Panduan Pemahaman Kode React untuk Aplikasi Kota

Pengertian Umum

Aplikasi ini mengelola data kota menggunakan **React Context** dan **useReducer**. Aplikasi ini menyediakan fungsi untuk mengambil daftar kota, mengambil detail kota tertentu, membuat kota baru, dan menghapus kota. Semua data kota ini dikelola dalam state global menggunakan **useReducer**, dan dapat diakses di seluruh aplikasi menggunakan **Context API**.

Cara Berpikir React

Pada React, kita membagi aplikasi menjadi komponen-komponen kecil. Setiap komponen dapat mengelola dan menampilkan data yang dibutuhkan. Data ini dikelola dalam **state**, dan React otomatis memperbarui tampilan ketika state berubah.

Pada aplikasi ini, kita menggunakan **Context API** untuk menyediakan data kota ke seluruh aplikasi dan **useReducer** untuk mengelola perubahan state yang lebih kompleks, seperti loading, error, dan daftar kota yang terus berubah.

Analogi Sederhana

Bayangkan kamu sedang mengelola **daftar kota** yang ingin kamu simpan di database. Kamu bisa melakukan beberapa hal:

1. **Ambil Daftar Kota**: Mendapatkan daftar kota yang sudah ada.
2. **Ambil Detail Kota**: Mengambil informasi lebih lanjut tentang kota tertentu.
3. **Tambah Kota Baru**: Menambahkan kota baru ke dalam daftar.
4. **Hapus Kota**: Menghapus kota dari daftar.

Dengan menggunakan aplikasi ini, setiap kali ada perubahan dalam daftar kota, aplikasi akan memperbarui tampilan agar informasi yang tampil selalu up-to-date.

Penjelasan Kode Tiap Baris

1. Mengimpor Library dan Membuat Context

```
import { useReducer } from "react";
import { createContext, useContext, useEffect, useState } from "react";
```

- `useReducer` digunakan untuk mengelola state yang lebih kompleks dibandingkan dengan `useState`, misalnya ketika ada banyak aksi yang mempengaruhi state.

- `createContext` digunakan untuk membuat **Context** yang akan menyediakan data kota ke komponen lain di aplikasi.
- `useContext` digunakan untuk mengakses data yang disediakan oleh `CitiesContext`.
- `useEffect` digunakan untuk melakukan side-effect, seperti pengambilan data kota dari server saat komponen pertama kali dimuat.

2. Mendefinisikan State Awal dan Reducer

```
const BASE_URL = "http://localhost:3000";
const CitiesContext = createContext();

const initialState = {
  cities: [],
  isLoading: false,
  currentCity: {},
  error: "",
};
```

- `BASE_URL` adalah URL tempat API kota berada.
- `CitiesContext` adalah konteks yang akan digunakan untuk menyediakan data kota kepada seluruh aplikasi.
- `initialState` adalah nilai awal state aplikasi, yang mencakup:
  - `cities`: Daftar kota yang akan diambil dari server.
  - `isLoading`: Status apakah data sedang dimuat.
  - `currentCity`: Kota yang sedang dilihat secara detail.
  - `error`: Pesan error jika terjadi kesalahan saat memuat data.

3. Fungsi `reducer` untuk Mengelola State

```
function reducer(state, action) {
  switch (action.type) {
    case "loading":
      return { ...state, isLoading: true };

    case "cities/loaded":
      return {
        ...state,
        isLoading: false,
        cities: action.payload,
      };

    case "city/loaded":
      return { ...state, isLoading: false, currentCity: action.payload };

    case "city/created":
      return {
        ...state,
        isLoading: false,
```

```
        cities: [...state.cities, action.payload],
      };

    case "city/deleted":
      return {
        ...state,
        isLoading: false,
        cities: state.cities.filter((city) => city.id !== action.payload),
        currentCity: {},
      };

    case "rejected":
      return {
        ...state,
        isLoading: false,
        error: action.payload,
      };

    default:
      throw new Error("Unknown action type");
  }
}
```

- `reducer` adalah fungsi yang menangani perubahan state berdasarkan **action** yang dikirimkan. Berdasarkan `action.type`, reducer akan mengubah state dengan cara yang sesuai:
  - `loading`: Mengatur status loading menjadi `true`.
  - `cities/loaded`: Menyimpan data kota yang diterima dari server ke dalam state dan mengubah status loading menjadi `false`.
  - `city/loaded`: Mengubah detail kota yang sedang dilihat.
  - `city/created`: Menambahkan kota baru ke dalam daftar kota.
  - `city/deleted`: Menghapus kota dari daftar.
  - `rejected`: Menyimpan pesan error jika ada kesalahan saat pengambilan data.

4. `CitiesProvider` untuk Menyediakan Context

```
function CitiesProvider({ children }) {
  const [{ cities, isLoading, currentCity, error }, dispatch] = useReducer(
    reducer,
    initialState
  );
```

- `useReducer` digunakan di sini untuk mengelola state aplikasi. Kita mendapatkan state yang terdiri dari daftar kota, status loading, kota yang sedang dilihat, dan pesan error, serta fungsi `dispatch` untuk memicu perubahan state berdasarkan aksi yang diberikan.

```
useEffect(function () {
  async function fetchCities() {
    dispatch({ type: "loading" });
```

```
    try {
      const res = await fetch(`${BASE_URL}/cities`);
      const data = await res.json();
      dispatch({ type: "cities/loaded", payload: data });
    } catch {
      dispatch({
        type: "rejected",
        payload: "There was an error loading cities.....",
      });
    }
  }
  fetchCities();
}, []);
```

- `useEffect` digunakan untuk mengambil data kota dari server saat aplikasi pertama kali dimuat. Fungsi `fetchCities` akan melakukan pengambilan data dan mengubah state berdasarkan hasilnya. Jika berhasil, action `cities/loaded` akan dipicu dengan data kota yang diterima, jika gagal, action `rejected` akan dipicu dengan pesan error.

```
async function getCity(id) {
  if (Number(id) === currentCity.id) return;

  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.....",
    });
  }
}
```

- `getCity` digunakan untuk mengambil detail kota berdasarkan **id**. Jika kota yang diminta sudah ada di state (`currentCity.id`), maka tidak akan ada perubahan.

```
async function createCity(newCity) {
  dispatch({ type: "loading" });
  try {
    const res = await fetch(`${BASE_URL}/cities`, {
      method: "POST",
      body: JSON.stringify(newCity),
      headers: {
        "Content-Type": "application/json",
      },
    });
    const data = await res.json();
```

```
    dispatch({ type: "city/created", payload: data });
  } catch {
    dispatch({
      type: "rejected",
      payload: "There was an error creating the city",
    });
  }
}
```

- `createCity` digunakan untuk membuat kota baru dengan mengirimkan data kota ke server melalui `POST` request. Setelah kota baru berhasil dibuat, data kota baru akan ditambahkan ke daftar kota di state.

```
async function deleteCity(id) {
  dispatch({ type: "loading" });
  try {
    await fetch(`${BASE_URL}/cities/${id}`, {
      method: "DELETE",
    });
    dispatch({ type: "city/deleted", payload: id });
  } catch {
    dispatch({
      type: "rejected",
      payload: "There was an error deleting the city",
    });
  }
}
```

- `deleteCity` digunakan untuk menghapus kota dari server menggunakan `DELETE` request. Setelah kota berhasil dihapus, kota tersebut akan dihapus dari daftar di state.

5. Menyediakan Context dengan `CitiesContext.Provider`

```
  return (
    <CitiesContext.Provider
      value={{
        cities,
        isLoading,
        currentCity,
        error,
        getCity,
        createCity,
        deleteCity,
      }}
    >
      {children}
    </CitiesContext.Provider>
  );
}
```

- `CitiesContext.Provider` menyediakan data yang terkait dengan kota (seperti daftar kota, status loading, detail kota, dll.) ke seluruh komponen yang membutuhkan data tersebut dalam aplikasi.

6. Hook `useCities` untuk Mengakses Data

```
function useCities() {
  const context = useContext(CitiesContext);
  if (context === undefined)
    throw new Error("CitiesContext was used outside the CitiesProvider");
  return context;
}
```

- `useCities` adalah hook

kustom untuk mengakses data yang disediakan oleh `CitiesContext`. Jika digunakan di luar komponen yang dibungkus oleh `CitiesProvider`, maka akan menghasilkan error.

Kesimpulan

Aplikasi ini mengelola data kota menggunakan **React Context** dan **useReducer**. Setiap perubahan data kota (seperti mengambil, menambah, atau menghapus kota) diproses melalui reducer untuk memperbarui state aplikasi. **useContext** memungkinkan data ini diakses di seluruh aplikasi, dan **useEffect** digunakan untuk pengambilan data kota dari server saat aplikasi pertama kali dimuat.

Dengan pendekatan ini, kamu dapat dengan mudah mengelola dan menampilkan data kota dalam aplikasi React.

Panduan Pemahaman Kode React untuk Autentikasi Pengguna

Pengertian Umum

Aplikasi ini menggunakan **React Context** dan **useReducer** untuk mengelola status autentikasi pengguna. Dengan menggunakan **Context API**, aplikasi dapat menyimpan dan mengakses informasi terkait status login pengguna (apakah pengguna terautentikasi atau tidak) di seluruh komponen aplikasi. **useReducer** digunakan untuk mengelola perubahan status autentikasi, seperti login dan logout.

Cara Berpikir React

Pada React, kita membagi aplikasi menjadi komponen-komponen kecil yang saling berinteraksi. Setiap komponen bisa memiliki state, dan React akan secara otomatis memperbarui tampilan ketika state berubah.

Di sini, kita menggunakan **Context API** untuk membagikan status autentikasi (apakah pengguna terautentikasi atau tidak) ke seluruh aplikasi, sehingga kita tidak perlu melewatkan props ke setiap komponen yang membutuhkannya.

Analogi Sederhana

Bayangkan kamu memiliki aplikasi yang mengharuskan pengguna untuk login agar dapat mengakses halaman tertentu.

- **Login**: Jika pengguna berhasil memasukkan email dan password yang benar, mereka akan masuk ke aplikasi dan statusnya berubah menjadi terautentikasi.
- **Logout**: Jika pengguna ingin keluar, kita akan menghapus data autentikasi dan statusnya berubah menjadi tidak terautentikasi.

Di dalam kode ini, kita menggunakan **React Context** untuk menyimpan status autentikasi ini, dan **useReducer** untuk mengelola perubahan status berdasarkan aksi yang dilakukan (login atau logout).

Penjelasan Kode Tiap Baris

1. Membuat Context dan State Awal

```
const AuthContext = createContext();
```

- `createContext` digunakan untuk membuat konteks yang akan menyimpan informasi autentikasi. Context ini bisa diakses oleh komponen manapun dalam aplikasi yang membutuhkannya.

```
const initialState = {
  user: null,
  isAuthenticated: false,
};
```

- `initialState` adalah nilai awal state yang menyimpan:
  - `user`: Informasi pengguna yang terautentikasi. Nilainya `null` jika belum ada pengguna yang login.
  - `isAuthenticated`: Status apakah pengguna terautentikasi atau tidak. Nilainya `false` jika belum terautentikasi.

2. Reducer untuk Mengelola Aksi

```
function reducer(state, action) {
  switch (action.type) {
    case "login":
      return { ...state, user: action.payload, isAuthenticated: true };
```

```
    case "logout":
      return {
        ...state,
        user: null,
        isAuthenticated: false,
      };

    default:
      throw new Error("Unknown action");
  }
}
```

- `reducer` adalah fungsi yang mengelola perubahan state berdasarkan **action** yang diterima.
  - **"login"**: Ketika pengguna berhasil login, kita menyimpan informasi pengguna di state (`user`) dan mengubah status autentikasi menjadi `true`.
  - **"logout"**: Ketika pengguna logout, kita menghapus informasi pengguna (`user`) dan mengubah status autentikasi menjadi `false`.

3. Menyediakan Data Autentikasi dengan `AuthProvider`

```
function AuthProvider({ children }) {
  const [{ user, isAuthenticated }, dispatch] = useReducer(reducer,
initialState);
```

- `useReducer` digunakan di sini untuk mengelola state autentikasi (apakah pengguna terautentikasi atau tidak). `dispatch` digunakan untuk mengirimkan aksi yang mengubah state.

```
function login(email, password) {
  if (email === FAKE_USER.email && password === FAKE_USER.password)
    dispatch({ type: "login", payload: FAKE_USER });
}
```

- `login` adalah fungsi untuk melakukan proses login. Jika email dan password yang dimasukkan cocok dengan data pengguna palsu (`FAKE_USER`), maka fungsi ini akan mengirimkan aksi dengan tipe **"login"** untuk memperbarui state dan menyimpan data pengguna ke dalam state.

```
function logout() {
  dispatch({ type: "logout" });
}
```

- `logout` adalah fungsi untuk keluar dari aplikasi, yang akan mengirimkan aksi **"logout"** untuk menghapus data pengguna dan mengubah status autentikasi menjadi `false`.

```
  return (
    <AuthContext.Provider value={{ user, isAuthenticated, login, logout }}>
      {children}
    </AuthContext.Provider>
  );
}
```

- `AuthContext.Provider` digunakan untuk menyediakan data autentikasi ke seluruh komponen anak (komponen yang dibungkus oleh `AuthProvider`). Data yang disediakan meliputi:
  - `user`: Informasi pengguna yang terautentikasi.
  - `isAuthenticated`: Status autentikasi.
  - `login` dan `logout`: Fungsi untuk melakukan login dan logout.

4. Hook Kustom `useAuth` untuk Mengakses Data Autentikasi

```
function useAuth() {
  const context = useContext(AuthContext);
  if (context === undefined)
    throw new Error("Auth context was used outside the AuthProvider");
  return context;
}
```

- `useAuth` adalah hook kustom untuk mengakses data autentikasi dari `AuthContext`. Jika hook ini digunakan di luar komponen yang dibungkus oleh `AuthProvider`, maka akan menghasilkan error, karena data autentikasi hanya tersedia di dalam `AuthProvider`.

Kesimpulan

Aplikasi ini menggunakan **React Context** untuk menyediakan status autentikasi pengguna secara global ke seluruh aplikasi, dan **useReducer** untuk mengelola perubahan state autentikasi (login dan logout). Dengan pendekatan ini, kamu bisa mengakses status login dan informasi pengguna di mana saja dalam aplikasi menggunakan hook `useAuth`.

Fitur utama:

- `AuthProvider`: Menyediakan state autentikasi (pengguna dan status login) ke seluruh aplikasi.
- `useAuth`: Hook untuk mengakses data autentikasi di dalam komponen.
- `login` **dan** `logout`: Fungsi untuk mengelola proses autentikasi.

Dengan menggunakan konsep ini, kamu dapat membuat aplikasi dengan manajemen autentikasi yang lebih terorganisir dan terpisah dari logika komponen lain.