06. Advanced Topics on React.js

Ric Huang / NTUEE

(EE 3035) Web Programming

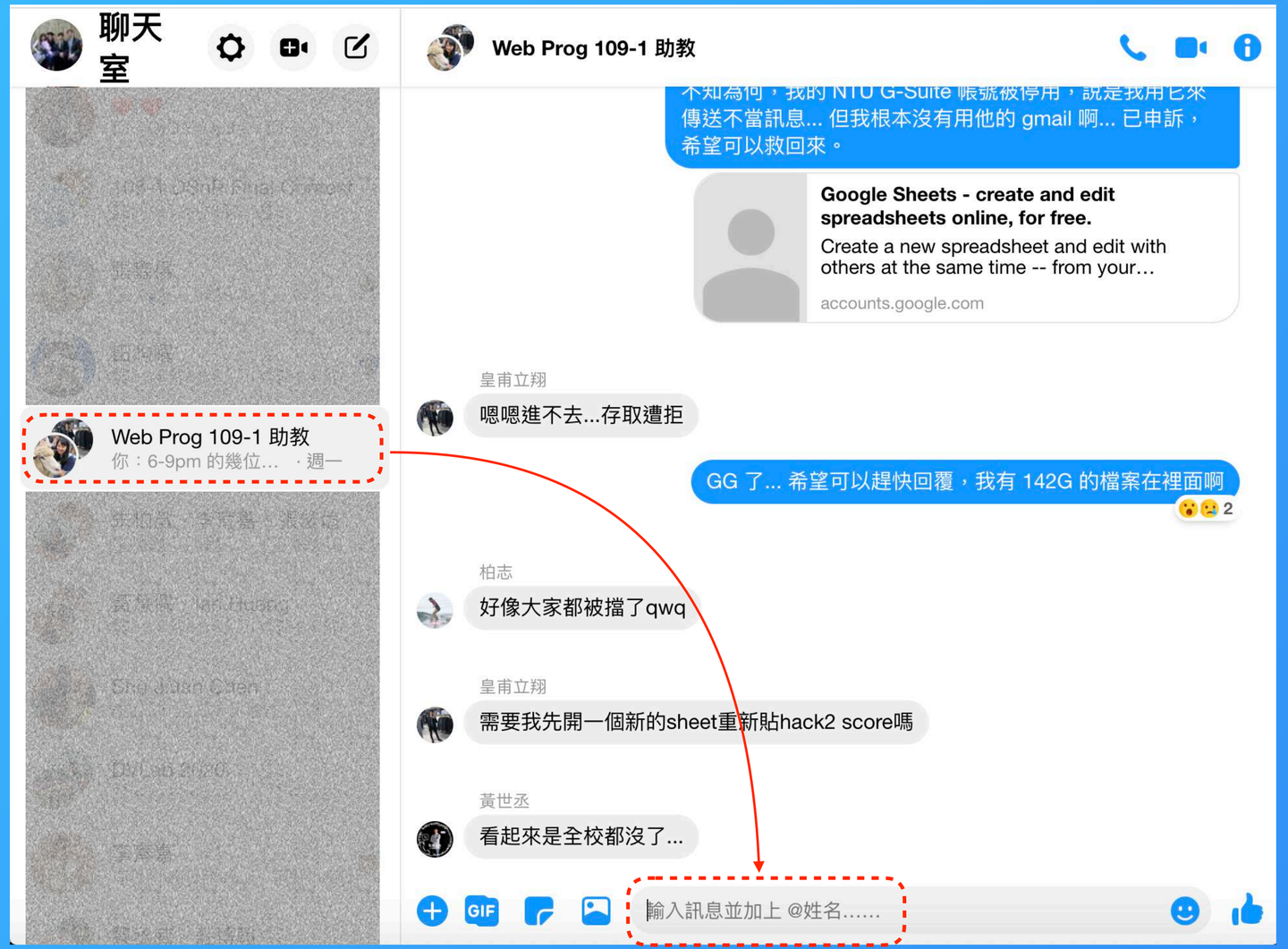
我們還有最後一些 advanced React topics 尚未教完

- Refs
- Context / Redux
- props types
- React router

Ref in React

- 一個常見的 UX 設計：

  當使用者的 mouse click 在某個通訊者的名字時，右邊的視窗除了切換對話記錄之外，也會直接將 keyboard 的 focus 放在 input box 上面



Refs provide a way to access DOM nodes or React elements created in the render method.

An Input Box without Ref

```
class CustomTextInput extends React.Component {
  constructor(props) { super(props); }
  render() {
    return (
      <div>
        <input type="text" />
        <input type="button"
               value="Focus the text input" />
      </div>
    );
  }
}
export default CustomTextInput;
```

- 按了按鈕後 Keyboard 不會自動 focus 在 input box
  => 因為按鈕沒有辦法控制到 input box 的 this

Using React Ref

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  focusTextInput = () => this.textInput.current.focus();

  render() {
    return (
      <div>
        <input type="text" ref={this.textInput} />
        <input type="button"
               value="Focus the text input"
               onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

1. Create 一個 ref variable (as a state)

2. 將這個 ref variable 指到 text input box

3. 利用 “current” 這個 property 來指到對應的 DOM node (i.e. text input box)

4. 綁定 onClick 的 event handler

Using React Ref + Life Cycle Method

- 但上頁的寫法有一個缺點，就是一開始進入畫面時，input box 是沒有被 focus 的...
  => 可以在 componentDidMount() 手動呼叫 focusTextInput()

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }
  focusTextInput = () => this.textInput.current.focus();

  componentDidMount() {
    this.focusTextInput();
  }
  render() {
    return (...
    );
  }
}
```

使用 "useRef" Hook，簡單很多

```
function CustomTextInput() {
  const textInput = useRef();
  const focusTextInput = () => textInput.current.focus();
  useEffect(() => focusTextInput());

  return (
    <div>
      <input type="text" ref={textInput} />
      <input type="button"
             value="Focus the text input"
             onClick={focusTextInput}
      />
    </div>
  );
}
```

1. 使用 useRef 產生一個 local state variable (as a reference)

2. 不用 this 即可將這個 ref 綁定 text input DOM

3. 一樣利用 “current” 這個 property 來指到對應的 DOM node (i.e. text input box)

4. 用 local function object 直接綁定 onClick 的 event handler

5. 使用 useEffect() 確定每次 render 都會 re-focus

Forwarding Refs

- For automatically passing a ref through a component to one of its children.
- 通常不需要也不應該去 reference 到 child component.
- 不過，有時候對於常常被呼叫，但不想 reveal implement 細節的 utility components, 我們會建個 wrapper component 將 child component 的 DOM node forward 過來

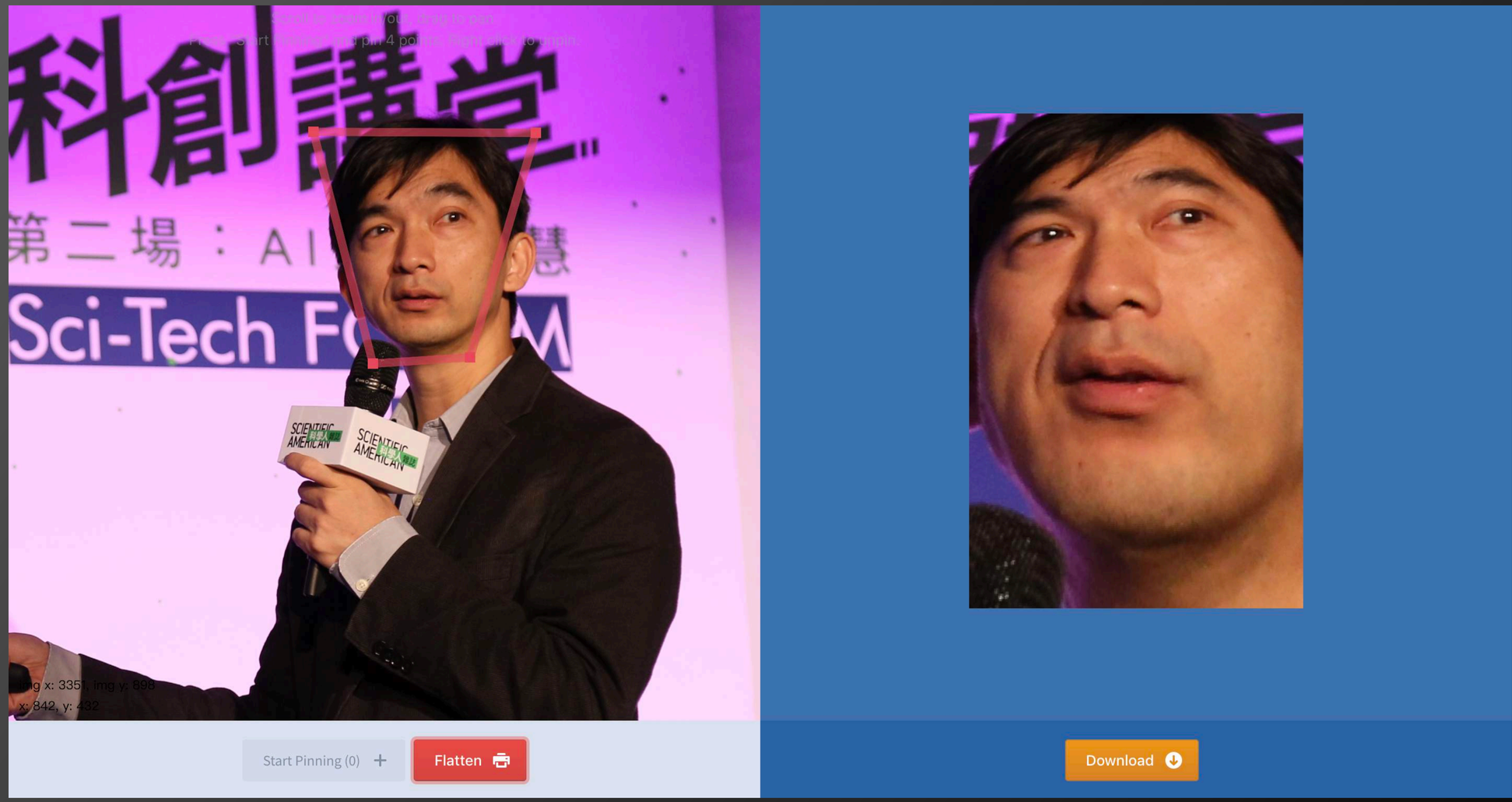
```
function FancyButton(props) {
  return (
    <button className="FancyButton">
      {props.children}
    </button>
  );
}
```

對於上層 render <FancyButton> 的 function 而言,
它無法直接 access button 的 DOM node,
因此，一些 event 的控制就變得相當麻煩

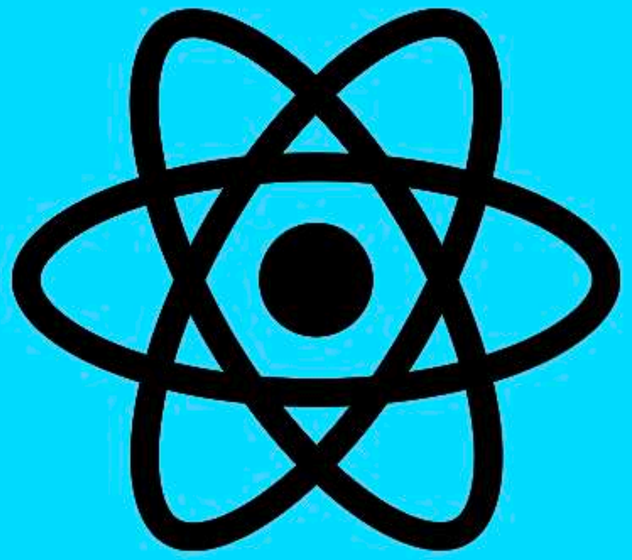
Forwarding Refs

```
const FancyButton = React.forwardRef((props, ref2) => (
  <button ref={ref2} className="FancyButton">
    {props.children}
  </button>
));

_____________


const ref1 = React.createRef();
<FancyButton ref2={ref1}>Click me!</FancyButton>;

// ref1.current now refers to <button>'s DOM node
```

More “ref” Example



Ref: https://github.com/ian13456/image-scanner



React Context
+
Hooks

VS

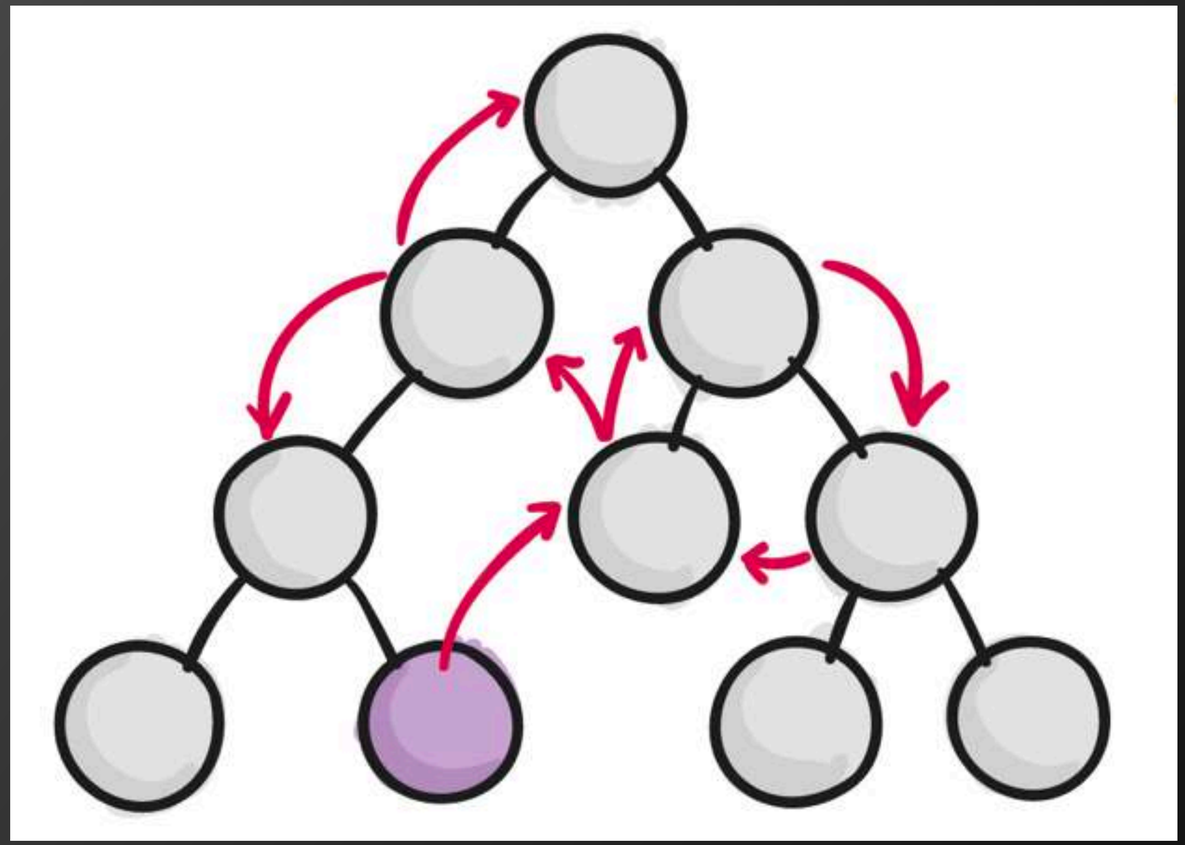
Redux

Recall: React props and state

- "this.props" are read-only
  - You cannot assign or change values to this.props
- "state" is private to the class
  - You cannot pass in value to it
- React 的出現，已經讓前端的可預測性大大提高

但當 React Web App 複雜到一定程度的時候...

- Components 的關係錯綜複雜，states 的邏輯也隱晦在不同的 components 當中，越來越難理解與維護



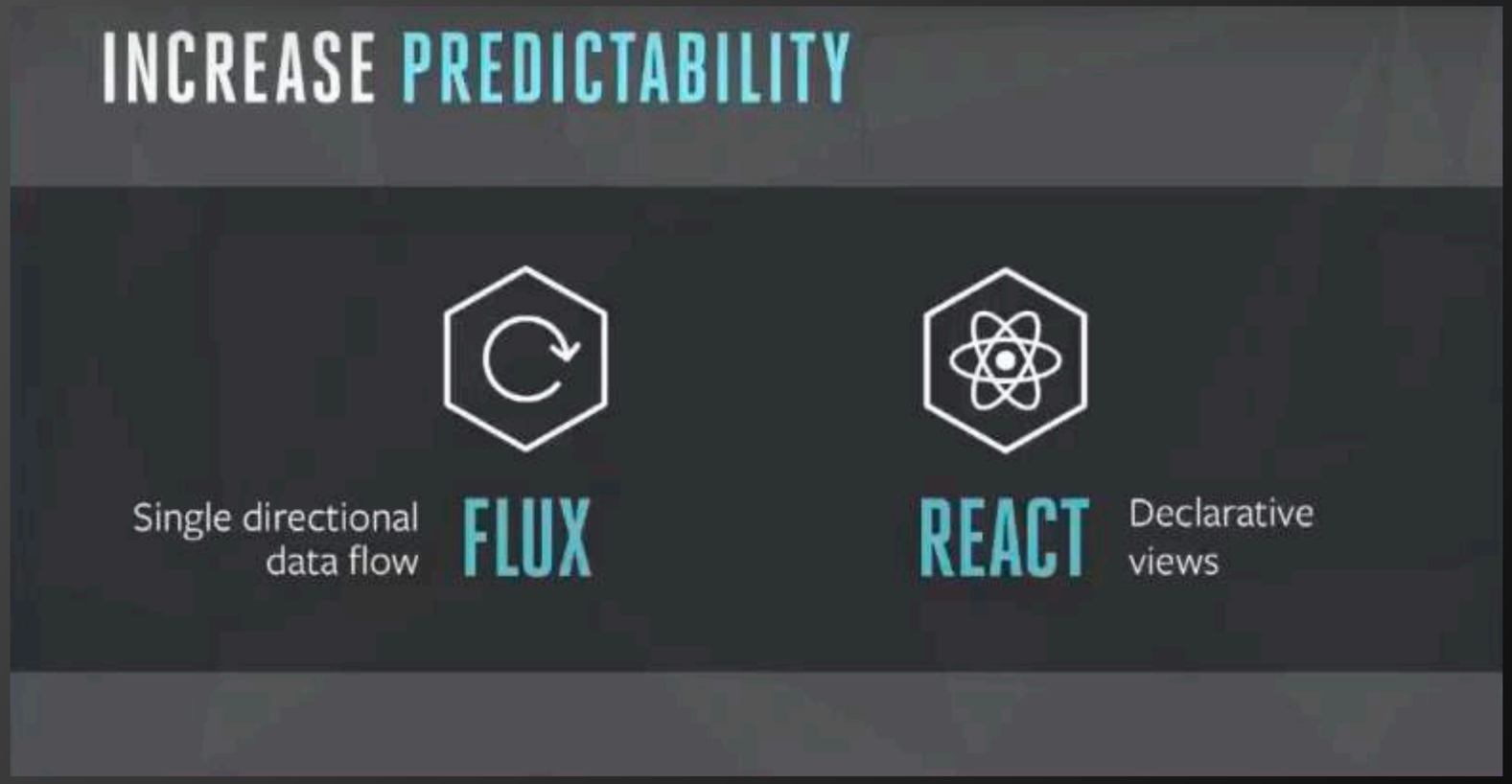
States 的愛恨情仇 [ref]

- Multiple React components needs to access the same state but do not have any parent/child relationship
- You start to feel awkward passing down the state to multiple components with props

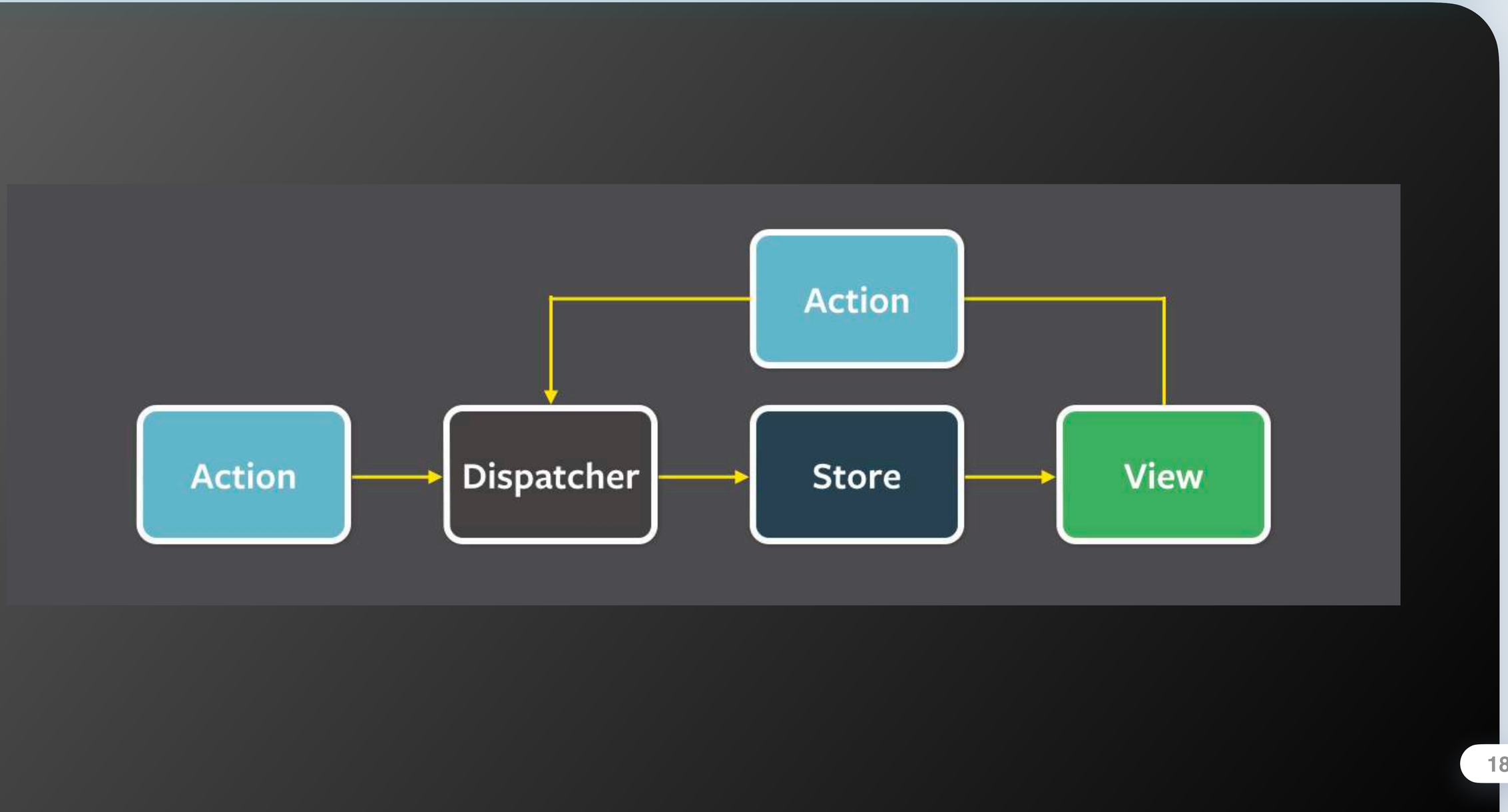
能不能集中管理
State ?

Flux: Single Directional Data Flow

- 大概從 React 公開不久，state & data flow 就是一個討論度很高的問題，其中 "Flux" 是一個早期的 solution, 但缺點是較為複雜，因此，大家還是一直在找比較好的替代方案



Unidirectional Data Flow

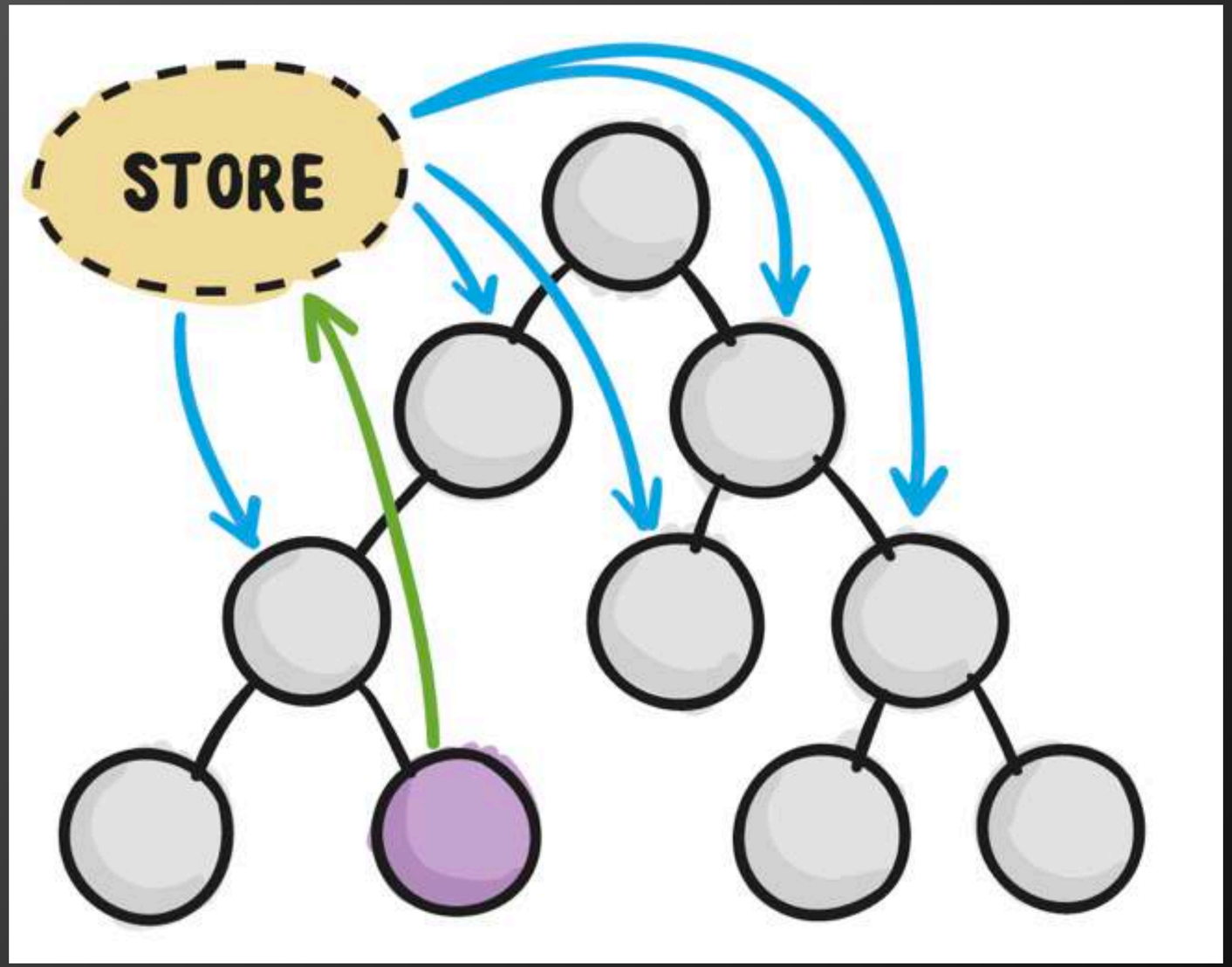


Flux 因為一些歷史因素和
本身的缺點，被另一套更簡單
的框架 Redux 取代

Redux

- Dan Abramov 在參加 React EU Conference 意外做出來的

集中管理的 state

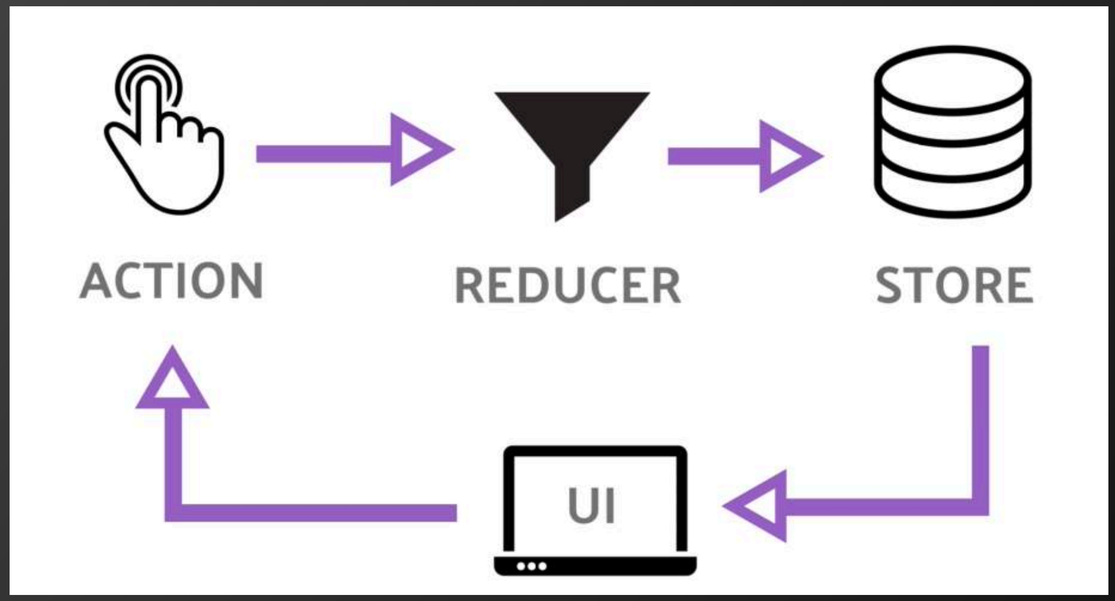


Redux 三大原則

1. Single source of Truth
   - Store: 整個前端 App 的 state 全存在唯一的樹狀 store 裡面
2. The only way to change the state is by sending a signal to the store
   - Action: 改變 store 的唯一方式是送一個描述改變的 object (i.e. dispatching an action)
3. The state is immutable and cannot change in place**
   - Reducer: store 根據 action 決定 state 如何變化，但必須寫成一個 Pure Function

A quick glance on Redux

- Store: a single place to store all the states for the app
- Action: a object that contains two properties: type and payload, to describes what updates the states will be
- Reducer: a function that takes the current state and an action as inputs and determines the next state



A simple practice on Redux [ref]

- First let's create a new React project and change "src/App.js" to the following:

```
import React, { Component } from "react";
class App extends Component {
  constructor() {
    super();
    this.state = {
      articles: [
        { title: "React Redux Tutorial for Beginners", id: 1 },
        { title: "Redux e React: cos'è Redux e come usarlo con React", id: 2 }
      ]
    };
  }
  render() {
    const { articles } = this.state;
    return <ul>{articles.map(el => <li key={el.id}>{el.title}</li>)}</ul>;
  }
}
export default App;
```

“yarn start”, what do you see?

Adding "Redux" to this example (1)

- "yarn add redux"
- Create the "Store"

```
// src/js/store/index.js
import { createStore } from "redux";
import rootReducer from "../reducers/index";
const store = createStore(rootReducer);
export default store;
```

Adding "Redux" to this example (2)

- Define constants first

```
// src/js/constants/action-types.js
export const ADD_ARTICLE = "ADD_ARTICLE";
```

- Define the "Action"

```
// src/js/actions/index.js
import { ADD_ARTICLE }
        from "../constants/action-types";
export function addArticle(payload) {
  return { type: ADD_ARTICLE, payload };
}
```

Adding "Redux" to this example (3)

- Define the "Reducer"

```
// src/js/reducers/index.js
import { ADD_ARTICLE }
        from "../constants/action-types";
const initialState = {
  articles: []
};
function rootReducer(state = initialState, action) {
  if (action.type === ADD_ARTICLE) {
    state.articles.push(action.payload);
  }
  return state;
}
export default rootReducer;
```

Adding "Redux" to this example (4)

- However, you should make the reducer "pure"
- Change this line:

```
      state.articles.push(action.payload);
```

- To —

```
      return Object.assign({}, state, {
        articles:
          state.articles.concat(action.payload)
      });
```

Let's stop and see what we have now...

- We have a React app that defines an App with a init state of two articles in an array

```
src/index.js
src/App.js
```

- We have defined a store, reducer, and an action in —

```
src/js/store/index.js
src/js/reducers/index.js
src/js/actions/index.js
```

However, the Redux store and React App are not connected...

How do we connect Redux state and React App?

- First let's see how store is operated
- To test it, add "src/js/index.js" to expose the store to a "window" property so that we can test it on console

```
import store from "../js/store/index";
import { addArticle } from "../js/actions/index";
window.store = store;
window.addArticle = addArticle;
```

- Modify "src/index.js" as:

```
import index "./js/index";
```

Testing "store"

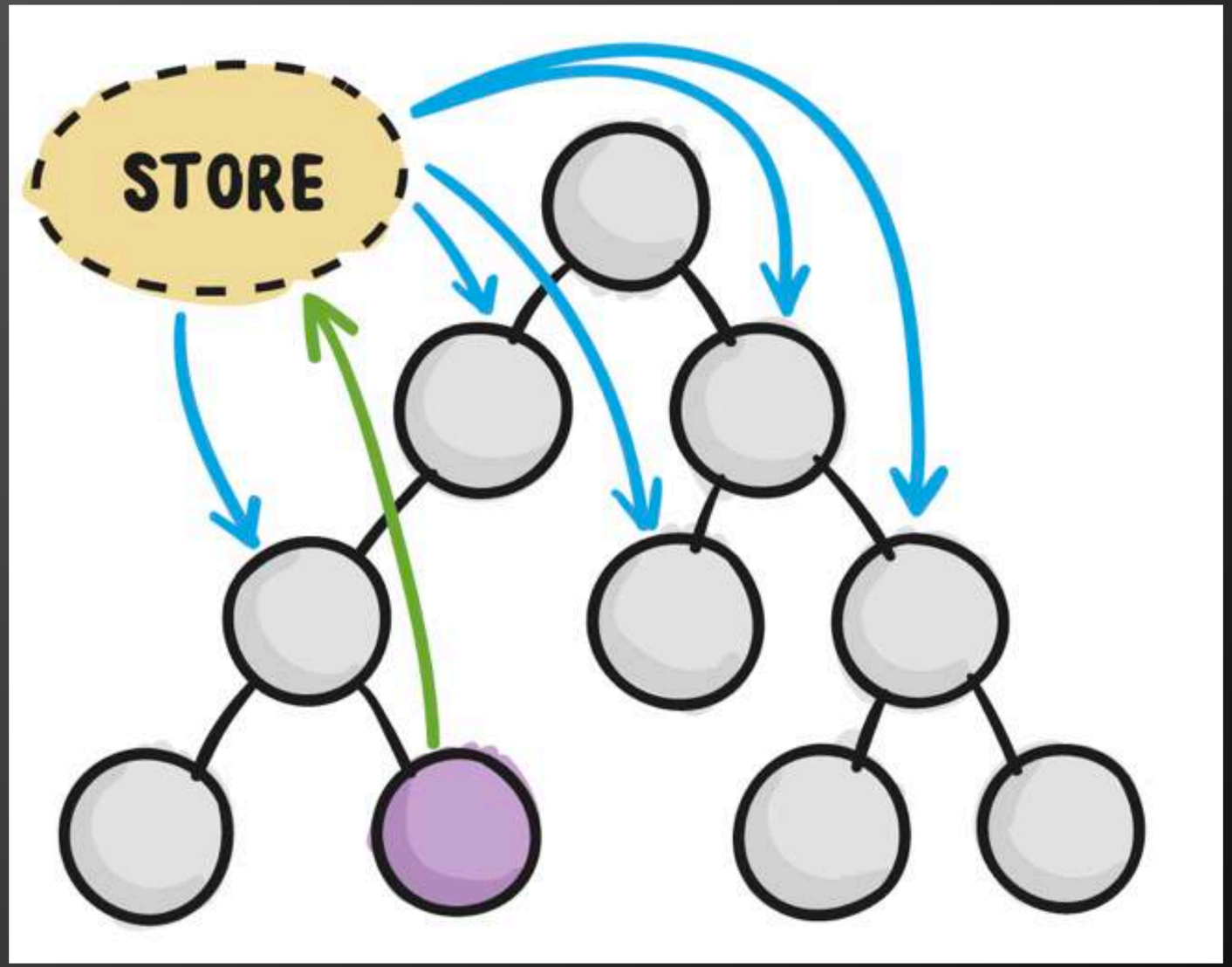
- "yarn start" and open console for "localhost:3000"
- Test the following:

```
// Check the state now
store.getState()
// The subscribe method accepts a callback that will fire whenever an action is dispatched.
store.subscribe(() => console.log('Look ma, Redux!!'))
// Manually dispatch an action
store.dispatch(addArticle
({ title: 'React Redux Tutorial for Beginners', id: 1 }) )
// Check the state again!
store.getState()
```

所以結論是...

- 我們可以透過 console (i.e. expose "store" to "window") 來直接操作 store 的 methods，更改 state 裡頭的資料
- Yes, Redux is framework agnostic. 你可以把 Redux 用在任何的 framework, 除了 React, 也可以用在 Angular, Venilla, etc...
- 那到底要怎麼把 Redux 用在 React 呢？
- 重點是：要如何讓 React 的 states 被 connected 到 Redux 的 store 呢？

像前幾頁的這張圖...



React-Redux

- For React to use Redux, you should install react-redux

```
yarn add react-redux
```

Provider: the wrapper

- We will first use "Provider", an high order component coming from react-redux which wraps up your React application and makes it aware of the entire Redux's store.

Linking Redux to React (1)

- Modify "src/js/index.js" as:

```
import React from "react";
import { render } from "react-dom";
import { Provider } from "react-redux";
import store from "./js/store/index";
import App from "./js/components/App.jsx";
render(
  <Provider store={store}>
    <App />
  </Provider>,
  document.getElementById("root")
);
```

Linking Redux to React (2)

- Create the List of articles

```
// src/js/components/List.jsx
import React from "react";
import { connect } from "react-redux";
const mapStateToProps = state => {
  return { articles: state.articles };
};
const ConnectedList = ({ articles }) => (
  <ul className="list-group list-group-flush">
    {articles.map(el => (
      <li className="list-group-item" key={el.id}>
        {el.title}
      </li>
    ))}
  </ul>
);
const List = connect(mapStateToProps)(ConnectedList);
export default List;
```

Linking Redux to React (3)

- Create Form.jsx

```
// src/js/components/Form.jsx
import React, { Component } from "react";
import { connect } from "react-redux";
import uuidv1 from "uuid";
import { addArticle } from "../actions/index";
function mapDispatchToProps(dispatch) {
  return {
    addArticle:
      article => dispatch(addArticle(article))
  };
}
class ConnectedForm extends Component {
  constructor() {
    super();
    this.state = {
      title: ""
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState
    ({ [event.target.id]: event.target.value });
  }
  handleSubmit(event) {
    event.preventDefault();
    const { title } = this.state;
    const id = uuidv1();
    this.props.addArticle({ title, id });
    this.setState({ title: "" });
  }
  render() {
    const { title } = this.state;
    return (
      <form onSubmit={this.handleSubmit}>
        <div className="form-group">
          <label htmlFor="title">Title</label>
          <input
            type="text"
            className="form-control"
            id="title"
            value={title}
            onChange={this.handleChange}
          />
        </div>
        <button type="submit"
          className="btn btn-success btn-lg">
          SAVE
        </button>
      </form>
    );
  }
}
const Form = connect(null, mapDispatchToProps)
(ConnectedForm);
export default Form;
```

Linking Redux to React (3)

- Create App.jsx

```
// src/js/components/App.jsx
import React from "react";
import List from "./List.jsx";
import Form from "./Form.jsx";
const App = () => (
  <div className="row mt-5">
    <div className="col-md-4 offset-md-1">
      <h2>Articles</h2>
      <List />
    </div>
    <div className="col-md-4 offset-md-1">
      <h2>Add a new article</h2>
      <Form />
    </div>
  </div>
);
export default App;
```

Linking Redux to React (4)

- Lastly, add the Bootstrap style
- Add this line to public/index.html

```
    <link rel="stylesheet" href="https://
maxcdn.bootstrapcdn.com/bootstrap/4.0.0-beta.2/css/
bootstrap.min.css" >
```

- "yarn start" to test it!!

Redux helps us solve the problem of the property chain and help manage the states in a proper way.

However, its setup is a bit complicated.

Since React 16.3, Context API was proposed to replace some of the usage of Redux...

【Context】 (ref)

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

When to Use Context

- Context is designed to share data that can be considered “global” for a tree of React components, such as the current authenticated user, theme, or preferred language.

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}
function Toolbar(props) {
  // The Toolbar component must take an extra
  // “theme” prop and pass it to the ThemedButton.
  // This can become painful if every single button
  // in the app needs to know the theme because it
  // would have to be passed through all components.
  return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );
}
class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.props.theme} />;
  }
}
```

Using “Context”

```
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}

function Toolbar() {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

1. 使用 createContext 產生一個 context (‘light’ 為 default)

2. 使用 Provider 來把 ThemeContext 往下傳，且 value 改成 ‘dark’

3. 使用 context 後，中間的 components 不用再傳 theme property 了

4. 使用 contextType property 來接受 Context object，並且使用 this.context 來獲得最近的 context

Using “useContext” hook

```
const ThemeContext = React.createContext('light');

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <Button theme={theme} />;
}
```

1. 改成 functional components

2. 用 useContext 將 context state 包成一個 local variable

Another Example: Creating Your useTodos Hook!

```
// 建立一個 Context
const TodoContext = React.createContext({
  todos: []
})
// 使用 ContextStore
function Application() {
  return (
    <TodoContext.Provider value={{todos: ['run']}}>
      <Todos />
    </TodoContext.Provider>
  )
}
// Todos
function useTodos() {
  const todos = useContext(TodoContext);
  return
    todos.map(todo => <div key={todo}>todo</div>)
}
export default useTodos
```

Define your own hook

我們還有最後一些 advanced React topics 尚未教完

- Refs
- Context / Redux
- props types
- React router

Typechecking With PropTypes

- 由於 JS 本身是弱型別的關係，browser 不時會很雞婆的幫你做型別的轉換，雖然有時這樣做會讓人覺得很方便，但也常常會因此而造成一些奇怪，很難抓出來的 bugs, 因此，在某些地方強制規定型別會有助於讓不合型別規定的地方提早噴出 error 出來，讓你比較容易 debug.

Typechecking With PropTypes

```
// 語法
class MyClass extends Component {
  ... this.props.someProp1...
  ... this.props.someProp2...
}

MyClass.propTypes = {  // 注意 'p' 的大小寫
  someProp1: PropTypes.string,
  someProp2: PropTypes.func
};
```

Examples of PropTypes

- 基本的 JS types

```
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol
```

- Anything that can be rendered or an React element

```
  optionalNode: PropTypes.node,
  optionalElement: PropTypes.element
```

Other Examples of PropTypes

```
// an instance of a class
  optionalMessage: PropTypes.instanceOf(Message),
// some value in an enum
  optionalEnum: PropTypes.oneOf(['News', 'Photos']),
// one of the types
  optionalUnion: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number
  ]),
// Add '.isRequired' to make sure a warning
// is shown if the prop isn't provided.
  requiredFunc: PropTypes.func.isRequired,
  requiredAny: PropTypes.any.isRequired,
// '.element.isRequired' 用來指定只能有一個 child node
  children: PropTypes.element.isRequired
```

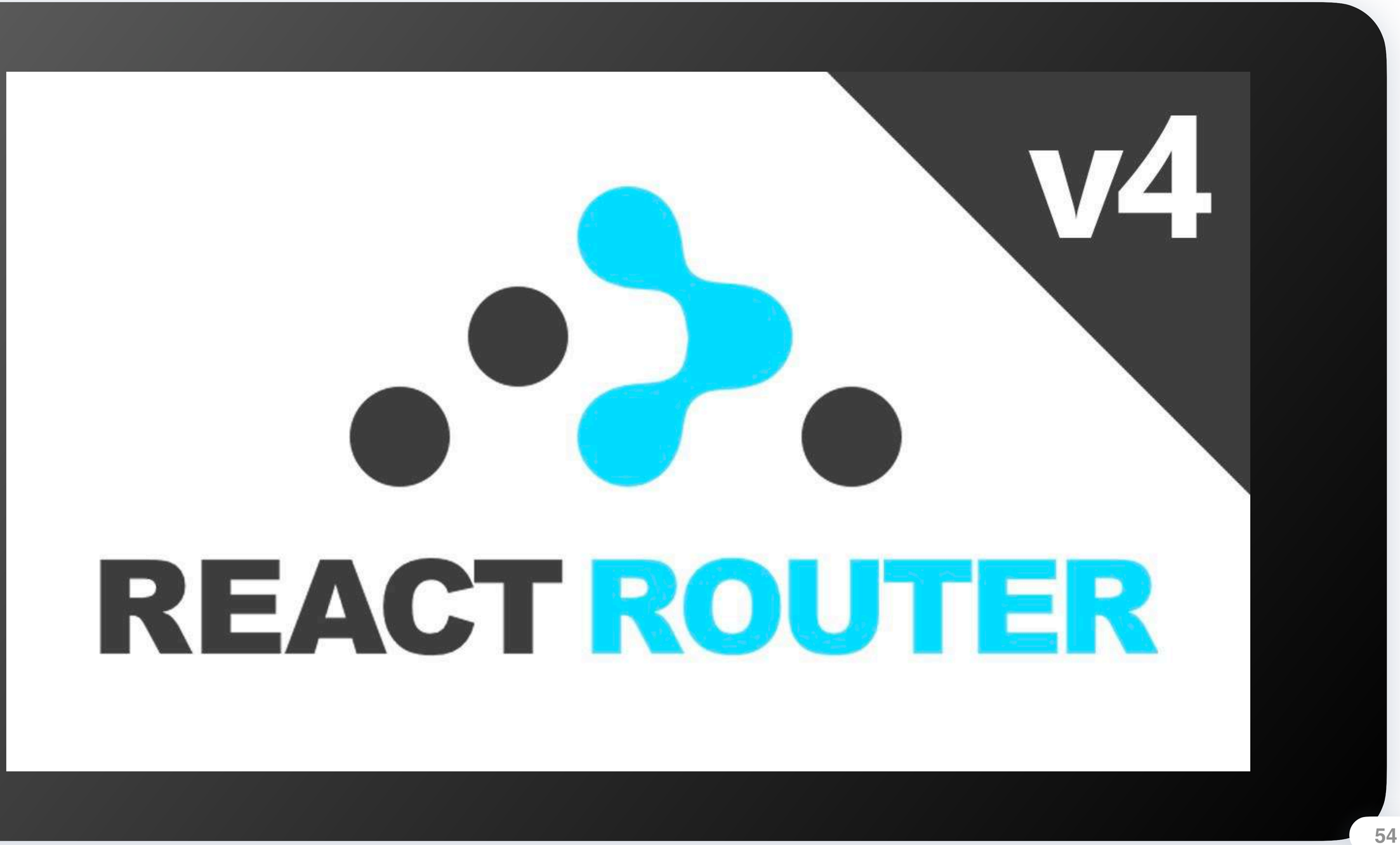
Default Prop Values

- 用 “defaultProps” 來指定 default property value

```
class Greeting extends React.Component {
  render() {
    return (<h1>Hello, {this.props.name}</h1>);
  }
}

// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
};

// Renders "Hello, Stranger":
ReactDOM.render(
  <Greeting />,
  document.getElementById('example')
);
```

事實上，近年來流行的 "TypeScript"
就是把強型別引進 JavaScript.

許多公司都開始採用，他可以解決
不少因為型別轉會換所造成的問題
(We will cover it later if we have time)


v4
REACT ROUTER

Motivations and Backgrounds

- Server-Side (後端) vs Client-Side (前端) Rendering
- Recall: 我們這邊講的「前端」是指你所使用的瀏覽器，負責收到 HTML & data 以後顯示出網頁，而「後端」是指 Web Server, 在收到 http request 之後一方面視需求到資料庫存取、修改資料，然後把 HTML & data 回傳給前端顯示

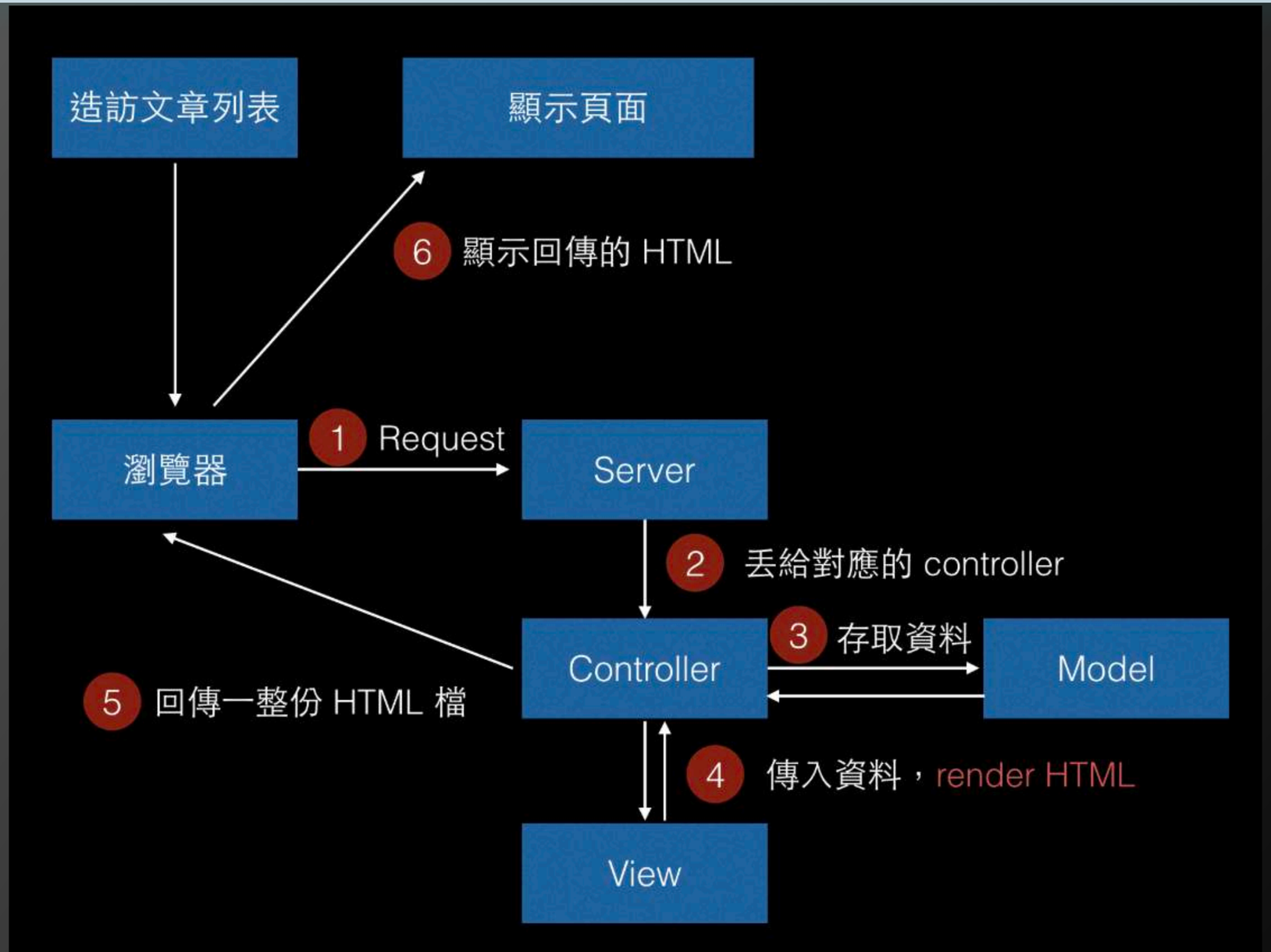
Server-Side（後端）vs Client-Side（前端）Rendering

- "Render" 一般翻譯成 "渲染" 在網頁上就是把頁面畫出來的意思。當使用者點選連結、切換頁面的時候，到底是後端把整 HTML 處理好之後，再傳給前端畫出來 (i.e. server-side rendering)，還是後端只把必要資料處理好之後傳給前端，再由前端處理產生 HTML 再畫出來 (i.e. client-side rendering) 呢？

Server-Side（後端）Rendering

- 如果使用者開啟一個新的網址，server-side rendering 會讓前端拿到一個新的 HTML, 他會看到畫面刷新，如果網路 lag 或是網頁寫得不夠好的話，甚至會看到「白畫面」
  - 想想如果是在聽音樂、玩遊戲，這樣的體驗當然很不 OK!

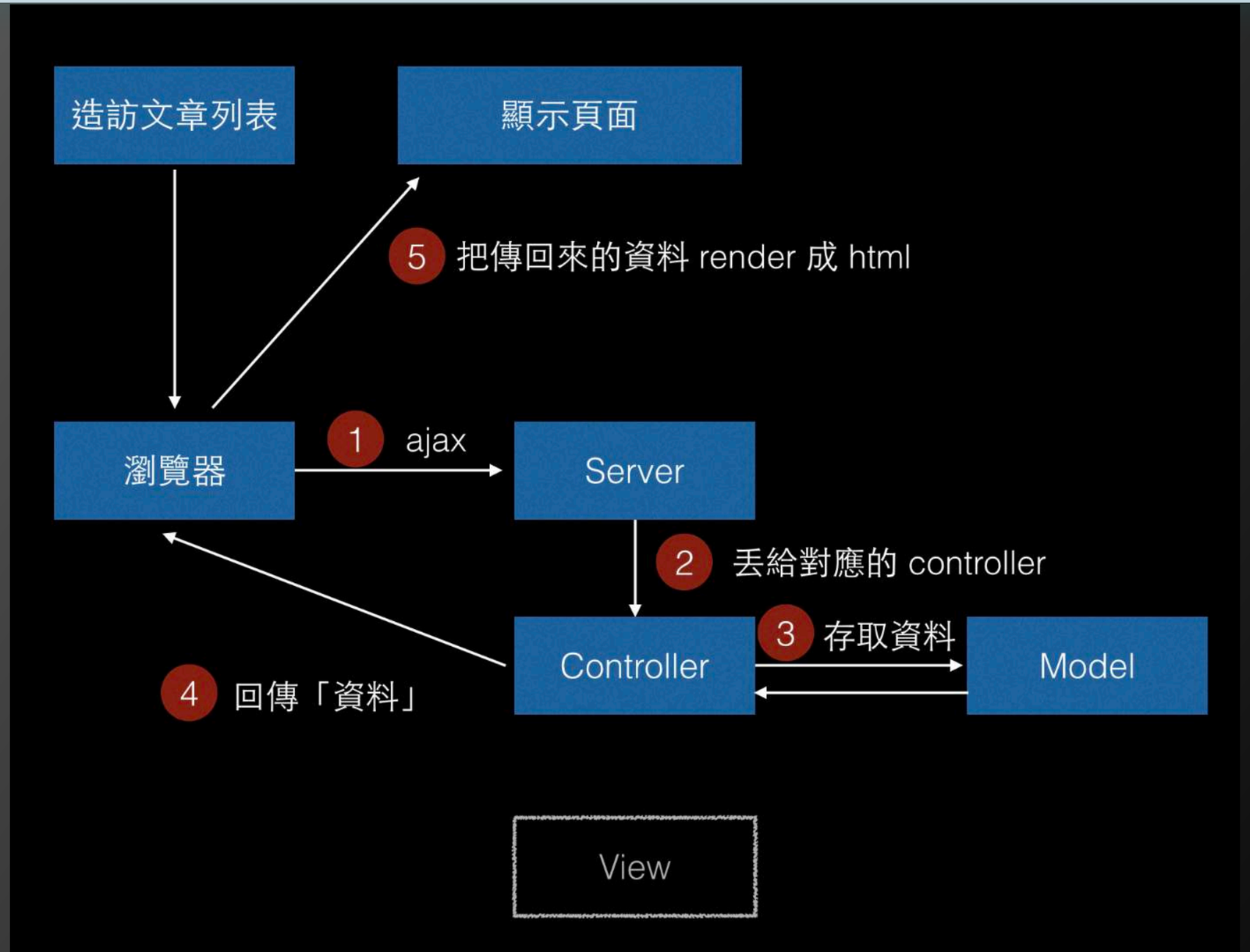
Server-Side Rendering (ref)



Client-Side（前端）Rendering

- Clinet-side rendering 就是利用一些像是 VDOM 的技術，讓使用者點選一些連結的時候，前端網頁只是透過 API 向後端要資料，而前端拿回資料後再更新 DOM 需要更新的 HTML, 動態的更新那部份的頁面。
- 這樣的做法通常會讓前端的 code 變得複雜許多，但還好現在許多前端技術 (e.g. React Routing, GraphQL) 讓這一切寫起來比較乾淨、也比較模組化

Client-Side Rendering (ref)



SPA (Single Page Application)

- 不過前述的 client-side rendering 常常配合著所謂的 SPA (Single Page Application) 的實現方法，也就是說，前端事實上只有一個 index.html, 所以使用者在切換連結的時候只是發出 Ajax/JSON API request, 從後端拿資料，然後前端並沒有切換頁面，所以可以做到像是使用者一邊在網頁上看影片，一方面點選頁面上的連結去查看作者、影片相關資訊，而不會影響到影片的播放。

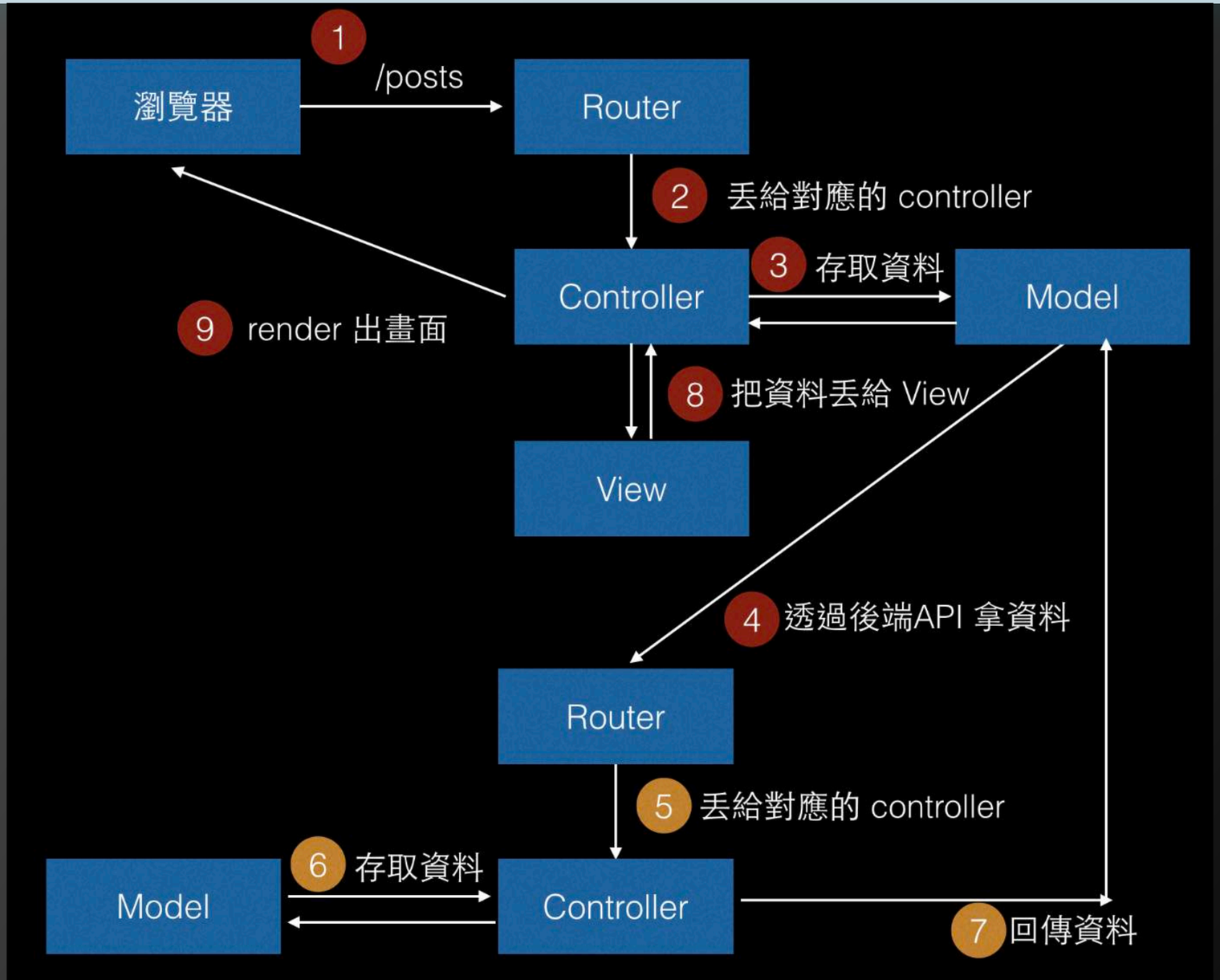
Client-Side Routing

- 不過再想像一個情況，假設你在瀏覽一個部落格或是論壇，從一篇文章切換到另外一篇文章，由於 client-side rendering 的關係，所以頁面上只有文章更新的部分被 update, 所以看起來很順。
- 但問題是，當你很直覺的按瀏覽器的「上一頁」，想要回到上一篇文章的時候，你會發現沒有用！因為你從頭到尾都是在 "index.html" 這頁上面啊！

Client–Side Routing

- Client-Side Routing 讓你在 local 端產生瀏覽器的 routing, 像是：
  - ...myblog.com/home
  - ...myblog.com/posts
  - ...myblog.com/posts/13
  - ...myblog.com/users/ric
- 在瀏覽到不同頁面的時候會有對應到不同的 routing (web address)，而被存到瀏覽器的 history 中，可以使用前/後一頁

Client-Side Rendering/Routing (ref)



React Router

- React Router 是整個「React 生態系」的一部分，通過管理 URL，實現頁面以及狀態切換可以「模組化」，讓 code 更好管理，也比較容易理解，也能符合 React 基本只更新 minimum difference 的概念

安裝與使用 React Router

- 安裝：npm install react-router-dom
  - 如果遇到建議要 "npm audit fix"，就 fix 吧！
- 使用：

```
import { BrowserRouter } from 'react-router-dom'
import { NavLink, Switch, Route } from 'react-router-dom'
```

一個簡單的應用情境

- 假設你寫了一個 blog page, 你規劃了：
  - '/' or '/home'：主畫面
  - '/posts'：顯示所有文章列表
  - '/posts/<postId>'：顯示某篇文章
  - '/authors'：顯示所有作者列表
  - '/authors/<authorName>'：顯示某位作者的文章列表

Top-level "App.js"

- 定義了這個 APP 的 root directory (i.e. '/')

```
class App extends Component {
  render() {
    return (
      <BrowserRouter>
        <div className="App">
          <Blog / >
        </div>
      </BrowserRouter>
    )
  }
}
```

In "Blog.js"

```
class Blog extends Component {
  render() {
    return (
      <div>  // Define your blog layout
        ...
        <NavLink to="/home">Home</NavLink>
        <NavLink to="/posts">Posts</NavLink>
        <NavLink to="/authors">Authors</NavLink>
        ...
        <Switch>
          <Route exact path="/posts" component={Posts} / >
          <Route path="/posts/:id?" component={PostRoute} / >
          <Route exact path="/authors" components={Authors} / >
          <Route path="/authors/:name?" components={AuthorRoute} / >
          <Redirect from="/Home" to="/" / >
        </Switch>
      </div>
    )
  }
}
```

<NavLink to="/home">Home</NavLink>

- 定義 "Home" 這個字所對應的 routing path
- 其中，'/' 代表這個 App 的根目錄
- <NavLink> 只是用來代表一個 link 而已，真正在頁面上畫出來，還是要在外面包一個 HTML tag, 例如：

```
<p><NavLink to="/home">Home</NavLink></p>
<li><NavLink to="/home">Home</NavLink></li>
<button><NavLink to="/home">Home</NavLink></button>
```

- 換句話說，在畫面點下 "Home" 的時候，頁面會 route 到 "...AppHome/home"

<NavLink> vs. <Link>?

- 有時候你會在別的範例看到別人使用 <Link>, 而非 <NavLink>, what's the difference?
- 官方說明：A special version of the <Link> that will add styling attributes to the rendered element when it matches the current URL.

<Switch>...</Switch>

- 用來定義這個 App 的所有 routings 如何產生畫面
- 一個 <Switch> 裡面包著多個 <Route / >，而每個 <Route / > 用來指定在 <NavLink> 所定義的 path 連結，要用哪一個 React component 來產生畫面呢？
- 基本 <Route> 的語法

```
<Route path="/someDir" component={SomeComponent} / >
```

Putting things together...

```
class Blog extends Component {
  render() {
    return (
      <div>  // Define your blog layout
        <ul>
          <li><NavLink to="/posts">Posts</NavLink></li>
          <li><NavLink to="/authors">Authors</NavLink></li>
        </ul>
        <Switch>
          <Route path="/posts" component={Posts} / >
          <Route path="/authors" components={Authors} / >
        </Switch>
      </div>
    )
  }
}
```

URL Parameters

- 通常會把文章根據 IDs, 或者是作者根據名字，來安排至不同的 routings, 例如：
  - .../posts/12345678
  - .../authors/ric
- 但隨著 Blog 的文章會增加、讀者/作者數量也會增加，不可能在 Blog.js 裡頭把這些文章、作者頁面的 routings 全部預先寫死。因此，我們要用 "參數" 來指定 routing 的規則。例如：

```
<Route path="/posts/:id?" component={PostRender} / >
```

```
<Route path="/posts/:id?" component={PostRender} />
```

- 當你定義這行時，你事實上就定義了指定的 component (i.e. PostRender) 的 props.match.params 多了一個 "id" 這個 property
- 換句話說，當你連結 ".../posts/3838" 的時候，就等於把 3838 當作參數傳給 PostRender 的 props.match.params.id, 你可以在 PostRender 裡頭根據 id 去處理拿到文章的邏輯。

"exact path" for <Route>

- 不過當這兩行同時存在的時候...

```
// 列舉所有文章
<Route path="/posts" component={Posts} />
// 展示某篇文章
<Route path="/posts/:id?" component={PostRender} />
```

- 當網址是 ".../posts/3838" 的時候，事實上兩條 routing rules 都會符合，所以照順序，會吐出第一個 match (列舉所有文章)，而不是展示 3838 這篇文章
- 所以，第一條應該要改成 exact path:

```
// 列舉所有文章
<Route exact path="/posts" component={Posts} />
```

URL Redirect

- 用途：將某個 path ".../pathA" redirect 到另一個 path ".../pathB"

```
<Redirect from="pathA" to="pathB" />
```

Let's look at a simple yet complete example!

- Create a react project "react-router-test1"
- Download "react-router-boilerplate-src.tgz" from Ceiba!
- Install and Run!

```
cd react-router-test1
yarn add react-router-dom
rm -rf src
tar zxvf react-router-boilerplate.tgz
yarn start
```

Code tree (under "src")

- index.js // to mount "root" DOM node
- App.js // Define Routing root '/'
- containers/
  - Blog/
    - Blog.js // Define main page and routing rules
    - Posts/
      - Posts.js // List all posts (Posts module)
      - PostRender.js // Define how to generate posts
- components/
  - Post
    - Post.js // Define Post module

Posts.js (列舉文章列表)

- Note: 理論上文章等資料都應該是從後台 (backend server) 過來，但我們現在還沒有(教)後台，所以我們會把資料寫死在 JS 檔案裡

```
export default class Posts extends Component {
  render() {
    const postIDs = ["1", "3", "5", "7", "9"];
    const lists = postIDs.map((i, index) => (
      <li key={index}>
        <NavLink to={"/posts/" + i}>Posts #{i}</NavLink>
      </li>));
    return (
      <div>
        <h3>Click to view article ---</h3> {lists}
      </div>);
  }
}
```

<li key={index}>

- 如果把 "key={index}" 拿掉，你會看到這樣的 message: Warning: Each child in a list should have a unique "key" prop.
  - 原則上如果文章在後台管理，你可以 assign 每一篇文章一個 unique ID, 然後就可以利用這個 ID 來當 <li> 的 unique keys
  - 所以我們在這邊先使用 Array 的 index 來當 key

<NavLink to={"/posts/" + i}>Posts #{i}</NavLink>

- 用來指定當點選某篇文章連結的時候，會 route 到當篇文章的網址 (defined in “Blog.js")

PostRender.js (定義如何產生 posts)

```
export default class PostRender extends Component {
    render() {
        const postIDs = ["1", "3", "5", "7", "9"];
        const { id } = this.props.match.params;
        return id && postIDs.includes(id) ? (
            <Post id={id} />
        ) : (
            <div>
                <h3>Error: Post #{id} NOT FOUND</h3>
            </div>
        );
    }
}
```

“const { id } = this.props.match.params”?

- This is "Destructuring Assignment", which is equivalent to:
  - const id = this.props.match.params.id;
- You can also do something like:
  - const { a: b } = obj.someProp,
- which is equivalent to:
  - const b = obj.someProp.a;

- That's it!
- You are recommended to go through this official React Router Tutorial

感謝聆聽！

Ric Huang / NTUEE

(EE 3035) Web Programming