What is React Context?

React Context is a powerful feature that allows you to manage global state (or data) in a React application without having to pass props down manually through every level of the component tree. It's ideal for situations where many components need access to the same data, such as user authentication, themes, or localization settings.

When to Use Context?

Context is useful when:

- You have global data that many components need to access.
- You want to avoid "prop drilling," where props are passed down from parent to child components multiple layers deep.
- Your app's data structure doesn't require a full-fledged state management solution like Redux.

Basic Concept of React Context

1. **Create a Context**: First, you create a context using `React.createContext()`.
2. **Provide the Context**: Use the `Provider` component to make the context data available to all components inside it.
3. **Consume the Context**: Use the `useContext` hook (or `Context.Consumer` in class components) to access the context data within any child component.

Example 1: Simple Counter with React Context

Step-by-Step Explanation:

1. **Create a Context**: This will hold our global state (in this case, the `count`).
2. **Provider**: The `Provider` will wrap around the components where you want to access the context data.
3. **Consume Context**: Components will consume the context to get or modify the global state.

1. Create `CounterContext.js`

jsx code:-

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

// Create a Context with default value of 0 for count
const CounterContext = createContext();

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

  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);

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

export { CounterContext, CounterProvider };
```

Explanation:

- We created a `CounterContext` to manage the `count`.
- `CounterProvider` is a wrapper that makes `count`, `increment`, and `decrement` functions available to all components that consume this context.

2. Using the Context in a Component

Now, we will use the context to access and modify the `count`.

Jsx code:-

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

function Counter() {
  const { count, increment, decrement } = useContext(CounterContext);  //
Access context values

  return (
    <div>
      <h1>Count: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

export default Counter;
```

Explanation:

- `useContext(CounterContext)` allows the `Counter` component to access the values provided by `CounterProvider`. The component now has access to the `count` value, and the `increment` and `decrement` functions.

3. Putting It All Together in `App.js`

`Jsx code`

```
import React from 'react';
import { CounterProvider } from './CounterContext';
import Counter from './Counter';

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

export default App;
```

Explanation:

- We wrap the `Counter` component with `CounterProvider` to make the context available inside the `Counter` component.