

# ReactJS Dasar

• • •

Eko Kurniawan Khannedy

# Eko Kurniawan Khannedy

- Technical architect at one of the biggest ecommerce company in Indonesia
- 14+ years experiences
- [www.programmerzamannow.com](http://www.programmerzamannow.com)
- [youtube.com/c/ProgrammerZamanNow](https://youtube.com/c/ProgrammerZamanNow)



# Eko Kurniawan Khannedy

- Telegram : [@khannedy](https://t.me/khannedy)
- Linkedin : <https://www.linkedin.com/company/programmer-zaman-now/>
- Facebook : [fb.com/ProgrammerZamanNow](https://www.facebook.com/ProgrammerZamanNow)
- Instagram : [instagram.com/programmerzamannow](https://www.instagram.com/programmerzamannow)
- Youtube : [youtube.com/c/ProgrammerZamanNow](https://www.youtube.com/c/ProgrammerZamanNow)
- Telegram Channel : [t.me/ProgrammerZamanNow](https://t.me/ProgrammerZamanNow)
- Tiktok : [https://tiktok.com/@programmerzamannow](https://www.tiktok.com/@programmerzamannow)
- Email : echo.khannedy@gmail.com

# Sebelum Belajar

- Kelas JavaScript
- Kelas NodeJS
- Kelas Vite
- Kelas TypeScript (optional)

# Pengenalan

# Kenapa Butuh Framework?

- Mungkin menjadi pertanyaan, kenapa kita butuh framework untuk membuat web Frontend?
- Salah satu alasannya adalah agar ada standarisasi saat membuat project terutama ketika bekerja dalam tim
- Tanpa adanya Framework, maka setiap orang akan membuat kode dengan gaya masing-masing

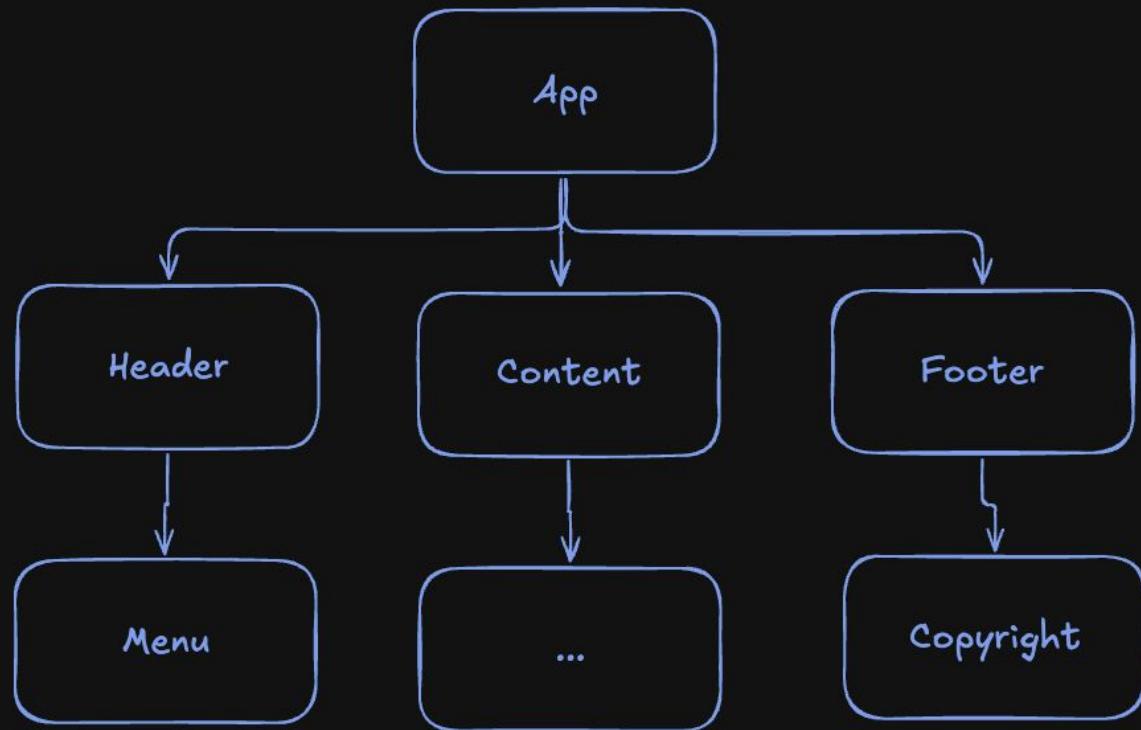
# Sejarah ReactJS

- FaxJS mulai dikembangkan di Facebook sekitar tahun 2010 untuk menangani masalah update halaman feed / timeline di Facebook tanpa harus refresh
- Tahun 2011, FaxJS diintegrasikan dengan XHP (pengembangan PHP di Facebook), yang akhirnya dinamai ReactJS
- Tahun 2013, Facebook merilis ReactJS ke umum sebagai OpenSource Tool pada saat JavaScript Conference
- Saat ini, ReactJS menjadi salah satu Frontend Framework yang paling populer, dan orang banyak memanggilnya dengan React
- <https://github.com/facebook/react>
- <https://react.dev/>

# Component

- Saat belajar React, kita harus terbiasa dengan istilah Component
- Component adalah kumpulan kode yang bisa digunakan secara independen
- Component bisa berisikan satu atau lebih HTML Element, kode JavaScript dan CSS
- Tidak ada aturan harus seberapa kecil atau besar sebuah Component
- Anggap saja seperti membuat Function, kita bisa membuat Function yang besar, atau kecil, karena tujuan Function adalah agar bisa digunakan secara berulang-ulang, begitu juga tujuan dari Component
- Struktur Component mirip seperti DOM (Document Object Model), dimana kita bisa membuat Component di dalam Component lain

# Struktur Component



# JSX

- React menggunakan JSX ketika membuat Component
- JSX (JavaScript XML atau JavaScript Syntax Extension), adalah kombinasi kode JavaScript dan XML (HTML), dimana kita bisa membuat Component dengan mudah menggunakan kode XML dan JavaScript dalam satu file
- JSX diperkenalkan oleh Facebook di React, namun saat ini JSX sudah banyak diadopsi oleh banyak Framework JavaScript lainnya

# Membuat Project

# Membuat Project

- Untuk membuat project React, kita akan menggunakan Vite sebagai build tool nya
- Kita bisa membuat project React menggunakan beberapa perintah
- Untuk membuat project React menggunakan JavaScript :  
`npm create vite@latest nama-aplikasi -- --template react`
- Untuk membuat project React menggunakan TypeScript :  
`npm create vite@latest nama-aplikasi -- --template react-ts`
- <https://vite.dev/guide/#scaffolding-your-first-vite-project>

# Kode : Membuat Project

```
cd my-react-app
→ REACT npm create vite@latest belajar-react-dasar -- --template react
Need to install the following packages:
create-vite@6.0.1
Ok to proceed? (y) y

Scaffolding project in /Users/khannedy/Developments/REACT/belajar-react-dasar...
```

Done. Now run:

```
cd belajar-react-dasar
npm install
npm run dev
```

# Struktur Folder

```
✓  └─ belajar-react-dasar ~/Developments/REACT/belajar-react-dasar
    >   └─ .idea
    >   └─ node_modules library root
    ✓  └─ public
        └─ vite.svg
    ✓  └─ src
        >   └─ assets
            └─ App.css
            └─ App.jsx
            └─ index.css
            └─ main.jsx
        └─ .gitignore
        └─ eslint.config.js
    └─ index.html
    └─ package.json
    └─ package-lock.json
    └─ README.md
    └─ vite.config.js
```

# Update versi ReactJS

- Pastikan menggunakan ReactJS yang terbaru
- <https://www.npmjs.com/package/react>
- <https://www.npmjs.com/package/react-dom>

# Hello World

# Hello World

- Seperti biasa, hal yang biasa kita buat ketika belajar adalah membuat aplikasi Hello World
- Sekarang kita akan buat halaman Hello World menggunakan React

# Membuat Component

- Biasanya, Component akan dibuat dalam satu file JSX dengan nama sesuai dengan nama komponen nya
- Misal kita akan membuat Component bernama HelloWorld
- Maka kita bisa membuat file HelloWorld.jsx
- Selanjutnya, kita perlu membuat default function dengan nama Component nya, yaitu HelloWorld
- Return dari function tersebut adalah element UI yang akan ditampilkan

# Kode : hello-world/HelloWorld.jsx

```
>HelloWorld.jsx ×  
1  function HelloWorld() {  
2      return (  
3          <div>  
4              <h1>Hello World</h1>  
5              <p>Selamat belajar ReactJS</p>  
6          </div>  
7      );  
8  }  
9  
10 export default HelloWorld;  
11
```

# Menampilkan Component

- Untuk menampilkan Component, diperlukan instance dari React Root
- Kita bisa membuat React Root menggunakan method `createRoot(element)`
- <https://react.dev/reference/react-dom/client/createRoot>
- Selanjutnya untuk menampilkan Component di React Root, kita bisa gunakan method `render(component)`
- Untuk membantu mencari masalah saat development, React menyediakan component `StrictMode`, kita bisa menggunakan `StrictMode` untuk menampilkan komponen di Root
- <https://react.dev/reference/react/StrictMode>

# Kode : hello-world/main.jsx

main.jsx ×

```
1 import {createRoot} from "react-dom/client";
2 import HelloWorld from "./HelloWorld.jsx";
3 import {StrictMode} from "react";
4
5 createRoot(document.getElementById("root"))
6     .render(
7         <StrictMode>
8             <HelloWorld/>
9         </StrictMode>
10    );
11
```

# Kode : hello-world.html

```
<> hello-world.html <
1  <!doctype html>
2  <html lang="en">
3  <head>
4      <meta charset="UTF-8"/>
5      <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
6      <title>Hello React</title>
7  </head>
8  <body>
9  <div id="root"></div>
10 <script type="module" src="/src/hello-world/main.jsx"></script>
11 </body>
12 </html>
13
```

# Kode : Vite Config

JS vite.config.js ×

```
1 import {defineConfig} from 'vite'
2 import react from '@vitejs/plugin-react'
3
4 // https://vite.dev/config/
5 export default defineConfig({
6     plugins: [react()],
7     build: {
8         rollupOptions: {
9             input: {
10                 index: "index.html",
11                 hello_world: "hello-world.html"
12             }
13         }
14     }
15 })
```

# Component

# Component

- React Component mendukung semua elemen HTML dan SVG
- Jadi kita tidak perlu khawatir ketika membuat Component, karena semua elemen pasti didukung
- <https://react.dev/reference/react-dom/components>

# Multiple Component

- Sebelumnya kita membuat satu file JSX untuk satu Component
- Walaupun itu adalah praktek yang baik, tapi bukan berarti itu wajib dilakukan
- Component sebenarnya hanyalah sebuah function yang mengembalikan React Element, oleh karena itu jika kita ingin membuat Component, kita hanya cukup membuat function yang mengembalikan React Element

# Kode : hello-world/HelloWorld.jsx

>HelloWorld.jsx ×

```
1 export function HeaderHelloWorld() {  
2     return (  
3         <h1>Hello World</h1>  
4     );  
5 }  
6  
7 export function ParagraphHelloWorld() {  
8     return (  
9         <p>Selamat belajar ReactJS</p>  
10    );  
11 }  
12
```

```
function HelloWorld() {  
    return (  
        <div>  
            <HeaderHelloWorld/>  
            <ParagraphHelloWorld/>  
        </div>  
    );  
}  
  
export default HelloWorld;
```

# JSX

# Kenapa JSX

- Web dibuat menggunakan HTML, CSS dan JavaScript. Biasanya kita akan menyimpan konten di HTML, desain di CSS dan logika aplikasi di JavaScript. Dan biasanya kita akan simpan di file-file yang terpisah
- Tapi saat ini, Web sudah lebih interaktif, seringnya konten HTML ditampilkan berdasarkan logika aplikasi di JavaScript
- Oleh karena itu, React melakukan logika aplikasi dan membuat konten di satu tempat yang sama, JSX

# Mengubah HTML ke JSX

- Saat mengubah kode HTML ke JSX, kita tidak bisa lakukan semudah Copy Paste kodennya
- JSX memiliki aturan yang lebih ketat dibanding HTML, contohnya saat menggunakan tag element, wajib menggunakan tag tutup
- Misal kita tidak bisa menggunakan  
`<img src="">`
- Kita harus gunakan tag tutup ketika menggunakan JSX  
`<img src="" />`
- Beberapa attribute di tag element pun berbeda. Kita akan bahas secara bertahap

# Aturan JSX

- Component hanya boleh mengembalikan satu element, jika kita ingin mengembalikan beberapa element, kita harus bungkung dalam parent element
- Wajib menutup semua tag element
- Attribute menggunakan camelCase. Attribute di element JSX akan dikonversi ke variable JavaScript, oleh karena itu nama attribute harus mengikuti cara pembuatan nama variable di JavaScript, yaitu tidak bisa menggunakan - (strip). Karena keterbatasan ini, kebanyakan attribute di element JSX akan menggunakan camelCase, contoh className (bukan class-name)

# JSX Converter

- Jika misal kita sudah punya kode HTML, dan ingin mengkonversi ke JSX
- Disarankan menggunakan Converter sehingga kita tidak perlu lakukan secara manual lagi
- <https://transform.tools/html-to-jsx>

# JavaScript di JSX

# JavaScript di JSX

- Kadang-kadang, kita kasus dimana kita ingin mengakses kode JavaScript di JSX
- Pada kasus ini, kita bisa menggunakan kurung kurawal untuk mengakses kode JavaScript di JSX

# Kode : JavaScript di JSX

>HelloWorld.jsx ×

```
1 export function HeaderHelloWorld() {
2     const text = 'Hello World';
3     return (
4         <h1>{text.toUpperCase()}</h1>
5     );
6 }
7
8 export function ParagraphHelloWorld() {
9     const text = 'Selamat Belajar ReactJS';
10    return (
11        <p>{text.toLowerCase()}</p>
12    );
13 }
14
```

# Kurung Kurawal di JSX

- Penggunaan kurung kurawal di JSX hanya bisa dilakukan pada dua lokasi
- Sebagai teks, seperti pada contoh sebelumnya
- Sebagai nilai atribut pada tag element, misal <img src={location} />

# Kurung Kurawal Double pada JSX

- Pada beberapa attribute, contohnya style. Kita bisa menggunakan JavaScript Object sebagai parameter
- Oleh karena itu, kadang sekilas terlihat seperti kurung kurawal double, padahal sebenarnya hanya satu kurung kurawal, dan datanya adalah JavaScript Object

# Kode : Kurung Kurawal Double

```
✓ export function HeaderHelloWorld() {  
    const text = 'Hello World';  
    return (  
        <h1 style={{  
            color: "red",  
            backgroundColor: "aqua"  
        }}>{text.toUpperCase()}</h1>  
    );  
}
```

```
✓ export function ParagraphHelloWorld() {  
    const text = 'Selamat Belajar ReactJS';  
    const style = {  
        color: "blue",  
        backgroundColor: "yellow"  
    }  
    return (  
        <p style={style}>{text.toLowerCase()}</p>  
    );  
}
```

# Props

# Props

- React Component menggunakan Props untuk berkomunikasi
- Parent Component bisa mengirim informasi ke Child Component dengan menggunakan Props
- Props itu mirip seperti attribute di HTML Element, tapi kita bisa mengirim nilai JavaScript seperti object, array, function atau yang lainnya

# Menambah Props

- Untuk menambahkan Props pada Component, kita hanya perlu menambahkan parameter object pada function di Component
- Parameter Props tersebut merupakan JavaScript Object, sehingga kita bisa mengakses detail attribute yang dikirim dari parent melalui Props

# Kode : Props

>HelloWorld.jsx ×

```
1 export function HeaderHelloWorld(props) {  
2     return (  
3         <h1 style={{  
4             color: "red",  
5             backgroundColor: "aqua"  
6         }}>{props.text.toUpperCase()}</h1>  
7     );  
8 }
```

# Destructuring Props

- Salah satu yang biasa dilakukan oleh programmer React adalah, melakukan destructuring parameter pada Props
- Hal ini untuk mempermudah ketika membaca, sehingga kita tahu attribute apa yang tersedia pada Component tersebut
- Karena Props adalah JavaScript Object, jadi kita juga bisa menambahkan fitur seperti Default Value pada Props

# Kode : Destructuring Props

```
❶  HelloWorld.jsx ×
  1 export function HeaderHelloWorld({text = "Ups, lupa kasih teks"}) {
  2     return (
  3         <h1 style={{
  4             color: "red",
  5             backgroundColor: "aqua"
  6         }}>{text.toUpperCase()}</h1>
  7     );
  8 }
  9 }
```

# Mengirim Props

- Untuk mengirim Props, kita bisa menggunakan atribut seperti HTML Element ketika menggunakan Component

# Kode : Mengirim Props

```
function HelloWorld() {  
    return (  
        <div>  
            <HeaderHelloWorld text="Hello World"/>  
            <ParagraphHelloWorld/>  
        </div>  
    );  
}  
  
export default HelloWorld;
```

# Spread Syntax

- Kadang-kadang, mengirim Props dari Parent Component ke Child Component sangat merepotkan
- Jika kita hanya ingin melakukan forward semua Props ke Child Component, kita bisa menggunakan Spread Syntax di JavaScript

# Kode : Spread Syntax

```
function HelloWorld() {  
  const props = {  
    text: "Hello World"  
  }  
  return (  
    <div>  
      <HeaderHelloWorld {...props}/>  
      <ParagraphHelloWorld/>  
    </div>  
  );  
}  
  
export default HelloWorld;
```

# Nested Component

# Nested Component

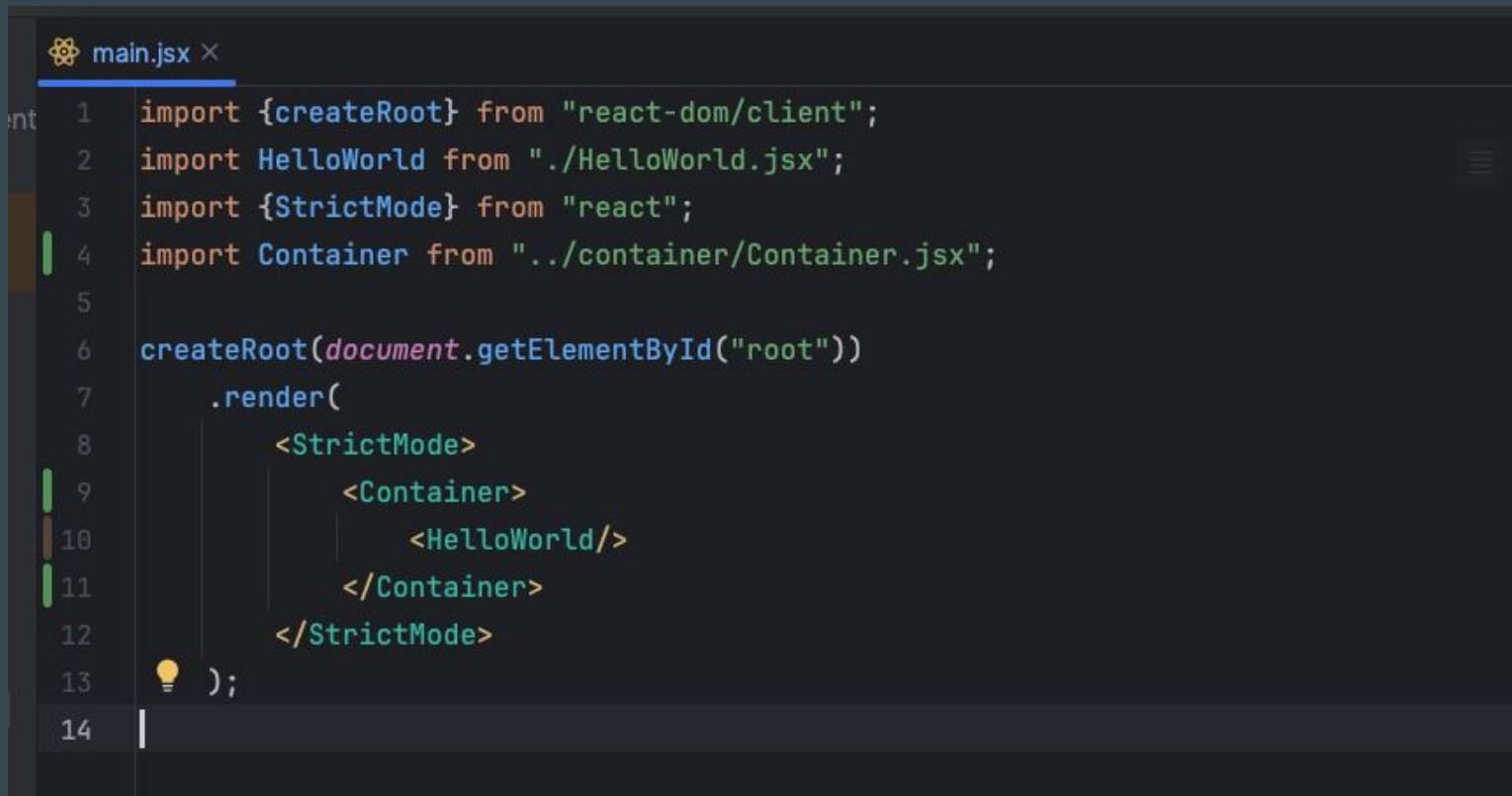
- JSX mendukung pembuatan Nested Component
- Hal ini memungkinkan kita bisa membuat Component yang di dalamnya bisa ditambahkan Component lain secara dinamis
- Agar component bisa memiliki Component lain didalamnya, kita bisa menggunakan attribute children pada Props
- Misal sekarang kita akan membuat Component bernama Container, dan didalamnya kita bisa berisi Component lainnya

# Kode : container/Container.jsx

A screenshot of a code editor window titled "Container.jsx". The code is a functional component named "Container" that takes an array of children as a prop. It renders a main content area with a heading and the provided children, followed by a footer containing a copyright notice.

```
1 export default function Container({children}) {
2     return (
3         <div>
4             <h1>Programmer Zaman Now</h1>
5             {children}
6             <footer>
7                 <p>© 2024 Programmer Zaman Now</p>
8             </footer>
9         </div>
10    )
11 }
12
```

# Kode : hello-world/main.jsx



A screenshot of a code editor showing the file `main.jsx`. The code is a React application that creates a root element, renders a `StrictMode` component containing a `Container` which in turn contains a `HelloWorld` component.

```
main.jsx
1 import {createRoot} from "react-dom/client";
2 import HelloWorld from "./HelloWorld.jsx";
3 import {StrictMode} from "react";
4 import Container from "../container/Container.jsx";
5
6 createRoot(document.getElementById("root"))
7   .render(
8     <StrictMode>
9       <Container>
10         <HelloWorld/>
11       </Container>
12     </StrictMode>
13   );
14
```

# Style

# Style

- Sampai saat ini, kita hanya baru membahas HTML dan JavaScript di JSX, bagaimana dengan CSS?
- CSS sendiri bukan bagian dari JSX, jika kita ingin membuat style CSS, maka kita perlu buat dalam file CSS, atau langsung di attribute style menggunakan kurung kurawal double yang pernah dicontohkan sebelumnya
- Namun sekarang kita akan coba dalam file CSS
- Menggunakan Style pada JSX, tidak menggunakan atribut class, melainkan className
- <https://react.dev/reference/react-dom/components/common#applying-css-styles>

# Kode : hello-world/HelloWorld.css



A screenshot of a code editor window titled "HelloWorld.css". The code is written in CSS and includes two styles: ".title" and ".content". The ".title" style sets the color to red and the background-color to aqua. The ".content" style sets the color to blue and the background-color to yellow. The code is numbered from 1 to 10 on the left side. A small orange lightbulb icon is located at the end of the ninth line, indicating a potential issue or suggestion.

```
nt 1 .title {  
2     color: red;  
3     background-color: aqua;  
4 }  
5  
6 .content {  
7     color: blue;  
8     background-color: yellow;  
9 }  
10
```

# Kode : hello-world/HelloWorld.jsx



A screenshot of a code editor window titled "HelloWorld.jsx". The code is written in JSX and contains two export functions: HeaderHelloWorld and ParagraphHelloWorld. The HeaderHelloWorld function takes a text prop and returns an H1 element with the text converted to uppercase. The ParagraphHelloWorld function returns a P element with the text converted to lowercase. The code also imports a CSS file named "HelloWorld.css".

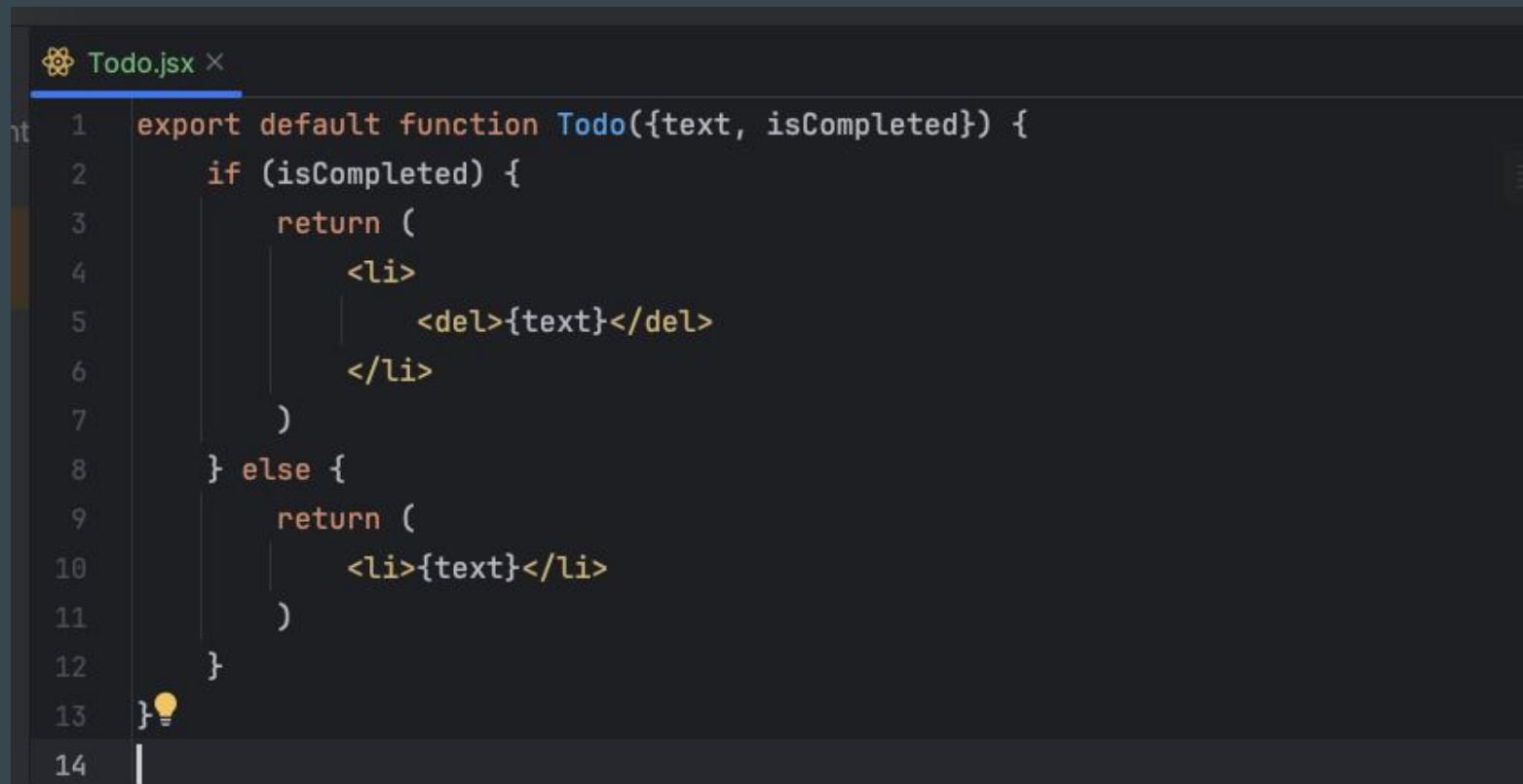
```
1 import './HelloWorld.css';
2
3 export function HeaderHelloWorld({text = "Ups, lupa kasih teks"}) {
4     return (
5         <h1 className="title">{text.toUpperCase()}</h1>
6     );
7 }
8
9 export function ParagraphHelloWorld() {
10     const text = 'Selamat Belajar ReactJS';
11     return (
12         <p className="content">{text.toLowerCase()}</p>
13     );
14 }
```

# Conditional

# Conditional

- Saat kita membuat halaman Web, sering sekali kita menampilkan tampilan berbeda pada kondisi tertentu
- Hal ini juga bisa dilakukan di JSX
- Kita bisa menambahkan kondisi menggunakan JavaScript, dan mengembalikan Component yang berbeda berdasarkan kondisi yang diinginkan
- Contoh kita akan membuat halaman TodoList, dan jika Todo nya susah selesai, kita akan coret element teks nya

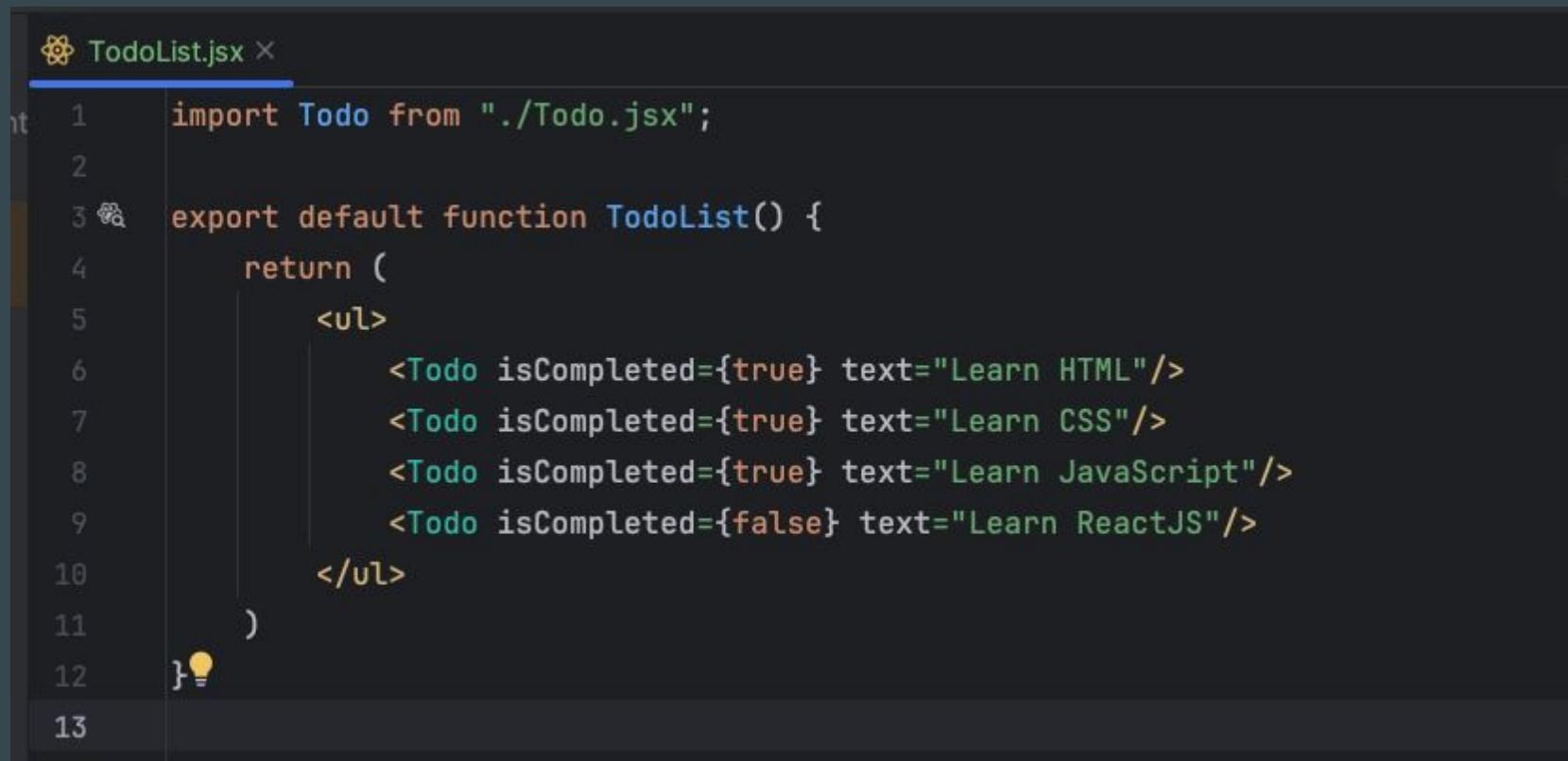
# Kode : todolist/Todo.jsx



A screenshot of a code editor window titled "Todo.jsx". The code is a functional component named "Todo" that takes two props: "text" and "isCompleted". If "isCompleted" is true, it returns an - element containing a  ~~text~~  element. If "isCompleted" is false, it returns an - element containing the text prop. The code is numbered from 1 to 14. A yellow lightbulb icon is visible at the bottom left of the editor area.

```
1  export default function Todo({text, isCompleted}) {
2      if (isCompleted) {
3          return (
4              <li>
5                  <del>{text}</del>
6              </li>
7          )
8      } else {
9          return (
10             <li>{text}</li>
11         )
12     }
13 }
```

# Kode : todolist/TodoList.jsx



A screenshot of a code editor showing a file named `TodoList.jsx`. The code defines a component `TodoList` that returns a list of four items, each represented by a `<Todo>` element. The first three items have the `isCompleted` prop set to `true`, while the fourth item has it set to `false`. The text for each item is a learning goal: "Learn HTML", "Learn CSS", "Learn JavaScript", and "Learn ReactJS". The code editor interface includes a tab bar at the top with the file name, a vertical scrollbar on the left, and a status bar at the bottom.

```
TodoList.jsx
1 import Todo from "./Todo.jsx";
2
3 export default function TodoList() {
4     return (
5         <ul>
6             <Todo isCompleted={true} text="Learn HTML"/>
7             <Todo isCompleted={true} text="Learn CSS"/>
8             <Todo isCompleted={true} text="Learn JavaScript"/>
9             <Todo isCompleted={false} text="Learn ReactJS"/>
10        </ul>
11    )
12 }
13
```

# Kode : hello-world/main.jsx

main.jsx ×

```
1 import {createRoot} from "react-dom/client";
2 import HelloWorld from "./HelloWorld.jsx";
3 import {StrictMode} from "react";
4 import Container from "../container/Container.jsx";
5 import TodoList from "../todolist/TodoList.jsx";
6
7 createRoot(document.getElementById("root"))
8   .render(
9     <StrictMode>
10       <Container>
11         <HelloWorld/>
12         <TodoList/>
13       </Container>
14     </StrictMode>
15   );
16
```

# Null Component

- Pada kasus tertentu, mungkin ada kondisi dimana kita ingin mengembalikan Component, atau tidak ingin mengembalikan Component apapun
- Kita bisa mengembalikan null untuk menandai bahwa tidak ada Component yang kita kembalikan

# Kode : todolist/Todo.jsx



A screenshot of a code editor showing the file `Todo.jsx`. The code defines a functional component `Todo` that takes three props: `text`, `isCompleted`, and `isDeleted` (with a default value of `false`). The component returns `null` if `isDeleted` is true, or a list item (`<li>`) containing a strikethrough text (`<del>{text}</del>`) if `isCompleted` is true. Otherwise, it returns a regular list item (`<li>{text}</li>`).

```
Todo.jsx ×
1  export default function Todo({text, isCompleted, isDeleted = false}) {
2      if (isDeleted) {
3          return null
4      } else if (isCompleted) {
5          return (
6              <li>
7                  <del>{text}</del>
8              </li>
9          )
10     } else {
11         return (
12             <li>{text}</li>
13         )
14     }
15 }
```

# Ternary Operator

- Kadang pada kasus yang lebih sederhana, kita bisa menggunakan Ternary Operator JavaScript di JSX

# Kode : Ternary Operator

Todo.jsx ×

```
1  export default function Todo({text, isCompleted, isDeleted = false}) {  
2      if (isDeleted) {  
3          return null  
4      } else {  
5          return (  
6              <li>  
7                  {isCompleted ? <del>{text}</del> : text}  
8              </li>  
9          )  
10     }  
11 }
```

# Logical AND

- Di beberapa kasus yang lebih sederhana, kadang di JSX sering memanfaatkan Logical AND di JavaScript
- Misal jika kondisi terpenuhi maka akan menampilkan, jika tidak maka tidak menampilkan

# Kode : Logical AND

```
Todo.jsx ×
1  export default function Todo({text, isCompleted, isDeleted = false}) {
2      if (isDeleted) {
3          return null
4      } else {
5          return (
6              <li>
7                  {text} {isCompleted && '✓'}
8              </li>
9          )
10     }
11 }
```



# Collection Component

# Collection Component

- Kita pasti akan sering menampilkan Component yang sama berkali-kali sesuai koleksi data. JSX sendiri tidak memiliki fitur perulangan
- Untuk menampilkan multiple Component, sama seperti Conditional, kita akan memanfaatkan JavaScript
- Kita bisa menggunakan JavaScript Array untuk mengubah data Array menjadi Component, misal menggunakan method map() pada Array
- Misal sekarang kita akan coba ubah data TodoList yang sebelumnya kita buat menjadi array

# Kode : todolist/TodoList.jsx

```
export default function TodoList() {  
  const data = [  
    {  
      text: "Learn HTML",  
      isCompleted: true  
    },  
    {  
      text: "Learn CSS",  
      isCompleted: true  
    },  
    {  
      text: "Learn JavaScript",  
      isCompleted: true  
    },  
    {  
      text: "Learn ReactJS",  
      isCompleted: false  
    }  
  ]  
}
```

```
const todos = data.map((todo) => (  
  <Todo {...todo}>  
>));  
return (  
  <ul>  
    {todos}  
  </ul>  
)  
}
```

# Component Key

- Jika kita perhatikan di Text Editor, mungkin kita akan melihat pesan peringatan : “Missing key prop for element in iterator”
- Saat membuat Collection Component, tiap Component diperlukan id (unique, string atau number) menggunakan attribute key
- Kenapa butuh Component Key? Hal ini agar React bisa mengenali Component ketika berubah, seperti posisinya diubah, dihapus atau ditambahkan pada Collection
- Biasanya, Component Key akan diambil dari data, sehingga lebih konsisten

# Kode : todolist/TodoList.jsx

```
export default function TodoList() {
  const data = [
    {
      id: 0,
      text: "Learn HTML",
      isCompleted: true
    },
    {
      id: 1,
      text: "Learn CSS",
      isCompleted: true
    },
    {
      id: 2,
      text: "Learn JavaScript",
      isCompleted: true
    },
    {
      id: 3,
      text: "Learn ReactJS",
      isCompleted: false
    }
  ]
  return (
    <ul>
      {data.map((todo) => (
        <Todo key={todo.id} {...todo}>/)
      ))}
    </ul>
  )
}
```

# Pure Component

# Pure Function

- Dalam pemrograman, kita mengenal yang namanya Pure Function
- [https://en.wikipedia.org/wiki/Pure\\_function](https://en.wikipedia.org/wiki/Pure_function)
- Sebuah Function kita sebut sebagai Pure Function jika memenuhi dua kriteria berikut :
- Function akan mengembalikan nilai yang sama untuk nilai parameter yang sama
- Function tidak memiliki efek samping, tidak ada perubahan pada variable non local (variable yang berada diluar function)

# Kode : Pure Function

```
// pure function
✓ export function double(num) {
    return num * 2;
}

// bukan pure function
let count = 0;
✓ export function increment() {
    count++;
    return count;
}
```

# Pure Component

- React mengasumsikan bahwa setiap Component yang kita buat adalah Pure Function.
- Ini berarti bahwa React Component yang kita buat harus selalu mengembalikan JSX yang sama dengan input yang sama
- Walaupun sebenarnya kita bisa saja membuat React Component yang tidak Pure, tapi sangat tidak disarankan, karena setiap memanggil Component dengan input sama bisa menghasilkan nilai yang tidak konsisten
- Kita akan coba membuat contoh React Component yang tidak Pure

# Kode : table/Row.jsx

Row.jsx ×

```
AC 1 let counter = 0;
2
3 export default function Row({text}) {
4     counter++;
5     return (
6         <tr>
7             <td>{counter}</td>
8             <td>{text}</td>
9         </tr>
10    )
11 }
12
```

# Kode : table/Table.jsx



A screenshot of a code editor showing a file named "Table.jsx". The code defines a functional component "Table" that returns a table with three rows containing the text "Satu", "Dua", and "Tiga". The code is numbered from 1 to 14. A yellow lightbulb icon is visible at the bottom left, indicating a potential issue or suggestion.

```
1 import Row from "./Row.jsx";
2
3 export default function Table() {
4     return (
5         <table border="1">
6             <tbody>
7                 <Row text="Satu"/>
8                 <Row text="Dua"/>
9                 <Row text="Tiga"/>
10            </tbody>
11        </table>
12    )
13 }
14
```

# Membuat Pure Component

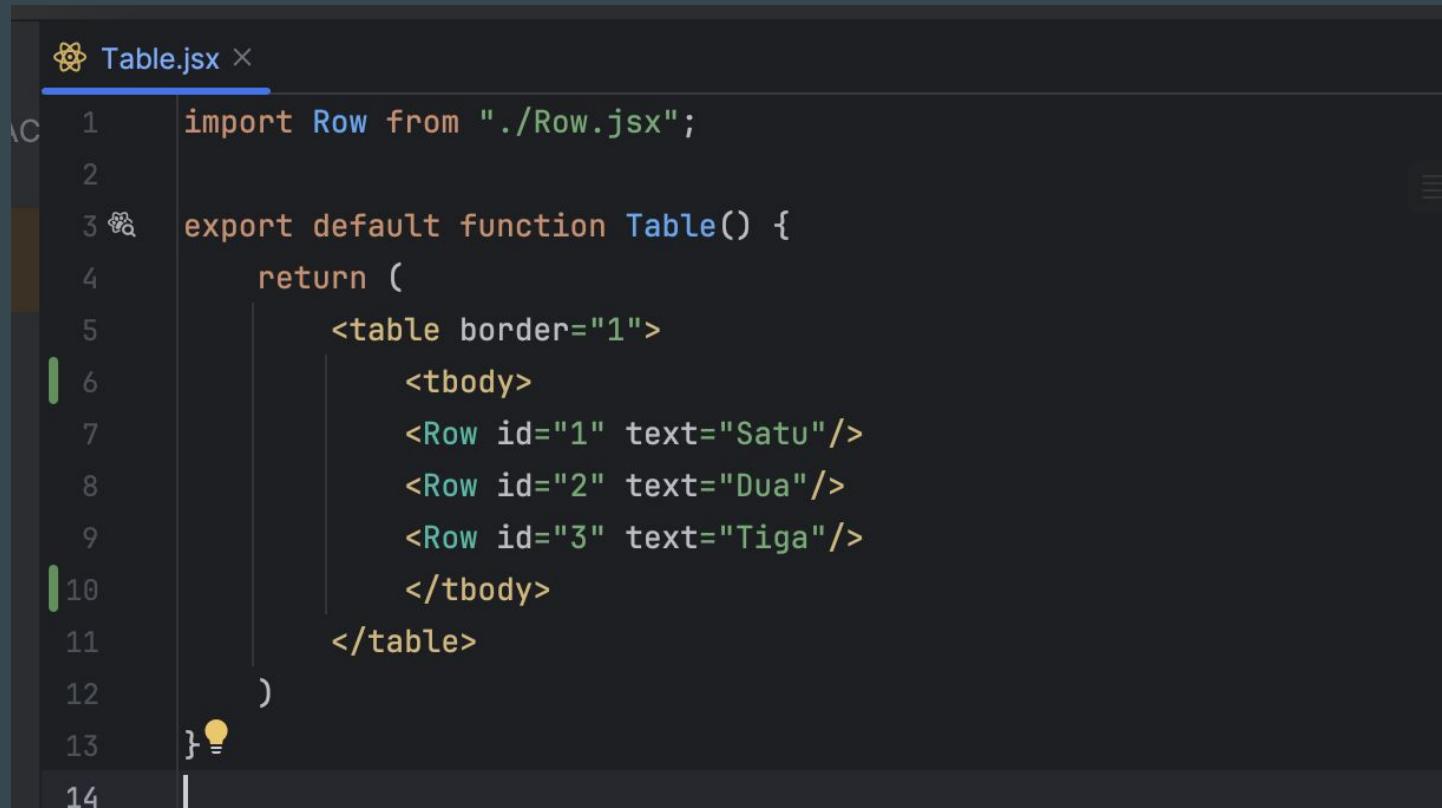
- Lantas bagaimana acara Component Row dan Table menjadi Pure Component?
- Kita harus menghapus efek samping dari Component, contohnya kita bisa pindahkan variable counter menjadi local variable di Table, dan gunakan Props sebagai counter nya

# Kode : table/Row.jsx

Row.jsx ×

```
1  export default function Row({id, text}) {  
2      return (  
3          <tr>  
4              <td>{id}</td>  
5              <td>{text}</td>  
6          </tr>  
7      )  
8  }  
9  |
```

# Kode : table/Table.jsx



A screenshot of a code editor window titled "Table.jsx". The code is a functional component named "Table" that returns a table with three rows. The rows are labeled "Satu", "Dua", and "Tiga". The code uses the "Row" component from "Row.jsx". The code editor has a dark theme with syntax highlighting. A yellow lightbulb icon is visible at the bottom left, indicating a potential issue or suggestion.

```
1 import Row from "./Row.jsx";
2
3 export default function Table() {
4     return (
5         <table border="1">
6             <tbody>
7                 <Row id="1" text="Satu"/>
8                 <Row id="2" text="Dua"/>
9                 <Row id="3" text="Tiga"/>
10            </tbody>
11        </table>
12    )
13 }
14 |
```

# Dimana bisa melakukan efek samping?

- React menyediakan tempat khusus jika kita ingin membuat Component yang bisa menghasilkan efek samping
- Efek samping dari Component biasanya ditempatkan di Event Handler, yaitu aksi yang terjadi ketika kita berinteraksi dengan Component
- Atau ketika misal Component tergantung dengan external system (misal API), maka React menyediakan function useEffect()
- <https://react.dev/reference/react/useEffect>
- Semua ini akan kita bahas di materinya masing-masing

# Event Handler

# Event Handler

- Seperti yang sudah kita tahu bahwa di HTML Element, kita bisa menambahkan Event Handler
- Sama juga dengan React Component, kita bisa menambahkan Event Handler pada Element di React Component
- Ada banyak sekali jenis Event Handler yang bisa kita tambahkan, kita bisa lihat di halaman Reference React Component
- <https://react.dev/reference/react-dom/components/common>

# Menambah Event Handler

- Untuk menambah Event Handler, biasanya kita akan tambahkan Function sebagai Handler nya
- Bisa dalam bentuk Anonymous Function, Arrow Function atau membuat Function terlebih dahulu di dalam scope Component nya
- Nama Function untuk Handler biasanya diawali dengan nama “handle” dan diikuti dengan jenis Event Handler, misal handleClick(), handleMouseEnter(), dan lain-lain
- Sekarang kita akan coba membuat Component untuk Alert Button

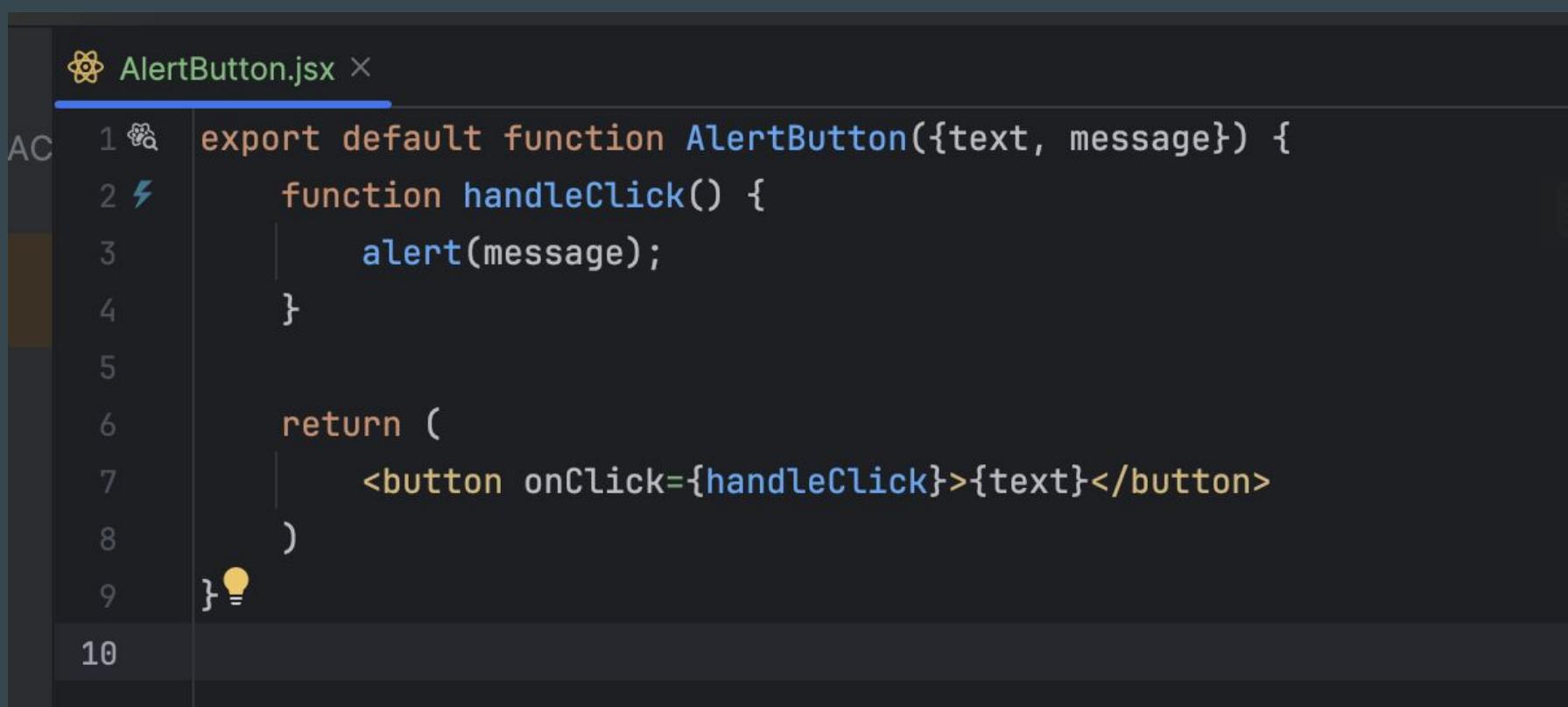
# Kode : button/AlertButton.jsx

```
AC AlertButton.jsx ×  
1 export default function AlertButton({text}) {  
2     function handleClick() {  
3         alert("Button clicked");  
4     }  
5  
6     return (  
7         <button onClick={handleClick}>{text}</button>  
8     )  
9 }  
10 |
```

# Membaca Props di Event Handler

- Salah satu keuntungan membuat Function Event Handler di dalam Component adalah, kita bisa membaca Props yang digunakan oleh Component tersebut

# Kode : button/AlertButton.jsx



A screenshot of a code editor showing a file named `AlertButton.jsx`. The code defines a functional component `AlertButton` that takes `text` and `message` props. It contains a local function `handleClick` that alerts the `message`. The component returns a button that triggers this function when clicked.

```
AC 1  export default function AlertButton({text, message}) {  
  2    function handleClick() {  
    3       alert(message);  
  4   }  
  5  
  6   return (  
  7     <button onClick={handleClick}>{text}</button>  
  8   )  
  9 }   
10
```

# Event Handler sebagai Props

- Seperti yang di materi Props dibahas, Props sebenarnya adalah JavaScript Object
- Kita tahu bahwa JavaScript Object bisa memiliki attribute dengan tipe Function
- Oleh karena itu, kita juga bisa membuat Event Handler di Props
- Saat membuat attribute di Props yang berisikan Event Handler, biasanya nama attribute nya akan diawali dengan “on”, misal “onSmash”, “onHit” dan lain-lain

# Kode : button/MyButton.jsx

MyButton.jsx ×

```
1 export default function MyButton({text, onSmash}) {  
2     return (  
3         <button onClick={onSmash}>{text}</button>  
4     )  
5 }  
6 |
```

# Kode : hello-world/main.jsx

```
createRoot(document.getElementById("root"))
  .render(
    <StrictMode>
      <Container>
        <HelloWorld/>
        <TodoList/>
        <Table/>
        <AlertButton text="Click me" message="You click me"/>
        <AlertButton text="Click you" message="You click you"/>

        <MyButton text="Smash me" onSmash={() => alert("You smash me")}/>
        <MyButton text="Hit me" onSmash={() => alert("You hit me")}/>
      </Container>
    </StrictMode>
  );

```

# Event Object

# Event Object

- Saat kita membuat Event Handler Function, kita bisa menambahkan Event Object sebagai parameter di Function tersebut
- React Event Object kompatibel dengan standard DOM Event Object
- <https://react.dev/reference/react-dom/components/common#react-event-object>
- <https://developer.mozilla.org/en-US/docs/Web/API/Event>

# Kode : button/AlertButton.jsx



A screenshot of a code editor showing a file named "AlertButton.jsx". The code defines a functional component "AlertButton" that takes "text" and "message" props. It logs the event to the console and then displays an alert with the provided message. The code editor interface includes tabs, a status bar, and a sidebar.

```
AlertButton.jsx ×
C 1 export default function AlertButton({text, message}) {
  2   function handleClick(e) {
  3     console.info(e);
  4     alert(message);
  5   }
  6
  7   return (
  8     <button onClick={handleClick}>{text}</button>
  9   )
10 }💡
11 |
```

# Event Propagation

# Event Propagation

- Event di React Component akan selalu disebarluaskan ke Component yang ada diatasnya (Event Propagation)
- Misal kita memiliki Div dengan Event onClick, lalu didalamnya kita memiliki Button dengan Event onClick
- Ketika Button diklik, maka onClick di Button akan dipicu dan selanjutnya onClick di Div juga akan dipicu
- Kadang mungkin kita tidak ingin hal itu terjadi, maka kita bisa hentikan proses Event Propagation tersebut menggunakan method stopPropagation()
- <https://developer.mozilla.org/en-US/docs/Web/API/Event/stopPropagation>

# Kode : button/Toolbar.jsx

```
Toolbar.jsx ×  
1 export default function Toolbar({onClick}) {  
2     return (  
3         <div onClick={onClick}>  
4             <button onClick={onClick}>First</button>  
5             <button onClick={onClick}>Second</button>  
6         </div>  
7     )  
8 }💡  
9
```

# Kode : hello-world/main.jsx

```
<MyButton text="Smash me" onSmash={() => alert("You smash me")}/>
<MyButton text="Hit me" onSmash={() => alert("You hit me")}/>

<Toolbar onClick={(e) => {
    e.stopPropagation();
    alert("You click toolbar");
}}/>
</Container>
</StrictMode>
```

# Prevent Default

- Selain menghentikan Event Propagation, hal yang biasa kita lakukan ketika membuat Event Handler adalah menghentikan default action menggunakan preventDefault()
- <https://developer.mozilla.org/en-US/docs/Web/API/Event/preventDefault>
- Misal kita membuat Form, ketika di dipicu Event onClick, kita ingin hentikan default action Form Submit

# Kode : form/SearchForm.jsx

SearchForm.jsx ×

```
AC 1 export default function SearchForm() {  
2     return (  
3         <form>  
4             <input type="text"/>  
5             <button onClick={(e) => {  
6                 e.preventDefault();  
7                 alert("You search");  
8             }}>Search  
9             </button>  
10        </form>  
11    )  
12}
```

# Side Effect

# Side Effect

- Apakah Component boleh memiliki Side Effect (efek samping)?
- Tentu saja boleh, namun biasanya Side Effect terjadi dikarenakan adanya interaksi dari pengguna melalui Event Handler
- Misal, kita akan membuat Form Say Hello, dimana ketika Button di klik, kita ingin menampilkan Hello + nama pada Text

# Kode : form/SayHelloForm.jsx

SayHelloForm.jsx ×

```
1  export default function SayHelloForm() {
2      return (
3          <div>
4              <form>
5                  <input id="input_name"/>
6                  <button onClick={
7                      (e) => {
8                          e.preventDefault();
9                          const name = document.getElementById("input_name").value;
10                         document.getElementById("text_hello").innerText = `Hello ${name}`;
11                     }
12                     }>Say Hello
13                     </button>
14                 </form>
15                 <h1 id="text_hello">Hello World</h1>
16             </div>
17         )
18     }
```

# DOM Manipulation

- Kode sebelumnya, kita menggunakan DOM Manipulation untuk mengubah Component di React
- Hal ini sebenarnya tidak terlalu direkomendasikan, terutama jika misal data yang memicu perubahan Element di Component bersumber dari berbagai Event Handler
- Oleh karena itu, hal yang direkomendasikan adalah menggunakan State, yang akan kita bahas di materi selanjutnya

# Hooks

# Hooks

- Hooks adalah fitur di React yang bisa digunakan di Component
- Ada banyak sekali fitur yang bisa kita gunakan di React Hooks, dan kita akan bahas secara bertahap
- <https://react.dev/reference/react/hooks>

# State

# State

- Component kadang perlu untuk berubah dikarenakan interaksi yang dilakukan pengguna
- Misal, input di klik bisa menaikkan data counter. Tombol next bisa mengubah gambar banner yang sedang muncul, dan lain-lain
- Component harus bisa mengingat nilai saat ini, seperti counter saat ini, gambar saat ini, dan lain-lain
- Di React, memori spesific di Component disebut State

# State Menggunakan Local Variable Biasa

- Apakah local variabel biasa di Component bisa digunakan untuk State? Sayangnya hal ini tidak bisa dilakukan
- Ketika React melakukan render Component untuk yang kedua kali dan seterusnya, maka semua kode Component akan dieksekusi ulang, oleh karena itu local variable akan kembali ke nilai awal
- Perubahan di local variable juga, tidak akan memicu render ulang Component

# Kode : form/Counter.jsx (Contoh Salah)



A screenshot of a code editor window titled "Counter.jsx". The code is a functional component named "Counter" that contains a button to increment a counter and log its value to the console.

```
1 export default function Counter() {
2     let counter = 0;
3     return (
4         <div>
5             <button onClick={
6                 () => {
7                     counter++;
8                     console.log(counter);
9                 }
10            }>Increment
11            </button>
12
13            <h1>Counter : {counter}</h1>
14        </div>
15    )
}
```

# useState

- Untuk membuat State, kita bisa menggunakan function useState(initial)
- <https://react.dev/reference/react/useState>
- Function useState akan mengembalikan array dengan dua nilai, pertama adalah State-nya, dan kedua ada function untuk mengubah value di State tersebut
- Component yang menggunakan State tersebut, secara otomatis akan di render ulang

# Kode : form/Counter.jsx

Counter.jsx ×

```
1 import {useState} from "react";
2
3 export default function Counter() {
4     let [counter, setCounter] = useState(0);
5     return (
6         <div>
7             <button onClick={
8                 () => {
9                     setCounter(counter + 1);
10                    console.log(counter);
11                }
12            }>Increment
13            </button>
14
15            <h1>Counter : {counter}</h1>
16        </div>
17    )
}
```

# State Terisolasi dan Private

- State merupakan data yang terisolasi dan private secara local terhadap Component yang menggunakan
- Artinya, jika kita me-render Component yang sama berkali-kali, maka State dari tiap Component tersebut akan terpisah satu sama lain

# Kode : hello-world/main.jsx

```
1 // src/App.js
2
3     <SayHelloForm/>
4
5     <Counter/>
6     <Counter/>
7     </Container>
8
9 </StrictMode>
0 );
```

# Render

# Render

- Sebelum Component yang kita buat ditampilkan di layar, Component harus di render oleh React
- Oleh karena itu, kita perlu tahu bagaimana cara kerja proses React ini ketika menampilkan Component yang kita buat
- Terdapat 3 tahapan proses menampilkan Component di React
- Pertama, trigger (memicu) proses render
- Kedua, melakukan proses render Component
- Ketiga, menempatkan hasil render Component ke DOM (Document Object Model)

# Proses Render di React



# Trigger Render

- Pemicu render biasanya terjadi karena dua hal
- Pertama adalah inisialisasi awal Component, yang kita lakukan menggunakan method render()
- Kedua adalah ketika ada pemicu perubahan State
- Setiap terjadi perubahan State, secara otomatis React akan mengirimkan antrian untuk memicu proses render ulang

# Render Component

- Setelah proses Trigger Render terjadi, React akan memanggil Component yang kita buat untuk mencari tahu apa yang harus ditampilkan di layar
- Rendering adalah proses React memanggil Component yang kita buat
- Pada proses inisialisasi awal, React akan memanggil Root Component (paling atas)
- Pada saat proses render ulang, React hanya akan memanggil ulang Component yang state nya berubah
- Setelah proses render selesai, React akan melakukan proses Commit

# Commit Changes

- Setelah proses render selesai, React akan melakukan proses commit changes (menyimpan perubahan) ke DOM
- Untuk inisialisasi awal, karena baru pertama kali, artinya element di DOM belum ada, maka React akan menggunakan appendChild() untuk menambahkan element ke DOM
- Sedangkan untuk proses render ulang, React akan mencoba melakukan perubahan seminimal mungkin untuk menyamakan DOM saat ini dengan hasil rendering
- React hanya akan mengubah element di DOM, jika memang element tersebut berbeda dari hasil rendering

# Snapshot

# Snapshot

- Variable State sekilas mungkin terlihat seperti variable JavaScript biasa
- Tapi sebenarnya, State itu mirip seperti snapshot (kondisi saat itu). Mengubah nilai variable State tidak akan mengubah Snapshot, melainkan akan memicu render ulang untuk membuat Snapshot baru
- Kita mungkin berpikir bahwa tampilan web berubah secara langsung karena response dari event yang dilakukan oleh pengguna, seperti klik tombol
- Namun sebenarnya tidak seperti itu, kita sudah tahu di materi sebelumnya, ketika terjadi perubahan State, itu akan memicu render ulang, sehingga akan membuat Snapshot baru yang ditampilkan di layar

# Kesalahan Mengubah State

- Paham tentang Snapshot ini, akan memberi gambaran cara pandang kita terhadap data di State
- Berikut adalah contoh kesalahan yang biasa dilakukan ketika mengubah State
- Kita berpikir jika mengubah State, saat itu juga data akan berubah, padahal mengubah State sebenarnya hanyalah mentrigger render ulang dengan nilai state yang baru

# Kode : form/Counter.jsx

```
Counter.jsx ×
1 import {useState} from "react";
2
3 export default function Counter() {
4     let [counter, setCounter] = useState(0);
5     return (
6         <div>
7             <button onClick={
8                 () => {
9                     setCounter(counter + 1);
10                    setCounter(counter + 1);
11                    setCounter(counter + 1);
12                    console.log(counter);
13                }
14            }>Increment +3
15        </button>
16
17            <h1>Counter : {counter}</h1>
18        </div>
19    )
}
```

# Kenapa Counter Tidak Berubah 3x?

- Hal ini dikarenakan, setCounter() tidak akan mengubah data counter yang ada di Snapshot saat ini
- setCounter() hanya akan melakukan render ulang dengan data counter yang baru
- Saat kita memanggil setCounter() sebanyak 3x, bukan berarti React akan melakukan render ulang sebanyak 3x, React akan menunggu sampai Event Handler selesai, jika ada perubahan State, maka akan dilakukan render ulang
- Artinya render ulang akan dilakukan sekali saja, walaupun kita mengubah State berkali-kali

# Kode : form/Counter.jsx

```
C 1 import {useState} from "react";
2
3 export default function Counter() {
4     let [counter, setCounter] = useState(0);
5     return (
6         <div>
7             <button onClick={
8                 () => {
9                     setCounter(counter + 1); // setCounter(0 + 1)
10                    setCounter(counter + 1); // setCounter(0 + 1)
11                    setCounter(counter + 1); // setCounter(0 + 1)
12                     console.log(counter);
13                 }
14             }>Increment +3
15             </button>
16
17             <h1>Counter : {counter}</h1>
18         </div>
19     )
}
```

# State Updates

# State Updates

- Seperti yang sebelumnya dibahas, melakukan update State berkali-kali, tidak akan mengubah data State di Snapshot saat itu, melainkan hanya memicu untuk render ulang dengan data State baru
- Tapi, kadang-kadang, kita memang mungkin ada keperluan untuk mengubah data di State yang sama berkali-kali
- Dan jika kita memang ingin mengubah data di State dengan data yang harapannya sudah diubah sebelumnya (walaupun belum di render ulang)
- Kita bisa menggunakan lambda sebagai parameter ketika memanggil function untuk update data State

# Kode : State Updates



A screenshot of a code editor showing a file named "Counter.jsx". The code defines a functional component "Counter" that uses the "useState" hook from React. It creates a state variable "counter" and a function "setCounter". Inside the component, there is a button with an "onClick" handler that increments the counter by 1 three times and then logs the current value to the console. The component also displays the current value of "counter" in an 

# element.

```
1 import {useState} from "react";
2
3 export default function Counter() {
4     let [counter, setCounter] = useState(0);
5     return (
6         <div>
7             <button onClick={
8                 () => {
9                     setCounter(c => c + 1);
10                    setCounter(c => c + 1);
11                    setCounter(c => c + 1);
12                    console.log(counter);
13                }
14            }>Increment +3
15            </button>
16
17            <h1>Counter : {counter}</h1>
18        </div>
19    )
}
```

# Object di State

# Object di State

- State bisa menyimpan jenis data JavaScript apapun, termasuk Object
- Tapi kita tidak disarankan untuk mengubah object yang terdapat di State
- Jika kita ingin mengubah object di State, kita disarankan membuat object baru lalu mengubah data di State tersebut dengan object baru

# Immutable Data

- Saat kita membuat data di State, kita harus memperlakukan data di State sebagai Immutable data (tidak bisa berubah)
- Artinya data di State hanya digunakan untuk dibaca (read only)
- Jika kita ingin mengubah data di State, maka kita harus ubah menggunakan data baru, yang artinya data lama tidak dimodifikasi
- Ini adalah praktek yang biasa dilakukan di React. Walaupun pada kenyataannya object di JavaScript tidak immutable
- Hal ini direkomendasikan agar kita tidak salah mengubah data langsung, padahal kita tahu bahwa mengubah data tidak akan memicu proses render ulang
- Untungnya di JavaScript kita bisa menggunakan Spread Syntax untuk membantu meng-copy attribute di Object

# Tugas

- Buat halaman baru dengan file
- contact.html
- src/contact/main.jsx
- src/contact/ContactForm.jsx
- Registrasikan ke vite.config.js

# Kode : contact/ContactForm.jsx (1)

ContactForm.jsx ×

```
1 import {useState} from "react";
2
3 export default function ContactForm() {
4     const [contact, setContact] = useState({
5         name: "",
6         message: ""
7     })
8
9     function handleNameChange(e) {
10         setContact({...contact, name: e.target.value})
11     }
12
13    function handleMessageChange(e) {
14        setContact({...contact, message: e.target.value})
15    }
16}
```

# Kode : contact/ContactForm.jsx (2)

```
return (
  <div>
    <h1>Contact Form</h1>
    <form>
      <input type="text" placeholder="Name" value={contact.name} onChange={handleNameChange}>
      <br/>
      <input type="text" placeholder="Message" value={contact.message} onChange={handleMessageChange}>
    </form>
    <h1>Contact Detail</h1>
    <p>Name: {contact.name}</p>
    <p>Message: {contact.message}</p>
  </div>
)
}
```

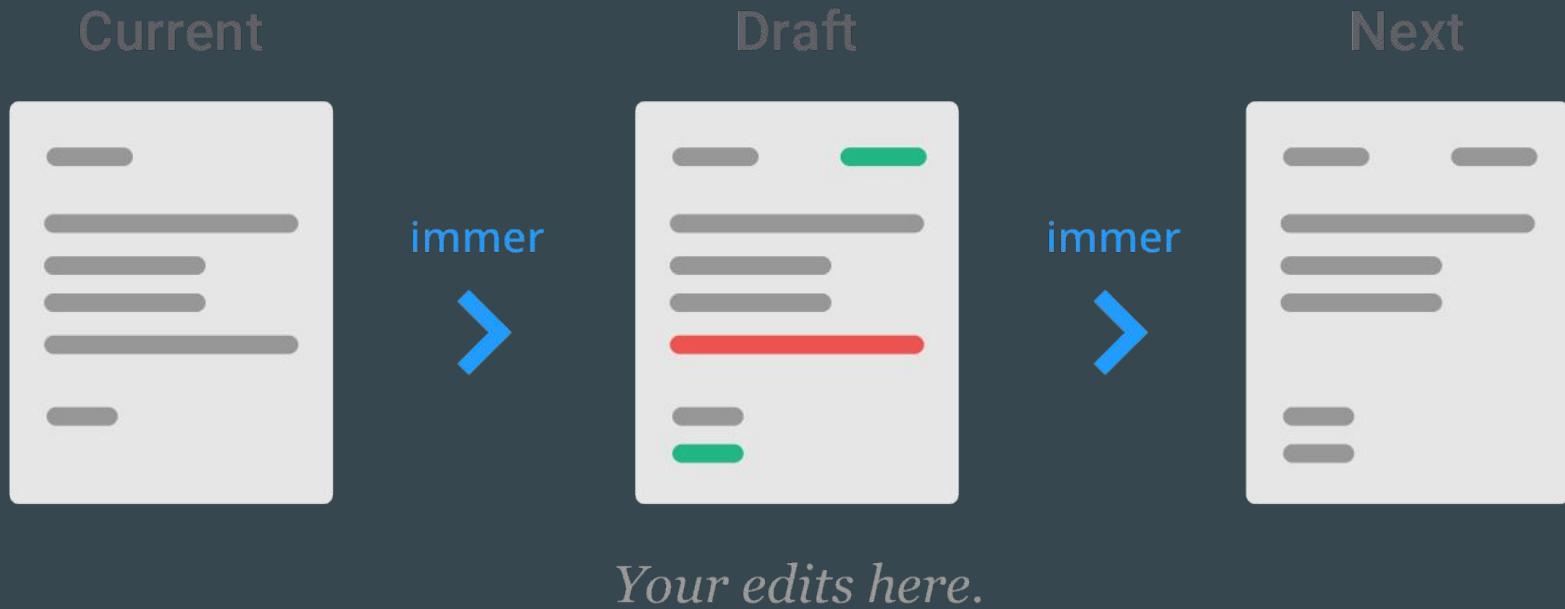
# Nested Object

- Kadang ada kasus kita menggunakan Nested Object
- Sama seperti sebelumnya, kita disarankan untuk selalu membuat object baru ketika mengubah State
- Kadang memang menyulitkan ketika Nested Object nya terlalu besar, oleh karena itu disarankan tidak terlalu dalam membuat Nested Object nya

# Immer Library

- Salah satu library yang sering digunakan ketika develop aplikasi menggunakan React adalah Immer
- Immer adalah library yang digunakan untuk membuat immutable object dari object saat ini
- Menggunakan Immer akan lebih mudah dibandingkan menggunakan Spread Syntax, terutama untuk Object yang kompleks dan Nested
- <https://github.com/immerjs/immer>

# Cara Kerja Library Immer



<https://immerjs.github.io/immer/>

# Use Immer Library

- Library Immer juga bisa diintegrasikan dengan React State dengan mudah
- Kita bisa menggunakan library Use-Immer
- <https://github.com/immerjs/use-immer>
- Kita cukup mengganti useState() menjadi useImmer()
- Dan untuk mengupdate Object di State, kita bisa menggunakan Function sebagai parameter di Update Method nya

# Menambah Library Immer

- npm install immer use-immer

# Kode : contact/ContactForm.jsx

# Array di State

# Array di State

- Sama seperti Object, Array di State juga harus kita perlakukan sebagai Immutable data
- Artinya operasi menambah data, mengubah atau menghapus data di Array, kita harus buat Array baru untuk di update ke State
- Hal ini memang agak menyulitkan, oleh karena itu dengan bantuan library seperti Immer, hal ini jadi lebih mudah

# Mengubah Array

	Hindari	Gunakan
Menambah	push, unshift	concat, [...arr] spread syntax
Menghapus	pop, shift, splice	filter, slice
Mengubah	splice, arr[i] = ...	map
Mengurutkan	reverse, sort	buat array baru

# Tugas

- Buat halaman baru dengan file
- task.html
- src/task/main.jsx
- src/task/Task.jsx
- Registrasikan ke vite.config.js

# Kode : task/Task.jsx (1)

```
Task.jsx ×
1 import {useState} from "react";
2 import {useImmer} from "use-immer";
3
4 export default function Task() {
5     const [item, setItem] = useState("");
6     const [items, setItems] = useImmer([]);
7
8     function handleChange(e) {
9         setItem(e.target.value);
10    }
11
12     function handleClick(e) {
13         e.preventDefault();
14         setItems((items) => {
15             items.push(item);
16         });
17         setItem("");
18    }
19}
```

# Kode : task/Task.jsx (2)

```
return (
  <div>
    <h1>Create Task</h1>
    <form>
      <input value={item} onChange={handleChange}>
      <button onClick={handleClick}>Add
      </button>
    </form>
    <h1>List Tasks</h1>
    <ul>
      {items.map((item, index) =>
        <li key={index}>{item}</li>
      )}
    </ul>
  </div>
);
```

# Sharing State

# Sharing State

- Kadang, kita ingin membuat State untuk beberapa Component yang selalu berubah bersama-sama, atau sederhananya adalah Sharing (berbagi) State
- Untuk melakukan ini, kita harus mengubah lokasi State dari Component-Component itu, ke Parent Component mereka, lalu kita kirim State nya melalui Props
- Misal, pada Form Task sebelumnya, kita akan buat 2 Child Component, Component untuk TaskForm nya, dan Component untuk TaskList nya
- Kita akan share State nya dari Task ke TaskForm dan TaskList melalui Props

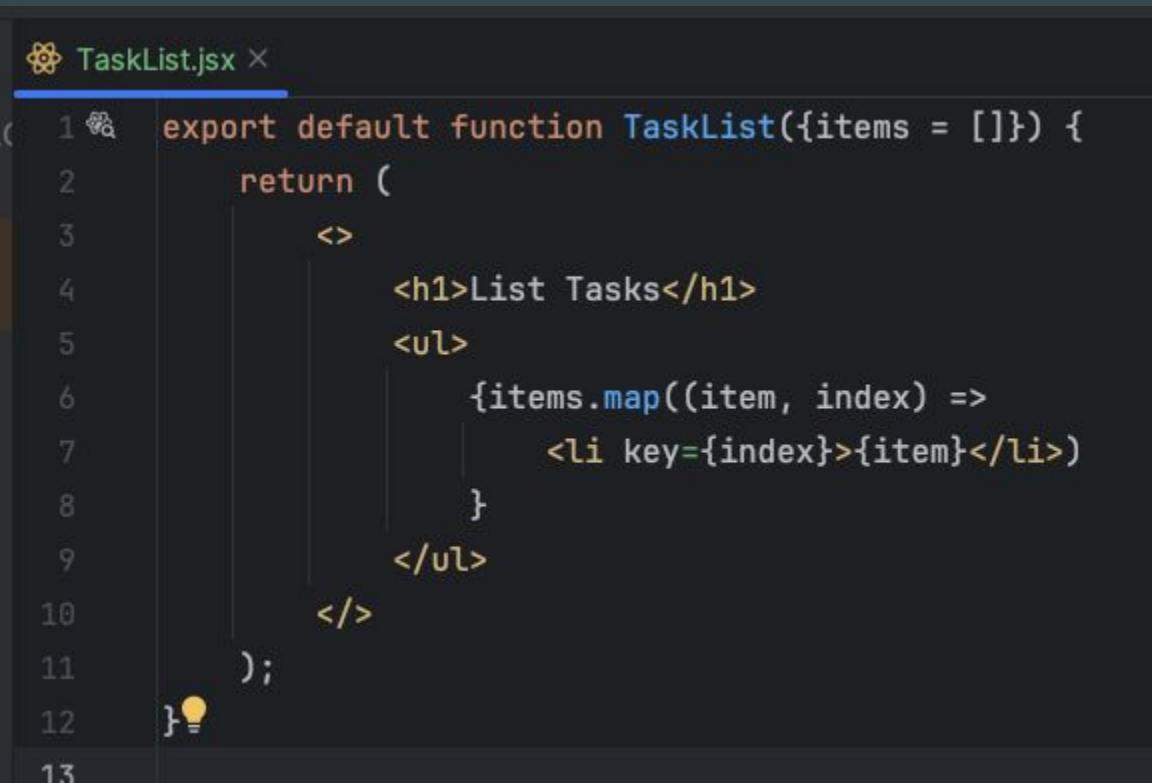
# Kode : task/TaskForm.jsx

TaskForm.jsx ×

```
1 import {useState} from "react";
2
3 export default function TaskForm({onSubmit}) {
4     const [item, setItem] = useState("");
5
6     function handleChange(e) {
7         setItem(e.target.value);
8     }
9
10    function handleClick(e) {
11        e.preventDefault();
12        onSubmit(item);
13        setItem("");
14    }
15}
```

```
return (
  <>
    <h1>Create Task</h1>
    <form>
      <input value={item} onChange={handleChange}>
      <button onClick={handleClick}>Add
      </button>
    </form>
  </>
);
```

# Kode : task/TaskList.jsx



A screenshot of a code editor showing a file named "TaskList.jsx". The code is a functional component that returns a JSX structure. It includes an import statement for React, a function component definition, and a return statement containing an H1 element and a UL element. The UL element uses a map operation to generate list items, each with a unique key. The code is numbered from 1 to 13. A yellow lightbulb icon is visible at the bottom left of the code area.

```
1 import React from 'react';
2
3 export default function TaskList({items = []}) {
4     return (
5         <>
6             <h1>List Tasks</h1>
7             <ul>
8                 {items.map((item, index) =>
9                     <li key={index}>{item}</li>
10                )
11            }
12        </ul>
13    );
}
```

# Kode : task/Task.jsx

```
export default function Task() {
  const [items, setItems] = useImmer([]);

  function handleSubmit(item) {
    setItems(draft => {
      draft.push(item);
    });
  }

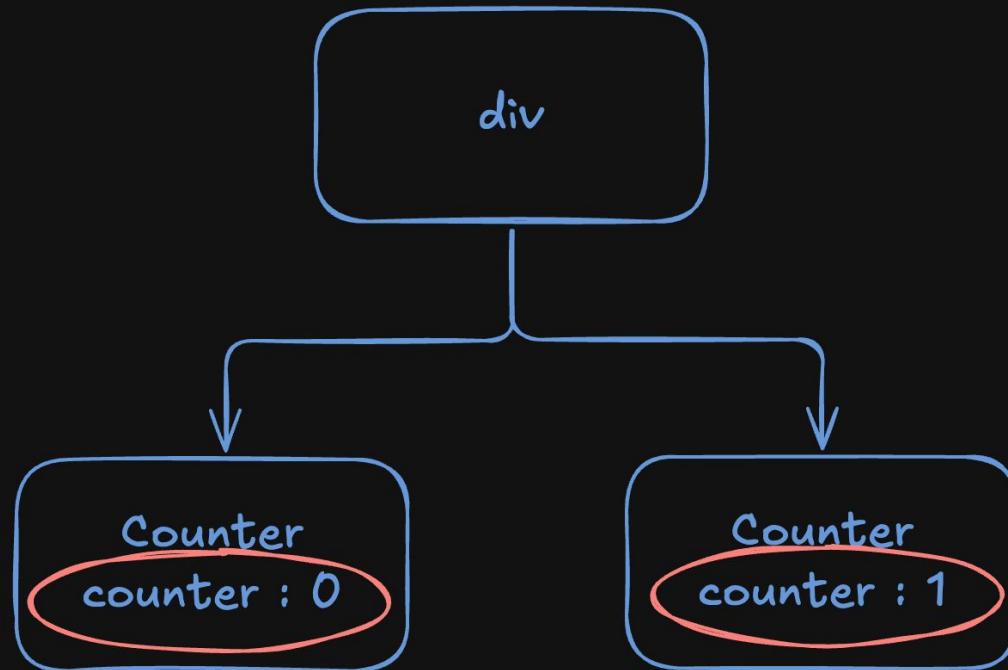
  return (
    <div>
      <TaskForm onSubmit={handleSubmit}>
      <TaskList items={items}>
    </div>
  );
}
```

# Mempertahankan State

# Mempertahankan State

- State terisolasi antar Component. React melacak State mana yang dimiliki oleh Component berdasarkan tempatnya di struktur UI
- Kita bisa mengatur, kapan kita ingin mempertahankan State, kapan kita akan mereset State
- State sendiri tidak disimpan di dalam Component. State itu disimpan di React, sedangkan ketika kita menggunakan State di Component, maka sebenarnya kita akan menggunakan State yang ada di React
- Cara React bisa tahu State mana yang digunakan oleh Component adalah, melihat posisi Component di struktur UI

# State di Struktur UI



# Tugas

- Buat halaman baru dengan file
- counter.html
- src/counter/main.jsx
- src/counter/Counter.jsx
- Registrasikan ke vite.config.js

# Kode : counter/Counter.jsx

Counter.jsx ×

```
1 import {useState} from "react";
2
3 export default function Counter() {
4     const [count, setCount] = useState(0);
5
6     function handleClick() {
7         setCount(count + 1);
8     }
9
10    return (
11        <div>
12            <h1>Counter {count}</h1>
13            <button onClick={handleClick}>Increment</button>
14        </div>
15    )
}
```

# Kode : counter/main.jsx

main.jsx ×

```
1 import {createRoot} from "react-dom/client";
2 import {StrictMode} from "react";
3 import Counter from "./Counter.jsx";
4
5 createRoot(document.getElementById("root"))
6   .render(
7     <StrictMode>
8       <div>
9         <Counter/>
10        <Counter/>
11       </div>
12     </StrictMode>
13   );
14
```

# Posisi Component

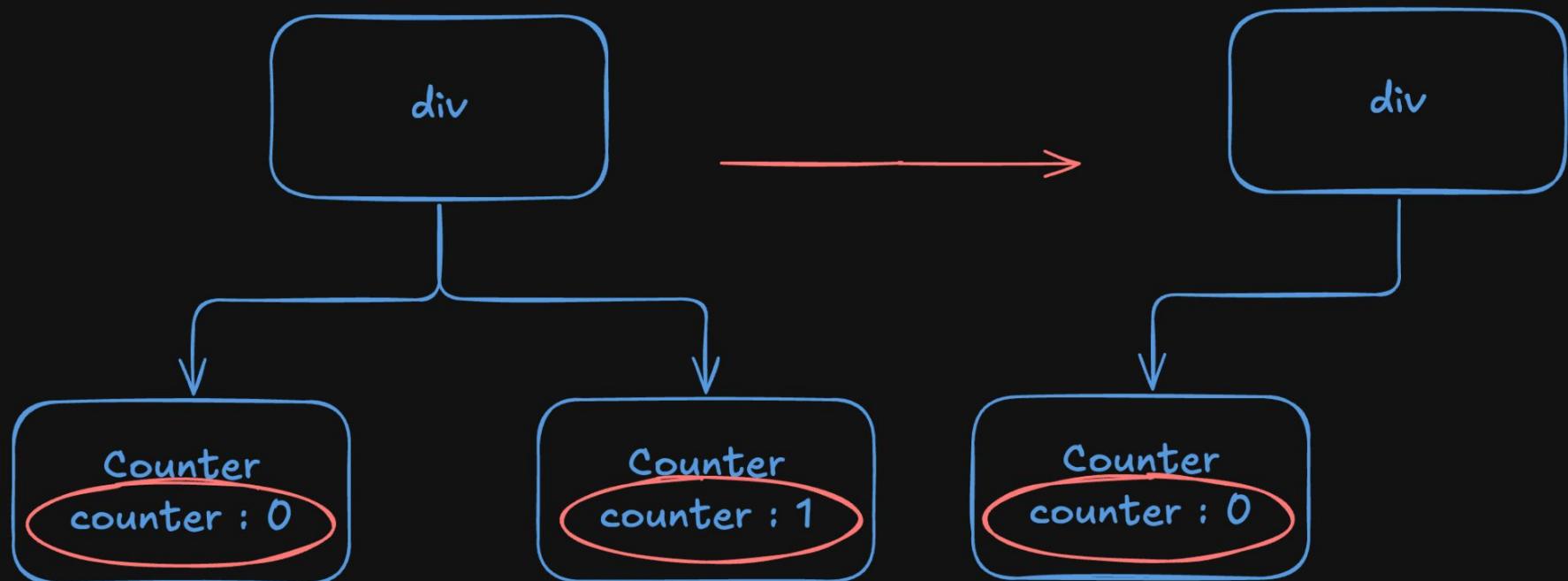
- Seperti dijelaskan sebelumnya, React menyimpan State sesuai dengan posisi Component
- Ketika posisi Componen berubah, misal hilang dari tampilan layar
- Secara otomatis State akan dihapus dari React
- Misal, kita akan membuat Component Counter bisa dihilangkan
- Ketika nanti Component tersebut dihilangkan dari tampilan, mana secara otomatis State nya juga akan hilang

# Kode : counter/CounterApp.jsx

CounterApp.jsx ×

```
1 import {useState} from "react";
2 import Counter from "./Counter.jsx";
3
4 export default function CounterApp() {
5     const [show2, setShow2] = useState(true);
6
7     function handleChange(e) {
8         setShow2(e.target.checked);
9     }
10
11     return (
12         <div>
13             <Counter/>
14             {show2 && <Counter/>}
15             <input type="checkbox" checked={show2} onChange={handleChange}> Tampilkan Counter 2
16         </div>
17     );
}
```

# Posisi Component



# Component Sama di Posisi Sama

- Jika terdapat kasus kita menampilkan Component yang sama
- Tapi secara struktur UI dia berada di posisi yang sama
- Maka State akan dipertahankan oleh React, yang artinya tidak akan dihapus
- Hal ini mungkin akan membingungkan, tapi kita harus mengerti hal ini, karena React akan menyimpan informasi State mengikuti Posisi Component, jika Component nya sama, dan posisinya sama, maka State akan dipertahankan

# Kode : counter/Counter.jsx

Counter.jsx ×

```
1 import {useState} from "react";
2
3 export default function Counter({name}) {
4     const [count, setCount] = useState(0);
5
6     function handleClick() {
7         setCount(count + 1);
8     }
9
10    return (
11        <div>
12            <h1>Counter {name} : {count}</h1>
13            <button onClick={handleClick}>Increment</button>
14        </div>
15    )
}
```

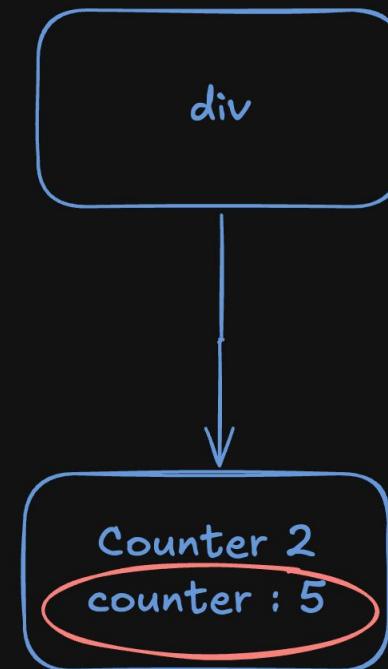
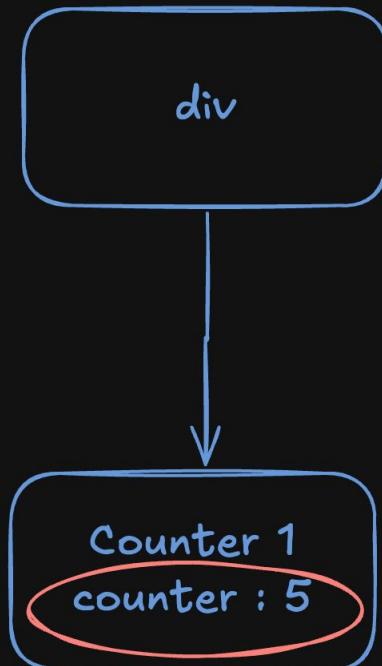
# Kode : counter/CounterApp.jsx

```
⚙️ CounterApp.jsx ×
1 import {useState} from "react";
2 import Counter from "./Counter.jsx";
3
4 export default function CounterApp() {
5   const [show2, setShow2] = useState(true);
6
7   function handleChange(e) {
8     setShow2(e.target.checked);
9   }
10
11   return (
12     <div>
13       {show2 ? <Counter name="2"/> : <Counter name="1"/>}
14       <input type="checkbox" checked={show2} onChange={handleChange}> Tampilkan Counter 2
15     </div>
16   );
17 }
```

# Kenapa State Masih Sama?

- Hal ini terjadi karena secara struktur UI, posisi Component ada di posisi yang sama
- Jika posisi Component dan jenis Component nya sama, secara otomatis React akan mempertahankan State
- Kecuali posisinya berubah atau Component nya berbeda, maka React akan menghapus State nya

# Component Sama di Posisi Sama



# Reset State

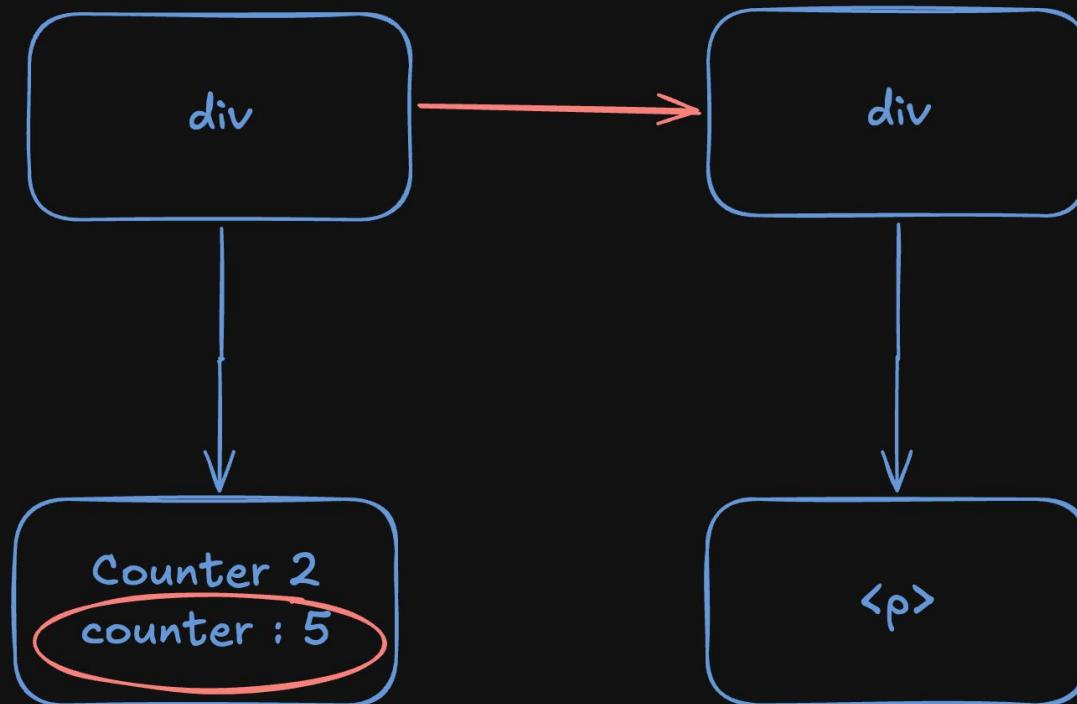
# Reset State

- Sekarang kita tahu bagaimana React mempertahankan data State
- Lantas bagaimana jika misal, pada kondisi tertentu, kita memang ingin melakukan Reset State, tidak mau mempertahankan State nya?
- Ada beberapa cara untuk me-reset State
- Yang pernah kita lakukan di materi sebelumnya adalah dengan cara menghapus Component dari tampilan UI
- Ketika Component hilang dari tampilan UI, secara otomatis State akan ikut hilang

# Mengubah Dengan Component Lain

- Selain menghapus Component, kita juga bisa mengubah Component dengan Component lain di posisi yang sama
- Ketika posisi yang sama masih ada di struktur UI, tapi Component nya berbeda, secara otomatis State juga akan di reset
- Misal sebelumnya kita menampilkan Component Counter, lalu kita ubah menjadi element paragraph. Secara otomatis State di Component Counter akan dihapus

# Mengubah Dengan Component Lain



# Kode : counter/CounterApp.jsx

```
CounterApp.jsx ×
1 import {useState} from "react";
2 import Counter from "./Counter.jsx";
3
4 export default function CounterApp() {
5     const [show2, setShow2] = useState(true);
6
7     function handleChange(e) {
8         setShow2(e.target.checked);
9     }
10
11     return (
12         <div>
13             {show2 ? <Counter name="2"/> : <p>Hilang</p>}
14             <input type="checkbox" checked={show2} onChange={handleChange}> Tampilkan Counter 2
15         </div>
16     );
17 }
```

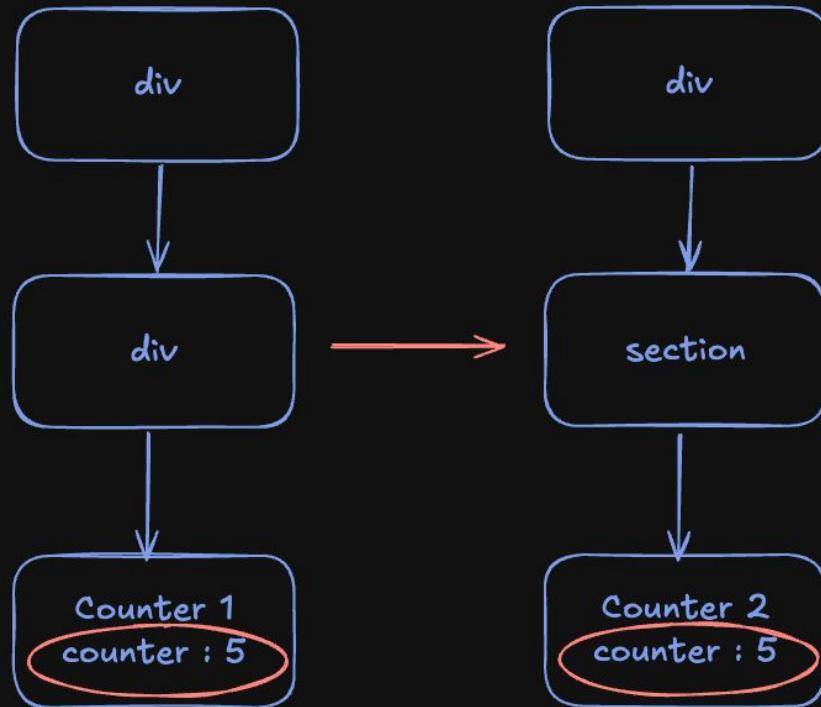
# Mengubah Posisi Component

- Karena React akan menyimpan State sesuai dengan Component dan posisinya
- Artinya jika posisi Component diubah atau dipindahkan, secara otomatis State juga akan di reset

# Kode : counter/CounterApp.jsx

```
0
1      return (
2         <div>
3           {show2 ? (
4             <div>
5               <Counter name="2"/>
6             </div>
7           ) : (
8             <section>
9               <Counter name="1"/>
10              </section>
11            )}
12           <input type="checkbox" checked={show2} onChange={handleChange}/> Tampilkan C
13         </div>
14     );
15 }
```

# Mengubah Posisi Component



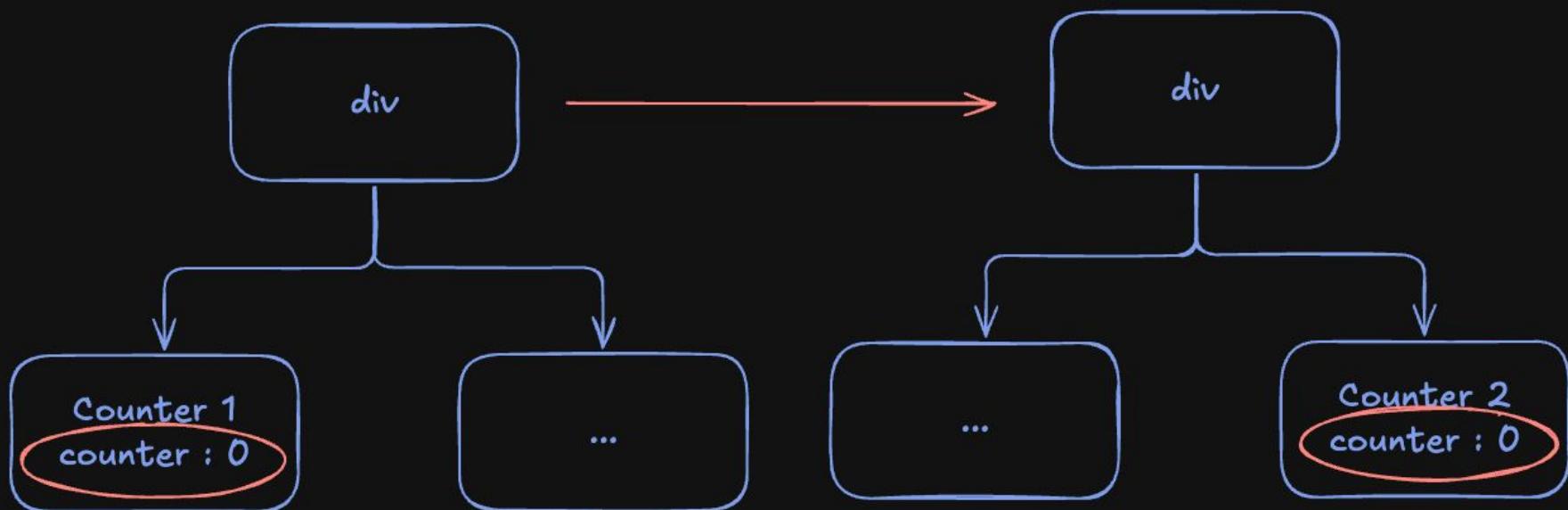
# Posisi Component Berbeda

- Selain mengubah posisi, kita juga bisa jika Component sama berada di posisi yang berbeda, secara otomatis State nya juga akan berbeda
- Ini mungkin akan membingungkan, karena secara DOM mungkin posisinya sama, tapi secara posisi di Component berbeda

# Kode : counter/CounterApp.jsx

```
return (
  <div>
    {!show2 && <Counter name="1"/>}
    {show2 && <Counter name="2"/>}
    <input type="checkbox" checked={show2} onChange={handleChange}> Tampilkan !
  </div>
);
}
```

# Posisi Component Berbeda



# Menggunakan Key

- Cara yang umum biasanya digunakan untuk mereset State, yaitu menambahkan key pada Component
- Saat Component yang sama ditampilkan di posisi yang sama, jika key nya berbeda, maka akan dianggap Component yang berbeda, dengan demikian, secara otomatis State akan di reset
- Ini mungkin cara yang paling direkomendasikan, dibanding harus mengubah-ubah posisi Component

# Kode : counter/CounterApp.jsx

```
    setShow2(e.target.checked);
}

return (
  <div>
    {show2 ? <Counter key="1" name="2"/> : <Counter key="2" name="1"/>}
    <input type="checkbox" checked={show2} onChange={handleChange}/> Tampilkan C
  </div>
);
}
```

# Reducer

# Hooks State Reducer

- Sebelumnya kita susah bahas banyak tentang Hooks State menggunakan useState()
- Selain itu, terdapat fitur Hooks State lain, yaitu Reducer, menggunakan useReducer()
- <https://react.dev/reference/react/useReducer>

# Reducer

- Pada kasus kita membuat Component yang memiliki banyak proses update State, kadang menyulitkan untuk maintain-nya, karena terlalu banyak Event Handler yang harus dibuat juga untuk mengubah data State-nya
- Pada kasus seperti ini, kita bisa mengkonsolidasi semua logic untuk update State di sebuah Function diluar Component, atau kita sebut Reducer
- Agar ada gambaran, kita akan coba membuat Component dengan State yang banyak di update tanpa menggunakan Reducer terlebih dahulu, nanti kita akan ubah menjadi menggunakan Reducer

# Tugas

- Buat halaman baru dengan file
- note.html
- src/note/main.jsx
- src/note/NoteApp.jsx
- src/note/NoteForm.jsx
- src/note/NoteList.jsx
- src/note/Note.jsx
- Registrasikan ke vite.config.js

# Kode : src/note/NoteForm.jsx

NoteForm.jsx ×

```
1 import {useState} from "react";
2
3 export default function NoteForm({onAddNote}) {
4     const [text, setText] = useState('');
5
6     function handleChange(e) {
7         setText(e.target.value);
8     }
9
10    function handleClick() {
11        setText('');
12        onAddNote(text);
13    }
14
15    return (<>
16        <input placeholder="Add Note" value={text} onChange={handleChange}/>
17        <button onClick={handleClick}>Add</button>
18    </>)
19
```

# Kode : src/note/Note.jsx (1)

Note.jsx ×

```
1 import {useState} from "react";
2
3 export default function Note({note, onChange, onDelete}) {
4     const [isEditing, setIsEditing] = useState(false);
5     let component;
6
7     function handleChangeText(e) {
8         const mewNote = {...note, text: e.target.value};
9         onChange(mewNote);
10    }
11
12    if (isEditing) {
13        component = (
14            <>
15                <input value={note.text} onChange={handleChangeText}/>
16                <button onClick={() => setIsEditing(false)}>Save</button>
17            </>
18        )
19    } else {
```

# Kode : src/note/Note.jsx (2)

```
19     } else {
20         component = (
21             <>
22                 {note.text}
23                 <button onClick={() => setIsEditing(true)}>Edit</button>
24             </>
25         )
26     }
27
28 ⚡️ function handleChangeDone(e) {
29     const mewNote = {...note, done: e.target.checked};
30     onChange(mewNote);
31 }
32
33     return (
34         <label>
35             <input type="checkbox" checked={note.done} onChange={handleChangeDone}/>
36             {component}
37             <button onClick={() => onDelete(note)}>Delete</button>
38         </label>
39     )

```

# Kode : src/note/NoteList.jsx

```
□ NoteList.jsx ×
1 import Note from "./Note.jsx";
2
3 export default function NoteList({notes, onChange, onDelete}) {
4     return (
5         <ul>
6             {notes.map(note => (
7                 <li key={note.id}>
8                     <Note note={note} onChange={onChange} onDelete={onDelete}/>
9                 </li>
10            ))}
11        </ul>
12    )
13 }
14 |
```

# Kode : src/note/NoteApp.jsx

NoteApp.jsx ×

```
1 import {useImmer} from "use-immer";
2 import NoteForm from "./NoteForm.jsx";
3 import NoteList from "./NoteList.jsx";
4
5 let id = 0;
6 const initialNotes = [
7     {id: id++, text: 'Learn HTML', done: false},
8     {id: id++, text: 'Learn CSS', done: false},
9     {id: id++, text: 'Learn JavaScript', done: false},
10    {id: id++, text: 'Learn React', done: false},
11];
12
13 export default function NoteApp() {
14     const [notes, setNotes] = useImmer(initialNotes);
15
16     function handleAddNote(text) {
17         setNotes(draft => {
18             draft.push({
19                 id: id++,
20                 text: text,
21                 done: false
22             });
23         });
24     }
25 }
```

```
function handleChangeNote(note) {
    setNotes(draft => {
        const index = draft.findIndex(item =>
            item.id === note.id);
        draft[index] = note;
    })
}

function handleDeleteNote(note) {
    setNotes(draft => {
        const index = draft.findIndex(item =>
            item.id === note.id);
        draft.splice(index, 1);
    });
}

return (
    <div>
        <h1>Note App</h1>
        <NoteForm onAddNote={handleAddNote}/>
        <NoteList notes={notes}
                  onChange={handleChangeNote}
                  onDelete={handleDeleteNote}/>
    </div>
)
```

# Menggunakan Reducer

- Sekarang kita akan fokus ke NoteApp.jsx
- Kita bisa lihat bahwa terdapat 3 aksi untuk update notes, ada add, update dan delete. Dan itu semua disimpan di dalam function yang berbeda-beda dan dari Event Handler yang berbeda-beda
- Menggunakan Reducer agak sedikit berbeda dengan mengubah State secara langsung. Alih-alih memberi tahu ke React apa yang harus dilakukan dengan mengubah State, menggunakan Reducer kita memberi tahu React apa yang sudah dilakukan pengguna (action)
- Implementasi logic dari action nya dilakukan ditempat yang terpisah

# Membuat Reducer Function

- Reducer Function adalah kode dimana kita menyimpan logic kita
- Reducer Function memiliki dua parameter, State saat ini, dan Action object
- Return function nya adalah State selanjutnya

# Kode : src/note/NoteApp.jsx - notesReducer

```
function notesReducer(notes, action) {
  switch (action.type) {
    case 'ADD_NOTE':
      return [
        ...notes,
        {
          id: id++,
          text: action.text,
          done: false
        }
      ];
    case 'CHANGE_NOTE':
      return notes.map(note =>
        note.id === action.id ? {...note, text: action.text, done: action.done} : note
      );
    case 'DELETE_NOTE':
      return notes.filter(note => note.id !== action.id);
    default:
      return notes;
  }
}
```

# Kode : src/note/NoteApp.jsx

```
export default function NoteApp() {
  const [notes, dispatch] = useReducer(notesReducer, initialNotes);

  function handleAddNote(text) {
    dispatch({
      type: 'ADD_NOTE',
      text: text
    })
  }

  function handleChangeNote(note) {
    dispatch({
      type: 'CHANGE_NOTE',
      id: note.id,
      text: note.text,
      done: note.done
    })
  }

  function handleDeleteNote(note) {
    dispatch({
      type: 'DELETE_NOTE',
      id: note.id
    })
  }
}
```

# Menggunakan Immer

- Library Use-Immer juga mendukung Reducer, sehingga kita bisa lebih mudah karena kita bisa mengedit data draft dibanding membuat data baru dari State yang ada
- Kita bisa menggunakan method `useImmerReducer()`

# Kode : Menggunakan Immer Reducer

```
function notesReducer(draft, action) {
  if (action.type === 'ADD_NOTE') {
    draft.push({
      id: id++,
      text: action.text,
      done: false
    });
  } else if (action.type === 'CHANGE_NOTE') {
    const index = draft.findIndex(note => note.id === action.id);
    draft[index].text = action.text;
    draft[index].done = action.done;
  } else if (action.type === 'DELETE_NOTE') {
    const index = draft.findIndex(note => note.id === action.id);
    draft.splice(index, 1);
  }
}
```

# Context

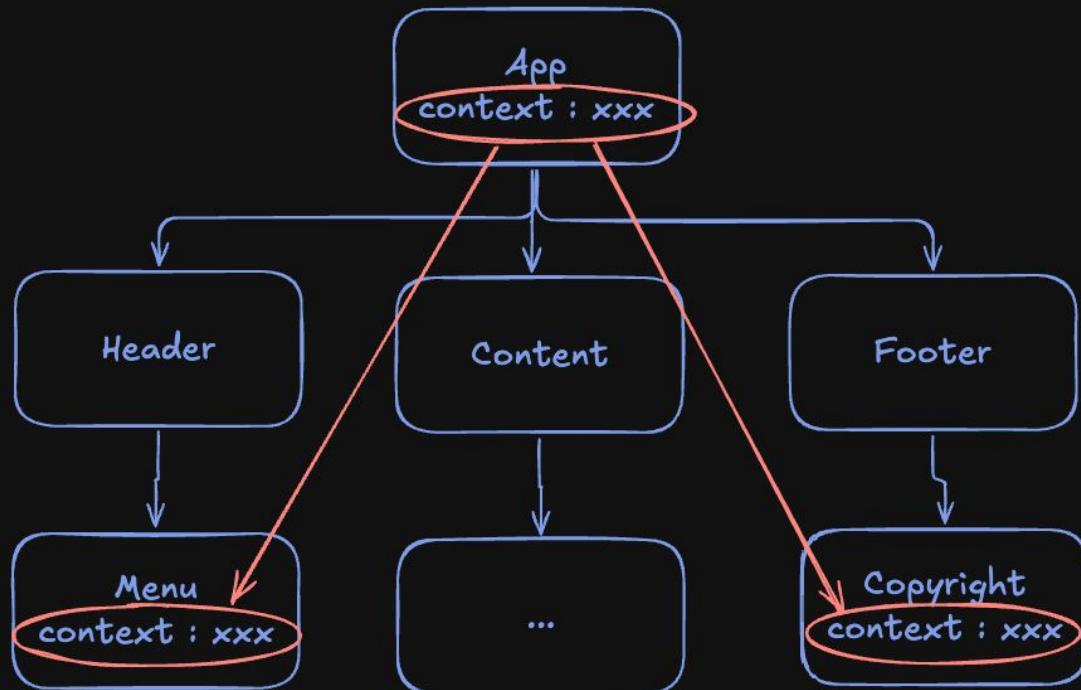
# Context Hooks

- Selain State Hooks yang sudah kita bahas sebelumnya menggunakan useState() dan useReduce()
- Masih ada Hooks yang lainnya, salah satunya adalah Context Hooks, menggunakan useContext()
- <https://react.dev/reference/react/useContext>

# Context

- Biasanya, untuk mengirim informasi dari parent Component ke child Component, kita biasa menggunakan Props
- Tapi mengirim informasi melalui banyak Component, mungkin akan membuat kita terlalu sulit melakukan maintain Props nya
- Atau, misal saja kita punya satu informasi yang digunakan oleh banyak Component, maka mengirim ke semua Component juga akan terlalu sulit
- Context membolehkan parent Component membuat informasi dan bisa digunakan oleh Component manapun di bawah nya, tidak peduli seberapa dalam Component di bawah nya
- Banyak juga yang bilang jika Context adalah Global State

# Diagram : Context



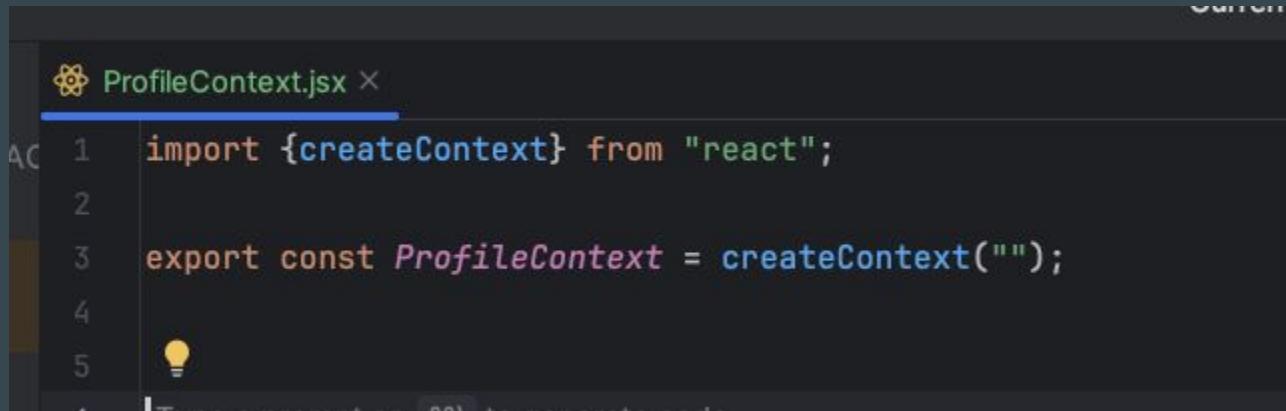
# Membuat Context

- Untuk membuat Context, kita bisa menggunakan function createContext()
- <https://react.dev/reference/react/createContext>
- Selanjutnya, setelah membuat Context, untuk menggunakan Context tersebut, kita bisa menggunakan useContext()
- <https://react.dev/reference/react/useContext>
- Untuk mengubah data di Context, kita gunakan Provider yang terdapat di Context. Secara otomatis semua Component dibawahnya akan mendapat nilai sesuai yang kita ubah di Context Provider
- <https://react.dev/reference/react createContext#provider>

# Tugas

- Misal kita akan membuat halaman profile, dimana kita ingin menampilkan nama pengguna di semua Component child nya
- Buat halaman baru dengan file
  - profile.html
  - src/profile/main.jsx
  - src/profile/ProfileContext.jsx
  - src/profile/ProfileApp.jsx
  - src/profile/Profile.jsx
  - src/profile/ProfileAddress.jsx
- Registrasikan ke vite.config.js

# Kode : src/profile/ProfileContext.jsx



A screenshot of a code editor window titled "ProfileContext.jsx". The code editor has a dark theme. The file contains the following code:

```
1 import {createContext} from "react";
2
3 export const ProfileContext = createContext("");
4
5
```

The first line of code, "import {createContext} from "react";", has a yellow lightbulb icon in the gutter, indicating a potential issue or suggestion.

# Kode : src/profile/ProfileAddress.jsx

ProfileAddress.jsx ×

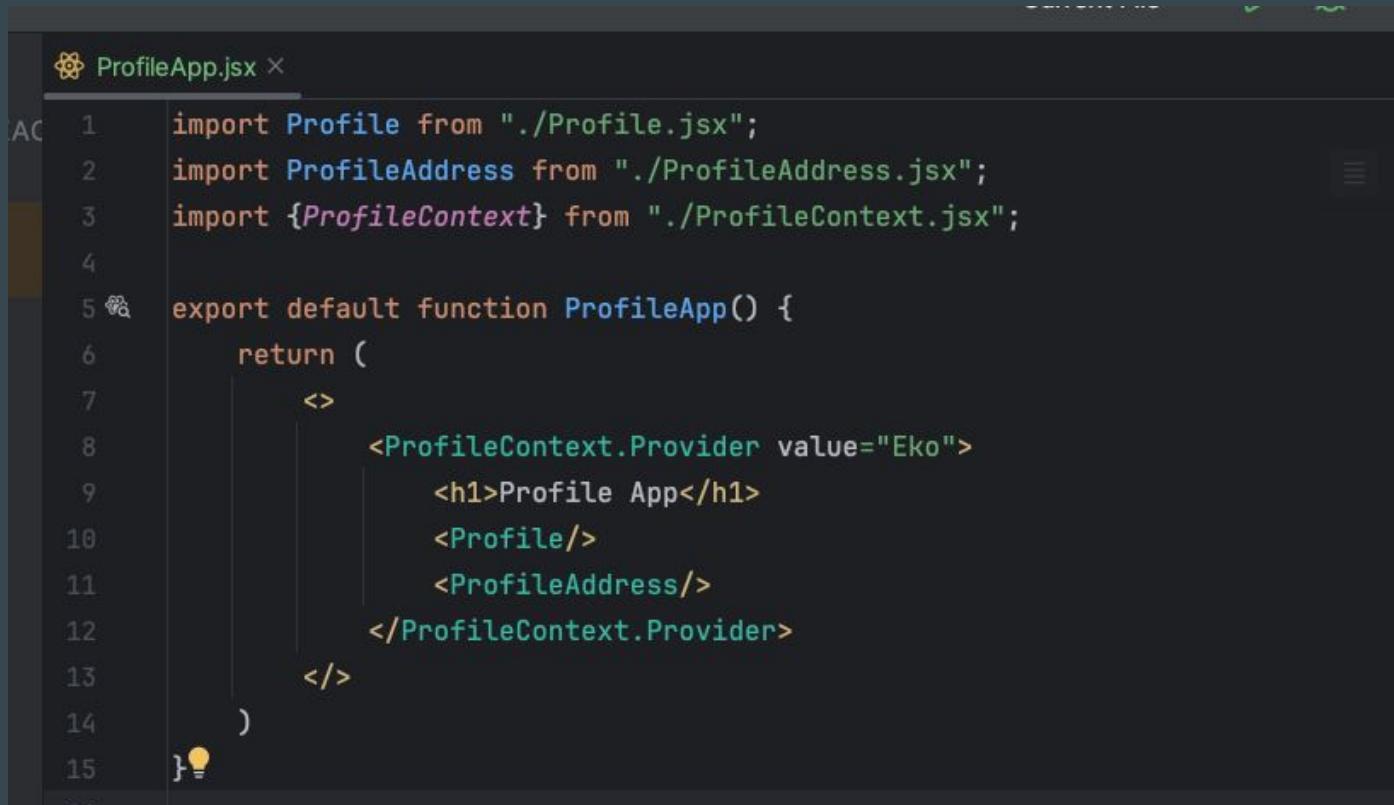
```
1 import {useContext} from "react";
2 import {ProfileContext} from "./ProfileContext.jsx";
3
4 export default function ProfileAddress() {
5     const profile = useContext(ProfileContext);
6     return (
7         <>
8             <h2>Profile Address</h2>
9             <p>Alamat {profile}</p>
10        </>
11    )
12 }
```

# Kode : src/profile/Profile.jsx

Profile.jsx ×

```
1 import {ProfileContext} from "./ProfileContext.jsx";
2 import {useContext} from "react";
3
4 export default function Profile() {
5     const profile = useContext(ProfileContext);
6     return (
7         <>
8             <h2>Profile</h2>
9             <p>Hello {profile}</p>
10        </>
11    )
12 }
```

# Kode : src/profile/ProfileApp.jsx



A screenshot of a code editor window titled "ProfileApp.jsx". The code is written in JSX and defines a functional component named "ProfileApp". The component returns a functional component that uses the "ProfileContext.Provider" to wrap its children. The provider's value is set to "Eko". Inside the provider, there is an "h1" element with the text "Profile App", a "Profile" component, and a "ProfileAddress" component.

```
ProfileApp.jsx
import Profile from "./Profile.jsx";
import ProfileAddress from "./ProfileAddress.jsx";
import {ProfileContext} from "./ProfileContext.jsx";

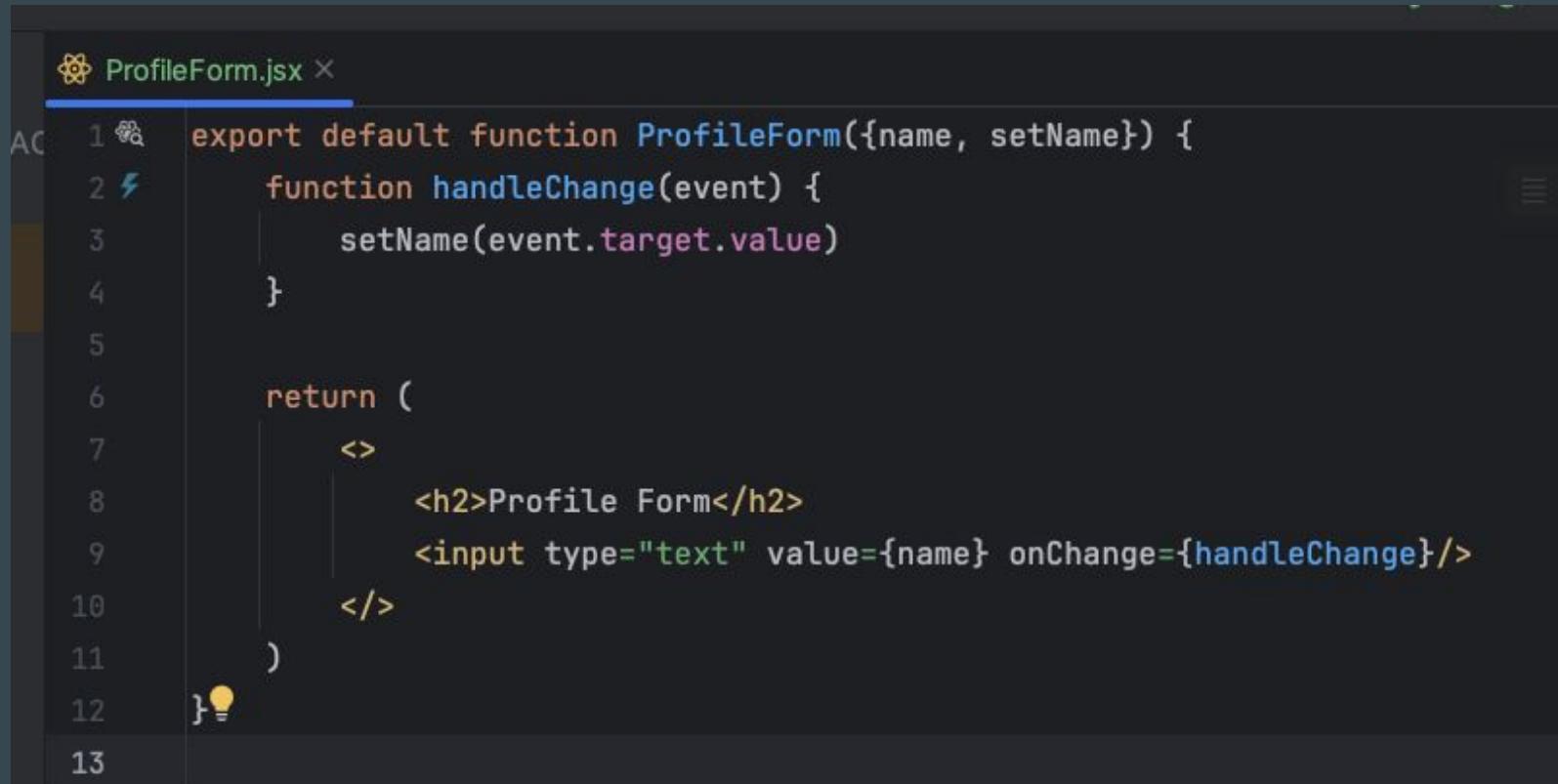
export default function ProfileApp() {
    return (
        <>
            <ProfileContext.Provider value="Eko">
                <h1>Profile App</h1>
                <Profile/>
                <ProfileAddress/>
            </ProfileContext.Provider>
        </>
    )
}
```

# Context dan State

# Context dan State

- Untuk mengubah Context, kita harus menggunakan Provider
- Hal ini karena Context itu hanya bisa diakses oleh Component di bawahnya, tidak bisa Component diatasnya atau yang sejajar
- Selain itu, Component di bawahnya hanya bisa membaca data dari Context, tidak bisa mengubah data di Context
- Oleh karena itu, jika kita ingin membuat data di Context bisa diubah dengan mudah, kita bisa menggunakan bantuan State

# Kode : src/profile/ProfileForm.jsx



A screenshot of a code editor showing a file named `ProfileForm.jsx`. The code is a functional component that renders an input field with a placeholder "Profile Form". The component has a state variable `name` and a function `handleChange` that updates the state when the input value changes.

```
AC 1 export default function ProfileForm({name, setName}) {  
  2   function handleChange(event) {  
  3     setName(event.target.value)  
  4   }  
  5  
  6   return (  
  7     <>  
  8       <h2>Profile Form</h2>  
  9       <input type="text" value={name} onChange={handleChange}>  
10     </>  
11   )  
12 }  
13
```

# Kode : src/profile/ProfileApp.jsx

```
ProfileApp.jsx ×

1 import Profile from "./Profile.jsx";
2 import ProfileAddress from "./ProfileAddress.jsx";
3 import {ProfileContext} from "./ProfileContext.jsx";
4 import ProfileForm from "./ProfileForm.jsx";
5 import {useState} from "react";
6
7 export default function ProfileApp() {
8     const [name, setName] = useState("Eko")
9
10    return (
11        <>
12            <ProfileContext.Provider value={name}>
13                <h1>Profile App</h1>
14                <ProfileForm name={name} setName={setName}/>
15                <Profile/>
16                <ProfileAddress/>
17            </ProfileContext.Provider>
18        </>
19    )
}
```

# Sebelum Menggunakan Context

- Context mungkin lebih mudah digunakan dibanding mengirim semua data State via Props
- Tapi jangan terlalu sering menggunakan Context sampai ke hal yang sederhana
- Gunakan Context jika memang perlu, jika masih sederhana, bisa gunakan State dan Props dulu, jika sudah terlalu kompleks dan terlalu dalam mengirim Props nya, baru diubah ke Context

# Context dan Reducer

# Context dan Reducer

- Sebelumnya kita sudah bahas tentang Reducer, sama seperti State, Reducer juga bisa kita integrasikan dengan Context
- Pada kasus ketika kita membuat Component yang sudah kompleks, dan menggunakan Reducer, kita bisa mengirim semua State dan Reducer menggunakan Context ke child Component
- Sehingga lebih mudah dibanding menggunakan Props
- Misal kita coba modifikasi halaman Notes yang sebelumnya sudah kita buat

# Kode : src/note/NoteContext.jsx

NoteContext.jsx

```
1 import {createContext} from "react";
2
3 export const NotesContext = createContext(null);
4
5 export const NotesDispatchContext = createContext(null);
6
```

# Pindahkan Event Handler

- Karena method Dispatch ada di Context, jadi kita tidak perlu mengirim Event Handler lagi dari Parent Component ke Child Component melalui Props
- Kita bisa langsung pindahkan ke Child Component, karena Child Component bisa mengakses Dispatch Function

# Kode : src/note/NoteForm.jsx

NoteForm.jsx ×

```
1 import {useContext, useState} from "react";
2 import {NotesDispatchContext} from "./NoteContext.jsx";
3
4 export default function NoteForm() {
5     const [text, setText] = useState('');
6     const dispatch = useContext(NotesDispatchContext);
7
8     function handleChange(e) {
9         setText(e.target.value);
10    }
11
12     function handleClick() {
13         dispatch({
14             type: 'ADD_NOTE',
15             text: text
16         });
17         setText('');
18    }
19
20     return (<>
21         <input placeholder="Add Note" value={text} onChange={handleChange}/>
22         <button onClick={handleClick}>Add</button>
23     </>)
}
```

# Kode : src/note/NoteList.jsx

```
⚙ NoteList.jsx ×
1 import Note from "./Note.jsx";
2 import {useContext} from "react";
3 import {NotesContext} from "./NoteContext.jsx";
4
5 ❄ export default function NoteList() {
6     const notes = useContext(NotesContext);
7     return (
8         <ul>
9             {notes.map(note => (
10                 <li key={note.id}>
11                     <Note note={note}/>
12                 </li>
13             ))}
14         </ul>
15     )
16 }
17
```

# Kode : src/note/Note.jsx

```
export default function Note({note}) {
  const dispatch = useContext(NotesDispatchContext);
  const [isEditing, setIsEditing] = useState(false);
  function handleChangeText(e) {
    dispatch({
      ...note,
      type: 'CHANGE_NOTE',
      text: e.target.value
    })
  }
  function handleChangeDone(e) {
    dispatch({
      ...note,
      type: 'CHANGE_NOTE',
      done: e.target.checked
    })
  }
  function handleDelete() {
    dispatch({
      type: 'DELETE_NOTE',
      id: note.id
    })
  }
}

let component;
if (isEditing) {
  component = (
    <>
      <input value={note.text} onChange={handleChangeText}/>
      <button onClick={() => setIsEditing(false)}>Save</button>
    </>
  )
} else {
  component = (
    <>
      {note.text}
      <button onClick={() => setIsEditing(true)}>Edit</button>
    </>
  )
}

return (
  <label>
    <input type="checkbox" checked={note.done} onChange={handleChangeDone}/>
    {component}
    <button onClick={handleDelete}>Delete</button>
  </label>
)
```

# Kode : src/note/NoteApp.jsx

```
30
31  export default function NoteApp() {
32      const [notes, dispatch] = useImmerReducer(notesReducer, initialNotes);
33
34      return (
35          <div>
36              <NotesContext.Provider value={notes}>
37                  <NotesDispatchContext.Provider value={dispatch}>
38                      <h1>Note App</h1>
39                      <NoteForm/>
40                      <NoteList/>
41                  </NotesDispatchContext.Provider>
42              </NotesContext.Provider>
43          </div>
44      )
45  }
46
```

Ref

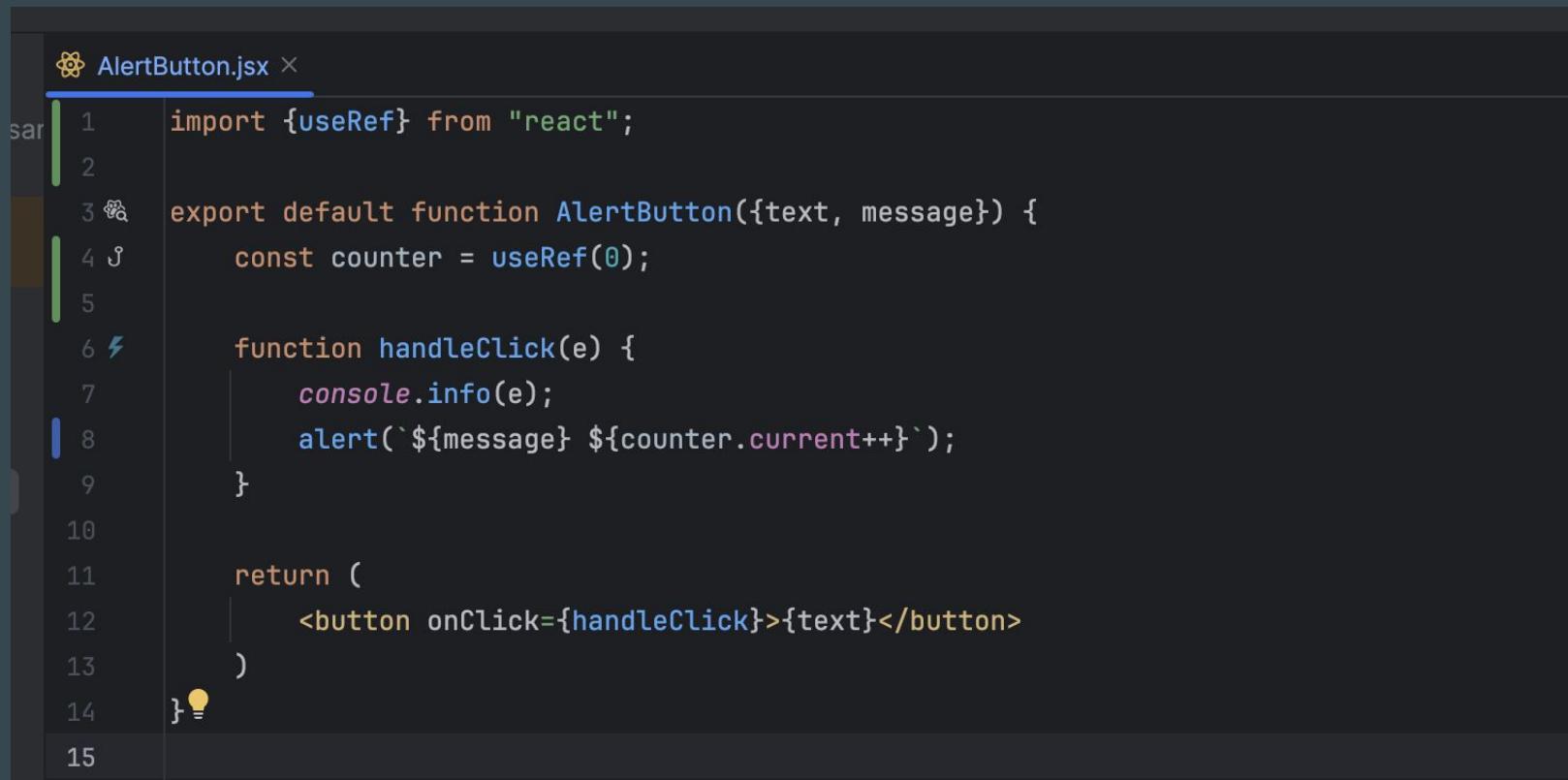
# Ref Hooks

- Fitur Hooks yang lain selain State dan Context, adalah Ref Hooks
- Ketika kita ingin Component mengingat informasi, tapi tidak mau memicu render ulang, maka kita bisa menggunakan Ref
- Menggunakan Ref Hooks bisa menggunakan method useRef()
- <https://react.dev/reference/react/useRef>
- Function useRef() akan mengembalikan Object yang memiliki attribute current, dimana attribute current berisi value yang dipegang oleh Ref

# Perbedaan Ref dan State

Ref	State
<code>userRef(initial)</code> mengembalikan object <code>{current: initial}</code>	<code>useState(initial)</code> mengembalikan [value, setValue]
Tidak memicu render ulang ketika diubah	Memicu render ulang ketika diubah
Mutable, bisa dimodifikasi dan diubah current value nya	Immutable, kita hanya bisa mengubah menggunakan function setValue
Tidak direkomendasikan membaca atau menulis current value ketika proses rendering. Lebih cocok dibaca atau diubah pada Event Handler	Bisa dibaca kapanpun, tapi tiap render akan memiliki Snapshot masing-masing

# Kode : src/button/AlertButton.jsx



A screenshot of a code editor showing a file named "AlertButton.jsx". The code is a functional component that uses the "useRef" hook from React to manage a counter. It logs the click event to the console and displays an alert with the message and the current value of the counter. The code is numbered from 1 to 15.

```
AlertButton.jsx
1 import {useRef} from "react";
2
3 export default function AlertButton({text, message}) {
4     const counter = useRef(0);
5
6     function handleClick(e) {
7         console.info(e);
8         alert(`${message} ${counter.current++}`);
9     }
10
11     return (
12         <button onClick={handleClick}>{text}</button>
13     )
14 }
15
```

# Tugas

- Buat halaman baru dengan file
- timer.html
- src/timer/main.jsx
- src/timer/Timer.jsx
- Registrasikan ke vite.config.js

# Kode : src/timer/Timer.jsx

Timer.jsx ×

```
1 import {useRef, useState} from "react";
2
3 export default function Timer() {
4   const [start, setStart] = useState(null);
5   const [now, setNow] = useState(null);
6   const timer = useRef(null);
7
8   function handleStart() {
9     setStart(Date.now());
10    setNow(Date.now());
11
12    timer.current = setInterval(() => {
13      setNow(new Date().getTime());
14    }, 10);
15  }
16
17  return (
18    <>
19      <h1>Time : {now - start} ms</h1>
20      <button onClick={handleStart}>Start</button>
21      <button onClick={handleStop}>Stop</button>
22    </>
23  );
24}
```

```
function handleStop() {
  clearInterval(timer.current);
}

return (
  <>
    <h1>Time : {now - start} ms</h1>
    <button onClick={handleStart}>Start</button>
    <button onClick={handleStop}>Stop</button>
  </>
)
```

# Manipulasi DOM dengan Ref

# Manipulasi DOM dengan Ref

- React secara otomatis akan mengupdate DOM ketika melakukan render ulang, jadi kita tidak perlu memanipulasi DOM secara manual lagi
- Tapi, kadang kita mungkin perlu mengakses DOM secara manual, contoh memindahkan fokus ke salah satu element, atau scroll ke element tertentu, dan lain-lain
- Sayangnya, tidak ada cara untuk melakukan hal ini menggunakan React, jadi kita perlu tangani hal ini secara manual
- Salah satu caranya kita menggunakan Ref menuju ke DOM element
- Caranya, pada element nya, kita bisa tambahkan attribute ref

# Tugas

- Buat halaman baru dengan file
- guestbook.html
- src/guestbook/main.jsx
- src/guestbook/GuestBook.jsx
- Registrasikan ke vite.config.js

# Kode : src/guestbook/GuestBook.jsx

```
export default function GuestBook() {
  const [name, setName] = useState("");
  const [message, setMessage] = useState("");

  const nameInput = useRef(null);

  function handleSubmit(e) {
    e.preventDefault();
    setName("");
    setMessage("");

    nameInput.current.focus();

    alert(`Name: ${name}, Message: ${message}`);
  }
}
```

```
return (
  <>
    <h1>Guest Book</h1>
    <form>
      <label htmlFor="name">Name</label><br/>
      <input ref={nameInput} type="text" name="name" value={name}
        onChange={(e) => setName(e.target.value)} /> <br/>
      <label htmlFor="message">Message</label><br/>
      <textarea name="message" value={message}
        onChange={(e) => setMessage(e.target.value)} /> <br/>
      <button type="submit" onClick={handleSubmit}>Submit</button>
    </form>
  </>
)
```

# Ref untuk Component

- Ref hanya bisa digunakan di DOM element, kita tidak bisa menggunakan ref di Component
- Jika kita menambahkan Ref ke Component, maka attribute current akan bernilai null
- Kita akan coba praktikan ini dengan membuat GuestBookInput Component

# Kode : src/guestbook/GuestBookInput.jsx

GuestBookForm.jsx ×

```
1 export default function GuestBookForm({name, setName}) {  
2     return (  
3         <>  
4             <label htmlFor="name">Name</label><br/>  
5             <input type="text" name="name" value={name}  
6                 onChange={(e) => setName(e.target.value)} /> <br/>  
7         </>  
8     )  
9 }💡  
10 |
```

# Kode : src/guestbook/GuestBook.jsx

```
return (
  <>
    <h1>Guest Book</h1>
    <form>
      <GuestBookForm ref={nameInput} name={name} setName={setName}/>
      <label htmlFor="message">Message</label><br/>
      <textarea name="message" value={message}
                onChange={(e) => setMessage(e.target.value)}><br/>
      <button type="submit" onClick={handleSubmit}>Submit</button>
    </form>
  </>
)
}
```

# Mengakses Component DOM Element

- Karena Component tidak bisa dijadikan sebagai Ref, oleh karena itu, jika kita ingin menggunakan Ref untuk Component, kita bisa menggunakan Props
- Kita bisa membuat Props ref yang kita isi Ref
- Props ref bisa kita gunakan pada DOM element di dalam Component tersebut
- Sekarang kita coba tambahkan Ref ke dalam Component GuestBookInput

# Kode : src/guestbook/GuestBookInput.jsx

```
GuestBookForm.jsx ×  
1 export default function GuestBookForm({ref, name, setName}) {  
2     return (  
3         <>  
4             <label htmlFor="name">Name</label><br/>  
5             <input ref={ref} type="text" name="name" value={name}  
6                 onChange={(e) => setName(e.target.value)} /> <br/>  
7         </>  
8     )  
9 }💡  
10 |
```

# Effect

# Effect Hooks

- Hooks selanjutnya yang akan kita bahas adalah Effect Hooks
- Beberapa Component mungkin butuh untuk berkomunikasi dengan External System, misal berkomunikasi dengan Non-React Component, berkomunikasi dengan Server, dan lain-lain
- Effect Hooks memungkinkan kita membuat kode yang dijalankan setelah proses render sehingga kita bisa berkomunikasi dengan system diluar React
- Effect terjadi setelah proses render selesai
- Kita bisa menggunakan `useEffect()` untuk membuat Effect
- <https://react.dev/reference/react/useEffect>

# Kenapa Tidak Cukup Event Handler?

- Biasanya kita tahu bahwa Component itu harus Pure, dan tidak memiliki efek samping
- Untuk kode yang memiliki efek samping, biasanya kita gunakan Event Handler
- Namun, kadang ada kebutuhan kita memang perlu membuat Component yang memiliki efek samping ketika dirender
- Misal ketika Component di render, kita ingin mengambil data dari Server, sehingga bisa menyebabkan isi Component berubah (efek samping) sesuai dengan response Server
- Disinilah Effect Hooks diperlukan

# Tugas

- Buat halaman baru dengan file
- product.html
- src/product/main.jsx
- src/product/ProductList.jsx
- src/product/Product.jsx
- public/products.json
- Registrasikan ke vite.config.js

# Kode : public/products.json

{} products.json ×

```
t-c 1  [
2   {
3     "id": 1,
4     "name": "Product 1",
5     "price": 100
6   },
7   {
8     "id": 2,
9     "name": "Product 2",
10    "price": 200
11  },
12  {
13    "id": 3,
14    "name": "Product 3",
15    "price": 300
16  }
```

# Kode : src/product/Product.jsx

A screenshot of a code editor window titled "Product.jsx". The code is a simple functional component named "Product". It returns a `<div>` element containing an `<h2>` with the product's id and name, and a `<p>` with the product's price. The code editor has a dark theme and shows line numbers from 1 to 8. There are small icons next to the line numbers: a magnifying glass for line 1, a lightbulb for line 8, and a yellow exclamation mark for line 9. A status bar at the bottom says "Type a prompt or ⌘\ to generate code".

```
1 export default function Product({product}) {
2     return (
3         <div>
4             <h2>{product.id} : {product.name}</h2>
5             <p>Harga : {product.price}</p>
6         </div>
7     )
8 }
```

# Kode : src/product/ProductList.jsx

```
ProductList.jsx ×  
1 import {useEffect, useRef, useState} from "react";  
2 import Product from "./Product.jsx";  
3  
4 export default function ProductList() {  
5     const [products, setProducts] = useState([]);  
6     const loaded = useRef(false);  
7  
8     useEffect(() => {  
9         if (loaded.current === false) {  
10             fetch("/products.json")  
11                 .then((response) => response.json())  
12                 .then((data) => setProducts(data))  
13                 .then(() => loaded.current = true);  
14     }  
15 };
```

```
return (  
    <>  
        <h1>Product List</h1>  
        {products.map((product) => (  
            <Product key={product.id} product={product}/>  
        ))}  
    </>  
)
```

# Infinite Loop

- Secara default, Effect akan dieksekusi setelah proses render
- Oleh karena itu, kita perlu berhati-hati, karena jika di dalam Effect kita mengubah State, maka bisa terjadi infinite loop yang menyebabkan proses render ulang terus-terusan terjadi tanpa henti

# Kode : Infinite Loop (Jangan Dilakukan)

```
ProductList.jsx ×  
1 import {useEffect, useState} from "react";  
2 import Product from "./Product.jsx";  
3  
4 export default function ProductList() {  
5     const [products, setProducts] = useState([]);  
6  
7     useEffect(() => {  
8         fetch("/products.json")  
9             .then((response) => response.json())  
10            .then((data) => setProducts(data));  
11    });  
12}
```

# Effect Clean Up

- Pada beberapa kasus, mungkin kita butuh melakukan sesuatu setelah eksekusi Effect selesai dilakukan
- Seperti misal di Try Catch, terdapat Finally
- Di Effect terdapat proses yang namanya Clean Up, Clean Up akan dieksekusi setelah proses Effect selesai
- Caranya cukup mudah, kita tinggal return kan Closure Function di dalam Effect
- Clean Up akan dieksekusi sebelum Effect selanjutnya di eksekusi, atau Component di hilangkan

# Kode : Effect Clean Up

```
useEffect(() => {
  if (loaded.current === false) {
    fetch("/products.json")
      .then((response) => response.json())
      .then((data) => setProducts(data))
      .then(() => loaded.current = true);
  }

  return () => {
    console.log("ProductList component unmounted");
  }
});
```

# Effect Dependencies

# Effect Dependencies

- Secara default, Effect akan dieksekusi setiap selesai proses render
- Artinya jika proses render dilakukan berkali-kali, Effect akan dieksekusi juga berkali-kali setelah proses render selesai
- Hal ini menjadikan kita harus melakukan pengecekan secara manual seperti di materi sebelumnya jika tidak ingin kode Effect dijalankan berkali-kali
- useEffect() memiliki parameter kedua yaitu Dependencies
- Kita bisa mengisi Dependencies dengan Array dari State, dimana artinya Effect hanya akan dieksekusi jika State tersebut berubah, jika tidak berubah maka Effect tidak akan dieksekusi
- Hal ini mungkin lebih mudah dibanding manual melakukan if else

# Kode : src/product/ProductList.jsx

```
ProductList.jsx ×  
1 import {useEffect, useState} from "react";  
2 import Product from "./Product.jsx";  
3  
4 export default function ProductList() {  
5   const [products, setProducts] = useState([]);  
6   const [load, setLoad] = useState(false);  
7  
8   function handleClick() {  
9     setLoad(true);  
10  }  
11  
12  useEffect(() => {  
13    console.log("Load products");  
14    if (load) {  
15      fetch("/products.json")  
16        .then((response) => response.json())  
17        .then((data) => setProducts(data));  
18    }  
19  }, [load]);  
20}
```

```
return (  
  <>  
    <h1>Product List</h1>  
    <button onClick={handleClick}>Load Products</button>  
    {products.map((product) => (  
      <Product key={product.id} product={product}/>  
    ))}  
  </>  
)
```

# Empty Dependencies

- Kadang, kadang, mungkin kita hanya ingin memanggil Effect sekali saja setelah pertama kali Component di Render, setelah itu kita tidak mau memanggil Effect lagi, bahkan jika terjadi render ulang
- Pada kasus seperti itu, kita bisa gunakan empty array sebagai dependencies, maka Effect hanya akan dipanggil sekali saja setelah render pertama kali, selanjutnya tidak akan dipanggil lagi

# Kode : src/product/ProductList.jsx

```
export default function ProductList() {
    const [products, setProducts] = useState([]);
    const [load, setLoad] = useState(false);

    function handleClick() {
        setLoad(true);
    }

    useEffect(() => {
        console.log("UseEffect");
    }, []);

    useEffect(() => {
        console.log("Load products");
        if (load) {
            fetch("/products.json")
        }
    }, [load]);
}
```

# Async Code di Effect

# Async Code di Effect

- Kadang, saat berhubungan dengan System External, misal memanggil API, biasanya kita akan membuat kode Async Await
- Sayangnya, Effect tidak mendukung Async Function, oleh karena itu jika kita bisa menggunakan Promise secara langsung, kita bisa langsung saja gunakan Promise, tanpa menggunakan Async Await
- Namun, jika kita memang mau menggunakan Async Await, maka kita harus buat function yang nanti dipanggil di useEffect()
- Misal, kita akan coba ubah kode sebelumnya menjadi kode Async Await ketika mengambil data Products

# Kode : src/product/ProductList.jsx

```
useEffect(() => {
  async function fetchProducts() {
    const response = await fetch("/products.json");
    const data = await response.json();
    setProducts(data);
  }

  console.log("Load products");
  if (load) {
    fetchProducts();
  }
}, [load]);
```

# Jangan Gunakan Effect

# Jangan Gunakan Effect

- Effect Hook adalah cara di luar dari yang biasa dilakukan di React.
- Effect memungkinkan kita keluar dari kebiasaan di React, dan memungkinkan Component kita berinteraksi dengan External System seperti Non React Component, Network, dan lain-lain
- Jika tidak ada interaksi dengan External System, maka sebaiknya jangan gunakan Effect
- Meminimalisir penggunaan Effect akan membuat kode kita mudah untuk dibaca, cepat dieksekusi dan minim error

# Jangan Gunakan Effect untuk Inisialisasi Aplikasi

- Kadang beberapa proses dilakukan di awal sebelum halaman di tampilkan
- Pada kasus ini, gunakan saja kode JavaScript langsung diluar React
- Jangan gunakan Effect untuk melakukan inisialisasi aplikasi
- Hal ini karena Effect akan dieksekusi setelah Render (setelah ditampilkan), dan akan dieksekusi ulang jika terjadi render ulang (bisa berkali-kali)

# Jangan Gunakan Effect untuk Mengubah Data di Server

- Jika terdapat logic di Effect melakukan perubahan data di Server, perlu hati-hati, karena bisa saja Effect dieksekusi berkali-kali karena proses render ulang
- Oleh karena ini, tidak direkomendasikan untuk menggunakan Effect untuk mengubah data di Server
- Gunakan Event Handler jika ingin mengubah data di Server, sehingga jelas kapan perubahan terjadi
- Menggunakan Effect bisa menyebabkan perubahan data di Server terjadi berkali-kali karena render ulang Component

# Memo

# Memoization

- Memoization adalah teknik optimasi untuk mempercepat program komputer, dengan cara menyimpan data secara sementara dari hasil kalkulasi (yang biasanya berat), sehingga tidak perlu dikalkulasi ulang
- Ini adalah bagian dari Performance Hooks, dimana kita bisa menggunakan function `useMemo(callback, [dependencies])`
- <https://react.dev/reference/react/useMemo>
- Ini cocok untuk kasus misal kita perlu memanggil kode yang berat, dan hasilnya selalu sama, dibandingkan kita panggil terus-terusan setiap proses render, lebih baik kita lakukan sekali saja, dan pada proses pemanggilan berikutnya, kita cukup kembalikan hasil yang pertama

# Contoh Penggunaan Memo

- Misal pada pada aplikasi Notes, kita ingin tambahkan proses untuk mencari Note
- Jika misal pengguna melakukan pencarian dengan value yang sama, dibanding kita lakukan pencarian ulang, kita bisa kembalikan hasil yang sama saja seperti sebelumnya

# Kode : src/note/NoteList.jsx

NoteList.jsx ×

```
1 import Note from "./Note.jsx";
2 import {useContext, useMemo, useRef, useState} from "react";
3 import {NotesContext} from "./NoteContext.jsx";
4
5 export default function NoteList() {
6     const notes = useContext(NotesContext);
7     const [search, setSearch] = useState('');
8     const searchInput = useRef(null);
9
10    const filteredNotes = useMemo(() => {
11        console.log('Filtering notes');
12        return notes.filter(note => note.text.includes(search));
13    }, [notes, search])
14
15    function handleSearch() {
16        console.info('Search:', searchInput.current.value);
17        setSearch(searchInput.current.value);
18    }
19}
```

```
return (
    <div>
        <input ref={searchInput} placeholder="Search"/>
        <button onClick={handleSearch}>Search</button>
        <ul>
            {filteredNotes.map(note => (
                <li key={note.id}>
                    <Note note={note}/>
                </li>
            ))}
        </ul>
    </div>
)
```

# Custom Hooks

# Custom Hooks

- React secara default sudah memiliki banyak Hooks untuk kita gunakan untuk mempermudah membuat aplikasi menggunakan React
- Namun, kita juga bisa membuat Hooks secara manual jika kita mau
- Membuat Hooks biasanya menggunakan function dengan awalan “use”
- Contoh, kita akan membuat Hooks untuk mendeteksi apakah sedang Online atau Offline

# Tugas

- Buat halaman baru dengan file
- online.html
- src/online/main.jsx
- src/online/Online.jsx
- src/online/OnlineHook.jsx
- Registrasikan ke vite.config.js

# Kode : src/online/OnlineHook.jsx

OnlineHook.jsx ×

```
1 import {useEffect, useState} from "react";
2
3 export function useOnline() {
4     const [isOnline, setIsOnline] = useState(true);
5
6     useEffect(() => {
7         function handleOnline() {
8             setIsOnline(true);
9         }
10
11         function handleOffline() {
12             setIsOnline(false);
13         }
14
15         window.addEventListener("online", handleOnline);
16         window.addEventListener("offline", handleOffline);
17
18         return () => {
19             window.removeEventListener("online", handleOnline);
20             window.removeEventListener("offline", handleOffline);
21         };
22     }, []);
23
24     return isOnline;
25 }
```

# Kode : src/online/Online.jsx

```
Online.jsx ×
1 import {useOnline} from "./OnlineHook.jsx";
2
3 export default function Online() {
4     const isOnline = useOnline();
5     return (
6         <h1>
7             {isOnline ? "Online" : "Offline"}
8         </h1>
9     )
10 }
11
```

# Referensi Lengkap

# Referensi Lengkap

- <https://react.dev/reference/react>

# Materi Selanjutnya

# Materi Selanjutnya

- ReactJS Router
- ReactJS Redux