實戰 Restful API and AJAX

林新德

shinder.lin@gmail.com

https://bitbucket.org/lsd0125/mfee63-node.git

https://bitbucket.org/lsd0125/mfee63-next.git

- 1. Restful API
- 2. 建立 NextJS 專案（App Router）
- 3. JSX (參考資料)
- 4. 頁面架構與規則
- 5. Hooks 基本認識 (參考資料)
- 6. State（狀態）概念 (參考資料)
- 7. 通訊錄列表
- 8. 元件生命週期 (參考資料)
- 9. 通訊錄的新增、刪除、修改
- 10. JSON Web Token
- 11. 登入使用 JWT
- 12. 加入最愛使用 JWT (參考資料)

1. Restful API

- REST為Representational State Transfer（表現層狀態轉換）的縮寫。
- 2000年由 Dr. Roy Thomas Fielding 在其博士論文中提出的 HTTP 資料交換風格。

1.1 什麼是 Restful API

- 要點：

1. 以 URI 指定資源，使用 HTTP 或 HTTPS 為操作協定。
2. 透過操作資源的表現形式來操作資料。
3. 就是以 HTTP 的 GET, POST, PUT, DELETE 方法對應到操作資源的 CRUD。
4. 資源的表現形式沒有限定，可以是 HTML, XML, **JSON** 或其它格式。
5. REST 是設計風格並**不是標準**，所以沒有硬性的規定。

- 實作 REST 的 後端 API 一般稱作 Restful API

1.2 以商品資料為說明

- 取得列表：
- http://my-domain/products (GET)
- 取得單項商品：
- http://my-domain/products/17 (GET)
- 新增商品：
- http://my-domain/products (POST)
- 修改商品：
- http://my-domain/products/17 (PUT)
- 刪除商品：
- http://my-domain/products/17 (DELETE)

1.3 管理端 URI (需要呈現表單)

- 呈現新增商品的表單：
- http://my-domain/products/add (GET)
- 呈現修改商品的表單：
- http://my-domain/products/17/edit (GET)
- 呈現刪除商品的表單：
- http://my-domain/products/17/delete (GET)

1.4 API 實作以 CRUD 的角度切入

- 商品 API：
  - 1. 列表
  - 2. 搜尋
  - 3. 單項商品

- 購物車 API：
  - 1. 讀取
  - 2.加入
  - 3.移除
  - 4.修改
  - 5.清空

2 建立 NextJS 專案（App Router）

- 依 https://nextjs.org/docs/getting-started/installation 指示安裝
- 目前的 NextJS 版本為 15.3，NodeJS 需要是 18.18 以上的版本
- 安裝最新版時，於終端機（terminal），在要放置專案資料夾的目錄路徑下，執行：

```
npx create-next-app@latest
```

- 安裝 14 版 NextJS，執行下列指令

```
npx create-next-app@14
```

```
Need to install the following packages:
create-next-app@15.1.0
Ok to proceed? (y) y

✔ What is your project named? … next-app
✔ Would you like to use TypeScript? … No / Yes
✔ Would you like to use ESLint? … No / Yes
✔ Would you like to use Tailwind CSS? … No / Yes
✔ Would you like your code inside a `src/` directory? … No / Yes
✔ Would you like to use App Router? (recommended) … No / Yes
✔ Would you like to use Turbopack for `next dev`? … No / Yes
✔ Would you like to customize the import alias (`@/*` by default)? … No / Yes
Creating a new Next.js app in /Users/shinder/practices/next-app.
```

- TypeScript **No**
- ESLint **Yes**
- Tailwind CSS **No** ← (視需求而定)
- src/ directory **No**
- App Router **Yes** ← (選 Yes, 若選 No 為 Pages Router)
- Turbopack for `next dev` **Yes** ← (15版才有的選項, 14版之前的版本沒有)
- Customize the import alias **No**

```
// 修改設定檔: next.config.mjs

/** @type {import('next').NextConfig} */
const nextConfig = {

  // 嚴謹模式更改為 false
  reactStrictMode: false,

  // 在 SSR 模式可以設定環境變數
  env: {
    API_SERVER: 'http://localhost:3001'
  }
};
export default nextConfig;
```

3. JSX (參考資料)

- JSX 為 JS 延伸的語法，用來轉換成 DOM 的元素，本質上依然是 JS 不是 HTML。
- JSX 為 React.createElement() 的語法糖。
- JSX 通常視為 JavaScript XML 的簡稱，顧名思義 JSX 必須符合 XML 的規定。
- 由於 JSX 語法貼近 HTML，方便用來描述 HTML。
- JSX 的目的是呈現 HTML 頁面的內容，和呈現無關的操作不應該放在 JSX 裡。
- 會自動做 HTML 跳脫（HTML escape）。

JSX 的優點

參考來源：React Quickly, Manning Publications

1. 改善開發人員的體驗，和 React.createElement() 相比較。
2. 較好的錯誤訊息。
3. 快速的執行速度，JSX 轉換成 JS 語法後，執行速度依然很快速。
4. 非專業的程式設定師或視覺設計師也容易參與編輯。
5. 較少的錯誤發生，程式碼越少表示錯誤越少。

```
const el = <Card>
  <Title>Hello</Title>
  <Link href="/article/1">Article 1</Link>
</Card>;
```

```
const el = React.createElement(
  Card,
  null,
  React.createElement(Title, null, 'Hello'),
  React.createElement(Link, {href: "/article/1"}, 'Article 1'),
);
```

JSX 常用規則

1. 只能有一個根節點。
2. 有起始標籤，也要有結束標籤，若為空元素時（沒有子節點），使用空元素的表示方式。
3. 可以使用在 JSX 檔（JS 檔）的任何位置，視為一個類型的物件。
4. 自訂標籤（元件），必須大寫字母開頭如：Card、MyCard。
5. 不可以使用到 JS 關鍵字，如：for、class，應以 htmlFor 和 className 取代。
6. 標籤的屬性必須使用 Camel 的方式表達（aria- 開頭者為例外）。
7. 標籤的屬性值的大括號表示綁定值或參照，可以是 JS 的任何類型物件。
8. 屬性沒有設定值時，表示綁定 true。
9. 行內 style 屬性必須綁定 Object 類型的物件。
10. 標籤之間的大括號表示要**輸出**成 jsx 內容的運算區塊。
11. 標籤之間若換行，則生成 html 時，兩標籤中間將不會有空白。

```
// 規則 1. 只能有一個根節點。
// 規則 2. 若為空元素時（沒有子節點），使用空元素的表示方式。
// 規則 3. 可以使用在 JSX 檔（JS 檔）的任何位置，視為一個類型的物件。

const myForm = <form>
  <img src="/img/a.png" alt="img1" />
  <hr/>
  <input type="text" name="account" />
  <br/>
  <input type="submit" name="送出"/>
</form>;
```

```
// 規則 5. 不可以使用到 JS 關鍵字，如：for、class，應以 htmlFor 和 className 取代。
// 規則 6. 標籤的屬性必須使用 Camel 的方式表達（aria- 開頭者為例外）。
// 規則 8. 屬性沒有設定值時，表示綁定 true。

const jsx = <form>
  <div className="mb-3">
    <label htmlFor="email" className="form-label">Email address</label>
    <input type="email"
           className="form-control"
           name="email"
           aria-describedby="emailHelp"
           required
    />
  </div>
  <button type="submit" className="btn btn-primary">Submit</button>
</form>
```

```
// 規則 7. 標籤的屬性值的大括號表示綁定值或參照，可以是 JS 的任何類型物件。

const dataObj = {name: "David", age: 25};
const jsx2 = <form onSubmit={(e) => e.preventDefault()}>
  <div className="mb-3">
    <label htmlFor="email" className="form-label">Email address</label>
    <input type="email"
           className="form-control"
           name="email"
           value={myState}
           onChange={(e) => setMyState(e.target.value)}
    />
    <MyInput {...dataObj} myValue={dataObj}/>
  </div>
  <button type="submit" className="btn btn-primary">Submit</button>
</form>
```

```jsx
// 規則 10. 標籤之間的大括號表示要輸出成 jsx 內容的運算區塊。

const dataObj = {name: "David", age: 25};
const jsx3 = <ul>
  {
    Object.keys(dataObj).map(el => {
      return <li key={el}>{el}: {dataObj[el]}</li>;
    })
  }
</ul>
```

JSX 流程控制的轉換用法：

1. if 敘述使用 **&&**。
2. if/else 敘述使用**三元運算子**。
3. 迴圈敘述使用**陣列的 map() 方法**。
4. switch/case敘述則無對應的方式，可以採用 Object 物件 key-value 對應的性質。

JSX 裡的類似 for 迴圈的作法

```
{
  Array(11)
   .fill(1)
   .map(
      (v, i) => {}
   )
}
```

```
{
  [...Array(11)].map(
    (v, i) => {}
  )
}
```

4. 頁面架構與規則

- App router 的架構是以 app 資料夾為頁面起始資料夾。
- 裡面的 layout.js 為頁面樣版檔（格局檔），page.js 為頁面元件。
- /app/page.js 是 /app/layout.js 的子元件。
- 資料夾名稱為路徑的一部份，app/products/page.js 的 url 為 /products/
- 子層的 layout.js 為父層layout.js 的子元件。
- 每個 layout.js 和 page.js 都可以有自己的 metadata。
- 若要放 cdn 的 css 可以放在 /app/layout.js 裡（不建議使用 cdn）。
- 若要放 cdn 的 js 也可以放在 /app/layout.js 裡（不建議使用 cdn）。

4.1 CSS CDN 設定在 layout.js

```
export default function RootLayout({ children }) {
  return (
    <html lang="zh">
      <head>
        <link
          rel="stylesheet"
          href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css"
        />
      </head>
      <body>
        {children}
        <script
          src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js"
          defer
        ></script>
      </body>
    </html>
  );
}
```

4.2 NextJS App Router 路由架構

- App router 以 app/ 為根目錄。

檔案路徑	對應的 URL
app/login/page.js	http://localhost:3000/login
app/products/page.js	http://localhost:3000/products
app/products/[pid]/page.js	http://localhost:3000/products/17
app/blog/[...slug2]/page.js	http://localhost:3000/blog/news/taiwan
app/(my_group)/hello/page.js	http://localhost:3000/hello

4.3 SSR 和 CSR

- 安裝 Chrome 擴充功能「Page Source Kitty」可查看 HTML 原始內容。
- 注意頁面來源：
  - 什麼狀況頁面由 Server-Side Rendering (SSR) 來？
  - 什麼狀況頁面由 Client-Side Rendering (CSR) 來？
- 以 'use client' 標示的元件也可能在 Server-Side render。
- SSR 時不要在元件中直接使用前端 DOM 相關物件或功能 (或在 render 的 JSX 裡使用)。例如，在元件內不要直接使用 window 物件，若要使用應該放在 useEffect() 的 callback 內使用，以確保是在前端執行。
- SSR 元件的子元件可以是 CSR 元件（父元件先 render，之後再 render 子元件）；反之可能會有問題。
- 練習: 舉個在頁面元件中直接使用 window.location 會造成問題的例子。

4.4 URL 上的資料

- React hooks 只能用在前端元件（元件檔前標示 `'use client'`）。
- useRouter() 用來取得 router 物件，用以程式的跳轉頁面，如 push() 方法。
- usePathname() 用來取得 url 裡的路徑（字串）。
- useParams() 用來取得一物件，其中包含路徑變數（動態路由）的資料。
- useSearchParams() 用來取得一物件（ReadonlyURLSearchParams）其中包含 query string 參數的資料。使用 useSearchParams() 的元件應該使用 Suspense 包裹。
- 以上四個勾子都屬於 next/navigation 模組。

```
// 檔案: app/products/[pid]/page.js
// URL: http://localhost:3000/products/12?name=david&age=25
"use client";
import { useParams, usePathname, useRouter, useSearchParams } from "next/navigation";

const ProductItemPage = () => {
  const router = useRouter();
  const params = useParams();
  const searchParams = useSearchParams();
  const pathname = usePathname();
  console.log({ router, params, searchParams, pathname });
  return (
    <div>
      <p>{pathname}</p>
      <p>{params.pid}</p>
      <p>{searchParams.get("name")}</p>
    </div>
  );
};
export default ProductItemPage;
```

4.5 API URL 可放在設定檔 /config/api-path.js

```
export const API_SERVER = `http://localhost:3001`;
// export const API_SERVER = `http://172.18.103.106:3001`;

// 取得通訊錄列表資料
export const AB_LIST = `${API_SERVER}/address-book/api`;

// 新增通訊錄資料 method: POST
export const AB_ADD_POST = `${API_SERVER}/address-book/api`;

// 刪除通訊錄項目 method: DELETE
// `${API_SERVER}/address-book/api/${ab_id}`
export const AB_DEL_DELETE = `${API_SERVER}/address-book/api`;
```

```
// 讀取單筆通訊錄項目 method: GET
// `${API_SERVER}/address-book/api/${ab_id}`
export const AB_GET_ONE = `${API_SERVER}/address-book/api`;

// 修改單筆通訊錄項目 method: PUT
// `${API_SERVER}/address-book/api/${ab_id}`
export const AB_ITEM_PUT = `${API_SERVER}/address-book/api`;

// **** JWT 登入, method: POST
export const JWT_LOGIN_POST = `${API_SERVER}/login-jwt`;

// **** JWT LIKE, method: POST
// 路徑 `/address-book/api/jwt-like/${ab_id}`
export const JWT_LIKE_POST = `${API_SERVER}/address-book/api/jwt-like`;
```

5. Hooks 基本認識 (參考資料)

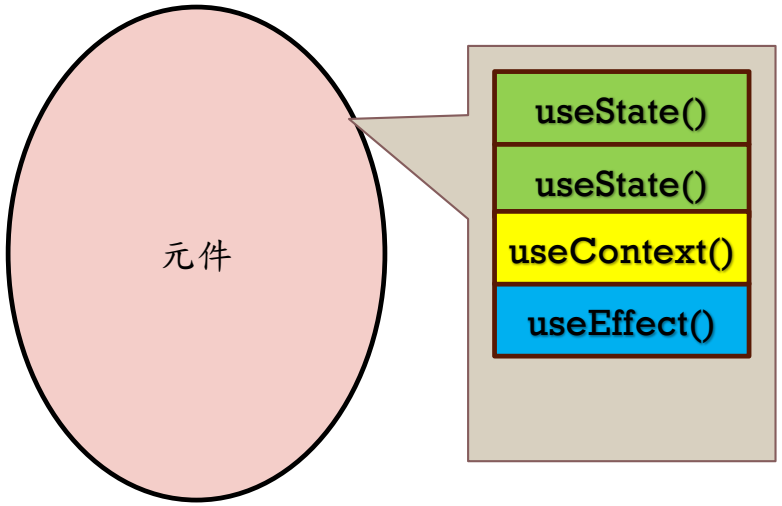
- Hooks（勾子）是因應 function components 語法而生的。
- Components 使用 function 寫法時，基本上它就是一個 function，主要功能就是呈現（render）內容，沒有別的功能。
- 要像 class-based 元件有其它功能時，就必須借由 hooks 來賦予。
- 勾子的目的就是從 React 核心架構中勾一個功能到函式（元件）中來使用。
- Hooks 的名稱必須是以 **use** 為開頭，使用 Camel 的命名方式。
- Hooks 本身也是函式，可以組合基本的 hooks 來加以變化使用。
- Hooks 不可以放在 if/else 或者迴圈中使用。Hooks 使用時是跟著元件，並放在一個陣列中記錄，每次 render 時順序都必須一樣。
- Hooks 方便單元測試。

```jsx
// Hooks 錯誤的使用方式
import {useState} from "react";

function MyButton() {
  if (Math.random() < .5) {
    return null;
  }
  const [val, setVal] = useState(0)
  return (
    <button>
      I'm a button {val}
    </button>
  );
}
```

Lint Error

React Hook "useState" is called conditionally. React Hooks must be called in the exact same order in every component render. Did you accidentally call a React Hook after an early return?

元件使用 Hooks 示意圖



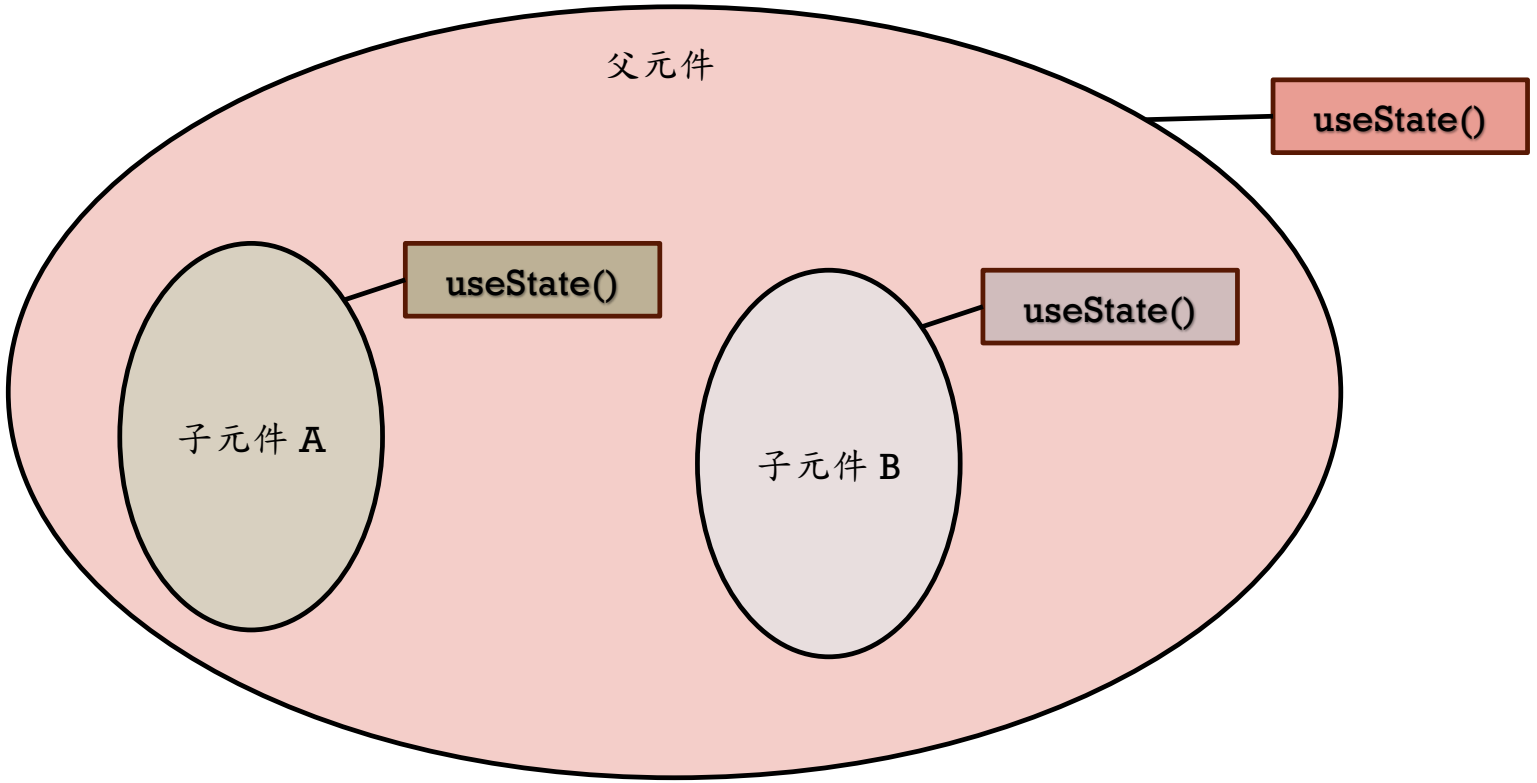
使用 Hooks 的元件，相當於把使用的工具背在身上。
元件更新時（re-render），並不會影響 hooks 本身。

6. State（狀態）概念 (參考資料)

- useState() 將取得一個包含兩個參照的陣列，一個為**值**（getter），另一個為**變更函式**（setter）。
- **值**為狀態，視為「**唯讀**」，不可變更。（底層可能是使用 Proxy 實作的 getter）
- 變更狀態必須使用「**變更函式**」，而且其操作為**非同步**。意即呼叫**變更函式**後，值並沒有立即改變，而是必須等下次 render 時，**值**才能被觀察到變更。
- 狀態是跟在元件上的特殊屬性，當 state 變更時，表示元件需要更新（update），意即會觸發元件的 re-render 。
- 元件為函式時，re-render 相當於重新呼叫一次函式。函式內的所有區域變數將會被重新設定，包含存取狀態的**值**和**變更函式**。
- 狀態的**值**和**變更函式**，是由勾子處理，並不是在函式中處理。所以更新時，雖然**值**和**變更函式**被重新設定，但依然可以保有應有的狀態。

子元件 A 更新時的兩種情況

1. 子元件 A 自身的狀態改變
2. 父元件的狀態改變，以重新 render 子元件 A



使用狀態時應注意的事項

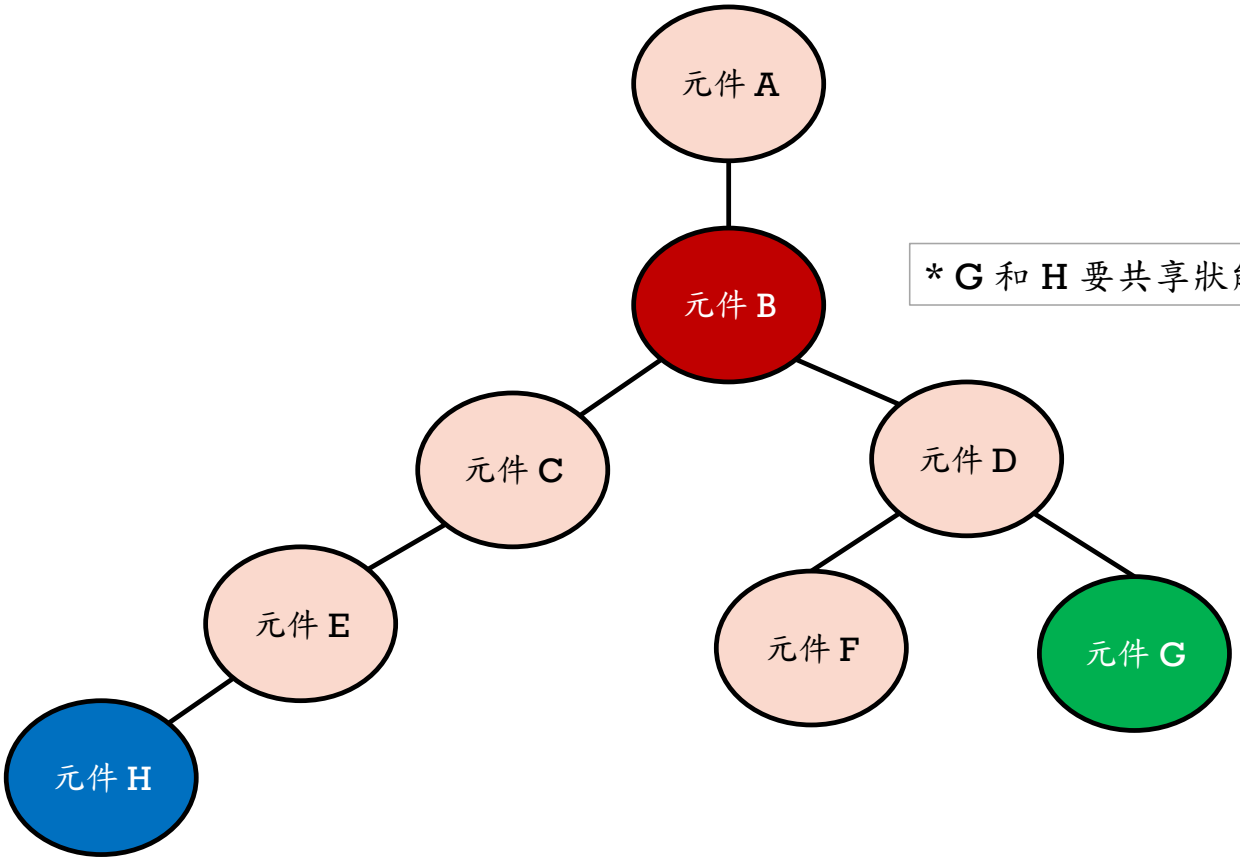
- 父元件的狀態和子元件的狀態是各自獨立的。
- 各元件的狀態是各自獨立的。
- **單向資料流**，只有父元件傳屬性給子元件，沒有子元件傳給父元件。
- 兩元件要溝通時，應將狀態設定在「共同上層元件」上。
- 共同上層元件的「狀態變更函式」往下一層一層傳給下游元件，讓該元件有能力直接變更共同上層元件的狀態，以瀑布更新的方式，讓下游元件更新。
- 要在全域共享資料時，狀態應該設定在最頂層元件（App 或 _app 或 layout.jsx）。
- 越頂層的狀態，若變更太頻繁可能造成效能不佳的情況。
- Context.Provider 中的狀態不應該有太頻繁的變更。

* 父元件狀態改變時會 **render** 子元件，子元件（函式）會被呼叫。

```
// app/render-try1/page.js
"use client";
import React, { useState } from "react";
import ChildA from "@/components/common/child-a";

export default function RenderTry1() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <h2>{count}</h2>
      <button onClick={() => setCount(count + 1)}>click</button>
      <br />
      <ChildA name="第一個" />
      <ChildA name={`sec: ${count}`} />
    </div>
  );
}
```

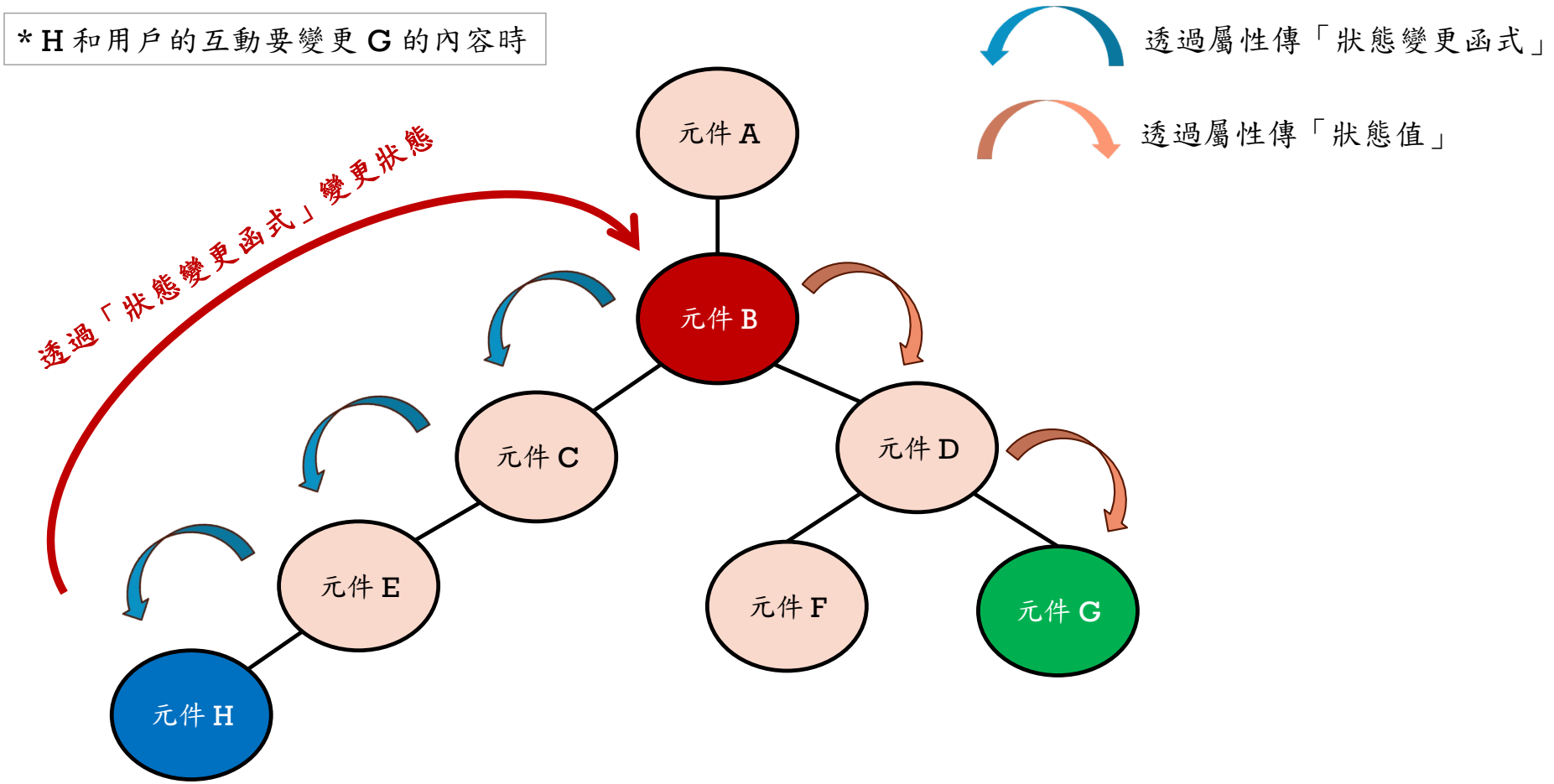
```
// components/common/child-a.js
"use client";
export default function ChildA({ name = "" }) {
  console.log({ name });

  return <div>ChildA: {name}</div>;
}
```

- 在測試 http://localhost:3000/render-try1 之前
- 請先將 next.config.mjs 裡的 reactStrictMode 設定成 false，以避免影響觀察。
- 父元件透過 render 子元件（相當於呼叫子元件）而把 props 傳給它。



* G 和 H 要共享狀能時應該設定在 B

*H 和用戶的互動要變更 G 的內容時



7. 通訊錄列表功能

- API Server 使用原 NodeJS 課程中的 Node/Express 伺服器

7.1 列表頁的狀態

```
// app/address-book/page.js
"use client";
import { useEffect, useState } from "react";
import { AB_LIST, AB_DEL_DELETE } from "@/config/api-path";
import Link from "next/link";
import { useRouter, useSearchParams } from "next/navigation";
import { FaRegTrashCan, FaRegPenToSquare } from "react-icons/fa6";
export default function ABListPage() {
  const [refresh, setRefresh] = useState(false);  // 為了刪除項目時觸發 re-render
  const router = useRouter();
  // 取得 URL 中的 query string
  const searchParams = useSearchParams();
  // 存放載入進來的資料的狀態
  const [listData, setListData] = useState({
    totalPages: 0,
    totalRows: 0,
    page: 0,
    rows: [],
  });
```

7.2 取得列表資料

```
useEffect(() => {
  fetch(`${AB_LIST}${location.search}`)
    .then((r) => r.json())
    .then((obj) => {
      // api 回應的資料中，success 為 true 時，才更新 state
      if (obj.success) {
        setListData(obj);
      } else if (obj.redirect) {
        router.push(obj.redirect);
      }
    })
    .catch(console.warn);
}, [searchParams, refresh]);
```

7.3 表格呈現內容

```
<tbody>
  {listData.rows.map((v, i) => {
    return (
      <tr key={i}>
        <td>{v.sid}</td>
        <td>{v.name}</td>
        <td>{v.email}</td>
        <td>{v.mobile}</td>
        <td>{v.birthday}</td>
        <td>{v.address}</td>
      </tr>
    );
  })}
</tbody>
```

7.4 分頁按鈕

```
<ul className="pagination">
  {Array(11)
    .fill(1)
    .map((v, i) => {
      let p = listData.page - 5 + i;
      if (p < 1 || p > listData.totalPages) return null;
      const addActive = searchParams.get("page") == p ? "active" : "";
      return (
        <li className={`page-item ${addActive}`} key={p}>
          <Link className="page-link" href={`?page=${p}`}>
            {p}
          </Link>
        </li>
      );
    })}
</ul>
```

7.5 有資料才呈現表格

```
<>
  {listData.rows.length ? (
    <>
      {/* 呈現表格 */}
    </>
  ) : (
      <h5>沒有資料</h5>
  )}
</>
```

7.6 後端回應延遲模擬競爭的 AJAX

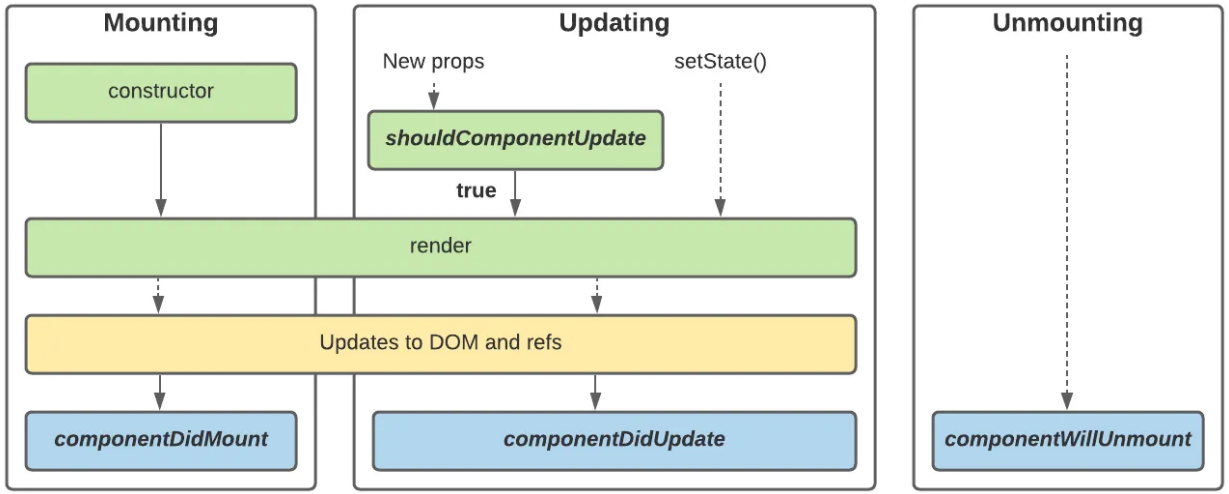
```
// ExpressJS 自訂的頂層的 middlewares 內

// ****** 測試用，模擬 2 sec 內的延遲 ***** 測試完記得註解掉
const waitMSec = Math.random() * 2000;
setTimeout(() => {
  next();
}, waitMSec);
```

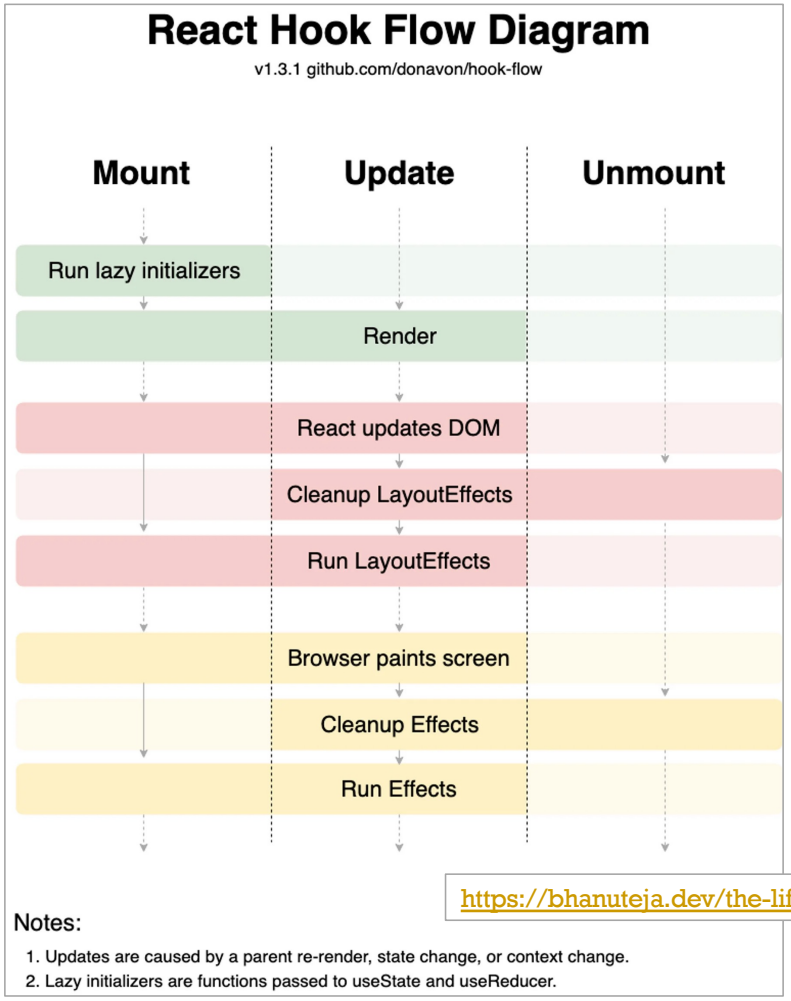
** 可快速點擊分頁按鈕測試，此時可以看到 **effect cleanup** 的功能作用。

7.7 使用 AbortController 避免 AJAX 競爭問題

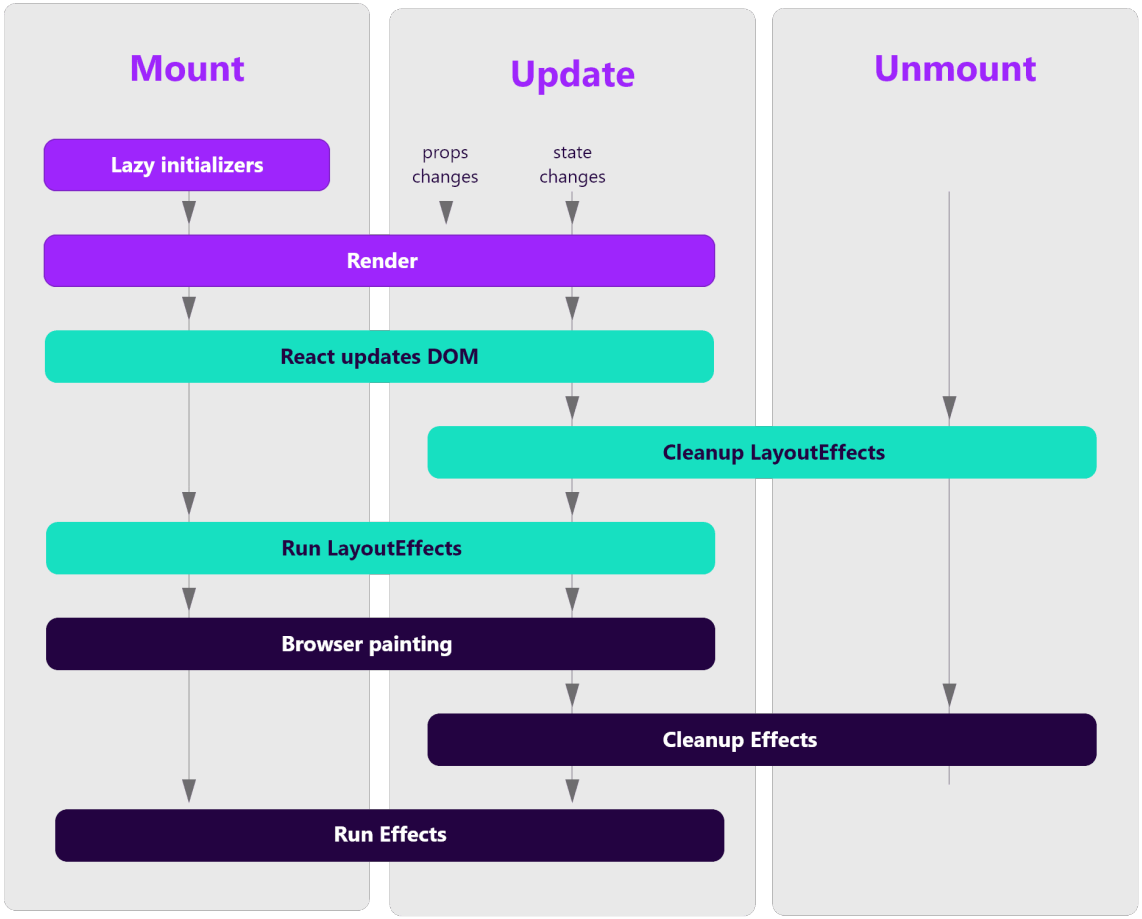
```
useEffect(() => {
  const controller = new AbortController();  // 用來取消的控制器
  const signal = controller.signal;
  fetch(`${AB_LIST}${location.search}`, {
    signal,
  })
    .then((r) => r.json())
    .then((obj) => {
      if (obj.success) {
        setListData(obj);
      } else if (obj.redirect) {
        router.push(obj.redirect);
      }
    })
    .catch(console.warn);  // 用戶取消時會發生 exception
  return () => controller.abort();  // 取消未完成的 ajax
}, [searchParams, refresh]);
```

8. 元件生命週期（參考資料）





https://bhanuteja.dev/the-lifecycle-of-react-hooks-component



https://hello-js.com/articles/react-class-and-hooks-lifecycle-explained/

8.1 useEffect() 的 Cleanup

```
// app/cleanup/page.js
"use client";
import {useState} from "react";
import CleanupChild1 from "@/components/cleanup-child-1";
export default function CleanUpPage() {
  const [showChild, setShowChild] = useState(false);
  return (
    <div>
      <div>
        <button onClick={() => setShowChild(!showChild)}>toggle</button>
      </div>
      {showChild ? (
        <>
          <CleanupChild1/>
        </>
      ) : null}
    </div>
  );
}
```

```
// components/cleanup-child-1.js
import {useEffect, useState} from "react";

export default function CleanupChild1() {
  const [num, setNum] = useState(0);
  useEffect(() => {
    console.log("CleanupChild1 已掛載");

    const interval_id = setInterval(() => {
      // console.log({ num }); // 因為 closure, num 只會拿到 0
      setNum((old) => old + 1);
    }, 500);

    return () => {
      console.log("CleanupChild1 將要卸載");
      clearInterval(interval_id); // 把計時器停下來
    };
  }, []);
  return <div>CleanupChild1 {num}</div>
}
```

```
// components/cleanup-child-2.js
import React, {useEffect, useRef, useState} from “react”;
export default function CleanupChild2() {
  const myRef = useRef();
  let n = 0; // 此元件沒有更新 (re-render)

  useEffect(() => {
    console.log("CleanupChild2 已掛載");
    const interval_id = setInterval(() => {
      n++;
      myRef.current.innerHTML = n;
    }, 500);
    return () => {
      console.log("CleanupChild2 將要卸載");
      clearInterval(interval_id); // 把計時器停下來
    };
  }, []);

  return <div>CleanupChild2 <span ref={myRef}>0</span></div>
}
```

8.2 模擬 didMount, willUnmount, didUpdate

```
// *** 2-1 **** components/cleanup-child-3.js
"use client";
import { useEffect, useState } from "react";

export default function CleanupChild3() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log("CleanupChild-3: 已掛載");
    return () => {
      console.log("CleanupChild-3: 將要卸載");
    };
  }, []);
```

```
// *** 2-2 **** components/cleanup-child-3.js
useEffect(() => {
  console.log("CleanupChild-3: [count 已更新] + [已掛載]");
  return () => {
    console.log("CleanupChild-3: [清除上次 count 更新時的設定] + [將要卸載]");
  };
}, [count]);

return (
  <div>
    <h2>CleanupChild-3</h2>
    <p>
      <button onClick={() => setCount(count + 1)}>click</button>
    </p>
    <p>{count}</p>
  </div>
);
}
```

9. 通訊錄的新增、刪除、修改

9.1 新增資料頁的表單（使用可控表單，部份內容）

```
<h5 className="card-title">新增資料</h5>
<form name="form1" method="post" onSubmit={submitHandler}>
  <div className={"mb-3 " + (errors.name ? styles.error : "")}>
    <label htmlFor="name" className="form-label">
      姓名
    </label>
    <input
      type="text"
      className="form-control"
      id="name"
      name="name"
      value={myForm.name}
      onChange={changeHandler}
    />
    <div className="form-text">{errors.name}</div>
  </div>
```

9.2 新增資料頁的狀態

```
const router = useRouter();

const [myForm, setMyForm] = useState({
  name: "",
  email: "",
  mobile: "",
  birthday: "",
  address: "",
});

// 呈現錯誤訊息的狀態
const [errors, setErrors] = useState({
  name: "",
  email: "",
  mobile: "",
});
```

9.3 新增資料頁的輸入事件處理器

```
const onChange = (e) => {
  // console.log(e.target.name, e.target.value);

  const newForm = { ...myForm, [e.target.name]: e.target.value };
  // console.log(newForm);

  setMyForm(newForm);
};
```

9.4 新增資料頁的表單發送前檢查

```
const onSubmit = async (e) => {
  e.preventDefault();

  // 使用 zod 驗證表單
  const schemaForm = z.object({
    name: z.string().min(2, { message: "姓名至少兩個字" }),
    email: z.string().email({ message: "請填寫正確的電郵格式" }),
    mobile: z
      .string()
      .regex(/09\d{2}-?\d{3}-?\d{3}/, { message: "請填寫正確的手機格式" }),
  });

  const result2 = schemaForm.safeParse(myForm);
  // console.log(JSON.stringify(result2, null, 4));
```

```
// 重置 myFormErrors
const newFormErrors = {
  name: "",
  email: "",
  mobile: "",
};
if (!result2.success) {
  if (result2?.error?.issues?.length) {
    for (let issue of result2.error.issues) {
      newFormErrors[issue.path[0]] = issue.message;
    }
    setMyFormErrors(newFormErrors);
  }
  return; // 表單資料沒有通過檢查就直接返回
}
// 走到這邊表示, 表單有通過驗證
```

9.5 新增資料頁的表單發送

```
try {
  const r = await fetch(AB_ADD_POST, {
    method: "POST",
    body: JSON.stringify(myForm),
    headers: {
      "Content-Type": "application/json",
    },
  });
  const result = await r.json();
  console.log(result);
  if (result.success) {
    router.push(`/address-book`); // 跳頁
  } else {
  }
} catch (ex) {
  console.log(ex);
}
```

9.6 列表頁中刪除資料的圖示

```
{listData.rows?.map((v, i) => {
  return (
    <tr key={i}>
      <td>
        <a
          href="#"
          onClick={(e) => {
            e.preventDefault(); // 避免刷新頁面
            delItem(v.ab_id);   // 呼叫刪除項目的函式
          }}
        >
          <FaRegTrashCan />
        </a>
      </td>
```

9.7 列表頁中刪除資料的函式

```
const delItem = async (ab_id) => {
  console.log({ ab_id });
  try {
    const r = await fetch(`${AB_DEL_DELETE}/${ab_id}`, {
      method: "DELETE",
    });
    const result = await r.json();
    if (result.success) {
      setRefresh((v) => !v);  // 變更狀態，重新載入資料
    }
  } catch (ex) {}
};
```

9.8 列表頁中編輯資料的圖示

```
<td>
  <Link href={"/address-book/" + v.ab_id}>
    <FaRegPenToSquare />
  </Link>
</td>
```

9.9 編輯資料頁（從新增資料頁複製過來修改）

```
const params = useParams();
useEffect(() => {
  // 讀取欲編輯的資料項目
  const ab_id = +params.ab_id;
  if (!ab_id) {
    router.push("/address-book"); // 回列表頁
  }
  fetch(`${AB_GET_ONE}/${ab_id}`)
    .then((r) => r.json())
    .then((obj) => {
      if (obj.success) {
        delete obj.data.created_at; // 去掉屬性
        setMyForm(obj.data);
      } else {
        router.push("/address-book"); // 回列表頁
      }
    });
}, [params.ab_id, router]);
```

9.10 表單上呈現失能的項目（提示用戶資料編號的值）

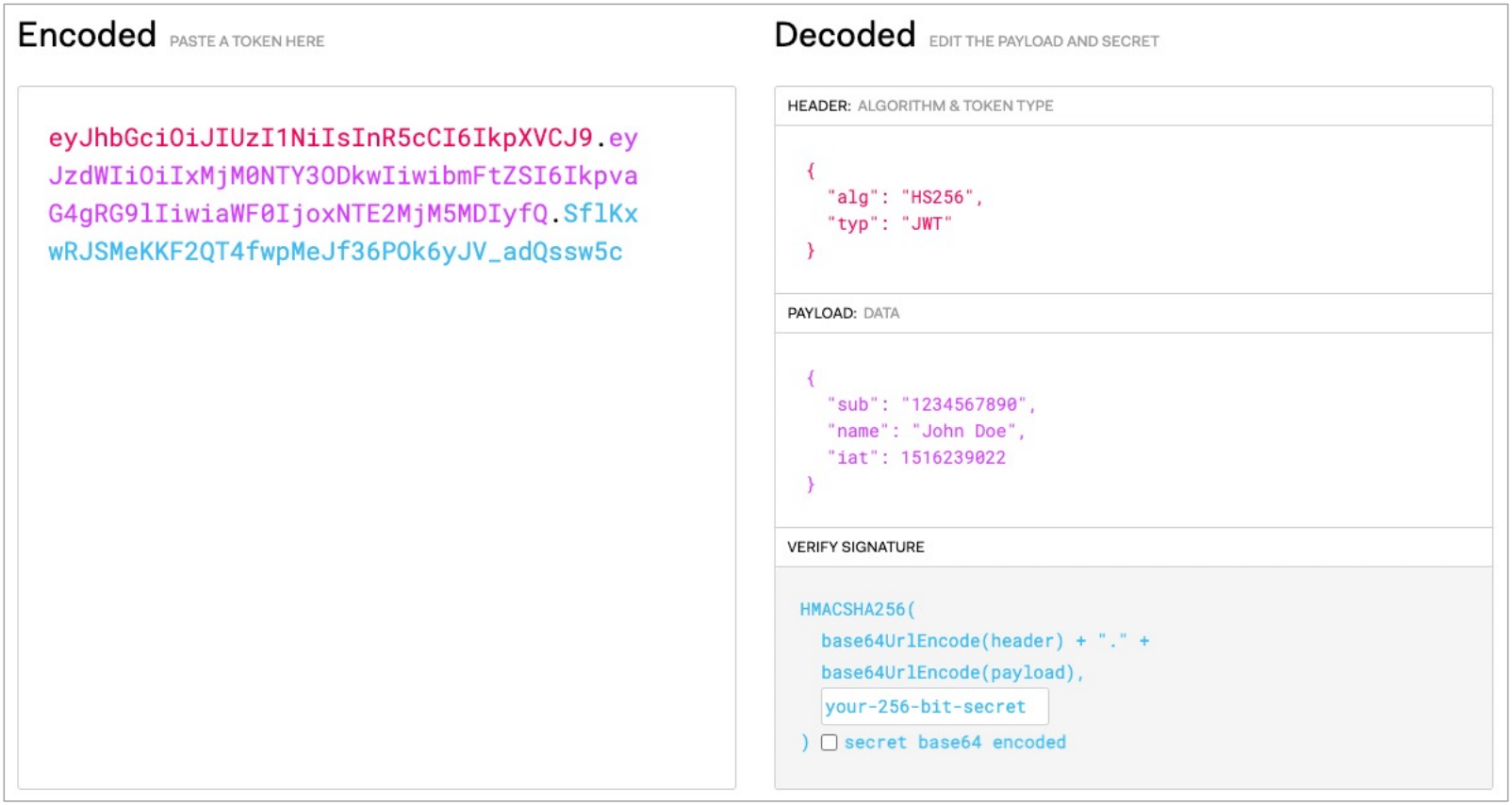
```
<div className="mb-3">
  <label htmlFor="" className="form-label">
    編號
  </label>
  <input
    type="text"
    className="form-control"
    value={myForm.ab_id}
    disabled
  />
</div>
```

9.11 送出修改的表單

```
const onSubmit = (e) => {
  e.preventDefault();  // TODO: 欄位檢查
  fetch(`${AB_ITEM_PUT}/${params.ab_id}`, {
    method: "PUT",
    body: JSON.stringify(myForm),
    headers: {
      "Content-Type": "application/json",
    },
  })
    .then((r) => r.json())
    .then((obj) => {
      if (obj.success) {
        alert("修改成功");
        router.back();
      } else {
        alert("資料沒有修改");
      }
    });
};
```

10. JSON Web Token

- JSON Web Token 社群官網 https://jwt.io/
- 使用 https://www.npmjs.com/package/jsonwebtoken 套件。
- 先決條件：資料傳送過程必須在加密的環境中使用，如 HTTPS
- 優點：可在不同的用戶端環境使用，不局限於網站。
- 缺點：需存放在用戶端，由 JavaScript 發送，或其他前端技術發送。
- 過期時間 exp，也可以使用套件的設定 expiresIn。
- （使用 bcrypt 套件加密用戶密碼）


Encoded
PASTE A TOKEN HERE
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiaWF0IjoxNTE2MjM5MDIyfQ.SflKxwRJSMeKKF2QT4fwpMeJf36POk6yJV_adQssw5c
Decoded
EDIT THE PAYLOAD AND SECRET
HEADER: ALGORITHM & TOKEN TYPE
{
"alg": "HS256",
"typ": "JWT"
}
PAYLOAD: DATA
{
"sub": "1234567890",
"name": "John Doe",
"iat": 1516239022
}
VERIFY SIGNATURE
HMACSHA256(
base64UrlEncode(header) + "." +
base64UrlEncode(payload),
your-256-bit-secret
) secret base64 encoded

10.1 JWT 編碼解碼

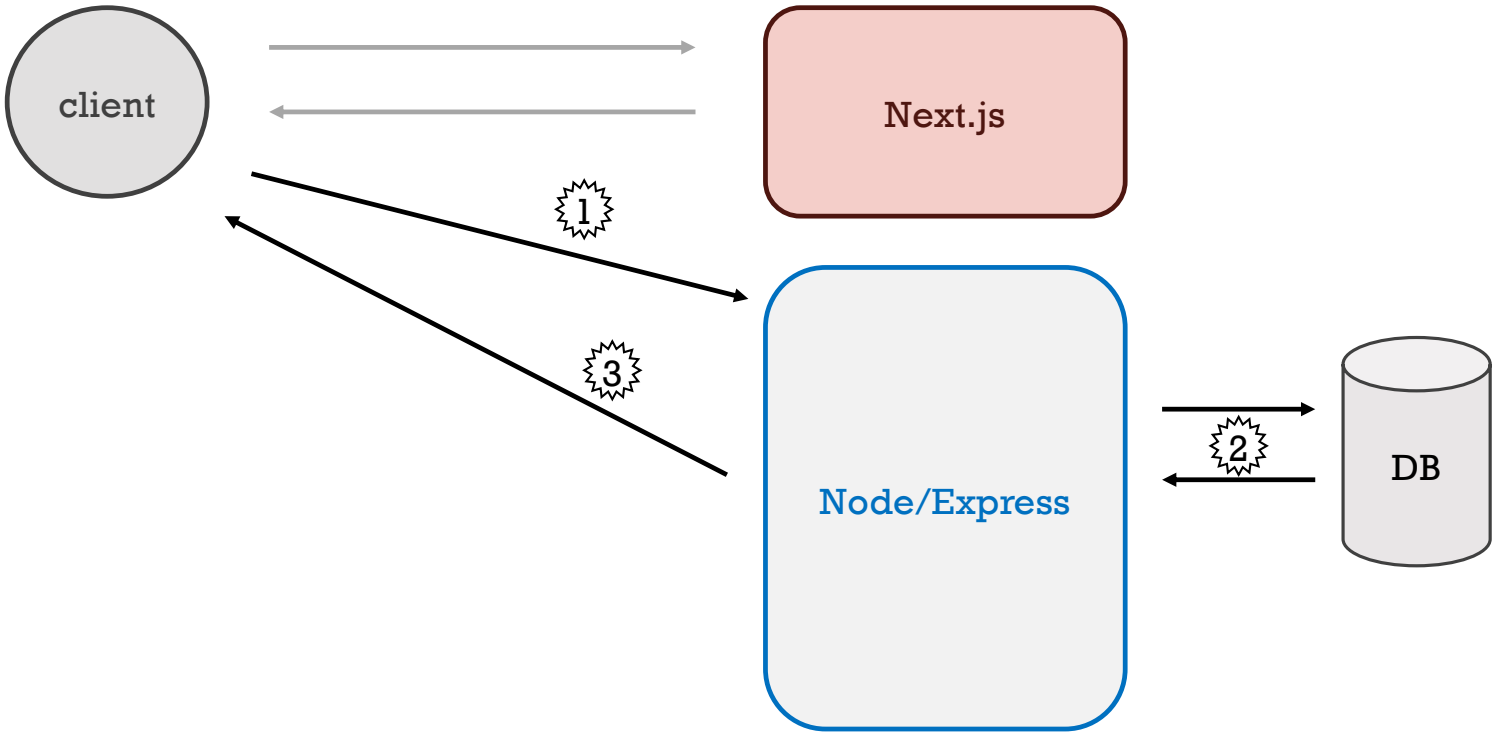
```
// 在 NodeJS 環境測試

import jwt from "jsonwebtoken";
const JWT_KEY = "kjdgdk3453JYUGUYG57438"; // 自訂的密碼

// 將可以辨別用戶的資料加密為 JWT（編碼）
const data = {
  id: 26,
  account: "Shinder",
};
// 加密為 token
const token = jwt.sign(data, JWT_KEY);
console.log({token});
```

```
// 在 NodeJS 環境測試

import jwt from "jsonwebtoken";

const JWT_KEY = "kjdgdk3453JYUGUYG57438"; // 自訂的密碼

// 接收到的 token
const token =
'eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MjYsImFjY291bn
QiOiJTaGluZGVyIiwiaWF0IjoxNzExMzMwMzE1fQ.YkdiG6utKCyhRAQHK4f
04YQ7nuxM9e0GHRNe61rrCcQ';

// 解密成為 JavaScript plain object
const payload = jwt.verify(token, JWT_KEY);
console.log(payload);
```

11. 登入使用 JWT

- 1. 用戶在前端登入頁面發送 account 和 password 資料。
- 2. 後端接收資料後比對帳號和密碼，若正確則往下一步。
- 3. 後端將用以識別用戶的資料包在 JWT 內，並送至前端。
- 4. 前端收到相關資料後存至 localStorage，並改變 AuthContextProvider 狀態，完成登入狀態。
- 5. 用戶刷頁面時（refresh），AuthContextProvider 會先判斷 localStorage 的 token 相關資料，是否為已登入的狀態。更好的作法，是讀取 token 後，再發 Ajax 確認 token 是否有效。

- ** 如果架構允許，token 存放在 http-only 的 cookie 裡，會比存放在 localStorage 裡安全，可以避免 XSS 攻擊。
- ** 在前後端分離的開發環境，token 存放在 localStorage 是比較容易實現的作法。


client
Next.js
1
3
Node/Express
2
DB

11.1 後端登入服務

```
app.post("/login-jwt", upload.none(), async (req, res) => {
  const output = {
    success: false,
    code: 0,
    error: "",
    bodyData: req.body,
    data: {}, // 傳給用戶端, 存到 localStorage
  };
  let { email, password } = req.body;
  email = email ? email.trim() : "";
  password = password ? password.trim() : "";
  // 0. 兩者, 若有一個沒有值就結束
  if (!email || !password) {
    output.error = "欄位資料不足";
    return res.json(output);
  }
```

```
// 1. 先確定帳號是不是對的
const sql = `SELECT * FROM members WHERE email=?`;
const [rows] = await db.query(sql, [email]);
if (!rows.length) {
  // 帳號是錯的
  output.code = 400;
  output.error = "帳號或密碼錯誤";
  return res.json(output);
}
```

```
// 2. 確定密碼是不是對的
const result = await bcrypt.compare(password, row.password_hash);
if (!result) {
  // 密碼是錯的
  output.code = 450;
  output.error = "帳號或密碼錯誤";
  return res.json(output);
}
```

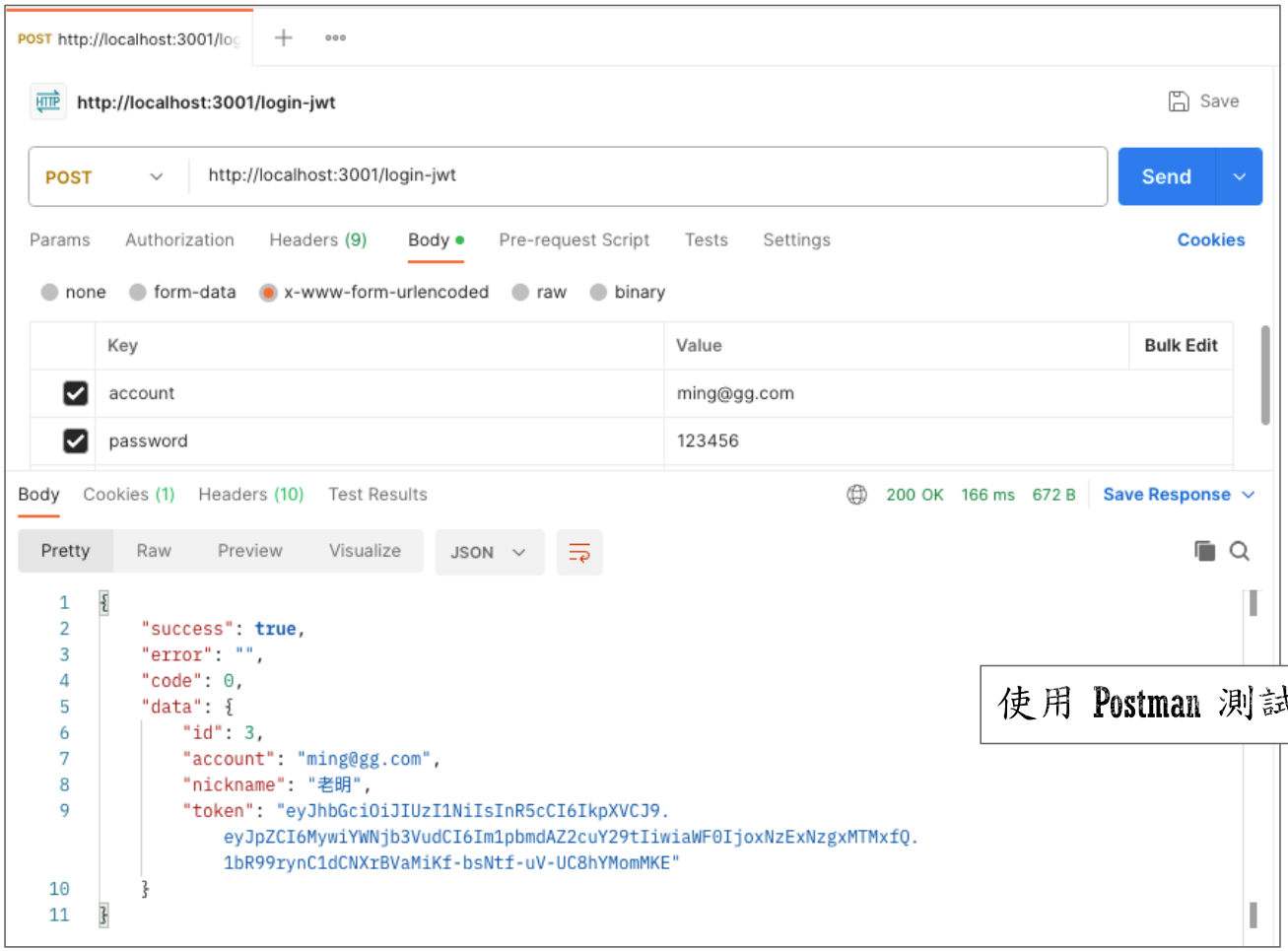
```
// 帳號密碼都是對的，打包 JWT
const payload = {
  id: row.member_id,
  email,
};
const token = jwt.sign(payload, process.env.JWT_KEY);
output.data = {
  id: row.member_id,
  email,
  nickname: row.nickname,
  token,
};

output.success = true;
res.json(output);
```

11.2 後端驗證 token

```
// 在 app top-level middleware 處理 JWT token

const auth = req.get("Authorization");
if (auth && auth.indexOf("Bearer ") === 0) {
  const token = auth.slice(7); // 去掉 "Bearer "
  try {
    // my_jwt 為我們決定的屬性名稱，勿與已存在的屬性重複
    req.my_jwt = jwt.verify(token, process.env.JWT_SECRET);
  } catch (ex) {}
}
```

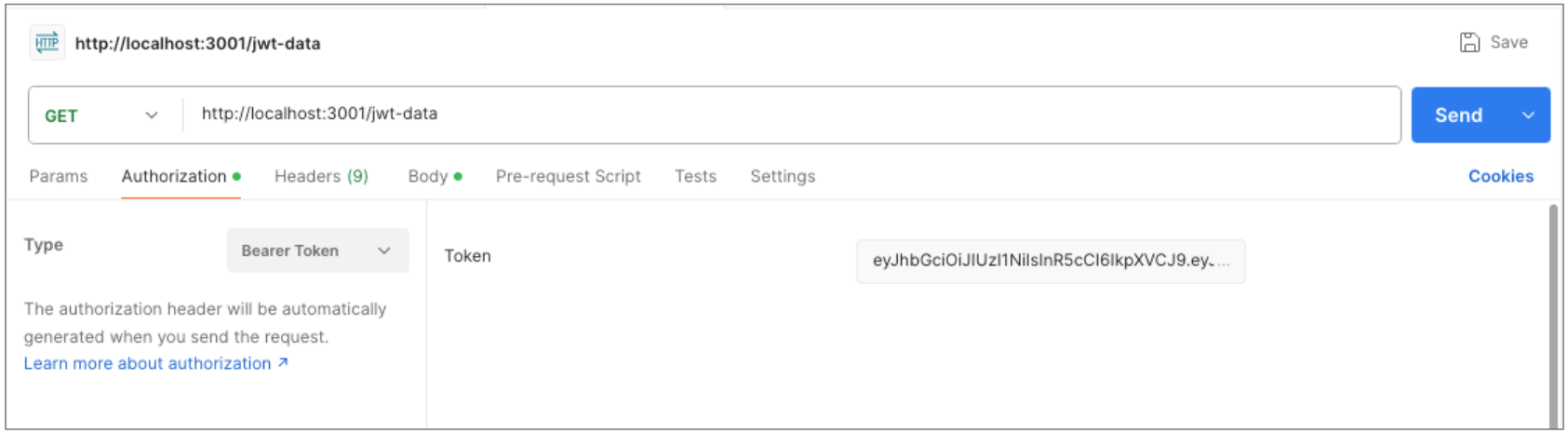
11.3 後端驗證 token 的測試路由

```
app.get("/jwt-data", async (req, res) => {
  res.json(req.my_jwt);
});
```

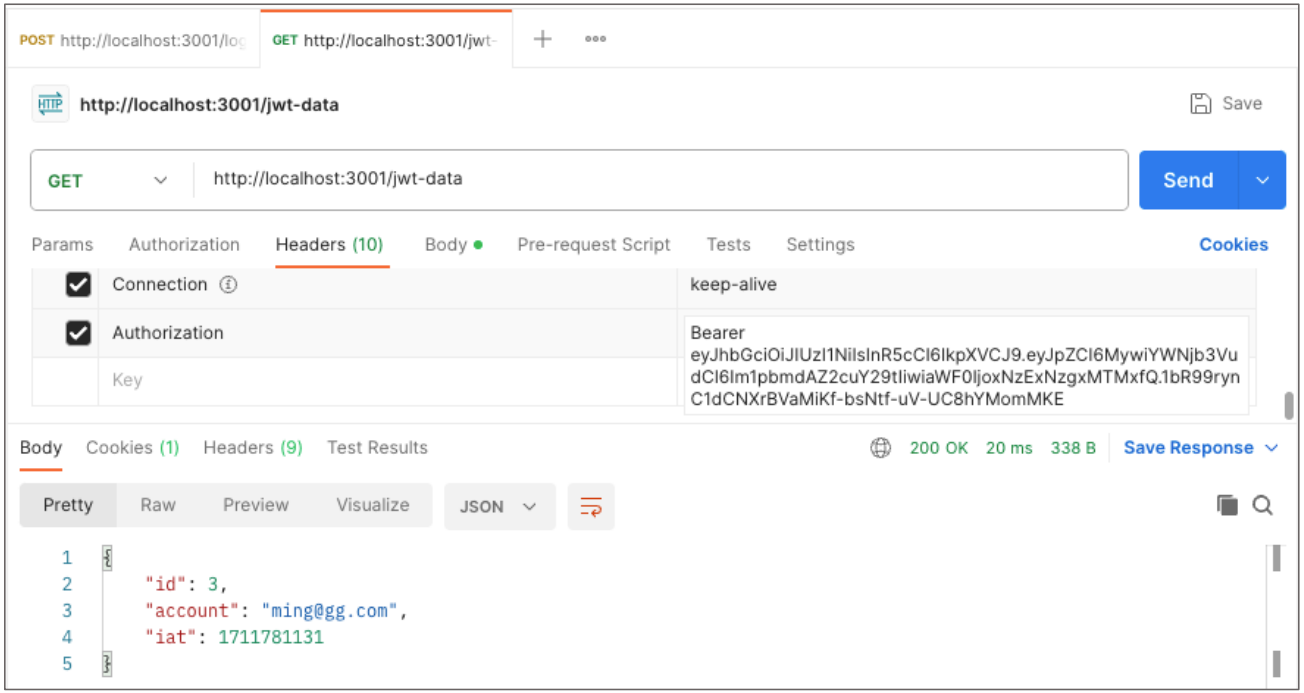


使用 Postman 測試登入功能

使用 Postman 測試授權要求，在 Authorization 分頁選 Bearer Token 並在右側輸入 token



另一種作法，在 Headers 分頁加入 Authorization 檔頭並放入 token



11.4 前端 AuthContext 檔案基本架構

```
// contexts/auth-context.js
"use client";
import { createContext, useContext, useState, useEffect } from "react";
const AuthContext = createContext();
/*
1. 登入
2. 登出
3. 取得登入者的資料
4. 取得已登入者的 token (或直接拿到 Authorization headers)
5. 是否已從 localStorage 取得登入狀態資料 (authInit)
*/
export function AuthContextProvider({ children }) {
  return (
    <AuthContext.Provider value={{ auth, logout, login, getAuthHeader }}>
      {children}
    </AuthContext.Provider>
  );
}
export const useAuth = () => useContext(AuthContext);
export default AuthContext;
```

app/layout.js 為可處理的根元件

- 在 Next.js 的 App Router 中，要讓整個網站的頁面都能使用某個 context provider，應該將 context provider 放置在 app/layout.js 或 app/layout.tsx 文件中。因為 layout.js 是每個頁面渲染時的根元件，適合用來設定全域的共享功能或狀態管理。

```
"use client";
import { AuthContextProvider } from "@/contexts/auth-context";
export default function RootLayout({ children }) {
  return (
    <AuthContextProvider>
      <html lang="zh">
        <body>
          {children}
        </body>
      </html>
    </AuthContextProvider>
  );
}
```

11.5 前端 AuthContext 檔案內全域變數

```
// 預設的狀態, 沒有登入
const emptyAuth = {
  id: 0,
  email: "",
  nickname: "",
  token: "",
};
const storageKey = "shinder-auth";
```

11.6 前端 AuthContextProvider 裡的登出功能

```
const [auth, setAuth] = useState({ ...emptyAuth });

// 登出的功能
const logout = () => {
  localStorage.removeItem(storageKey);
  setAuth({ ...emptyAuth });
};
```

11.7 前端 AuthContextProvider 裡的登入功能

```
// 登入的功能
const login = async (email, password) => {
  try {
    const r = await fetch(JWT_LOGIN_POST, {
      method: "POST",
      body: JSON.stringify({ email, password }),
      headers: {
        "Content-Type": "application/json",
      },
    });
    const result = await r.json();
    if (result.success) {
      // 把取得的用戶資料和 token 記錄在 localStorage
      localStorage.setItem(storageKey, JSON.stringify(result.data));
      setAuth(result.data);
      return true;
    }
  } catch (ex) {}
  return false;
};
```

11.8 前端 getAuthHeader()

```
const getAuthHeader = () => {
  if (auth.token) {
    return { Authorization: "Bearer " + auth.token };
  } else {
    return {};
  }
};
```

11.9 前端判斷 localStorage 是否有登入的資料

```
useEffect(() => {
  // 刷新頁面時, 從 localStorage 讀取登入的狀態資料
  const str = localStorage.getItem(storageKey);
  if (str) {
    try {
      const authData = JSON.parse(str);
      setAuth(authData);
    } catch (ex) {}
  }
  setAuthInit(true); // 做完初始化
}, []);
```

** 刷新頁面時，讀取 localStorage

11.10 前端快速登入的頁面

```
// app/address-book/quick-login/page.js
"use client";
import { useAuth } from "@/contexts/auth-context";
export default function QuickLoginPage() {
  const { auth, login } = useAuth();
  return (
    <>
      <button
        className="btn btn-primary"
        onClick={() => login("shin@test.com", "123456")}
      >
        登入 shin@test.com
      </button>
      <hr />
      <div>目前登入的 email: {auth.email}</div>
    </>
  );
}
```

11.11 前端 Navbar

```
{auth.id ? (
  <>
    <li className="nav-item">
      <a className="nav-link">{auth.nickname}</a>
    </li>
    <li className="nav-item">
      <a className="nav-link" href="#"
        onClick={(e) => {
          e.preventDefault();
          logout();
        }} >登出</a>
    </li>
  </>
) : (
  <li className="nav-item">
    <Link className="nav-link"
      style={
        pathname === "/address-book/quick-login"
          ? selectedStyle : {}
      }
      href="/address-book/quick-login"
    >快速登入</Link>
  </li>
)}
```

11.12 前端頁面做權限判斷

```
useEffect(() => {
  if (authInit && !auth.id) {
    // 如果沒有登入, 不能拜訪這個頁面
    router.push("/quick-login");
  }
}, [auth, authInit, router]);
```

12. 加入最愛使用 JWT

```
CREATE TABLE `ab_likes` (
    `like_id` int(11) NOT NULL,
    `member_id` int(11) NOT NULL,
    `ab_id` int(11) NOT NULL,
    `created_at` datetime NOT NULL DEFAULT
CURRENT_TIMESTAMP
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

```
-- 資料表索引 `ab_likes`
ALTER TABLE `ab_likes`
    ADD PRIMARY KEY (`like_id`),
  ADD UNIQUE KEY `member_id_2` (`member_id`,`ab_id`),
  ADD KEY `member_id` (`member_id`),
  ADD KEY `ab_id` (`ab_id`);
```

```
-- 使用資料表自動增長(AUTO_INCREMENT) `ab_likes`
--
ALTER TABLE `ab_likes`
    MODIFY `like_id` int(11) NOT NULL AUTO_INCREMENT;
```

```
-- 資料表的限制(constraint) `ab_likes`
--
ALTER TABLE `ab_likes`
    ADD CONSTRAINT `ab_likes_ibfk_1`
        FOREIGN KEY (`member_id`)
        REFERENCES `members` (`member_id`),
    ADD CONSTRAINT `ab_likes_ibfk_2`
        FOREIGN KEY (`ab_id`)
        REFERENCES `address_book` (`ab_id`);
```

12.1 後端 toggle-like 功能

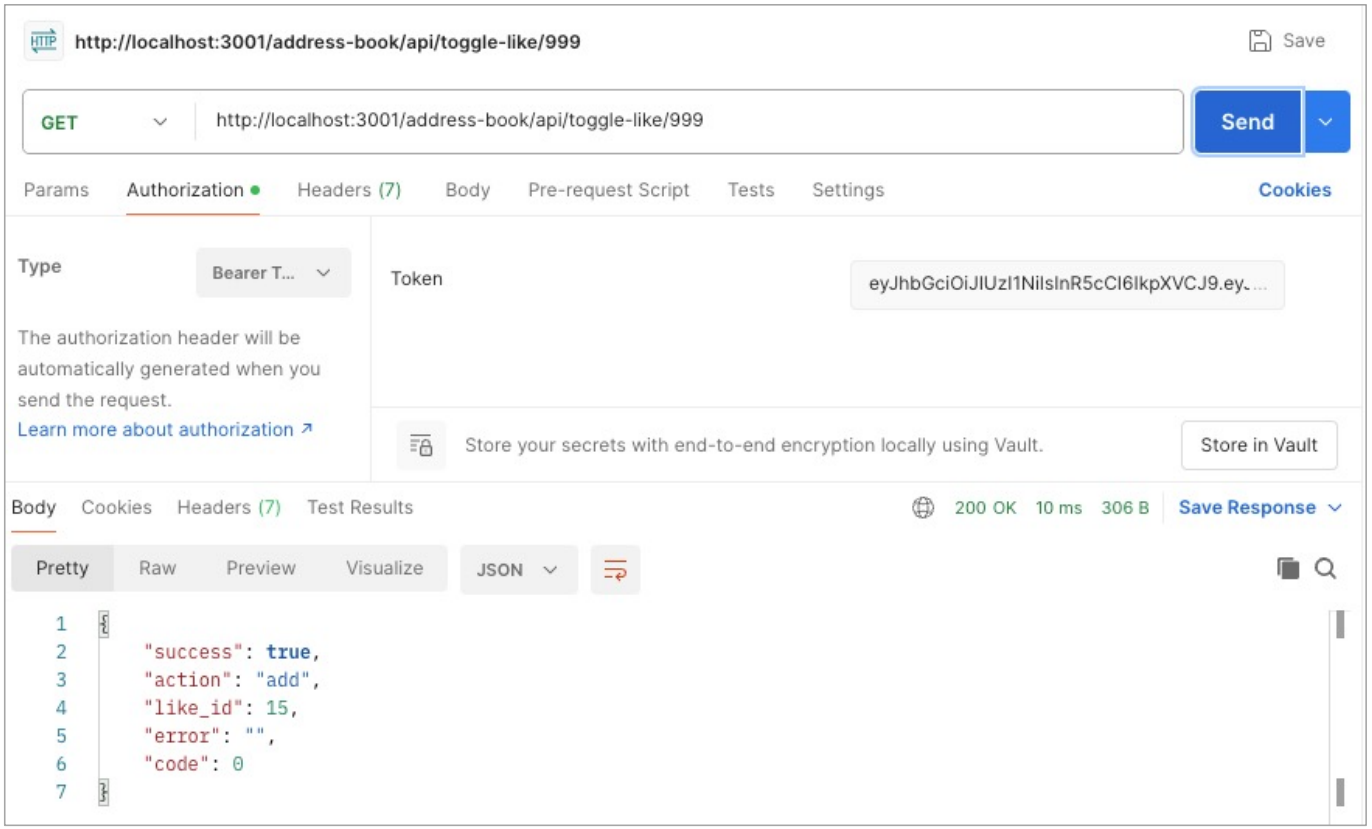
```
// routes/address-book.js

router.get("/api/toggle-like/:ab_id", async (req, res) => {
  const output = {
    success: false,
    action: "", // add, remove
    like_id: 0,
    error: "",
    code: 0,
  };
  if (!req.my_jwt?.id) {
    // 沒有授權
    output.code = 430;
    output.error = "沒有授權";
    return res.json(output);
  }
  const member_id = req.my_jwt.id; // 從 JWT 來的
```

```
// 先確認有沒有這個項目

const sql1 = "SELECT ab_id FROM address_book WHERE ab_id=? ";
const [rows1] = await db.query(sql1, [req.params.ab_id]);
if (!rows1.length) {
  output.code = 401;
  output.error = "沒有這個朋友";
  return res.json(output);
}

const sql2 = "SELECT * FROM `ab_likes` WHERE `member_id`=? AND `ab_id`=?";
const [rows2] = await db.query(sql2, [member_id, req.params.ab_id]);
```

```
if (rows2.length) {
  // 如果已經有這個項目, 就移除
  const sql3 = ` DELETE FROM ab_likes WHERE like_id=${rows2[0].like_id} `;
  const [result] = await db.query(sql3);
  if (result.affectedRows) {
    output.success = true;
    output.action = "remove";
  } else {
    output.code = 410;
    output.error = "無法移除";
    return res.json(output);
  }
}
```

```
 else {
    // 如果沒有這個項目, 就加入

    const sql4 = "INSERT INTO `ab_likes` (`member_id`, `ab_id`) VALUES (?, ?)";
    const [result] = await db.query(sql4, [member_id, req.params.ab_id]);
    if (result.affectedRows) {
      output.success = true;
      output.action = "add";
      output.like_id = result.insertId;
    } else {
      output.code = 420;
      output.error = "無法加入";
      return res.json(output);
    }
  }
  res.json(output);
});
```

12.2 使用 Postman 測試 API



12.3 後端取得列表時的 SQL

```
-- 使用 SQL 子查詢
-- 子查詢結果必須使用別名

SELECT ab.*, li.like_id
    FROM address_book ab
    LEFT JOIN (
        SELECT * FROM ab_likes WHERE member_id=3
    ) li ON ab.ab_id=li.ab_id
    ORDER BY ab.ab_id DESC
    LIMIT 30
```

12.4 後端 getListData() 修改 SQL

```
const member_id = req.my_jwt?.id || 0; // 授權的用戶
console.log("授權的用戶: ", member_id, new Date());

let where = " WHERE 1 "; // SQL 條件的開頭

// 關鍵字的查詢
const keyword = req.query.keyword || "";
if (keyword) {
  const keyword_ = db.escape(`%${keyword}%`); // SQL 的跳脫, 同時會用單引號包起來
  where += ` AND ( ab.\`name\` LIKE ${keyword_} OR ab.mobile LIKE ${keyword_} ) `;
}
```

```
// 生日的篩選
const birth_begin = req.query.birth_begin
  ? moment(req.query.birth_begin)
  : null;
const birth_end = req.query.birth_end ? moment(req.query.birth_end) : null;

if (birth_begin && birth_begin.isValid()) {
  where += ` AND ab.birthday >= '${birth_begin.format(fmDate)}' `;
}
if (birth_end && birth_end.isValid()) {
  where += ` AND ab.birthday <= '${birth_end.format(fmDate)}' `;
}

const t_sql = `SELECT COUNT(1) totalRows FROM address_book ab ${where} `;
// 多層的解構
const [[{ totalRows }]] = await db.query(t_sql);
const totalPages = Math.ceil(totalRows / perPage);
```

```
let rows = []; // 預設值
let totalPages = 0;
if (totalRows) {
  totalPages = Math.ceil(totalRows / perPage);
  if (page > totalPages) {
    // 包含其他的參數
    const newQuery = { ...req.query, page: totalPages };
    const qs = new URLSearchParams(newQuery).toString();
    return { success: false, redirect: `?` + qs };
  }
  const sql = `SELECT ab.*, li.sid like_sid
      FROM address_book ab
      LEFT JOIN (
        SELECT * FROM ab_likes WHERE member_sid=${member_sid}
      ) li ON ab.sid=li.ab_sid
      ${where}
      ORDER BY ab.sid DESC
      LIMIT ${(page - 1) * perPage}, ${perPage}`;
```

```
  [rows] = await db.query(sql);
   for (let r of rows) {
     // 直接用 moment 做轉換, 空值就不做傳換
     if (r.birthday) {
       r.birthday2 = moment(r.birthday).format(fmDate);
     }
   }
}
return { perPage, page, totalRows, totalPages, rows };
```

12.5 前端取得列表資料時要發送 token

```
useEffect(() => {
  const controller = new AbortController(); // 用來取消的控制器
  const signal = controller.signal;
  fetch(`${AB_LIST}${location.search}`, {
    headers: { ...getAuthHeader() },
    signal,
  })
    .then((r) => r.json())
    .then((obj) => {
      if (obj.success) {
        setListData(obj);
      } else if (obj.redirect) {
        router.push(obj.redirect);
      }
    })
    .catch(console.warn); // 用戶取消時會發生 exception
  return () => controller.abort(); // 取消未完成的 ajax
}, [searchParams, refresh, getAuthHeader, router]);
```

12.6 前端愛心圖示呈現

```
<td>
  {/* 顯示 like_id 測試 */}
  {v.like_id} {` `}
  <a
    href="#"
    onClick={(e) => {
      e.preventDefault();
      toggleLike(v.ab_id);
    }}
  >
    {v.like_id ? <FaHeart /> : <FaRegHeart />}
  </a>
</td>
```

12.7 前端 toggleLike 功能

```
const toggleLike = async (ab_id) => {
  const r = await fetch(`${AB_LIKE}/${ab_id}`, {
    headers: { ...getAuthHeader() },
  });
  const result = await r.json();
  if (result.success) {
    const newListData = { ...listData };
    newListData.rows = listData.rows.map((item) => {
      if (item.ab_id === ab_id) {
        const like_id = result.action === "add" ? result.like_id : null;
        return { ...item, like_id };
      } else {
        return { ...item };
      }
    });
    setListData(newListData);
  }
};
```

Thank You