

```
import { itemContext } from "./itemContext";
import { totalContext } from "./totalContext";
```

```
function App() {
  const [total, setTotal] = useState(0);
  const [item, setItem] = useState(0);
  return (
    // providing multiple contexts
    <itemContext.Provider value={{ item, setItem }}>
      <totalContext.Provider value={{ total, setTotal }}>
        <div className="App">
          <h2>Shopping Cart</h2>
          <Navbar />
          <Items />
        </div>
      </totalContext.Provider>
    </itemContext.Provider>
  );
}
export default App;
```

Custom Provider

It is a component which acts as a provider and it makes use of the default provider. Custom providers are created using the **createContext** function from the React library, which creates a new context object that can be passed down to child components using a provider component. The provider component is responsible for passing the context data down to its child components via a special prop called **value**.

By using a custom provider, you can centralize the management of shared data and state in a single place, making it easier to maintain and update your application. This can be particularly useful when working with complex applications that require a lot of shared state management, such as e-commerce sites or large data-driven applications.

For Example:

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

```
const itemContext = createContext();

function CustomItemContext({children}) {
  const [total, setTotal] = useState(0);
  const [item, setItem] = useState(0);

  return(
    <itemContext.Provider value={{total,setTotal,item, setItem}}>
        {children}
    </itemContext.Provider>
  )
}

export { itemContext };
export default CustomItemContext;
```

Using Custom Providers:

```
import CustomItemContext, { itemContext } from "./itemContext";

function App() {
  return (
    // providing multiple contexts
    <CustomItemContext>
      <div className="App">
        <h2>Shopping Cart</h2>
        <Navbar />
        <Items />
      </div>
    </CustomItemContext>
  );
}
export default App;
```

Using Custom Hooks

Creation of a custom provider component that provides context data to its child components, as well as the creation of a custom hook that consumes the context

data. Using custom hooks with custom providers allows for greater flexibility and reusability in sharing data across a React application. All the logic of the context file, updating logic and event handling, will be at one place

For Example, The **useValue** hook is defined to consume the context data provided by the **itemContext** using the **useContext** hook. The hook returns the total and item variables, functions handleAdd, and handleRemove from the context, making them available to any components that use the useValue hook.

```
import { createContext, useState, useContext } from "react";

const itemContext = createContext();

function useValue() {
  const value = useContext(itemContext);
  return value;
}

function CustomItemContext({ children }) {
  const [total, setTotal] = useState(0);
  const [item, setItem] = useState(0);

  const handleAdd = (price) => {
    setTotal(total + price);
    setItem(item + 1);
  };

  const handleRemove = (price) => {
    if (total <= 0) {
      return;
    }
    setTotal((prevState) => prevState - price);
    setItem(item - 1);
  };

  return (
    <itemContext.Provider value={{ total, item, handleAdd, handleRemove }}>
      {children}
```

```
    </itemContext.Provider>
  );
}

export {  useValue };
export default CustomItemContext;
```

Summarising it

Let's summarise what we have learned in this Lecture:

- Learned about Prop Drilling.
- Learned how to create Context.
- Learned how to provide Context.
- Learned how to consume the Context.
- Learned how to use multiple contexts.
- Learned about custom providers.
- Learned about custom hooks.

Some References:

- Context: link
- React Context for beginners: link