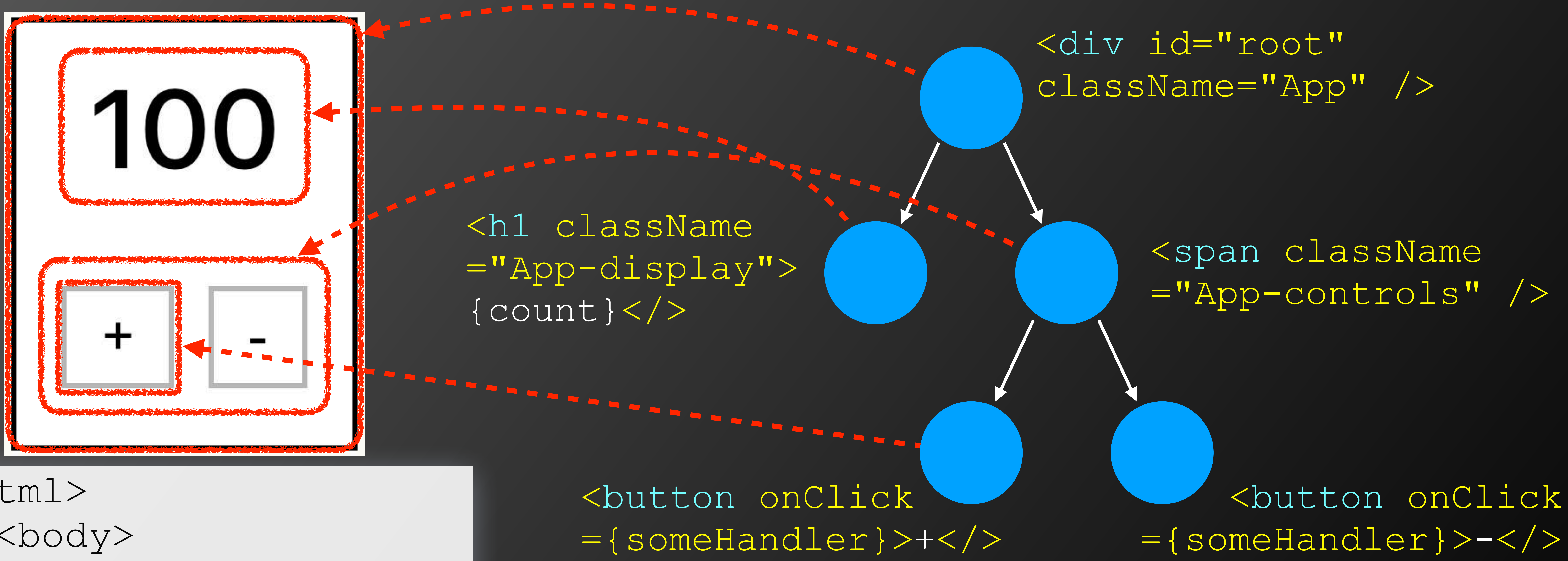
05. More on React.js

Ric Huang / NTUEE

(EE 3035) Web Programming

Quick Review on the “Counter” Example

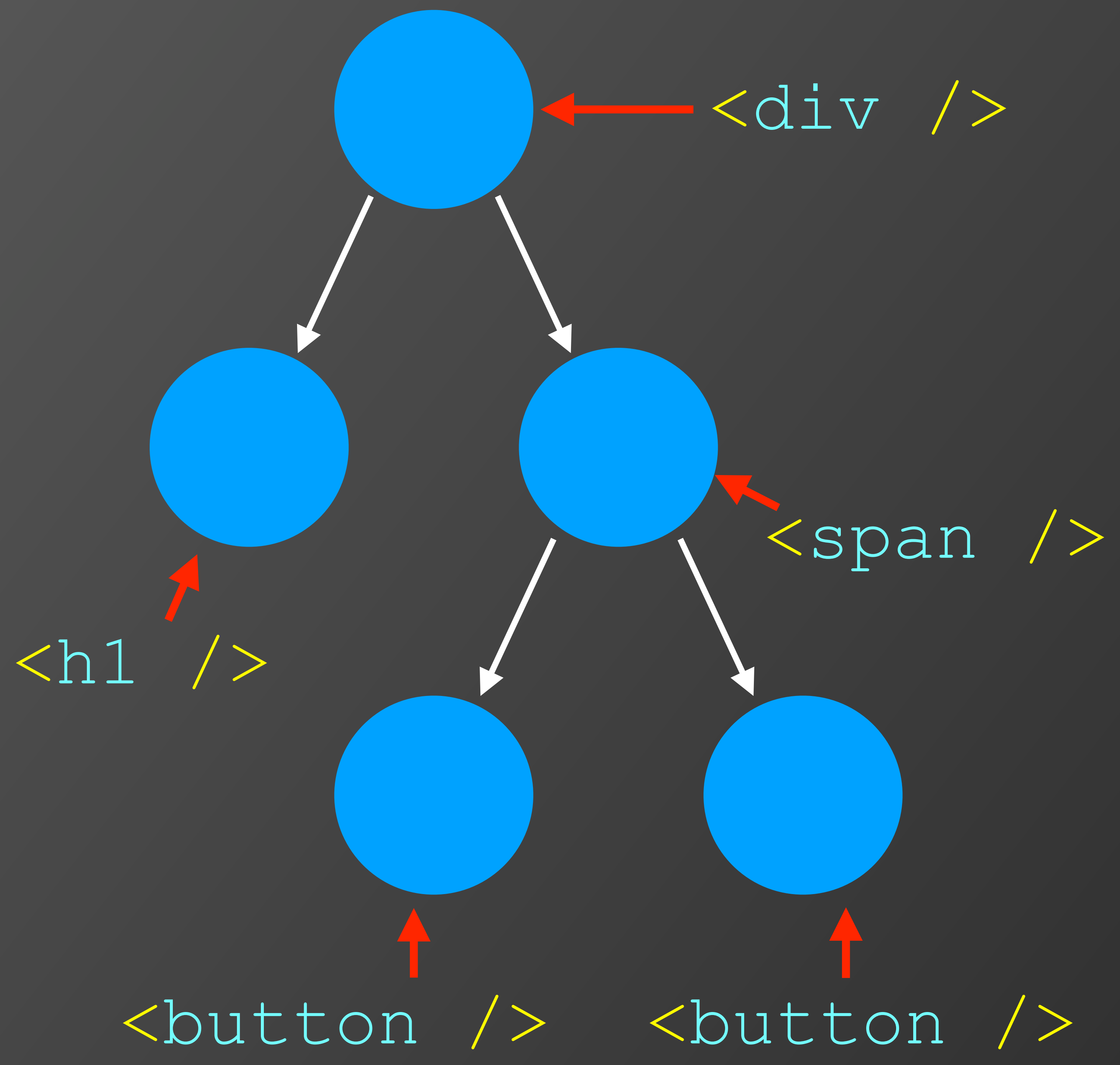
1. 先寫好 index.html, 並且規劃好 DOM structure



```
<html>
  <body>
    <div id="root" />
  </body>
<html>
```

Quick Review on the “Counter” Example

2. 寫好 src/App.js, 定義一個 top level class, 並且用它的 render() 來產生 DOM



```
import React, {Component} from 'react';
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display">100</h1>
        <span className="App-controls">
          <button>+</button>
          <button>-</button>
        </span>
      </div>
    );
  }
}
export default Counter;
```

JavaScript / React class

HTML/CSS class

先寫成靜態的值，
有畫面再說

Quick Review on the “Counter” Example

3. 用一個 src/index.js 來把 class Counter 與 HTML 串起來

先 "yarn start"，看看是否有正常看到畫面

public/index.html

```
<html>
  <body>
    <div id="root" />
  </body>
<html>
```

src/index.js

```
import Counter from './App';
ReactDOM.render(
 <Counter />,
  document.getElementById('root')
);
```

src/App.js

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        ...
      </div>
    );
  }
}
```

Quick Review on the “Counter” Example

4. 由於 {counter} 的值會 depends on 動作之前一刻的值 (i.e. stateful), 所以它應該要是 class Counter 的一個 state value

```
class Counter extends Component {
  constructor(props) {
    super(props);
    this.state = { count: 100 };
  }
  render() {
    return (
      <div className="App">
        <h1 className=“App-display">{this.state.count}</h1>
        …
      </div>
    );
  }
}
```

this.state 是一個物件，所以用 { } 來初始化其值

<div…> 進入 JSX 的語法範圍

- HTML tag 的內文
- 由於內文是從 "變數" 來動態決定，所以用 { } 來表示 JS 的 expression

Quick Review on the “Counter” Example

5. 定義 handler functions 來處理 button onClick 的事件

先來個錯誤示範…

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick={()=>this.state++}>+</button>
          <button onClick={()=>this.state--}>-</button>
        </span>
      </div>
    );
  }
}
```

Recall: state 的更新要用 setState(), 不能直接設定！！

Quick Review on the “Counter” Example

再來個錯誤示範...

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick=
              {()=>this.setState(this.state + 1)}>+</button>
          <button onClick=
              {()=>this.setState(this.state - 1)}>-</button>
        </span>
      </div>
    );
  }
}
```

Recall: state 是個 object！！

Quick Review on the “Counter” Example

錯誤示範 #3…

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick={()=>this.setState
                 ({count: this.state.count + 1})}>+</button>
          <button onClick={()=>this.setState
                 ({count: this.state.count - 1})}>-</button>
        </span>
      </div>
    );
  }
}
```

咦！可以 work 啊！錯在哪裡？？

Quick Review on the “Counter” Example

根據 React 的官方說法，State Updates May Be Asynchronous

```
// Wrong: state 的 value 可能沒有被 update 到
this.setState({
  counter: this.state.counter + this.props.increment
});
```

```
// Correct: 這樣才會拿 previous state 的值來 update
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

Quick Review on the “Counter” Example

Try this…

```
class Counter extends Component {
  handlePlus2 = () => {
    this.setState({count: this.state.count + 1});
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className=“App-controls">
          <button onClick={this.handlePlus2}>+2</button>
          <button onClick={()=>this.setState
                 ({count: this.state.count + 1})}>+</button>
          <button onClick={()=>this.setState
                 ({count: this.state.count - 1})}>-</button>
        </span>
      </div>
    );
  }
}
```

State Updates May Be Asynchronous

Note: JS 的 statements 是 non-blocking 的依序執行

```
handlePlus2 = () => {
    this.setState({count: this.state.count + 1});
    this.setState({count: this.state.count + 1});
    console.log("在這邊 console.log 試試看");
}
```

先 evaluate,
兩者都會變成
{count: 101}

而所謂的 asynchronous 的執行，就是把 async functions (通常是 batch 的方式)丟出去給這個執行 async functions 的 engine, 執行完畢之後，再用 callback 通知主程式

```
// 所以 React engine 收到的就是 —
setState({count: 101});
setState({count: 101});
```

Quick Review on the “Counter” Example

5. 定義 handler functions 來處理 button onClick 的事件

```
class Counter extends Component {
  handleInc = () => this.setState
                    (state => ({ count: state.count + 1 }));
  handleDec = () => this.setState
                    (state => ({ count: state.count - 1 }));
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick={this.handleInc}>
              +</button>
          <button onClick={this.handleDec}>
              -</button>
        </span>
      </div>
    );
  }
}
```

Closer look on the onClick's handler

```
<button onClick={this.handleInc}>
```

- 因為要綁定一個 function, 所以用 { } 進入 JS 的 expression, 且不能寫成 this.handleInc(), 否則就變成先呼叫 function 後的 return 值了

```
handleInc = () => this.setState
                  (state => ({ count: state.count + 1 }));
```

- 用 arrow function, 因為要 return 一個 function 給 handleInc

一定要用 arrow function 嗎？

Try this...

- 這樣寫會給 “Failed to compile” (Why?)

```
function handleInc() {
     this.setState
          (state => ({ count: state.count + 1 })); }
```

- 這樣寫 compile 會過，但按了 ‘+’ 號後會有 “TypeError: Cannot read property 'setState' of undefined” 的 error (Why?)

```
handleInc
= function() {
     this.setState
          (state => ({ count: state.count + 1 })); }
```

Recall: ‘this’ refers to the function scope

this and bind()...

- 如果堅持要用前頁 function 的寫法，一個解決的辦法是在 caller 把 this bind() 起來！

```
<button onClick={this.handleInc.bind(this)}>
```

- 實在是 awkward⋯ 好險現在有 arrow function
=> arrow function 裡頭的 this refers to the caller's scope

Closer look on the onClick's handler

```
handleInc = () => this.setState
            (state => ({ count: state.count + 1 }));
```

為什麼不用 'this'?

- this.setState() 吃的參數是一個 "stateUpdateFunction" (所以要用 arrow function), 而這個 function 吃一個參數 (i.e. local variable)，setState 會把 current this.state assign 給它
- 所以，也可以寫成：

```
handleInc = () => this.setState
            (s => ({ count: s.count + 1 }));
```

但，"count" 不能改成別的名字！ (why?)

And remember...

- “props” is pure. It should be read-only.
  - It's value is assigned when passed through tag attribute and should remain unchanged afterwards.
- “state” is private. You should use “this.setState()” to update state's value.
  - Otherwise, it won't trigger updates on VDOM.

Quick Review on the “Counter” Example

6. 用 functional component 把 logic 跟 component 分開

```
// in "App.js"
import React, { Component } from 'react'
import Button from '../components/Button'
...
    <Button text="+" onClick={this.handleInc} />
    <Button text="-" onClick={this.handleInc} />
```

```
// 加一個 “src/components/Button.js”
import React from 'react'
export default ({ onClick, text }) => {
  return <button onClick={onClick}>{text}
         </button>;
}
```

Closer look at the functional component...

- Recall: 當上層的邏輯 (e.g. containers/App.js) 呼叫下層的 components 時，是用 JSX 與 tag attributes 打包成 object 傳給 component 的 props.

```
class Welcome extends Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }                                {name: "Ric"}
}
const element = <Welcome name="Ric" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Closer look at the functional component...

- 如果改寫成 function...

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
const element = <Welcome name="Ric" />;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

- 事實上，props 只是 local variable (function argument), 你要把它改成 'p' 也是可以的 (但寫成 props 才會符合一些 linter 的規則)

Closer look at the functional component...

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

- 改寫成 functional component…

```
// components/Welcome.js
export default
(props) => return <h1>Hello, {props.name}</h1>;
```

- 應用 destructuring assignment 的概念，簡化成：

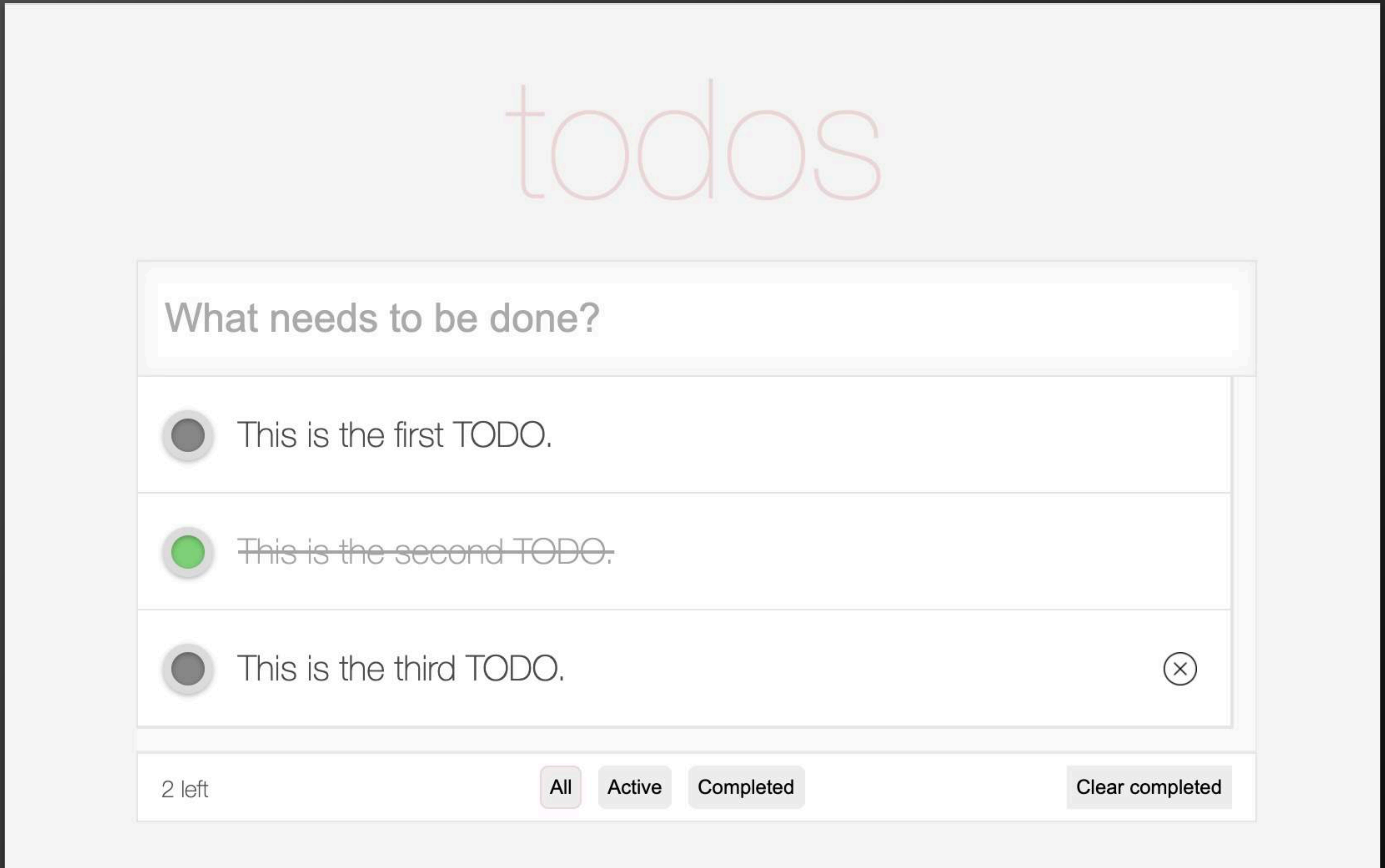
```
// components/Welcome.js
export default
({name}) => return <h1>Hello, {name}</h1>;
```

Thinking in React (ref)

1. Break The UI Into A Component Hierarchy
2. Build A Static Version in React
3. Identify The Minimal (but complete) Representation Of UI State
4. Identify Where Your State Should Live
5. Add Inverse Data Flow

How to apply it to HW#3?

0 Start With A Mock

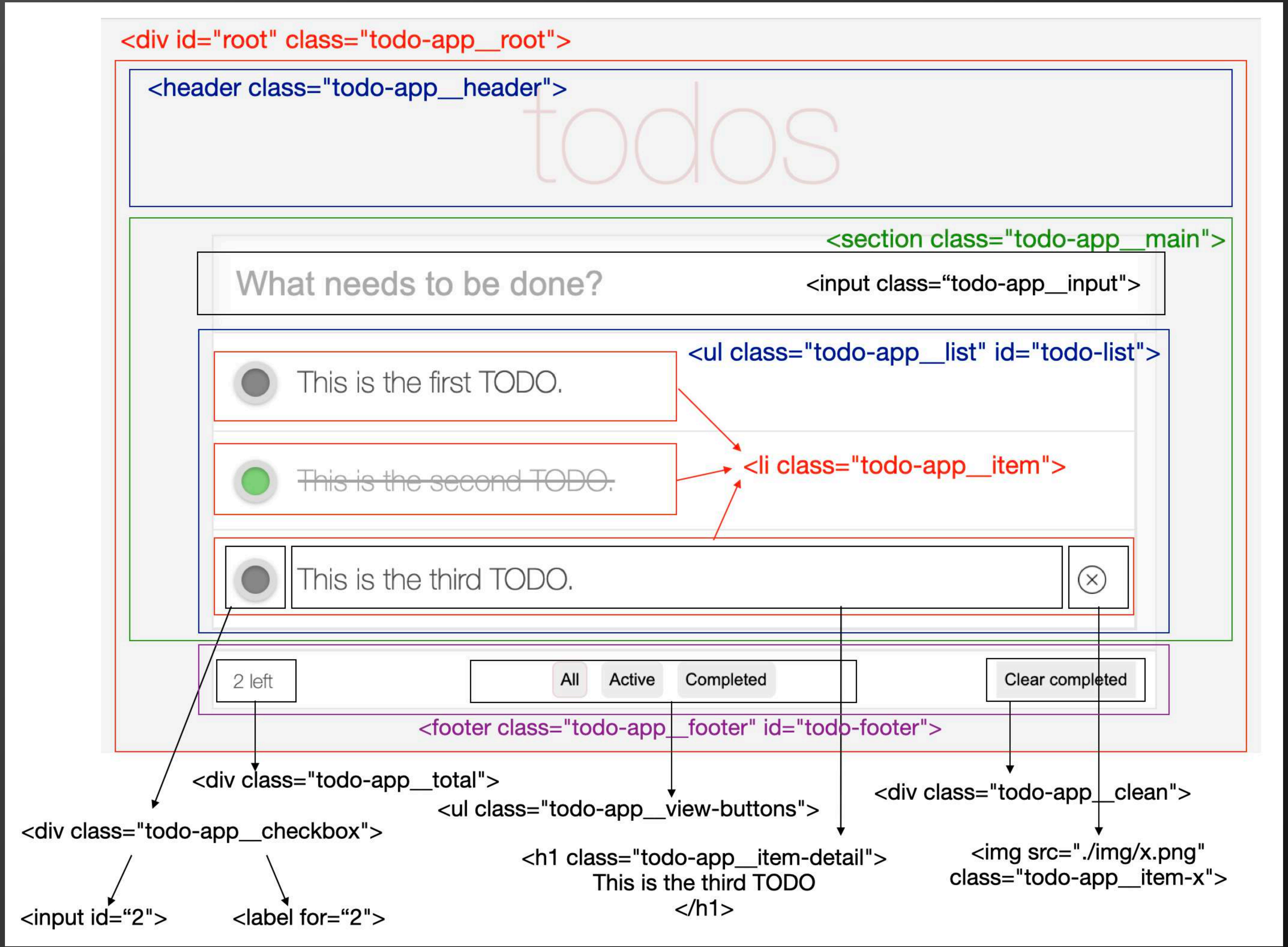


What needs to be done?

- This is the first TODO.
- ~~This is the second TODO.~~
- This is the third TODO.

How to apply it to HW#3?

1 Break The UI Into A Component Hierarchy

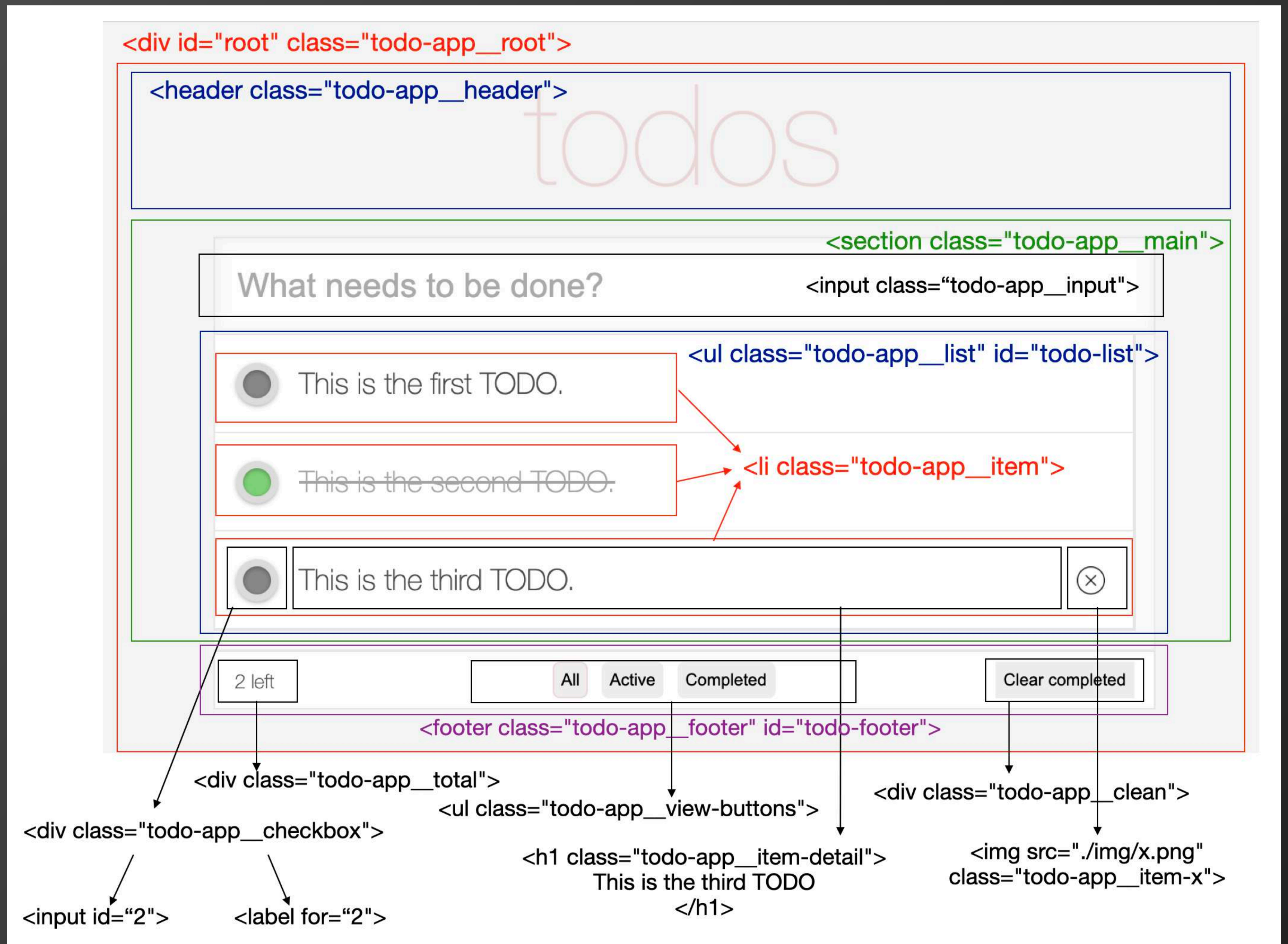


<div id="root" class="todo-app__root">
<header class="todo-app__header">
todos
<section class="todo-app__main">
What needs to be done?
<input class="todo-app__input">
<ul class="todo-app__list" id="todo-list">
This is the first TODO.
~~This is the second TODO.~~
<li class="todo-app__item">
This is the third TODO.
2 left
All
Active
Completed
Clear completed
<footer class="todo-app__footer" id="todo-footer">
<div class="todo-app__total">
<ul class="todo-app__view-buttons">
<div class="todo-app__clean">
<div class="todo-app__checkbox">
<input id="2">
<label for="2">
<h1 class="todo-app__item-detail"> This is the third TODO </h1>
<img src="./img/x.png" class="todo-app__item-x">

Thinking in React (ref)

2 Build A Static Version in React

3 Identify The Minimal (but complete) Representation Of UI State



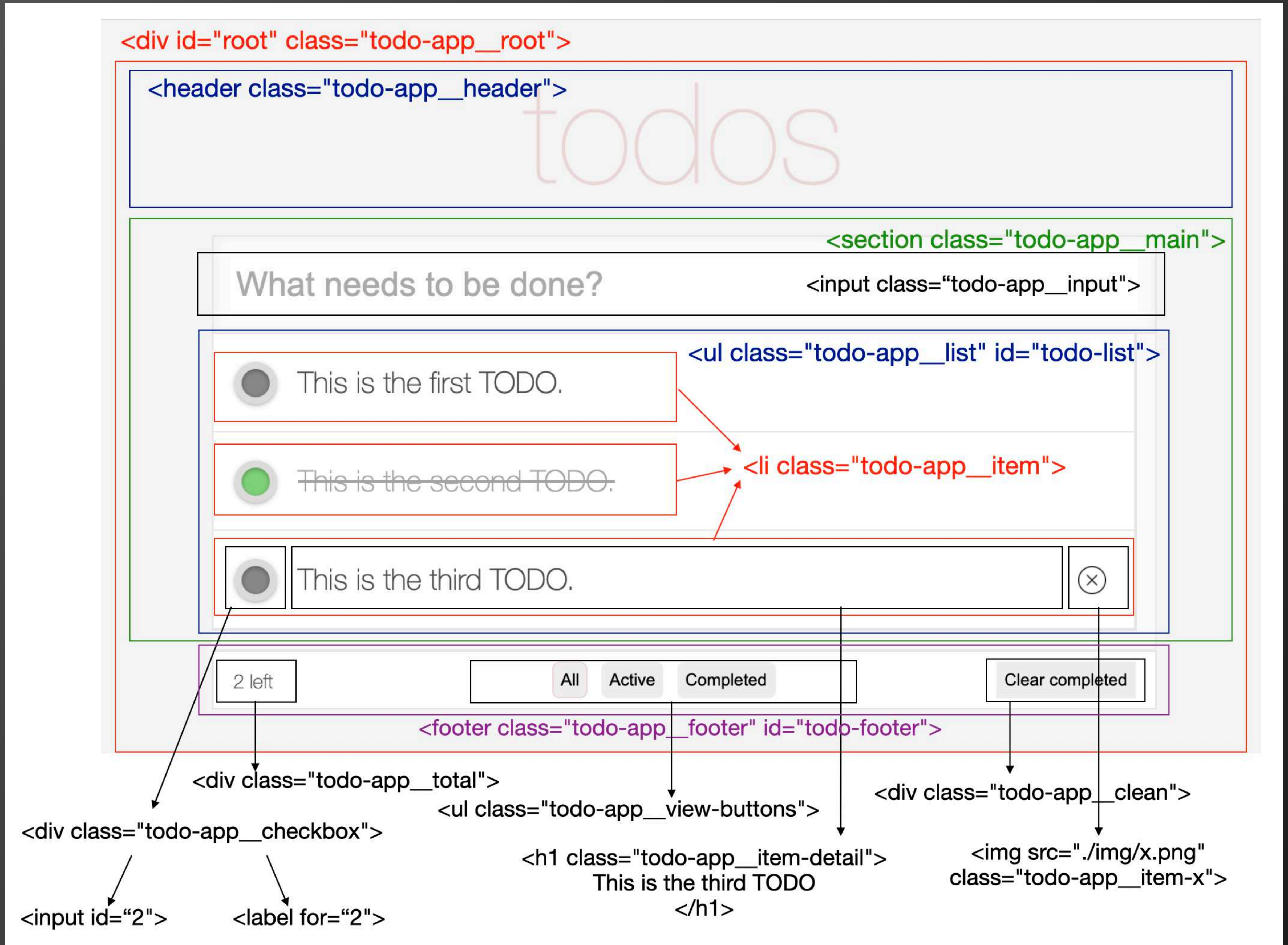
<div id="root" class="todo-app__root">
<header class="todo-app__header">
todos
<section class="todo-app__main">
What needs to be done?
<input class="todo-app__input">
<ul class="todo-app__list" id="todo-list">
This is the first TODO.
~~This is the second TODO.~~
<li class="todo-app__item">
This is the third TODO.
2 left
All
Active
Completed
Clear completed
<footer class="todo-app__footer" id="todo-footer">
<div class="todo-app__total">
<ul class="todo-app__view-buttons">
<div class="todo-app__clean">
<div class="todo-app__checkbox">
<h1 class="todo-app__item-detail"> This is the third TODO </h1>
<img src="./img/x.png" class="todo-app__item-x">
<input id="2">
<label for="2">

What data should be carried over time?

Be aware of keeping the "single source of truth"

Thinking in React (ref)

4 Identify Where Your State Should Live



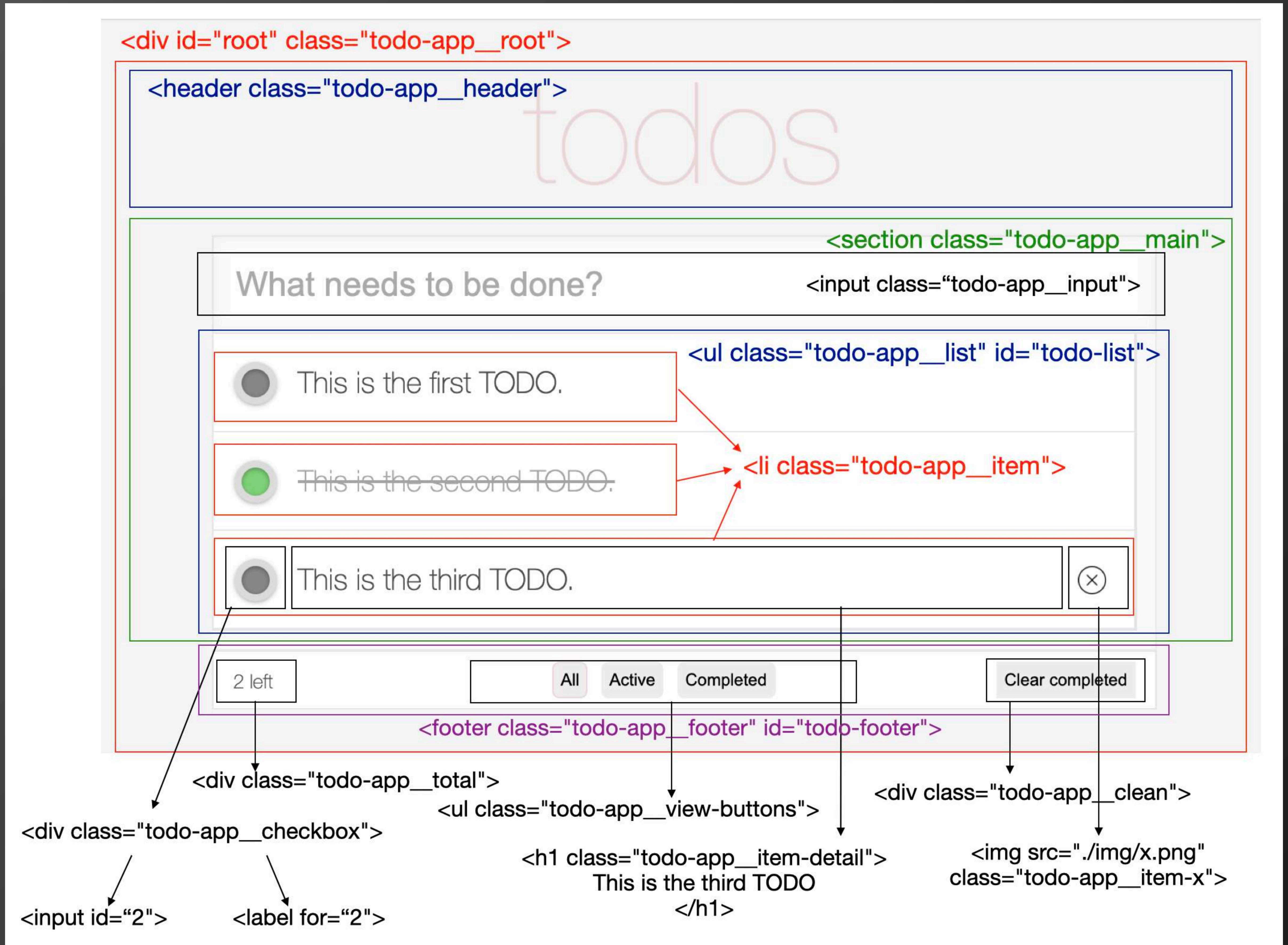
<div id="root" class="todo-app__root">
<header class="todo-app__header">
todos
<section class="todo-app__main">
What needs to be done?
<input class="todo-app__input">
<ul class="todo-app__list" id="todo-list">
This is the first TODO.
~~This is the second TODO.~~
<li class="todo-app__item">
This is the third TODO.
2 left
All
Active
Completed
Clear completed
<footer class="todo-app__footer" id="todo-footer">
<div class="todo-app__total">
<ul class="todo-app__view-buttons">
<div class="todo-app__clean">
<div class="todo-app__checkbox">
<h1 class="todo-app__item-detail"> This is the third TODO </h1>
<img src="./img/x.png" class="todo-app__item-x">
<input id="2">
<label for="2">

Bring the state up!

Thinking in React (ref)

5

Add Inverse Data Flow



<div id="root" class="todo-app__root">
<header class="todo-app__header">
todos
<section class="todo-app__main">
What needs to be done?
<input class="todo-app__input">
<ul class="todo-app__list" id="todo-list">
This is the first TODO.
<li class="todo-app__item">
~~This is the second TODO.~~
This is the third TODO.
2 left
All
Active
Completed
Clear completed
<footer class="todo-app__footer" id="todo-footer">
<div class="todo-app__total">
<div class="todo-app__clean">
<ul class="todo-app__view-buttons">
<div class="todo-app__checkbox">
<input id="2">
<label for="2">
<h1 class="todo-app__item-detail"> This is the third TODO </h1>
<img src="./img/x.png" class="todo-app__item-x">

Passing data to props

希望你到目前為止都聽得懂，
可以掌握 React 的基本語法與精神。
接下來我們將介紹一系列的進階語法，
讓你通透 React 最新的發展現況！

Advanced React Topics

- Compositional Model
- React.Fragment
- Higher Order Component (HOC)
- React Hooks

React Compositional Model

- React Composition is a development pattern based on React's original component model where we build components from other components using explicit defined props or the implicit children prop (ref)
- Component's relationship in React (ref)
  - The parent may or may not know who the children are ahead of time.
  - Children never know who is the parent
  - Children never know who are the siblings
  - The relationship has a well-defined interface, (i.e. props).

React Compositional Model

- 在之前的 React 範例，上層的 JSX 利用 tag attribute 來傳遞參數給下層的 class component (as props)
- 而下層的 class component 其 DOM structure 則是固定寫死在 render() 裏頭。像是 —

```
class Counter extends Component {
  render() {
    return (
      <div className="App">
        <h1 className="App-display”>{this.state.count}</h1>
        <span className="App-controls">
          <button onClick={this.handleInc}>+</button>
          <button onClick={this.handleDec}>-</button>
        </span>
      </div>
    );
  }
}
```

但如果我們想要利用上層的邏輯 動態的決定下層的 DOM structure 呢？

使用“props.children”這個 property

```
function Welcome(props) {
  return (
    <div>
      {props.children}    // props 可以叫別的名字，
    </div>                // 但 children 不行
  );
}
const element =
  <Welcome>
    <h1> Welcome, Ric </h1>
    <p> Thank you for visiting our spacecraft! </p>
  </Welcome>
;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

- 在定義 Welcome 的時候，我們不知道上層會 instantiate 幾個 child components…
- 用<h1>, <p> 產生兩個 components, 傳給 Welcome 的 props.children 這個 array

事實上，可以有各種變化，像是...

```
function Welcome(props) {
  return (
    <div>
      <h1> Welcome, {props.name} </h1>
      {props.children}
    </div>
  );
}
const element =
  <Welcome name="Ric">
   <p> Thank you for visiting our spacecraft! </p>
  </Welcome>
;
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

Composition vs. Inheritance

- 根據 React 官方的說法，當我們需要「特別化」一些 components 時，我們應該用 props 來傳遞訊息，而不是用 class inheritance

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">{props.left}</div>
      <div className="SplitPane-right">{props.right}</div>
    </div>
  );
}
function App() {
  return (
    <SplitPane left={<Contacts />} right={<Chat />} />
  );
}
```

React.Fragment

- Recall: 在 React Component 的 render() 裏頭，你必須 return 一個 single root node. 但當我們需要 return multiple nodes 的時候，一個解法是用 "<div />" 包起來。但是，如果 caller 的 children 不可以是 "<div>" 怎麼辦呢？

Oops, <tr> is expected for <table>

```
class MyTable extends Component {
  render() {
    return
      <table>
        <MyData dataInput={data1}/ >
        <MyData dataInput={data2}/ >
      </table>
  }
}
```

```
class MyData extends Component {
  render() {
    return (
      <div>
        <tr>{some data}</tr>
        <tr>{some data}</tr>
      </div>
    );
  }
}
```

Use React.Fragment to solve it!

```
import React, { Fragment } from 'react';
class MyData extends Component {
  render() {
    return (
      <Fragment>
        <tr>{some data}</tr>
        <tr>{some data}</tr>
      </Fragment>
    );
  }
}
```

It can also be written as...

```
import React, { Fragment } from 'react';
class MyData extends Component {
  render() {
    return (
      <>
        <tr>{some data}</tr>
        <tr>{some data}</tr>
      </>
    );
  }
}
```

- Note: "key" is the only attribute that can be used in <Fragment>. Event handlers are not supported yet.

Note: HW#3 的 containers/TodoApp.js 就有用到了

```
import React, { Component } from "react";
import Header from "../components/Header";

class TodoApp extends Component {
  render() {
    return (
      <>
        <Header text="todos" />
      </>
    );
  }
}

export default TodoApp;
```

Higher Order Component (HOC)

- "Higher Order Component" 是另外一個 React 裡頭常用的 programming technique --- 想像你的 navigation bar 會隨著 logged in user 不同而有不同的內容/layout，或者是你的 blog page 會隨著文章種類的不同而選擇不同的來源... 等，但他們的 event binding, error handling, 或者是其他的邏輯是一樣的，所以你會想要有一個 "產生 component 的 function", 可以吃進一個 component 當參數，然後也許吃進另一個 callback 當作客製化 layout/data source 的方法，像這樣 (next page):

Higher Order Component (HOC)

```
const generalNavBar = (WrappedNavBar, layoutMethod) =>
  return class extends Component {
    constructor(props) {...}
    ... some life-cycle methods or event handling logic
    render() {
      return <WrappedNavBar ... / >;
    }
  }
```

- 基本上, HOC 就是用一個 function, 吃進一個 wrapped component, 產生另一個 higher-order component
- 參數列不限個數，但第一個 arg 通常是 wrapped component
- "return class extends" <== anonymous class

Another HOC Example

```
const withAuthGuard = WrappedComponent => {
  return class extends Component {
    render() {
      return (
        <Query query={ME_QUERY}
               onError={error => {...; }}}> {
          ({ data, loading, error }) => {
            if (loading)
              return <WrappedComponent loading={true} />
            if (error)
              return <WrappedComponent isAuth={false} />
            return (
              <WrappedComponent isAuth={true}
                  username={data.me.username} />)}
        } </Query>)}
  }
}
```

HOC should be pure!

- Note that a HOC doesn't modify the input component, nor does it use inheritance to copy its behavior. Rather, a HOC composes the original component by wrapping it in a container component. A HOC is a pure function with zero side-effects.
- However, don't apply HOC in render(). This will cause the subtree to unmount/remount each time! (ref)

React
Hooks

React Hooks。Motivation

- 你有沒有覺得 React state 有點迷樣...

1. 在 Component 之間重用 Stateful 的邏輯很困難
   - 常常會有不同的 components 的 stateful 邏輯很像或是一樣，但你可以做的卻只是用 wrapper 或是 HOC 把它塞到不同的 components 裡頭去，變得很難 debug
2. 複雜的 component 變得很難理解
   - 常常必須在 life cycle 的不同階段去把一些不相關的 states 的邏輯放在一起，而無法針對某個 state 把它完整的邏輯抽象化出來
3. Class 讓人們和電腦同時感到困惑
   - 你必須了解 this 在 JavaScript 中如何運作，而這跟它在大部分程式語言中的運作方式非常不同

React Hooks (ref)

Hook 是 React 16.8 中增加的新功能。它讓你不必寫 class 就能使用 state 以及其他 React 的功能。

Using React Hook for the "Counter" Example

```
import React, { useState } from 'react'
import './App.css'

function Counter() {
  const [number, setNumber] = useState(100)

  const increment = () => setNumber(number + 1)
  const decrement = () => setNumber(number - 1)

  return (
    <div className="App">
      <h1 className="App-display">{number}</h1>
      <span className="App-controls">
        <button onClick={increment}>+</button>
        <button onClick={decrement}>-</button>
      </span>
    </div>
  )
}

export default Counter
```

Closer look at the previous example...

```
const [number, setNumber] = useState(100)
```

- Hook 是 function，他讓你可以從 function component 「hook into」React state 與生命週期功能
- "useState" 回傳一組數值：目前 state 數值和一個可以讓你更新 state 的 function (通常命名為 setXXX)
- "useState" 傳入的參數是這個 state 的初始值
- "useState()" 是 React 一個內建的 hook, 用它在 function component 中加入一個 local state (number)，React 會在重新 render 的頁面之間保留這些 state

State variables become local!

- 所以基本上可以不用管 this
- Using React class

```
handleInc = () => this.setState
                  (state => ({ count: state.count + 1 }));
handleDec = () => this.setState
                  (state => ({ count: state.count - 1 }));
...
<button onClick={this.handleInc}>+</button>
<button onClick={this.handleDec}>-</button>
```

- Using React Hooks

```
const increment = () => setNumber(number + 1)
const decrement = () => setNumber(number - 1)
...
<button onClick={increment}>+</button>
<button onClick={decrement}>-</button>
```

一個 function component 可以有多個 useState()

- 將不同的 states 宣告成不同的 local variables
- 基本上可以是任何型別的 object variable

```
function ExampleWithManyStates() {
  // 宣告多個 state 變數！
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState
                              ([{ text: 'Learn Hooks' }]);
```

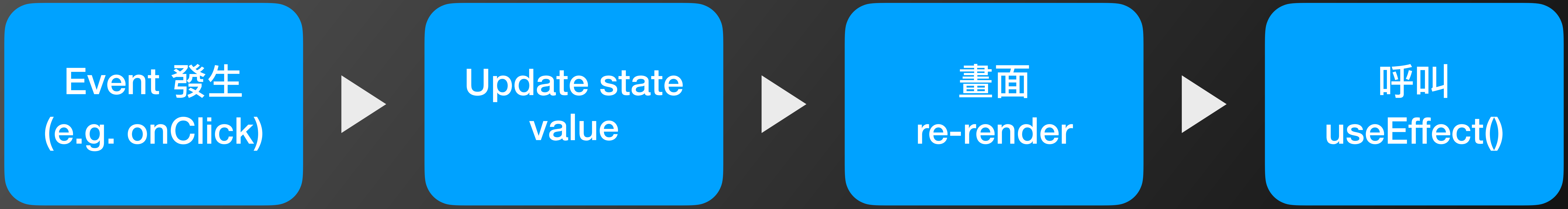
除了 useState()
另一個內建最常用的 Hook
叫 useEffect()

【useState() vs. useEffect()】

useState() 讓我們可以將某個事件 (e.g. button click) 綁定 state value 的 update

useEffect() 讓我們可以將 state value 的 update 綁定某種 side effect (e.g. 改變 document title, 設定 subscription, 改變 DOM 等)

像這樣...


Event 發生
(e.g. onClick)
Update state
value
畫面
re-render
呼叫
useEffect()

React 如何讓 state change 觸發 side effects?

- componentDidMount()
- componentDidUpdate()
- componentWillUnmount()

在 component 被 render() 以後，以及之後每次 state 被 update 都應該要產生一次 side effect

使用 life-cycle methods 與 useEffect() 的差別

- Using life-cycle methods

```
class Example extends React.Component {
  constructor(props) { ... }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={(state) => this.setState
                        ({ count: state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

使用 life-cycle methods 與 useEffect() 的差別

- Using useEffect()

```
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

但有時候 side effect 需要清除...

- (e.g.) 登入後 subscribe to some service, 登出後應該就要把 service 相關的資源清除掉。
- 直覺地來說，我們會在 component 裡頭紀錄 user.id, 以及 isStatusOnline, 然後在 componentWillUnmount() 去清除相關資源需求
- 但有時候同一個 component 會重新 render()，可能會綁定到別的 subscription, 因此，如果沒有在 render() 時也清除資源，就會造成系統的 memory leak or crash

```
  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
```

useEffect() 的資源清除機制

- useEffect() 讓使用者在 return 時指定一個 function, 作為資源清除的機制 (optional)　// 也可以是 anonymous arrow function
- 每次 component render() 時都會跟著 useEffect() 被呼叫一次，確保沒有資源沒有被清乾淨

```
  useEffect(() => {
    const handleStatusChange =
          (status) => setIsOnline(status.isOnline);
    ChatAPI.subscribeToFriendStatus(props.friend.id,
                                    handleStatusChange);
    // 指定如何在這個 effect 之後執行清除：
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
                                    handleStatusChange);
    };
  });
```

將不同 states 分成不同的 hooks，讓 code 更乾淨

- 讓同個 state 的邏輯放在一起，而不是被 life-cycle functions 拆開

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    const handleStatusChange =
                (status) => setIsOnline(status.isOnline);
    ChatAPI.subscribeToFriendStatus(props.friend.id,
                                    handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
                                          handleStatusChange);
    };
  });
  // ...
}
```

Hook 的規則

- 只在最上層呼叫 Hook
  - 不要在迴圈、條件式或是巢狀的 function 內呼叫 Hook
  - 確保當每次一個 component render 時 Hooks 都依照正確的順序被呼叫
- 只在 React Function 中呼叫 Hook
  - 別在一般的 JavaScript function 中呼叫 Hook
  - (i.e.) 在 React function component or 自己定義的 Hook 中呼叫 Hook

自行定義的 Hooks

- 只要遵循上頁的兩項規則，你也可以自行定義 Hooks
- 常見應用情境：
  有兩個相似的 functions 都會用到相同的 state hook,
  因此，可以共同的部分抽取出來變成一個自訂的 hook

```
function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    const handleStatusChange =
        (status) => setIsOnline(status.isOnline);
    ChatAPI.subscribeToFriendStatus
        (props.friend.id, handleStatusChange);
    return () =>{
      ChatAPI.unsubscribeFromFriendStatus
        (props.friend.id, handleStatusChange); };
  });
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

```
function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    const handleStatusChange =
        (status) => setIsOnline(status.isOnline);
    ChatAPI.subscribeToFriendStatus
        (props.friend.id, handleStatusChange);
    return () =>{
      ChatAPI.unsubscribeFromFriendStatus
        (props.friend.id, handleStatusChange); };
  });
  return (
    <li style={{ color: isOnline ? 'green' :
      'black' }}>{props.friend.name}
    </li>
  );}
```

自行定義的 Hooks

- 把 common part 提出來，照 convention 把 Hook 名稱前頭加個 'use'
- 兩種應用都需要 Hook 回傳 isOnline

```
 function useFriendStatus(friendID) {
   const [isOnline, setIsOnline] = useState(null);
   useEffect(() => {
     const handleStatusChange =
          (status) => setIsOnline(status.isOnline);
     ChatAPI.subscribeToFriendStatus(friendID,
                                    handleStatusChange);
     return () => {
         ChatAPI.unsubscribeFromFriendStatus(friendID,
                                            handleStatusChange);
     };
   };
   return isOnline;  // Hook 的回傳值型態可以自行定義
 }
```

用了自行定義的 Hook 之後

- 重複的 code 就不見了！

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);
  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

```
function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);
  return (
    <li style={{ color: isOnline ? 'green' :
      'black' }}>{props.friend.name}
    </li>
  );}
```

【其他內建的 Hooks】

useContext()
useReducer()

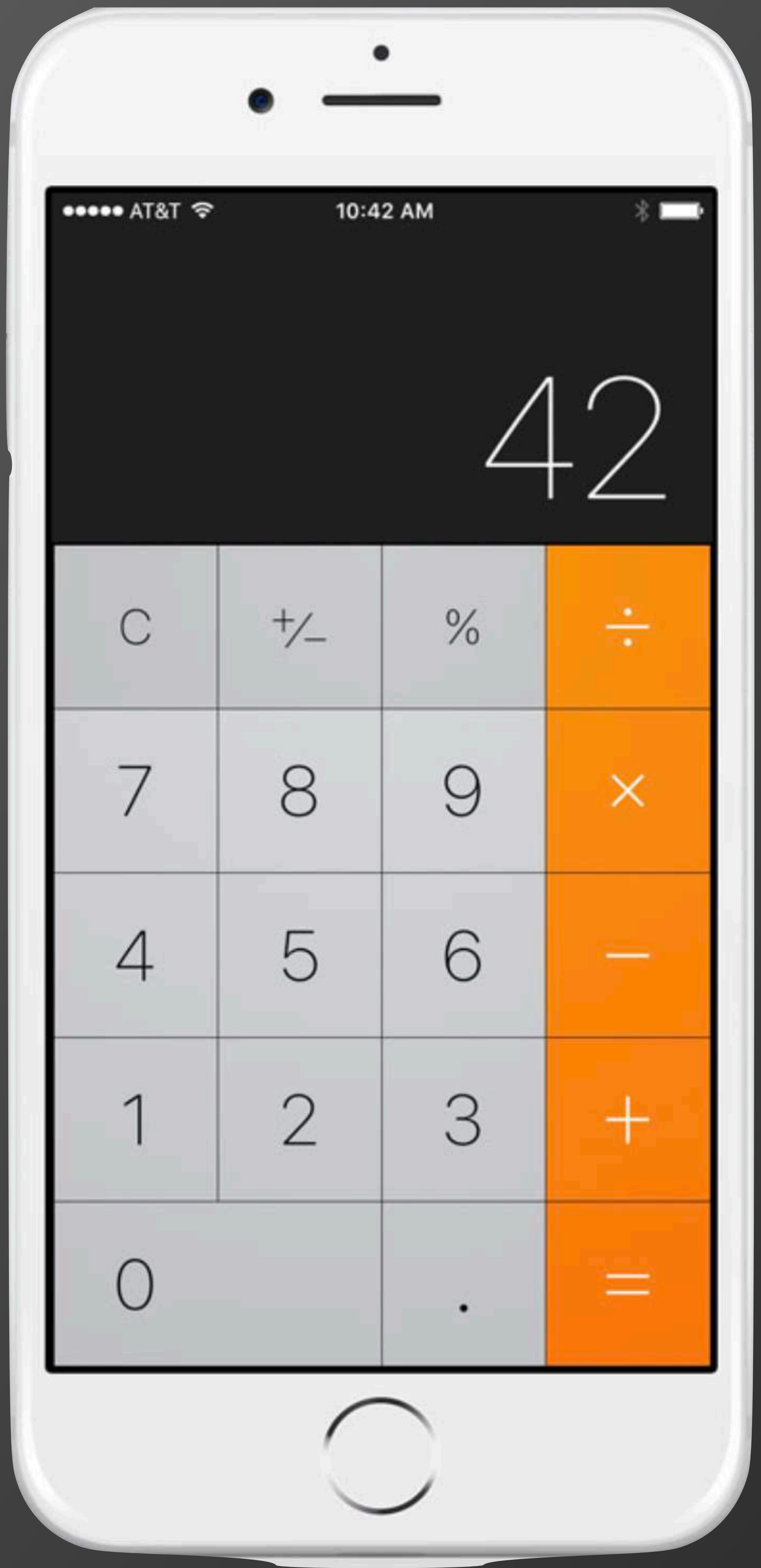
等我們下次教完 React Router 以及 Redux 再來學！

HW#3 寫得還順利嗎？

~~有沒有想要用 React Hooks 改寫的衝動啊…~~

建議還是用 class + state 寫，否則就沒有機會練習了… 要寫 hooks, 馬上就會有 HW#4 了

HW#4 will be online by 04/06....



- Topic: (TBD) a pure front-end calculator
- Basic layout is given
- Follow 一般手機上計算機的功能
  - e.g. 3 + 6, you will see 6
  - e.g. 3 + - 1 =, you will see 2
- Please do not change the class names. However, feel free to change the ref code if, for example, you want to use React Hooks.
- Due at 9pm, Monday, 04/19/2021

感謝聆聽！

Ric Huang / NTUEE

(EE 3035) Web Programming