React 2

CS571: Building User Interfaces

Cole Nelson

What will we learn today?

- A Review of React
- Using Routing in React
- More React Hooks (`useContext` , `useCallback` , and `useMemo`)
- React Memoization

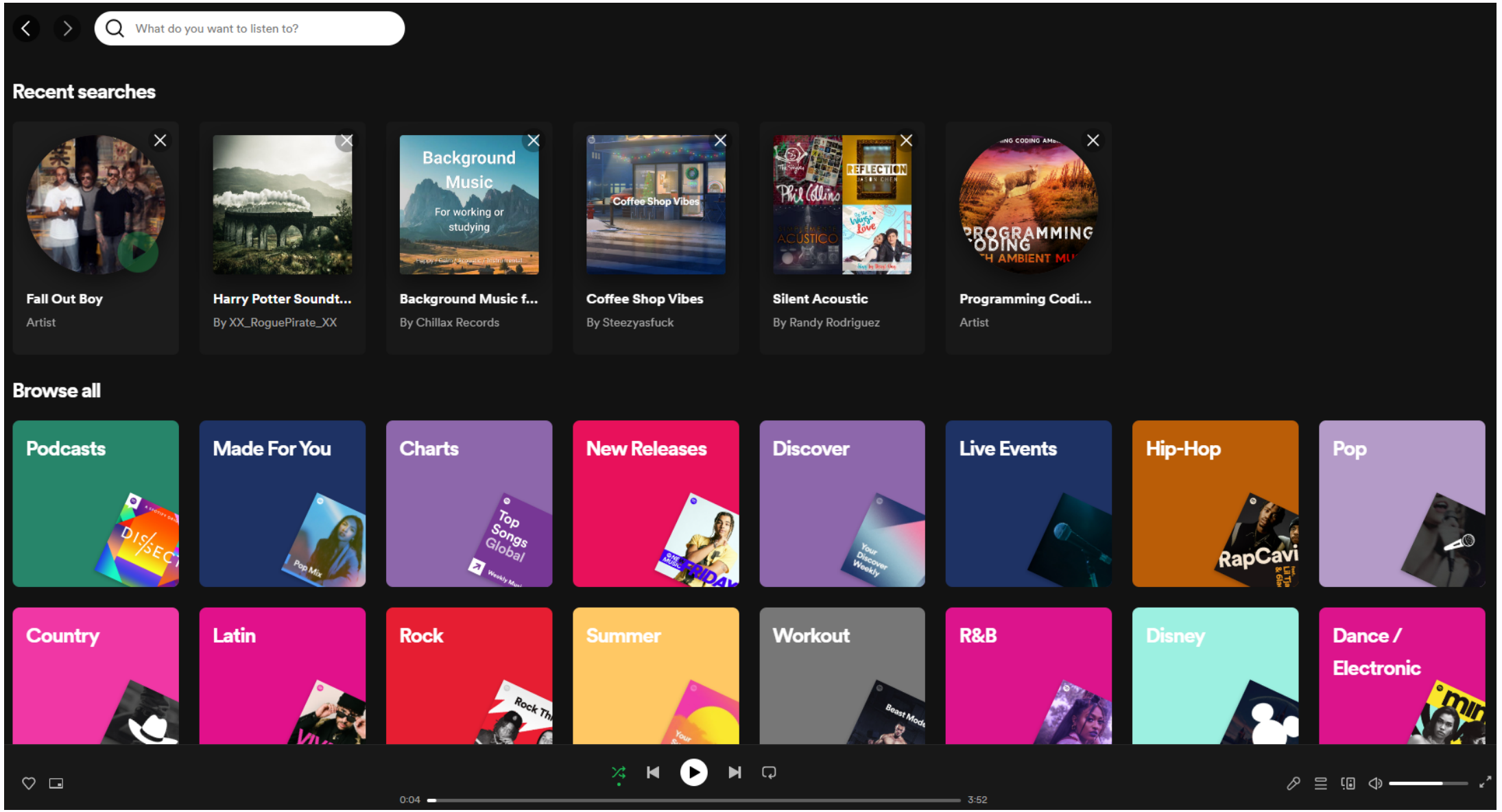
React Review

React Essentials

Every "thing" is a component (usually written in JSX).

Every component is a function, inheriting `props` and maintaining an internal `state`.

```
function Welcome() {
  return <h1>Hello World!</h1>;
}
```


What do you want to listen to?
Recent searches
Fall Out Boy
Artist
Harry Potter Soundt...
By XX_RoguePirate_XX
Background Music
For working or studying
Background Music f...
By Chillax Records
Coffee Shop Vibes
Coffee Shop Vibes
By Steezyasfuck
Silent Acoustic
By Randy Rodriguez
Programming Codi...
Artist
Browse all
Podcasts
Made For You
Charts
New Releases
Discover
Live Events
Hip-Hop
Pop
Country
Latin
Rock
Summer
Workout
R&B
Disney
Dance / Electronic
0:04
3:52

`useState` Hook

Used to maintain state! Takes an initial value as an argument. Returns a pair of the *read-only* state value and a mutator function.

Always use the mutator function to modify state.

Never modify the state directly.

```
const [name, setName] = useState("James");
```

`useEffect` Hook

Used to perform an action on page load or on state change. Takes a callback function and, optionally, an array of state dependenices as arguments.

```
useEffect(() => {
  alert("The page has been reloaded!");
})
```

```
useEffect(() => {
  alert("You changed your name to " + name);
}, [name])
```

Digression: Ref, Shallow, & Deep Copy

In JavaScript 2, we had learned about a *reference* and a *deep* copy. There also exists a *shallow* copy.

```
let myBasket = {
  basketId: 154,
  items: ["Apples", "Bananas", "Grapes"]
};

let refCopyBasket = myBasket;
let shallowCopyBasket = {... myBasket};
let deepCopyBasket = JSON.parse(JSON.stringify(myBasket));
```

Implications for Setting State

We must use a *shallow* or *deep copy* for setting state.

```
// BAD **DO NOT USE** - this is just a reference copy!
let badgersCopy = badgers;
badgersCopy.push({
  name: "Jennifer Mnookin",
  roles: ["chancellor"]
});
setBadgers(badgersCopy);
```

We should do...

```
let badgersShallowCopy = [... badgers];
let badgersDeepCopy = JSON.parse(JSON.stringify(badgers));
```

In-Class Example

Use a new React w/ JavaScript StackBlitz on the web, or use `create-react-app` on your own device!

See StackBlitz Solution

React is a *library*, not a *framework*!

This means that *batteries are not included*. You'll be choosing many of your own tools and libraries!

- **Layout & Design:** Bootstrap React-Bootstrap, Reactstrap, Material, Elemental, Semantic
- **Routing & Navigation:** React Router, React Navigation, React Location
- **State Management:** Redux, Recoil, MobX, XState

Navigation w/ React Router

See StackBlitz

Types of Routers

- `BrowserRouter` : What you typically think of!
- `MemoryRouter` : Same as `BrowserRouter` , but the path is hidden from the browser in memory! 🤫
- `HashRouter` : Ugly, early days `#` implementation.
- `StaticRouter` : Used for server-side rendering.
- `NativeRouter` : We'll use react-navigation instead!

Routing

Using a `Router`, `Routes`, and `Route`!

```
<BrowserRouter>
  <Routes>
    <Route path="/" element={<Layout />}>
      <Route index element={<Home />} />
      <Route path="about-us" element={<AboutUs />} />
      <Route path="other-info" element={<OtherInfo />} />
      <Route path="*" element={<Home />} />
    </Route>
  </Routes>
</BrowserRouter>
```

Navigable Components

Notice how each route maps to a component.

```
function Home() {
  return <h2>Home</h2>
}
function AboutUs() {
  return <h2>About Us :)</h2>
}
function OtherInfo() {
  return <h2>Other Info!</h2>
}
```

Navigation

Navigation for a `BrowserRouter` is done via URLs.

```
<>
  <Navbar bg="dark" variant="dark">
    <Nav className="me-auto">
      <Nav.Link as={Link} to="/">Home</Nav.Link>
      <Nav.Link as={Link} to="/about-us">About Us</Nav.Link>
      <Nav.Link as={Link} to="/other-info">Other Info</Nav.Link>
    </Nav>
  </Navbar>
  <Outlet />
</>
```

React Hooks

React Hooks

Last week we covered...

- `useState`
- `useEffect`

Today we will cover...

- `useContext`
- `useCallback`
- `useMemo`

React Hooks

- `useContext` : A useful hook for managing state across web apps with large component hierarchies.
- `useCallback` : An optimization hook for preserving callback functions.
- `useMemo` : An optimization hook for preserving calculated values.

`useContext` Hook

Motivation: How can we effectively manage state for web apps with large component hierarchies?

```
SpotifyLandingPage
 - NavBar
   - NavArrows
   - SearchBox
 - RecentSearches
   - AuthorCard
     - AuthorImage
     - AuthorName
```

Alt. Solution: State Management Libraries

`useContext` Hook

Three steps to using context.

1. Create and export a context.
2. Provide the context with some value.
3. Use the context in a child component.

Often used in combination with `useState`.

`useContext` Hook

A context must be exported.

```
export const MyDataContext = createContext([]);
```

`useContext` Hook

A context must be provided to child component(s).

```
function ParentComponent() {
  const [data, setData] = useState([]);
  return (
    <MyDataContext.Provider value={[data, setData]}>
        <SomeChildComponent />
        <SomeOtherChildComponent />
    </MyDataContext.Provider>
  );
}
```

`useContext` Hook

The context can be used by any of child, grandchild, great-grandchild, etc. component(s).

```
function SomeChildComponent() {
  const [data, setData] = useContext(MyData);
  return (
    { /* Do something interesting with data here! */ }
  );
}
```

See StackBlitz

`useCallback` Hook

Consider the following functional component...

```
function MyApp() {
  const myComplicatedFunction = () => {
    // ...
  }

  return <>
    <button onClick={myComplicatedFunction}>Click Me</button>
  </>
}
```

How many times do we *create* the function `myComplicatedFunction`? We do on *every render*!

`useCallback` Hook

`useCallback` is used to 'memoize' a callback function.

```
function MyApp() {
  const myComplicatedFunction = useCallback(() => {
    // ...
  }, []);

  return <>
    <button onClick={myComplicatedFunction}>Click Me</button>
  </>
}
```

Takes a callback function to 'memoize' and an optional list of dependencies (e.g. when to re-'memoize').

`useMemo` Hook

Same thing as `useCallback`, except memoizes the *value* of a *callback* rather than the *callback* itself.

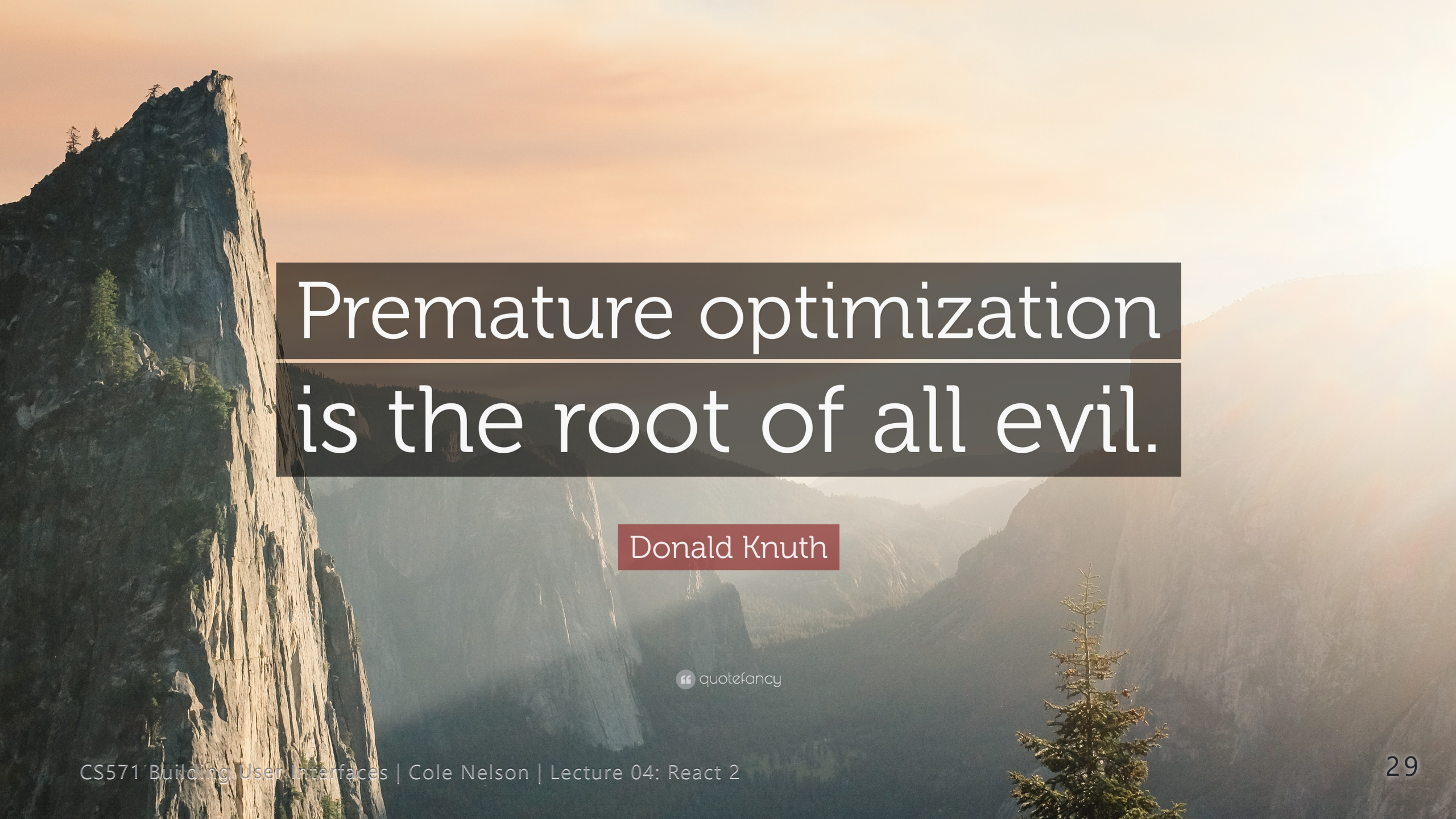
```
function MyApp() {
  const myComplicatedValue = useMemo(() => { /* Some complex call */}, []);

  return <>
    <p>{myComplicatedValue}</p>
  </>
}
```

`memo` -ized Components

Used for creating *purely functional* components. Given the same props, the function renders the same output.

```
//                      v--- Name of functional component!
export default memo(GroceryList, (prevProps, nextProps) => {
  return prevProps.apples === nextProps.apples &&
    prevProps.bananas === nextProps.bananas &&
    prevProps.coconuts === nextProps.coconuts;
})
```

See StackBlitz for `useCallback`, `useMemo`, and `memo`


Premature optimization is the root of all evil.
Donald Knuth
quotefancy

A Plea for Lean Software

Niklaus Wirth
ETH Zürich

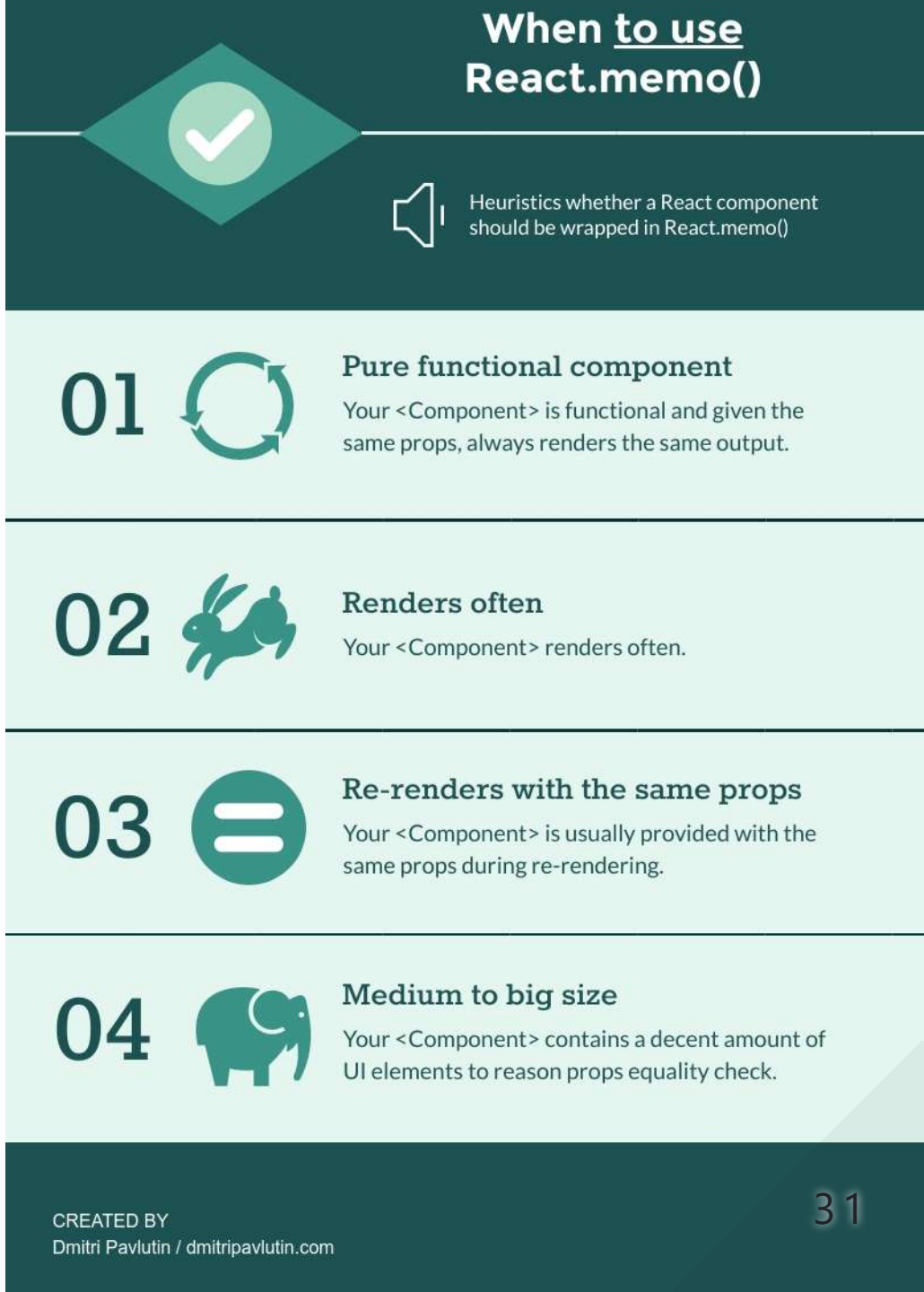
Memory requirements of today's workstations typically jump substantially—from several to many megabytes—whenever there's a new software release. When demand surpasses capacity, it's time to buy add-on memory. When the system has no more extensibility, it's time to buy a new, more powerful workstation. Do increased performance and functionality keep pace with the increased demand for resources? Mostly the answer is no.

About 25 years ago, an interactive text editor could be designed with as little as 8,000 bytes of storage. (Modern program editors request 100 times that much!) An operating system had to manage with 8,000 bytes, and a compiler had to fit into 32 Kbytes, whereas their modern descendants require megabytes. Has all this inflated software become any faster? On the contrary. Were it not for a thousand times faster hardware, modern software would be utterly unusable.

Finding a Balance

1. Given the same input, renders the same output.
2. Is rendered often.
3. Does not change often.
4. Is of substantial size.

Dmitri Pavlutin Blog Post



Badger Bingo

Cumulative example, see StackBlitz.

What did we learn today?

- A Review of React
- Using Routing in React
- More React Hooks (`useContext` , `useCallback` , and `useMemo`)
- React Memoization

On to Web Design!