📅 Hari 1 (Senin) - Mengambil Data dengan `fetch` dan `useEffect`

🎯 Tujuan Pembelajaran Hari Ini

- Memahami konsep side effects dalam konteks React.
- Menguasai penggunaan `useEffect` hook untuk mengelola side effects di komponen fungsional.
- Memahami cara kerja dependency array pada `useEffect`.
- Mampu menggunakan `fetch` API bawaan browser untuk mengambil data dari API.
- Menggabungkan `useEffect` dan `fetch` untuk fetching data saat komponen di-render.
- Mampu menangani state loading dan error saat proses fetching data.

📚 Materi Inti (2 Jam)

1. Pengantar Side Effects di React

- **Apa itu Side Effect?**
  - Dalam pemrograman fungsional, fungsi murni (pure function) selalu menghasilkan output yang sama untuk input yang sama dan tidak memiliki efek samping (side effect).
  - Side effect adalah segala sesuatu yang terjadi di luar cakupan fungsi itu sendiri, yang memengaruhi dunia luar atau dipengaruhi oleh dunia luar.
  - Contoh side effect dalam aplikasi web:
    - Mengambil data dari API (fetching data).
    - Memanipulasi DOM secara langsung (meskipun di React ini jarang dilakukan).
    - Mengatur atau membersihkan timer (setTimeout, setInterval).
    - Setup atau membersihkan event listeners.
    - Berinteraksi dengan local storage atau cookies.
    - Setup subscriptions (misalnya, ke WebSocket).
- **Mengapa Kita Butuh Mengelola Side Effects di Komponen Fungsional?**
  - Komponen fungsional di React pada dasarnya adalah fungsi murni yang menerima props dan mengembalikan elemen React (JSX).
  - Namun, aplikasi nyata seringkali perlu melakukan operasi yang memiliki side effect, seperti mengambil data dari backend.
  - Jika side effect dilakukan langsung di dalam body komponen fungsional, ini bisa menyebabkan masalah:
    - Fetching data setiap kali komponen me-render ulang (infinite loop).
    - Tidak ada cara untuk membersihkan side effect (misalnya, membatalkan request yang sedang berjalan saat komponen di-unmount).
  - React menyediakan mekanisme khusus, yaitu Hooks, untuk mengelola state dan side effects di komponen fungsional.

2. `useEffect` Hook

- **Sintaks Dasar `useEffect`**

- `useEffect` adalah hook yang memungkinkan Anda melakukan side effects di komponen fungsional.
- Sintaksnya adalah `useEffect(setup, dependencies?)`.
- Argumen pertama (`setup`) adalah fungsi yang berisi kode side effect Anda.
- Argumen kedua (`dependencies`) adalah array opsional dari nilai-nilai yang menjadi "ketergantungan" effect Anda.

```
import React, { useEffect } from 'react';

function MyComponent() {
  useEffect(() => {
    // Kode side effect Anda di sini
    console.log('Effect dijalankan!');

    // Opsional: Cleanup function
    return () => {
      console.log('Cleanup dijalankan!');
      // Kode untuk membersihkan side effect
    };
  }, [/* dependency array opsional */]); // <-- Dependency Array

  return (
    <div>
      {/* JSX komponen */}
    </div>
  );
}
```

- **Kapan `useEffect` Dijalankan?**
  - Secara default (jika dependency array tidak ada), `useEffect` dijalankan:
    1. Setelah render pertama komponen.
    2. Setelah *setiap* update (setiap kali props atau state berubah).
  - Jika dependency array kosong (`[]`), `useEffect` hanya dijalankan:
    1. Setelah render pertama komponen.
       - Ini setara dengan `componentDidMount` pada class component.
  - Jika dependency array berisi nilai (`[dep1, dep2]`), `useEffect` dijalankan:
    1. Setelah render pertama komponen.
    2. Setelah setiap update *hanya jika* nilai `dep1` atau `dep2` berubah dari render sebelumnya.
       - Ini setara dengan `componentDidMount` dan `componentDidUpdate` (dengan kondisi pada props/state tertentu) pada class component.
- **Cleanup Function**
  - Fungsi yang dikembalikan dari fungsi `setup` di dalam `useEffect` disebut cleanup function.
  - Cleanup function dijalankan:
    1. Sebelum effect dijalankan kembali (jika dependency array ada dan dependencies berubah).
    2. Saat komponen di-unmount.
  - Ini sangat penting untuk membersihkan side effect yang mungkin terus berjalan dan menyebabkan memory leak atau perilaku yang tidak diinginkan (misalnya, membatalkan timer,

unsubscribe dari subscription, membatalkan request API yang belum selesai).

3. Mengambil Data dengan Native `fetch` API

- **Pengantar `fetch` API**
  - `fetch` adalah API bawaan browser modern untuk membuat permintaan HTTP (seperti GET, POST, PUT, DELETE) ke server.
  - Ini adalah alternatif yang lebih modern dan fleksibel dibandingkan `XMLHttpRequest`.
  - `fetch` mengembalikan Promise, yang memudahkan penanganan respons secara asynchronous.
- **Sintaks Dasar `fetch`**
  - `fetch(url, options?)`
  - `url`: URL endpoint API yang ingin diakses.
  - `options`: Objek konfigurasi opsional (method, headers, body, dll.). Default method adalah GET.

```
fetch('https://api.example.com/data')
  .then(response => {
    // Handle response
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json(); // Mengubah response menjadi JSON
  })
  .then(data => {
    // Gunakan data yang diterima
    console.log(data);
  })
  .catch(error => {
    // Tangani error
    console.error('There was a problem with the fetch operation:',
error);
  });
```

- **Menggunakan `async/await` dengan `fetch`**
  - `async/await` adalah sintaks yang membuat kode asynchronous terlihat dan berperilaku lebih seperti kode synchronous, membuatnya lebih mudah dibaca dan ditulis.
  - Fungsi yang menggunakan `await` harus dideklarasikan sebagai `async`.

```
async function fetchData() {
  try {
    const response = await fetch('https://api.example.com/data');
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('There was a problem with the fetch operation:',
```

```
error);
  }
}

fetchData();
```

- **Menangani Response dan Error**
  - Response dari `fetch` adalah objek Response. Anda perlu memanggil metode seperti `.json()`, `.text()`, `.blob()` untuk mendapatkan body response.
  - Penting untuk memeriksa properti `response.ok` atau `response.status` untuk mengetahui apakah permintaan berhasil (status 2xx) sebelum mencoba memproses body response.
  - Error jaringan (misalnya, server tidak merespons) akan ditangkap oleh `.catch()` atau blok `catch` pada `async/await`. Namun, respons dengan status error HTTP (misalnya, 404 Not Found, 500 Internal Server Error) *tidak* akan dianggap sebagai error oleh `fetch` itu sendiri; Anda harus memeriksanya secara manual seperti di contoh di atas.

4. Menggabungkan `useEffect` dan `fetch`

- **Contoh Implementasi Fetching Data Saat Komponen Pertama Kali Di-render**
  - Kasus paling umum adalah mengambil data saat komponen pertama kali muncul di layar (mount).
  - Gunakan `useEffect` dengan dependency array kosong (`[]`) agar effect hanya berjalan sekali setelah render awal.
  - Fungsi fetching data (bisa berupa fungsi `async`) dipanggil di dalam callback `useEffect`.

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

function ProductList() {
  const [products, setProducts] = useState([]);
  const [loading, setLoading] = useState(true);
  const [error, setError] = useState(null);

  useEffect(() => {
    async function fetchProducts() {
      try {
        const response = await
fetch('https://api.example.com/products');
        if (!response.ok) {
          throw new Error('Failed to fetch products');
        }
        const data = await response.json();
        setProducts(data); // Update state dengan data produk
      } catch (err) {
        setError(err); // Update state jika ada error
      } finally {
        setLoading(false); // Selesai loading
      }
    }
```

```
    fetchProducts(); // Panggil fungsi fetching

    // Cleanup function opsional jika diperlukan (misalnya,
membatalkan request)
    // return () => { abortController.abort(); };

  }, []); // Dependency array kosong: effect hanya berjalan saat mount

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

  if (error) {
    return <div>Error: {error.message}</div>;
  }

  return (
    <div>
      <h1>Product List</h1>
      <ul>
        {products.map(product => (
          <li key={product.id}>{product.name} - ${product.price}</li>
        ))}
      </ul>
    </div>
  );
}

export default ProductList;
```

- **Menangani Loading State dan Error State**
  - Saat fetching data, ada tiga kemungkinan state:
    1. **Loading:** Data sedang diambil.
    2. **Success:** Data berhasil diambil.
    3. **Error:** Terjadi kesalahan saat mengambil data.
  - Penting untuk mengelola state ini di komponen Anda menggunakan `useState`.
  - Tampilkan indikator loading saat `loading` bernilai `true`.
  - Tampilkan pesan error jika `error` tidak `null`.
  - Tampilkan data saat `loading` bernilai `false` dan `error` bernilai `null`.
  - Gunakan blok `try...catch...finally` dalam fungsi fetching asynchronous untuk menangani sukses, error, dan menyelesaikan state loading.

🏋 Praktik Mandiri (8 Jam)

1. **Setup Proyek:** Buat proyek React baru menggunakan Create React App atau Vite.
2. **Buat Komponen:** Buat komponen fungsional baru, misalnya `PostList.js`.
3. **Gunakan `useState`:** Tambahkan state menggunakan `useState` untuk menyimpan daftar post (`posts`), status loading (`loading`), dan pesan error (`error`). Inisialisasi `posts` dengan array kosong, `loading` dengan `true`, dan `error` dengan `null`.

4. **Gunakan `useEffect`:** Tambahkan `useEffect` hook ke komponen `PostList`.
5. **Implementasi Fetching:** Di dalam callback `useEffect`, buat fungsi asynchronous (misalnya `fetchPosts`) yang menggunakan `fetch` API untuk mengambil data dari placeholder API, contoh: `https://jsonplaceholder.typicode.com/posts`.
6. **Handle Response:** Di dalam fungsi `fetchPosts`, tunggu response, periksa `response.ok`, parse response menjadi JSON (`response.json()`).
7. **Update State:** Jika fetch berhasil, update state `posts` dengan data yang diterima. Jika terjadi error (baik jaringan atau status HTTP non-OK), update state `error`.
8. **Handle Loading:** Gunakan blok `finally` di dalam fungsi `fetchPosts` untuk mengatur state `loading` menjadi `false` setelah fetch selesai (baik sukses maupun error).
9. **Render Kondisional:** Di dalam JSX komponen `PostList`, render konten secara kondisional berdasarkan state `loading` dan `error`.
    - Tampilkan "Loading..." jika `loading` true.
    - Tampilkan pesan error jika `error` tidak null.
    - Tampilkan daftar post (misalnya, judul post dalam list `<ul>`) jika loading false dan error null.
10. **Tampilkan Komponen:** Render komponen `PostList` di `App.js` atau komponen utama lainnya.
11. **Verifikasi:** Jalankan aplikasi Anda dan pastikan daftar post muncul setelah loading, atau pesan error muncul jika ada masalah.
12. **(Opsional) Cleanup:** Jika Anda menggunakan timer atau subscription, tambahkan cleanup function di `useEffect`.

💡 Tips Belajar Tambahan

- **Pahami Lifecycle Komponen:** Kaitkan penggunaan `useEffect` dengan lifecycle komponen (mounting, updating, unmounting) untuk memahami kapan effect dan cleanup function dijalankan.
- **Perhatikan Dependency Array:** Ini adalah bagian paling krusial dari `useEffect`. Salah menentukan dependency array bisa menyebabkan effect berjalan terlalu sering (masalah performa) atau tidak berjalan sama sekali saat seharusnya (bug).
- **Jangan Lupa Error Handling:** Selalu siapkan mekanisme untuk menangani error saat fetching data.
- **Gunakan AbortController:** Untuk request API yang mungkin berjalan lama, pertimbangkan menggunakan `AbortController` untuk membatalkan request di cleanup function `useEffect` saat komponen di-unmount. Ini mencegah update state pada komponen yang sudah tidak ada, menghindari error.

🔗 Referensi Tambahan

- React Documentation - Using the Effect Hook
- MDN Web Docs - Using the Fetch API
- A Complete Guide to useEffect

Hari ini kita sudah mempelajari cara mengambil data dari API menggunakan `fetch` dan mengelola prosesnya dengan `useEffect`. Ini adalah pola dasar yang akan sering kita gunakan. Besok, kita akan melihat alternatif `fetch` yaitu library `axios` dan mulai menampilkan data produk dari backend yang sudah kita buat di minggu sebelumnya.