ReactJS Learning -

State, Events

Self notes, Key concepts to remember

1. all nodes, HTML elements in DOM by default have access to events.

2. all events have their props equivalent to insert. Like- <button onClick></button>.

3. event props takes function as value. onClick= { function}.

4. define the function outside JSX for clean code.

5. function name only inside { }, not function( ) call. react calls it when needed.

6. it's convention to include '...Handler' in event callback function name.

7. simply reassigning a variable through eventHandler and trying to re-render components doesn't work in react.

8. because react goes through all components and render only once.

9. to make react render a component again we have to understant the concept 'state'.

10. we import { useState } hook from react library and call useState( ) function to introduce a special variable, like- useState(props.title).

11. if this variable changes, or so called state changes react re-renders again.

12. useState( ) returns an array of two elements; first is the variable, second is the updating function which changes the variable.

13. array destructuring is used to catch these two elements:

```
const [title, setTitle] = useState(props.title);
```

14. set... in the name is conventionally used, this setTitle( ) function updates the variable: `const clickHandler = ( ) => setTitle( 'Updated' );`

15. updating function isn't for reassigning the variable, it schedules state updates to re-evaluate the component.

16. react key concept: any changes to variable state reflect the change on user interface.

17. only the associated component reflects the change, also multiple instances of the same component don’t get affected.

18. we use const before variable array and its okay, because we never reassign a new value to the variable; react manages the states for us.

19. react remembers the states, it never re-initializes the happened states.

20. so initial value of state variable only matters when the function renders first time.

21. state adds reactivity to our page.

22. onChange is similar to onClick event, we use onInput/onChange to listen to changes made to fields by user input.

23. an e/event object is automatically generated in javascript and react when an event happens: const titleChangeHandler = (event) => console.log(event);

24. event object has nested object target, we are interested in value property

of the target object.

25. value property holds current value at the time of event occurance.

26. we gather all values, combine them into an object when form is submitted.

27. first we store target.value to survive and do nothing with it:

```
const [enteredTitle, setEnteredTitle] = useState(‘ ’);
const titleChangeHandler = (event) => setEnteredTitle(event.target.value);
```

28. we can call setState( ) multiple times to gather all input values, multiple setState( ) calls can live under the same component function.

29. we use empty string to set initial value, by default any input value is string.

30. important key point: we can manage multiple states in one component.

31. multiple states can be written side by side or alternatively in object form:

```
const [enteredTitle, setEnteredTitle] = useState(‘ ’);
const [enteredAmount, setEnteredAmount] = useState(‘ ’);
```

```
const [enteredDate, setEnteredDate] = useState(‘ ’);
```

alternative object approach:

```
const [userInput, setUserInput] = useState( {
                                    enteredTitle: ‘ ’ ,
                                    enteredAmount: ‘ ’ ,
                                    enteredDate: ‘ ’ } );
```

32. but with object approach, it is more responsibility for small project to manage the states when update happens:

```
                        setUserInput( {
                                    ...userInput,
                                    enteredAmount: event.target.value } );
```

33. setSate(‘ ’) or setState({object}) neither approach works when new state depends on previous state.

34. we pass a function parameter to setState( ) when previous state matters.

35. for example: a counter app where every state depends on previous state.

36. we pass prevState parameter to the callback function:

```
                setUserInput( (prevState) =>
                        { enteredAmount: prevState.enteredAmount + 1 }; )
multiple:     setUserInput( (prevState) => {
                            return { ...prevState,
                        enteredAmount: event.target.value };};
```

37. prevState keeps a snapshot of the latest state.

38. prevState callback function returns an object.

39. onSubmit event is used for submitting input form, not onClick/onChange.

40. specialty of onSubmit event is- it can sit under a <form> element and listen to clicks on any <button> element nested inside <form>, with a type = ‘submit’.

41. we attach onSubmit event to a <form> element.

42. by default onSubmit event reloads the page, so we have to use preventDefault( ) method:

```
const submitHandler = event =>
    event.preventDefault( );
```

43. upon submitting we gather all the values into an object:

```
const submitHandler = event => {
    event.preventDefault( );
const expenseData = {
        title: enteredTitle,
        amount: enteredAmount,
        date: enteredDate }; };
```

44. so we took the values from input fields through state variables, finally stored them into expenseData object inside eventHandler function.

45. we need to reset the input fields to empty and state variables to their initial empty ‘ ’ strings.

46. we use ‘two way binding’ to reset these two at once.

47. we reset the state variables to empty strings and set value attribute of <input> elements equal to state variables:

inside submitHandler( ):

setEnteredTitle(‘ ’);

setEnteredAmount(‘ ’);

inside JSX:

<input value={enteredTitle}/>

48. passing data to upper elements via event propagation is called ‘event bubbling’ in javascript.

49. similarly we can set props of components equal to a function value and thus send data upwards.

50. props not only sends data downwards to children, it can send upwards too when we give a function value, we learned it from event functions.

51. we can give props function name only, but execute it inside children

component functions, thus generating value there and sending to parent.

52. say onSaveData is a props function, we use on... in name conventionally for props functions: <ExpenseForm onSaveData /></div

53. props only helps to communicate data between direct child to parent or parent to child, no skipping or to siblings.

54. example of sending data upwards:

say, we need to send data from NewExpense to App:

in App.js :

```
comp. func.( ) {
    const addExpenseHandler = expense =>
        console.log(expense);
return <div>
    <NewExpense onAddExpense={addExpenseHandler}/>
    </div>; }
```

in NewExpense.js :

comp. func.(props) {

expenseData = { *, *, * };

props.onAddExpense(expenseData);

we can check on console expenseData object showing, from App.js

55. usually we generate state/data in one component and need it in some other component.

56. we look for closest parent component which has access to both.

57. we have to ‘lift state up’ to the parent and then send to the other.

58. controlled components: receives value from parent and sends value to parent by two way binding.

59. presentational, dumb vs. smart, stateful components- stateful components have internal states.

60. we counter more dumb components than smart, so they’re not bad.