Contents

1 Part 1

1.1 Component state, event handlers

Kita mulai dengan contoh aplikasi baru:

```
const Hello = (props) => {
  return (
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>
    </div>
  )
}

const App = () => {
  const name = 'Peter'
  const age = 10

  return (
    <div>
      <h1>Greetings</h1>
      <Hello name="Maya" age={26 + 10} />
      <Hello name={name} age={age} />
    </div>
  )
}
```

1.1.1 Component Helper Function

Sekarang kita kembangkan aplikasi lita sehingga bisa menerka tahun kelahiran.

```
const Hello = (props) => {

  const bornYear = () => {
    const yearNow = new Date().getFullYear()
```

```
    return yearNow - props.age
  }

  return (
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>

      <p>So you were probably born in {bornYear()}</p>
    </div>
  )
}
```

Untuk logic perhitungan tahun kelahiran diletakkan pada function terpisah. Dimana function ini akan dipanggil ketika komponen di-render.

1.1.2 Destructuring

Destructuring adalah fitur dari JavaScript yang digunakan untuk memecah nilai (*destructuring*) dari object atau array selama *assignment*.

Pada proses sebelumnya kita melewatkan `props` sebagai parameter, yang mana props sendiri adalah object. Dengan kemampuan *destructuring* dari JS kita memecah object tersebut secara langsung.

Fitur ini mempermudah proses assignment variabel, sehingga kita bisa mengekstrak dan mengumpulkannya dalam sebuah object properties ke dalam variabel terpisah.

```
const Hello = (props) => {

  const bornYear = () => {
    const yearNow = new Date().getFullYear()
    return yearNow - props.age
  }

  return (
    <div>
      <p>
        Hello {props.name}, you are {props.age} years old
      </p>

      <p>So you were probably born in {bornYear()}</p>
    </div>
  )
}
```

atau kita secara langsung menggunakan *destructuring* pada parameter.

```
const Hello = ({ name, age }) => {
  const bornYear = () => new Date().getFullYear() - age
```

```
  return (
    <div>
      <p>
        Hello {name}, you are {age} years old
      </p>
      <p>So you were probably born in {bornYear()}</p>
    </div>
  )
}
```

dengan anggapan bahwa object tersebut memiliki nilai seperti berikut:

```
props = {
  name: 'Arto Hellas',
  age: 35,
}
```

props yang dilewatkan ke komponen sekarang secara langsung telah dipecah / *destructure* ke dalam variabel `name` dan `age`.

1.1.3 Page re-rendering

Selama ini aplikasi kita hanya memiliki tampilan yang sama setelah rendering awal. Bagaimana jika kita membuat penghitung (*counter*) dimana nilai bertambah dikarenakan fungsi waktu atau dikarenakan click pada tombol.

Mari kita ubah file `App.jsx` menjadi seperti ini:

```
const App = (props) => {
  const {counter} = props
  return (
    <div>{counter}</div>
  )
}

export default App
```

dan file `main.jsx` menjadi:

```
import ReactDOM from 'react-dom/client'

import App from './App'

let counter = 1

ReactDOM.createRoot(document.getElementById('root')).render(
  <App counter={counter} />
)
```

Komponen App diberikan nilai dari counter melalui `counter` props. Komponen ini kemudian merender nilai tersebut ke layar. Apa yang terjadi ketika nilai dari `counter` berubah? Bahkan jika kita melakukan panambahan seperti berikut

```
counter += 1
```

Komponen tersebut tidak akan direnderer ulang. Kita bisa mendapatkan render ulang dengan memanggil method `render` dua kali, misal dengan cara berikut:

```
let counter = 1

const root = ReactDOM.createRoot(document.getElementById('root'))

const refresh = () => {
  root.render(
    <App counter={counter} />
  )
}

refresh()
counter += 1
refresh()
counter += 1
refresh()
```

Perintah rerendering dibungkus di dalam function `resfresh`. Dan sekarang komponen akan merender 3x, pertama, kedua kemudian ketiga. Namun perubahan tersebut tidak bisa kita cermati.

Kita bisa mengimplementasikan setInterval untuk melakukan re-rendering dan increment.

```
setInterval(() => {
  refresh()
  counter += 1
}, 1000)
```

Namun membuat pemanggilan berulang [ada method `render` tidak direkomendasikan sebagai cara untuk merender ulang komponen.

1.1.4 Stateful Component

Semua komponen aplikasi kita selama ini tidak memiliki *state* yang bisa berubah selama alur hidup komponen.

Selanjutnya kita tambahkan state ke komponen App dengan bantuan dari state hook React.

Kali ini kita kembalikan *main.jsx* seperti semula.

```
import ReactDOM from 'react-dom/client'

import App from './App'

ReactDOM.createRoot(document.getElementById('root')).render(<App />)
```

dan ubah App.jsx seperti berikut:

```
import { useState } from 'react'

const App = () => {
```

```
  const [ counter, setCounter ] = useState(0)


  setTimeout(
    () => setCounter(counter + 1),
    1000
  )

  return (
    <div>{counter}</div>
  )
}

export default App
```

Pada baris pertama, import function `useState`:

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

selanjutnya function tersebut dipanggil pada *function body*

```
const [ counter, setCounter ] = useState(0)
```

Pada saat function dipanggil ditampbah *state* ke komponen dan merendernya dengan inisialisasi nilai `0`. Function tersebut mengembalikan sebuah array yang berisi dua item. Dengan cara desctructuring kemudian item tersebut kita pecah menjadi dua yakni variable `counter` dan `setCounter`.

Variable `counter` berisi nilai awal `0` saat inisialisasi awal. Sedangkan `setCounter` merupakan sebuah function yang digunakan untuk merubah nilai *state* yakni `counter`.

Applikasi ini memanggil *setTimeout* dan melewatkan dua parameter: sebuah function untuk menambah *counter state* dan nilai batas waktu sebesar 1 detik.

```
setTimeout(
  () => setCounter(counter + 1),
  1000
)
```

Function yang dilewatkan sebagai parameter pada function `setTimeout` akan dipanggil nanti setelah 1 detik berlalu.

```
() => setCounter(counter + 1)
```

Dan ketika function `setCounter` yang digunakan untuk merubah *state* dipanggil. React kemudian akan merender ulang / *re-render* komponen tersebut, yakni seluruh *function body* dari komponen tersebut akan dieksekusi kembali.

```
() => {
  const [ counter, setCounter ] = useState(0)

  setTimeout(
    () => setCounter(counter + 1),
    1000
```

```
  )

  return (
    <div>{counter}</div>
  )
}
```

Pada kedua kalinya function komponen dieksekusi ia memanggil function `useState` dan mengembalikan berupa nilai state yang baru: `1`. Mengeksekusi *function body* lagi juga akan memanggil function `setTimeout` lagi, yang mana akan dieksekusi setelah 1 detik dan menambahkan state `counter` lagi. Karena nilai `counter` saat ini bernilai 1 maka akan ditambahkan 1 lagi da counter saat ini bernilai 2. Dan secara otomatis body function akan direnderer ulang dengan nilai counter = 2.

Tiap kali `setCounter` merubah state, hal ini akan menyebabkan komponen akan direnderer ulang.

1.1.5 Event handling

Pada aplikasi sebelumnya kita sudah menggunakan *event handlers* yang terdaftar untuk dipanggil ketika kurun waktu tertentu. Interaksi pengguna dengan elemen-elemen yang berbeda dapat menyebabkan terpicunya suatu event dari kumpulan bermacam event.

Kita ubah kode kita sehingga menambah nilai *counter* terjadi ketika user menekan tombol, yang akan diimplementasikan dengan elemen *button*.

Elemen *button* mendukung yang dinamakan *mouse event*, salah satu event yang digunakan adalah even *click*. Meskipun event ini bisa dipicu menggunakan touch screen atau tombol keyboard namun tetap saja dinamakan mouse event.

Untuk mendaftarkan sebuah function event handler ke event click bisa dibuat dengan cara berikut:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)


  const handleClick = () => {
    console.log('clicked')
  }

  return (
    <div>
      <div>{counter}</div>

      <button onClick={handleClick}>
        plus
      </button>
    </div>
  )
}
```

Kita set nilai atribut *onClick* yang merujuk ke fungsi `handleClick`.

Sekarang tiap kali kita klik tombol *plus* akan menyebabkan fungsi `handleClick` dipanggil.

Fungsi event handler juga bisa didefinisikan secara langsung pada assignment nilai pada atribute *onClick*.

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  return (
    <div>
      <div>{counter}</div>

      <button onClick={() => console.log('clicked')}>
        plus
      </button>
    </div>
  )
}
```

Dengan merubah event handler ke bentuk berikut:

```
<button onClick={() => setCounter(counter + 1)}>
  plus
</button>
```

Dengan ini maka kita mendapatkan hasil yang sesuai harapan. Ketika tombol plus ditekan maka secara keseluruhan body fungsi App akan dieksekusi ulang dengan memberikan nilai useState dengan nilai counter saat pemanggilan di atas. Kemudian *re-render* komponen-komponen.

Lanjut ke tombol reset pada counter. Yup, kita akan menambahkan fungsi reset.

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  return (
    <div>
      <div>{counter}</div>
      <button onClick={() => setCounter(counter + 1)}>
        plus
      </button>

      <button onClick={() => setCounter(0)}>
        zero
      </button>
    </div>
  )
}
```

1.1.6 Sebuah Event Handler adalah sebuah Function

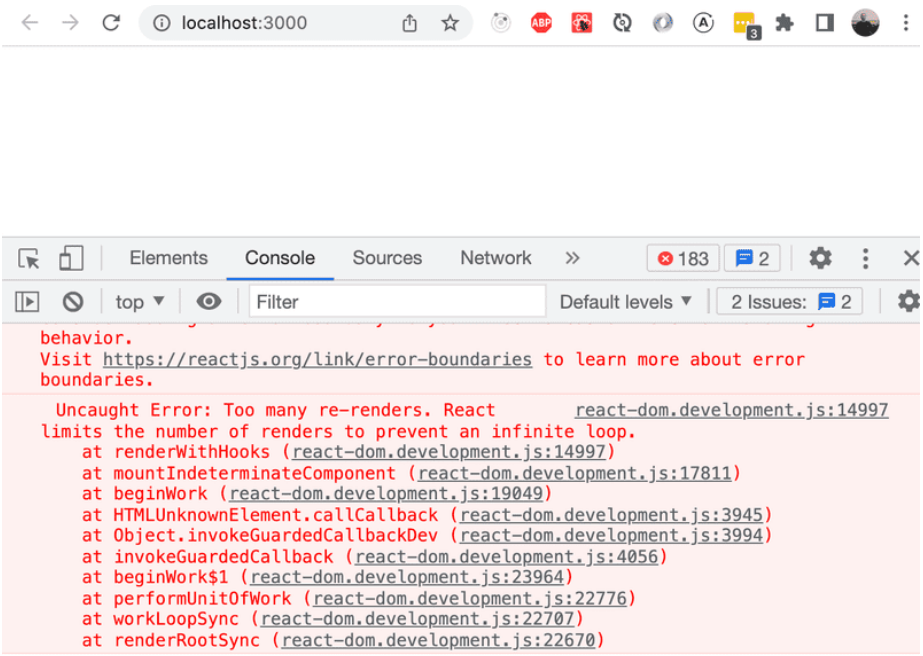
Kita sudah mendefinisikan event handler untuk tombol dimana kita juga mendeklarasikan atribut *onClick*:

```
<button onClick={() => setCounter(counter + 1)}>
  plus
</button>
```

Lalu bagaimana jika kita mencoba mendefinisikan handler dengan bentuk yang lebih sederhana / secara langsung.

```
<button onClick={setCounter(counter + 1)}>
  plus
</button>
```

Dan ini akan menjadikan aplikasi kita error:



Event handler seharusnya adalah sebuah function atau *function reference*, dan pada kode di atas dinmakan function call / invoke, pemanggilan function.

Sehingga ketika kita render aplikasi tersebut secara otomatis `setCounter` akan dipanggil yang menyebabkan perenderan ulang, dan begitu seterusnya.

Jadi, kita ubah kode kita seperti sebelum error:

```
<button onClick={() => setCounter(counter + 1)}>
  plus
</button>
```

Sekarang `setCounter` hanya akan dieksekusi ketika tombol diklik.

Mendefinisikan event handler di dalam *JSX-template* tidak direkomendasikan. Karena event handler yang kita punya sangat simple, it's ok. Namun jika event

handler yang kita punya cukup kompleks maka lebih baik kita pisahkan dalam function tersendiri.

Jadi mari kita pisahkan event handler pada kode:

```
const App = () => {
  const [ counter, setCounter ] = useState(0)


  const increaseByOne = () => setCounter(counter + 1)

  const setToZero = () => setCounter(0)

  return (
    <div>
      <div>{counter}</div>

      <button onClick={increaseByOne}>
        plus
      </button>

      <button onClick={setToZero}>
        zero
      </button>
    </div>
  )
}
```

Jadi pada kode di atas, nilai dari atribut *onClick* berisi variabel yang berisi reference ke function.

1.1.7 Melewatkan state ke komponen turunannya

React merekomendasikan untuk menulis react component yang kecil dan *resuable* antar aplikasi atau bahkan antar project. Selanjutnya kita akan *refactor* aplikasi kita, sehingga tersusun menjadi 3 komponen yang lebih kecil. 1 komponen untuk menampilkan hasil perhitungan, dan dua komponen untuk tombol.

Pertama-tama kita implementasikan komponen *Display* yang bertanggungjawab untuk menampilkan nilai dari *counter*.

Cara yang terbaik di React adalah dengan melakukan lift the state up pada hirarki komponen.

> *Often, several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor.*

Intinya ketika ada beberapa komponen yang saling terkait / punya perubahan data yang saling terkait. Maka hal direkomendasikan adalah dengan meletakkan state pada parent terdekat yang bisa menghubungkan komponen-komponen yang bersangkutan.

Ok, lanjut meletakkan *state* aplikasi pada komponen *App* dan meneruskan ke komponen *Display* melalui *props*.

```
const Display = (props) => {
  return (
    <div>{props.counter}</div>
  )
}
```

Dan kita tinggal meneruskan *state* `counter` ke Display.

```
const App = () => {
  const [ counter, setCounter ] = useState(0)

  const increaseByOne = () => setCounter(counter + 1)
  const setToZero = () => setCounter(0)

  return (
    <div>

      <Display counter={counter}/>
      <button onClick={increaseByOne}>
        plus
      </button>
      <button onClick={setToZero}>
        zero
      </button>
    </div>
  )
}
```

1.2 More Complex State, debugging React Apps

1.2.1 Complex State

Pada contoh sebelumnya hanya memiliki nilai state yang sederhana. Bagaimana jika aplikasi yang kita bangun membutuhkan nilai *state* yang lebih kompleks.

Pada kebanyakan kasus, cara paling mudah adalah dengan menggunakan function `useState` berkali-kali untuk membuat beberapa state.

Pada kode kita membuat dua buah *state* dengan nama `left` dan `right` yang keduanya memiliki initial value `0`.

```
const App = () => {
  const [left, setLeft] = useState(0)
  const [right, setRight] = useState(0)

  return (
    <div>
      {left}
      <button onClick={() => setLeft(left + 1)}>
        left
```

```
      </button>
      <button onClick={() => setRight(right + 1)}>
        right
      </button>
      {right}
    </div>
  )
}
```

Komponen dapat mengaksesnya dengan menggunakan function `setLeft` dan `setRight` yang digunakan untuk mengubah kedua *state* tersebut.

State bisa berupa tipe apapun, kita juga bisa mengimplementasikan fungsionalitas yang sama dengan menyimpan jumlah klik tombol *left* dan *right* ke dalam object yang sama.

```
{
  left: 0,
  right: 0
}
```

Pada kasus ini, aplikasi akan terlihat seperti ini:

```
const App = () => {
  const [clicks, setClicks] = useState({
    left: 0, right: 0
  })

  const handleLeftClick = () => {
    const newClicks = {
      left: clicks.left + 1,
      right: clicks.right
    }
    setClicks(newClicks)
  }

  const handleRightClick = () => {
    const newClicks = {
      left: clicks.left,
      right: clicks.right + 1
    }
    setClicks(newClicks)
  }

  return (
    <div>
      {clicks.left}
      <button onClick={handleLeftClick}>left</button>
      <button onClick={handleRightClick}>right</button>
      {clicks.right}
    </div>
  )
```

```
}
```

Nah, saat ini komponen tersebut hanya punya sebuah *state* dan *event handler* yang bertanggung jawa untuk merubah keseluruhan *state* aplikasi.

```
const handleLeftClick = () => {
  const newClicks = {
    left: clicks.left + 1,
    right: clicks.right
  }
  setClicks(newClicks)
}
```

Object berikut menempatkan *state* baru dari aplikasi.

```
{
  left: clicks.left + 1,
  right: clicks.right
}
```

Kita juga bisa mendefiniskan *state* baru dengan cara yang lebih rapi menggunakan *object spread*.

```
const handleLeftClick = () => {
  const newClicks = {
    ...clicks,
    left: clicks.left + 1
  }
  setClicks(newClicks)
}

const handleRightClick = () => {
  const newClicks = {
    ...clicks,
    right: clicks.right + 1
  }
  setClicks(newClicks)
}
```

Pada penerapan `{ ...clicks }` menciptakan sebuah object baru yang memiliki semua salinan dari properti object `clicks`. Jika kita menentukan misal saja *right* pada `{ ...clicks, right: 1 }`, nilai dari `right` pada object baru akan menjadi `1`.

Pada contoh di atas:

```
{ ...clicks, right: clicks.right + 1 }
```

kode tersebut akan menciptakan salinan dari `clicks` dimana nilai dari properti `right` telah ditambah dengan 1.

Menentukan object ke variable pada *event handler* sebetulnya tidak dibutuhkan dan kita dapat menyederhanakan fungsi tersebut menjadi bentuk berikut:

```
const handleLeftClick = () =>
  setClicks({ ...clicks, left: clicks.left + 1 })
```

```
const handleRightClick = () =>
  setClicks({ ...clicks, right: clicks.right + 1 })
```

Mungkin kita kita juga berpikir mengapa kita tidak mengubah langsung saja *state*-nya seperti ini:

```
const handleLeftClick = () => {
  clicks.left++
  setClicks(clicks)
}
```

Aplikasi nampak bisa berjalan. Bagaimanapun juga, hal ini dilarang di React untuk mengubah *state* secara langsung, hal ini dikarenakan bisa menghasilkan efek samping yang tidak diinginkan. Mngubah *state* harus selalu dilakukan dengan men-set *state* ke object baru.