Analogi Performa dalam Aplikasi React

Bayangkan aplikasi React seperti sebuah pabrik. Setiap kali ada pesanan baru (misalnya, data kota diminta atau URL berubah), pabrik harus memprosesnya dan menghasilkan produk (rendering komponen). Jika pesanan yang sama datang berulang kali, pabrik tidak perlu memproses ulang dari awal, melainkan cukup menggunakan produk yang sudah jadi (hasil yang sudah dihitung atau data yang sudah dimuat sebelumnya).

Di React, kita dapat menghindari pembuatan ulang data atau komponen yang tidak perlu menggunakan teknik **memoization**. Hal ini akan mengurangi "overhead" atau beban performa yang tidak perlu.

Penjelasan Performa dalam Kode Komponen `City`

1. **Menghindari Fetch Ulang Data dengan `useCallback` dan `useEffect`**

Pada komponen `City`, kita melakukan pemanggilan data kota berdasarkan `id` dari URL. Namun, setiap kali komponen dirender ulang, React bisa saja memanggil API lagi jika tidak mengontrol pemanggilan tersebut.

Untuk mengatasi hal ini, kita menggunakan `useCallback` untuk memastikan bahwa fungsi `getCity` tidak dibuat ulang setiap kali rendering.

Penyebab Masalah Performa:

- **Pemanggilan API yang tidak perlu**: Setiap kali komponen `City` dirender ulang, fungsi `getCity` akan dipanggil lagi, yang menyebabkan data kota dimuat ulang berulang kali.
- **Render ulang yang tidak perlu**: Jika state atau props berubah, maka React akan merender ulang komponen yang bergantung pada state tersebut. Hal ini dapat menyebabkan render yang berlebihan.

Solusi: **Memoize dengan `useCallback`**

Fungsi `getCity` di dalam `useEffect` dapat memanggil API setiap kali `id` berubah. Agar pemanggilan API hanya terjadi ketika benar-benar dibutuhkan, kita bisa menggunakan `useCallback` untuk memastikan bahwa `getCity` hanya dibuat sekali selama `id` tetap sama.

```
const { getCity, currentCity, isLoading } = useCities();
```

Di dalam `useCities`, kita telah membungkus `getCity` dengan `useCallback`, yang memastikan bahwa `getCity` hanya berubah jika `currentCity.id` berubah.

```
const getCity = useCallback(
  async function getCity(id) {
    if (Number(id) === currentCity.id) return; // Cek apakah kota yang
dimuat sudah sesuai
    dispatch({ type: "loading" });
    try {
      const res = await fetch(`${BASE_URL}/cities/${id}`);
      const data = await res.json();
      dispatch({ type: "city/loaded", payload: data });
    } catch {
```

```
        dispatch({
          type: "rejected",
          payload: "There was an error loading the city.....",
        });
      }
    },
    [currentCity.id] // Hanya berubah jika currentCity.id berubah
  );
```

Penjelasan:

- **`useCallback`** mencegah fungsi `getCity` dibuat ulang setiap kali komponen merender ulang, yang bisa menghindari pemanggilan API yang tidak perlu.
- Fungsi `getCity` hanya dipanggil jika `id` yang diminta berbeda dengan `currentCity.id`. Ini memastikan kita tidak melakukan fetch data untuk kota yang sama.

2. **Optimasi `useEffect` untuk Menghindari Render Ulang yang Tidak Perlu**

Secara default, **`useEffect`** akan dipanggil setiap kali komponen dirender ulang, yang menyebabkan data di-fetching berulang kali. Namun, dengan memberikan **`id`** sebagai dependensi, kita memastikan bahwa **`getCity`** hanya dipanggil ketika `id` berubah.

```
useEffect(
  function () {
    getCity(id); // Panggil API hanya jika id berubah
  },
  [id, getCity] // Hanya bergantung pada id dan getCity (yang sudah di-
memoize)
);
```

Penjelasan:

- Dengan memasukkan `id` dan `getCity` dalam array dependensi, **`useEffect`** hanya akan mengeksekusi `getCity(id)` ketika `id` berubah. Ini mencegah pemanggilan API yang tidak perlu saat render ulang.

3. **Mencegah Rendering Berlebihan pada Komponen Anak**

Jika komponen `City` memiliki banyak komponen anak yang bergantung pada state atau props yang sama, kita bisa menggunakan teknik **`React.memo`** pada komponen anak untuk memblokir render ulang yang tidak perlu.

Misalnya, komponen `BackButton` bisa dibungkus dengan `React.memo` untuk mencegah rendering ulang saat tidak ada perubahan pada props.

```
const BackButton = React.memo(() => {
  return <button>Back</button>;
});
```

Dengan menggunakan `React.memo`, React hanya akan merender ulang `BackButton` jika props yang diterimanya berubah, menghindari render ulang yang tidak perlu.

4. **Menangani `isLoading` dengan Spinner**

Ketika data sedang dimuat, kita menampilkan `Spinner`. Untuk performa, kita pastikan bahwa `Spinner` hanya dirender saat `isLoading` benar-benar `true`. React secara otomatis menghindari merender ulang komponen yang tidak berubah.

```
if (isLoading) return <Spinner />;
```

Kesimpulan Peningkatan Performa

1. **Memoization dengan `useCallback`**: Menghindari pembuatan fungsi baru setiap kali komponen dirender, sehingga mengurangi pemanggilan API yang tidak perlu.
2. **`useEffect` dengan dependensi yang tepat**: Memastikan pemanggilan API hanya terjadi saat `id` berubah, bukan setiap kali komponen dirender ulang.
3. **`React.memo` pada komponen anak**: Menghindari render ulang yang tidak perlu pada komponen yang props-nya tidak berubah.
4. **Optimasi render spinner**: Menampilkan `Spinner` hanya saat data benar-benar sedang dimuat, mengurangi beban render pada komponen.

Dengan menggunakan teknik-teknik ini, kita dapat menghindari pengulangan pemanggilan API yang tidak perlu, mengoptimalkan render komponen, dan secara keseluruhan meningkatkan performa aplikasi React.