📆 Hari 4 (Kamis) - State Autentikasi Global (Context API)

🎯 Tujuan Pembelajaran Hari Ini

- Memahami keterbatasan pengecekan status autentikasi langsung dari `localStorage`.
- Memahami konsep state global dan kebutuhannya dalam manajemen autentikasi.
- Mampu mengimplementasikan state autentikasi global menggunakan React Context API.
- Mengintegrasikan state autentikasi dari Context API dengan komponen `ProtectedRoute` dan komponen UI lainnya.

📚 Materi Inti (2 Jam)

1. Masalah: Mengecek `localStorage` Berulang, Kebutuhan State Global

Di Hari 3, kita mengecek status autentikasi dengan membaca `localStorage.getItem('authToken')` langsung di komponen `ProtectedRoute`. Ini berfungsi, tetapi memiliki beberapa kelemahan:

- **Redundansi:** Jika kita perlu menampilkan status login (misalnya, menampilkan nama user atau tombol Logout di header) di banyak komponen, kita harus mengulang logika pengecekan `localStorage` di setiap komponen tersebut.
- **Reaktivitas:** Perubahan pada `localStorage` (misalnya, saat user login atau logout) tidak secara otomatis memicu re-render komponen yang membaca `localStorage`. Kita perlu mekanisme tambahan untuk memberi tahu komponen bahwa status autentikasi telah berubah.
- **Komunikasi Antar Komponen:** Sulit untuk memberi tahu komponen lain bahwa user telah login atau logout tanpa prop drilling atau mekanisme state management lainnya.

Solusinya adalah menggunakan **State Global**. State global memungkinkan kita menyimpan status autentikasi di satu tempat yang bisa diakses oleh *komponen mana pun* dalam aplikasi, tanpa perlu passing props secara manual dari atas ke bawah (prop drilling). Ketika state global ini berubah, semua komponen yang 'mendengarkan' state tersebut akan otomatis di-render ulang.

2. Solusi: State Global

State global adalah data yang disimpan di luar hierarki komponen biasa, namun dapat diakses dan diubah oleh komponen mana pun yang membutuhkannya. Ini sangat berguna untuk data yang bersifat 'global' atau dibutuhkan di banyak bagian aplikasi, seperti:

- Status autentikasi user
- Data user yang sedang login
- Tema aplikasi (terang/gelap)
- Pengaturan bahasa
- Status loading global

React menyediakan beberapa cara untuk mengelola state global, salah satunya adalah Context API.

3. Menggunakan Context API untuk Autentikasi

React Context API menyediakan cara untuk berbagi data (state) antar komponen tanpa harus secara eksplisit meneruskan prop melalui setiap level tree komponen. Ini sangat cocok untuk data seperti status autentikasi.

Context API terdiri dari tiga bagian utama:

- **`React.createContext`**: Untuk membuat objek Context baru. Objek ini berisi dua komponen: `Provider` dan `Consumer` (atau lebih umum digunakan dengan hook `useContext`).
- **`Provider`**: Komponen yang 'menyediakan' nilai (state dan fungsi) ke komponen-komponen di bawahnya dalam tree. Semua komponen di dalam `Provider` dapat mengakses nilai yang disediakan.
- **`useContext` Hook**: Hook yang digunakan di komponen fungsional untuk 'mengonsumsi' nilai dari Context terdekat di atasnya.

4. Membuat Context dan Provider (`AuthContext`, `AuthProvider`)

Kita akan membuat file baru, misalnya `src/contexts/AuthContext.js`, untuk menampung Context dan Provider autentikasi kita.

```
// src/contexts/AuthContext.js
import React, { createContext, useContext, useState, useEffect } from
'react';

// 1. Buat Context
const AuthContext = createContext(null);

// 2. Buat Provider
export const AuthProvider = ({ children }) => {
  // State untuk menyimpan status autentikasi (boolean) dan data user
(opsional)
  const [isAuthenticated, setIsAuthenticated] = useState(false);
  const [user, setUser] = useState(null); // Bisa simpan data user jika ada

  // Efek samping untuk membaca token dari localStorage saat aplikasi
pertama kali dimuat
  useEffect(() => {
    const token = localStorage.getItem('authToken');
    if (token) {
      // Di sini Anda bisa menambahkan logika untuk memvalidasi token
      // atau mengambil data user dari backend jika diperlukan
      setIsAuthenticated(true);
      // setUser(decodeToken(token)); // Contoh: jika menggunakan JWT dan
perlu data user
    } else {
      setIsAuthenticated(false);
      setUser(null);
    }
  }, []); // Array kosong berarti efek ini hanya berjalan sekali saat mount

  // Fungsi untuk login
```

```
  const login = (token, userData = null) => {
    localStorage.setItem('authToken', token);
    setIsAuthenticated(true);
    setUser(userData);
    // Mungkin perlu redirect user setelah login di komponen Login itu
sendiri
  };

  // Fungsi untuk logout
  const logout = () => {
    localStorage.removeItem('authToken');
    setIsAuthenticated(false);
    setUser(null);
    // Mungkin perlu redirect user ke halaman login setelah logout
  };

  // Nilai yang akan disediakan oleh Provider
  const value = {
    isAuthenticated,
    user,
    login,
    logout,
  };

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

// 3. Buat Hook Kustom untuk Menggunakan Context
export const useAuth = () => {
  const context = useContext(AuthContext);
  if (context === undefined) {
    throw new Error('useAuth must be used within an AuthProvider');
  }
  return context;
};

// Export AuthContext juga jika diperlukan, tapi useAuth lebih umum
digunakan
export default AuthContext;
```

Penjelasan:

- `AuthContext` dibuat dengan nilai default `null`.
- `AuthProvider` adalah komponen yang membungkus bagian aplikasi yang memerlukan akses ke state autentikasi. Ia mengelola state `isAuthenticated` dan `user`.
- `useEffect` digunakan untuk membaca `localStorage` saat aplikasi pertama kali dimuat. Ini penting agar status login tetap terjaga saat user me-refresh halaman.
- Fungsi `login` dan `logout` disediakan untuk mengubah state dan `localStorage`.
- Objek `value` berisi state (`isAuthenticated`, `user`) dan fungsi (`login`, `logout`) yang akan dibagikan.

- `AuthContext.Provider value={value}` membuat nilai `value` tersedia untuk semua komponen di dalamnya.
- Hook kustom `useAuth` adalah cara yang disarankan untuk mengonsumsi Context. Ini membuat kode lebih bersih dan menambahkan pengecekan apakah hook digunakan di dalam `AuthProvider`.

Untuk menggunakan `AuthProvider`, kita perlu membungkus root komponen aplikasi kita (biasanya di `src/index.js` atau `src/App.js`) dengan `AuthProvider`:

```
// src/index.js (atau src/App.js jika Anda me-render App langsung di sini)
import React from 'react';
import ReactDOM from 'react-dom/client';
import App from './App';
import { AuthProvider } from './contexts/AuthContext'; // Import
AuthProvider

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <AuthProvider> {/* Bungkus App dengan AuthProvider */}
      <App />
    </AuthProvider>
  </React.StrictMode>
);
```

5. Mengintegrasikan Context dengan `ProtectedRoute`

Sekarang, komponen `ProtectedRoute` kita tidak perlu lagi membaca `localStorage` secara langsung. Ia bisa mendapatkan status autentikasi dari Context menggunakan hook `useAuth`.

```
// src/components/ProtectedRoute.jsx
import React from 'react';
import { Navigate, Outlet } from 'react-router-dom';
import { useAuth } from '../contexts/AuthContext'; // Import useAuth hook

const ProtectedRoute = () => {
  // Ambil status autentikasi dari Context API
  const { isAuthenticated } = useAuth();

  // Jika user terautentikasi, render child routes
  // Jika tidak, redirect ke halaman login
  return isAuthenticated ? <Outlet /> : <Navigate to="/login" replace />;
};

export default ProtectedRoute;
```

Kode ini jauh lebih bersih dan `ProtectedRoute` sekarang sepenuhnya bergantung pada state global yang dikelola oleh `AuthProvider`.

6. Mengintegrasikan Context dengan Komponen Lain (misalnya Header)

Komponen UI lain yang perlu mengetahui status autentikasi (misalnya, untuk menampilkan tombol Login/Register atau tombol Logout dan nama user) juga bisa menggunakan hook `useAuth`.

```
// src/components/Header.jsx (Contoh)
import React from 'react';
import { Link } from 'react-router-dom';
import { useAuth } from '../contexts/AuthContext'; // Import useAuth hook

const Header = () => {
  const { isAuthenticated, user, logout } = useAuth(); // Ambil state dan
fungsi dari Context

  return (
    <header>
      <nav>
        <Link to="/">Home</Link>
        {isAuthenticated ? (
          // Tampilkan ini jika user sudah login
          <>
            <span>Selamat datang, {user?.name || 'User'}</span> {/*
Tampilkan nama user jika ada */}
            <Link to="/dashboard">Dashboard</Link>
            <button onClick={logout}>Logout</button> {/* Panggil fungsi
logout dari Context */}
          </>
        ) : (
          // Tampilkan ini jika user belum login
          <>
            <Link to="/login">Login</Link>
            <Link to="/register">Register</Link>
          </>
        )}
      </nav>
    </header>
  );
};

export default Header;
```

Dengan `useAuth()`, komponen `Header` bisa dengan mudah mengakses `isAuthenticated`, `user`, dan fungsi `logout` tanpa prop drilling.

🏋️ Praktik Mandiri (8 Jam)

1. **Lanjutkan proyek e-commerce Anda.** Pastikan praktik Hari 3 sudah selesai dan Protected Routes berfungsi menggunakan pengecekan `localStorage`.

2. **Buat file `AuthContext.js`:** Buat folder baru `src/contexts` dan di dalamnya buat file `AuthContext.js`. Salin kode Context API, Provider (`AuthProvider`), dan hook `useAuth` seperti contoh di atas.
3. **Integrasikan `AuthProvider`:** Di file entry point aplikasi Anda (misalnya `src/index.js` atau `src/App.js`), impor `AuthProvider` dan bungkus komponen root (`<App />`) dengannya.
4. **Modifikasi login/logout handler:** Di komponen `LoginPage` dan komponen lain yang menangani proses login/logout, impor hook `useAuth`. Ganti logika `localStorage.setItem/removeItem` dengan memanggil fungsi `login()` dan `logout()` yang didapat dari `useAuth()`.

   ```
   // Contoh di LoginPage.jsx setelah berhasil login
   import { useAuth } from '../contexts/AuthContext';
   import { useNavigate } from 'react-router-dom';

   // ... di dalam komponen LoginPage ...
   const { login } = useAuth();
   const navigate = useNavigate();

   const handleLoginSuccess = (token, userData) => {
     login(token, userData); // Panggil fungsi login dari Context
     navigate('/dashboard'); // Redirect ke halaman dashboard
   };
   // ...
   ```

   Lakukan hal serupa untuk logout di komponen yang relevan (misalnya, tombol Logout di Header).
5. **Modifikasi `ProtectedRoute`:** Buka `src/components/ProtectedRoute.jsx`. Hapus logika pengecekan `localStorage` dan ganti dengan menggunakan `const { isAuthenticated } = useAuth();`.
6. **Gunakan `useAuth()` di komponen UI:** Identifikasi komponen UI yang perlu menampilkan status login (misalnya, Header, Sidebar, atau komponen lain yang menampilkan data user). Impor `useAuth()` dan gunakan `isAuthenticated` serta `user` untuk menampilkan UI yang sesuai.
7. **Pastikan alur autentikasi berfungsi dengan Context API:**
   - Refresh halaman: Pastikan status login tetap terjaga.
   - Login: Pastikan status `isAuthenticated` berubah menjadi `true`, `user` terisi, dan UI (misalnya Header) berubah menampilkan tombol Logout.
   - Akses Protected Route: Pastikan Anda bisa mengakses `/dashboard` setelah login dan diarahkan ke `/login` saat belum login.
   - Logout: Pastikan status `isAuthenticated` berubah menjadi `false`, `user` menjadi `null`, dan UI kembali menampilkan tombol Login/Register.

💡 Tips Belajar Tambahan

- **Struktur Folder:** Menggunakan folder `src/contexts` adalah praktik umum untuk menampung Context API.
- **Initial State:** Perhatikan bagaimana `useEffect` digunakan di `AuthProvider` untuk membaca `localStorage` saat aplikasi pertama kali dimuat. Ini krusial untuk 'mengingat' status login user.

- **Data User:** Anda bisa menyimpan lebih banyak data user (seperti nama, email, role) di state `user` dalam Context setelah login berhasil, agar data ini mudah diakses di seluruh aplikasi.
- **Loading State:** Untuk pengalaman user yang lebih baik, Anda bisa menambahkan state `isLoading` di Context untuk menandai saat aplikasi sedang memverifikasi token awal atau sedang dalam proses login/logout.

🔗 Referensi Tambahan

- React Documentation - Context
- React Documentation - useContext Hook
- React Documentation - createContext
- React Documentation - useEffect Hook

Hari ini kita telah berhasil mengimplementasikan state autentikasi global menggunakan React Context API. Ini adalah langkah besar dalam membuat manajemen autentikasi di aplikasi kita lebih terpusat, reaktif, dan mudah diakses. Besok, kita akan mereview kembali alur autentikasi, membahas kelebihan dan kekurangan Context API untuk state global yang lebih kompleks, dan mendapatkan pengantar tentang library state management lanjutan seperti Redux Toolkit dan Zustand, serta mempersiapkan tugas mingguan.