



# NLP與網路應用

## React & HF-API

Sean Tseng  
2022/12/8



# 今天要做的事

- 用React寫出右邊類似聊天機器人的東西，Backend是👉的語言模型。
- 很不幸的bloomz停掉了，但幸好有其他的。
- 內容很多，但會拆成很多步驟。一不小心跟丟了，每步都有escape hatch (Git tags)

你好阿

 bigscience/bloomz 

昨天我去看電影。把它翻成英文

I went to the movies yesterday.

「他昨天去看黑豹2。」他昨天看了什麼？

黑豹2

學網頁設計要學：

HTML、CSS、JavaScript、PHP、SQL、  
Ruby on Rails、Python、Ruby、Java、

Type here

# 請確定可以跑 `npm install`或CRA

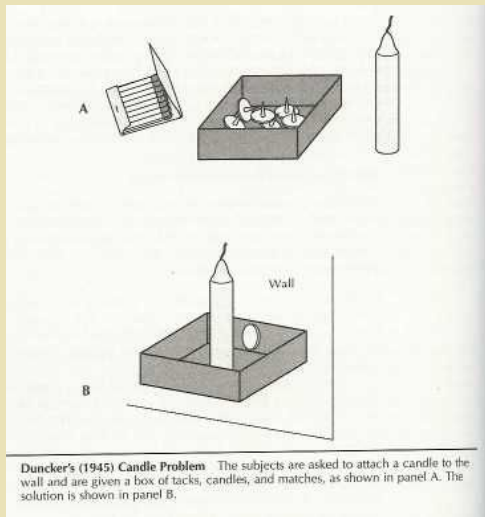
- 這一步預設你已經有git、node, 以及npm
- 這次的code在 repo, /lab/deepdive/wander
  - 應該可以直接`npm install` 或者是 `yarn install`
- 但你也可以試試看另開一個專案從create-react-app開始, 雖然到最後做出來的東西不見得一樣, 但過程會很有趣!
  - `npx creat-react-app mywander`
  - `yarn create react-app mywander`

# 等待npm install的同時...

- 為什麼我們要學前端？Python 不夠嗎？
  - 雖然，等大家學會前端之後，我們就一定會想到理由來正當化自己的行為：某種「認知失調」(cognitive dissonance)的效果。
  - 但除此之外，我覺得(可能也是認知失調導致的藉口)前端設計比較容易讓我們想到我們程式的使用者，於是我們比較容易從應用角度切入想事情。

# Python已經太接近NLP了

- Python—在特定用法下—已經非常非常靠NLP或ML的取向。我們對於工具多少都有點功能固化 (functional fixedness)的傾向
  - 當我們在Python都是看著notebook, 一開始就`import pandas`, 甚至`import torch`的時候...
  - 我們忙著想資料怎麼清, 張量維度多少, 模型怎麼調參, 我們還有多少餘裕想著我們的程式作品其實有「使用者」?



# JS剛好是個嶄新開始

- 有工具幫我們區隔思維是好事，更何況JS本來就是在前端脈絡發展出來的語言。
- JS打從出生起就在與使用者事件、非同步程式邏輯、前後端整合的問題搏鬥，甚至整個開發社群也是。在這個語言裡，我們很容易順著程式語言的思路，想「使用者」會怎麼跟程式互動。
- 而且，現實上，我們當真玩得起現在NLP模型的遊戲嗎？純學術研究當然永遠有機會，但不是每個人都會對那種東西有興趣。



## Hardware

## Bloom (176B) 的訓練環境

- 384 A100 80GB GPUs (48 nodes)
- Additional 32 A100 80GB GPUs (4 nodes) in reserve
- 8 GPUs per node Using NVLink 4 inter-gpu connects, 4 OmniPath links



# 我們不見得只有一籃雞蛋

- 「兩個都學，不是分散時間嗎？會不會反而兩個都學不好？」
  - 與其說分散時間，不如說是分散風險。而且，如果自己沒興趣，時間也不可能放得太多；但相對地，如果對Python沒感覺，說不定反而對JS有感應。那反而是幫自己多開了一扇窗。
  - 就前端開發而言，Python的知識可以讓自己瞭解後端大概包含什麼；但模型開發和訓練，就交給那些有資源的大人們吧。
- 
- 

# 友善的大人

- 就現在看起來，不少人對NLP模型有興趣但沒資源維護後端。
- 所以，有些地方會提供NLP模型的API服務：
  - 有的要付費: OpenAI
  - 有的免費: HF Inference API
  - 如果只是要開發測試, [colab+ngrok](#)也可以試試看。
- 我們今天要用到的就是HF API, 它的API使用Access token認證, 請大家先在Huggingface 註冊帳號, 並取得token



# 最近大人們(OpenAI)的新玩具




Create a multi-files React projects which has a chat-bot interface and interfaces with inference API from Huggingface. Users should be able to type in their inputs in the textbox at the bottom of the screen, and the conversations would appear on the top.



To create a multi-file React project with a chatbot interface that interfaces with the Huggingface inference API, you can follow these steps:

1. First, make sure you have Node.js and npm installed on your computer. You can check if you have them installed by running the following commands in your terminal:

 Copy code

```
node -v  
npm -v
```

希望這有天可以  
成真

8. Import the Huggingface `transformers` package in your `index.js` file and use it to interface with the inference API. For example, you could add the following code to your `index.js` file to initialize a GPT-3 model and generate a response to a user's input:

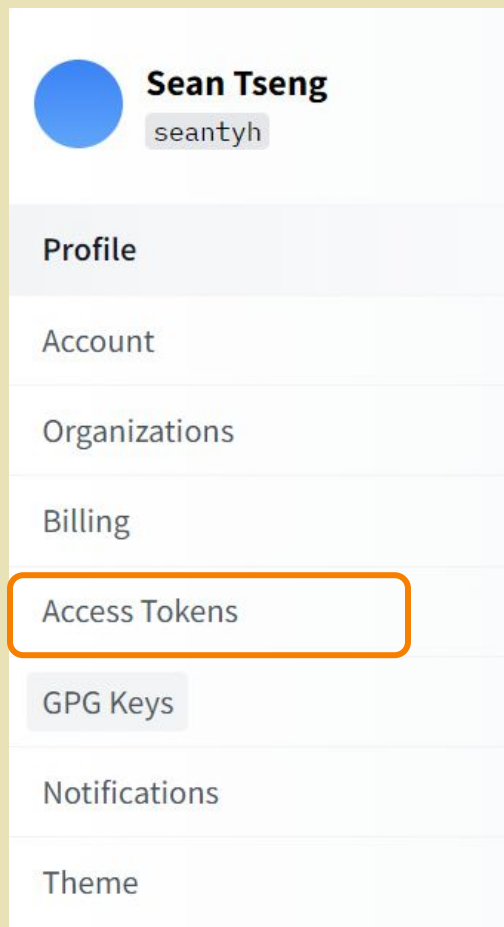
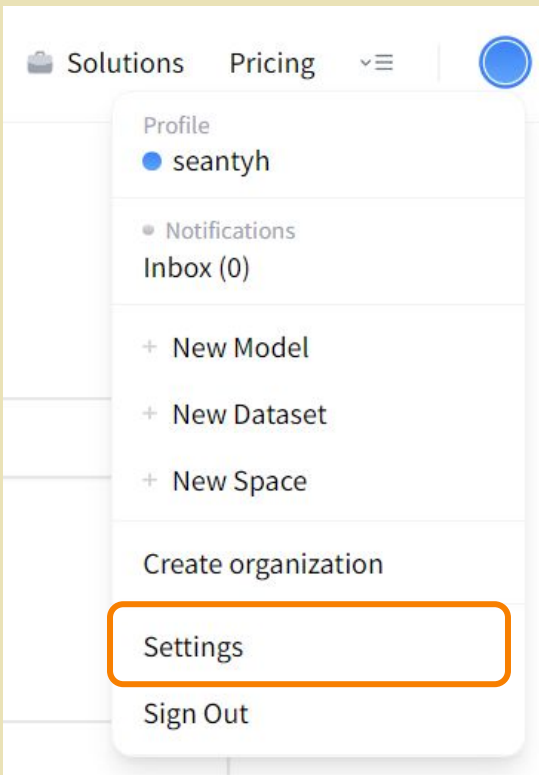
Copy code

```
import { GPT3LMHeadModel } from '@huggingface/transformers';

const model = new GPT3LMHeadModel('gpt3');

async function generateResponse(input) {
  const response = await model.generate(input, {
    max_length: 100,
    temperature: 0.5,
  });

  return response;
}
```



請到 <https://huggingface.co/> 申請帳號

Access Token相當於API用的帳密，  
請妥善保管，勿與他人共用

## Access Tokens

### User Access Tokens

Access tokens programmatically authenticate your identity to the Hugging Face Hub, allowing applications to perform specific actions specified by the scope of permissions (read, write, or admin) granted. Visit [the documentation](#) to discover how to use them.



# 測試Access Token

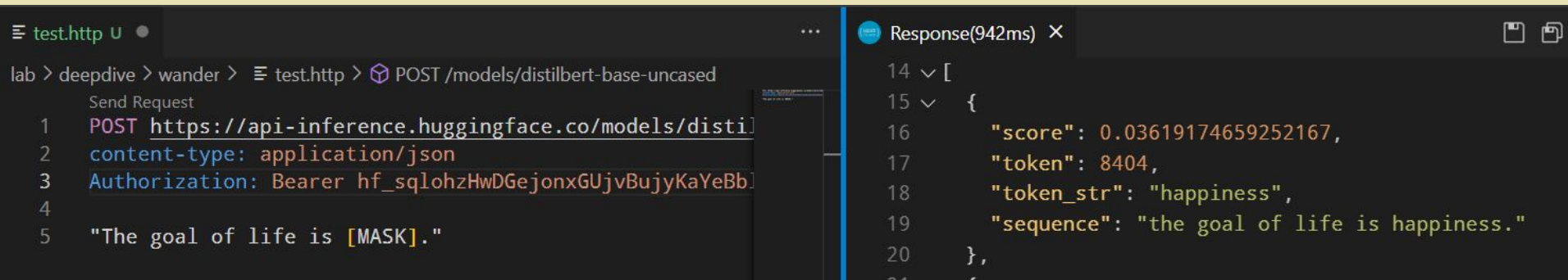
- 開發前後端時，通常會避免在程式邏輯裡直接測API；因為如果有問題，我們很難知道是API還是程式的問題。
- 所以開發者通常會選擇一個測試用 client。在VSCode裡有多人用的選擇：REST Client、Thunder Client
- 請在專案裡任意地方開一個檔案test.http，並打入以下文字

```
POST https://api-inference.huggingface.co/models/distilbert-base-uncased
content-type: application/json
Authorization: Bearer <YOUR_HF_TOKEN>

"The goal of life is [MASK]."
```

# 送出API

- 按下編輯器左上角的 Send Request, 如果一切正常, 應該就會跳出右側結果。
- Response第一行應該會寫 **HTTP/1.1 200 OK**



The screenshot shows a web browser's developer tools interface. On the left, the 'Send Request' tab is active, displaying a POST request to the Hugging Face API. The request body is a JSON object with 'content-type' and 'Authorization' headers, and a text prompt. On the right, the 'Response(942ms)' tab is active, showing the JSON response from the API, which includes a 'score' and a 'token'.

```
lab > deepdive > wander > test.http > POST /models/distilbert-base-uncased  
Send Request  
1 POST https://api-inference.huggingface.co/models/distilbert-base-uncased  
2 content-type: application/json  
3 Authorization: Bearer hf_sqlohzHwDGejonxGUjvBujyKaYeBbJ  
4  
5 "The goal of life is [MASK]."  
  
...  
Response(942ms) X  
14 [  
15 {  
16   "score": 0.03619174659252167,  
17   "token": 8404,  
18   "token_str": "happiness",  
19   "sequence": "the goal of life is happiness."  
20 }  
21 ]
```

# 接下來

- 做一個對話介面，然後用HF提供的API來回應使用者。
- 這個範例原本想要用bloomz，但很巧的該API在12/1就中止服務了。幸好有類似的mt0-xxl-mt。
- 整個範例程式在lab/deepdive/wander
- 等一下有7個步驟，分別對應7個tags

```
▼ TAGS (8)  +  ≡  ↶  ↷  ...
> wander-07-complete-api add api_request
> wander-06-process-time-out add process_...
> wander-05-input-event add onInputHandler
> wander-04-dialogue-state extract dialogue ...
> wander-03-chat-turn factoring out ChatTurn
> wander-02-layout-chatbox layout the chatb...
> wander-01-sanity-check update app.js and t...
> wander-00-cra add wander CRA
```



00

簡單編輯

Back



Next

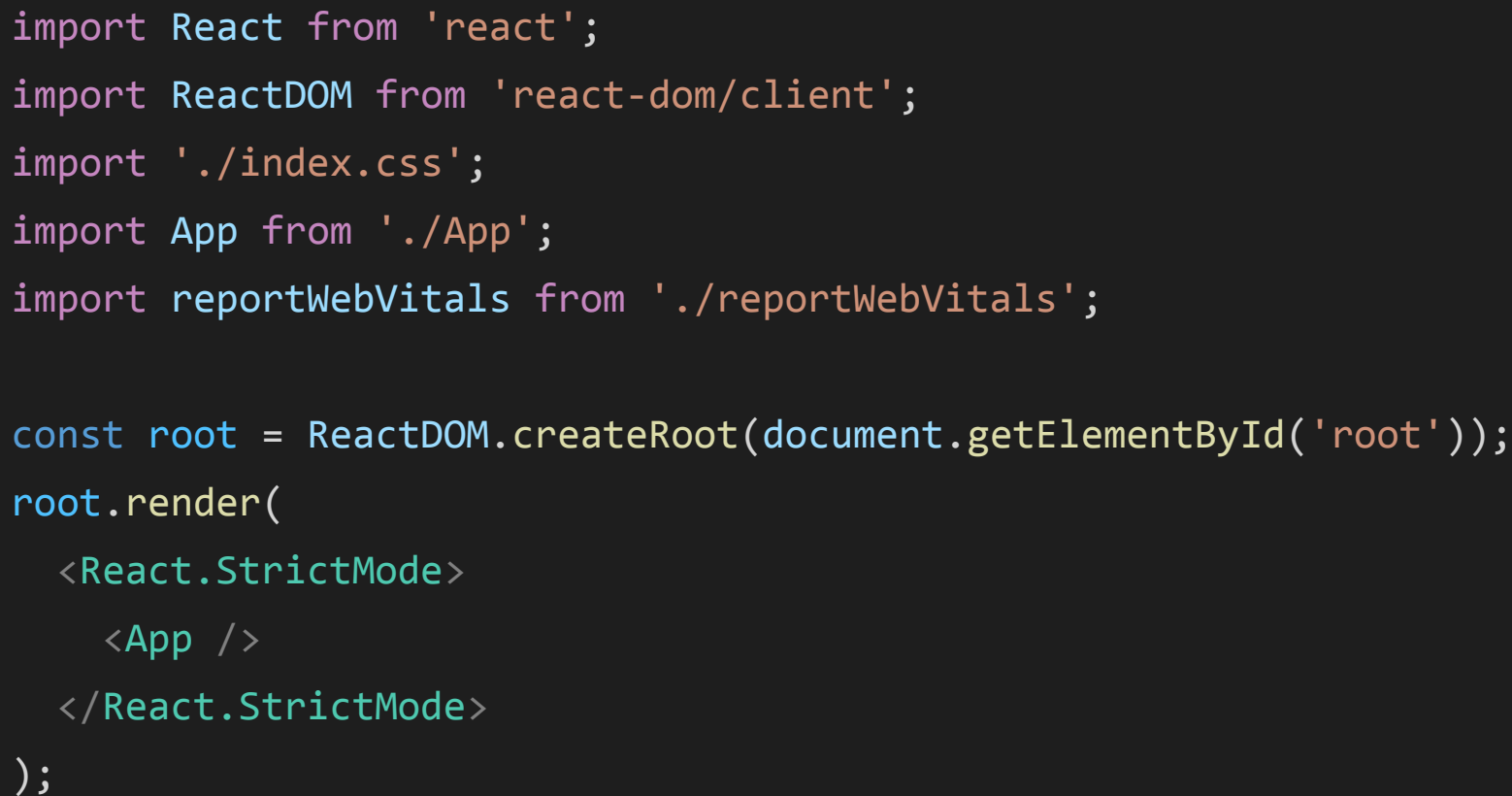
# 簡單編輯

- Checkpoint: [Github link](#)
- 這個版本幾乎就是CRA後的樣子，但我改了幾個地方。
- 這些小地方其實不是為了程式開發，而比較像是某種sanity checks：確定我跟React頻率有對上。
  - 把CRA的預設頁面拿掉，換上一個簡單的訊息
  - 換掉網頁標題
- 從CRA到Step 1 [Diff](#)



# React的幾個重要檔案

- public/index.html
  - 是最終網頁的「模版」，這裡還是一般的HTML
  - 整個React的掛載點 (mount) 就在裡面的 `<div id="root">`
- src/index.js
  - Node的程式進入點，在這裡引入App.js，並且把React程式程式掛到div#root上
- src/App.js
  - 最「上層」的Component



```
import React from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
);
```





Edit src/App.js and save to reload.

[Learn React](https://reactjs.org)

```
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <p>  
          Edit src/App.js and save to reload.  
        </p>  
        <a  
          className="App-link"  
          href="https://reactjs.org"  
          target="_blank"  
          rel="noopener noreferrer"  
        >  
          Learn React  
        </a>  
      </header>  
    </div>  
  );  
}  
  
export default App;
```



# React

- 雖然這不是React首創，但以往網頁設計有三個元素：HTML/CSS/JS，分別各自在不同的檔案，而且有各自的環境。
  - 在React，這三個東西都可以寫在.js裡。
  - 我們今天會用functional component styles，也是React社群比較推薦的作法。
  - Functional component可以想成是一個函數 (function)，它會回傳一個component。這個component，其實是一個HTML element。
- 
- 

# React component的動機

- 很可能在專案裡，我們需要一個鬧鈴介面。這個介面需要顯示時間和圖示。
- 第一種HTML寫得很瑣碎。
- 但如果可以寫成第二種方法，是不是簡單漂亮好懂多了？
- 「表達」永遠是程式語言演變的動力。

02:30:25



```
<div class="some-compo">
  <span id="timestamp">02:30:25</span>
  <span id="alarm">🔔</span>
</div>

<!-- -- vs -- -->
<AlarmClock time="02:30:25"/>
```

# React's Component的好處

- 它讓我們可以任意定義新的HTML element, 並且得以完全控制它在頁面裡, 要扮演什麼功能以及怎麼呈現。
- 這是元件 (component) 的主要功能之一: 抽象化(abstraction)
  - 在視覺上, 我只需要說 <AlarmClock> 就代表了一個鬧鐘介面
  - 那個鬧鐘長得像什麼樣子, 要怎麼寫成HTML, 怎麼套用CSS, 那都是AlarmClock元件自己的「實作細節」(implementation detail)

# App.js

- App.js裡面就是一個App()函數，它回傳App元件。
- 刮號內(...)就是JSX的語法，和HTML非常像。除了，`{...}`可跳回JS，class需要寫成`className`

```
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <p>  
          Edit <code>src/App.js</code> and save to reload.  
        </p>  
        <a  
          className="App-link"  
          href="https://reactjs.org"  
          target="_blank"  
          rel="noopener noreferrer"  
        >  
          Learn React  
        </a>  
      </header>  
    </div>  
  );  
}  
  
export default App;
```

# 試著玩一下App.js

- 看看改動裡面的東西，  
會怎麼影響預設的App  
頁面。

```
function App() {  
  return (  
    <div className="App">  
      <header className="App-header">  
        <img src={logo} className="App-logo" alt="logo" />  
        <p>  
          Improve ---/App.js</code> and refresh.  
        </p>  
        <a  
          className="App-link"  
          href="https://reactjs.org"  
          target="_blank"  
          rel="noopener noreferrer"  
        >  
          Learn How to React Nicely  
        </a>  
      </header>  
    </div>  
  );  
}  
  
export default App;
```





# 01

## 靜態元件

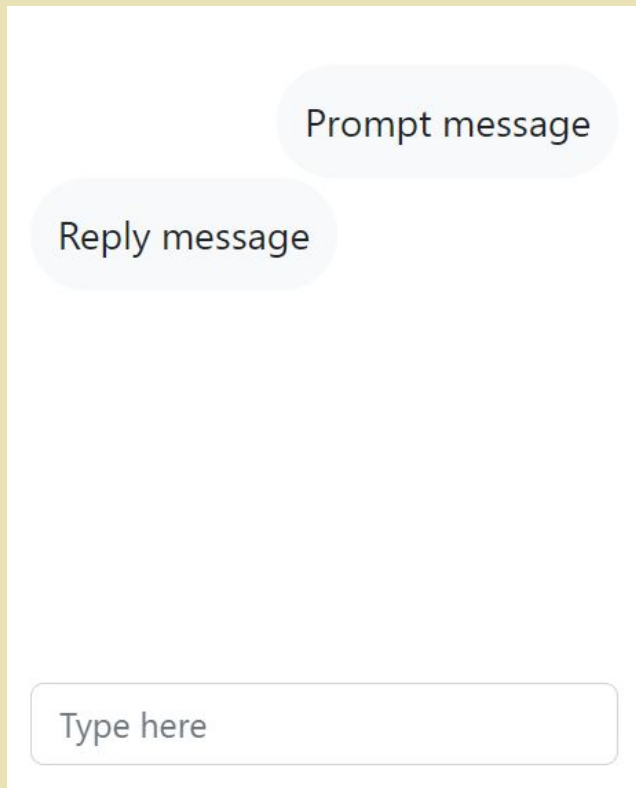
Back



Next

# 靜態元件

- Checkpoint: [Github link](#)
- 當我們清掉App.js預設的內容後，我們可以想一下我們要做什麼
- 我想做個類似聊天視窗的東西，所以至少要有個像對話視窗的東西。
- 從Step 1到Step 2 [Diff](#)



# App.js 包裝

- 因為App.js是CRA預設給的，所以通常習慣把它當成一個「包裝用」的元件。
- 真正跟程式邏輯有關的東西——這裡是指對話清單——放在另一個元件裡：ChatBox

```
import ChatBox from "../chatbox";  
import './App.css';
```

```
function App() {  
  return (  
    <ChatBox/>  
  );  
}
```

```
export default App;
```

# 引入ChatBox

- React的慣例是用ES Modules, 所以都是用import 引入, export匯出。
- “./chatbox” 指的是和當前檔案同一目錄的chatbox.js 並且從中引入 ChatBox 函數。
- 這個函數會回傳我們要的 ChatBox元件

```
import ChatBox from "./chatbox";
```

```
import './App.css';
```

```
function App() {  
  return (  
    <ChatBox/>  
  );  
}
```

```
export default App;
```

# ChatBox元件

- 同樣的，這只是個完全靜態的元件：它回傳的元件是固定的，完全沒有變數。
- 這個專案有用到Bootstrap CSS，它用class(Name)來選擇樣式，所以class裡的東西都只是美觀用，和程式無關。

```
export default function ChatBox() {  
  return (  
    <div className="w-50 mx-auto mt-5 fs-4">  
      <div className="d-flex flex-column">  
        <div className="[...]">  
          Prompt message  
        </div>  
        <div className="[...]">  
          Reply message  
        </div>  
      </div>  
      <div>  
        <div className="[...]">  
          <div className="w-50 mx-auto mt-5 fs-4">  
            <input type="text"  
              className="[...]"  
              placeholder="Type here"/>  
          </div>  
        </div>  
      </div>  
    </div>  
  )  
}
```

# Bootstrap CSS

- Bootstrap有很多功能，但我們這裡只用到它的CSS樣式表。它有很多預設的樣式(排版、方塊形狀等)，讓畫面看起來可以比較漂亮。
- 因為我們只用CSS，跟React可以完全無關。這裡選擇的引入方法是在[public/index.html](https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css)，把它當純CSS引入。
- 意思是，React完全不知道有這個Bootstrap CSS存在。但沒關係，瀏覽器知道就好。

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.2.3/dist/css/bootstrap.min.css" rel="stylesheet"
      integrity="sha384-rbsA2VBKQhggwzxH7pPCaAqO46MgnOM80zW1RWuH61DGLwZJEdK2Kadq2F9CUG65" crossorigin="anonymous">
```



# 02

## 加入變數

Back



Next

# 動態元件

- Checkpoint: [Github link](#)
- 現在看起來ChatBox運作得不錯, 我們可以一步步加入動態元素
- 動態並不是說畫面上真的會動(animated), 而是它會跟著資料而動態地(dynamically) 改變
- 從Step 2到Step 3 [Diff](#)

Prompt Message - 1

Reply Message - 1

Prompt Message - 2

Reply Message - 2

Type here



# 一步步的把元件「抽」出來

- ChatBox裝對話，而對話應該是一個陣列 (array)
- 在概念上，對話陣列的元素可以有很多可能，這邊用 turns

```
export default function ChatBox() {  
  return (  
    <div className="w-50 mx-auto mt-5 fs-4">  
      <div className="d-flex flex-column">  
        <div className="[...] ">  
          Prompt message  
        </div>  
        <div className="[...] ">  
          Reply message  
        </div>  
      </div>  
      [...]  
    </div >  
  )  
}
```



```
export default function ChatBox() {  
  return (  
    <div className="w-50 mx-auto mt-5 fs-4">  
      <ChatTurn  
        prompt="Prompt Message - 1"  
        reply="Reply Message - 1"/>  
      <ChatTurn  
        prompt="Prompt Message - 2"  
        reply="Reply Message - 2"/>  
      [...]  
    </div >  
  )  
}
```

## <ChatTurn/>

- 在HTML元素(elements)上, 每個turn如果都可以由一個元件(components)負責是最簡單的事, 每個 turn 包含:
  - prompt: 使用者輸入
  - reply: 使用者輸出
  - 兩個都是簡單字串
- 我們把這個元件叫做ChatTurn

```
export default function ChatBox() {  
  return (  
    <div className="w-50 mx-auto mt-5 fs-4">  
      <ChatTurn  
        prompt="Prompt Message - 1"  
        reply="Reply Message - 1"/>  
      <ChatTurn  
        prompt="Prompt Message - 2"  
        reply="Reply Message - 2"/>  
      [...]  
    </div >  
  )  
}
```

# ChatTurn裡面

- 先把原本的HTML整個搬過來
- 看看現在網頁變成什麼樣子。
- 因為我們在chatbox.js那裡放了兩個ChatTurn, 所以就是同樣code跑兩次, 產生兩個一樣的ChatTurn元件。

```
export default function ChatTurn(){  
  return (  
    <div className="d-flex flex-column">  
      <div className="...">  
        Prompt-1  
      </div>  
      <div className="...">  
        Reply-1  
      </div>  
    </div>  
  )  
}
```

# 對ChatTurn加上 Property

- 在chatbox.js裡, 我們給ChatTurn兩個attributes: prompt和reply。他們會傳給ChatTurn變成屬性 (props)。
- 一個小地方: 在函數的參數位置放置 `{...}` 是一種方便寫法, 但參數名稱必須和HTML的屬性 (attributes)名稱相同

```
export default function ChatTurn({
  prompt, reply
}){
  return (
    <div className="d-flex flex-column">
      <div className="...">
        {prompt}
      </div>
      <div className="...">
        {reply}
      </div>
    </div>
  )
}
```

# <ChatTurn/> 與 ChatTurn()

- chatbox.js裡的<ChatTurn/>是HTML元素 (elements)
- chat-turn.js裡的ChatTurn() 是JS函數
- ChatTurn() 負責定義以及製造出 <ChatTurn/> 應該要長什麼樣子, 有什麼行為。
- 這是React的魔法之一。

```
export default function ChatBox() {  
  return (  
    <div className="w-50 mx-auto mt-5 fs-4">  
      <ChatTurn  
        prompt="Prompt Message - 1"  
        reply="Reply Message - 1"/>  
      <ChatTurn  
        prompt="Prompt Message - 2"  
        reply="Reply Message - 2"/>  
      [...]  
    </div>  
  )  
}
```

```
export default function ChatTurn({  
  prompt, reply  
}){ [...] }
```



# 03

## 對話狀態

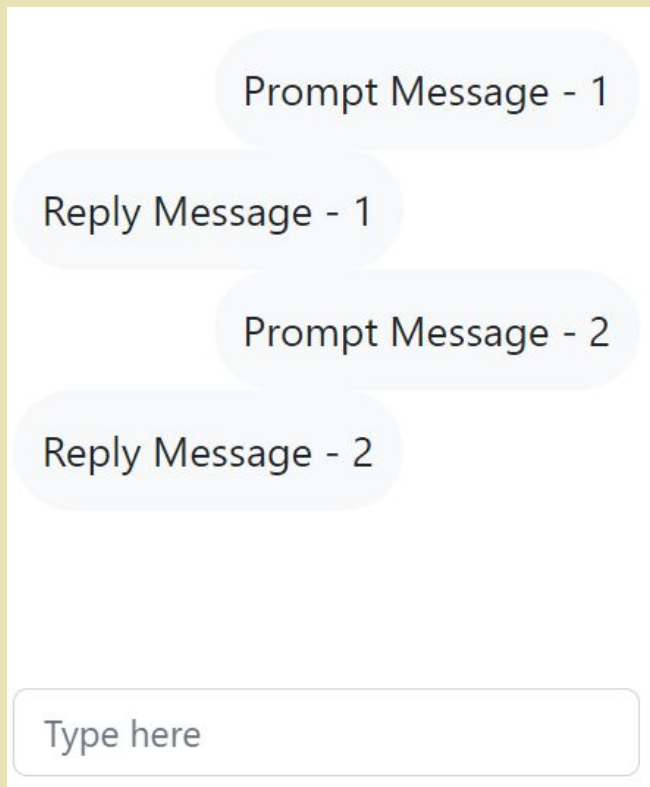
Back



Next

# 動態元件

- Checkpoint: [Github link](#)
- 這步和前一步的介面是相同的。
- 但我們原本ChatTurn的數量是「寫定的」: Copy-paste兩次
- turns會有不定數量, 所以我們要讓它的數量跟著一個不定長度的陣列走。
- 從Step 3到Step 4 [Diff](#)



# 把ChatTurn數量動態化

- 這步要作的事情是把原本複製貼上兩次的ChatTurn，變成一個`.map`。同時定義我們也要定義 `dialogue` 變數。

```
export default function ChatBox() {  
  return (  
    <div className="w-50 mx-auto mt-5 fs-4">  
      <ChatTurn  
        prompt="Prompt Message - 1"  
        reply="Reply Message - 1"/>  
      <ChatTurn  
        prompt="Prompt Message - 2"  
        reply="Reply Message - 2"/>  
      [...]  
    </div >  
  )  
}
```



```
export default function ChatBox() {  
  [...]  
  return (  
    <div className="w-50 mx-auto mt-5 fs-4">  
      {dialogue.map((dialogue_x, idx) => {  
        return (  
          <ChatTurn key={`dialogue_${idx}`}  
            prompt={dialogue_x.prompt}  
            reply={dialogue_x.reply} />  
        )  
      })}  
      [...]  
    </div >  
  )  
}
```



# dialogue 變數

- `dialogue`裡要存我們原本會放在`<ChatTurn/>`裡的attributes。
- 為了方便, 原本給`<ChatTurn/>`的`prompt`和`reply`就把它當作一個物件(object, 用`{...}`包起來的)
- 然後`dialogue`有很多這樣的物件, 所以它是一個array(用`[...]`包起來)。我們先試圖複製出原本的樣子, 所以只放兩個。

```
<ChatTurn
  prompt="Prompt Message - 1"
  reply="Reply Message - 1"/>
```

```
let dialogue = [
  {
    prompt: "Prompt - 1",
    reply: "Reply - 1",
  }, {
    prompt: "Prompt - 2",
    reply: "Reply - 2"
  }
];
```

# dialogue.map(...)

- 在往下一步前，來理解一下 `.map`
- `.map` 完全就是for迴圈。但由於JSX的語法限制，不能直接寫for-loop。
- `dialogue.map(f(x)=>{...})` 意思是「把 `dialogue` 中的每個元素都用 `f(x)` 對應到(map to)一個新的元素」
- `.map` 的回傳值就是這個新元素所構成的新array。

```
> let dialogue = [
  {
    prompt: "Prompt - 1",
    reply: "Reply - 1",
  }, {
    prompt: "Prompt - 2",
    reply: "Reply - 2"
  }
];

< undefined

> dialogue
< ▼ (2) [{...}, {...}] ⓘ
  ▶ 0: {prompt: 'Prompt - 1', reply: 'Reply - 1'}
  ▶ 1: {prompt: 'Prompt - 2', reply: 'Reply - 2'}
  length: 2
  ▶ [[Prototype]]: Array(0)

> dialogue.map((x)=>x)
< ▶ (2) [{...}, {...}]

> dialogue.map((x)=>x.prompt)
< ▶ (2) ['Prompt - 1', 'Prompt - 2']
```

# dialogue 變 <ChatTurn>s

- 從JS角度看起來有點花式，但這在React裡卻是慣用的 (idiomatic)
- 每個dialogue的元素都要被對應到 (map to) `<ChatTurn />`，而且每個 `<ChatTurn />` 的 attributes 就用元素的 attributes。
- `(...)` 和 `{...}` 可以交錯出現：用 `(...)` 換到JSX後，還是可以用 `{...}` 換到JS

```
dialogue.map((dialogue_x, idx) => {  
  return [A HTML element]  
})
```



```
dialogue.map((dialogue_x, idx) => {  
  return (<ChatTurn />)  
})
```



```
dialogue.map((dialogue_x, idx) => {  
  return (  
    <ChatTurn key="a key"  
      prompt="a prompt"  
      reply="a reply" />  
  )  
})
```



```
dialogue.map((dialogue_x, idx) => {  
  return (  
    <ChatTurn key={`dialogue_${idx}`}  
      prompt={dialogue_x.prompt}  
      reply={dialogue_x.reply} />  
  )  
})
```

# Pure ChatBox()

- 這個函數開始有故事了，每call一次 `ChatBox()` (在App.js)
  - 製作出 `dialogue` 變數，裡面放我們的對話清單。
  - 然後用這個對話清單產生出 `<ChatTurn/>` 元素
- 每次呼叫 `ChatBox()`，都會產生同樣的一組 `<ChatTurn/>`

```
export default function ChatBox() {
  let dialogue = [
    {
      prompt: "Prompt - 1",
      reply: "Reply - 1"
    },
    {
      prompt: "Prompt - 2",
      reply: "Reply - 2"
    }
  ];

  return (
    <div className="w-50 mx-auto mt-5 fs-4">
      {dialogue.map((dialogue_x, idx) => {
        return (
          <ChatTurn key={`dialogue_${idx}`}
            prompt={dialogue_x.prompt}
            reply={dialogue_x.reply} />)
        )}}
      [...]
    </div >
  )
}
```

# 有state的ChatBox()

- 但好像不對，如果每次呼叫 `ChatBox()`，都是同樣的 `<ChatTurn/>` 那介面永遠不會更新？
- 如果我們期待ChatBox要更新，那代表裡面有東西在動，這個東西就是「對話狀態」。
- 用 `useState` 把狀態標出來。下一步才會真的用到state

```
export default function ChatBox() {
  const [dialogue, setDialogue] = useState(
    [{
      prompt: "Prompt - 1",
      reply: "Reply - 1"
    },
    {
      prompt: "Prompt - 2",
      reply: "Reply - 2"
    }
  ]
);

  return (
    <div className="w-50 mx-auto mt-5 fs-4">
      {dialogue.map((dialogue_x, idx) => {
        return (
          <ChatTurn key={`dialogue_${idx}`}
            prompt={dialogue_x.prompt}
            reply={dialogue_x.reply} />)
        )
      })}
      [...]
    </div >
  )
}
```



# 04 更新狀態

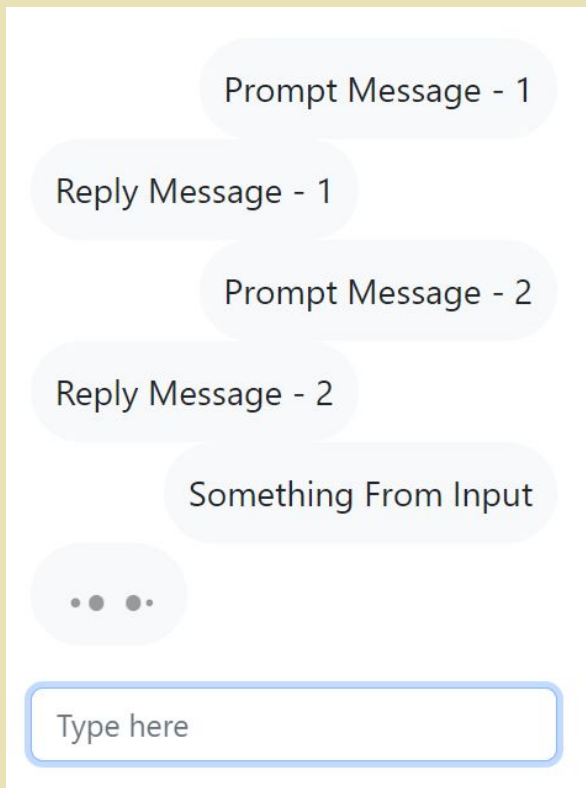
Back



Next

# 更新狀態

- Checkpoint: [Github link](#)
- 讓使用者在文字方塊中打的文字出現在對話視窗中。
- 同時在回覆端顯示一個spinner, 代表對方回覆中。
- 這步驟還不會真的做出回覆。
- 從Step 4到Step 5 [Diff](#)



# 因為要顯示Spinner

- 網路上有很多現成的Spinner, 所以這是loading.io現成的CSS。
- 雖然大部分跟React無關, 但還是有多幾個檔案和調整:
  - public/spinner.css: spinner的CSS樣式檔
  - src/spinner.js: 把Spinner包成一個React Element
  - public/index.html : 把CSS引入html

```
export default function Spinner() {  
  return (  
    <div className="lds-ellipsis">  
      <div></div><div></div><div></div><div></div>  
    </div>  
  )  
}
```

```
<link rel="stylesheet" href="%PUBLIC_URL%/spinner.css"></style>
```



# 先設定文字方塊的事件

- 基本上HTML有什麼事件, JSX 就可以用什麼事件。
- 但因為某些原因, 我們這邊用的事件是 `onKeyDown`, 他會在文字方塊有任何按鍵按下的時候發出事件。
- 注意JSX的事件是直接傳「要執行的」函數本身 (不能call)

```
function onInputHandler(event) {  
  console.log(event.target.key);  
  console.log(event.target.value);  
}
```

```
// In JSX:
```

```
<input type="text"  
  className="form-control form-control-lg"  
  placeholder="Type here"  
  onKeyDown={onInputHandler}/>
```

# 先設定文字方塊的事件

- 這裡目的是使用者按下Enter後，才會真正開始回覆 (不是 auto-completion)，所以我們只處理Enter事件。
- `Array.from(dialogue)` 目的是複製出一份新的 `dialogue` 變數，當作新的狀態。
- 不必然要這麼做，但會讓整個程式好理解很多。

```
function onInputHandler(event) {  
  if (event.key==="Enter"){  
    console.log(event.target.key);  
    console.log(event.target.value);  
    let new_dialogue = Array.from(dialogue);  
  }  
}  
  
// In JSX:  
<input type="text"  
  className="form-control form-control-lg"  
  placeholder="Type here"  
  onKeyDown={onInputHandler}/>
```

# [Optional] JS的變數特性

- JS的變數(特指陣列或物件)都是可變的。即便指派到不同變數,兩個名字還是指同一個東西(完全是同義詞)。
- 我把一個**錢包**叫「A」,你把**同個錢包**叫「B」;但我們花錢都是是花**同個錢包**。
- 這種可變性會讓程式很難思考,所以解決方法之一是乾脆複製一份出來。Array.from就只是用來複製陣列而已。

```
> let a = [1,2,3]
< undefined
> b = a
< ▶ (3) [1, 2, 3]
> b.push(100)
< 4
> a
< ▶ (4) [1, 2, 3, 100]
> x = Array.from(a)
< ▶ (4) [1, 2, 3, 100]
> x.push(999)
< 5
> a
< ▶ (4) [1, 2, 3, 100]
> x
< ▶ (5) [1, 2, 3, 100, 999]
```

# 先設定文字方塊的事件

- 我們有了複製出來的新「錢包」後，就可以放新東西了
- `.push`是在陣列末端加入新的元素
- 然後用`setDialogue`跟React說，這是新錢包，接下來我們要用這個新錢包了。

```
function onInputHandler(event) {  
  if (event.key==="Enter"){  
    console.log(event.target.key);  
    console.log(event.target.value);  
    let new_dialogue = Array.from(dialogue);  
    new_dialogue.push({  
      prompt: event.target.value.trim(),  
      reply: ""  
    });  
    event.target.value = "";  
    setDialogue(new_dialogue);  
  }  
}  
  
// In JSX:  
<input type="text"  
  className="form-control form-control-lg"  
  placeholder="Type here"  
  onKeyDown={onInputHandler}/>
```

# useState()

- useState不是「純函數」，它每次回傳的dialogue不見得是一樣的。
- setDialogue讓我們跟React說，dialogue的狀態改變了，頁面需要更新。React會再呼叫一次ChatBox()
- useState就會回傳React現在知道的最新狀態是什麼。

```
export default function ChatBox() {
  const [dialogue, setDialogue] = useState(
    [...]);

  function onInputHandler(event){
    if (event.key==="Enter"){
      [...]
      setDialogue(new_dialogue);
    }
  }

  return (
    <div className="w-50 mx-auto mt-5 fs-4">
      {dialogue.map((dialogue_x, idx) => {
        return (
          <ChatTurn key={`dialogue_${idx}`}
            prompt={dialogue_x.prompt}
            reply={dialogue_x.reply} />)
        )}}
    </div>
  )
}
```

# 加上Spinner

- 剛剛狀態更新都會加入一整個turn, 包含prompt和reply。
- reply還沒有產生, 所以是空字串
- 在ChatTurn, 我們用JSX來控制是要顯示reply或Spinner
  - 空字串會被當成false, 字串裡有任何內容都會是true
  - `<cond>? <x-if-true>: <y-if-false>`

```
> "some string"? 99: 0
< 99
> ""? 99: 0
< 0
```

```
import Spinner from "./spinner";

export default function ChatTurn({
  prompt, reply
}) {
  return (
    <div className="[...]">
      <div className="[...]">
        {prompt}
      </div>

      <div className="[...]">
        {reply ? reply: (<Spinner/>)}
      </div>
    </div>
  )
}
```



# 05

## 回應對話

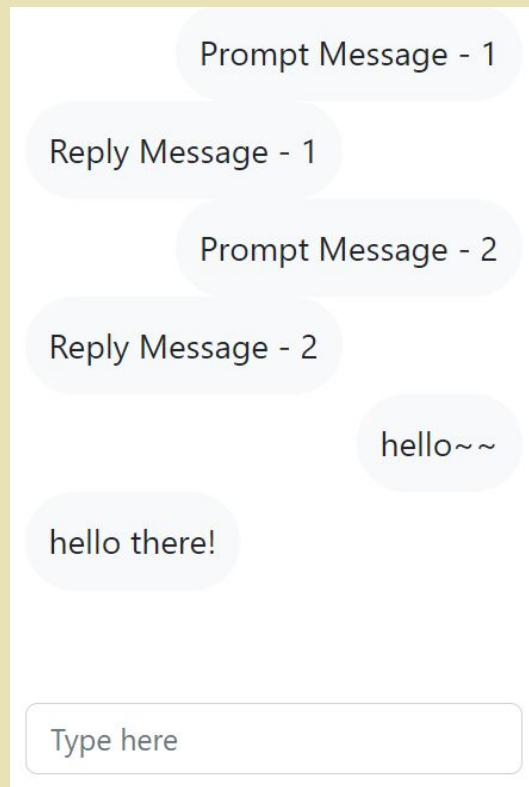
Back



Next


# 回應對話

- Checkpoint: [Github link](#)
- 這步要讓使用者打入訊息後，畫面要出現100ms 的Spinner，然後固定回覆”Hello there!”
- 這步會用setTimeout模擬API的回覆延遲，但不會真正對接(interface) API。
- 從Step 5到Step 6 [Diff](#)







# 模擬回覆產生



```
> setTimeout(() => {  
  console.log("123")  
}, 500)  
36  
123
```

- 傳送API要求(request), 到取得回應 (response)之間需要一點時間(~1s)。瀏覽器把這步做成非同步的函數。
  - 也就是我們不能直接寫成reply=process\_prompt(...)。我們一定會需要一個回呼函數(callback), 讓瀏覽器通知我們它已經取得回應了。
  - 這個步驟我們用setTimeout來模擬這個過程。這樣也可以確定程式的其他部分在非同步狀況下的行為是對的。
- 
- 

# 模擬回覆產生

- 我們另外寫一個 `process_prompt` 函數來處理這些事情。
- 它會需要兩個參數，`prompt_text`，和目前的 `dialogue` 狀態。
- 我們先測試看看 `process_prompt` 的執行順序。

```
function onInputHandler(event) {  
  if (event.key === "Enter") {  
    let new_dialogue = Array.from(dialogue);  
    const prompt_text = event.target.value.trim();  
    new_dialogue.push({  
      prompt: prompt_text,  
      reply: ""  
    });  
    event.target.value = "";  
    ➡ process_prompt(prompt_text, new_dialogue);  
    setDialogue(new_dialogue);  
  }  
}  
  
function process_prompt(intext, dialogue_data) {  
  console.log("process_prompt called");  
  setTimeout(() => {  
    console.log("setTimeout callback");  
  }, 500)  
  console.log("process_prompt exited");  
}
```

# 模擬回覆產生

- 現在把回覆的更新邏輯寫進函數裡。
- 同樣先複製一份狀態出來。然後取出最後一個turn。
- 把回覆寫進最後那輪對話的reply。
- 然後用setDialogue設定新的對話狀態。

```
function process_prompt(intext, dialogue_data) {  
  setTimeout(() => {  
    let new_dialogue = Array.from(dialogue_data);  
    let last_turn = new_dialogue[new_dialogue.length - 1];  
    last_turn.reply = "hello there!"  
    setDialogue(new_dialogue);  
  }, 100)  
}
```

# 防止使用者送出多個對話

- 我們更新對話的時候是取 dialogue 的最後一個 turn。
- 但如果使用者在 API 回覆之前輸入多組對話，那更新的時候我們要管理很多不同版本的對話狀態。
- 為了簡化程式，我們乾脆讓使用者送出訊息，直到回覆之前，都沒辦法再用文字方塊輸入其他文字。

```
return (
  [...]
  <div className="mx-auto mt-1 fs-4">
    <input type="text" disabled={isProcessing}
      className="form-control form-control-lg"
      placeholder="Type here"
      onKeyDown={onInputHandler} />
  </div>
</div>
</div >
)
```

新增一個 state 控制文字方塊是否停用

# 控制文字方塊停用狀態

- 新增一個`useState()`，預設值是`false`，代表文字方塊啟用。
- 當`process_prompt`一被呼叫的時候就把它設定為`true`，文字方塊停用。
- 直到回覆可以被寫入對話狀態時，再設定為`false`，文字方塊啟用。

```
export default function ChatBox() {  
  const [dialogue, setDialogue] = useState([...])  
  const [isProcessing, setIsProcessing] = useState(false);  
  
  function process_prompt(intext, dialogue_data) {  
    setIsProcessing(true);  
    setTimeout(() => {  
      let new_dialogue = Array.from(dialogue_data);  
      let last_turn = new_dialogue[...];  
      last_turn.reply = "hello there!"  
      setIsProcessing(false);  
      setDialogue(new_dialogue);  
    }, 100)  
  }  
  [...]
```



# 06

## 對接API

Back



Next

# 對接API

- Checkpoint: [Github link](#)
- 萬事俱備，只剩對接API了。
- 把剛剛申請的API放進 .env.local
- 建立一個新檔案api.js，負責對接API的前後處理工作。
- 從Step 6到Step 7 [Diff](#)

你好阿

 bigscience/mt0-xxl-mt 

電影看得好想睡， positive or negative?

negative

Type here

# 讓React找得到HF\_TOKEN

```
// .env.local  
REACT_APP_HF_TOKEN="hf_alsKevbH..."
```

- lab/deepdive/wander底下有個檔案叫 `.env`。
- 請複製一份那個檔案，然後改名為 `.env.local`。確定這個檔案有在 `.gitignore`
- 把剛剛申請的Access token複製進字串裡。引號要留著。
- 這樣在React裡就可以用 `process.env.REACT_APP_HF_TOKEN` 使用這組字串
- 開發時，盡量不要把自己的access token寫進原始碼裡。因為我們無法確定原始碼只會在私人開發環境裡，而且很容易不小心push到GitHub上。



# 呼叫HF API

- 這是HuggingFace提供的[sample code](#)
- 我們稍微做點包裝，讓它多接受一個API Token參數。
- 但這個函數是async，所以它回傳的是一個Promise

```
async function query(data, api_token) {  
  const response = await fetch(  
    "https://api-inference.huggingface.co/models/bigscience/mt0-xxl-mt",  
    {  
      headers: { Authorization: `Bearer ${api_token}` },  
      method: "POST",  
      body: JSON.stringify(data),  
    }  
  );  
  const result = response.json();  
  return result;  
}
```

getImage



readFile

STATUS

pending

VALUE

undefined

```
.then(res => console.log(res))
```

```
.catch(err => console.log(err))
```



node

```
> getImage("./image.png")  
  .then(res => console.log(res))  
  .catch(err => console.log(err))
```

# [optional] Promise vs. async/await

- 兩者幾乎只是語法上的差異。
- await只能出現在async func裡。
- 如果比喻有幫助的話：
  - Promise是一個盒子，裡面裝著結果。
  - .then是我手動把盒子打開，對裡面的東西做事
  - await是JS幫我把東西取出盒子，並把盒子丟掉

```
function with_promise() {  
  getImage("./image.png")  
    .then(res => console.log(res));  
}  
  
async function with_await() {  
  let res = await getImage("./image.png")  
  console.log(res);  
}
```

# 包裝底層的query

- 在api.js裡，我們另外包一層函數 `process_api`，讓它可以幫我們載入access token，做寫簡單的前處理，再送出要求。
- 等回傳後，再對訊息做些後處理，最後回傳回覆。
- `await` 一定是接一個Promise，而且只能在`async`裡

```
export async function process_api(prompt_text){
  const HF_TOKEN = process.env.REACT_APP_HF_TOKEN;
  const query_prompt = prompt_text.replace(/[\.\!\]\$]/, " ");
  const response = await query({ inputs: query_prompt }, HF_TOKEN);
  const gentext = response[0].generated_text;
  const reply_text = gentext.replace(new RegExp(`^${query_prompt}`), "");
  return reply_text;
}
```

# Promise.then

- 有了 `process_api`, 接下來就只是處理Promise而已了。
- `.then(...)` 裡的回呼函數會把Promise的結果當引數(argument)傳入, 就是這裡的reply。
- 其他的更新邏輯都和前一步驟是相同的。

```
function process_prompt(intext, dialogue_data) {
  setIsProcessing(true);
  const process_handler = process_api;
  // const process_handler = debug_api;
  process_handler(intext).then((reply) => {
    let new_dialogue = Array.from(dialogue_data);
    let last_turn = new_dialogue[new_dialogue.length - 1];
    console.log("API respond: ", reply);
    if (reply) {
      last_turn.reply = reply;
    } else {
      last_turn.reply = "我不知道🙄";
    }
    setIsProcessing(false);
    setDialogue(new_dialogue);
  });
}
```

# Promise.then(...)

- `process_handler`的用意只是方便debug用。`api.py`裡有`process_api`和`debug_api`。多一個變數只是方便切換。
- `reply`的條件式是一個回應修飾手段。萬一API沒有產生任何回覆，就丟一個罐頭回覆。

```
function process_prompt(intext, dialogue_data) {
  setIsProcessing(true);
  const process_handler = process_api;
  // const process_handler = debug_api;
  process_handler(intext).then((reply)=>{
    let new_dialogue = Array.from(dialogue_data);
    let last_turn = new_dialogue[new_dialogue.length - 1];
    console.log("API respond: ", reply);
    if (reply){
      last_turn.reply = reply;
    } else {
      last_turn.reply = "我不知道🙄";
    }
    setIsProcessing(false);
    setDialogue(new_dialogue);
  });
}
```

# All done!

顯然沒有ChatGPT聰明，但還滿可愛的。

畢竟，參數量差了10倍不止，也別要求太多了。

你好阿

 bigscience/mt0-xxl-mt 

臺大的環境鬱鬱蔥蔥。translate to English

Taiwan has a very busy atmosphere.

Taiwan has a very busy atmosphere. Translate to Chinese

台湾的空气非常燥热。




遠望那玉山突出雲表 正象徵我們目標的高崇 近看蜿蜒的淡水  
他不捨晝夜地流動 正顯示我們百折不撓的作風 這百折不撓的作  
風 定使我們 一切事業都成功。什麼東西是蜿蜒的

淡水

Type here

# 結語

- 寫React真的不容易，從頭到尾頭腦要很清楚。基本的步驟：
  - 用完全靜態的想法，先把HTML寫出來，即便先全部塞在App.js也沒關係。
  - 一步步分析出相對而言獨立的（視覺的、功能的）元件
  - 加入props、加入events、加入states
  - 先別想著效能，東西會動最重要。



```
▼ wander
  > node_modules
  > public
▼ src
  JS api.js
  # App.css
  JS App.js
  JS App.test.js
  JS chat-turn.js
  JS chatbox.js
  # index.css
  JS index.js
  logo.svg
  JS reportWebVitals.js
  JS setupTests.js
  JS spinner.js
  .env
  .env.local
  .gitignore
  package.json
  README.md
  yarn.lock
```



# Happy React-ing



Have a great weekend

I hope your code behaves on Monday  
the same way it did on Friday



NLP與網路應用  
React & HF-API