React Notes
Tanishq Kumar

React Official Documentation & Tic-Tac-Toe notes

to start your app via CLI, use;

npx create-react-app my-app
cd my-app
npm start

In JavaScript classes, you need to always call super when defining the constructor of a subclass. All React component classes that have a constructor should start it with a super(props) call.

To collect data from multiple children, or to have two child components communicate with each other, you need to declare the shared state in their parent component instead. The parent component can pass the state back down to the children by using props; this keeps the child components in sync with each other and with the parent component.

Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called “components” that contain both.

We call this a “root” DOM node because everything inside it will be managed by React DOM. React components are comprised of elements.

React elements are immutable. Once you create an element, you can’t change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called “props”) and return React elements describing what should appear on the screen.

5-hour React tutorial on YouTube

Essentially, React is powerful because:
1) It works a lot faster than vanilla JS by virtue of diffing, and only updating the difference in the DOM of a page.
2) It abstracts away the mechanics of HTML to make components look good. Instead of copying 40 lines of CSS from Bootstrap every time you want to make a pretty navbar, you can just call that “navbar” using React and implement that again and again seamlessly and easily in different places in your codebase.

There is a lot of functionality that React takes care of under the hood to get you to be able to write in JSX, which involves separation of concerns and mixing of languages and therefore is more intuitive for you, but involves more computational machinery to get that into pure HTML/CSS/JS that is machine-parsable.

The most common way to render elements using React is just to do ReactDOM.render(what to render, where to render it).

When people say it operates using a “declarative” paradigm, that means that you can simply declare what you want done (render X in Y tag) as opposed to imperative prompts, whereby

normally you'd have to change the state of the system by telling the computer to create a new variable to store X, then take the action of putting X in Y.innerHTML, and so forth. This is more intuitive and simple for the programmer.

Conventionally, make sure you use camelcase or functions, as well as put everything on its own line for clarity since mixing HTML/JS can otherwise be quite confusing.

A functional component is just a component (thing composed of small elements; like an HTML list) that comes about as a result of a JS function being called to put it in the DOM.

It's also good convention to make new files for each component and then link them to each other via importing things. Anywhere you use JSX—in any of these new files, you need to import React from "react", as well as export default thisFileName so that you can send it to the main react file that you're creating for use.

You can create DOM trees very easily with react, where by elements are just the base HTML and components are more complex abstractions (header, footer) that can contain several html elements (header can have header, logo, title, picture, and more, etc.) and all of these will be linked together via imports/exports so that you don't have one fat HTML file, and can abstract away all the header stuff into simple <header /> that calls that React component.

When styling things with CSS, use className instead of class in the JSX because it uses the Javascript DOM API under the hood which thinks of CSS classes with .className. You can only apply className to *elements* not components.

You can apply inline style using Javascript within the broader JSX, so you need style={{ key: value, key2: value2 }} or you can just set const styles = { key: value, key2: value2 } then style = {styles}

All the CSS magic doesn't come from some hidden React library, but is just something you find by looking for templates or cool CSS online for, say, a form, or list, and then putting it all together.

When you look at something like the YT homepage, you can see how the video list is composed of the same pattern, just with different content. This is the concept of a component—the whole point of these frameworks (Angular in this case) is that you avoid redundancy and so don't copy-paste code. Instead, you have the parent component (row) and then the children (boxes) and then those themselves contain elements. You can now start to get a sense for how all complex web apps make use of JS frameworks/libraries like React. Props, much like HTML attributes/*properties* modify the behaviour of your vanilla component and act to tailor it—much like you take the original YouTube video box and add the information (views, duration, etc.) of a particular video to get that on the YT home page.

The ternary operator—the concise if statement that goes like: a? x : y is a very commonly used structure in React, since you're always looking for efficiency in JS code.

Class based components can extends the functionality of functional components in that they can incorporate state and life-cycle, whereas the functional components can't do that—this is the different between function MyApp() and class MyApp extends React.Component. They both just take props as inputs and output HTML for a UI, though. Make sure to add the render() method at the top of the class-based component, and then its innards are exactly the same as the functional component.

All the React event handlers in JSX HTML are written like JS—just as we use className as the HTML attribute not class, we use onClick and not onclick, onMouseOver, not onmouseover, and so on. A full list of all events you can control is found here: https://reactjs.org/docs/events.html.

Note that state is powerful because props can't be modified after a component receives them, but state is something that is *meant* to be changed; and when it is, all the other components that have certain properties relying on the state of the first component get changed accordingly—without refreshing the whole page. You're making a 15-line AJAX call in 1 line of React.

State is designed to be immutable. It's like your clothes—you don't alter your original clothes when you want a change, you go get new ones. So be careful to treat state accordingly, only making duplicates and then modifying those.

when you make a new function within a class-based component aside from the native contructor() and render(), you need to bind it to the class by doing this.function() = this.function.bind(this) in the constructor.

when passing a method as a prop, don't include ()

when using setState, always return an object you want to define new state at the end of whatever logic you've implemented to get to that object's values

remember that .map() is a non-mutating function that doesn't change the original array, so you have to define a new var that will assume that new property you're mapping() to. and on that note, always return something at the end of a computation to get the thing to be mapped to that new something, as shown below with the "return item"

```
    this.setState(prevState => {
        const newTodos = prevState.todos.map(item => {
            if (item.id == id) {
                item.completed = !item.completed;
                }
            return item
        }) // closing the map()
        return {
            todos: newTodos
        }
    })
}
```

Lifecycle methods are built-in methods that allow developers to have more control over what happens to the state/props of a component or the system when any components appear, update, disappear, much like you have the birth, re-clothing (rendering), and death of humans. Common lifecycle methods include:

```
componentDidMount() {
        // specify what happens when the component first renders
}

render() {
        // specify what happens when you render it to the screen
}

componentDidUpdate() {
        // specify what happens on update, for example if a counter app has a number update, then you might want to change the colour of the number—so change state every time the number increments, putting a conditional on the number having changed from prevState to avoid infinite looping.
}
```

Conditional rendering—you just put an if() statement in the functional component (so you avoid render) that checks whether a boolean prop passed from a parent component is true or not, and return something accordingly. it's good design though, to keep the component that determines the boolean as the same one that implements the conditional rendering property so that you don't have to pass necessary props between the two. This is a good place to use the ternary operator inside the return() and you can even use the && operator to return something if(a condition is met) and nothing otherwise, since, under the hood, the && operator actually just evaluates one input to see if it's true, and if it is, it returns the boolean form of the other input.

remember to bind functions you create within the constructor of class based components!

you can use the native fetch() function to get data from an API passed as an argument; and then add follow-on .then's to say what happens when it receives the data, as below:

```
componentDidMount() { // executes when app is live
    this.setState({loading: true}) // before API call is made, we're loading
    fetch("https://swapi.co/api/people/1") // get raw data
        .then(response => response.json()) // convert to json
        .then(data => {
            this.setState({ // store data in state
                loading: false, // we've stored data from API now and are thus no longer loading
                character: data
            })
        })
}
```

note that there are several other promise functions aside from .then(), like .handleFailure() and .catch(), and more—read the documentation for more on these. also note how much more concise and intuitive this is, than, an AJAX request, and how it requires a more imperative as opposed to declarative way of thinking.

study done showing the programmer's understanding of how a file works diminishes rapidly with the amount of scrolling they have to do in the file—which is why architecting systems using micro services is so important. so a common paradigm is to separate your components into two categories—"smart" and "dumb" where the former is stateful, with all the newly written functions, and the latter simply has the stuff you need to render(). That said, don't stick to this too dogmatically, as Hooks let you do the same thing less arbitrarily.

Building a meme-generator project

To start off, you need to make "index.js" which actually sets everything up, including:

```
import React from "react"
import ReactDOM from "react-dom"
import App from "./App"

ReactDOM.render(<App />, document.getElementById("root"))
```

where App is another (functional) component looking like this:

```
import React from "react"

function App() {
    return (
        <h1>Hello world!</h1>
```

```
    )
}

export default App
```

—

Frameworks change as people work on them and discover more efficient ways of doing things. Version 16.3 deprecated some life-cycle methods and replaced them with some others. The reason that reading docs efficiently is such an important skill is because of how often you're going to be doing it, given the fact that 1) it's how you learn to use a new tech stack, library, or framework 2) it's how you extend your existing knowledge of said frameworks.

updates include: class-based methods that contain other functions. if those are functions rare defined as arrow fucntions, you no longer need to bind them in the class-based component's constructor.

Hooks are a recent addition to react that make functional components able to interact with state and lifecycle, therefore making class-base components less useful. they also make the organisation of the React app cleaner and simpler.

a quick word on object destructuring: you can create variables more neatly via this mechanism—
const { value } = event.target // this means you just created a variable value that captures event.target.value, a technique often used in React forms
const { age } = person // grabs the age of the person object and stores it in a variable called "age"

you can do a similar thing with useState() a new function under the Hooks update (it is a hook itself) that allows you to change state from within the functional component. it's a function that natively returns an array, where the first value is the value of state, and the second is a function.

useState("yes") sets the value, that is, the array[0] of what was returned, to "yes". and the reason that useState() returns an array is because the developer is expected to use array restructuring in this context to extract the value of state, via:

const [value] = useState("yes") where you can extract the first value of the array returned that way, and store its value in, well, value (but you could've named it anything in this circumstance). now you've been able to create state using hooks, and establish that state as the first value in an array. but now we will learn how to change state once it's already been created using the functional property of the useState() hook.

when you destructure your array, you can pull out both a value (of current state), and define the function in array[1], which is the function you create that sets that previous value (changes it), that takes in as an argument the prevCount and returns prevCount + 1, for example, via =>

example of implementation:

```
function App() {
    const [count, setCount] = useState(0)

    function increment() {
        setCount(prevCount => prevCount + 1)
}
    function decrement() {
        setCount(prevCount => prevCount - 1)
}
```

```
    return (
      <div>
        <h1>{count}</h1>
        <button onClick={increment}>Increment</button>
        <button onClick={decrement}>Decrement</button>
      </div>
    )
}
```

because we've lost the functionality of getting at individual state properties by using useState(), if you have separate properties (count, answer…) in state and want to manipulate them, do separate useState() for each—almost as if you have different versions of state you're altering.

```
const [count, setCount] = useState(0)
const [answer, setAnswer] = useState("yes")
```

Just as you have useState() to alter state as a functional component, useEffect() is a hook designed to give functional components access to lifecycle functionality without extending React.Component.

In JSX, if you're adding inline style, do so with {{ }}

useEffect is used whenever you want to introduce some functionality that isn't to do with the core business logic of the component, like hitting an API, waiting (sleep/timeout), manual DOM manipulation, evenListeners, and more. It takes two args, first is a function (you normally make it arrow anonymous) that affects some state property, and the second, and more relevant arg, is when you want it used. Naturally it acts like componentWillMount() and runs anytime there's a change to anything related to the component (itself, props, state, etc.), but you can specify you only want colour of number to change when number increments by setting [num] as the second arg.

there are some times where you have an effect that leaves some change to state/props/the system that you want "cleaned up" when the component unmounts (reset, for example). in these cases, it's best to create just another useEffect() with the intended functionality (setInterval to make a timer, for example) and then leave the second argument as an empty list, in this case, because you don't want to set a new Interval (automatically ticks when started) every time there is a change to the original one (infinite loop) and so you need the second argument different to the first use case of useEffect(), and thus make a new one. the way you cleanup, now, is by returning a function, which react stores as the cleanup function. clearInterval for the above, named, interval would work in this case.