**2nd Video**

IN JS WE WROTE:

      <button *onclick*='clicked()'></button>

BUT WE WILL WRITE IN REACT:

      <button *onClick*={clicked}>Click Me</button>

What are the changes?

* onclick 🡪 onClick
* ‘’ 🡪 {}
* functionName() 🡪 functionName

We can directly use the arrow function inside the {} Example:

      <button *onClick*={()=>alert('Button Two Clicked')}>Click This</button>

There is a problem while passing parameters in the function. Like if the function is this:

  function addNumber(num){

    alert(num+5);

  }

And we just call the function like below:

<button *onClick*={addNumber(10)}>Add Five</button>

Then this will create chaos like the function will be called automatically even before clicking the button and also the button wont work. Thus to resolve this use **Arrow Function.**

<button *onClick*={() => addNumber(10)}>AddFive</button>

Now the above functionality will work perfectly.

Notes:

Responding to Events

React lets you add event handlers to your JSX. Eventhandlers are your own functions that will be triggered in response to interactions like clicking, hovering, focusing form inputs etc.

Rules of Adding Event Handler

* Event handler functions:
  + Are usually defined inside your components.
  + Have names that start with handle, followed by the name of the event.
* By convention, it is common to name event handlers as handle followed by the event name:
  + onClick = {handleClick}
  + onMouseEnter= {handleMouseEnter}

e.preventDefault() – by using this we can remove the default behaviour of the forms

Summary

1. You can handle events by passing a function as a prop to an element like <button>.

2. Event handlers must be passed, not called! onClick={handleClick], not onClick={handleClick03.

3. You can define an event handler function separately or inline.

4. Event handlers are defined inside a component, so they can access props.

5. You can declare an event handler in a parent and pass it as a prop to a child.

6. You can define your own event handler props with application-specific names.

7. Explicitly calling an event handler prop from a child handler is a good alternative to

**3rd Video**

**4th Video**

What exactly is state?

* Components often need to change what's on the screen as a result of an interaction.
* Typing into the form should update the input field, clicking "next" on an image carousel should change which image is displayed, clicking "buy" should put a product in the shopping cart.
* Components need to "remember" things: the current input value, the current image, the shopping cart. In React, this kind of component-specific memory is called state.

How to write a useState Hook?

The useState Hook provides those two things:

* A state variable to retain the data between renders.
* A state setter function to update the variable and trigger React to render the component again.

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

count🡪 current state

setCount 🡪 function to update state

0 🡪 initial value

Updating value without state

Here's a component that renders a sculpture image. Clicking the "Next" button should show the next sculpture by changing the index to 1, then 2, and so on. However, this won't work.

Example code of the previous slide: The handleClick event handler is updating a local

variable, index. But two things prevent that change from being visible:

* Local variables don't persist between renders. When React renders this component a second time, it renders it from scratch
* Changes to local variables won't trigger renders. React doesn't realize it needs to render the component again with the new data.

Why useState0 is needed?

To update a component with new data, two things need to happen:

* Retain the data between renders.
* Trigger React to render the component with new data (re-rendering).

Variables do not preserve data between renders and cannot trigger React to render. Hence, useState is needed as it preserves data since it is a memory as well trigger React to render with the help of setter function.

5 use cases of useState()

1. State management

2. Conditional rendering

3. Toggle flags (true/false)

4. Counter

5. Store API data in state

React Hooks

What are react hooks?

Hooks were first introduced in React 16.8. Hooks let you use different React features from your components. You can either use the built-in Hooks or combine them to build your own.

Different Types

* State hooks
* Effect hooks
* Context hooks
* Performance hooks
* Ref hooks
* Others

State Hooks

* State lets a component "remember" information like user input.
* To add state to a component, use one of these Hooks:
  + useState declares a state variable that you can update directly
  + useReducer declares a state variable with the update logic inside a reducer function

Context Hooks

* Context lets a component receive information from distant parents without passing it as props
* useContext reads and subscribes to a context

Ref Hooks

* Refs let a component hold some information that isn't used for rendering, like a DOM node or a timeout ID.
* useRef declares a ref. You can hold any value in it, but most often it's used to hold a DOM node.

Effect Hooks

* Effects let a component connect to and synchronize with external systems. This includes dealing with network, browser DOM, animations, widgets written using www.uchmag.ru a com and other non-React code.
* useEffect connects a component to an external system.

**5th Video**

What are Effects?

There are two types of logic inside React components:

* Rendering code: lives at the top level of your component. This is where you take the props and state, transform them, and return the JSX you want to see on the screen.
* Event handlers: An event handler might update an input field, submit an HTTP POST request to buy a product, or navigate the user to another screen.

Event handlers contain "side effects" (they change the program's state) caused by a specific user action (for example, a button click or typing).

What are Effects?

Consider a ChatRoom component that must connect to the chat server whenever it's visible on the screen.

* Sending a message in the chat is an event because it is directly caused by the user clicking a specific button.
* However, setting up a server connection is an Effect because it should happen no matter which interaction caused the component to appear.

What are side effects in react?

* Not predictable
* Actions which are performed with the "outside world"
* A side effect is performed when we need to reach outside the scope of our current react components to do something
* React component rendering and side-effect logic are independent

Some common side effects –

* Making a request to an API for data from a backend server
* To interact with browser APIs (that is, to use document or window directly) / Manipulating DOM directly
* Using unpredictable timing functions like setTimeout() or setInterval()
* Reading data from local storage

What is useEffect?

useEffect exists –

* To synchronize a component with an external system.
* To provide a way to handle performing these side effects
* Doesn't affect the rendering or performance of the component that it's in
* Performs asynchronous tasks

How to write an Effect

To write an Effect, follow these three steps:

1. Declare an Effect. By default, your Effect will run after every render.

import { useEffect } from 'react';

To declare an Effect in your component, import the useEffect Hook from React

Then, call it at the top level of your component and put some code inside your Effect:

function MyComponent() {

useEffect(() => {

// Code here will run after \*every\* render

});

return <div ;>

}

1. Specify the Effectodependencies. Most Effects should only re-run when needed rather than after every render. For example, a fade-in animation should only trigger when a component appears.
2. Add cleanup if needed. Some Effects need to specify how to stop, undo, or clean up whatever they were doing. For example, "connect" needs "disconnect".

Different types of dependencies in useEffect

1. Run after every render

useEffect(() => {

*// This runs after every render*

});

1. Run only once after Initial Render

useEffect(() => {

*// Run only once after Initial Render*

}, []);

1. This runs on mount \*and also\* if either a or b have changed since the last render

useEffect(1) => {

*/\* This runs on mount \*and also\**

*if either a or b have changed*

*since the last render\*/*

}, [a, b]);

What is the useEffect cleanup function?

Let's look at this scenario:

Imagine the ChatRoom component is a part of a larger app with many different screens. The user starts their journey on the ChatRoom page. The component mounts , i.e. "appears on the screen for the first time" and calls connection.connect0 Then imagine the user navigates to another screen-for example, to the Settings page.

The ChatRoom component unmounts.

Finally, the user clicks Back and ChatRoom mounts again. This would set up a second connection-but the first connection was never destroyed! As the user navigates the app, the connections would

What is the useEffect cleanup function?

To fix the issue, return a cleanup function from your Effect:

useEffect(() =>

const connection = createConnection();

connection. connect();

return ( ) => {

connection.disconnect();

};

}, []);

What is the useEffect cleanup function?

* The useEffect cleanup allows us to tidy up our code before our component unmounts.
* When our code runs and reruns for every render, useEffect also cleans up after itself using the cleanup function.
* The cleanup function prevents memory leaks and removes some unnecessary and unwanted behaviors.
* prevent unwanted behaviors and optimizes application performance.

When should we use the useEffect cleanup?

If our component unmounts before our promise resolves, useEffect will try to update the state (on an unmounted component) and send an error that looks like this:

Warning: Can't perform a React state update on an unmounted component. This is a no-op, but it index.15:27 indicates a memory leak in your application. To fix, cancel all subscriptions and asynchronous tasks in a useEffect cleanup function.

To fix this error, we use the cleanup function to resolve it.

**6th Video**

What does Render means?

* Before your components are displayed on screen, they must be rendered by React.
* "Rendering" means that React is calling your component, which is a function.

Trigger, render and commit

Imagine that your components are cooks in the kitchen. In this scenario, React is the waiter who puts in requests from customers and brings them their orders. This process of requesting and serving UI has three steps:

1. Triggering a render (delivering the guest's order to the kitchen)
2. Rendering the component (preparing the order in the kitchen)
3. Committing to the DOM (placing the order on the table)

Step 1: Trigger a render

There are two reasons for a component to render:

* It's the component's initial render.
* The component's (or one of its ancestors') state has been updated.

Initial render

When your app starts, you need to trigger the initial render. It's done by calling createRoot with the target DOM node, and then calling its render method with your component:

Re-renders when state updates

Once the component has been initially rendered, you can trigger further renders by updating its state with the set function. Updating your component's state automatically queues a render.

You can imagine these as a restaurant guest ordering tea, dessert, and all sorts of things after putting in their first order, depending on the state of their thirst or hunger.

Step 2: React renders your components

After you trigger a render, React calls your components to figure out what to display on screen. "Rendering" is React calling your components.

* On initial render, React will call the root component.
* For subsequent renders, React will call the function component whose state update triggered the render.

This process is recursive: if the updated component returns some other component, React will render that component next, and if that component also turns something, it will render that component next, and so on.

Step 3: React commits changes to the DOM

After rendering (calling) your components, React will modify the DOM.

* For the initial render, React will use the appendChild0 DOM API to put all the DOM nodes it has created on screen.
* For re-renders, React will apply the minimal necessary operations (calculated while rendering!) to make the DOM match the latest rendering output.

React only changes the DOM nodes if there's a difference between renders.

Recap

Any screen update in a React app happens in three steps:

1. Trigger
2. Render
3. Commit

React does not touch the DOM if the rendering result is the same as last time

/\*\*

\* 1. state to hold data

\* 2. use effect with dependency array

\* 3. use fetch to load data

4. data to the state

\* 5. display data on the component

\*/

Thinking in react way

When you build a user interface with React -

Step-1: Break it apart into pieces called components.

Step-2: Describe the different visual states for each of the components.

Step-3: Connect the components together so that the data flows through them.

Start With the Mockup

* 1. You have already a JSON API
  2. You have a mockup from designer