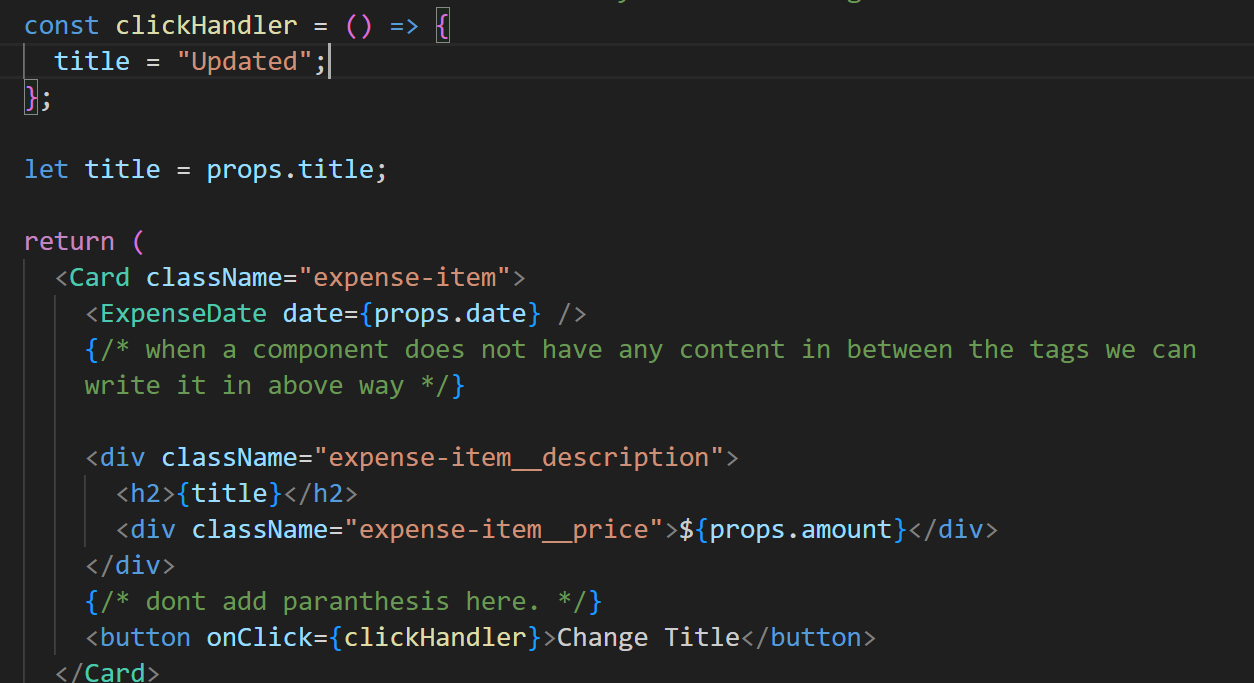
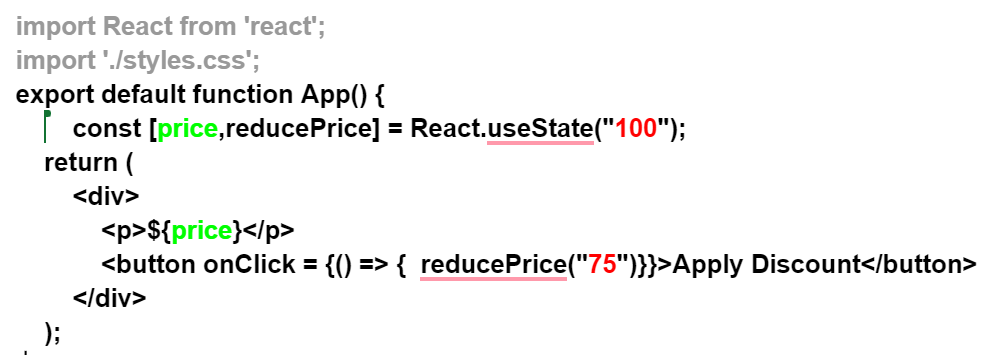
[WDS – State vs Props](https://youtu.be/IYvD9oBCuJI)

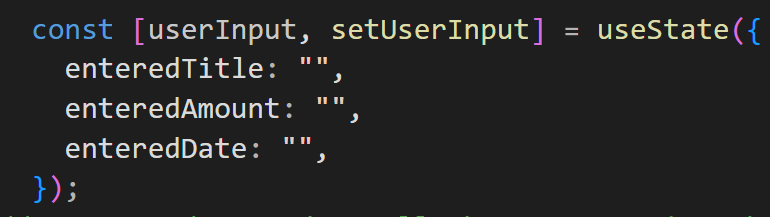
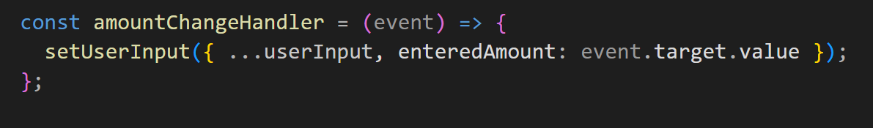
**Using react events and state:**

1. HTMLButtonElement🡪HTMLElement🡪Element🡪Node🡪EventTarget
2. In react, we add the event listener by going to the jsx element, and adding an event listener, starting with ***on***, it adds an event listener. Then we have to define what happens when this event happens by adding a value to it with an equal to sign.
3. All these event handler props, require a function to be passed to it as a value, which is called when the event occurs. We could create an anonymous function right in the html inline, or we can define it somewhere else and then use it, which is more preferable.
4.   
   In this above code, you would assume that, when the button is clicked, the clickHandler function will be updated, and the title will be changed to *Updated*  value, and it would render a new title on screen but it does not happen. WHY?  
   Ans: the new function does not get executed again, and even if it were executed, we would not reach the desired result, because the function would initialise the variables again.
5. Max says, to understand why it did not happen, even though, for sure the clickHandler() was executed, but nothing rendered because:   
   Keep in mind that your component is a function, which can be used as an html tag. By using our components as JSX, by listing our custom components in the root component, they will be called, which returns the jsx code of the custom component, then calls those function which are inside the jsx mode, until there are no more function calls or anything inside the code.  
   So you see that once the function has been called and rendered, it would go inside and call the nested functions and render the final output.  
   So the problem arises when we do the click event and it updates the value in variable, which was rendered in its initial value, and it never repeats the rendering if the click happens in the visited mode of page. So we can not update it by a click, because it needs to be rendered again.
6. So we need to tell react, that something changed, and you need to update the rendering of certain component.
7. This concept is called STATES.
8. 



1. Also, why are we using ***const*** here, if we are updating the value? The variable is initialised inside react modules, in a let variable which we do not see. The const is created inside react.   
   React keeps track of useState for the first time, but not for the next further times, and will only grab the latest update, so the default value is used only for the first time.

**Form inputs**

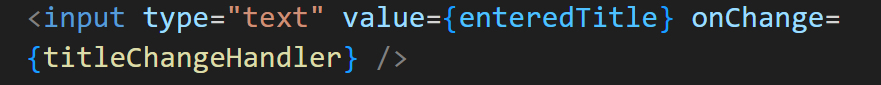
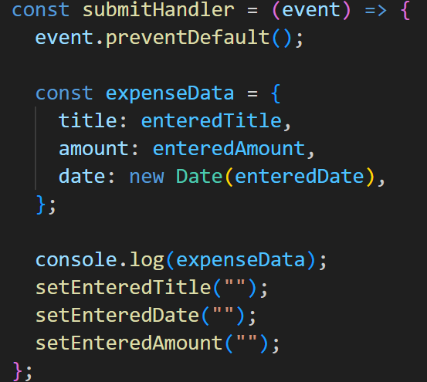
1. Not relevant but 🡪 [event is a keyword for onClick or other event listeners in most browsers.](https://stackoverflow.com/questions/4968194/event-keyword-in-js)
2. 
3. When you are using single useState, which has a JS object with keys and values for different variables involved in the state, you need to update them as shown on the right, which is because if we do not use the rest operator to pick the remaining keys-value pairs, react will update the state without them, and next time if we try to update any key, it will be impossible because we have already lost it.
4. **Drawback with above one state approach?**  The previous use state is a dependency for our new useState to update to copy in existing values, and overwrite one new value, which causes an issue.  
   **Issue:** Since react schedules all the state changes instead of immediately executing them, so we use functions inside the state update function, ie setUserInput method.
5. Solution?   
   **use an alternative way of updating the state.  
     
   setUserIput( (prevState) => {  
     
   return {  
    …prevState, enteredTitle:event.target.value;   
     
   } }**
6. The new approach makes react Guarantee that the latest state is used to operate on updating the changes, and not a scheduled one.

**Form default behaviour:**

When we click a button with type submit in a form, it sends a POST request to the server, and the webpage also refreshes itself. So, we need to prevent this default behaviour, and use simple custom javaScript to decide the behaviour and stop reloading.  
*event.preventDefault()* function is used to disable the default behaviour and use it to submit and handle data on our own.

**Two way binding:**

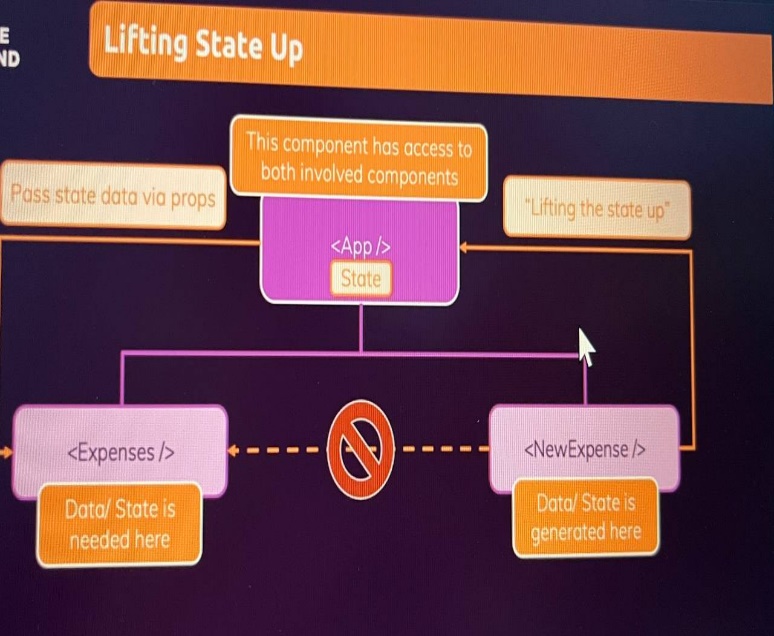
How can we clear the inputs?

For inputs, we don’t just listen to changes, but we can also pass a new value back into the input. All we have to do is add the value attribute to the input element, which sets the internal value property, and we can set it to any new value.

We feed the state back to the input, which was entered by the user, by binding it to the variable we just captured by submitting the form.

It refreshes the form state.

**Child to Parent component Communication? Like we used props to send data from parent to child, like app.js->expenses.js->expenseItem.js. But how do we do the reverse of this?**

1. We can not skip intermediate components in either direction.
2. We can contact bottom up, from child to parent, by using a function as the prop value. And we can do the thing in our function, in the just parent component and child component.
3. **The steps to do so:**
   1. Define the function in parent and pass it as a prop based on some form of functionality you want it to be triggered with, or let the child-parent communication be a default functionality.
   2. Call the function in child where this function [same as the prop name to which the defined function is being called] inside the component function.
   3. When you call the function in child component, do so with **some data** that you need to communicate to the parent as **the arguments of this prop function.**
4. So basically the idea is to give the parent component an ability to listen to the child, through a function call and receiving data upwards – from child to parent.
5. We learnt about **child-to-parent** and **parent-to-child** communication. But there is no way to communicate between the siblings, without the presence of a parent. This is done with the help of a concept called **lift up the state.**  
   In React, "lifting state up" refers to the process of moving the shared state of multiple components to their common parent component. This allows the parent component to manage and control the state, and then pass it down as props to its child components.  
     
   Eg - let's say *ChildComponentA* and *ChildComponentB* both need access to a shared piece of data, such as a counter value. Instead of managing the counter separately in each child component, we can lift the state up to the *ParentComponent* and pass it down as props.
6. Thus the three modes of inter-component communication are through the following concepts:
   1. **Parent to child – usual prop system.**
   2. **Child to parent – send a function as prop from parent to child which receives data back as argument from child to the parent**
   3. **Sibling – lift up the state in parent component and manage state values in the parent component.**

Stateful vs Stateless and Controlled vs Uncontrolled components:

**Stateless vs. Stateful Components:**

**Stateless components**, also known as dumb components or presentational components, are components in a software system that do not maintain any state or data on their own. They receive all the necessary data and behaviour from their parent components or external sources via props (properties) and render them to the user interface. Stateless components are purely focused on rendering the UI based on the given input and do not store any internal state. For **example**, consider a simple button component. It receives information about its label, style, and click event handler as props.

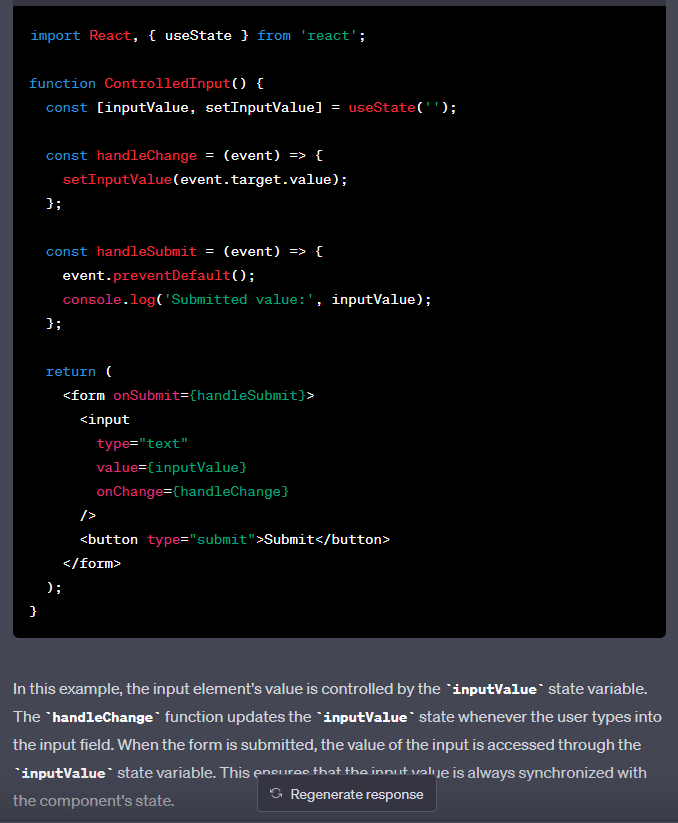
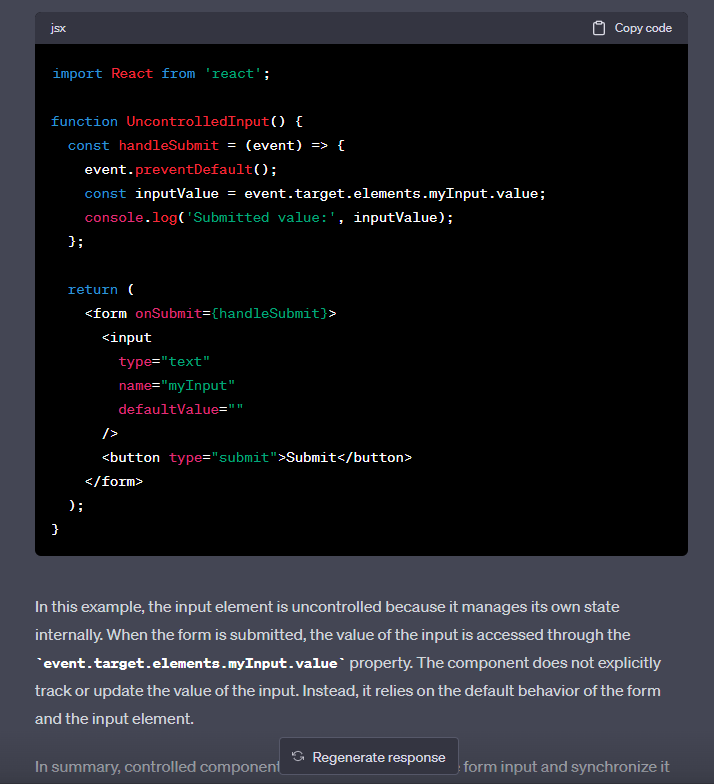
**Stateful components**, are components that manage and maintain their own state or data. They can change their internal state based on user interactions or other factors. Stateful components often encapsulate complex logic and are responsible for managing the state of the application or a specific feature.

For **instance**, a form component might be a stateful component. It maintains the state of the form inputs, such as user input values or selected options. It handles user interactions like input changes or form submission and updates its internal state accordingly.

**Controlled vs. Uncontrolled Components:**

Controlled components and uncontrolled components are terms **commonly used in the context of form inputs** in React.   
**Controlled components** are components where the value of the form input is controlled by the component's state. The component listens for changes in the input and updates its internal state accordingly, which in turn updates the value of the input. When the form is submitted, the component can access the updated value from its state. Controlled components provide a straightforward way to validate and manipulate the input data.

**Uncontrolled components**, allow the form input to manage its own state internally, without being controlled by the component. The component does not keep track of the input's value explicitly.

Controlled Component Example Uncontrolled Component Example  
 

In summary, controlled components manage the state of the form input and synchronize it with the component's state, allowing for more control and validation. Uncontrolled components, on the other hand, rely on the DOM and the form's default behaviour to access the input value when needed, without explicitly managing its state in the component.