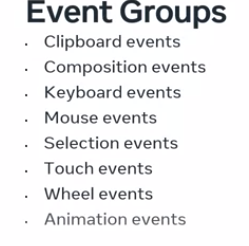
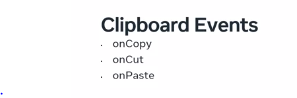
TYPE OF EVENTS

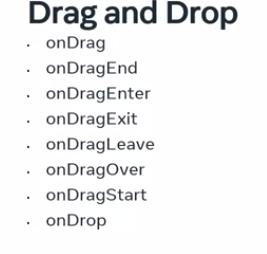


MOUSE EVENT

Example







**Eventful issues**

You’re now aware that React can work with most of the same events found in HTML, although React handles them differently.

This means that you may encounter unfamiliar errors when you run your event-driven React code. However, in this reading, you’ll learn about some of the most common errors associated with events and how you can deal with them.

**Event Errors**

When you work in any programming environment, language, or framework, you are bound to write code that throws errors, for a variety of reasons.

Sometimes it's just about writing the wrong syntax. Other times it's about not thinking of all the possible scenarios and all the possible ways that things can go wrong in your code.

Regardless of what causes them, errors are a part of everyday life for a developer.

The JavaScript language comes with a built-in error handling syntax, the **try...catch** syntax.

Let’s examine an example of an error in JavaScript:

Obviously, you cannot uppercase a number value, and thus, this throws the following error:

To handle this TypeError, you can update the code with a try...catch block that instructs the code to continue running after the error is encountered:

The try-catch block will output some text in the console:

**Oops, you can't uppercase a number. Trying to do it resulted in the following TypeError: 5.toUpperCase is not a function**

It is assumed that if you are taking this course that you are already familiar with how the try...catch syntax works, so I won't go into any details after this quick refresher.

Back to React, here's an example of a simple error in a React component:

In React, an error in the code, such as the one above, will result in the error overlay showing in the app in the browser.

In this specific example, the error would be:

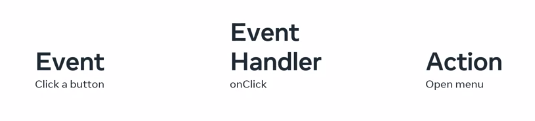
ReferenceError

prop is not defined

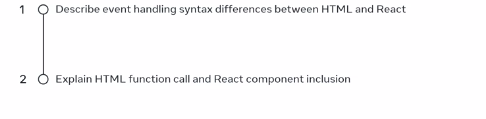
Note: *You can click the X button to close the error overlay.*

Since event-handling errors occur after the UI has already been rendered, all you have to do is use the error-handling mechanism that already exists in JavaScript – that is, you just use the **try...catch** blocks.

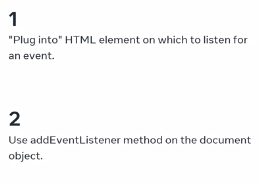
Syntax for Handlers



There are few approaches by the end of lecture







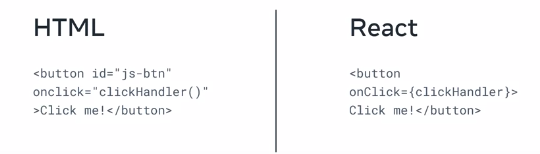


In react the rule is to not manipulate DOM.

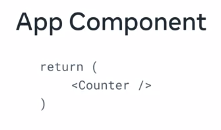
JavaScript – pass an invocation to an event handling fxn as a value to onclick

React – don’t invoke a fxn instead you pass a reference to an event handling fxn without invoking it

Comparison



Example App component





# Event handling and embedded expressions

In this reading, you’ll learn the different ways to embed expressions in event handlers in React:

* With an inline anonymous ES5 function
* With an inline, anonymous ES6 function (an arrow function)
* Using a separate function declaration
* Using a separate function expression

You may find this reading useful as a reference sheet.

For clarity and simplicity: a function will simply console log some words. This will allow you to compare the difference in syntax between these four approaches, while the result of the event handling will always be the same: just some words output to the console.

## ****Handling events using inline anonymous ES5 functions****

This approach allows you to directly pass in an ES5 function declaration as the **onClick** event-handling attribute’s value:

1

2

3

<button onClick={function() {console.log('first example')}}>

    An inline anonymous ES5 function event handler

</button>

Although it's possible to write your click handlers using this syntax, it's not a common approach and you will not find such code very often in React apps.

## ****Handling events using inline anonymous ES6 functions (arrow functions)****

With this approach, you can directly pass in an ES6 function declaration as the **onClick** event-handling attribute’s value:

1

2

3

<button onClick={() => console.log('second example')}>

    An inline anonymous ES6 function event handler

</button>

This approach is much more common then the previous one. If you want to keep all your logic inside the JSX expression assigned to the onClick attribute, use this syntax.

## ****Handling events using separate function declarations****

With this approach, you declare a separate ES5 function declaration, and then you reference its name in the event-handling **onClick** attribute, as follows:

function App() {

    function thirdExample() {

        console.log('third example');

    };

    return (

        <div className="thirdExample">

            <button onClick={thirdExample}>

                using a separate function declaration

            </button>

        </div>

    );

};

export default App;

This syntax makes sense to be used when your onClick logic is too complex to easily fit into an anonymous function. While this example is not really showing this scenario, imagine a function that has, for example, 20 lines of code, and that needs to be ran when the click event is triggered. This is a perfect use-case for a separate function declaration.

## ****Handling events using separate function expressions****

**Tip**: A way to determine if a function is defined as an expression or a declaration is: if it does not start the line with the keyword **function**, then it’s an expression.

In the following example, you’re assigning an anonymous ES6 arrow function to a **const** variable – hence, this is a function expression.

You’re then using this const variable’s name to handle the **onClick** event, so this is an example of handling events using a separate function expression.

function App() {

    const fourthExample = () => console.log('fourth example');

    return (

        <div className="fourthExample">

            <button onClick={fourthExample}>

                using a separate function expression

            </button>

        </div>

  );

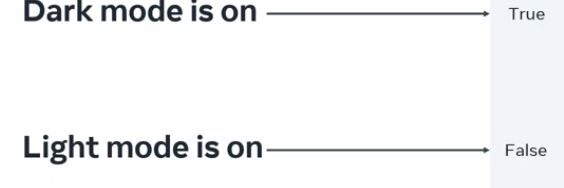
};

export default App;

The syntax in this example is very common in React. It uses arrow functions, but also allows us to handle situations where our separate function expression spans multiple lines of code.

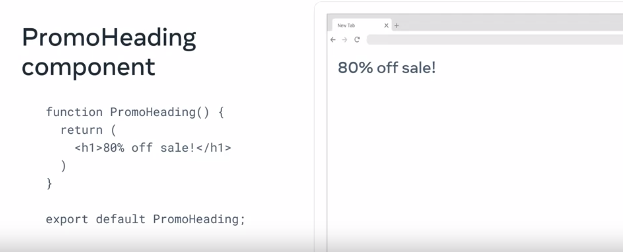
In this reading lesson item, you’ve learned the several types of functions you can use to handle events in React. Some of those are more common than others, but now that you know all the different ways of doing this, you can understand other people’s code more easily, as well as choose the syntax that best suits your given use case, such as a specific company coding style guide.

**User events**



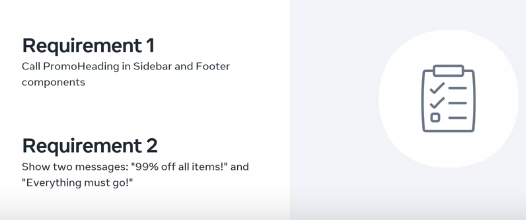
**Parent-child data flow**



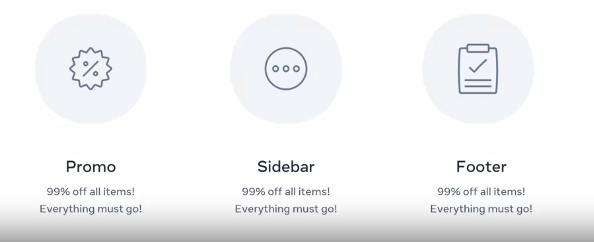


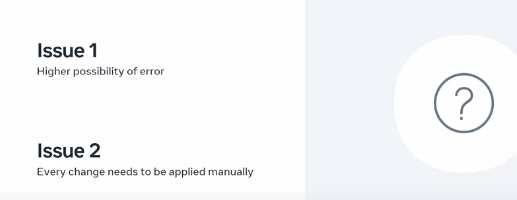


**New requirement**



The approach above won’t work quite well as you to update multiple elements which is against the principle of repeating oneself and leads to possible type error and if there is change you have to do so manually again

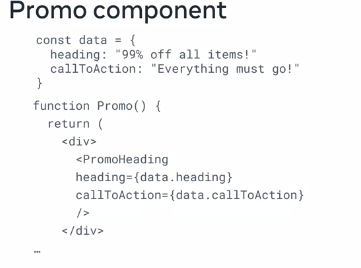




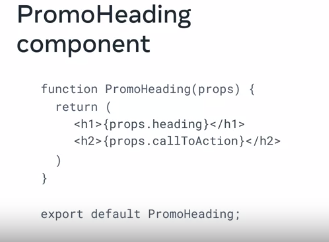
To prevent this you establish a single source

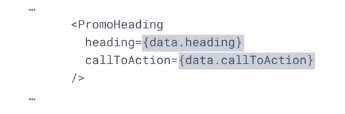


To update the promo above you do this



You delete the h1 and add a new h1





And always know data goes from parent to child in react

# Data flow in React

You’ve just learned how the parent-child relationship can be set up so that data flows from parent to child.

In this reading, you’ll learn how to detail the flow of data from parent to child. You will then learn why code samples need to be clear and concise. Finally, you will explore data flow in greater detail by looking at more examples. This should act as a refresher to knowledge gained in previous courses.

## ****Parent-child data flow****

In React, data flow is a one-way street. Sometimes it's said that the data flow is unidirectional. Put differently, the data in React flows from a parent component to a child component. The data flow starts at the root and can flow to multiple levels of nesting, from the root component (parent component) to the child component, then the grandchild component, and further down the hierarchy.

A React app consists of many components, organized as a component tree. The data flows from the root component to all the components in the tree structure that require this data, using props.

Props are immutable (cannot be changed).

The two main benefits of this unidirectional data flow are that it allows developers to:

1. comprehend the logic of React apps more quickly and
2. simplify the data flow.

Here’s a practical example of this:

Imagine that the parent component passes a prop (name) to the child component. The child component then uses this prop to render the name in the UI.

## ****Parent component:****

function Dog() {

    return (

        <Puppy name="Max" bowlShape="square" bowlStatus="full" />

    );

};

## ****Child component:****

function Puppy(props) {

    return (

        <div>

            {props.name} has <Bowl bowlShape="square" bowlStatus="full" />

        </div>

    );

};

## ****Grandchild component:****

function Bowl(props) {

    return (

        <span>

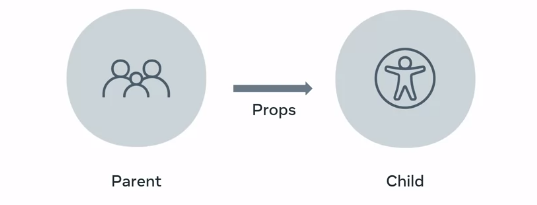
            {props.bowlShape}-shaped bowl, and it's currently {props.bowlStatus}

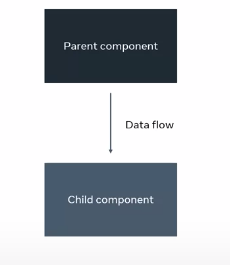
        </span>

    );

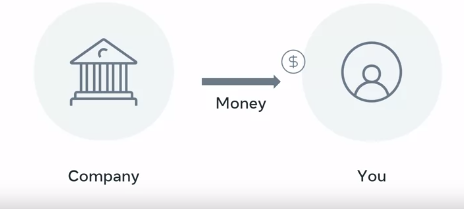
};

Having data move through props in only one direction makes it simpler to understand the logic of how the components interact. If data were moving everywhere, all the time, then it would be much harder to comprehend its logical flow. Any optimization you tried to implement would likely not be as efficient as it could be, especially in modern React.

  
CHILDREN AND DATA



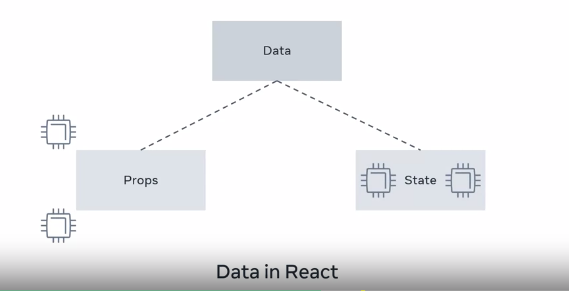
imagine data is money and money is controlled by your employer







**OTHER WAY TO WORK USIG STATE**



State belongs to the data itself and can mutate

HOOKS

As we continue we will encounter complex components. Keeping track of stateful can be hard



Step.1



Step.2





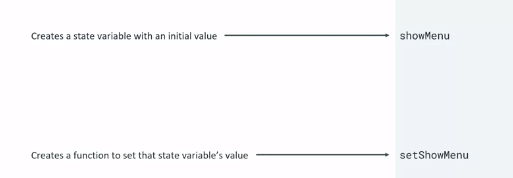






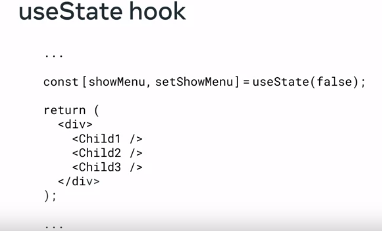
Summary

Calling the useState has 2 things





useState should be called at the top level of your component



Can be used to track any number of data and also build your own hooks

The benefit is the readability and simplicity

# Using hooks

Now that you understand what hooks are in React and have some basic knowledge on the **useState** hook, let’s dive in deeper. In this reading, you will learn how to use hooks in React components and understand the use-cases for the **useState** hook.

Let’s say you have a component with an input text field. The user can type into this text field. The component needs to keep track of what the user types within this text field. You can add state and use the **useState** hook, to hold the string.

As the user keeps typing, the local state that holds the string needs to get updated with the latest text that has been typed.

Let's discuss the below example.

import { useState } from 'react';

export default function InputComponent() {

  const [inputText, setText] = useState('hello');

  function handleChange(e) {

    setText(e.target.value);

  }

  return (

    <>

      <input value={inputText} onChange={handleChange} />

      <p>You typed: {inputText}</p>

      <button onClick={() => setText('hello')}>

        Reset

      </button>

    </>

  );

}

To do this, let's define a React component and call it **InputComponent**. This component renders three things:

* An input text field
* Any text that has been entered into the field
* A Reset button to set the field back to its default state

As the user starts typing within the text field, the current text that was typed is also displayed.



The state variable **inputText** and the **setText** method are used to set the current text that is typed. The **useState** hook is initialized at the beginning of the component.

1

const[inputText, setText] = useState('hello');

By default, the **inputText** will be set to “hello”.

As the user types, the **handleChange** function, reads the latest input value from the browser’s input DOM element, and calls the **setText** function, to update the local state of **inputText**.

function handleChange(e) {

    setText(e.target.value);

};

Finally, clicking the reset button will update the **inputText** back to “hello”.

Isn’t this neat?

Keep in mind that the **inputText** here is local state and is local to the **InputComponent**. This means that outside of this component, **inputText** is unavailable and unknown. In React, state is always referred to the local state of a component.

Hooks also come with a set of rules, that you need to follow while using them. This applies to all React hooks, including the **useState** hook that you just learned.

* You can only call hooks at the top level of your component or your own hooks.
* You cannot call hooks inside loops or conditions.
* You can only call hooks from React functions, and not regular JavaScript functions.

To demonstrate, let’s extend the previous example, to include three input text fields within a single component. This could be a registration form with fields for first name, last name and email.



import { useState } from 'react';

export default function RegisterForm() {

  const [form, setForm] = useState({

    firstName: 'Luke',

    lastName: 'Jones',

    email: 'lukeJones@sculpture.com',

  });

  return (

    <>

      <label>

        First name:

        <input

          value={form.firstName}

          onChange={e => {

            setForm({

              ...form,

              firstName: e.target.value

            });

          }}

        />

      </label>

      <label>

        Last name:

        <input

          value={form.lastName}

          onChange={e => {

            setForm({

              ...form,

              lastName: e.target.value

            });

          }}

        />

      </label>

      <label>

        Email:

        <input

          value={form.email}

          onChange={e => {

Notice that you are using a **form** object to store the state of all three text input field values:

const[form, setForm] =useState({

firstName:'Luke',

lastName:'Jones',

    email:'lukeJones@sculpture.com',

});

You do not need to have three separate state variables in this case, and instead you can consolidate them all together into one **form** object for better readability.

In addition to the **useState** hook, there are other hooks that come in handy such as **useContext**, **useMemo**, **useRef**, etc. When you need to share logic and reuse the same logic across several components, you can extract the logic into a custom hook. Custom hooks offer flexibility and can be used for a wide range of use-cases such as form handling, animation, timers, and many more.

Next, I'll give you an explanation of how the useRef hook works.

## The useRef hook

We use the **useRef** hook to access a child element directly.

When you invoke the **useRef** hook, it will return a **ref** object. The **ref** object has a property named **current**.

function TextInputWithFocusButton() {

  const inputEl = useRef(null);

  const onButtonClick = () => {

    // `current` points to the mounted text input element

    inputEl.current.focus();

  };

  return (

    <>

      <input ref={inputEl} type="text" />

      <button onClick={onButtonClick}>Focus the input</button>

    </>

  );

}

Using the ref attribute on the input element, I can then access the current value and invoke the focus() method on it, thereby focusing the input field.

There are situations where accessing the DOM directly is needed, and this is where the useRef hook comes into play.

## Conclusion

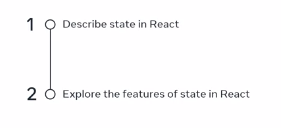
In this reading, you have explored hooks in detail and understand how to use the **useState** hook to maintain state within a component. You also understand the benefits of using hooks within a React component.

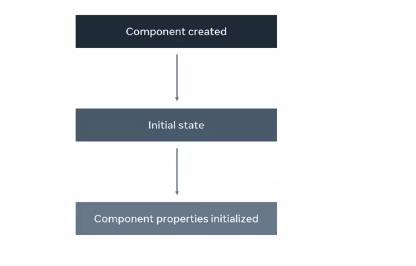
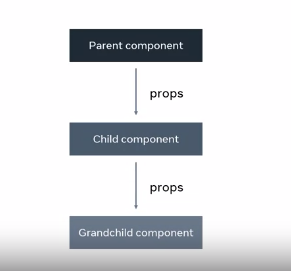
**What Is State**

Create a clock component



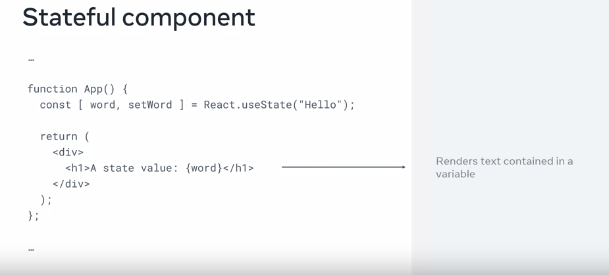
By the end of the lecture you will be able to





Component can be stateful or stateless

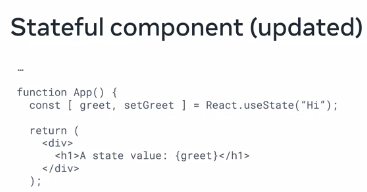
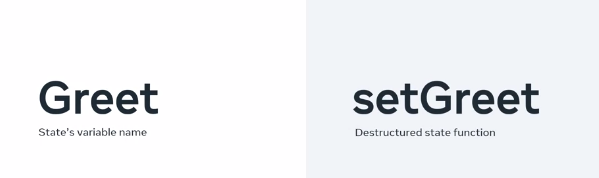




Example for the stateful







OBSERVING STATE

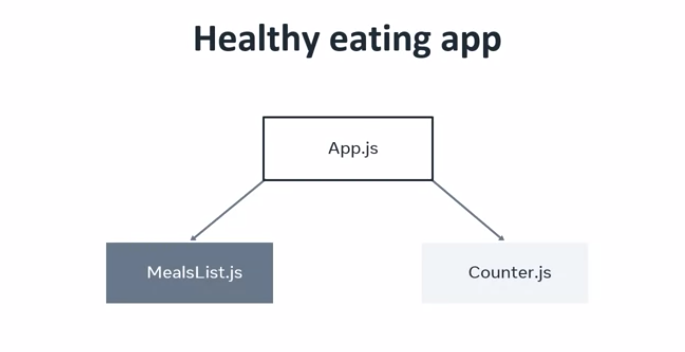
Why do we use state in React?

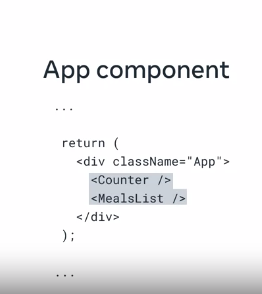
Because it's one way to deal with data in our React apps.State is a powerful tool in Reacts that developers use to manage data that is likely to change in an application. Recall that the state data is internal to the component itself. This allows the components to re-render based on the changes in the states data and present the newest updates to the user.



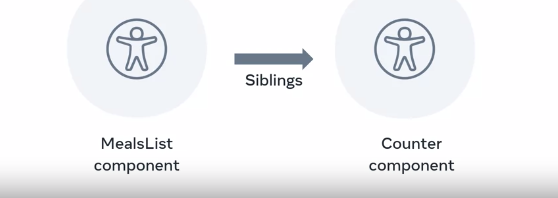
**MANAGING STATE**

E.g a daily meal plan

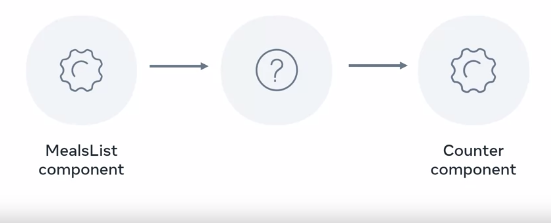




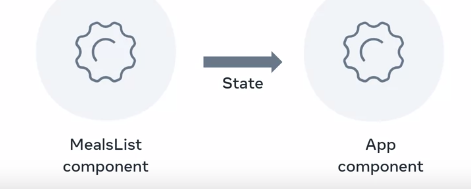
It I good but the meallist and counter are siblings

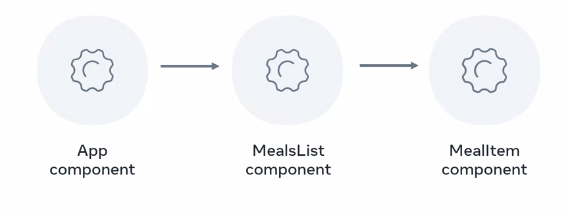


Then the question is

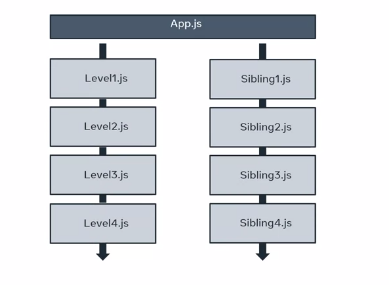


To solve this, you can use the practice known as lifting state up. This means that you move the state from mealsList up to the app components, then you can pass stage three props using the mealsList components as a bridge to the mealItem component. Then you just have to count the data available in the counter components. However, this approach relies on two practices, lifting state up and prop drilling. To understand the issues this can cause, let's recall the current relationship between your app components. The state has now moved up to the app component and my mealsList component becomes a conduits for the state data to be passed to its destination, the mealItem component





The quiz with prop drilling notice that



The other way to phrase is global state. The solution is react context API which cut the middleman there is no need for prop drilling that was you place a file in one depository that when you need it you just fetch

**Prop drilling**

As you’ve learned previously, prop drilling is a situation where you are passing data from a parent to a child component, then to a grandchild component, and so on, until it reaches a more distant component further down the component tree, where this data is required.

Here is a very simple app that focuses on the process of props passing through several components.

Please note that the goal here is not to build an app that would exist in the real world. The goal of this app is to examine the practice of prop drilling, so that you can focus on it and understand it in isolation.

Here is the code for the app:

function Main(props) {

  return <Header msg={props.msg} />;

};

function Header(props) {

  return (

    <div style={{ border: "10px solid whitesmoke" }}>

      <h1>Header here</h1>

      <Wrapper msg={props.msg} />

    </div>

  );

};

function Wrapper(props) {

  return (

    <div style={{ border: "10px solid lightgray" }}>

      <h2>Wrapper here</h2>

      <Button msg={props.msg} />

    </div>

  );

};

function Button(props) {

  return (

    <div style={{ border: "20px solid orange" }}>

      <h3>This is the Button component</h3>

      <button onClick={() => alert(props.msg)}>Click me!</button>

    </div>

  );

};

function App() {

  return (

    <Main

      msg="I passed through the Header and the Wrapper and I reached the Button component"

    />

  );

};

export default App;

This app is simple enough that you should be able to understand it on your own. Let’s address the main points to highlight what is happening in the code above.

The top-most component of this app is the **App** component. The **App** component returns the **Main** component. The **Main** component accepts a single attribute, named **msg**, as in “message”.

At the very top of the app, the **Main** function declares how the **Main** component should behave. The **Main** component is responsible for rendering the **Header** component. **Note that when the Header component is rendered from inside Main, it also receives the msg prop**.

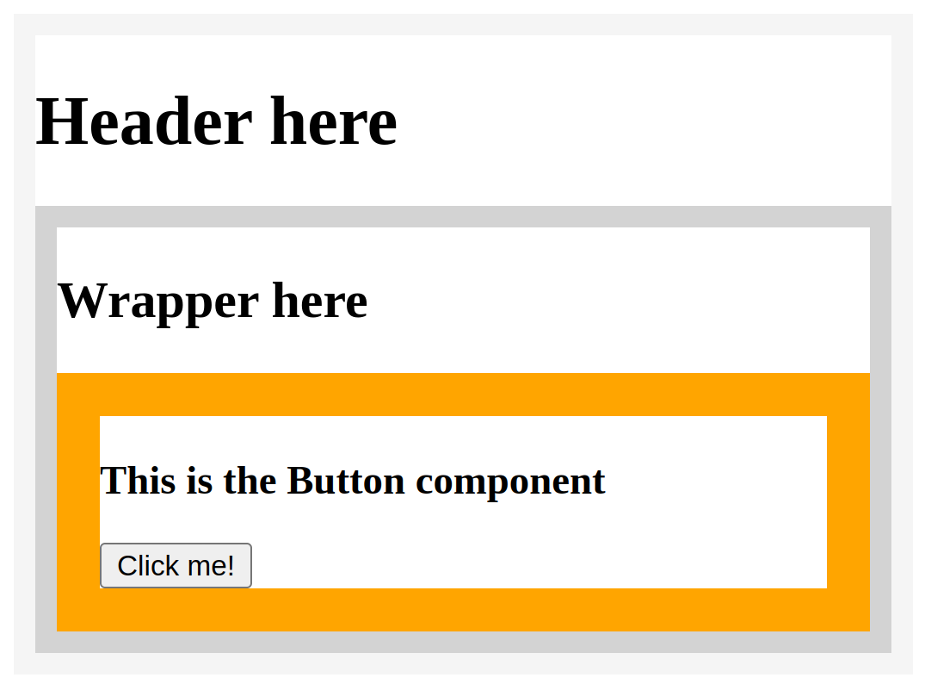
The **Header** component’s function declaration renders an **h1** that reads “Header here”, then another component named **Wrapper**. Note that the naming here is irrelevant – the components **Header** and **Wrapper** are named to make it a bit more like it might appear in a real app – but ultimately, the focus is on having multiple components, rather than describing specific component names properly.

So, the **Header** component’s function declaration has a return statement, which **renders the Wrapper component with the msg prop passed to it**.

In the **Wrapper** component’s function declaration, there’s an **h2** that reads “Wrapper here”, in addition to **the rendering of the Button component, which also receives the msg attribute**.

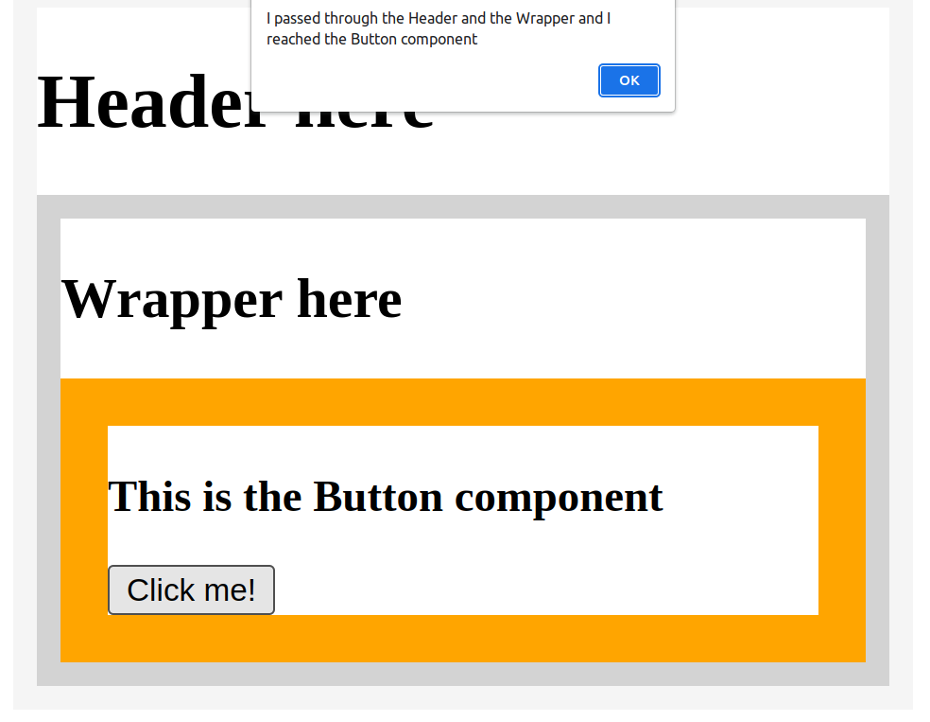
Finally, the **Button** component’s function declaration is coded to receive the props object, then inside of the wrapping **div**, show an **h3**. The **h3** reads “This is the Button component”, and then, under that, there’s a button element with an **onClick** event-handling attribute. This is passed to an arrow function which should alert the string that comes from the **props.msg** prop.

All this code results in the following UI rendered on the screen:



This screenshot illustrates the boundaries of each component. The **Main** component can’t be found in the UI because it’s just rendering the **Header** component. The **Header** component then renders the **Wrapper** component, and the **Wrapper** component then renders the **Button** component.

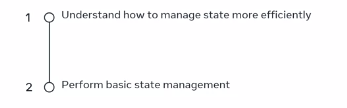
Note that the string that was passed on and on through each of the children component’s props’ objects is not found anywhere. However, it will appear when you click the “Click me!” button, as an alert:



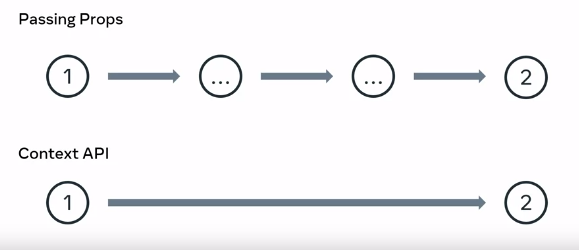
The alert’s message reads “I passed through the Header and the Wrapper and I reached the Button component”.

That’s really all there is to it. Props drilling simply means passing a prop through props objects through several layers of components. The more layers there are, the more repetitive and unnecessary this feels. There are various ways to deal with this, as you’ll learn in the lesson items that follow.

**REACT STATE MANAGEMENT**



D



one is faster than the other

API – application programming interface

