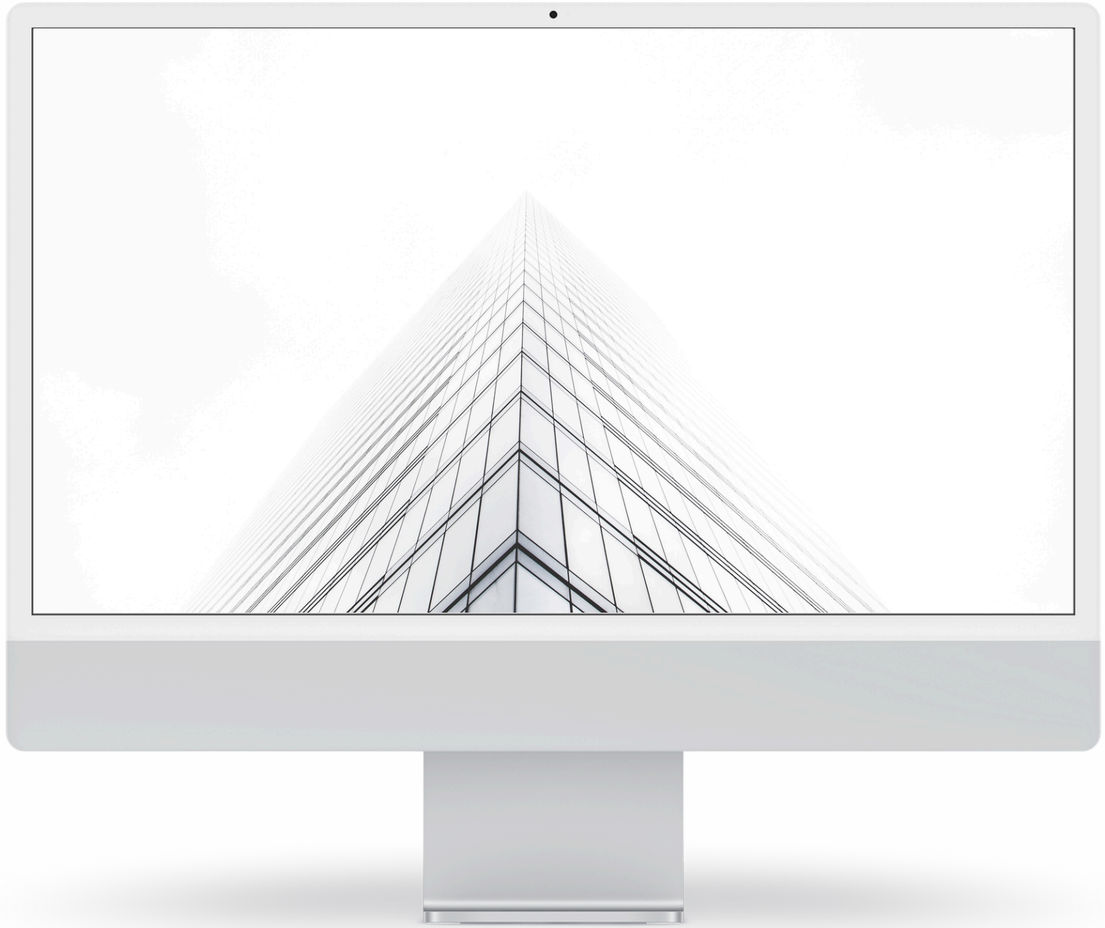
Pemrograman Web Modern

Insan Taufik, S.Kom., M.Kom.



Pengantar Pemrograman Web Modern & React JS

Sub-Materi

Konsep SPA (Single Page Application). Apa itu React? Sejarah, keunggulan, dan ekosistem React. Persiapan Lingkungan (Node.js, npm/yarn, VS Code). Membuat proyek React pertama (Vite/Create React App).

Tujuan Pembelajaran (Capaian)

Memahami tren web modern, mengenal React sebagai library utama, dan berhasil menjalankan proyek React sederhana.

Apa itu Data Fetching?

Definisi:
Proses mengambil data dari server atau API eksternal ke aplikasi React.

Mengapa Penting?

- Aplikasi modern butuh data real-time
- Data biasanya disimpan di server, bukan frontend
- Komunikasi dengan database melalui API

Konsep Kunci:

- **Asynchronous**: Tidak blocking UI
- **RESTful API**: Standar komunikasi client-server
- **HTTP Methods**: GET, POST, PUT, DELETE

Metode Data Fetching di React

1. Fetch API
   - Built-in browser API
   - Modern dan powerful
   - Promise-based

2. Axios
   - Library external
   - Fitur lebih lengkap
   - Automatic JSON transformation

3. React Query/SWR (Advanced)
   - Library khusus data fetching
   - Caching & synchronization
   - Untuk aplikasi kompleks

Fetch API vs Axios

Feature	Fetch API	Axios
Installation	Built-in browser	Need to install
JSON Transform	Manual .json()	Automatic
Request Timeout	Tidak ada	Ada
Error Handling	Hanya network errors	Semua HTTP errors
Browser Support	Modern browsers	Wider support

Data Fetching dengan useEffect

```
  useEffect(() => {
    const fetchData = async () => {
      try {
        const response = await fetch('https://api.example.com/data');
        if (!response.ok) throw new Error('Network response was not ok');
        const result = await response.json();
        setData(result);
      } catch (err) {
        setError(err.message);
      } finally {
        setLoading(false);
      }
    };

    fetchData();
  }, []); // Empty dependencies = run once

  // Render logic...
};
```

State Management untuk Data Fetching

Three States Pattern:

1. Loading: Data sedang dimuat
2. Error: Terjadi kesalahan
3. Success: Data berhasil dimuat

```
// Conditional Rendering berdasarkan state
if (loading) return <LoadingSpinner />;
if (error) return <ErrorMessage error={error} />;
if (!data) return <NoDataMessage />;

return <DataDisplay data={data} />;
```

Error Handling yang Baik

```
try {
  setLoading(true);
  setError(null);

  const response = await fetch(API_URL);

  if (!response.ok) {
    throw new Error(`HTTP error! status: ${response.status}`);
  }

  const result = await response.json();
  setData(result);

} catch (err) {
  setError(err.message);
  console.error('Fetch error:', err);
} finally {
  setLoading(false);
}
};
```

Best Practices:

- Reset error state sebelum request baru
- Handle HTTP error status codes
- Gunakan finally block untuk set loading false

Loading States & User Experience

Jenis Loading Indicator:

- Skeleton Screens: Placeholder konten
- Spinners: Untuk aksi sederhana
- Progress Bars: Untuk proses panjang

Prinsip UX:

- Berikan feedback segera
- Jangan biarkan user bingung
- Handle state "empty data"

Optimasi Data Fetching

```
// 1. Dependency-based fetching
useEffect(() => {
  fetchData(userId);
}, [userId]);
```

Ini adalah cara paling umum melakukan data fetching di React.
useEffect dijalankan setiap kali userId berubah.
Artinya, saat userId berubah → fetchData(userId) dipanggil lagi → data baru diambil.

Kelebihan:

- Simpel dan efisien untuk kasus di mana data tergantung pada state tertentu.
- React otomatis tahu kapan perlu mem-fetch ulang (berdasarkan dependency array).

Kelemahan:

- Tidak mencegah race condition:
- Jika userId berubah cepat, request pertama belum selesai tapi request kedua sudah jalan, hasil dari request lama bisa “menimpa” hasil baru.

Solusi

→

```
// 2. Cancel obsolete requests
useEffect(() => {
  const abortController = new AbortController();

  const fetchData = async () => {
    try {
      const response = await fetch(URL, {
        signal: abortController.signal
      });
      // Process data...
    } catch (err) {
      if (err.name !== 'AbortError') {
        setError(err.message);
      }
    }
  };

  fetchData();

  return () => abortController.abort();
}, [dependency]);
```

Ringkasan dan Best Practices

Yang telah dipelajari:

- ✓ Fetch data dengan Fetch API dan Axios
- ✓ Integrasi dengan useEffect
- ✓ Handling loading, error, dan success states
- ✓ Optimasi dan cleanup

Best Practices:

- Always handle loading and error states
- Use cleanup untuk avoid memory leaks
- Implement proper error handling
- Consider user experience during loading