CS 4530: Fundamentals of Software Engineering
Module 8: React Basics

Adeel Bhutta and Mitch Wand

Khoury College of Computer Sciences

© 2025 Released under the CC BY-SA license

Learning Objectives for this Lesson

- By the end of this lesson, you should be able to:
  - Understand how the React framework binds data (and changes to it) to a UI
  - Create simple React components that use state and properties

HTML: The Markup Language of the Web

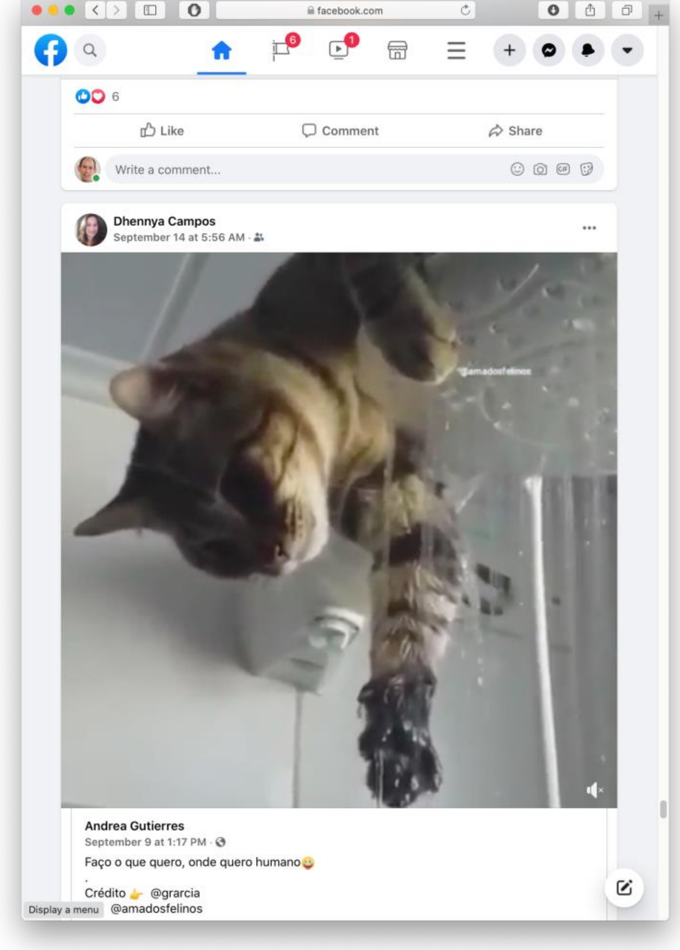
- Language for describing structure of a document

- Denotes hierarchy of elements

- What might be elements in this document?

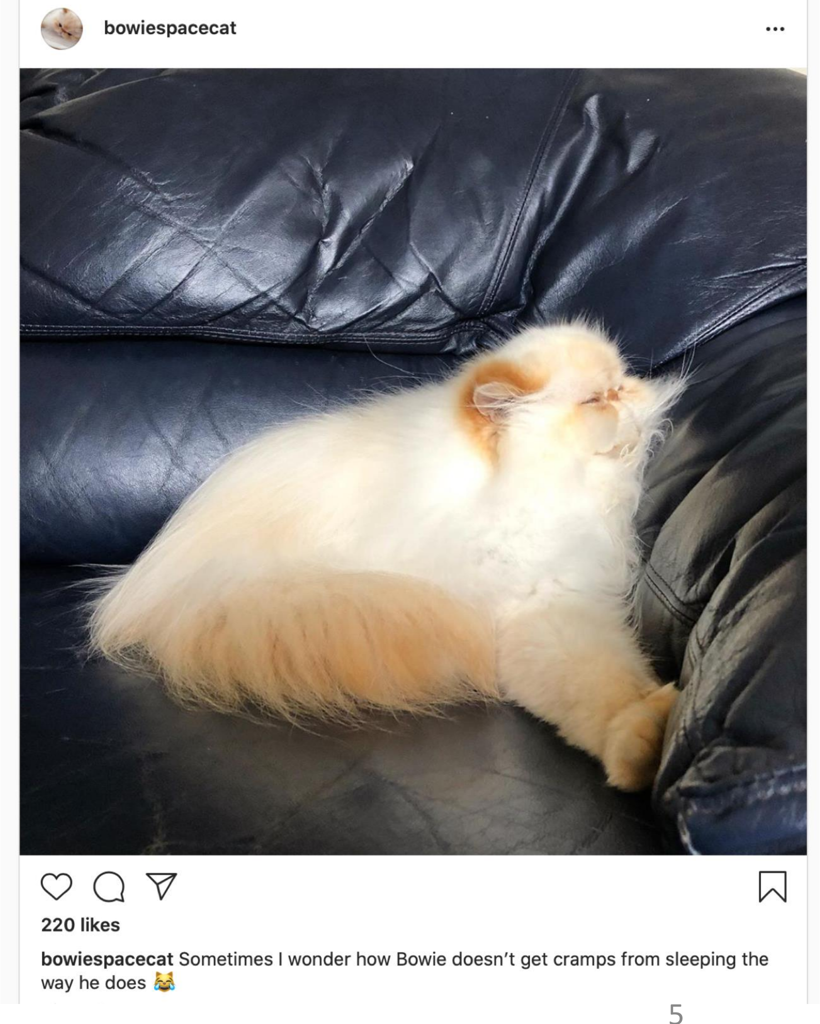
Rich, interactive web apps

- Infinite scrolling of cats



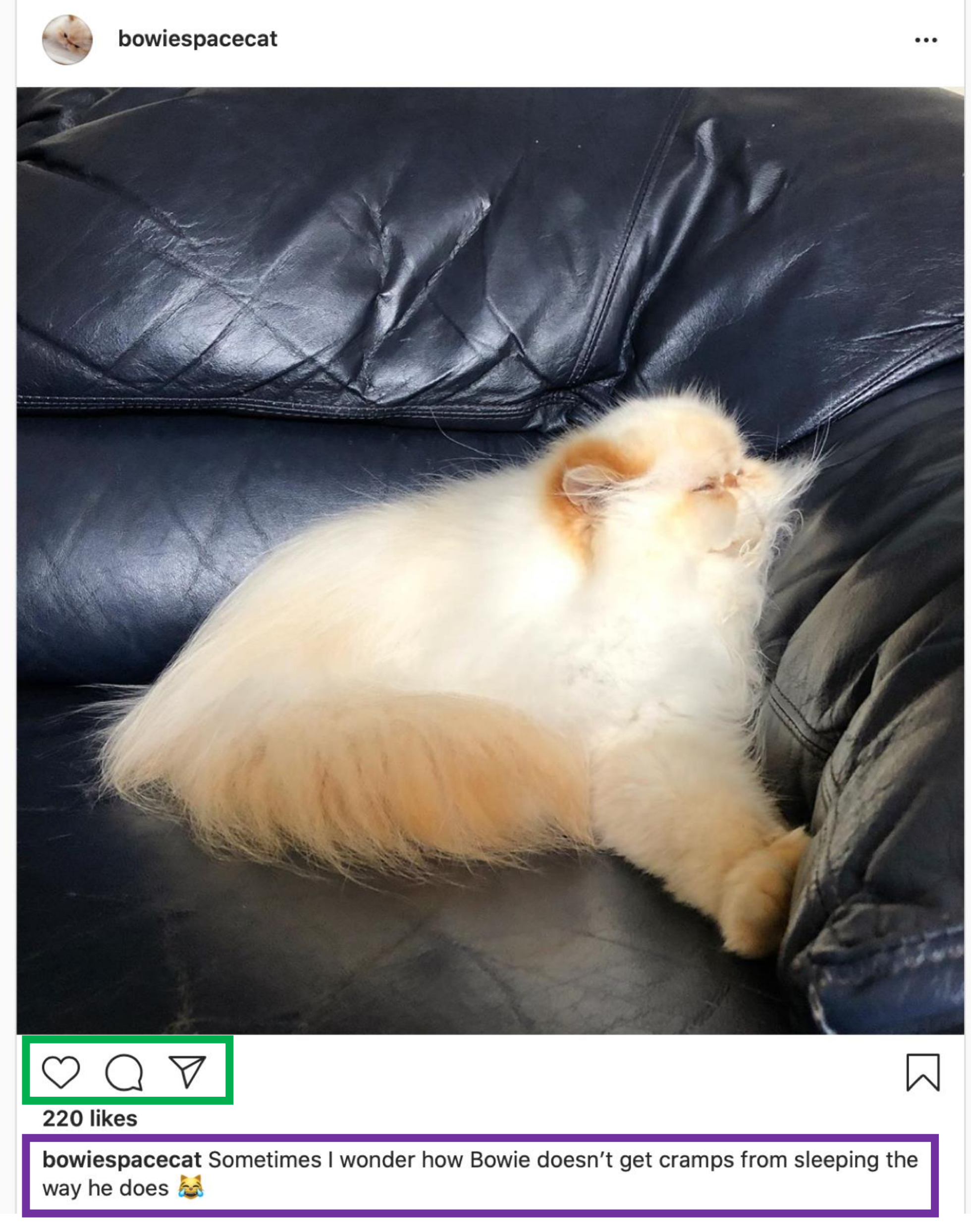
Typical properties of web app UIs

- Each widget has both visual presentation & logic
- Some widgets occur more than once
  - e.g., comment/like widgets
- Changes to data should cause changes to widget
  - e.g., new images, new comments should show up in real time
- Widgets have hierarchical structure
- Action on a widget may affect other widgets
  - e.g., clicking on 'like' button executes some logic related to the widget itself,
  - It may also affect the widget that contains the ‘like’ button



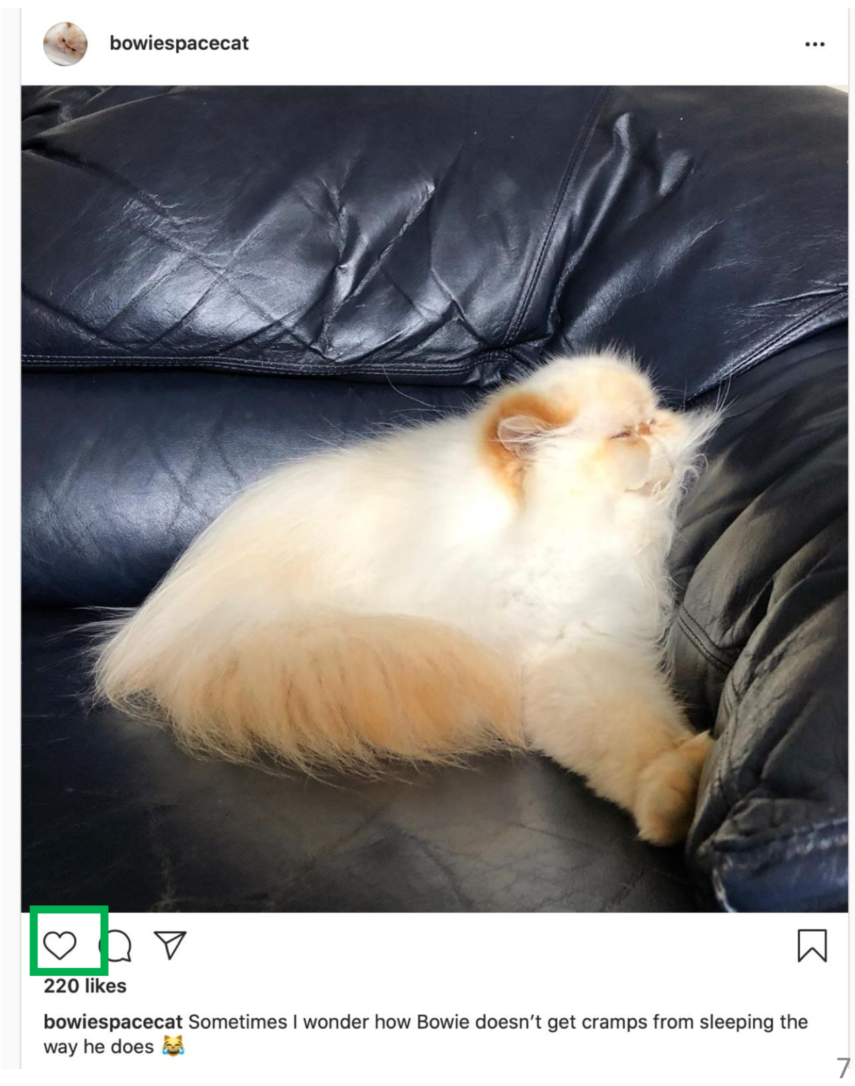
Components represent widgets in object-like style

- Organize related logic and presentation into a single unit
  - Includes necessary state and the logic for updating this state
  - Includes presentation for rendering this state into HTML
- Synchronizes state and visual presentation
  - Whenever state changes, HTML should be rendered again



Components
Example: Like button component

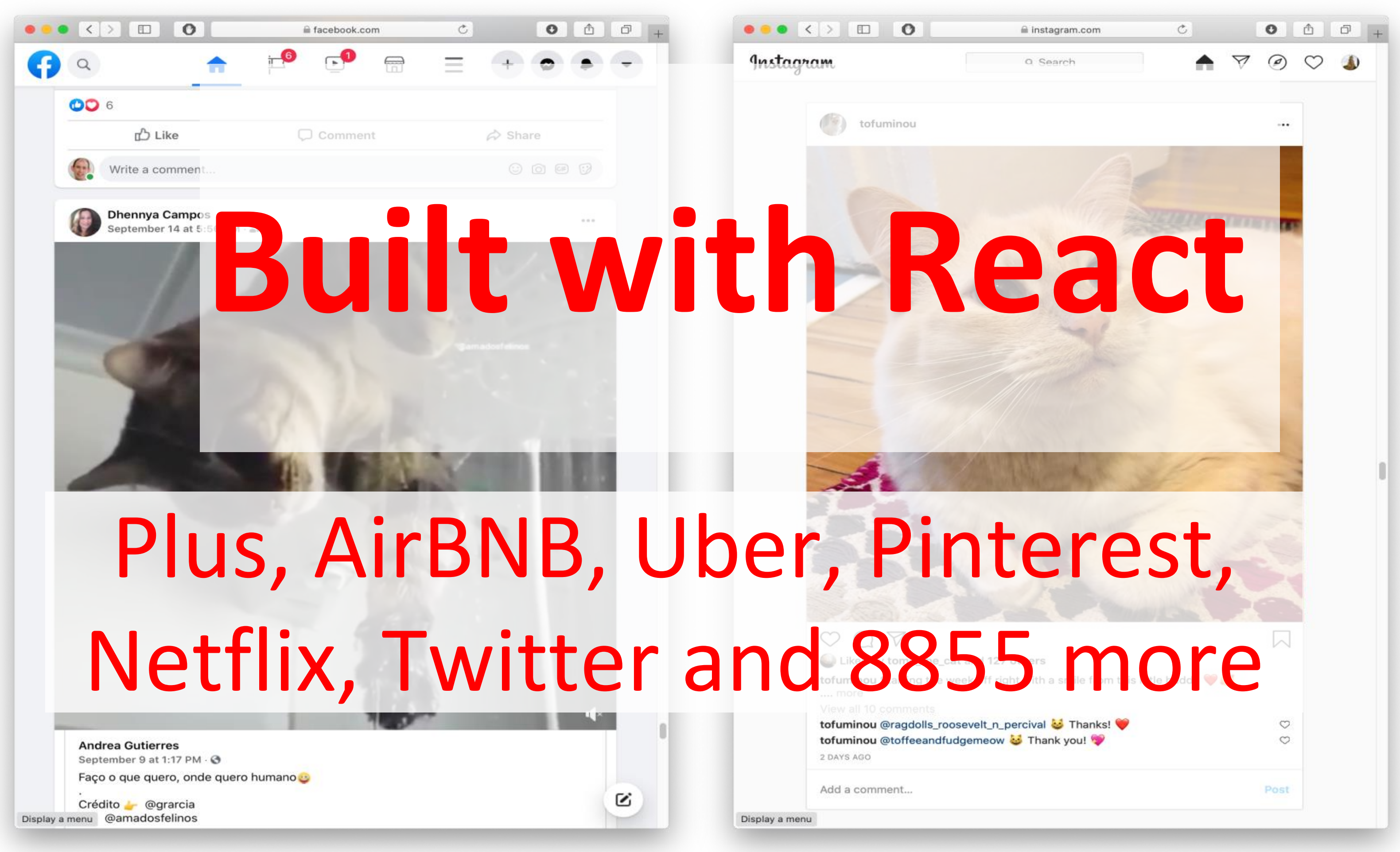
- What does the button keep track of?
  - Is it liked or not
  - What post this is associated with
- What logic does the button have?
  - When changing like status, send update to server
- How does the button look?
  - Filled in if liked, hollow if not



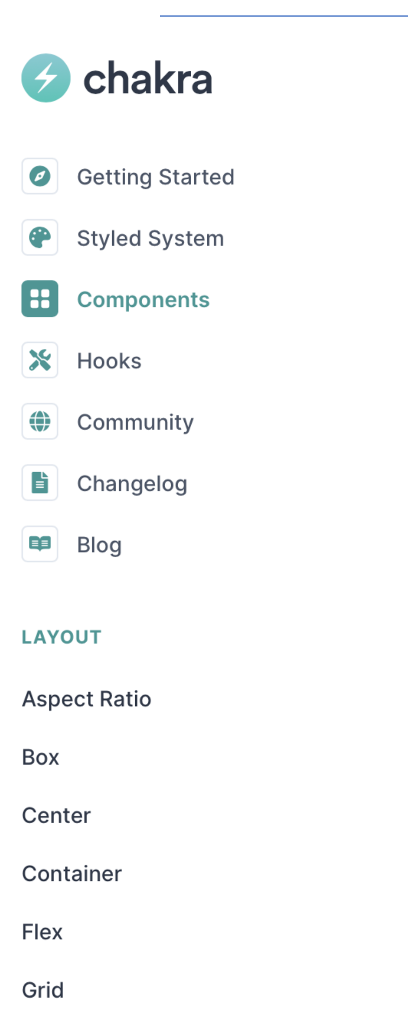
React is a Framework for Components

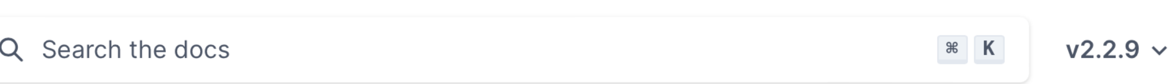
- Created by Facebook
- Powerful abstractions for describing UI components
- Official documentation & tutorials: https://reactjs.org/
- Components are constructed in the browser (“front-end”)
- Key concepts:
  - Embed HTML in TypeScript
  - Track application “state”
  - Automatically and efficiently re-render page in browser based on changes to state
- But: some implementations of React allow components to be pre-constructed in the server.

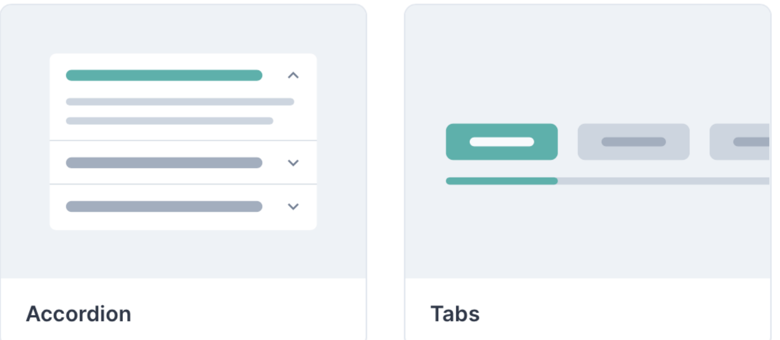
React makes it easy to build rich, interactive web apps (perhaps with infinite scrolling of cats!)



React Has a Rich Component Library







Installing Chakra for next.js:

- Just say:

```
npm i --save @chakra-ui/react @chakra-ui/next-js
npm i --save @emotion/react @emotion/styled framer-motion
```

Embedding HTML in TypeScript
Aka JSX or TSX

- How do you embed HTML in TypeScript and get syntax checking?
- Idea: extend the language: JSX, TSX
  - JavaScript (or TypeScript) language, with additional feature that expressions may be HTML
- It's a new language
  - Browsers do not natively run JSX (or TypeScript)
  - We use build tools that compile everything into JavaScript

```
export function HelloMessage(props: IProps) {
  return (
    <div>
      Hello, {props.name}
    </div>
  )
}

ReactDOM.render(
  <React.StrictMode>
    <HelloMessage name='Satya' />
  </React.StrictMode>,
  document.getElementById('root')
);
```

JSX/TSX Embeds HTML in TypeScript

- Example:

```
return <div>Hello {someVariable}</div>;
```

- HTML embedded in TypeScript
  - HTML can be used as an expression
  - HTML is checked for correct syntax
- Can use { expr } to evaluate an expression and return a value
  - e.g., { 5 + 2 }, { foo() }
- To wrap on multiple lines, wrap the TSX/JSX in parentheses (…)
- Value of expression is a piece of HTML

Hello World in React

src/app/Apps/HelloWorld.tsx

```
import * as React from 'react';
import { Heading, VStack } from '@chakra-ui/react';

function HelloWorldComponent() {
    return (
        <VStack>
            <Heading>Hello World</Heading>
        </VStack>
  )
}
```

“Return this HTML whenever the component is rendered”

The HTML is dynamically generated by the library.

Start your app by importing it into src/app/page.tsx

src/app/page.tsx

```
'use client'; // this app is client-side only.

import App from './Apps/HelloWorldApp' // or from wherever you app lives
// import App from './Apps/HelloWorldDave'
// import App from './Apps/App1';


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

React Components are Little Machines

- They start with input from their creator (parent) (props)
- They have additional local state ("component state")
- They retain their local state when their parent's state changes
- They may change their local state in response to external stimuli (button presses, etc.)
- They re-render when their local state changes, or when they are re-created by their parent with different props.

Component State is Data That Changes

- State is created by `useState.`
- The state is accessed through *state variables* in the component.
- The first variable is the accessor, the second is the setter.
- The only way to change the value of a state variable is with the setter
- In general, the state consists of several variables.
- The component only re-renders after its local state changes

```
import { useState } from 'react';
function Foo() {
 const [count, setCount] = useState(0)
 ….
}
```

You could choose any names for the variable and its setter; for this class, please follow the naming convention (goodVariableName, setGoodVariablename) that we'e used here.

app/Apps/SimplestState.tsx

Example

```
export default function SimplestState() {

  const [count, setCount] = useState(0)

  function handleClick() { setCount(count + 1) }

  return (
    <VStack>
      <Box> count = {count} </Box>
      <Button onClick={handleClick} >
        Increment Count!
      </Button>
    </VStack>
  )

}
```

(Some styling has been removed to reduce clutter on this screen. Look at the file for details}

Components don't change their state directly

```
export default function SimplestState() {

  const [count, setCount] = useState(0)

  function handleClick() {
      setCount(count + 1)
  }

  return (
    <VStack>
      <Box> count = {count} </Box>
      <Button onClick={handleClick} >
        Increment Count!
      </Button>
    </VStack>
  )
}
```

1. A setter is just a callback.
2. Every so often, React collects all the set requests it has received since the last redisplay.
3. React executes all the outstanding set requests.
4. Last, React redisplays the component with the new state.

src/app/Components/SimplestStatePlus3.tsx

Setters are not synchronous

- ***A setter doesn't change the state immediately***: it is just a request to REACT to update the state when this component is redisplayed.
- Consider the following:

```
function handleClick() {
    setCount(count + 1)
    setCount(count + 1)
    setCount(count + 1)
}
```

Console methods: https://developer.mozilla.org/en-US/docs/Web/API/console

In general, an app in React is a tree of components

- Each component has a single parent (except for the root component in page.tsx)
- A component may have children, which are other components
- A component initializes its children by passing them *properties* (typically called "props")

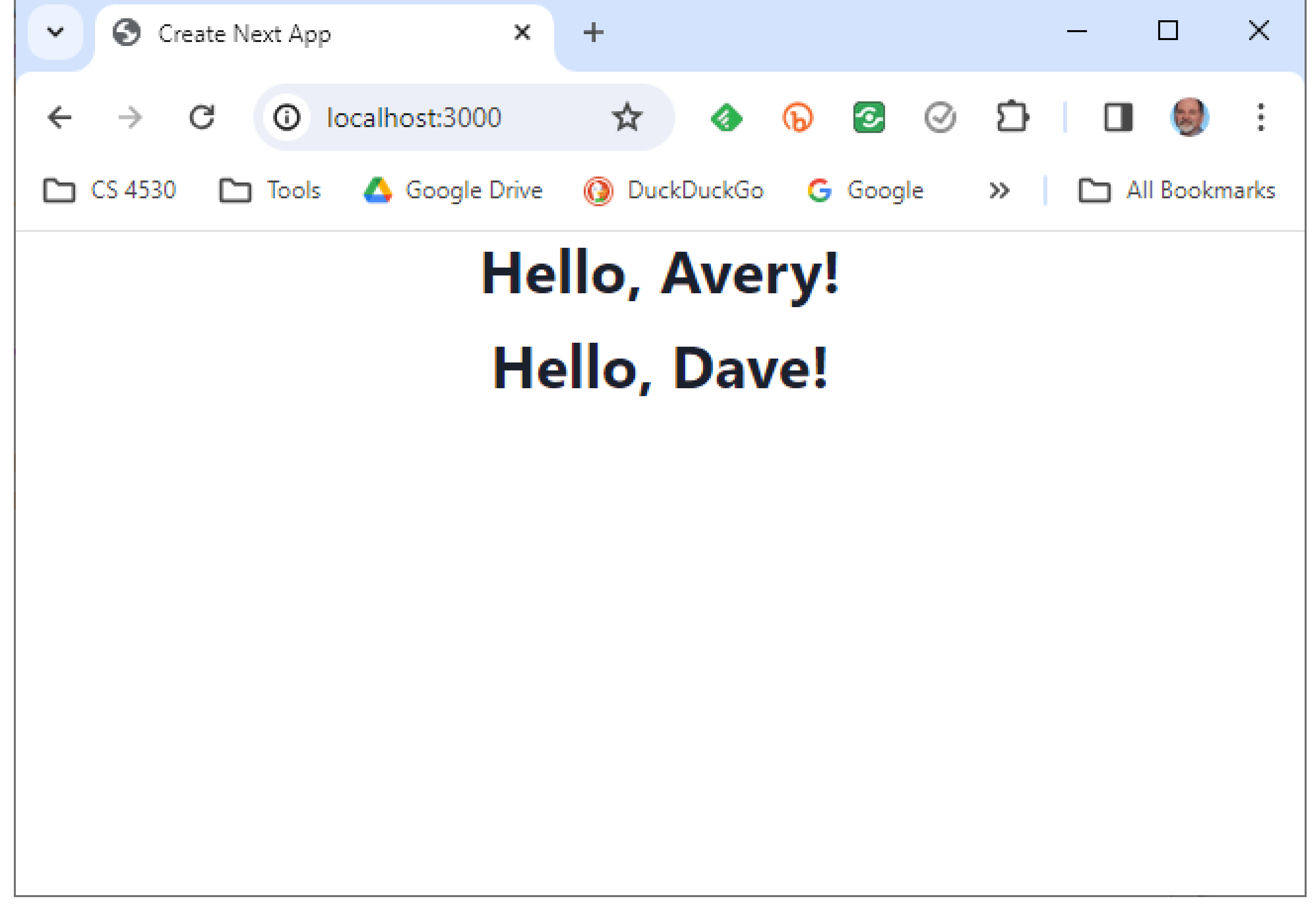
The parent passes properties to its children by name.

```
export default function HelloWorldWithAveryAndDave() {
    return (
    <VStack>
      <HelloWorldWithName name='Avery'/>
      <HelloWorldWithName name='Dave'/>
    </VStack>
    )
}
```

A component receives properties from its parent as a record

- Properties are passed as a single argument to the component
- Properties can *not* be changed by the component

```
export default function HelloWorldWithName(
  props: { name: string }
  ) {
  return (
    <VStack>
      <Heading>Hello, {props.name}!</Heading>
    </VStack>
  );
}
```


Create Next App
localhost:3000
CS 4530
Tools
Google Drive
DuckDuckGo
Google
All Bookmarks
Hello, Avery!
Hello, Dave!

A component can pass handlers to its children

```
// create two CountingButtons, and keep track of the total count.
import { CountingButton } from './CountingButton';

export default function App() {
  const [globalCount, setGlobalCount] = useState(0)

  function incrementGlobalCount() {
    setGlobalCount(globalCount + 1)
  }

  return (
    <VStack spacing='30px'>
      <Box border="1px" padding='1'>Total count = {globalCount}</Box>
      <CountingButton name="Button A" globalCount= {globalCount}
                      onClick={incrementGlobalCount} />
      <CountingButton name="Button B" globalCount= {globalCount}
                      onClick={incrementGlobalCount} />
    </VStack>
  )
}
```

src/app/Components/TwoCountingButtons.tsx

A child communicates with its parent by calling the handler it was passed

```tsx
export function CountingButton(props: {
  // display name of the button
  name: string;
  // global count from parent
  globalCount: number;
  // event handler to call when clicked
  onClick: () => void;
}) {
  const [localCount, setLocalCount]
    = useState(0);

  function handleClick() {
    setLocalCount(localCount + 1);
    props.onClick(); // propagate to parent}
  }
```

```tsx
  return (
    <VStack>
      <Box>
        local count for {props.name} = {localCount}
      </Box>
      <Box>
        globalCount = {props.globalCount}
      </Box>
      <Button onClick={handleClick}>
        Increment {props.name}!
      </Button>
    </VStack>
  );
```

src/app/Components/CountingButton.tsx

TwoCountingButtons demo

Total count = 0

local count for Button A = 0

globalCount = 0

Increment Button A!

local count for Button B = 0

globalCount = 0

Increment Button B!

Setters initiate the redisplay process

1. A setter sends a request to React.
2. Every so often, React collects all the set requests it has received since the last redisplay.
3. React executes all the outstanding set requests.
4. Last, React redisplays the component with the new state.
5. When a component re-renders, its children retain their state; the children are re-rendered only if their props change.

React works hard to make redisplay fast.

- Updating the DOM in the browser is slow - it is *vital* that React does efficient diff'ing
  - Example: adding a new comment on a YouTube video shouldn't make the browser re-layout the whole page
- React makes re-rendering faster by updating only the part that changes.
  - This is called "Reconciliation"

A New Pattern: display a list of items using **map**

```
export function ToDoListDisplay(props: { items: ToDoItem[],
                                          onDelete:(id:string) => void })

  return (
    <Table>
      <Tbody>
       {
          props.items.map((eachItem) =>
              <ToDoItemDisplay item={eachItem}
                key={eachItem.id}
                onDelete={props.onDelete} />)
        }
      </Tbody>
    </Table>
  )
}
```

src/app/Apps/ToDoApp/ToDoListDisplay.tsx

But using map comes with a big gotcha.

```
export function ToDoListDisplay(props: { items: ToDoItem[],
                                         onDelete:(id:string) => void })
  return (
    <Table>
      <Tbody>
       {
          props.items.map((eachItem) =>
              <ToDoItemDisplay item={eachItem}
                key={eachItem.id}
                onDelete={props.onDelete} />)
        }
      </Tbody>
    </Table>
  )
}
```

The ToDo App

src/app/Apps/ToDoApp/ToDoApp.tsx

```
export default function ToDoApp () {
  const [todoList,setTodolist] = useState<ToDoItem[]>([])
  const [itemKey,setItemKey] = useState<number>(0)   // first unused key

  function handleAdd (title:string, priority:string) {
    if (title === '') {return}   // ignore blank button presses
    setTodolist(todoList.concat({title: title, priority: priority, key: itemKey}))
    setItemKey(itemKey + 1)
  }

  function handleDelete(targetKey:number) {
    const newList = todoList.filter(item => item.key != targetKey)
    setTodolist(newList)
  }

  return (
  <VStack>
    <Heading>TODO List</Heading>
    <ToDoItemEntryForm onAdd={handleAdd}/>
    <ToDoListDisplay items={todoList} onDelete={handleDelete}/>
  </VStack>
  )
}
```

Typical Page

TODO List

Add TODO item here:

type item name here | type priority here | Add TODO item

TITLE	PRIORITY	DELETE
first item	11	🗑
second item	22	🗑
third item	optional	🗑

app/Apps/ToDoItemEntryForm.tsx

A new pattern: an input form

```
export function ToDoItemEntryForm (props: {onAdd:(item:ToDoItem)=>void}) {
    // state variables for this form
    const [title,setTitle] = useState<string>("")
    const [priority,setPriority] = useState("")
    const [key, setKey] = useState(0)     // key is assigned when the item is cre

    function handleClick(event) { --- } // on next slide…

    return (
      <VStack spacing={0} align='left'>
        <form>
          <FormControl>
            <VStack align='left' spacing={0}>
            <FormLabel as="b">Add TODO item here:</FormLabel>
            <HStack w='200' align='left'>

            <Input
              name="title"
              value={title}
              placeholder='type item name here'
              onChange={(event => {
                setTitle(event.target.value);
                console.log('setting Title to:', event.target.value)
              })}
            />
```

The state of the form is kept in the state variables of the component

One <Input> component for each blank space in the form.

Update the state variable at every keypress

handleClick

```
// state variables for this form
const [title,setTitle] = useState<string>("")
const [priority,setPriority] = useState("")

function handleClick(event) {
  event.preventDefault()  // magic, sorry.
  props.onAdd(title,priority)    // tell the parent about the new item
  setTitle('')   // resetting the values redisplays the placeholder
  setPriority('')   // resetting the values redisplays the placeholder
}
```

Treat values of state variables as read-only

- State can hold any kind of JavaScript value, including objects.
- But you shouldn't change objects that you hold in the React state directly.
- Instead, when you want to update an object, you need to create a new one (or make a copy of an existing one), and then set the state to use that copy.

https://react.dev/learn/updating-objects-in-state

Array update cheat sheet

	avoid (mutates the array)	prefer (returns a new array)
adding	`push`, `unshift`	`concat`, `[...arr]` spread syntax (example)
removing	`pop`, `shift`, `splice`	`filter`, `slice` (example)
replacing	`splice`, `arr[i] = ...` assignment	`map` (example)
sorting	`reverse`, `sort`	copy the array first (example)

https://react.dev/learn/updating-arrays-in-state

Use **spread** to update an object

```
const [position, setPosition] = useState({ x: 0, y: 0 });

position.y = 10;
setPosition(position);  // this will not trigger a re-render

setPosition({ ...position, y: 10 }); // this works

const [anArray, setAnArray] = useState([1, 2, 3]);
setAnArray([...anArray, 4]); // this works, too
```

Review

- Now that you've studied this lesson, you should be able to:
  - Understand how the React framework binds data (and changes to it) to a UI
  - Create simple React components that use state and properties
- In Module 09, we'll study another feature of React that enhances modularity: hooks.