Web Dev 2

CS571: Building User Interfaces

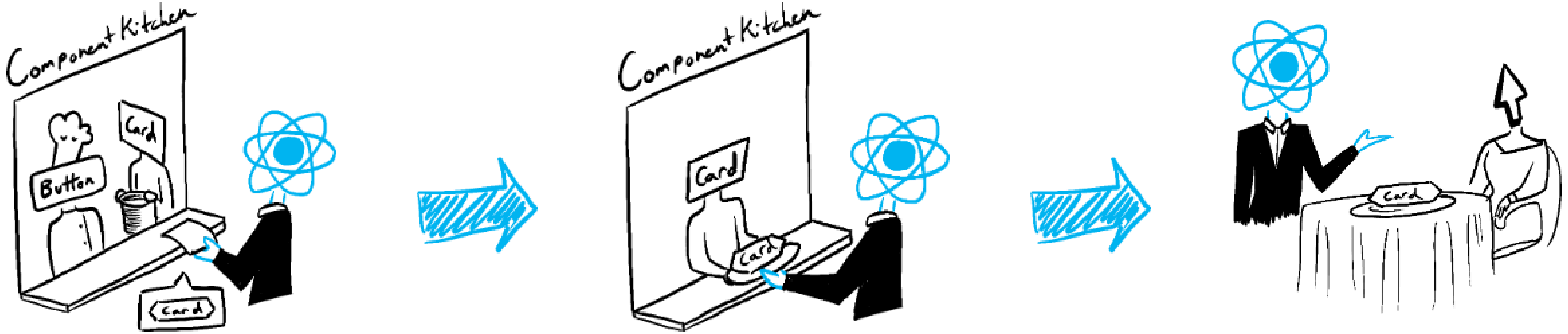
Cole Nelson

Before Lecture

- Clone today's code to your machine.
  - Run the command `npm install` inside of the `starter` and `solution` folders.

Learning Objectives

1. Be able to understand the React component lifecycle.
2. Be able to `map` out lists of data responsively.
3. Be able to break up data using pagination.
4. Be able to use controlled `input` components.



Trigger Render Commit

Illustration by Rachel Lee Narbors

React Component Updates

1. **Trigger:** Occurs on the initial mounting of a component, or on a state or `props` update.
2. **Render:** Runs the component to re-calculate its `return`, rendering to virtual DOM (cheap!)
3. **Commit:** Diffs the virtual DOM with the real DOM, painting the minimal number of changes to the screen. Because this is expensive, it occurs periodically with batched renders.

React Component Lifecycle

A `useEffect` allows us to listen at certain stages of the component lifecycle...

1. **Mounting:** Triggered when the component is instantiated by its parent.
2. **Commitment:** Triggered when an update has been painted to the screen.
3. **Unmounting:** Triggered when the component is no longer referenced by its parent.

`useEffect` in the Component Lifecycle

Used to run a function at a particular point of the lifecycle, specifically to synchronize with *external systems*, e.g. `fetch`'ing data from an API.

```
useEffect(cb, deps)
```

`cb` : The callback function we wish to run.
`deps` : On which changes this `cb` should be ran.

`useEffect` in the Component Lifecycle

```
useEffect(() => {
  console.log("I run on component mount!")
}, []);
```

```
useEffect(() => {
  console.log("I run on component mount and whenever name changes!")
}, [name]);
```

`useEffect` in the Component Lifecycle

We can also return a cleanup function that runs on dependency change and unmount.

```
useEffect(() => {
  console.log("I run on component mount!")

  return () => {
    console.log("And I run on component unmount!")
  }
}, []);
```

`useEffect` in the Component Lifecycle

```
useEffect(() => {
  console.log("I run on component mount and and whenever name changes!")

  return () => {
    console.log("And I run when name is about to change and on unmount!")
  }
}, [name]);
```

`useEffect` Anti-Patterns

Use `useEffect` to synchronize *with an external system*, not to represent derived state!

This is an *anti-pattern*!

```
function Person() {
  const [name, setName] = useState("James");
  const [derivedInitial, setDerivedInitial] = useState("J");
  useEffect(() => {
    setDerivedInitial(name[0]);
  }, [name])
}
```

`useEffect` Anti-Patterns

Use a variable to represent *derived state*; it is calculated on each render!

This is *preferred*.

```
function Person() {
  const [name, setName] = useState("James");
  const derivedInitial = name[0];
}
```

`useEffect` Anti-Patterns

Why? For many reasons...

- The code is more difficult to read!
- It is slower, requiring 2 real DOM updates!
- The React docs say so!

Of course, how you write your code is up to you! :) In this class, we grade based on functionality. Conform to the norms of your team.

Your Turn!

Reflect on the `console.log`s of the starter code.

React Lists Basics

Last time we tried to show many recipes...

```
function AllRecipes() {
  // Is there a better way to do this? We'll explore this next time!
  const [pizza, setPizza] = useState();
  const [pasta, setPasta] = useState();
  const [chili, setChili] = useState();

  // ...
}
```

... but we didn't have a good way to do it!

React Lists Basics

So we did the following...

```
<Recipe {...pizza}/>
<Recipe {...pasta}/>
<Recipe {...chili}/>
```

Objective: Create a `Recipe` component for each recipe; i.e. we want to tranfrom a list of JS objects to a list of JSX components.

...Remember a declarative way to do this?

React Lists Basics

`map` each JS object to a JSX component! e.g.

```
function AllRecipes() {
  const [recipes, setRecipes] = useState([]);

  // ... Still need to fetch the recipes! ...

  return <div>
    {
      recipes.map(r => <Recipe {...r}/>)
    }
  </div>
}
```

You'll still need to specify a `key`

Your Turn!

Use Postman to explore the recipe data from...

https://cs571api.cs.wisc.edu/rest/su24/ice/all-recipes

... then implement the code to display them all as `Recipe` components!

Note on Hot Reloading

React (Vite), for better or for worse, will keep your old state when hot reloading.

The prior solution will result in duplicates upon saving your solution, but not upon refreshing the page.

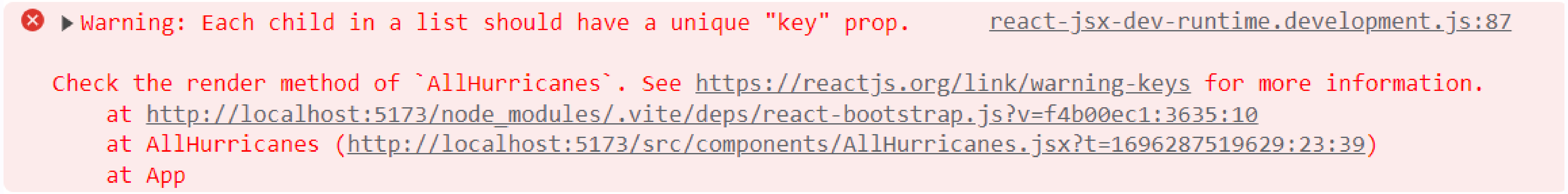
Fetching Data

Best. **Why?** We are *overwriting* the value. No need to worry about duplicates, they are overwritten.

```
function AllHurricanes() {
  const [recipes, setRecipes] = useState([]);
  useEffect(() => {
    fetch("https://cs571api.cs.wisc.edu/rest/su24/ice/all-recipes")
      .then(res => res.json())
      .then(data => {
        setRecipes(data) // Good :)
      })
  }, [])
  // ...
}
```

Uh oh!

Check your console!



```
▶ Warning: Each child in a list should have a unique "key" prop.     react-jsx-dev-runtime.development.js:87

Check the render method of `AllHurricanes`. See https://reactjs.org/link/warning-keys for more information.
    at http://localhost:5173/node_modules/.vite/deps/react-bootstrap.js?v=f4b00ec1:3635:10
    at AllHurricanes (http://localhost:5173/src/components/AllHurricanes.jsx?t=1696287519629:23:39)
    at App
```

Each component needs a unique `key`.

React `key` Prop

The `key` prop is used by React to uniquely identify elements within lists to speed up rendering.

- Always use a *unique* key for the *parent-most* element rendered in a list.
- This key needs to be *unique among siblings*.
- This key should *usually* not be the index of the item (e.g. what if the order changes?)

Learn More

Responsive Design

We use react-bootstrap.

See the grid docs.

Important takeaways...

- Use `Container`, `Row`, and `Col` components.
- `xs`, `sm`, `md`, `lg`, `xl`, and `xxl` are props.

Responsive Design

This is how we wrote Bootstrap in Vanilla JS...

```html
<div class="container">
  <div class="row">
    <div class="col-12 col-md-6 col-lg-3"></div>
    <div class="col-12 col-md-6 col-lg-3"></div>
    <div class="col-12 col-md-6 col-lg-3"></div>
    <div class="col-12 col-md-6 col-lg-3"></div>
  </div>
</div>
```

Responsive Design

...this is how we will in React!

```
<Container>
  <Row>
    <Col xs={12} md={6} lg={3}></Col>
    <Col xs={12} md={6} lg={3}></Col>
    <Col xs={12} md={6} lg={3}></Col>
    <Col xs={12} md={6} lg={3}></Col>
  </Row>
</Container>
```

StackBlitz

Your turn!

Make your display responsive.

Pagination

Also available in react-bootstrap.

Useful for handling large sums of data.

```
{/* Display items here. */}

<Pagination>
  <Pagination.Item active={false}>1</Pagination.Item>
  <Pagination.Item active={false}>2</Pagination.Item>
  <Pagination.Item active={false}>3</Pagination.Item>
  <Pagination.Item active={false}>4</Pagination.Item>
</Pagination>
```

Pagination

Use a state variable to track which page is active.

```
function SomeBigData() {
  const [page, setPage] = useState(1)
  return <div>
    {/* Display some data here! */}
    <Pagination>
      <Pagination.Item active={page === 1} onClick={() => setPage(1)}>1</Pagination.Item>
      <Pagination.Item active={page === 2} onClick={() => setPage(2)}>2</Pagination.Item>
      <Pagination.Item active={page === 3} onClick={() => setPage(3)}>3</Pagination.Item>
      <Pagination.Item active={page === 4} onClick={() => setPage(4)}>4</Pagination.Item>
    </Pagination>
  </div>
}
```

StackBlitz

Pagination

When displaying the data, use `slice` to only show the items on the current page!

```
function SomeBigData() {
  const [page, setPage] = useState(1)
  return <div>
    {
      bigData.slice((page - 1) * 16, page * 16).map(name => <p>{name}</p>)
    }
    {/* Display Pagination Items here! */}
  </div>
}
```

StackBlitz

Your turn!

Make your design responsive.

Manging State

Notice how the state disappears when flipping through the pages? Why is this?

We'll explore a way to handle this in the next lecture!

Handling Text Input

We can get user input using the HTML `input` tag or the React-Bootstrap `Form.Control` component.

We can get user input...

- in a *controlled* way using its `value` and tracking `onChange` events
- in an *uncontrolled* manner using `useRef`.
  - we'll cover this in the future!

Controlled Components

We can *control* an input component via its `value` and `onChange` properties.

Example of a controlled input component

Example of a controlled input component (Bootstrap)

Questions?