React Context API - Comprehensive Notes

What is React Context API?

The Context API is a built-in feature in React that enables you to share state across the entire app (or part of it) without passing props manually through every level of the component tree.

When to Use Context

- When data needs to be accessible by many components at different nesting levels
- When passing props through intermediate components becomes cumbersome ("prop drilling")
- For global state management: user authentication, theme preferences, language settings, etc.

Core Parts of Context API

1. Creating a Context

jsx

```jsx
// usercontext.js
import React from 'react';

// Create a Context object with optional default value
const UserContext = React.createContext(null);

export default UserContext;
```

2. Creating a Provider Component

jsx

```jsx
// UserContextProvider.jsx
import React from 'react';
import UserContext from './usercontext';

const UserContextProvider = ({children}) => {
    // State that will be shared
    const [user, setUser] = React.useState(null);

    return (
        // The Provider component makes value available to all children
        <UserContext.Provider value={{user, setUser}}>
            {children}
        </UserContext.Provider>
    );
}

export default UserContextProvider;
```

3. Wrapping Components with the Provider

jsx

```jsx
// App.jsx
import React from 'react';
import UserContextProvider from './context/UserContextProvider';
import Login from './components/Login';
import Profile from './components/Profile';

function App() {
  return (
    <UserContextProvider>
      <h1>My App</h1>
      <Login />
      <Profile />
    </UserContextProvider>
  );
}
```

4. Consuming the Context (Two Methods)

Method 1: useContext Hook (Recommended)

jsx

```jsx
// Profile.jsx
import React, { useContext } from "react";
import UserContext from "../context/usercontext";

function Profile() {
    // Extract values from context
    const { user } = useContext(UserContext);

    if (!user) {
        return <div>Please login</div>;
    }

    return <div>Welcome {user.username}</div>;
}
```

Method 2: Context Consumer Component (Older Approach)

jsx

```jsx
import React from "react";
import UserContext from "../context/usercontext";

function Profile() {
    return (
        <UserContext.Consumer>
            {({ user }) => {
                if (!user) {
                    return <div>Please login</div>;
                }
                return <div>Welcome {user.username}</div>;
            }}
        </UserContext.Consumer>
    );
}
```

Common Context-Related Patterns

1. Creating and Updating Context Data

jsx

```jsx
// Login.jsx
import React, { useState, useContext } from 'react';
import UserContext from '../context/usercontext';

function Login() {
    const [username, setUsername] = useState('');
    const [password, setPassword] = useState('');

    // Get the setUser function from context
    const { setUser } = useContext(UserContext);

    const handleSubmit = (e) => {
        e.preventDefault();
        // Update the context state
        setUser({ username, password });
    }

    return (
        <form onSubmit={handleSubmit}>
            {/* form inputs */}
            <button type="submit">Login</button>
        </form>
    );
}
```

2. Multiple Contexts

You can have multiple separate contexts in your application:

jsx

```jsx
// Different context for theme
const ThemeContext = React.createContext('light');

// Different context for language
const LanguageContext = React.createContext('en');

function App() {
    return (
        <ThemeContext.Provider value="dark">
            <LanguageContext.Provider value="fr">
                <ThemedComponent />
            </LanguageContext.Provider>
        </ThemeContext.Provider>
    );
}
```

3. Context with Reducer (For Complex State)

jsx

```jsx
import React, { useReducer } from 'react';

const UserContext = React.createContext();

// Define reducer
function userReducer(state, action) {
    switch (action.type) {
        case 'LOGIN':
            return { ...action.payload };
        case 'LOGOUT':
            return null;
        default:
            return state;
    }
}

function UserContextProvider({ children }) {
    const [user, dispatch] = useReducer(userReducer, null);

    return (
        <UserContext.Provider value={{ user, dispatch }}>
            {children}
        </UserContext.Provider>
    );
}
```

Best Practices

1. **Separate Concerns**: Create different contexts for different domains (user, theme, etc.)
2. **Default Values**: Provide meaningful default values when creating context

    jsx

    ```jsx
    const UserContext = React.createContext({
        user: null,
        setUser: () => {}
    });
    ```

3. **Custom Hook for Context**: Create a custom hook for each context

```jsx
// useUser.js
import { useContext } from 'react';
import UserContext from './usercontext';

export function useUser() {
    const context = useContext(UserContext);
    if (context === undefined) {
        throw new Error('useUser must be used within a UserProvider');
    }
    return context;
}
```

Then use it in components:

```jsx
import { useUser } from '../hooks/useUser';

function Profile() {
    const { user } = useUser();
    // ...
}
```

4. **Performance Optimization**: Context triggers re-renders when its value changes, so:
   - Memoize objects provided to context when needed
   - Consider splitting contexts that change at different rates

Context vs Other State Management

When to Use Context

- Simple to medium complexity applications
- When data doesn't change frequently
- When you want to avoid external dependencies

When to Consider Alternatives (Redux, Zustand, etc.)

- Very complex state logic
- Performance issues with large state changes
- Need for middleware, time-travel debugging, etc.

Context Limitations

1. **Performance**: Not optimized for high-frequency updates

2. **Complexity**: Can become unwieldy if overused for too many concerns
3. **Testing**: Components using context need to be wrapped in providers for testing

Example: Complete Authentication Flow with Context

Context Setup

jsx

```jsx
// authContext.js
import React, { createContext, useState, useContext, useEffect } from 'react';

const AuthContext = createContext();

export function AuthProvider({ children }) {
    const [user, setUser] = useState(null);
    const [loading, setLoading] = useState(true);

    // Check for saved user on initial load
    useEffect(() => {
        const savedUser = localStorage.getItem('user');
        if (savedUser) {
            setUser(JSON.parse(savedUser));
        }
        setLoading(false);
    }, []);

    // Login function
    const login = (userData) => {
        setUser(userData);
        localStorage.setItem('user', JSON.stringify(userData));
    };

    // Logout function
    const logout = () => {
        setUser(null);
        localStorage.removeItem('user');
    };

    return (
        <AuthContext.Provider value={{
            user,
            login,
            logout,
            isAuthenticated: !!user,
            loading
        }}>
            {children}
        </AuthContext.Provider>
    );
}

// Custom hook
export function useAuth() {
```

```
    return useContext(AuthContext);
}
```

Usage in Components

jsx

```jsx
// App.jsx
import { AuthProvider } from './context/authContext';

function App() {
    return (
        <AuthProvider>
            <Router>
                {/* Routes */}
            </Router>
        </AuthProvider>
    );
}

// LoginPage.jsx
import { useAuth } from '../context/authContext';

function LoginPage() {
    const { login } = useAuth();

    const handleSubmit = async (e) => {
        e.preventDefault();
        // Call API for authentication
        const userData = await loginApi(username, password);
        login(userData);
    };

    // Render login form
}

// PrivateRoute.jsx
import { useAuth } from '../context/authContext';
import { Navigate } from 'react-router-dom';

function PrivateRoute({ children }) {
    const { user, loading } = useAuth();

    if (loading) {
        return <div>Loading...</div>;
    }

    if (!user) {
        return <Navigate to="/login" />;
    }
```

```
    return children;
}
```