OPTIMISTIC UPDATE

Ví dụ bạn đang làm một trang web mạng xã hội, và bạn có thể like mỗi bài viết ở đó. Tưởng tượng rằng nếu người dùng bấm vào nút like và nó mất 2s để server trả về kết quả like của người dùng. Thật sự nó không đem lại cho người dùng cảm giác nhanh mà người dùng mong chờ. Để giải quyết vấn đề đó, chúng ta dùng Optimistic UI Update.

`Optimistic Update` là một trong những cách làm cho app chạy nhanh và mượt, tuy nhiên không phải lúc nào ta cũng nên dùng cách này, đôi khi ta cũng phải tạo cho người dụng cảm giác chậm trễ một chút. Vì thế hãy cân nhắc `Pessimistic update`

Demo

Đầu tiên ta có một component App, bên trong nó chứa một button và một state để hiện trạng thái like.

```
import React from "react";
import ReactDOM from "react-dom/client";

function App() {
  const [isLiked, setIsLiked] = React.useState(false);

  return (
    <div>
      <button>{isLiked ? "Liked" : "Like"}</button>
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.createRoot(rootElement).render(<App />);
```

Sau đó ta tạo một file tên là api.js để chứa các API. Vì demo nên mình sẽ chỉ cho nó sleep 2s rồi trả về kết quả:

```
export const likeApi = async () => {
  await new Promise((resolve) => setTimeout(resolve, 2000));

  return {
    success: true,
  };
};
```

Sau đó sửa component App, viết một function để lắng nghe sự kiện click của button

```
import React from "react";
import ReactDOM from "react-dom/client";
import { likeApi } from "./api";

function App() {
  const [isLiked, setIsLiked] = React.useState(false);

  const handleLike = async () => {
    const response = await likeApi();

    if (response.success) {
      setIsLiked(true);
    }
  };

  return (
    <div>
      <button onClick={handleLike}>{isLiked ? "Liked" : "Like"}</button>
    </div>
  );
}
```

> Lúc này app đã hoạt động đúng như mong đợi, tuy nhiên khi bấm vào nút like, ta phải đợi server trả về kết quả (2s) thì mới hiện được trạng thái đã like.

Chính vì thế, ta có thể sửa thành như sau:

```
import React from "react";
import ReactDOM from "react-dom/client";
import { likeApi } from "./api";

function App() {
  const [isLiked, setIsLiked] = React.useState(false);
  const [error, setError] = React.useState("");

  const handleLike = async () => {
    setIsLiked(true);

    const response = await likeApi();

    if (!response.success) {
      setIsLiked(false);
      setError("Something went wrong");
    }
  };

  return (
    <div>
      <p>{error}</p>

      <button onClick={handleLike}>{isLiked ? "Liked" : "Like"}</button>
    </div>
  );
}
```

> Ở function `handleLike` , thay vì đợi server trả về kết quả, ta sẽ set liked thành true. Và nếu server trả về lỗi, ta sẽ set liked thành false và hiện lỗi cho người dùng biết.

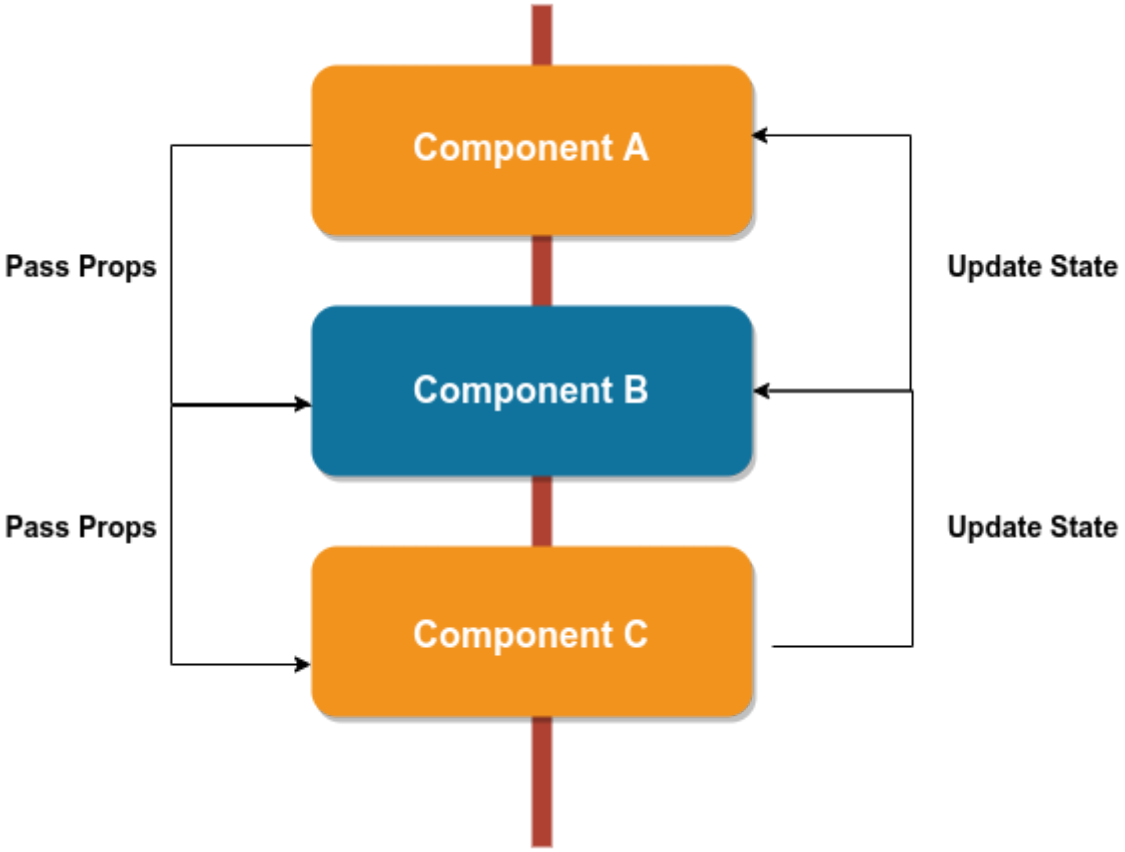
CONTEXT

Trong ứng dụng React, data thường truyền từ trên xuống dưới thông qua props. Nhưng cách này sẽ trở nên phức tạp đối với các loại dữ liệu global như locale, theme…, chúng ta phải truyền chúng qua nhiều lớp component để sử dụng. Context lúc này sẽ cung cấp một cách chia sẻ dữ liệu giữa các component như một biến global mà không cần phải truyền props qua mỗi cấp component.

Khi nào sử dụng Context

Thông thường, chúng ta sử dụng 2 khái niệm `“top-down-data-flow”` và `“Lifting state up”` để truyền dữ liệu từ component cha đến con và cập nhật dữ liệu từ component con lên cha.

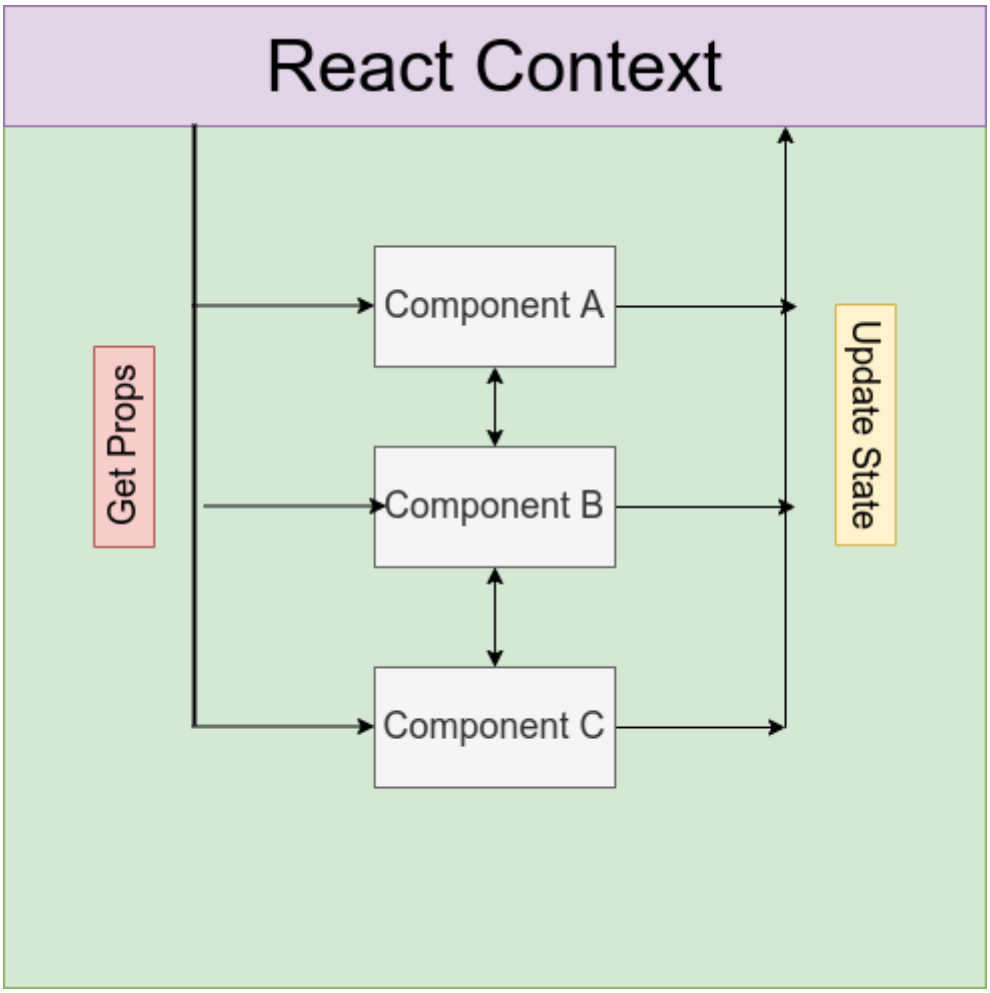
Giả sử chúng ta có 3 component A, B, C và lần lượt là con của nhau:



Nếu số lượng component lên đến hàng chục, chúng ta sẽ thấy việc truyền props như thế này sẽ trở nên dài dòng, dư code và khó quản lý. Ngay cả khi những component con trung gian không sử dụng đến props, nó cũng phải nhận props từ component cha để chuyển xuống component con của nó khi cần.

Đó là lý do mà React Context ra đời để giải quyết vấn đề nhập nhằng này!

Nguyên lý: tập trung lưu dữ liệu (props) ở một nơi. Sau đó React sẽ cung cấp các API để các component có thể lấy props đó trực tiếp mà không cần qua các component trung gian.



Ưu điểm của Context

- Giảm thiểu code dư thừa khi một component nhận props nhưng không sử dụng đến.
- Giảm thiểu việc lặp code khi gọi một props cho nhiều component.
- Quản lý dữ liệu ở một nơi giúp việc truy xuất và cập nhật dễ dàng.

Nhược điểm của Context

- Khó tái sử dụng lại component vì dữ liệu tập trung một chỗ.

CUSTOM HOOK

Custom hook là gì?

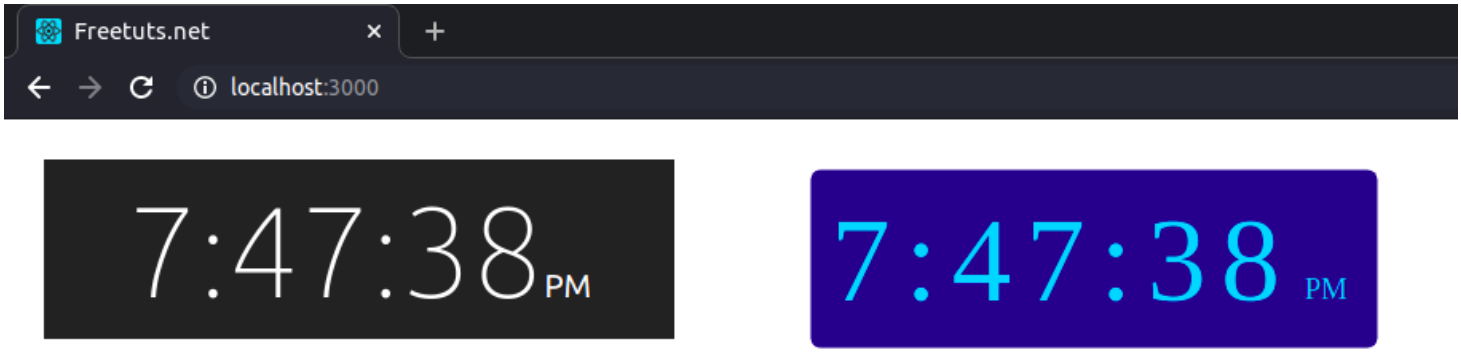
`Custom Hooks` là những hooks mà do lập trình viên tự định nghĩa với mục đích thực hiện một chức năng nào đó, nó thường được sử dụng để chia sẻ logic giữa các components.

React định nghĩa cho chúng ta các hooks như `useState`, `useEffect`, `useContext`,... cho phép chúng ta làm việc dễ dàng hơn. Để tự định nghĩa một hooks cho riêng mình, chúng ta chỉ cần xây dựng 1 hàm nhận và trả về các giá trị.

Khi nào phải `Custom hook`

`Custom Hooks` rất hay được sử dụng trong quá trình triển khai ứng dụng. Trong trường hợp chúng ta muốn tách biệt các phần xử lý logic riêng ra khỏi UI, hay chia sẻ logic giữa các component

VD: Trường hợp muốn xây dựng 2 components đều có 1 chức năng là hiển thị ngày giờ hiện tại nhưng khác nhau ở giao diện như bên dưới.



Thông thường, chúng ta sẽ viết tất cả các logic xử lý giờ ở trong component như này.

```jsx
import React, { useState } from "react";
import "./Clock1.css";
function Clock1() {
  const [time, setTime] = useState("");
  const [ampm, setampm] = useState("");
  const updateTime = function () {
    let dateInfo = new Date();
    let hour = 0;
    /* Lấy giá trị của giờ */
    if (dateInfo.getHours() === 0) {
      hour = 12;
    } else if (dateInfo.getHours() > 12) {
      hour = dateInfo.getHours() - 12;
    } else {
      hour = dateInfo.getHours();
    }
    /* Lấy giá trị của phút */
    let minutes =
      dateInfo.getMinutes() < 10
        ? parseInt("0" + dateInfo.getMinutes())
        : dateInfo.getMinutes();

    /* Lấy gía trị của giây */
    let seconds =
      dateInfo.getSeconds() < 10
        ? "0" + dateInfo.getSeconds()
        : dateInfo.getSeconds();

    /* Định dạng ngày */
    let ampm = dateInfo.getHours() >= 12 ? "PM" : "AM";

    /* Cập nhật state */
    setampm(ampm);
    setTime(`${hour}:${minutes}:${seconds}`);
  };
  setInterval(function () {
    updateTime();
  }, 1000);
  return (
    <div className="time">
      <span className="hms">{time}</span>
      <span className="ampm">{ampm}</span>
    </div>
  );
}

export default Clock1;
```

Rồi lại lặp lại logic này ở Clock2, có thể nhận thấy cách viết này không tối ưu một chút nào. Giả sử, dự án có 10 cái đồng hồ thì cần phải lặp lại logic này ở mỗi components. Điều này là không cần thiết và tốn thời gian. Giải pháp ở đây là sử dụng custom hooks.

Tự xây dựng một `custom hook`

Vậy làm sao để tự mình xây dựng một custom hooks, rất đơn giản chỉ cần tách phần xử lý logic ra một function.

Chúng ta sẽ có một custom hooks thay cho cách viết dài dòng ở ví dụ trên. Trong thư mục `src` tạo một thư mục có tên hooks, thư mục này sẽ chứa tất cả các custom hooks. Đây là một hooks có tên `useClock()` có nhiệm vụ trả về thời gian hiên tại.

```
// src/hooks/useClock.js
import { useState } from "react";

export default function useClock() {
  const [time, setTime] = useState("");
  const [ampm, setampm] = useState("");

  // Function cập nhật thời gian.
  const updateTime = function () {
    let dateInfo = new Date();
    let hour = 0;
    /* Lấy giá trị của giờ */
    if (dateInfo.getHours() === 0) {
      hour = 12;
    } else if (dateInfo.getHours() > 12) {
      hour = dateInfo.getHours() - 12;
    } else {
      hour = dateInfo.getHours();
    }
    /* Lấy giá trị của phút */
    let minutes =
      dateInfo.getMinutes() < 10
        ? parseInt("0" + dateInfo.getMinutes())
        : dateInfo.getMinutes();

    /* Lấy gía trị của giây */
    let seconds =
      dateInfo.getSeconds() < 10
        ? "0" + dateInfo.getSeconds()
        : dateInfo.getSeconds();

    /* Định dạng ngày */
    let ampm = dateInfo.getHours() >= 12 ? "PM" : "AM";

    /* Cập nhật state */
    setampm(ampm);
    setTime(`${hour}:${minutes}:${seconds}`);
  };

  setInterval(function () {
    updateTime();
  }, 1000);

  return [time, ampm];
}
```

Khi muốn sử dụng hooks này chỉ cần import nó vào component và gọi như các hooks thông thường. React dự vào tên để xem đâu là một hooks bởi vậy nên đặt tên đúng định dạng là `use + nameHooks` .

```
// src/components/Clock2.js

import React from "react";
import "./Clock2.css";
//Import Hooks
import useClock from "../hooks/useClock";
function Clock2() {
  //Gọi custom hook để sử dụng
  const [time, ampm] = useClock();

  return (
    <div id="MyClockDisplay" className="clock">
      {time}
      <span>{ampm}</span>
    </div>
  );
}

export default Clock2;
```

Vậy là chúng ta có thể sử dụng logic trong nhiều component khác nhau mà không cần lặp lại các đoạn logic phức tạp, chỉ cần import và gọi là có thể sử dụng. Với cộng đồng sử dụng rộng lớn thì ngoài các hooks có sẵn trong React, còn có các custom hooks do cộng đồng đóng góp.