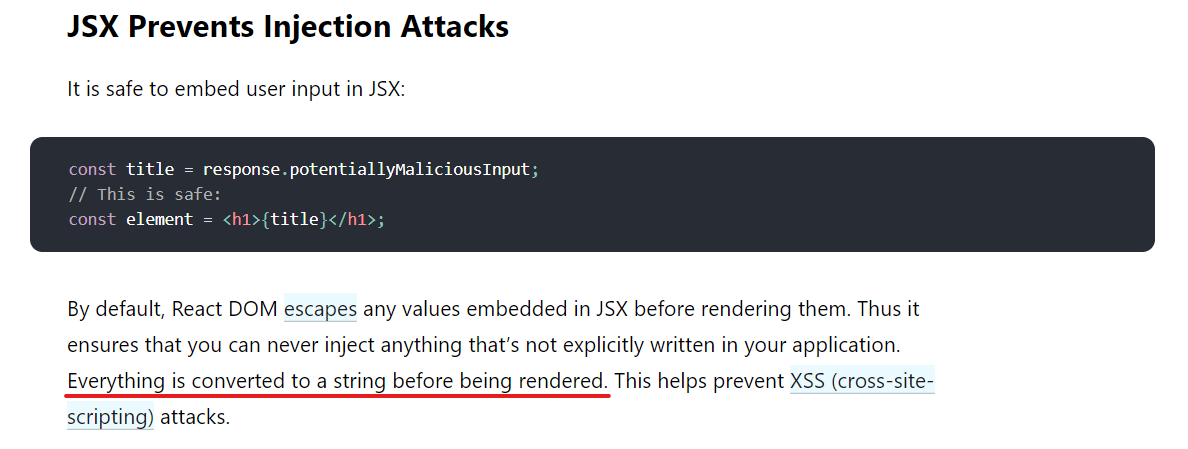
# Main Concepts

## Introducing JSX

### Advantage of JSX:

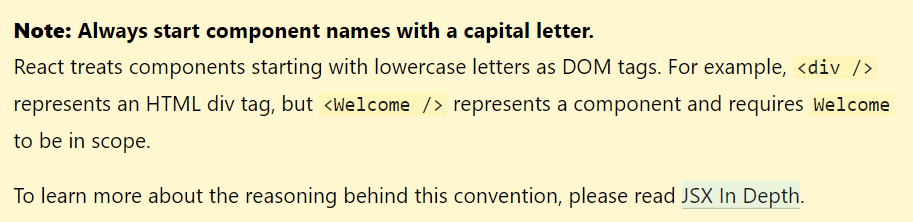
* JSX Prevents Injection Attacks



## Components and Props

### Note:

* Always start component names with a capital letter.



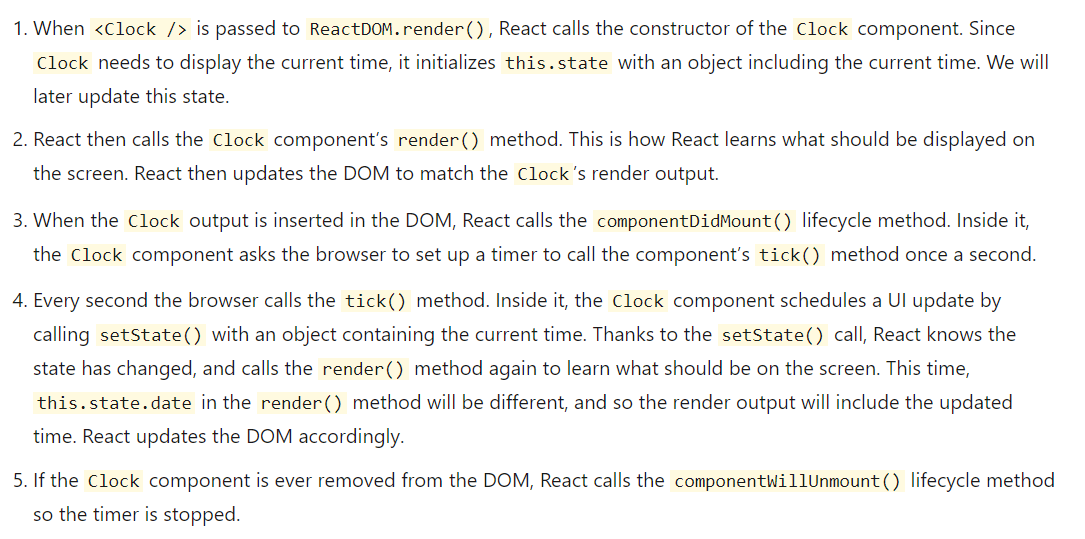
* Check this - <https://reactjs.org/docs/components-and-props.html#props-are-read-only>

## State and Lifecycle

### Class component:



#### Working of a class based component:

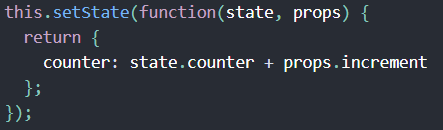


#### Class component notes:

* No need to use let, const and var in a class component. this.<variable or function name>
* this.props, this.state, this.setState() is reserved by react.
* props has to be passed to the parent clas via super(props).
* Only setState() will re-render a class component. Directly changing the state like \*this.state.timestamp = new Date()\* will not re-render the component with new data.
* The only place where you can assign this.state is the constructor.

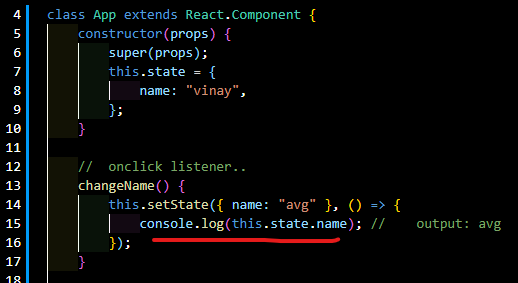
#### Second form of setState():

* Instead of accepting an object, this accepts a callback function.
* This is useful if you want to update the state on top of the previous state value.



#### Getting latest set state in log:

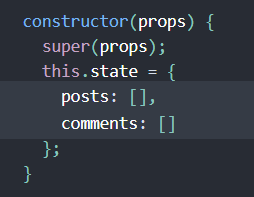
* the setState method takes in a callback as the second param, which gets executed after setting the state.
* So in this callback the latest state after update can be accessed before the Component is rendered.

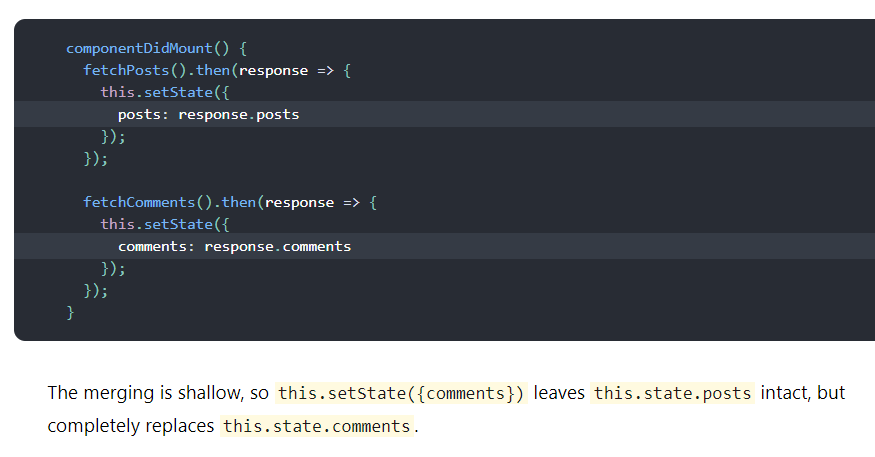


#### In setState() the new updates are merged:

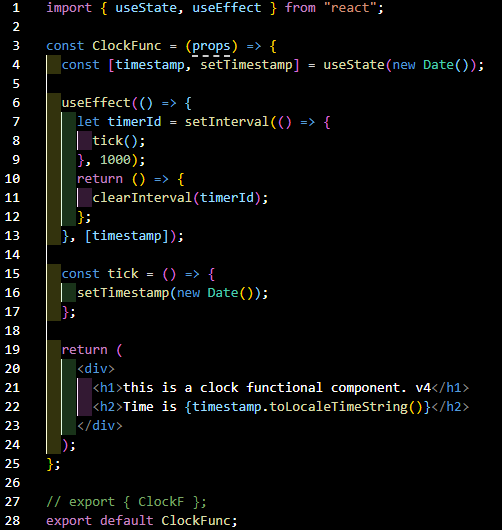
* If there are multiple variables tracked in a state and while updating any one variable, there is no need to list the other variables.
* Meaning, updating any one variable in a state can be done independently.
* If there are posts and comments in state, each of them can be independently updated. There is no need to list the whole state like done in functional component state update(shown below).







### Functional component:



### Data flow in react:

#### Unidirectional/top-down flow of data:

* Any state is always owned by some specific component, and any data or UI derived from that state can only affect components “below” them in the tree.
* A component's state is local and only that component can set/change it.
* A component may choose to pass its state down as props to its child components.
* The child component will not know if this data in the props came from the parent's state or property or hand typed.

## Handling Events

### Functional components:

* Add on<Event> listener to the button/element and pass a method that gets called when event is triggered.
* Define this method that is called on trigger of this event.
* Two ways of passing the method to the on<Event> attribute,
* Passing direct function name inside the flower brackets. Note: \*Passing\* not \*calling\*
* In this case, even if no param is passed, by default react event is available for the called function. Note the defaultEv in line 14, that is not passed in line 33 – react event object is available by default for this kind of event handler call.
* Passing a callback function and **calling** the event method inside it. This is used incase the event data or any other property has to be passed to the called method.



### Class component:

* Add on<Event> listener to the button/element and pass a method that gets called when event is triggered.
* Define this method that is called on trigger of this event.
* Event handler call is same as functional but here it should be called in thisway -> **this.<handlerName>**
* Note:
  + To change the state from a event handler function,
    - It should either be called inside a callback function. This syntax ensures `this` is bound within eventHandler function.
    - Or \*this\* has to be bound to the eventHandler function in the constructor itself. (line 8 below)
* Also check this note about passing params to event handler -> https://reactjs.org/docs/handling-events.html#passing-arguments-to-event-handlers

import React from "react";

class Event1Class extends React.Component {

  constructor(props) {

    super(props);

    this.state = {

      switchState: true,

    };

    this.toggleSwitchState1 = this.toggleSwitchState1.bind(this);

  }

  onClickHandler(e) {

    console.log(e);

    console.log(e.nativeEvent);

    console.log("button clicked from class component.");

  }

  toggleSwitchState1() {

    this.setState({

      switchState: !this.state.switchState,

    });

  }

  toggleSwitchState2() {

    console.log("different method.");

    this.setState({

      switchState: !this.state.switchState,

    });

  }

  render() {

    return (

      <div>

        <h1>Click this class button</h1>

        <button onClick={this.onClickHandler}>Click me.</button>

        <h1>{this.state.switchState ? "On" : "Off"}</h1>

        <button onClick={this.toggleSwitchState1}>Toggle 1</button>

        <h1>{this.state.switchState ? "true" : "fase"}</h1>

        <button onClick={this.toggleSwitchState3.bind(this)}>Toggle 2</button>

        <button

          onClick={() => {

            this.toggleSwitchState2();

          }}

        >

          Toggle 2

        </button>

      </div>

    );

  }

}

export default Event1Class;

#### Tags: binding this, this, bind event handler,

## Conditional Rendering

### If-else condition:

* When a component is invoked, it returns a specific html depending upon the set conditions.

function Greeting(props) {

  const isLoggedIn = props.isLoggedIn;

  if (isLoggedIn) {

    return <UserGreeting />;

  }

  return <GuestGreeting />;

}

ReactDOM.render(

  // Try changing to isLoggedIn={true}:

  <Greeting isLoggedIn={false} />,

  document.getElementById('root')

);

### Rendering via a variable:

* Similar to if-else rendering, here the component/html to be rendered is assigned to a variable and that variable is used in the place where it’s required.

const isLoggedIn = this.state.isLoggedIn;

let button;

if (isLoggedIn) {

  button = <LogoutButton onClick={this.handleLogoutClick} />;

} else {

  button = <LoginButton onClick={this.handleLoginClick} />;

}

return (

  <div>

    <Greeting isLoggedIn={isLoggedIn} />

    {button} // this will either show login or logout based on what gets assigned to it above.

  </div>

);

### Inline && operator/short circuit rendering:

* Here, the actual condition is added first followed by, the component/html to be rendered is added as second condition.
* If the first condition is true, it goes to second condition and \*evaluates\* that, meaning - considers it for the view.
* If the first condition is false, it will not evaluate the second condition, meaning – it won’t be sent to the view.
* Note that returning a falsy expression will still cause the element after && to be skipped but will return the falsy expression. Ex: If 0 is passed as condition, then <div>0</div> will be returned by the render method.

return (

  <div>

    <h1>Hello!</h1>

    {unreadMessages.length > 0 &&

      <h2>

        You have {unreadMessages.length} unread messages.

      </h2>

    }

  </div>

);

### Ternary operator rendering:

* A ternary operator can be used to render either this or that component/html inside the jsx.
* Note: if-else cannot be used inside the jsx. This is the way to achieve similar result.

  return (

    <div>

      <h1>conditional render - functional component</h1>

      {isUserLoggedIn ? (

        <UserRender></UserRender>

      ) : (

        <GuestRender user="Mr. guest"></GuestRender>

      )}

      <button onClick={toggleUserState}>Toggle state</button>

    </div>

  );

### Hiding the component:

* If null is returned from any component, nothing will be displayed to the view.
* Returning null from a component’s render method does not affect the firing of the component’s lifecycle methods. For instance componentDidUpdate will still be called.

## Lists and Keys

### Note:

* Use .map to loop via the Iterables. This will return a new array instead of mutating the original array.
* Two ways to list items,
* Use .map directly in the jsx.
* Use .map outside the return and assign it to a variable. Use that variable in the return.

const UserList = () => {

  const [userList, setstate] = useState([

    { name: "vinay1", age: 25, id: 1 },

    { name: "vinay2", age: 26, id: 2 },

    { name: "vinay3", age: 27, id: 3 },

  ]);

  let x = userList.map((user) => { // Assigning to variable and using it in return.

    return <li key={user.id}>{user.name}</li>;

  });

  return (

    <div>

      <h2>Users list v1</h2>

      <ol>{x}</ol>

      {/\* OR \*/}

      <ul>

        {userList.map((user) => { // Direct map loop.

          return <li key={user.id}>{user.name}</li>;

        })}

      </ul>

    </div>

  );

};

### Keys:

* Each html rendered in a loop requires a key to be included in the parent element. This is used to uniquely identify the element, so react can keep track of which items are changed, added or removed.
* A unique id should be used for each item. While using the index as id works, but it is not unique and can cause issues while the list is being modified in any way.
* Key should always be used on the outer most element which is inside the loop.
* Keys used within arrays should be unique among their siblings. However, they don’t need to be globally unique. We can use the same keys when we produce two different arrays

## Forms

### Note:

* onChange listener for input fields and onSubmit for <form> element.

### Controlled component:

* An input form element whose value is controlled by React (via state updates) in this way is called a \*controlled component\*.
* This is useful when the user entered data is required to manipulate other elements in the form at the same time.

### Select:

* Default html behaviour works as is. In the below example, the coconut gets selected on component render.

<select>

  <option value="">Choose one</option>

  <option value="grapefruit">Grapefruit</option>

  <option value="lime">Lime</option>

  <option selected value="coconut">Coconut</option>

  <option value="mango">Mango</option>

</select>

* In React, a \*value\* key can be passed to the select element and the option with that value gets selected.
* This state can come from props, state or where ever.
* On change of the dropdown the state can be updated.

<select value={selectFlavorValue} onChange={handleFlavorChange}>

<option value="">Choose one</option>

<option value="grapefruit">Grapefruit</option>

      <option value="lime">Lime</option>

      <option value="coconut">Coconut</option>

      <option value="mango">Mango</option>

</select>

### Multi select:

* Multi select can be set by passing \*multiple={true}\* attribute to the select.
* For selected values, an array that contains the selected values can be passed to the value attribute.

<select

          multiple={true}

          value={selectMultipleFlavorValue}

          onChange={handleMultipleFlavorChange}

        >

          <option value="">Choose one</option>

          <option value="grapefruit">Grapefruit</option>

          <option value="lime">Lime</option>

          <option value="coconut">Coconut</option>

          <option value="mango">Mango</option>

        </select>

* For deselection of an option – check if the value exist in array, if yes filter it out and update the state,

  const handleMultipleFlavorChange = (e) => {

    if (selectMultipleFlavorValue.includes(e.target.value)) {

      //  deselecting an option.

      let newState = selectMultipleFlavorValue.filter((eachElement) => {

        return eachElement != e.target.value;

      });

      setMultipleFlavorValue(newState);

    } else if (e.target.value) {

      setMultipleFlavorValue([...selectMultipleFlavorValue, e.target.value]);

    }

  };

### Handling multiple inputs in one method:

* Give different names for each input.
* Have a state object with input names as keys.
* Give same handler method for each input.
* On change of any input, get the name and value from the input and update that specific value in state.
* Do note the [] around name in the setInputData call. Check \* Computed property names\* in js\_notes.

const [inputData, setInputData] = useState({

    name: "a",

    username: "b",

    email: "c",

  });

  const handleInputChange = (e) => {

    let name = e.target.name;

    let value = e.target.value;

    setInputData({ ...inputData, [name]: value });

  };

        <label>name</label>

        <input

          type="text"

          name="name"

          value={inputData.name}

          onChange={handleInputChange}

        />

        <br />

        <label>username</label>

        <input

          type="text"

          name="username"

          value={inputData.username}

          onChange={handleInputChange}

        />

### File inputs:

* Because its value is read-only, it is an uncontrolled component in React.

### Controlled Input Null Value:

* If any outside value(other than undefined and null) is passed as value to an input, then that prevents the user from changing that input field.
* Either that variable has to be changed via setState or no truthy value should be provided as value for an input.

  let someVar = 'staticname';

...

        <input

          type="text"

          name="name"

          placeholder="Enter name"

          onChange={handleNameInput}

          // value={inputName}

          value={someVar}

        />

### Input value from uncontrolled components:

* Use ref is used to get the value from DOM.

## Lifting State Up

* This is used when multiple child components need read/write access to the single state data.
* Declare the state in the parent component.
* Declare all the methods that are required to update the state.
* Pass the state data and the methods to update the state as props to the child components.
* Child components can read the existing state data and change the data based on user input by calling the update methods provided by the parent component.
* Basically, the data stays in sync among all the child components.

## Composition vs Inheritance

* **children** is a reserved keyword under props. If any element/text is present within a called component, all that text/elements will be available to the called component via propss.children property.
* No limitations to what you can pass as props in react. string, numbers, objects, arrays, functions, other components, etc.
* To reuse non-UI functionality between components, extract it into a separate JavaScript module. The components may import it and use that function, object, or a class, without extending it.

## Thinking in React

### Step 1: Break The UI Into A Component Hierarchy

### Step 2: Build A Static Version in React

### Step 3: Identify The Minimal (but complete) Representation Of UI State

#### Identify which variables should be states and which should be props

* Is it passed in from a parent via props? If so, it probably isn’t state.
* Does it remain unchanged over time? If so, it probably isn’t state.
* Can you compute it based on any other state or props in your component? If so, it isn’t state.

### Step 4: Identify Where Your State Should Live

#### For each piece of state in your application:

* Identify every component that renders something based on that state.
* Find a common owner component (a single component above all the components that need the state in the hierarchy).
* Either the common owner or another component higher up in the hierarchy should own the state.
* If you can’t find a component where it makes sense to own the state, create a new component solely for holding the state and add it somewhere in the hierarchy above the common owner component.

### Step 5: Add Inverse Data Flow (lifting state up to manipulate parent’s state)

# Hooks

## useState

In functional components:

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

let isCountMoreThan10 = count > 10 ? 'yes' : 'no';

setCount(count + 1);

This can also be used directly in the listener as shown below,

<button onClick={() => setCount(count + 1)}>+</button>

Equivalent in class components: (inside the constructor)

this.state = {

  count: 0

};

let isCountMoreThan10 = this.state.count > 10 ? "yes" : "no";

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

### Second form of state setter in functional components:

Since setter is async, set the state using callback function which ensures that state is set before rendering UI.

setCount((prevCount) => {

  console.log(prevCount);

  return prevCount + 1;

});

### In setState() the new updates are NOT merged as done in class’ setState():

Old state data will have to be passed first followed by the new data that needs to be changed.

const [count, setCount] = useState({});

setCount(prevState => {

  // Object.assign would also work

  return {...prevState, ...updatedValues};

});

### Initial state can be taken from computation (lazy initial state):

const [state, setState] = useState(() => {

  const initialState = someExpensiveComputation(props);

  return initialState;

});

## useEffect

* This is used to run some additional code after React has updated the DOM(rendered the component). Network requests, manual DOM mutations, and logging etc.
* Multiple useEffects can be used within a functional component.
* In class components:

  componentDidMount() {

    console.log("running effects");

    document.title = `You clicked ${count} times`;

  }

  componentDidUpdate() {

    console.log("running effects on update of state");

    document.title = `You clicked ${count} times`;

  }

  componentWillUnmount() {

    console.log("cleaning up previous stuff");

  }

* In functional components:

  useEffect(() => {

    console.log("running effects");

    document.title = `You clicked ${count} times`;

    return () => {

      console.log("cleaning up previous stuff");

    };

  });

* Class lifecycle methods often contain unrelated logic, but related logic gets broken up into several methods.
* document.title is split between componentDidMount and componentDidUpdate. The subscription logic is also spread between componentDidMount and componentWillUnmount. And componentDidMount contains code for both tasks.
* This creates duplication of code as seen below.

  componentDidMount() {

    document.title = `You clicked ${this.state.count} times`;

    ChatAPI.subscribeToFriendStatus(

      this.props.friend.id,

      this.handleStatusChange

    );

  }

  componentDidUpdate() {

    document.title = `You clicked ${this.state.count} times`;

  }

  componentWillUnmount() {

    ChatAPI.unsubscribeFromFriendStatus(

      this.props.friend.id,

      this.handleStatusChange

    );

  }

* Hooks let us split the code based on what it is doing rather than a lifecycle method name. React will apply every effect used by the component, in the order they were specified.

  useEffect(() => {

    document.title = `You clicked ${count} times`;

  });

  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {

    function handleStatusChange(status) {

      setIsOnline(status.isOnline);

    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

    return () => {

      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);

    };

  });

### Syntax:

useEffect(() => {

console.log("effect statement");

return () => {

console.log("return statement");

};

}, [dependencies\_here]);

* If second param is empty array, then this only runs when the component is first mounted.
* If second param is not passed, then this runs on every component render.
* If any particular variable is passed to the dependency array, then this runs only if that gets changed.
* The return statement is executed first and then the body of the useEffect is run.
* Return is used to clean up all the listeners, timerIDs and stuff like that which were added during user interaction in the previous render.
* On initial rendering, cleanup function is not run.

## Rules of Hooks

* Only Call Hooks at the Top Level
* Don’t call Hooks inside loops, conditions, or nested functions. Use Hooks at the top level of React function, before any early returns. By following this rule, you ensure that Hooks are called in the same order each time a component renders.
* Only call hooks from react functional components and from user defined custom hooks.
* Read this: <https://reactjs.org/docs/hooks-rules.html>
* React uses Object.is comparision to check if the previous state and current state are same or not.
* https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global\_Objects/Object/is#description

## useContext

* Create a context with initial data. Initial state can be empty object also.
* Typically, in a different folder and export it.

import { createContext } from "react";

let initialState = {

  age: 25,

  rank: 97,

};

const UserContext = createContext(initialState);

export default UserContext;

* Import the created context in the main enclosing component.

import ContextTest2 from "./../../Contexts/ContextTest2";

* Pass the initial state object from the context as initial state in the main component.
* Note: \_currentValue will give the initial state of the context.

const [userData, setUserData] = useState(ContextTest2.\_currentValue);

* Pass the state data and the state setter as value in the provider.

<ContextTest2.Provider value={{ userData, setUserData }}> {/\*value here takes only single array or string or number or object \*/}

      <div>

        <ContextChild1></ContextChild1>

        <ContextChild2></ContextChild2>

      </div>

</ContextTest2.Provider>

* In the child components, import the context and pass it to the useContext method to get the value data passed by the provider.

import { useContext } from "react";

import UserContext from "./../../Contexts/ContextTest2";

const ContextChild1 = () => {

  const { userData, setUserData } = useContext(UserContext);

  const increaseRank = () => {

    setUserData({ ...userData, rank: userData.rank - 1 });

  };

  return (

    <div>

      <h3>

        Rank is {userData.rank || "-"} AND age is {userData.age || "-"}

      </h3>

      <button onClick={increaseRank}>Increase Rank</button>

    </div>

  );

};

## useCallback

* Two things are needed to try this out:
* Parent element with a function that does something.

const UseCallbackHook1 = () => {

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

  const [newcount, setnewcount] = useState(0);

  useEffect(() => {

    console.log("parent's use effect body. " + count);

  }, [count]);

  const newMethod = useCallback(() => {

    console.log("this should run only when newcount changes. " + newcount);

}, [newcount]); // This only gets recreated when the newcount changes. Else it remains the same function reference in all renders.

  const changeState = () => {

    setcount((prevCount) => {

      let count1 = prevCount + 1;

      if (count1 % 5 == 0) {

        setnewcount(newcount + 1);

      }

      return count1;

    });

  };

  return (

    <div>

      <h1>useCallback hook 1.</h1>

      <h2>count is {count}</h2>

      <button onClick={changeState}>Click me.</button>

      <UseCallbackChild1 newMethod={newMethod}></UseCallbackChild1>

    </div>

  );

};

* Child component that uses this function in its useEffect.

const UseCallbackChild1 = (props) => {

  useEffect(() => {

    props.newMethod();

}, [props.newMethod]); // The new method is called only when a new instance of it is created - which happens if useCallback is not wrapping the method.

  return (

    <div>

      <h2>child 1.</h2>

    </div>

  );

};

* Only when the “new count” variable changes, a new instance of that method will be created.
* If useCallback is removed, it gets logged on every state change. If callback is wrapping the method, it gets logged only when newcount changes.

## useRef

### Uses:

* Similar to useState, difference being it doesn’t re-render the component when changed, but keeps the value persistent during the component life cycle.

const UseRefHook1 = () => {

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

  const refCount = useRef(0); // initial value set to 0

  useEffect(() => {

    console.log("state updated " + count + " times.");

  });

  return (

    <div>

      <h2>useRef hook 1.</h2>

      <h3>count is {count}</h3>

      <button

        onClick={() => {

          setcount((prevCount) => {

            return prevCount + 1; // update state on state button click. component re-renders and runs useEffect.

          });

        }}

      >

        Increase state count

      </button>

      <h3>ref count is {refCount.current}</h3>

      <button

        onClick={() => {

          refCount.current += 5; // update state on useRef button click. No change in UI but the value is updated. When state is changed after this, the new ref count is shown in UI.

          console.log("ref count is " + refCount.current);

        }}

      >

        Increase useRef count

      </button>

    </div>

  );

};

* Used to select DOM elements

const UseRefHook2 = () => {

  const refElement = useRef();

  const refElement2 = useRef();

  useEffect(() => {

    console.log(refElement);

    console.log(refElement2);

  });

  const focusPlease = () => {

    refElement.current.placeholder = "Type name here..";

    refElement.current.focus();

    refElement2.current.placeholder = "Type description here..";

  };

  return (

    <div>

      <input ref={refElement} type="text" /> <br />

      <textarea ref={refElement2} type="text" cols={40} rows={5} />

      <h2>useRef hook 2.</h2>

      <button onClick={focusPlease}>Focus on input</button>

    </div>

  );

};

* Used to store previous state data

const UseRefHook3 = () => {

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

  const prevCount = useRef(0);

  return (

    <div>

      <h2>useRef hook 3.</h2>

      <h3>

        count is {count}, previous count was {prevCount.current}

      </h3>

      <button

        onClick={() => {

          prevCount.current = count;

          setcount(count + 1);

        }}

      >

        Increase count

      </button>

    </div>

  );

};

## useReducer

### Implementing steps:

* Have an initial state data

const initialState = { count: 0 };

* Create a reducer function that takes in the state and the dispatch data(action)

const reducer1 = (state, action) => {

  switch (action.type) {

    case reducerActions.INCREMENT\_COUNT:

      state = doIncrement(state); // Calling external method to modify state based on some logic.

      // state = { ...state, count: state.count + 1 };

      break;

    case reducerActions.DECREMENT\_COUNT:

      state = doDecrement(state);

      // state = { ...state, count: state.count - 1 };

      break;

    default:

      break;

  }

  return state;

};

* Methods that returns the new state based on some logic. To be defined above the reducer.

//  REDUCER CASE METHODS START

const doIncrement = (state) => {

  console.log("incrementor called.");

  state = { ...state, count: state.count + 1 };

  return state;

};

const doDecrement = (state) => {

  console.log("decrementor called.");

  state = { ...state, count: state.count - 1 };

  return state;

};

//  REDUCER CASE METHODS END

* Pass the reducer function in the useReducer call along with the initial state.
* This returns the latest state along with a action dispatcher that is used to dispatch events which modifies the state accordingly.

const [state, dispatch] = useReducer(reducer1, initialState);

* Call the dispatch method and pass the action type on click of buttons or what ever.

<button

  onClick={() => {

    dispatch({

      type: reducerActions.INCREMENT\_COUNT,

      extraDataUsuallyCalledPayload: { stuff: "yes" },

    });

  }}

>Increment</button>

### Summary:

* On click, a dispatcher is called and the action type and any additional data is passed.
* Based on the action type passed, the reducer will modify the state and returns it and the state will get updated.
* Reducer can also call external methods and pass the state to it, so it can give the new modified state based on more logics that has to be updated.

## useMemo

Only run a certain method if the depending variables change.

<https://www.youtube.com/watch?v=THL1OPn72vo>