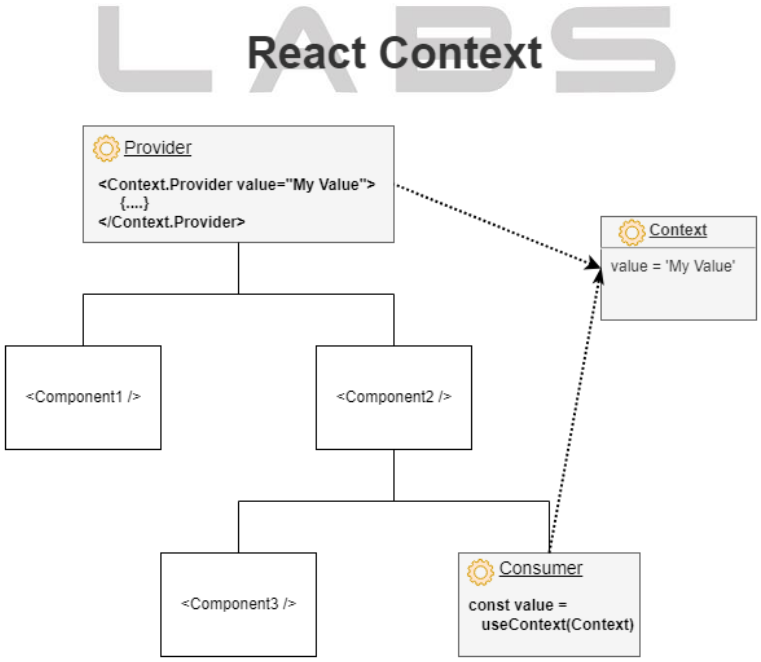
useContext Hook

- The useContext hook in React is a built-in hook that allows functional components to access and consume context values directly, making it easier and more intuitive to work with React Context.
- Context in React is a feature that allows you to share values (or "state") between components without having to pass props manually at every level of the component tree.
- It provides a way to avoid "prop drilling," which occurs when data is passed down through many layers of nested components, even if intermediate components don't use the data.
- The useContext hook is a powerful tool in React, simplifying data sharing and improving code readability when combined with the Context API.

Why is Context Needed

- **Avoid Prop Drilling:** It reduces the complexity of passing props through deeply nested component trees.
- **Global State Management:** It helps manage shared states like themes, authentication data, or language preferences across the app.
- **Simplifies Code:** Context makes code cleaner and easier to maintain when many components need access to the same state or data.



Why useContext is used?

- Before useContext, consuming a context value required using the <Context.Consumer> component, which often led to verbose and nested code.
- The useContext hook simplifies this process by providing direct access to the context value within a functional component.

Syntax

```
const contextValue = useContext(MyContext);
```

where,

MyContext: The context object created using React.createContext.

contextValue: The value provided by the nearest Provider component in the component tree.

Steps to Use useContext

- **Create a Context:** Use React.createContext to create a context object.
  ```
  const MyContext = React.createContext(MyValue);
  ```
- **Provide a Value:** Use the Provider component to supply a value to the context.
  ```
  <MyContext.Provider value={value}>
    <ChildComponent />
  </MyContext.Provider>
  ```
- **Consume the Value:** Use useContext in any functional component to access the context value.
  ```
  const value = useContext(MyContext);
  ```

Scenarios where Context is better than Prop Drilling

Prop drilling is manageable for small projects or shallow component trees, but it becomes difficult and complicated in some scenarios. Hence using context is better when dealing with operations like:

- **Theming:** Sharing a theme (e.g., dark/light mode) across components without passing the theme prop explicitly.
- **Authentication:** Providing the logged-in user's information (e.g., username, role) across the application.
- **Language Preferences:** Sharing the selected language for internationalization.
- **State Sharing:** Managing shared states between sibling components, such as data fetched from an API.

- **Complex Applications:** Avoiding prop drilling when multiple deeply nested components require the same data.

Example of Prop Drilling vs Context

Prop Drilling:

```
function App() {
  const user = { name: "Saurabh Bhardwaj", age: 30 };
  return <Parent user={user} />;
 }
 function Parent({ user }) {
  return <Child user={user} />;
 }
 function Child({ user }) {
  return <div>Hello, {user.name}, your age is {user.age}</div>;
 }
export default App;
```

Using Context:

```
import React from "react";
const UserContext = React.createContext();
function App() {
 const user = { name: "Saurabh", age: 30 };
 return (
  <UserContext.Provider value={user}>
   <Parent />
  </UserContext.Provider>
 );
}
function Parent() {
 return <Child />;
}
function Child() {
 const user = React.useContext(UserContext);
 return <div>Hello, {user.name}, your age is {user.age}</div>;
}
export default App;
```

Context with Default Value

If no Provider wraps the consumer component, the useContext hook returns the default value set in React.createContext.

Example

```
import React from "react";
const MyContext = React.createContext("Default Value");
function App() {
  return <ConsumerComponent />;
}
function ConsumerComponent() {
  const value = React.useContext(MyContext);
  return <div>{value}</div>;
}
export default App;
```

Context with Provided Value

To provide a value using a context provider, you need to wrap the ConsumerComponent (or any components that need the context value) with the MyContext.Provider and pass the desired value through the value prop.

Example

```
import React from "react";
const MyContext = React.createContext("Default Value");
function App() {
    return (
    <MyContext.Provider value="Provided Value">
      <ConsumerComponent />
    </MyContext.Provider>
  );
}
function ConsumerComponent() {
  const value = React.useContext(MyContext);
  return <div>{value}</div>;
}
export default App;
```

Example: Context Without Prop Drilling

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

const AuthContext = createContext();

function App() {
  const user = { username: "learn2earnlabs", isLoggedIn: false };

  return (
    <AuthContext.Provider value={user}>
      <MainComponent />
    </AuthContext.Provider>
  );
}

function MainComponent() {
  return <SubComponent />;
}

function SubComponent() {
  return <UserProfile />;
}

function UserProfile() {
  const auth = useContext(AuthContext); // Directly access context
  return (
    <div>
      {auth.isLoggedIn ? (
        <p>Welcome, {auth.username}!</p>
      ) : (
        <p>Please log in.</p>
      )}
    </div>
  );
}

export default App;
```

When to Avoid Using Context

While Context is powerful, it is not a one-size-fits-all solution. Overusing it or using it incorrectly can lead to challenges. Avoid Using Context When:

- **Tightly Coupled Components:** If the shared data is only relevant to a specific parent-child relationship, use props instead of Context.

  Example

```
import React from "react";

function App() {
  return (
    <div>
      <Parent />
    </div>
  );
}

function Parent() {
  const data = "My Secret Data";
  return <Child message={data} />;
}

function Child({ message }) {
  return <div>{message}</div>;
}

export default App;
```

- **Frequent Updates:** If the shared state updates frequently (e.g., large lists or animations), Context may cause unnecessary re-renders for all consumers.
  Better Approach: Use state management libraries like Redux or Zustand.

- **Unnecessary Overhead:** If data is not widely shared across multiple components, Context may add unnecessary complexity. Use Context only when multiple, deeply nested components need the same data.

- **Alternative Communication:** For sibling components that need to share data, consider lifting state up to a common parent instead of using Context.

Example

```
import React, { useState } from "react";
function App() {
  return (
    <div>      <Parent />    </div>
  );
}
function Parent() {
  const [value, setValue] = useState("Shared Data");
  return (
    <>
      <Child1 value={value} />
      <Child2 setValue={setValue} />
    </>
  );
}
function Child1({ value }) {
  return (
    <div>
      <h2>Child1 Component</h2>
      <p>Value from Parent: {value}</p>
    </div>
  );
}
function Child2({ setValue }) {
  return (
    <div>
      <h2>Child2 Component</h2>
      <button onClick={() => setValue("Updated Data")}>Update
Value</button>
    </div>
  );
}
export default App;
```

Key Takeaways with Prop Drilling & Context

Prop Drilling

- Suitable for shallow component hierarchies or when the data is specific to a single branch of the tree.
- Becomes problematic for deeply nested components.

Context

- Best for globally shared data (e.g., theme, authentication, language settings).
- Avoid overusing for simple data sharing, as it may lead to performance issues or unnecessary complexity.

Balance Between Both

- Use props for specific, localized data passing.
- Use Context for widely shared and static data that spans across multiple branches.

Benefits of useContext

- **Simplicity:** It removes the need for <Context.Consumer>, making the code shorter and cleaner.
- **Readability:** Using useContext results in more readable and maintainable functional components.
- **Direct Access:** It provides direct access to the context value without needing extra wrappers.

Best Practices for useContext

- **Combine Context with State Management:** Use useContext with useReducer or useState for efficient global state management.
- **Avoid Overuse:** Don't use context for state that only concerns a specific component or its immediate children.
- **Split Contexts:** For unrelated data, use separate contexts to prevent unnecessary re-renders.

Dynamic Context Management Using useContext Hook

Dynamic Context Management enables sharing of complex objects, state, or functions across a component tree without prop drilling. By combining the useContext hook with the Context API, components can easily consume shared logic or data.

- **Sharing Complex Objects or Functions in Context :** Instead of sharing just primitive values, you can share complex objects or functions to provide reusable logic across components.

Example: Sharing State and Functions

```
import React, { createContext, useContext, useState } from "react";
// Create Context
const UserContext = createContext();
// Context Provider
function UserProvider({ children }) {
  const [user, setUser] = useState({ name: "Neha Rathore", age: 25 });

  const updateUser = (newData) => {
                        setUser((prev) => ({ ...prev, ...newData }));
  };
  return (
    <UserContext.Provider value={{ user, updateUser }}>
     {children}
    </UserContext.Provider>
  );
}
// Consumer Component
function UserProfile() {
  const { user } = useContext(UserContext);
  return <p>User Name: {user.name}, Age: {user.age}</p>;
}
function UserUpdate() {
  const { updateUser } = useContext(UserContext);
  return <button onClick={() => updateUser({ age: 30 })}>Update
Age</button>;
}
```

```
function App() {
  return (
    <UserProvider>
                    <UserProfile />
                    <UserUpdate />
    </UserProvider>
  );
}

export default App;
```

- **Passing Down Callbacks:** Callbacks (e.g., increment, toggle) can be passed in Context to allow child components to modify the shared state.

Example: Counter with Increment/Decrement

```
import React, { createContext, useContext, useState } from "react";
// Create Context
const CounterContext = createContext();
// Context Provider
function CounterProvider({ children }) {
  const [count, setCount] = useState(0);
  const increment = () => setCount((prev) => prev + 1);
  const decrement = () => setCount((prev) => prev - 1);

 return (
    <CounterContext.Provider value={{ count, increment, decrement }}>
     {children}
    </CounterContext.Provider>
  );
}
// Consumer Components
function CounterDisplay() {
  const { count } = useContext(CounterContext);
  return <h2>Count: {count}</h2>;
}
```

```
function CounterControls() {
  const { increment, decrement } = useContext(CounterContext);
  return (
    <div>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

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

export default App;
```

Nested and Multiple Contexts in React

React allows you to use multiple contexts in a single application to manage distinct pieces of state independently. By nesting multiple context providers, you can structure your app's global state in a modular and scalable way.

Creating Multiple Context Providers

You can create multiple contexts to manage different concerns.

Example: ThemeContext + UserContext

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

// Create ThemeContext
const ThemeContext = createContext();
// Create UserContext
const UserContext = createContext();

// ThemeContext Provider
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");
  const toggleTheme = () => setTheme((prev) => (prev === "light" ? "dark" : "light"));

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </ThemeContext.Provider>
  );
}

// UserContext Provider
function UserProvider({ children }) {
  const [user, setUser] = useState({ name: "Garima Singh", role: "Tech Lead" });

  return (
    <UserContext.Provider value={{ user, setUser }}>
      {children}
```

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

// Consumer Component: Reads from ThemeContext and UserContext
function Dashboard() {
  const { theme, toggleTheme } = useContext(ThemeContext);
  const { user } = useContext(UserContext);

  return (
    <div style={{ background: theme === "light" ? "#fff" : "#333", color: theme ===
"light" ? "#000" : "#fff" }}>
      <h1>Welcome, {user.name}!</h1>
      <p>Your role: {user.role}</p>
      <button onClick={toggleTheme}>Toggle Theme</button>
    </div>
  );
}

// App Component with Nested Providers
function App() {
  return (
    <ThemeProvider>
            <UserProvider>
                    <Dashboard />
            </UserProvider>
    </ThemeProvider>
  );
}

export default App;
```

Managing Dependencies and Conflicts Between Contexts

When using multiple contexts, there may be dependencies or potential conflicts (e.g., theme depends on user preferences). These can be managed by:

- **Order of Providers:** Providers should wrap in a logical hierarchy to minimize dependency issues.
  For example, UserProvider can wrap ThemeProvider if the theme is user-specific.

- **Avoiding Context Overlap:** Avoid defining overlapping state concerns in multiple contexts.

Example: Managing Theme Based on User Preferences

```
function ThemeProvider({ children }) {
  const { user } = useContext(UserContext); // Access user from UserContext
  const [theme, setTheme] = useState(user.role === "Admin" ? "dark" : "light");
// Default theme depends on user role

  const toggleTheme = () => setTheme((prev) => (prev === "light" ? "dark" :
"light"));

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
     {children}
    </ThemeContext.Provider>
  );
}

// In App Component
function App() {
  return (
    <UserProvider>
      <ThemeProvider>
        <Dashboard />
      </ThemeProvider>
    </UserProvider>
  );
}
```

Optimizing Performance in Context Usage

When using React's Context API, it's essential to understand how it impacts component rendering. Without proper optimization, unnecessary re-renders can occur, degrading performance.

Understanding Re-Renders in Context

- **How Context Triggers Re-Renders:** When a context value changes:
  - All components consuming that context (using useContext) re-render.
  - Even components that don't rely on the changed part of the context will still re-render if they consume the context.

- **Why Unnecessary Re-Renders Happen**
  - **Large Context Values:** When you store a large object or multiple states in one context, changing any part of it triggers re-renders for all consuming components.
  - **Broad Consumption:** When multiple components consume the same context but don't need all the values, they still re-render when the context changes.

Example: Single Context Causing Unnecessary Re-Renders

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

// Create a single context
const AppContext = createContext();

function AppProvider({ children }) {
  const [theme, setTheme] = useState("light");
  const [user, setUser] = useState({ name: "Neha Rathore", age: 25 });

  const toggleTheme = () => setTheme((prev) => (prev === "light" ? "dark"
: "light"));

  return (
    <AppContext.Provider value={{ theme, toggleTheme, user }}>
     {children}
    </AppContext.Provider>
```

```
  );
}

function ThemeToggle() {
  const { theme, toggleTheme } = useContext(AppContext); // Only depends
on theme
  console.log("ThemeToggle re-rendered");
  return (
    <div>
      <p>Theme: {theme}</p>
      <button onClick={toggleTheme}>Toggle Theme</button>
    </div>
  );
}

function UserInfo() {
  const { user } = useContext(AppContext); // Only depends on user
  console.log("UserInfo re-rendered");
  return <p>User: {user.name}, Age: {user.age}</p>;
}

function App() {
  return (
    <AppProvider>
      <ThemeToggle />
      <UserInfo />
    </AppProvider>
  );
}

export default App;
```

Issue:- When theme changes, both ThemeToggle and UserInfo re-render, even though UserInfo does not depend on theme.

Technique to Optimize Context Usage

- **Splitting Contexts to Isolate State:** By separating concerns into multiple contexts, you can reduce unnecessary re-renders. Components only consume the context they depend on.

Example: Separate Contexts for UI Settings and User Data

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

// Create separate contexts
const ThemeContext = createContext();
const UserContext = createContext();

// ThemeContext Provider
function ThemeProvider({ children }) {
  const [theme, setTheme] = useState("light");
  const toggleTheme = () => setTheme((prev) => (prev === "light" ? "dark"
: "light"));

  return (
    <ThemeContext.Provider value={{ theme, toggleTheme }}>
     {children}
    </ThemeContext.Provider>
  );
}

// UserContext Provider
function UserProvider({ children }) {
  const [user, setUser] = useState({ name: "Neha Rathore", age: 25 });

  return <UserContext.Provider value={{ user
}}>{children}</UserContext.Provider>;
}

// Consumer Components
function ThemeToggle() {
  const { theme, toggleTheme } = useContext(ThemeContext);
  console.log("ThemeToggle re-rendered");
```

```
  return (
    <div>
      <p>Theme: {theme}</p>
      <button onClick={toggleTheme}>Toggle Theme</button>
    </div>
  );
}

function UserInfo() {
  const { user } = useContext(UserContext);
  console.log("UserInfo re-rendered");
  return <p>User: {user.name}, Age: {user.age}</p>;
}

// App Component
function App() {
  return (
    <ThemeProvider>
      <UserProvider>
        <ThemeToggle />
        <UserInfo />
      </UserProvider>
    </ThemeProvider>
  );
}

export default App;
```

Benefits:

- Changing theme only re-renders ThemeToggle.
- Updating user only re-renders UserInfo.