



การใช้ REFS, STATE, EFFECT และ CONTEXT

030513188 : Selected Topics in Computer Engineering Technology
Dr. Phollakrit Wongsantisuk

การสร้างค่าอ้างอิง (Refs)

ตามปกตินั้น วิธีการอ้างอิงถึงอีลิเมนต์ในแบบของจาวาสคริปต์ จะมีเมธอดให้เลือกใช้หลายอัน เช่น `getElementById()`, `getElementsByName()`, `getElementsByTagName()`, `getElementsByClassName()`, `querySelector()` เป็นต้น แต่เนื่องจากการสร้างเว็บเพจตามหลักการของ React จะเป็นการนำหลายๆ คอมโพเนนต์มารวมกัน ซึ่งในแต่ละคอมโพเนนต์อาจมีองค์ประกอบบางอย่างที่มีลักษณะเหมือนกัน เช่น สมมติว่ามี `<button id="bt">` ในหลายๆ คอมโพเนนต์ ถ้าเราอ้างอิงด้วย `document.getElementById('bt')` จะเกิดปัญหาว่า เราหมายถึง button ที่อยู่ในคอมโพเนนต์ไหน แม้การใช้งานจริงมันก็จะเลือกอันแรกที่เจอมาให้เรา แต่ผลลัพธ์ที่ได้อาจไม่ตรงตามเป้าหมายที่ต้องการ ดังนั้น เราไม่ควรใช้เมธอดของจาวาสคริปต์ดังที่กล่าวมาเพื่อเข้าถึงอีลิเมนต์ในคอมโพเนนต์ แต่ควรสร้างค่าอ้างอิงของอีลิเมนต์ (Refs หรือ Reference) แล้วเข้าถึงผ่านค่าอ้างอิงนั้นแทน ซึ่งมีหลักการบางอย่างที่แตกต่างกันระหว่าง Class Component และ Function Component ดังรายละเอียดต่อไปนี้

การสร้างค่าอ้างอิง (Refs)

- การสร้างค่าอ้างอิงใน Class Component

กรณีของ Class Component ให้สร้างค่าอ้างอิงด้วยเมธอด `createRef()` ซึ่งเราอาจนำเข้าเมธอดนี้โดยตรงจากโมดูล 'react' หรือจะเรียกใช้งานผ่านคอมโพเนนต์ React ก็ได้ เช่น

//วิธีที่ 1

```
import React, { createRef } from 'react'
```

```
class MyClass extends React.Component {  
  button = createRef() //หรือจะกำหนดไว้ใน constructor ก็ได้  
  ...  
}
```

//วิธีที่ 2

```
import React from 'react'
```

```
class MyClass extends React.Component {  
  button = React.createRef() //หรือจะกำหนดไว้ใน constructor ก็ได้  
  ...  
}
```

การสร้างค่าอ้างอิง (Refs)

ภายในคอมโพเนนต์ หลังจากที่เราสร้างตัวแปรสำหรับอ้างอิงถึงคอมโพเนนต์ด้วย `createRef()` ดังโค้ดที่ผ่านมา ต่อไปก็ต้องเชื่อมโยงกับอีลีเมนต์เป้าหมายโดยการนำตัวแปรดังกล่าวมากำหนดให้แก่พร็อพเพอร์ตี้ **ref** ของอีลีเมนต์นั้นๆ เช่น

```
buttonOK = React.createRef()
...
return <button ref={this.buttonOK}>OK</button>
```

ถ้ามีอีลีเมนต์อื่นๆ อีก ที่เราต้องอ้างอิง ก็ให้สร้างค่าอ้างอิงด้วย `createRef()` แยกแต่ละอีลีเมนต์ในแบบเดียวกันไปจนครบ เช่น

```
div = createRef()
span = createRef()
button = createRef()
...
return (
  <div ref={this.div}>
    <span ref={this.span}>...</span>
    <button ref={this.button}>...</button>
  </div>
)
```

การสร้างค่าอ้างอิง (Refs)

เมื่อเราต้องการเข้าถึงอิลิเมนต์ที่ได้สร้างค่าอ้างอิงเอาไว้ เช่น เพื่ออ่านหรือกำหนดข้อมูล หรือการเปลี่ยนแปลง สไตล์ หรืออื่นๆ ให้อ้างอิงด้วยรูปแบบดังนี้

```
this.<ref>.current.<property_or_method>
```

- ref คือชื่ออ้างอิงที่เรากำหนดได้เอาไว้
- current ต้องระบุลงไปด้วย เพื่อบ่งชี้ว่าเป็นอิลิเมนต์ที่อยู่ในคอมโพเนนต์ปัจจุบัน ทั้งนี้เพราะในคอมโพเนนต์อื่นๆ ซึ่งนำมารวมกันเป็นเว็บเพจ อาจมีอิลิเมนต์ที่ระบุชื่ออ้างอิงซ้ำกันก็ได้
- property_or_method คือพร็อพเพอร์ตี้หรือเมธอดของอิลิเมนต์ในแบบ DOM ที่เราต้องการเข้าถึงเช่น innerHTML, innerText, value (อินพุตของฟอร์ม), style, focus(), submit() และอื่นๆ ทั้งหมดตามข้อกำหนดของจาวาสคริปต์ เช่น

```
const title = this.button.current.innerText  
this.span.current.innerHTML = '<b>Hello</b>'  
this.div.current.style.backgroundColor = 'yellow'
```

การสร้างค่าอ้างอิง (Refs)

- การสร้างค่าอ้างอิงใน Function Component

กรณีของ Function Component ให้สร้างค่าอ้างอิงด้วยเมธอด `useRef()` ซึ่งจัดอยู่ในกลุ่ม React Hook หรือ `React.createRef()` วิธีใดวิธีหนึ่ง ดังโค้ดถัดไป แล้วเชื่อมโยงกับอิลิเมนต์เป้าหมายเช่นเดียวกับ Class Component เพียงแต่ไม่ต้องระบุคำว่า `this` เพราะเป็น Function Component

//วิธีที่ 1

```
import React, { useRef } from 'react'
function MyFunction() {
  const div = useRef()
  const button = useRef()
  ...
  return (
    <div ref={div}>
      <button ref={button}>OK</button>
    </div>
  )
}
```

//วิธีที่ 2

```
import React from 'react'
function MyFunction() {
  const div = React.useRef() //หรือใช้ React.createRef()
  const button = React.useRef() //หรือใช้ React.createRef()
  ...
  return (
    <div ref={div}>
      <button ref={button}>OK</button>
    </div>
  )
}
```

การสร้างค่าอ้างอิง (Refs)

ส่วนการเข้าถึงอีลิเมนต์ที่สร้างค่าอ้างอิงเอาไว้ ก็ทำแบบเดียวกับ Class Component เพียงแต่ไม่ต้องระบุคำว่า `this` เช่น

```
const title = button.current.innerText  
div.current.style.backgroundColor = 'yellow'
```

สำหรับการใช้งานโดยรวม ให้ดูจากการทดสอบต่อไปนี้ โดยภายในคอมโพเนนต์จะประกอบด้วย `span` 2 อันที่กำหนดตัวเลขไว้อย่างแน่นอน และ `span` อีก 2 อันสำหรับแสดงเครื่องหมายที่ใช้คำนวณ และผลลัพธ์จากการคำนวณตามลำดับ นอกจากนี้ก็จะมีปุ่มสำหรับคลิกเลือกเครื่องหมาย เมื่อคลิกปุ่มใด จะอ่านค่าจาก `span` ที่แสดงตัวเลขและเครื่องหมายบนปุ่มนั้น แล้วนำไปคำนวณเมื่อได้ผลที่ออกมาก็นำไปแสดงใน `span` ผลลัพธ์

การสร้างค่าอ้างอิง (Refs)

react/app1/src/refs-class.js

```
import React, { createRef } from 'react'

export default class RefsClass extends React.Component {
  num1 = createRef()
  num2 = createRef()
  operator = React.createRef()
  result = React.createRef()

  calculate = (ev) => {
    let op = ev.target.innerText
    let n1 = parseInt(this.num1.current.innerText)
    let n2 = parseInt(this.num2.current.innerText)
    let r = eval(`${n1} ${op} ${n2}`)
    this.result.current.innerText = r
    this.operator.current.innerText = op
  }
```

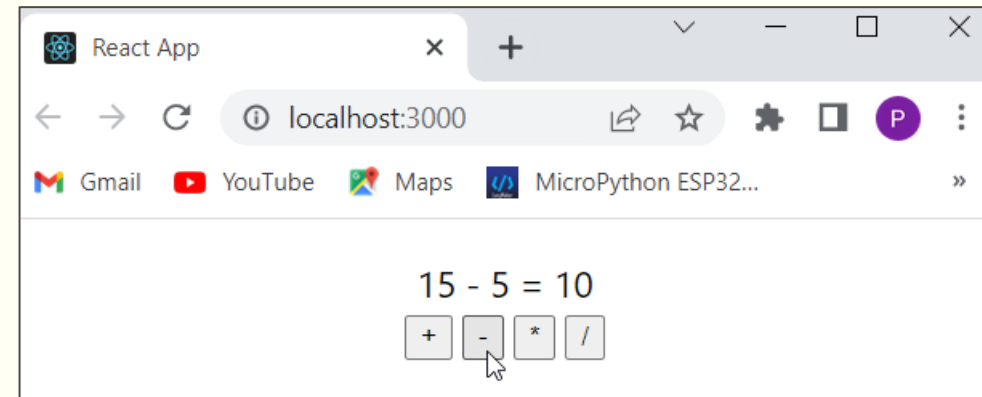
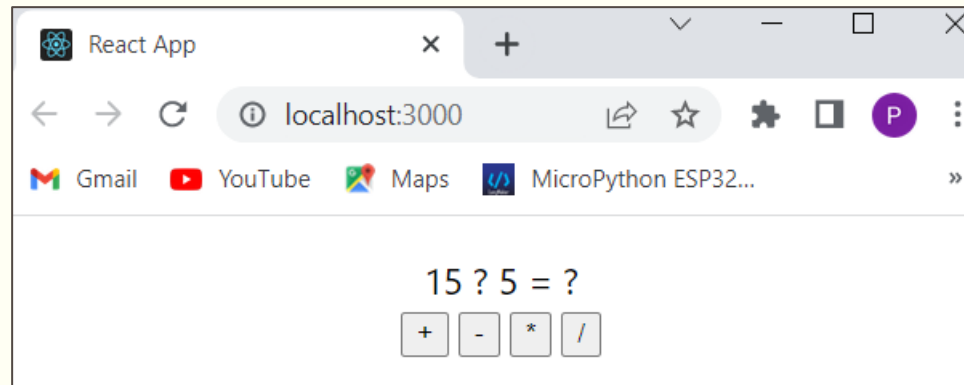
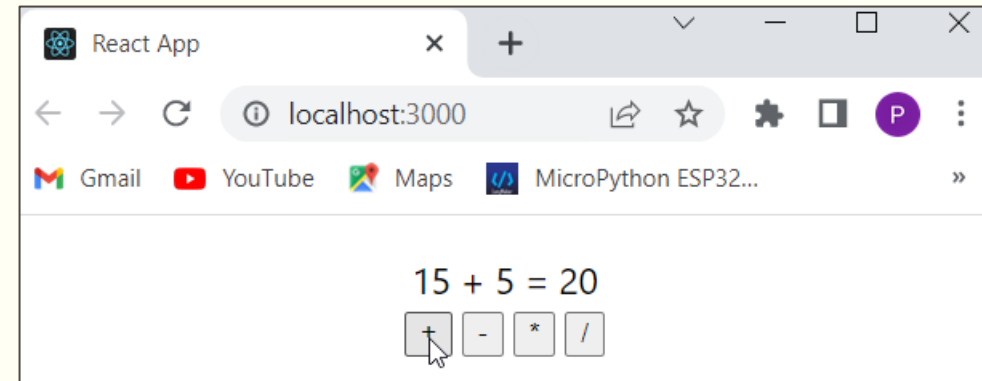
```
render() {
  return (
    <div style={{ textAlign: 'center', marginTop: 20, fontSize: 18 }}>
      <span ref={this.num1}>15</span>&nbsp;
      <span ref={this.operator}>?</span>&nbsp;
      <span ref={this.num2}>5</span>&nbsp;
      <span>=</span>&nbsp;
      <span ref={this.result}>?</span>
    </div>
    <button onClick={this.calculate}>+</button>&nbsp;
    <button onClick={this.calculate}>-</button>&nbsp;
    <button onClick={this.calculate}>*</button>&nbsp;
    <button onClick={this.calculate}>/</button>
  </div>
)
}
```


การสร้างค่าอ้างอิง (Refs)

react/app1/src/App.js

```
import React from 'react'
import RefsClass from './refs-class'

export default function App() {
  return <RefsClass/>
}
```



การสร้างค่าอ้างอิง (Refs)

react/app1/src/refs-func.js

```
import React, { useRef } from 'react'

export default function RefsFunc() {
  const num1 = useRef()
  const num2 = useRef()
  const operator = React.useRef()
  const result = React.useRef()

  const calculate = (ev) => {
    let op = ev.target.innerText
    let n1 = parseInt(num1.current.innerText)
    let n2 = parseInt(num2.current.innerText)
    let r = eval(`${n1} ${op} ${n2}`)
    result.current.innerText = r
    operator.current.innerText = op
  }
```

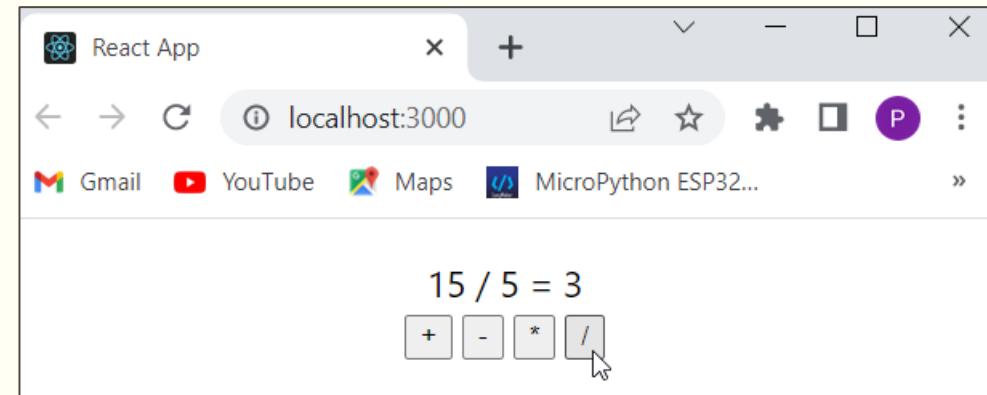
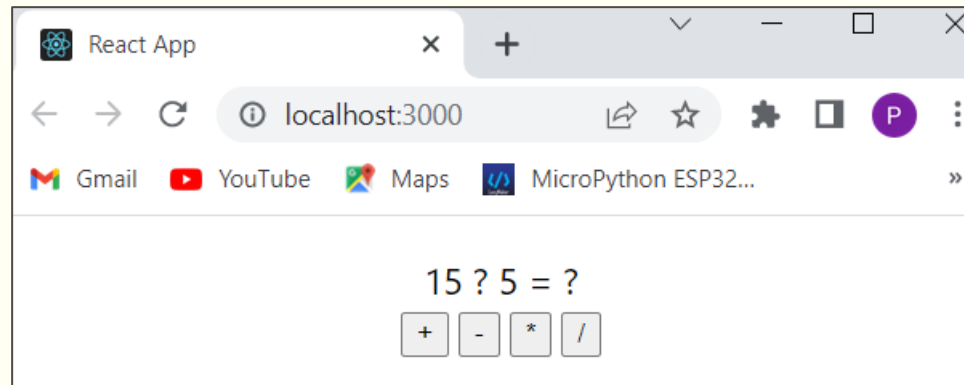
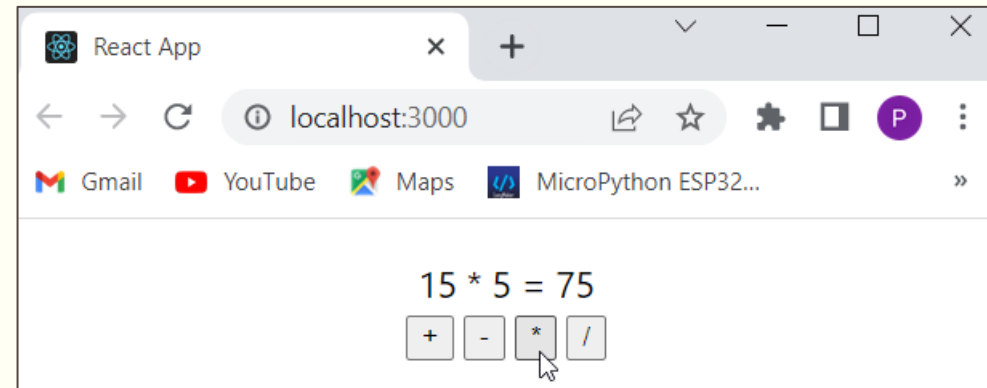
```
return (
  <div style={{ textAlign: 'center', marginTop: 20, fontSize: 18 }}>
    <span ref={num1}>15</span>&nbsp;
    <span ref={operator}>?</span>&nbsp;
    <span ref={num2}>5</span>&nbsp;
    <span>=</span>&nbsp;
    <span ref={result}>?</span>
    <div>
      <button onClick={calculate}>+</button>&nbsp;
      <button onClick={calculate}>-</button>&nbsp;
      <button onClick={calculate}>*</button>&nbsp;
      <button onClick={calculate}>/</button>
    </div>
  </div>
)
```

การสร้างค่าอ้างอิง (Refs)

react/app1/src/App.js

```
import React from 'react'
import RefsFunc from './refs-func'

export default function App() {
  return <RefsFunc/>
}
```



การสร้างค่าอ้างอิง (Refs)

- **การสร้างค่าอ้างอิงแบบอาร์เรย์**

บางครั้ง อิลิเมนต์ที่เราต้องการอ้างอิงอาจมีเป็นจำนวนมาก เช่น เซลล์หรือแถวของตาราง จึงอาจยุ่งยากต่อการสร้างค่าอ้างอิงให้ครบทุกอิลิเมนต์ ซึ่งในลักษณะเช่นนี้ เราอาจเปลี่ยนมาใช้วิธีการแบบอาร์เรย์เพื่อสร้างค่าอ้างอิงหลายๆ อิลิเมนต์ในตัวแปรเดียวกัน ดังแนวทางต่อไปนี้

- กำหนดตัวแปรเพื่อให้เป็นค่าอ้างอิงสำหรับอิลิเมนต์ในกลุ่มนั้นเพียงตัวเดียว พร้อมระบุค่าเริ่มต้นให้เป็นอาร์เรย์ว่าง
- ในขั้นตอนการแสดงผลคอมโพเนนต์ ซึ่งเราต้องกำหนดค่าอ้างอิงของอิลิเมนต์ โดยวิธีที่สะดวกที่สุดก็คือใช้ Arrow Function พร้อมระบุเลขลำดับให้กับอิลิเมนต์นั้น เช่น

```
const buttons = React.useRef([])

...

return (
  <div>
    <button ref={el => buttons.current[0] = el}>One</button><br/>
    <button ref={el => buttons.current[1] = el}>Two</button><br/>
    <button ref={el => buttons.current[2] = el}>Three</button><br/>
    ...
  </div>
)
```

การสร้างค่าอ้างอิง (Refs)

- หากมีอีลีเมนต์ชนิดเดียวกันเป็นจำนวนมากที่กำหนดค่าอ้างอิง เพื่อลดความยุ่งยากในการเขียนโค้ดเราควรนำค่าบางอย่างของแต่ละอีลีเมนต์ มาสร้างเป็นอาร์เรย์ แล้วใช้เมธอด `map()` เพื่อแสดงอีลีเมนต์ เช่น จากโค้ดที่ผ่านมา อาจแก้ไขเป็นดังนี้

```
const buttons = React.useRef([])
const title = ['One', 'Two', 'Three', 'Four', 'Five', ...]
...
return (
  <div>
    {
      title.map((t,i) => {
        return (
          <button ref={el => buttons.current[i] = el}>{t}</button><br/>
        )
      })
    }
  </div>
)
```

การสร้างค่าอ้างอิง (Refs)

- หากเราต้องการอ้างอิงถึงอีลิเมนต์ที่สร้างค่าอ้างอิงแบบอาร์เรย์ จะต้องระบุเลขลำดับของอีลิเมนต์นั้นลงไปด้วย เช่น

```
let bt1Title = buttons.current[0].innerText  
buttons.current[1].style.color = 'red'
```

การสร้างค่าอ้างอิง (Refs)

ตัวอย่างถัดไปเป็นการแสดงตาราง โดยในแต่ละแถวจะมีปุ่มสำหรับคลิกเพื่อลบแถวนั้นออกไป ทั้งนี้แม้จะทำได้หลายวิธี แต่เพื่อให้สอดคล้องกับที่เราศึกษาในหัวข้อนี้ จะสร้างค่าอ้างอิงสำหรับแต่ละแถวของตารางในแบบอาร์เรย์ แล้วที่แต่ละปุ่มก็จะแนบลำดับแถวเพื่อส่งไปยังตัวจัดการอีเวนต์สำหรับใช้เป็นตัวกำหนดให้ลบแถวในลำดับนั้นออกจากตาราง ส่วนรายละเอียดปลีกย่อยอื่นๆ ให้ดูจากโค้ดต่อไปนี่

react/app1/src/refs-array.js

```
import React from 'react'
```

```
export default function RefsArray() {
```

```
  const table = React.useRef()
```

```
  const tr = React.useRef([])
```

```
  const data = [
```

```
    ['JavaScript', 100],
```

```
    ['React', 150],
```

```
    ['React Native', 180],
```

```
    ['Node.js', 200],
```

```
    ['VS Code', 120]
```

```
  ]
```

```
  const onClickButton = (i) => {
```

```
    //การลบแถวออกจากตาราง จะทำให้ลำดับแถวเปลี่ยนไปจากเดิม
```

```
    //ซึ่งอาจไม่ตรงกับลำดับอ้างอิงของแถวนั้นที่ได้กำหนดไว้ล่วงหน้า
```

```
    //ดังนั้น เราต้องตรวจสอบลำดับที่แท้จริงอีกครั้ง เพื่อใช้ในการลบ
```

```
    const index = tr.current[i].rowIndex
```

```
    table.current.deleteRow(index)
```

```
  }
```

การสร้างค่าอ้างอิง (Refs)

react/app1/src/refs-array.js (ต่อ)

```
const onMouseOverRow = (i) => {
  tr.current[i].style.backgroundColor = 'yellow'
}

const onMouseOutRow = (i) => {
  tr.current[i].style.backgroundColor = 'white'
}

const tableStyles = { margin: 'auto', marginTop: 30 }

return (
  <table ref={table} border="1" cellPadding={5} style={tableStyles} >
    <tr><th>Product</th><th>Price</th><th>Delete</th></tr>
    {
```

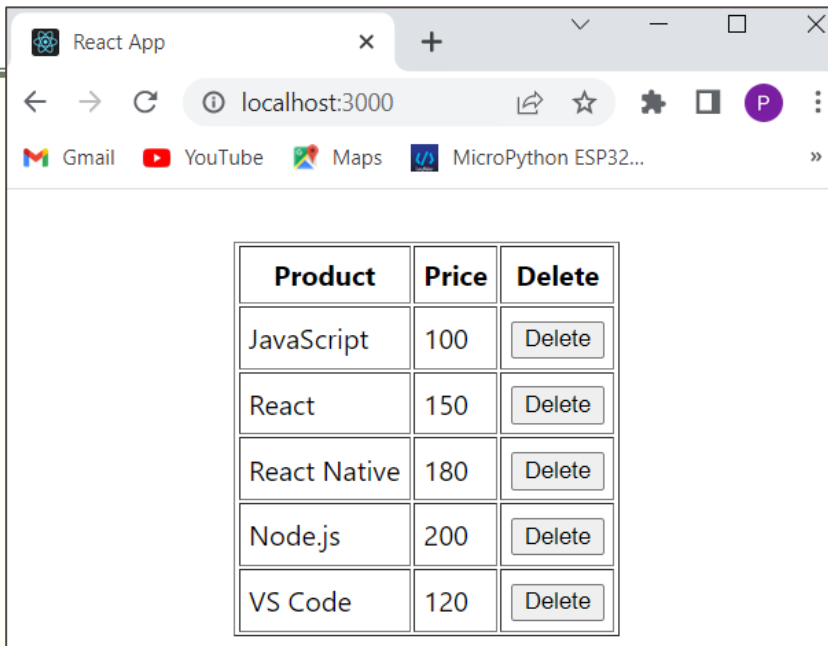
```
data.map((item, i) => {
  return (
    <tr ref={el => tr.current[i] = el}
      onMouseOver={() => onMouseOverRow(i)}
      onMouseOut={() => onMouseOutRow(i)}
    >
      <td>{item[0]}</td>
      <td>{item[1]}</td>
      <td><button onClick={() => onClickButton(i)}>
        Delete</button></td>
    </tr>
  )
})
}
</table>
)
```


การสร้างค่าอ้างอิง (Refs)

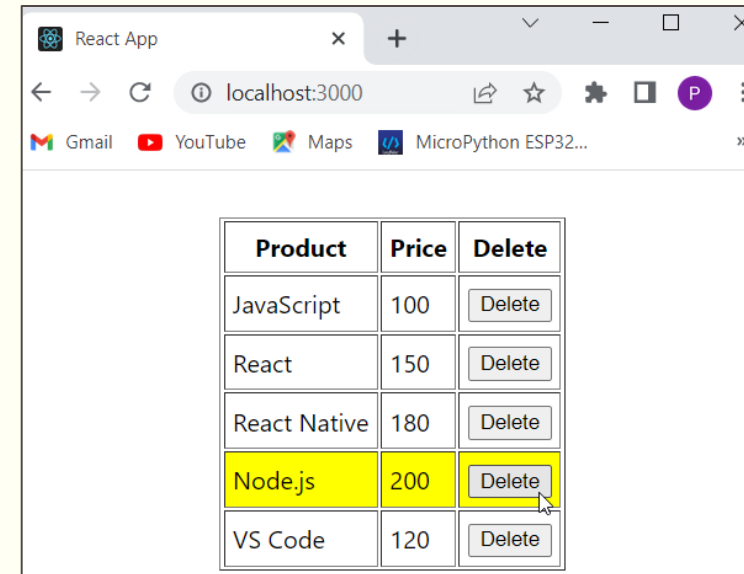
react/app1/src/App.js

```
import React from 'react'
import RefsArray from './refs-array'

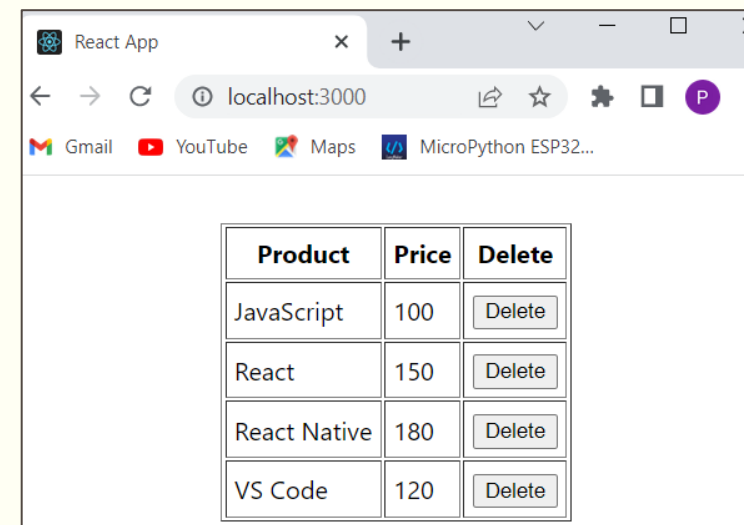
export default function App() {
  return <RefsArray/>
}
```



Product	Price	Delete
JavaScript	100	Delete
React	150	Delete
React Native	180	Delete
Node.js	200	Delete
VS Code	120	Delete



Product	Price	Delete
JavaScript	100	Delete
React	150	Delete
React Native	180	Delete
Node.js	200	Delete
VS Code	120	Delete



Product	Price	Delete
JavaScript	100	Delete
React	150	Delete
React Native	180	Delete
VS Code	120	Delete

ความรู้เพิ่มเติมเกี่ยวกับ Array Destructuring

Array Destructuring เป็นเทคนิคการนำค่าจากอาร์เรย์มากำหนดให้แก่ตัวแปรหลายๆ ตัว เพื่อใช้งานอื่นๆ ต่อไป ซึ่งตามปกตินั้นหากเราต้องการเข้าถึงค่าของสมาชิกแต่ละตัวในอาร์เรย์ ก็อาจต้องใช้เลขลำดับในการอ้างอิง เช่น

```
let arr = [1, 2, 3, 4, 5]
let a = arr[0]
let b = arr[1]
```

แต่ยังมีวิธีการที่ง่ายกว่าหากต้องการนำค่าจากอาร์เรย์ไปกำหนดให้กับตัวแปรแบบแยกกัน นั่นคือใช้วิธีการที่เรียกว่า Array Destructuring ในลักษณะดังนี้

```
let [a, b, c] = [1, 2, 3]
console.log(a)    //1
console.log(b)    //2
console.log(c)    //3
```

ความรู้เพิ่มเติมเกี่ยวกับ Array Destructuring

หรือหากเราต้องการใช้ข้อมูลเพียงบางส่วน ก็อาจสร้างตัวแปรเท่าที่จำเป็น และอาจแทนส่วนที่เหลือด้วย Spread Operator (...) นอกจากนี้ ยังอาจป้องกันกรณีที่มีจำนวนสมาชิกในอาร์เรย์ไม่ครบตามจำนวนตัวแปรด้วยการกำหนดค่าดีฟอลต์แทนค่าที่อาจขาดหายไป เช่น

```
let [a, b] = [1, 2, 3, 4, 5]
```

```
//a = 1, b = 2
```

```
let [c, d, ...e] = [6, 7, 8, 9]      //... คือ spread operator
```

```
//c = 6, d = 7, e = [8, 9], e[0] = 8, e[1] = 9
```

```
let [f, g, h=12, i] = [10, 11]      //กำหนดค่าดีฟอลต์ของ h เป็น 12
```

```
//f = 10, g = 11, h = 12, i = undefined
```

การจัดเก็บข้อมูลสถานะด้วย State

React Hook ที่เราจะศึกษาในลำดับต่อไปคือ State ซึ่งเป็นการจัดเก็บข้อมูลสถานะของคอมโพเนนต์โดยมีลักษณะที่สำคัญคือ เมื่อข้อมูลที่จัดเก็บในแบบ State เปลี่ยนแปลงไป คอมโพเนนต์ที่เป็นเจ้าของข้อมูลนั้นจะแสดงผลใหม่ทันที (re-render) ทั้งนี้ State ถือเป็นสิ่งสำคัญอย่างยิ่งที่เราต้องนำไปใช้งานอยู่ตลอดทั้งใน React และ React Native สำหรับแนวทางการใช้งานนั้น จะแยกพิจารณาระหว่าง Class Component และ Function Component ดังนี้

- **การกำหนด State สำหรับ Class Component**

สำหรับกรณีของ Class Component นั้น เราจะสร้าง State ให้เป็นพร็อพเพอร์ตี้ของคลาส แล้วจัดเก็บข้อมูลย่อยๆ แต่ละอย่างในแบบออบเจกต์ ซึ่งอาจเลือกใช้วิธีใดวิธีหนึ่งดังต่อไปนี้

- วิธีที่ 1 สร้าง State ในคอนสตรัคเตอร์ เช่น

```
class Cart extends React.Component {  
  constructor() {  
    super()  
    this.state = { numItems: 0 }  
  }  
  ...  
}
```

การจัดเก็บข้อมูลสถานะด้วย State

- วิธีที่ 2 สร้าง State ในแบบพร็อพเพอร์ตี้ของคลาสโดยตรง เช่น

```
class Cart extends React.Component {  
  state = { numItems: 0 }  
  ...  
}
```

ภายในคอมโพเนนต์ เราสามารถนำค่าจาก State ไปใช้งานตามต้องการ สำหรับกรณีของ Class Component ให้อ้างถึงค่าในรูปแบบดังนี้

```
this.state.<name>
```

โดย name ก็ชื่อหรือพร็อพเพอร์ตี้ของข้อมูลที่จัดเก็บในออบเจกต์ State ที่เราต้องการอ่านค่านั่นเอง เช่น

```
let n = this.state.numItems
```

ส่วนการแก้ไขค่าของ State จะต้องใช้เมธอดที่ชื่อ setState() โดยเรียกเมธอดนี้เมื่อต้องการเปลี่ยนค่า State แล้วกำหนดค่าใหม่ในแบบออบเจกต์ เช่น

```
this.setState({numItems: 2})
```

การจัดเก็บข้อมูลสถานะด้วย State

react/app1/src/state-class.js

```
import React from 'react'
```

```
export default class Cart extends React.Component {  
  state = { numItems: 0 }
```

```
  onClickAddCart = () => {  
    let n = this.state.numItems  
    n++  
    this.setState({ numItems: n })  
  }
```

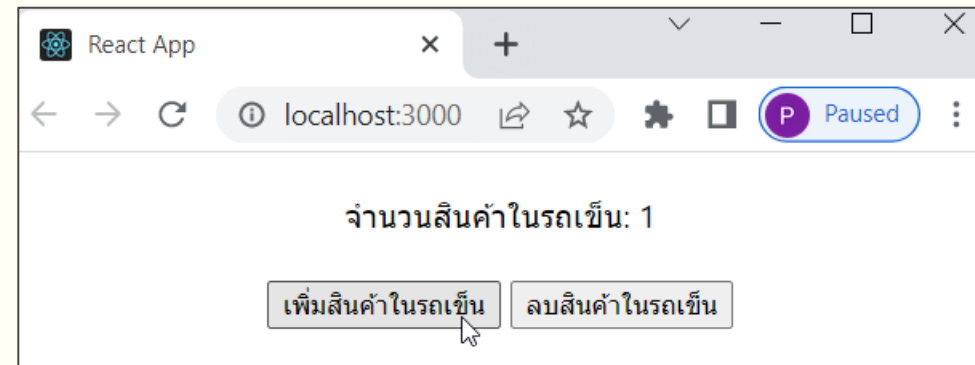
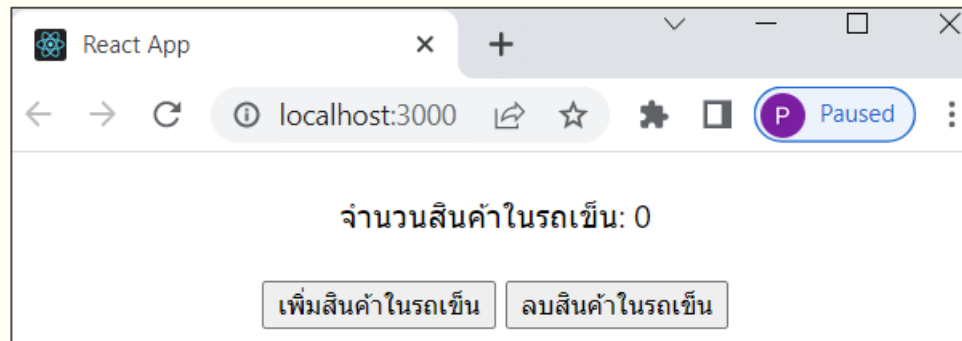
```
  onClickDeleteCart = () => {  
    if (this.state.numItems > 0) {  
      this.setState({ numItems: this.state.numItems - 1 })  
    }  
  }  
  
  render() {  
    return (  
      <div style={{textAlign:'center', marginTop: 20}}>  
        <div>จำนวนสินค้าในรถเข็น: {this.state.numItems}</div><br/>  
        <button onClick={this.onClickAddCart}>เพิ่มสินค้าในรถเข็น</button>&nbsp;  
        <button onClick={this.onClickDeleteCart}>ลบสินค้าในรถเข็น</button>  
      </div>  
    )  
  }  
}
```

การจัดเก็บข้อมูลสถานะด้วย State

react/app1/src/App.js

```
import React from 'react'
import Cart from './state-class'

export default function App() {
  return <Cart/>
}
```



การจัดเก็บข้อมูลสถานะด้วย State

- **การกำหนด State สำหรับ Function Component**

การใช้ State ร่วมกับ Function Component จะมีขั้นตอนบางส่วนที่แตกต่างไปจาก Class Component ซึ่งสามารถสรุปหลักการที่สำคัญได้ดังนี้

- การสร้าง State สำหรับ Function Component มีฟังก์ชันในกลุ่ม React Hook ให้ใช้งานโดยตรงอยู่แล้ว นั่นคือ `useState()` ซึ่งอาจนำเข้าฟังก์ชันนี้โดยตรงจากโมดูล 'react' หรือเรียกผ่านคอมโพเนนต์ React ก็ได้

//วิธีที่ 1

```
import React, {useState} from 'react'

function MyComponent() {
  let [...] = useState()
}
```

//วิธีที่ 2

```
import React from 'react'

function MyComponent() {
  let [...] = React.useState()
}
```


การจัดเก็บข้อมูลสถานะด้วย State

- ภายในคอมโพเนนต์ เราต้องกำหนดข้อมูลของ State ด้วยวิธีการแบบ Array Destructuring พร้อมกำหนดค่าเริ่มต้นให้กับมันด้วยฟังก์ชัน useState() เช่น สมมติว่าเราต้องการสร้าง State เพื่อเก็บขนาดของฟอนต์ภายใน Function Component ก็อาจกำหนดค่า ดังนี้

```
import React, {useState} from 'react'

function MyComponent() {
  let [fontSize, setFontSize] = useState(16)
  ...
}
```

- ◉ [fontSize, setFontSize] คือการสร้างตัวแปรแบบ Array Destructuring
- ◉ fontSize คือตัวแปรที่ใช้เก็บค่า State
- ◉ setFontSize คือชื่อฟังก์ชันที่จะใช้ในการเปลี่ยนค่า State ซึ่งส่วนใหญ่เรานิยมให้เริ่มต้นด้วยคำว่า set แล้วตามด้วยชื่อตัวแปรแต่ไม่ใช่ข้อบังคับ
- ◉ useState(16) เป็นฟังก์ชันที่ใช้สร้างและกำหนดค่าเริ่มต้นให้แก่ State โดยในที่นี้คือตัวแปร fontSize ซึ่งจะมีค่าเริ่มแรกเป็น 16 หรือ fontSize = 16 นั่นเอง

การจัดเก็บข้อมูลสถานะด้วย State

- การนำค่าของ State ไปใช้งาน ก็ระบุชื่อตัวแปรนั้นโดยตรงเหมือนกับตัวแปรทั่วไป
- สำหรับการแก้ไขหรืออัปเดตค่า State ให้กำหนดผ่านทางฟังก์ชันที่ระบุไว้ใน Array Destructuring ตอนสร้าง State ซึ่งจากโค้ดที่ผ่านมาคือ `setFontSize` แต่หากเราแก้ไขค่าของตัวแปร State โดยตรงจะเกิดข้อผิดพลาด เช่น

```
function MyComponent() {  
  let [fontSize, setFontSize] = useState(16)  
  ...  
  //สมมติว่าเมื่อคลิกปุ่ม zoom แล้วให้เพิ่มขนาดฟอนต์  
  const onClickButtonZoomIn = () => {  
    let newSize = fontSize + 2 //อ้างอิงตัวแปรใน State  
    setFontSize(newSize)      //อัปเดตค่าของ State
```

```
    /*  
    fontSize = 18 //Error เพราะแก้ไขค่าที่ตัวแปร State โดยตรง  
    fontSize = fontSize + 2 //Error  
    setFontSize(++fontSize) //Error (ผลลัพธ์ของ ++ จะเก็บไว้ในตัว  
                           แปรนั้น)  
    */  
  }  
  ...  
  <button onClick={onClickButtonZoomIn}>Zoom In</button>  
}
```

// หรือแบบง่ายๆ คือ เรียกฟังก์ชันที่ใช้อัปเดตค่าของ State โดยตรง เช่น

```
<button onClick={() => setFontSize(fontSize + 2)}>Zoom In</button>
```

การจัดเก็บข้อมูลสถานะด้วย State

- หลังจากที่เราเปลี่ยนค่าของ State คอมโพเนนต์จะแสดงผลใหม่หรือ re-render ทันที เช่นเดียวกับการเปลี่ยน State ใน Class Component ดังที่กล่าวมาแล้ว ทั้งนี้ เราสามารถเพิ่มการจัดเก็บข้อมูลอื่นๆ ไว้ใน State ได้ตามต้องการ เช่น

```
let [fontSize, setFontSize] = useState(16)
let [color, setColor] = useState()      //ไม่ระบุค่าเริ่มต้นก็ได้
let [fontWeight, setFontWeight] = useState('normal')
let [disabled, setDisabled] = useState(false)
```

การจัดเก็บข้อมูลสถานะด้วย State

react/app1/src/state-func.js

```
import React, {useState} from 'react'

export default function MessageBox() {
  let [text, setText] = useState('Hello World')
  let [size, setSize] = React.useState(16)

  const onClickSetText = () => {
    let t = prompt('กำหนดข้อความ')
    if (t) {
      setText(t)
    }
  }

  const onClickZoomIn = () => {
    let newSize = size + 1
    setSize(newSize)
  }
}
```

```
let msgboxSyle = {
  display: 'inline-block',
  width: 350,
  fontSize: size,    //ใช้ค่าจาก State
  backgroundColor: '#ccc',
  padding: 5,
  textAlign: 'left'
}

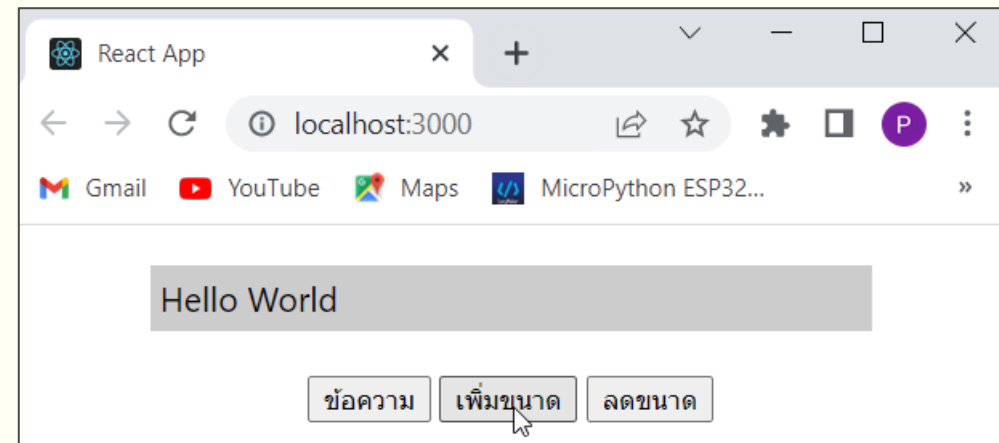
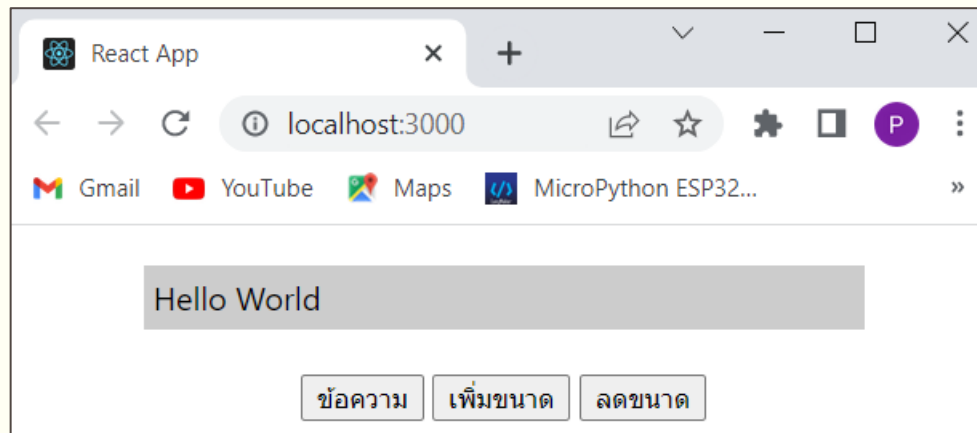
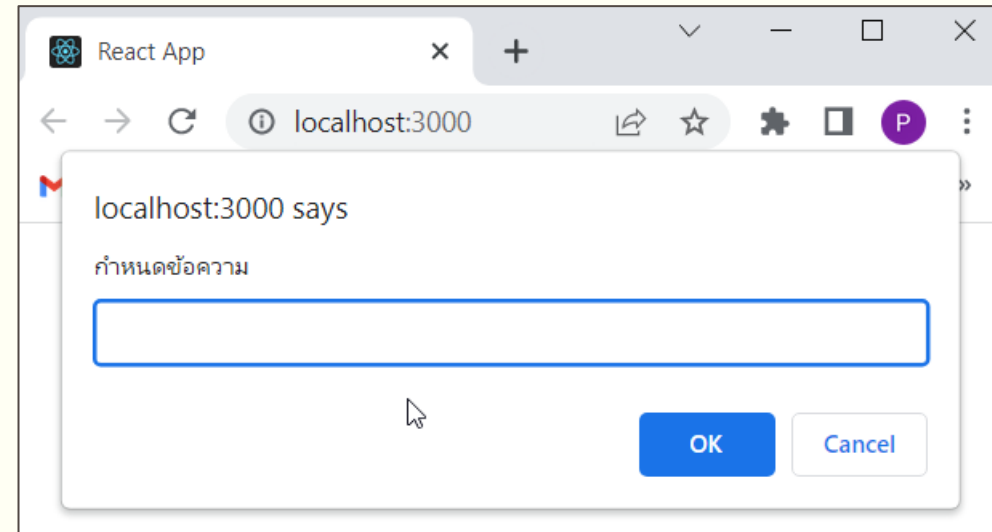
return (
  <div style={{textAlign:'center', marginTop:20}}>
    <div style={msgboxSyle}>{text}</div><br/><br/>
    <button onClick={onClickSetText}>ข้อความ</button>&nbsp;
    <button onClick={onClickZoomIn}>เพิ่มขนาด</button>&nbsp;
    <button onClick={() => setSize(size - 1)}>ลดขนาด</button>
  </div>
)
```

การจัดเก็บข้อมูลสถานะด้วย State

react/app1/src/App.js

```
import React from 'react'
import MessageBox from './state-class'

export default function App() {
  return <MessageBox/>
}
```



การตอบสนองต่อ Effect

ใน React นั้น มีลักษณะบางอย่างที่ถือว่าเป็นผลกระทบ (Effect) ต่อคอมโพเนนต์ เช่น การแสดงผลครั้งแรก, การแสดงผลซ้ำ (re-render), การเปลี่ยนแปลงค่าใน State, การรับส่งข้อมูลกับเซิร์ฟเวอร์ เป็นต้น นอกจากนี้ก็ยังมีลักษณะอื่นๆ อีกมากมายที่จัดว่าเป็น Effect ซึ่งในบางกรณี เราอาจต้องการตอบสนองหรือกำหนดการกระทำบางอย่างเมื่อมี Effect เกิดขึ้น แต่จะใช้วิธีการที่แตกต่างกันระหว่าง Function Component และ Class Component ดังรายละเอียดต่อไปนี้

- **การเกิด Effect กับ Function Component**

ในกรณีของ Function Component จะมีฟังก์ชันในกลุ่ม React Hook สำหรับดำเนินการเมื่อเกิด Effect มาให้เราใช้งานโดยตรงอยู่แล้ว นั่นคือ `useEffect()` โดยมีหลักการดังนี้

- เราอาจนำเข้าฟังก์ชัน `useEffect()` โดยตรง หรือจะเรียกผ่านคอมโพเนนต์ React ก็ได้
- การกระทำหรือการตอบสนองเมื่อเกิด Effect เราจะกำหนดในแบบ callback ให้แก่ฟังก์ชัน `useEffect()` ดังรูปแบบต่อไปนี้

```
useEffect(callback [, dependencies])
```

การตอบสนองต่อ Effect

```
import React, { useEffect } from 'react'

...

function MyComponent() {
  useEffect(function() {
    //สิ่งที่จะดำเนินการเมื่อเกิด Effect
  })

  /* หรือวิธีอื่นๆ เช่น
  React.useEffect(() => { //ใช้ Arrow Function
    //สิ่งที่จะดำเนินการเมื่อเกิด Effect
  })
  */

  return (...)
}
```

การตอบสนองต่อ Effect

- ต่อไปเราต้องพิจารณาว่า สิ่งที่มีผลต่อการเกิด Effect หรือ Dependencies หรือกล่าวอีกอย่างคือเราจะเรียก callback ที่กำหนดให้แก่ useEffect() ขึ้นมาทำงานเมื่อใด นั่นเอง ซึ่งมีทางเลือกดังนี้
 - ◉ ถ้าต้องการให้เรียก callback เมื่อเกิด Effect ในทุกกรณี (เช่น แสดงคอมโพเนนต์, re-render, ติดต่อกับเซิร์ฟเวอร์, ฯลฯ) ก็ไม่ต้องระบุ dependencies เช่น

```
React.useEffect(() => {  
    //...  
})
```

- ◉ ถ้าต้องการให้เรียก callback เฉพาะ **ครั้งแรก** ที่แสดงคอมโพเนนต์เพียงครั้งเดียวเท่านั้น ให้กำหนด dependencies เป็นอาร์เรย์ว่าง เช่น

```
React.useEffect(() => {  
    //...  
}, [] )    //กำหนดอาร์เรย์ว่าง
```


การตอบสนองต่อ Effect

- ๑ ถ้าต้องการให้เรียก callback เมื่อค่าตัวแปร State ตัวใดตัวหนึ่งเปลี่ยนแปลงไป ก็ให้ระบุชื่อตัวแปรเหล่านั้นไว้ในอาร์เรย์ (กำหนดได้มากกว่า 1 ตัว) เช่น

```
let [a, setA] = React.useState()
let [b, setB] = React.useState()

React.useEffect(() => {
  //...
}, [a, b]) //ให้เรียก callback เมื่อ a หรือ b เปลี่ยนแปลง
```

- ฟังก์ชันที่เป็น callback จะถูกเรียกหลังจากที่แสดงผลคอมโพเนนต์รวมถึง re-render ไปแล้ว

การตอบสนองต่อ Effect

- การเกิด Effect กับ Class Component

ในกรณีของ Class Component จะไม่มีเมธอดสำหรับจัดการ Effect โดยตรง ก็ยังมีลักษณะที่พอจะเทียบเท่ากันได้ นั่นคือเมธอด `componentDidMount()` และ/หรือ `componentDidUpdate()` ดังหลักการต่อไปนี้

- หากเราจะทำสิ่งนั้น **เพียงครั้งเดียว** หลังจากที่แสดงผลคอมโพเนนต์ไปแล้ว ก็ให้โอเวอร์ไรด์หรือกำหนดการกระทำไว้ในเมธอด `componentDidMount()`

```
class MyComponent extends React.Component {  
    ...  
    render() {  
        return ...  
    }  
  
    componentDidMount() {  
        //กำหนดการกระทำหลังจากแสดงผลคอมโพเนนต์ครั้งแรก  
    }  
}
```

การตอบสนองต่อ Effect

- หากเราจะทำสิ่งนั้นเฉพาะเมื่อคอมโพเนนต์ re-render ไปแล้วเท่านั้น (มักเกิดจากค่าใน State เปลี่ยนแปลงไป) ก็ให้โอเวอร์ไรด์เมธอด `componentDidUpdate()`

```
class MyComponent extends React.Component {  
  ...  
  render() {  
    return ...  
  }  
  
  componentDidUpdate() {  
    //กำหนดการกระทำหลังจากคอมโพเนนต์ re-render  
  }  
}
```

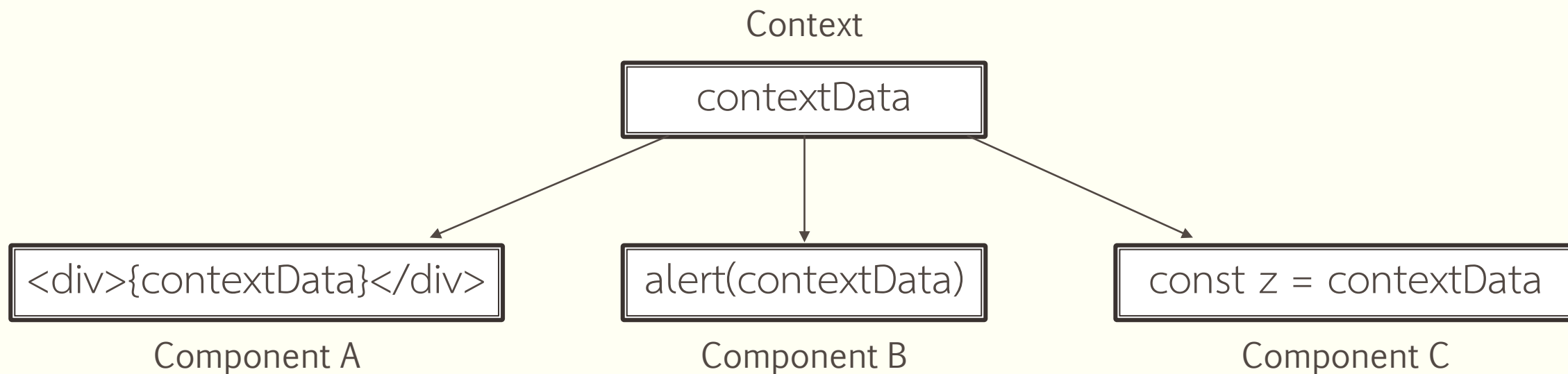
การตอบสนองต่อ Effect

- หากเราจะทำสิ่งนั้นทั้งกรณีที่แสดงผลคอมโพเนนต์ครั้งแรก และกรณีที่คอมโพเนนต์ re-render ก็ให้โอเวอร์ไรด์ร่วมกันทั้งเมธอด componentDidMount() และ componentDidUpdate()

```
class MyComponent extends React.Component {  
  ...  
  render() {  
    return ...  
  }  
  
  componentDidMount() {  
    //กำหนดการกระทำหลังจากแสดงผลคอมโพเนนต์ครั้งแรก  
  }  
  
  componentDidUpdate() {  
    //กำหนดการกระทำหลังจากคอมโพเนนต์ re-render  
  }  
}
```

การจัดเก็บข้อมูลส่วนกลางด้วย Context

การสร้างเว็บเพจในแบบของ React นั้น จะเป็นการนำคอมโพเนนต์ย่อยๆ ของแต่ละส่วนมารวมเข้าด้วยกัน จากเหตุผลดังกล่าว จึงอาจมีบางกรณีที่ต้องใช้ข้อมูลบางอย่างร่วมกันระหว่างคอมโพเนนต์ อย่างไรก็ตาม เราไม่สามารถอ้างถึงข้อมูลที่อยู่ต่างคอมโพเนนต์โดยตรงได้ แต่ต้องใช้วิธีการสร้างข้อมูลส่วนกลางที่เรียกว่า Context เพื่อให้คอมโพเนนต์ต่างๆ สามารถใช้งานร่วมกันได้ ดังหลักการในภาพถัดไป



การจัดเก็บข้อมูลส่วนกลางด้วย Context

- การสร้างและกำหนด Provider สำหรับ Context

เนื่องจากข้อมูลแบบ Context จะต้องใช้งานร่วมกันระหว่างคอมโพเนนต์ ดังนั้น เราควรแยกมาสร้างเป็นไฟล์ไว้ต่างหากในแบบโมดูล แล้วค่อยนำเข้าไปในคอมโพเนนต์ที่ต้องการใช้งาน และการกำหนดค่าให้กับ Context นั้น ต้องดำเนินการผ่าน Provider ดังแนวทางต่อไปนี้

- สร้างไฟล์เพิ่มเข้าไปในแอป เช่น ในที่นี้กำหนดชื่อเป็น context.js โดยตัวแปรที่จะเก็บข้อมูล Context นั้น ต้องกำหนดค่าด้วยฟังก์ชัน createContext() ซึ่งจะนำเข้าโดยตรง หรือจะเรียกผ่านคอมโพเนนต์ React ก็ได้ เช่น

```
react/app1/src/context.js
```

```
import React, { createContext } from 'react'
```

```
export const userContext = createContext() //หรือ React.createContext()
```

```
/* เนื่องจากเป็นตัวแปร ดังนั้น จึงส่งออกแบบ default ไม่ได้ */
```

การจัดเก็บข้อมูลส่วนกลางด้วย Context

- ตัวแปรหรือคอมโพเนนต์แบบ Context ที่เราสร้างขึ้นจะมีพร็อพเพอร์ตี้ Provider สำหรับใช้ในการจัดหาหรือกำหนดค่าให้กับ Context ซึ่งโดยส่วนใหญ่ เรามักจะวาง Provider ไว้ในไฟล์ App.js โดยนำเข้า Context จากไฟล์ที่สร้างเอาไว้ แล้วห่อหุ้มคอมโพเนนต์ทั้งหมดที่จะนำค่าจาก Context ไปใช้งานด้วย Provider แต่ถึงคอมโพเนนต์นั้นจะไม่ใช้ค่าจาก Context ก็ห่อหุ้มด้วย Provider ได้ เช่น สมมติว่าภายในแอปเรามี 2 คอมโพเนนต์คือ Header และ Content เราก็ห่อหุ้มด้วย Provider พร้อมกำหนดค่าผ่านพร็อพเพอร์ตี้ value ดังนี้

react/app1/src/App.js

```
import React from 'react'
import { useContext } from 'react' //ต้องนำเข้า Context
import Header from './context-header'
import Content from './context-content'
```

```
export default function App(){
  return (
    <userContext.Provider value={'Tom Jerry'}>
      <Header/>
      <Content/>
    </userContext.Provider>
  )
}
```

จากโค้ด Tom Jerry คือค่าที่กำหนดให้แก่ Context ซึ่งจะนำไปใช้ในคอมโพเนนต์ต่างๆ

การจัดเก็บข้อมูลส่วนกลางด้วย Context

- การใช้ข้อมูล Context ใน Class Component

คอมโพเนนต์ที่จะนำค่าจาก Context มาใช้งานได้ ต้องห่อหุ้มด้วย Provider เอาไว้แล้ว (ดังในไฟล์ App.js) ซึ่งหากเป็น Class Component การเข้าถึงค่าใน Context ก็มีวิธีการดังนี้

- นำเข้าตัวแปรหรือคอมโพเนนต์ของ Context ที่ได้สร้างเอาไว้แล้ว
- ต้องนำตัวแปร Context มากำหนดให้แก่พร็อพเพอร์ตี้ **contextType** ในแบบ static ซึ่งพร็อพเพอร์ตี้นี้เป็นของ React Component อยู่แล้ว เช่น

```
import { useContext } from './context'

export default class Header extends React.Component {
  static contextType = useContext
  ...
}
```

- ต่อไป การอ้างถึงค่าจาก Context ให้อ่านจากพร็อพเพอร์ตี้ที่ชื่อ **context** เช่น this.context ดังนั้นลักษณะโดยรวมของ Class Component กรณีที่ใช้ค่าจาก Context จะเป็นดังนี้

การจัดเก็บข้อมูลส่วนกลางด้วย Context

react/app1/src/context-header.js

```
import React from 'react'

import { useContext } from './context'

export default class Header extends React.Component {

  static contextType = useContext

  render() {

    const user = this.context

    const headerStyle = {
      backgroundColor: '#cee',
      textAlign: 'center',
      padding: 5
    }
  }
}
```

```
return (
  <div style={headerStyle}>
    <a href=" ">Home</a>&nbsp;-&nbsp;
    <a href=" ">Product</a>&nbsp;-&nbsp;
    <a href=" ">Contact Us</a>&nbsp;-&nbsp;&nbsp;
    [{user}&nbsp;:&nbsp;<a href=" ">Signout</a>]
  </div>
)
```

การจัดเก็บข้อมูลส่วนกลางด้วย Context

react/app1/src/context.js

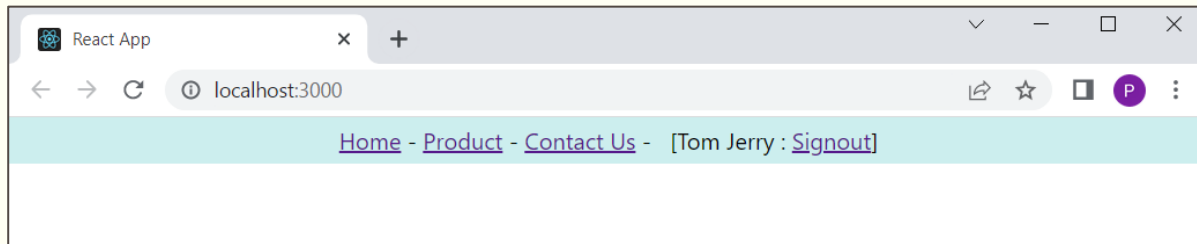
```
import React, {createContext} from 'react'

export const userContext = createContext()
```

react/app1/src/App.js

```
import React from 'react'
import { userContext } from './context'
import Header from './context-header'

export default function App() {
  return (
    <userContext.Provider value={'Tom Jerry'}>
      <Header/>
    </userContext.Provider>
  )
}
```



การจัดเก็บข้อมูลส่วนกลางด้วย Context

- การใช้ข้อมูล Context ใน Function Component

การนำค่าจาก Context มาใช้ใน Function Component สามารถทำได้ในแบบง่ายๆ เพราะมีฟังก์ชันในกลุ่ม React Hook ให้ใช้งานโดยตรงอยู่แล้ว นั่นคือ useContext() ดังแนวทางต่อไปนี้

react/app1/src/context-content.js

```
import React from 'react'
import { useContext } from './context'

export default function Content() {
  let user = React.useContext(userContext)

  const contentStyle = {
    backgroundColor: '#ddd',
    textAlign: 'center',
```

```
    margin: 10,
    padding: 10
  }

  return (
    <div style={contentStyle}>
      Hello {user}
    </div>
  )
}
```

การจัดเก็บข้อมูลส่วนกลางด้วย Context

react/app1/src/context.js

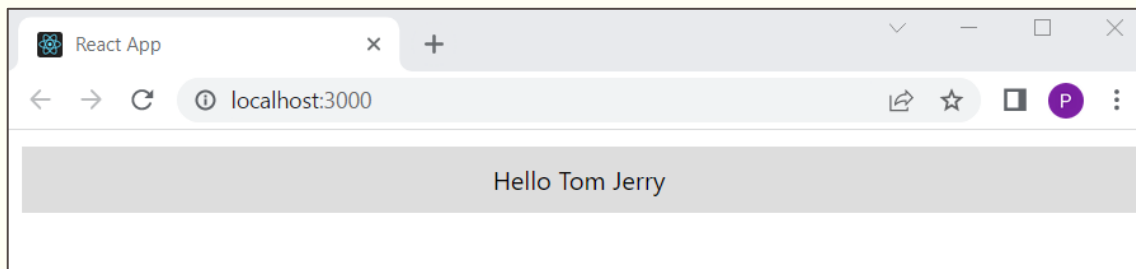
```
import React, {createContext} from 'react'

export const userContext = createContext()
```

react/app1/src/App.js

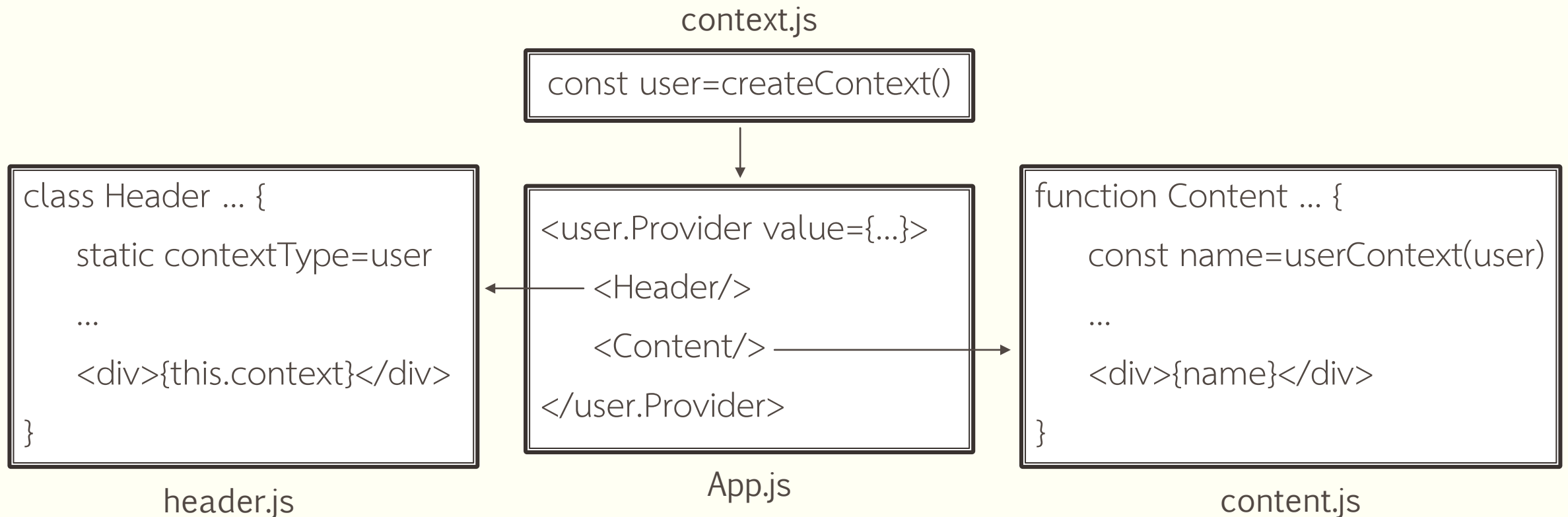
```
import React from 'react'
import { userContext } from './context'
import Content from './context-content'

export default function App() {
  return (
    <userContext.Provider value={'Tom Jerry'}>
      <Content/>
    </userContext.Provider>
  )
}
```



การจัดเก็บข้อมูลส่วนกลางด้วย Context

จากขั้นตอนสร้างทั้งหมดที่กล่าวมา นับตั้งแต่การสร้างตัวแปรแบบ Context การห่อหุ้มด้วย Provider รวมถึงการนำมาใช้งานทั้งใน Class และ Function Component สามารถสรุปได้ดังแผนภาพด้านล่าง ทั้งนี้หากกำหนดโค้ดดังที่ผ่านมา เมื่อรวมทั้งหมดเข้าด้วยกันแล้วทำการทดสอบก็จะได้ผลดังภาพต่อไปนี้

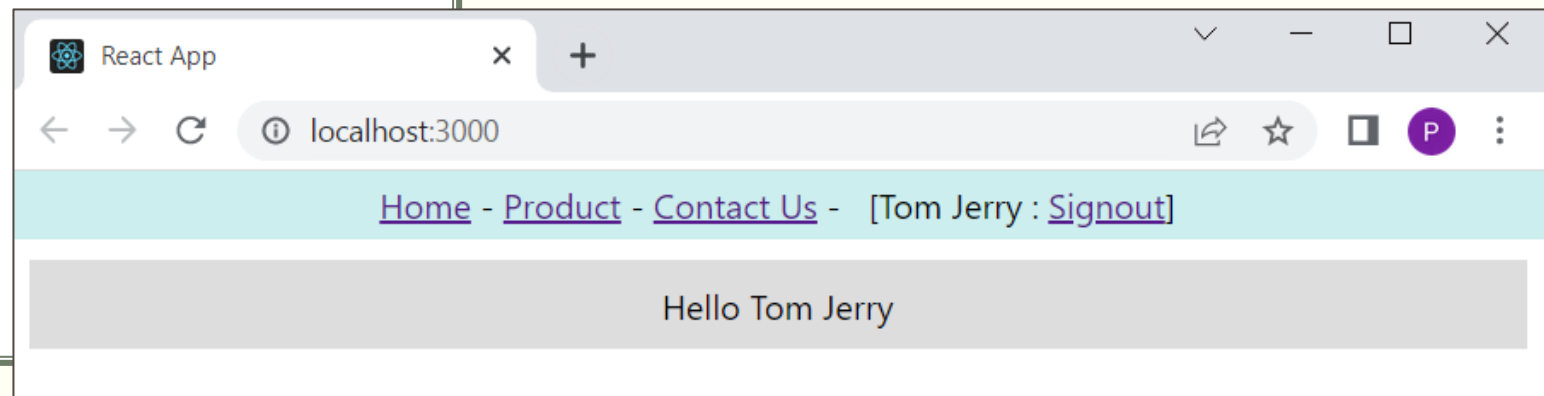


การจัดเก็บข้อมูลส่วนกลางด้วย Context

react/app1/src/App.js

```
import React from 'react'
import { useContext } from './context'
import Header from './context-header'
import Content from './context-content'

export default function App() {
  return (
    <userContext.Provider value={'Tom Jerry'}>
      <Header/>
      <Content/>
    </userContext.Provider>
  )
}
```



การจัดเก็บข้อมูลส่วนกลางด้วย Context

- การจัดเก็บข้อมูลแบบ State ไว้ใน Context

การใช้ข้อมูลจาก Context เรากำหนดค่าของมันแบบตายตัวไว้ที่ Provider จึงไม่สามารถอัปเดตข้อมูลดังกล่าวในภายหลังให้สอดคล้องกันทุกคอมโพเนนต์ได้ ซึ่งเราอาจแก้ปัญหานี้โดยการจัดเก็บข้อมูลแบบ State ลงใน Context แทนรูปแบบเดิม แล้วต่อไปเมื่อเราอัปเดตข้อมูล Context ที่คอมโพเนนต์ใด ก็จะส่งผลไปถึงคอมโพเนนต์อื่นๆ ทันที ซึ่งมีหลักการดังต่อไปนี้

- ที่ไฟล์การสร้าง Context ก็กำหนดเหมือนเดิม เช่น

```
react/app1/src/context.js
```

```
import React, { createContext } from 'react'
```

```
export const userContext=createContext() //หรือ React.createContext()
```

การจัดเก็บข้อมูลส่วนกลางด้วย Context

- ที่ไฟล์ App.js ซึ่งโดยปกติจะอยู่ในรูปแบบ Function Component ดังนั้น เราก็สร้าง State ตามหลักการของเดิม เช่น `let [user, setUser]=React.useState()`
- สร้าง Provider แล้วนำค่าจาก State มากำหนดให้แก่พร็อพเพอร์ตี้ value ในแบบอาร์เรย์ทั้งค่าของมันและฟังก์ชันสำหรับเปลี่ยนค่า เช่น

react/app1/src/App.js

```
import React from 'react'
import { useContext } from './context'
import Header2 from './context-header2'
import Content2 from './context-content2'
```

```
export default function App() {
  let [user, setUser] = React.useState("")
  return (
    <userContext.Provider value={[user, setUser]}>
      <Header2/>
      <Content2/>
    </userContext.Provider>
  )
}
```


การจัดเก็บข้อมูลส่วนกลางด้วย Context

- ที่คอมโพเนนต์ซึ่งเราจะนำค่าจาก Context ไปใช้งาน จะต้องอ่านค่าของ Context ในแบบ Array Destructuring เพื่อให้ได้ทั้งตัวแปรและฟังก์ชันในการเปลี่ยนค่าของ State จากนั้น เราก็นำจากค่าตัวแปรไปใช้งานในคอมโพเนนต์ตามปกติ และถ้าจะอัปเดตก็ทำผ่านฟังก์ชันของ State ดังแนวทางต่อไปนี้

react/app1/src/context-header2.js

```
import React from 'react'
import {userContext} from './context'

export default class Header2 extends
React.Component {
  static contextType = userContext

  render() {
    let [user, setUser] = this.context
```

```
const headerStyle = {
  backgroundColor:'#cee',
  textAlign:'center',
  padding: 5
}

const onClickSignout = (event) => {
  event.preventDefault()
  setUser('')
}
```

การจัดเก็บข้อมูลส่วนกลางด้วย Context

react/app1/src/context-header2.js (ต่อ)

[illegible]

```
{
  (user)
  ? <span>[{user}]&nbsp;  <a href=" "
      onClick={onClickSignout}>Signout</a>]</span>
  : <span>[<a href=" " onClick={onClickSignin}>
      Signin</a>]</span>
}
</div>
)
}
```

การจัดเก็บข้อมูลส่วนกลางด้วย Context

react/app1/src/context-content2.js (ต่อ)

```
import React from 'react'
import { useContext } from './context'

export default function Content2() {
  let [user, setUser] = useContext(userContext)

  const contentStyle = {
    backgroundColor: '#ddd',
    textAlign: 'center',
    margin: 10,
    padding: 10
  }
```

```
const onClickSignin = (event) => {
  event.preventDefault()
  setUser('Tom Jerry')
}

return (
  <div style={contentStyle}>
    {
      (user)
      ? <span>Hello {user}</span>
      : <span>Please <a href=" " onClick={onClickSignin}>
        Signin</a></span>
    }
  </div>
)
```

การจัดเก็บข้อมูลส่วนกลางด้วย Context

