# Mengatasi Masalah Performa dalam Aplikasi React (Next App)

## Pengertian

Pada aplikasi React, performa yang baik sangat penting, terutama ketika aplikasi mulai berkembang dengan banyak komponen dan data. Aplikasi ini menggunakan beberapa teknik untuk menangani masalah performa, seperti **lazy loading** dan **suspense**. Pada kode yang diberikan, kita melihat penggunaan teknik-teknik tersebut untuk meningkatkan performa dan mengurangi waktu pemuatan.

### Cara Berpikir React

Pada React, rendering ulang komponen dapat menyebabkan kinerja yang buruk, terutama jika banyak data yang harus dimuat atau komponen yang perlu dirender berulang kali. Untuk mengatasi masalah ini, kita bisa menggunakan beberapa teknik pengoptimalan seperti **lazy loading** dan **code splitting**.

## Analogi Sederhana

Bayangkan aplikasi Anda seperti sebuah restoran besar dengan banyak menu. Alih-alih menyajikan semua menu sekaligus di awal (yang bisa membebani dapur dan pelanggan), Anda hanya menyajikan menu yang dipesan oleh pelanggan. Dengan cara ini, dapur tidak perlu mempersiapkan semua bahan makanan sekaligus, dan pelanggan tidak perlu menunggu terlalu lama untuk mendapatkan apa yang mereka inginkan. Ini adalah prinsip dasar dari **lazy loading**: hanya memuat apa yang diperlukan saat itu juga.

## Teknik yang Digunakan untuk Mengatasi Masalah Performa

Pada kode ini, kita menggunakan beberapa teknik untuk mengoptimalkan performa aplikasi. Berikut penjelasan tentang teknik-teknik yang digunakan:

### 1. **Lazy Loading dengan `React.lazy`**

```
const Homepage = lazy(() => import("./pages/Homepage"));
const Product = lazy(() => import("./pages/Product"));
const Pricing = lazy(() => import("./pages/Pricing"));
const Login = lazy(() => import("./pages/Login"));
const AppLayout = lazy(() => import("./pages/AppLayout"));
const PageNotFound = lazy(() => import("./pages/PageNotFound"));
```

- **Lazy loading** adalah teknik untuk menunda pemuatan komponen sampai komponen tersebut benar-benar diperlukan. Dalam contoh ini, komponen `Homepage`, `Product`, `Pricing`, dan lainnya hanya akan dimuat ketika dibutuhkan, bukan saat aplikasi pertama kali dimuat.
- Dengan menggunakan `React.lazy`, kita membagi aplikasi menjadi **chunks** kecil, yang hanya dimuat jika pengguna menavigasi ke rute yang memerlukan komponen tersebut. Ini mengurangi ukuran bundle yang diunduh di awal dan mempercepat waktu muat halaman pertama.

## 2. **Suspense untuk Menangani Waktu Tunggu**

```
<Suspense fallback={<SpinnerFullPage />}>
```

- **Suspense** adalah fitur React yang memungkinkan kita untuk menampilkan fallback (misalnya, spinner atau loading state) saat menunggu pemuatan komponen yang di-*lazy load*. Ini memberi pengalaman pengguna yang lebih baik, karena mereka tidak akan melihat halaman kosong sementara aplikasi memuat data atau komponen.
- Dalam aplikasi ini, kita menggunakan `SpinnerFullPage` sebagai fallback, yang ditampilkan sementara komponen yang dibutuhkan sedang dimuat. Suspense memastikan bahwa halaman tetap responsif dan memberi tahu pengguna bahwa data sedang dimuat.

## 3. **Routing dan Pemrosesan Rute**

```
<Routes>
  <Route index element={<Homepage />} />
  <Route path="product" element={<Product />} />
  <Route path="pricing" element={<Pricing />} />
  <Route path="login" element={<Login />} />
  <Route
    path="app"
    element={
      <ProtectedRoute>
        <AppLayout />
      </ProtectedRoute>
    }
  >
    <Route index element={<Navigate replace to="cities" />} />
    <Route path="cities" element={<CityList />} />
    <Route path="cities/:id" element={<City />} />
    <Route path="countries" element={<CountryList />} />
    <Route path="form" element={<Form />} />
  </Route>
  <Route path="*" element={<PageNotFound />} />
</Routes>
```

- `BrowserRouter` dan `Routes` digunakan untuk menentukan bagaimana pengguna bisa menavigasi antara berbagai bagian aplikasi.
- Menggunakan `ProtectedRoute` untuk melindungi rute tertentu (misalnya, halaman dalam aplikasi yang hanya dapat diakses oleh pengguna yang terautentikasi).
- **Lazy loading** bekerja dengan baik di sini, karena komponen hanya dimuat ketika rute yang sesuai diakses. Sebagai contoh, komponen `Product` hanya dimuat jika pengguna mengunjungi rute `/product`, bukan dimuat di awal.

## 4. **Menyediakan Context dengan Provider**

```
<AuthProvider>
  <CitiesProvider>
    <BrowserRouter>
      <Suspense fallback={<SpinnerFullPage />}>
        <Routes>...</Routes>
      </Suspense>
    </BrowserRouter>
  </CitiesProvider>
</AuthProvider>
```

- **Provider** untuk `AuthProvider` dan `CitiesProvider` digunakan untuk menyediakan data dan state kepada komponen anak dalam aplikasi. **Context API** ini sangat berguna untuk berbagi data seperti status otentikasi pengguna dan daftar kota ke seluruh aplikasi tanpa harus mengoper props dari komponen ke komponen.
- Dengan menggunakan **Provider** untuk menyimpan state yang diperlukan, kita dapat menghindari render ulang komponen yang tidak relevan dengan perubahan state tertentu, yang membantu mengoptimalkan performa.

### 5. **Protected Route untuk Akses yang Terjaga**

```
<Route
  path="app"
  element={
    <ProtectedRoute>
      <AppLayout />
    </ProtectedRoute>
  }
>
```

- `ProtectedRoute` adalah komponen yang melindungi rute tertentu dari akses yang tidak sah. Jika pengguna tidak terautentikasi, mereka akan diarahkan ke halaman login. Teknik ini juga dapat membantu dalam pengoptimalan performa karena hanya mengizinkan render komponen tertentu jika pengguna memenuhi syarat (misalnya, sudah login).

## Masalah Performa yang Dapat Diperbaiki dengan Teknik Ini

### 1. **Mengurangi Waktu Muat Awal**

- Dengan menggunakan **lazy loading** pada komponen dan hanya memuat komponen yang dibutuhkan saat itu juga, kita mengurangi ukuran bundle yang perlu diunduh oleh browser pengguna. Ini mempercepat waktu muat halaman pertama dan meningkatkan pengalaman pengguna.

### 2. **Meningkatkan Responsif dengan Suspense**

- Dengan menggunakan **Suspense** dan menampilkan spinner atau indikator pemuatan, kita menghindari tampilan kosong atau halaman yang tidak responsif, yang dapat membuat pengguna merasa aplikasi tidak bekerja dengan baik.

### 3. **Menghindari Pemanggilan Berulang**

- Dengan menggunakan **React Context** dan **Provider**, kita memastikan bahwa data yang dibutuhkan (misalnya, informasi kota atau status autentikasi) dapat diakses dengan efisien tanpa perlu memanggil API atau memuat ulang komponen yang tidak perlu.

### 4. **Pengelolaan Akses dengan Protected Route**

- Dengan menggunakan **ProtectedRoute**, kita hanya merender komponen-komponen yang perlu diakses oleh pengguna yang terautentikasi, mengurangi jumlah komponen yang dimuat dalam aplikasi dan menjaga performa tetap optimal.

## Kesimpulan

Dalam aplikasi React, masalah performa bisa muncul ketika aplikasi memuat terlalu banyak data atau komponen sekaligus. Dengan menerapkan **lazy loading**, **Suspense**, dan **code splitting**, kita bisa mengurangi waktu muat dan menghindari rendering berlebihan. Selain itu, dengan menggunakan **Context API** dan **Protected Route**, kita dapat menjaga performa dengan hanya memuat data yang relevan dan memastikan akses aplikasi tetap terjaga.

Menggunakan teknik-teknik ini, aplikasi menjadi lebih responsif, lebih cepat dimuat, dan lebih efisien dalam hal penggunaan sumber daya.