React: (Side) Effects

Phase 2 | Week 1, Lesson 4

Sakib Rasul | Updated February 1, 2024. Created June 15, 2023.

Today's Objectives

Today, we'll answer the following questions:

1. *What's a **pure function**?*
2. *How (and why) should we write **pure components** in React?*
3. *What's a **(side) effect** in React?*
4. *How do we cause **(side) effects** in React?*

Pure Functions

- A **pure function** is one that, given the same input, returns the same output, each and every time.
  - Take `function double(x) { return x * 2 }`.
    - If we pass in `x = 2`, `double(x)` will always be 4.
    - If we pass in `x = 32`, `double(x)` will always be 64.
    - …and so on! Any given `x` will always output the same number.

Pure Components in React

- In React, a **pure component** is one that, given the same props and state, renders the same JSX.
- React assumes that all components are pure, meaning that, if one:
  - has no props or state, should always render the same JSX.
  - has any props, should act as though those props are immutable.
  - has state, should modify that state through state setters.
- Why does React care about purity? 🤔

Pure Components in React

- In React, a **pure component** is one that, given the same props and state, renders the same JSX.
- React assumes that all components are pure, meaning that, if one:
  - has no props or state, should always render the same JSX.
  - has any props, should act as though those props are immutable.
  - has state, should modify that state through state setters.
- Why does React care about purity? 🤔
  - It's how React knows *what* to re-render, *when*!

An Impure Component (don’t do this!)

```
let capacity = 0;

function Venue() {
  capacity = capacity + 100;
  return <h1>This venue holds {capacity} people.</h1>
}

function Venues() {
  return (<>
    <Venue/>
    <Venue/>
  </>)
}
```

An Impure Component (don’t do this!)

```
let capacity = 0;

function Venue() {
  {/* <Venue> is trying to modify the preexisting variable capacity! */}
  capacity = capacity + 100;
  return <h1>This venue holds {capacity} people.</h1>
}

function Venues() {
  return (<>
    <Venue/> {/* This venue holds 200 people. */}
    <Venue/> {/* This venue holds 400 people. */}
  </>)
}
```

An Impure Component (don’t do this either!)

```
function Venue({ capacity }) {
  capacity = capacity + 100;
  return <h1>This venue holds {capacity} people.</h1>
}

function Venues() {
  const capacity = 0;
  return (
    <Venue capacity={capacity} />
    <Venue capacity={capacity} />
  )
}
```

An Impure Component (don’t do this either!)

```
function Venue({ capacity }) {
  {/* Venue is trying to modify the preexisting variable capacity! */}
  capacity = capacity + 100;
  return <h1>This venue holds {capacity} people.</h1>
}

function Venues() {
  const capacity = 0;
  return (
    <Venue capacity={capacity} /> {/* This venue holds 100 people. */}
    <Venue capacity={capacity} /> {/* This venue holds 100 people. */}
  )
}
```

An Pure Component

```
const Venue = ({ capacity, setCapacity }) =>
  <h1 onClick={() => setCapacity(capacity + 100)}>
    This venue holds {capacity} people.
  </h1>

const Venues = () => {
  const [ capacity, setCapacity ] = useState(0);
  return <Venue capacity={capacity}
  setCapacity={setCapacity} />
}
```

An Pure Component (modifying state)

```
const Venue = ({ capacity, setCapacity }) =>
  {/* <Venue> modifies its prop indirectly with setCapacity */}
  <h1 onClick={() => setCapacity(capacity + 100)}>
    This venue holds {capacity} people.
  </h1>

const Venues = () => {
  const [ capacity, setCapacity ] = useState(0);
  return <Venue capacity={capacity} setCapacity={setCapacity} />
}
```

A Pure Component

```
function Venue({ capacity }) {
  return <h1>This venue holds {capacity} people.</h1>
}

function Venues() {
  return (
    <Venue capacity={100} />
    <Venue capacity={200} />
  )
}
```

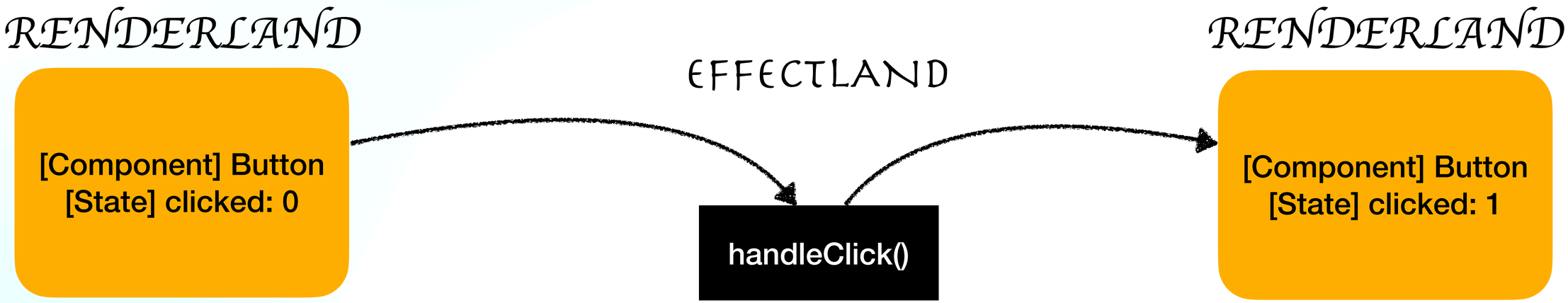
A Pure Component (modifying nothing!)

```
function Venue({ capacity }) {
  return <h1>This venue holds {capacity} people.</h1>
}

function Venues() {
  return (
    <Venue capacity={100} /> {/* This venue holds 100 people. */}
    <Venue capacity={200} /> {/* This venue holds 200 people. */}
  )
}
```

Side Effects in React

- In React, a **side effect** happens outs*ide* of rendering.
- Since side effects don't run during a render, they can be impure.
- The most common side effect in React is setting state.
- The most common place to write side effects is in event handlers.



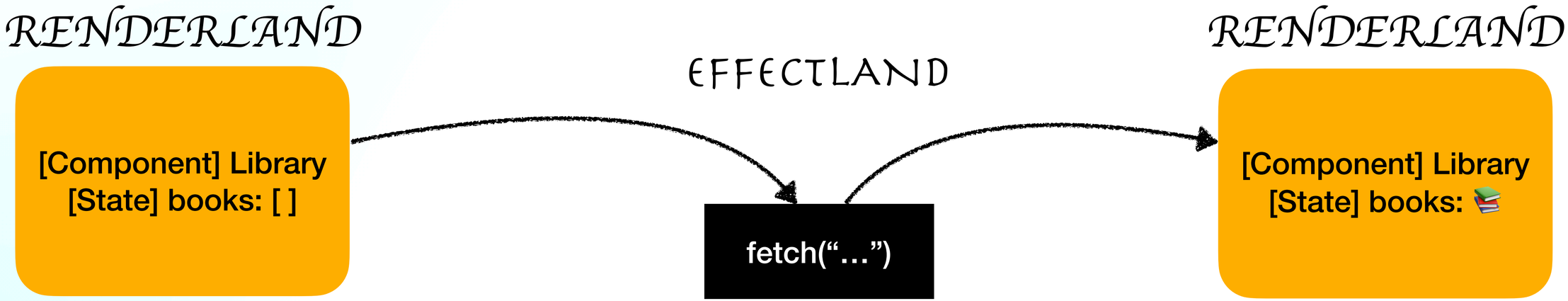
- …but not all side effects are triggered by events 🤔

(Side) Effects in React

- A small minority of side effects are triggered by rendering itself.
- In React, these side effects are known simply as **effects**.
- i.e., a change in state triggered by a render is a most common effect.
- When might we want a render to trigger a change in state? 🤔

(Side) Effects in React

- A small minority of side effects are triggered by rendering itself.
- In React, these side effects are known simply as **effects**.
- i.e., a change in state triggered by a render is a most common effect.
- When might we want a render to trigger a change in state? 🤔
  - A `GET` request triggered by a component's first render!



Writing an Effect in React

- To write an effect, follow these steps:
  1. `import { useEffect } from 'react'`.
  2. Invoke `useEffect()` with either one or two parameters.

Writing an Effect in React

- To write an effect, follow these steps:
  1. `import { useEffect } from 'react'`.
  2. Invoke `useEffect()` with either one or two parameters.
     A. The first parameter should be a function that runs effects.
     B. The second parameter, if any, should be an array of the effect's dependencies.

Writing an Effect in React

```
import { useEffect } from ‘react’;

export default function Component() {

  useEffect(() => {
    /* Write your effect(s) here */
  }), dependencies);

  return <></>;
}
```

The Dependency Array

- An effect's `dependencies` dictate when that effect will run.
- More specifically, the dependency array lets us choose which of a component's renders will trigger the execution of an effect.
- If we omit the dependency array, our effect runs after every re-render.
- If we pass an empty array `[]`, our effect runs only **on mount**.
- If we pass a non-empty array `[a, b]`, our effect runs on mount and whenever `a` or `b` change.
- Can we make any variable a dependency? 🤔

The Dependency Array

- If we omit the dependency array, our effect runs after every re-render.
- If we pass an empty array `[]`, our effect runs only **on mount**.
- If we pass a non-empty array `[a, b]`, our effect runs on mount and whenever `a` or `b` change.
- Can we make any variable a dependency?
  - Only props and state make sense as dependencies, because only they can change between renders.
  - More specifically, your dependencies should be any and all props and/or state referenced by your effect.

Fetching Data in React: An Example

```
export default function Museum() {
  const [galleries, setGalleries] = useState([]);
  useEffect(/* What goes here? */);
  return (
    <>{galleries.map(gallery => <Gallery […]/>)}</>
  )
}
```

Fetching Data in React: An Example

```
export default function Museum() {
  const [galleries, setGalleries] = useState([]);
  useEffect(() =>
    fetch(“http://museum.com/galleries”)
      .then(response => response.json())
      .then(data => setGalleries(data))
  ), []);
  return <>{galleries.map(gallery => <Gallery […]/>)}</>
}
```

The “Effect” of an Effect’s Dependency Array

Dependency Array	Outcome
none	Run after every render.
`[]`	Run only on mount, i.e. after the first render.
`[a, b, …]`	Run on mount and whenever a dependency changes.

Other Effects

- In general, effects are used to synchronize one, some, or all of a component's renders with some external system.
  - Fetching data on mount synchronizes a component's first render with an external resource (e.g. a JSON server or an API).
  - Others include DOM methods, animations, and subscriptions.
  - Some effects require cleanup.

and now, a demo 

Thanks!