React

A JavaScript library for building user interfaces

# Declarative

React makes it painless to create interactive UIs. Design simple views for each state in your application, and React will efficiently update and render just the right components when your data changes.

Declarative views make your code more predictable and easier to debug.

# Component-Based

Build encapsulated components that manage their own state, then compose them to make complex UIs.

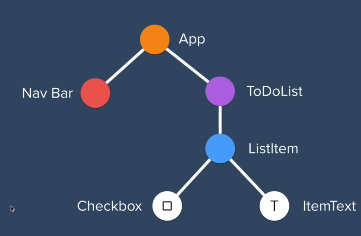
Since component logic is written in JavaScript instead of templates, you can easily pass rich data through your app and keep state out of the DOM.

# Learn Once, Write Anywhere

We don’t make assumptions about the rest of your technology stack, so you can develop new features in React without rewriting existing code.

React can also render on the server using Node and power mobile apps using [React Native](https://reactnative.dev/).

A very complex user interface structure is broken into a component tree.



We are basically creating our very own components.

React combines HTML, CSS and JS.

When a new data comes, the changes are seen and then only those changes are rendered by React. This process is called diffing.

# JSX

import React from "react";

import ReactDOM from "react-dom";

ReactDOM.render(

<div>

<h1>Hello World!</h1>

<p>This is a paragraph.</p>

</div>,

document.getElementById("root")

);

React works by creating these JSX files. JSX files have HTML right inside of javascript. HTML is picked up by compiler that is included in React and is converted into Javascript.

This is done by a compiler called Babel which converts all versions of JS that can be understood by all types of browsers. Babel also allows us to use next gen JS.

The render method can only take one entire HTML component.

The better code to perform above method while making use of ES6 methods is:

# Javascript Expressions in JSX & ES6 Template Literals

Expression is a piece of code that resolves into value.

A statement is a piece of code that is just acting onto the values in code but they themselves don’t resolve into values. For example, if statements or while statements.

import React from "react";

import ReactDOM from "react-dom";

const fname = "Mehul";

const lname = "Varshney";

const luckyNumber = 69;

ReactDOM.render(

<div>

<h1>

Hello {fname} {lname}

</h1>

<p>Your lucky number is {luckyNumber} </p>

</div>,

document.getElementById("root")

);

Everything between the backticks is interpreted as a string using the ES6 template literals and those strings are being inserted as JavaScript inside the curly braces from JSX, then it is being inserted as a string into h1 and then that is being inserted as JS code. This is called string interpolation.

# Styling components in React

Components can be styled in React easily. The HTML that is included within the render method and then we can give a class to that HTML. However, just saying <h1 class=’something’>, then we will get an error. This is because at the end that HTML is being decompiled into our JS code and correct attribute for setting classes in JS is className. Then the error goes away.

import React from "react";

import ReactDOM from "react-dom";

ReactDOM.render(

<div>

<h1 className="heading">My Favourite Foods</h1>

<ul>

<li>Bacon</li>

<li>Jamon</li>

<li>Noodles</li>

</ul>

</div>,

document.getElementById("root")

);

Also, the index.html documents in these sandboxes (<https://codesandbox.io/s/jsx-code-challenge-forked-37fgh>) see the JS code as JS only but now it is JSX code. So, inside the index.html we tell the browser that the code is now jsx type when we include the src tag.

<script src="../src/index.js" type="text/jsx"></script>

All the attributes within the JSX code for HTML elements should be written as in camelCase.

For ex. Instead of contenteditable it should be contentEditable.

All HTML elements must be closed.

<img />

# Inline Styling

Inline styling though not recommended by the React documentation is still pretty useful because you can change the style of your elements on the fly according to preference maybe if you want to change the appearance due to some special day.

All inline styles within the components must be entered as a JS object because again the code is going to be converted into JS code at the end of the day.

The correct way to do that will be this:

import React from "react";

import ReactDOM from "react-dom";

ReactDOM.render(

<h1 style={{ color: "red" }}>Hello World!</h1>,

document.getElementById("root")

);

Also css properties must be entered as JS object and thus follow the camelCase:

import React from "react";

import ReactDOM from "react-dom";

const customStyle = {

color: "red",

fontSize: "20px",

border: "1px solid black"

};

customStyle.color="blue";

ReactDOM.render(

<h1 style={customStyle}>Hello World!</h1>,

document.getElementById("root")

);

Here is an example of changing color of heading levels by time:

import React from "react";

import ReactDOM from "react-dom";

let greeting = "";

const timeHours = new Date().getHours();

const customStyle = {

color: ""

};

if (timeHours <= 6) {

greeting = "Good morning";

customStyle.color = "red"

} else if (timeHours <= 12) {

greeting = "Good afternoon";

customStyle.color = "green"

} else if (timeHours <= 18) {

greeting = "Good Evening";

} else {

greeting = "Good night";

customStyle.color = "blue"

}

ReactDOM.render(

<h1 style={customStyle}> {greeting} </h1>,

document.getElementById("root")

);

# React Components

Start by creating a function and that function should be in PascalCase. This allows React to differentiate between custom components that we will be creating and the inbuilt HTML components.

This is the most basic way of achieving it:

import React from "react";

import ReactDOM from "react-dom";

function Heading() {

return <h1>My Favourite Foods</h1>;

}

ReactDOM.render(

<div>

<Heading />

<ul>

<li>Bacon</li>

<li>Jamon</li>

<li>Noodles</li>

</ul>

</div>,

document.getElementById("root")

);

But all components should be saved as a separate .jsx file in the src folder.

import React from 'react';

function Heading() {

return <h1>My Favourite Foods</h1>;

}

*// creating a heading function and then exporting to whichever file needed.*

export default Heading;

And then the index.js consists of:

import React from "react";

import ReactDOM from "react-dom";

import Heading from './heading'

ReactDOM.render(

<div>

<Heading />

<ul>

<li>Bacon</li>

<li>Jamon</li>

<li>Noodles</li>

</ul>

</div>,

document.getElementById("root")

);

Now, the way that React apps are usually structured are they consist of a src folder and then inside the src folder, there is a components folder and then all the components are stored there. On top of that, there is an app.jsx file in which all the components are imported and then app.js is imported into the React app.

Code for sample app.jsx:

import React from "react";

import Heading from "./Heading";

import List from "./List";

function App() {

return (

<div>

<Heading />

<List />

</div>

);

}

export default App;

Code for corresponding index.js:

import React from "react";

import ReactDOM from "react-dom";

import App from "./components/App";

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

# ES6 Import, Export and Modules

Basically we are exporting values out of the JS/JSX files and then importing them inside the main App.jsx file.

Also, multiple functions can be exported from a single file. While importing into App.jsx, we must take care that the default function can be imported as anything however other functions must be same letter for letter.

Import and Export and Modules allows us to split JS code into smaller separate files.

Code for App.js

import React from "react";

import ReactDOM from "react-dom";

import PI, { doublePi, triplePi } from "./math.js";

ReactDOM.render(

<ul>

<li>{PI}</li>

<li>{doublePi()}</li>

<li>{triplePi()}</li>

</ul>,

document.getElementById("root")

);

Code for Math.js

const pi = 3.1415962;

function doublePi() {

return pi \* 2;

}

function triplePi() {

return pi \* 3;

}

export default pi;

export {doublePi, triplePi};

Alternatively we can also import everything at once from a js/jsx file using the wildcard \*

as randomName and then it will return it as on object randomName

The code for sample Index.js will be something like this:

import React from "react";

import ReactDOM from "react-dom";

import \* as pi from "./math.js";

ReactDOM.render(

<ul>

<li>{pi.default}</li>

<li>{pi.doublePi()}</li>

<li>{pi.triplePi()}</li>

</ul>,

document.getElementById("root")

);

# React Props

React Props allow us to use same components again and again and render them with different data each time.

If react components are though as custom HTML elements, then we can define custom attributes. Those attributes are called props (properties) which are arbitrary custom inputs.

The props is an object having various properties that are basically those custom attributes and the value of these properties is equal to attribute value.

Any custom element made earlier can not have normal properties applied to them. For ex. className attribute if applied to custom component will be taken as a property of Card element that wants to be accessed in props.

All HTML elements must have their attributes specified inside the component function itself.

import React from "react";

import ReactDOM from "react-dom";

function Card(props) {

return (

<div>

<h2>{props.name}</h2>

<img src={props.img} alt="avatar-img" />

<p>{props.tel}</p>

<p>{props.email}</p>

</div>

);

}

ReactDOM.render(

<div>

<h1>My Contacts</h1>

<Card

name="Jack"

img="https://pbs.twimg.com/profile\_images/625247595825246208/X3XLea04\_400x400.jpg"

tel="+987 654 321"

email="jack@nowhere.com"

/>

<Card

name="Chuck Norris"

img="https://i.pinimg.com/originals/e3/94/47/e39447de921955826b1e498ccf9a39af.png"

tel="+918 372 574"

email="gmail@chucknorris.com"

/>

</div>,

document.getElementById("root")

);

Whenever more than one time the tag is repeated with same className or some similarities it is a good idea to separate them out into new components and pass properties.

# React DevTools

Renders a tree of all components that have been rendered.

# Passing props to child components

This is a code for rendering image for a contact card.

Code for App.jsx:

import React from "react";

import Card from "./Card";

import contacts from "../contacts";

function App() {

return (

<div>

<h1 className="heading">My Contacts</h1>

<Card

name={contacts[0].name}

img={contacts[0].imgURL}

tel={contacts[0].phone}

email={contacts[0].email}

/>

<Card

name={contacts[1].name}

img={contacts[1].imgURL}

tel={contacts[1].phone}

email={contacts[1].email}

/>

<Card

name={contacts[2].name}

img={contacts[2].imgURL}

tel={contacts[2].phone}

email={contacts[2].email}

/>

</div>

);

}

export default App;

Code for Card.jsx:

import React from "react";

import Avatar from "./Avatar";

import Detail from "./Detail";

function Card(props) {

return (

<div className="card">

<div className="top">

<h2 className="name">{props.name}</h2>

<Avatar img={props.img} />

</div>

<div className="bottom">

<Detail detailInfo={props.tel} />

<Detail detailInfo={props.email} />

</div>

</div>

);

}

export default Card;

Card for Avatar.jsx:

import React from "react";

function Avatar(props) {

return <img className="circle-img" src={props.img} alt="avatar\_img" />;

}

export default Avatar;

Card for Detail.jsx:

import React from "react";

function Detail(props) {

return <p className="info">{props.detailInfo}</p>;

}

export default Detail;

# Mapping Components

Map function is a JS function that is useful for handling arrays. Map function includes functional programming where we our passing in functions in functions.

So what happens in a mapping function is that the map function loops through the array of JS objects and then it calls a function inside the map function itself which does the magic with data is present in JS objects.

And then we need to create the function (that does the magic) and then pass that object as a parameter.

However, each of the JS objects must have a unique key (ID) according to React docs recommendations for React to be able to render components efficiently.

Sample App.jsx:

import React from "react";

import Card from "./Card";

import contacts from "../contacts";

function createCard(contact) {

return (

<Card

key={contact.id} //Key prop must be spelled exactly like this.

name={contact.name}

img={contact.imgURL}

tel={contact.phone}

email={contact.email}

/>

);

}

function App() {

return (

<div>

<h1 className="heading">My Contacts</h1>

{contacts.map(createCard)}

</div>

);

}

export default App;

`key` is not a prop. Trying to access it will result in `undefined` being returned. If you need to access the same value within the child component, you should pass it as a different prop.

# ES6

# Map

*Create a new array by doing something with each item in an array.*

Map is allowing us to loop through an array but it is more concise than forEach prototype for arrays. We can also call an anonymous function inside the map function like:

For ex:

numbers.map(function(){

return numbers \* 2;

})// doubling an array of numbers.

# Filter

*Create a new array by keeping the items that return true.*

For ex:

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.filter(function num(){

return num > 10;

})

console.log(newNumbers)

This would log only 56 & 48.

# Reduce

*Accumulate a value by doing something to each item in an array.*

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.reduce(function (accumulator, currentNumber){

return accumulator += currentNumber

})

This would return the sum of all the numbers in the numbers array. Accumulator is the value to which currentNumber is being added into.

# Find

*find the first item that matches from an array.*

For ex:

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.find(function (num) {

return num > 10;

});

This will return 56.

# FindIndex

*find the index of the first item that matches.*

For ex:

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.findIndex(function (num) {

return num > 10;

});

This will return 1. (numbers[1] is 56 that is why it returns 1.)

# JavaScript Arrow Functions

Allows us to omit the function keyword by placing an equal two sign and angle bracket after parameters parentheses.

For ex:

var numbers = [3, 56, 2, 48, 5];

const newNumbers = numbers.map( (x)=> {

return x \* 2;

});

console.log(newNumbers)

For single parameter, the parentheses can be omitted too and if the function returns only one expression then the return keyword can also be omitted. (Both of these are independent.)

For ex:

const newNumbers = numbers.map( x => x \* 2) // code like this is not great for someone who is just entering into JS or for junior devs either.

# React Conditional Rendering with the Ternary Operator & AND Operator

Checking the value of some particular property and rendering components on the basis of the property.

CONDITION ? DO IF TRUE : DO IF FALSE

import React from "react";

import Login from "./Login";

var isLoggedIn = true;

function App() {

return (

<div className="container">{isLoggedIn ? <h1>Hello</h1> : <Login />}</div>

);

}

export default App;

If we don’t want to show anything when the condition is true or false, we can write null.

An even more simpler way is the use the && AND operator.

The && Operator executes the expression if condition is true and ignores the expression if condition is false.

For ex:

<div className="container">

{currentTime > 18 && <h1>Why are you still working</h1>}

</div>

Now if the time is 6 PM, it will show h1 and nothing if earlier than 6.

# React States

To make app more interactive.

UI = function(State)

Ice and Water are the same thing. But depending onto the state of the temperature, the water/ice will change their appearance. Temperature is the function here. Ice and Water are the components here.

We are declaring on how our User Interface should look depending on the state and this type of programming is called declarative programming.

Imperative programming is a paradigm of computer programming where the program describes steps that change the state of the computer. Unlike declarative programming, which describes "what" a program should accomplish, imperative programming explicitly tells the computer "how" to accomplish it.

This is an example of changing the text decoration through declarative programming:

import React from "react";

var isDone = false;

function App() {

return (

<div>

<p style={isDone ? { textDecoration: "line-through" } : null}>Buy milk</p>

</div>

);

}

export default App;

We can do the same using imperative programming and add event listeners for changing text decoration to strike through.

# React Hooks

Hooks is a new concept but very powerful. They are functions that allow us to hook into the state of our app and read or modify it. One of the most commonly used hooks is useState hook which comes from react module.

When we are writing HTML elements they are not really HTML elements in the sense that they are dependent on the ReactDOM to actually render each of these elements onto the screen.

Hooks are being used to dynamically update the state.

Rules for using Hooks:

1. The hook must be used inside a functional component.

import React, { useState } from "react";

function App() {

const state = useState();

return (

<div>

<div className="container">

<h1>{state[0]}</h1>

<button onClick={increase}>+</button>

</div>

</div>

);

}

The useState() has an array of two items. The first item is the parameter (it is the initial state), and the second item is a function. This is what we get on console.log(state)

*(2) [undefined, ƒ bound dispatchAction()]*

1. 0: undefined
2. ▶

1: *ƒ bound dispatchAction() {}*

* 1. ▶<constructor>: "Function"
     1. name: "Function"

On passing parameters inside the useState function the first item changes from undefined to that parameter.

# useState

const [state, setState] = useState(initialState);

Returns a stateful value, and a function to update it.

During the initial render, the returned state (state) is the same as the value passed as the first argument (initialState).

The setState function is used to update the state. It accepts a new state value and enqueues a re-render of the component.

setState(newState);

During subsequent re-renders, the first value returned by useState will always be the most recent state after applying updates.

# Destructuring in ES6

Destructuring is used for Complex JS structures like Objects and Arrays.

An array can be destructured for ex:

const rgb = [0, 99, 233]

Then we can destructure the array by:

const [red, green, blue] = [0, 99, 233]

The variable name red is mapped to the value of item at the same index at which red is in, here index 0, so value is also 0.

Here we are mapping the first item of animals array to cat, second to dog.

import animals from "./data";

const [cat, dog] = animals;

All the names must be unique inside the file.

If animals has been used once, it can’t be declared again.

Objects can be destructured in this manner.

const { name, sound } = cat;

console.log(name, sound);

The properties are mentioned inside the curly brackets literal.

There is a difference between the destructuring of arrays and objects. The arrays can be mapped to any variable name but the properties inside the object must be same as the properties of the object that is being destructured. For ex: name and sound are the properties inside the cat object so they need to be exactly be the same and not something like say animalName or animalSound. But if we want we can still use the animalName or animalSound by following manner:

const { name: animalName, sound: animalSound } = cat;

If there is no value in the object being destructured then they can be given some default value in this manner:

const {name = "Fluffy", sound = "Puzz"} = cat;

If there is an additional nested object, they can be destructured in in following manner.

const { name, sound, feedingRequirements:{food, water} } = cat;

A function can also be destructured if we know that the function is returning an array or an object.

const [name, makeSound] = useAnimals(cat)

Finally, here is how to destructure an entire file:

const [honda,tesla] = cars;

const {coloursByPopularity:[teslaTopColour],speedStats:{topSpeed: teslaTopSpeed}} = tesla;

const {coloursByPopularity:[hondaTopColour],speedStats:{topSpeed: hondaTopSpeed}} = honda;

From this file:

const cars = [

{

model: "Honda Civic",

*//The top colour refers to the first item in the array below:*

*//i.e. hondaTopColour = "black"*

coloursByPopularity: ["black", "silver"],

speedStats: {

topSpeed: 140,

zeroToSixty: 8.5

}

},

{

model: "Tesla Model 3",

coloursByPopularity: ["red", "white"],

speedStats: {

topSpeed: 150,

zeroToSixty: 3.2

}

}

];

So, for a counter app, here will be the code:

import React, { useState } from "react";

function App() {

const [count, setCount] = useState(0);

function increase() {

setCount(count + 1); // the setCount is the function that is the second item of the array in useState function.

console.log(count)

}

function decrease() {

setCount(count - 1);

}

return (

<div>

<div className="container">

<h1>{count}</h1>

<button onClick={increase}>+</button> // adding event handlers to our button.

<button onClick={decrease}>-</button>

</div>

</div>

);

}

export default App;

# Event Handling

Basically adding functions to trigger such as onMouseOver or onMouseOut and then using useState hook or other hooks to alter appearance or perform other things.

# React Forms

We can make a function to handle an event when the input changes.

function App() {

function handleChange(event){

console.log(event.target.value) // event is the event that triggered the handleChange() function, target is the element that caused the handleChange() function and value is the value of that input.

}

return (

<div className="container">

<h1>Hello </h1>

<input onChange={handleChange} type="text" placeholder="What's your name?" />

<button>Submit</button>

</div>

);

}

In HTML, the elements themselves are responsible for handling their own state. The value attribute for example of input element is handled by the input element only. But in React, it is recommended to set the value to the variable that comes from the event.

<input value={name} onChange={handleChange} type="text" placeholder="What's your name?" />

The input here is called a controlled component.

From React docs:

In HTML, form elements such as <input>, <textarea>, and <select> typically maintain their own state and update it based on user input. In React, mutable state is typically kept in the state property of components, and only updated with setState().

We can combine the two by making the React state be the “single source of truth”. Then the React component that renders a form also controls what happens in that form on subsequent user input. An input form element whose value is controlled by React in this way is called a “controlled component”.

Now usually we have the habit of putting a form elements between the form tag and adding a button with type = “submit”.

So, what we can do for above code is put the entire thing between form elements and then trigger an onSubmit function for form. Here is how to do it:

function handleSubmit(event) {

setName(value);

event.preventDefault(); // the preventDefault prevents the default behaviour of form component to refresh the page.

}

return (

<div className="container">

<h1>Hello {name}</h1>

<form onSubmit={handleSubmit}>

<input

onChange={onChange}

value={value}

type="text"

placeholder="What's your name?"

/>

<button type="submit">Submit</button>

</form>

</div>

);

)

# Class Components v/s Functional Components:

One way or what you might call functional components, look like this.

import React, { useState } from "react";

function FunctionalComponent() {

const [count, setCount] = useState(0);

function increase() {

setCount(count + 1);

}

return (

<div>

<h1>{count}</h1>

<button onClick={increase}>+</button>

</div>

);

}

export default FunctionalComponent;

Instead of splitting individual components into functions, you can also create a class. And the only difference is the keyword, instead of function becomes class. Classes are not called, so they don't have these parentheses. And this class must extend something that comes from the React module something called component. And this turns your app class into a React component class. And in order to render what you want to see inside this component, you have to add your code inside a render method like so for a simple counter app:

import React from "react";

class ClassComponent extends React.Component {

constructor() {

super();

this.state = {

count: 0

};

this.increase = this.increase.bind(this);

}

increase() {

this.setState({ count: this.state.count + 1 });

}

render() {

return (

<div>

<h1>{this.state.count}</h1>

<button onClick={this.increase}>+</button>

</div>

);

}

}

export default ClassComponent;

Now in the past, the main reason why people converted their functional components into class components was because it was required in order to have state.

If you take a look at some of the documentation on state and lifecycle, you'll see that in order to use state, it used to be that you had to convert your functions into a class.

Now using this class component, I'm able to render a <h1> and a button. And when this button gets clicked, it calls a function called increase. And increase will then in turn calla pre-built function called setState which allows us to pass over some new values to the object that is stored inside this property, state.

Now this works exactly the same as what we would have done using hooks but you can see that involves a lot more boilerplate. It also requires binding and it gets pretty complicated when you want to reuse some of your state functionality across different components.

So back in 2018, the React team set about trying to solve this problem and many others. And they came up with the idea of hooks. Now what the React team recommends is that if you're writing new code that you should start using hooks instead of classes because this is a much easier way of managing state.

Now the thing to remember is that you can only use hooks with functional components. You can't use it inside a class component. But they do say that if you already have classes written for a React app, then you can use it alongside.

We can create React apps that are entirely built using functional components instead of having a mix of functional components and then occasionally having to convert our functional components into class components just to be able to have a stateful app. But in the wild, you will occasionally see components built using classes. Most of the cases when people are telling you to write class component, it's to be able to use and set the state of an app which you can achieve in exactly the same way using this useState hook.

# Changing Complex State

Managing the state of JS objects where you might have to retrieve the previous value of object. For ex, in the login form, we have first name and then we also have a last name so it first statefully updates the first name and then also updates the second name while also keeping the first name stateful.

Now using previous knowledge, this can still be implemented in following manner:

function App() {

const [fname, setFName] = useState("");

const [lname, setLName] = useState("");

function onFNameChange(event) {

const firstName = event.target.value;

setFName(firstName);

}

function onLNameChange(event) {

const lastName = event.target.value;

setLName(lastName);

}

return (

<div className="container">

<h1>Hello {fname + " " + lname} </h1>

<form>

<input onChange={onFNameChange} name="fName" placeholder="First Name" />

<input onChange={onLNameChange} name="lName" placeholder="Last Name" />

<button>Submit</button>

</form>

</div>

);

}

However, a lot of this code is repetitive and we should have fName and lName in object rather than having them separately.

So, what we can do is we can make one object called fullName and then declare its initial state in following manner:

const [fullName, setFullName] = useState({

fName: "",

lName: ""

});

This way we can then have a function to handleChange on our inputs and then call setFullName function in the following manner to also retain the previous state of input elements:

import React, { useState } from "react";

function App() {

const [fullName, setFullName] = useState({

fName: "",

lName: ""

});

function handleChange(event) {

const newValue = event.target.value;

const inputName = event.target.name;

setFullName((prevValue) => {

if (inputName === "fName") {

return {

fName: newValue,

lName: prevValue.lName

};

} else if (inputName === "lName") {

return {

fName: prevValue.fName,

lName: newValue

};

}

});

}

return (

<div className="container">

<h1>

Hello {fullName.fName} {fullName.lName}{" "}

</h1>

<form>

<input

value={fullName.fName}

onChange={handleChange}

name="fName"

placeholder="First Name"

/>

<input

value={fullName.lName}

onChange={handleChange}

name="lName"

placeholder="Last Name"

/>

<button>Submit</button>

</form>

</div>

);

}

export default App;

One thing to remember is that we should not try to access anything related to event inside the setState function.

# JS ES6 Spread Operator

const citrus = ["Live", "Lemon", "Orange"]

const fruits = ["Mango","Pomengranate","Apple", ...citrus]

In this line the three dots followed by array is Spread operator. It basically converts the array we specify into individual elements and then push them into the array we are specifying it in.

We can also spread objects like this:

const fullName = {

fName: "James",

lName: "Bond"

};

const user = {

...fullName,

id: 1,

username: "jamesbond007"

};

# Managing a Component Tree

1. We are basically passing states from child components to the parent components.

So for example if we have a todolist app, then we can pass functions from the parent component to child component. And then trigger that passed on function by tapping into props. So when the function is triggered via child component, it is executed inside the parent component.

We can also pass functions from child component to parent components along with some parameters.

These functions must be passed through an anonymous function within the child component. Here is what I mean:

import React, { useState } from "react";

function ToDoItem(props) {

return <li onClick={props.onChecked(props.id)}>{props.name}</li>;

}

export default ToDoItem;

When props.onChecked is going to be written like this then the function will be executed as soon as component renders. But to prevent this we can put it in an anonymous function to trigger it only when it is clicked here for instance.

import React, { useState } from "react";

function ToDoItem(props) {

return <li onClick={()=>{

props.onChecked(props.id)

}}>{props.name}</li>;

}

export default ToDoItem;

For the todolist app code will then look something like this:

[Managing a Component Tree (forked) - CodeSandbox](https://codesandbox.io/s/managing-a-component-tree-forked-pv52b?file=/src/components/App.jsx:417-595) (hyperlink)