Learn React With This One Project **Link**: Learn React With This One Project

Every React Concept Explained in 12 Minutes
**Link**: Every React Concept Explained in 12 Minutes

## What Is React

- ReactJS is just a frontend tool that makes the process of creating the User Interface much easier.
- It is a JS framework that allows us to write JS in a more effective way to create UI.

## Thinking In React

- We must first know what the final version of our application is going to look like.
- We need to break that down into components and implement each of the components in React.
- Vanilla JS uses **imperative programming** in which we create programs by creating a series of instructions whereas React JS uses **declarative programming** which means that we tell React how our application is going to look and to fill in the details for us.
- In **Imperative Programming**, we think of a result and tell the system the steps to get to the result (Eg - Following a list of instructions and ingredients to make a sandwich)
- In **Declarative Programming**, we think of a result and tell the system the result and ask it to produce that result. (Eg - Going to a shop and asking for a sandwich).

## Todo List Project Setup

- To create a React Project, we need to create a **vite application** using **npm create vite@latest**.
- In that, we need to choose **React** and then **JavaScript + SWC**. We can also choose **JavaScript**. If we choose SWC (Speedy Web Compiler), then it can be compiled faster.
- Then, we need to run **npm i** to install all our dependencies.
- Then, we need to run **npm dev** to run the application.
- The **main.jsx** file hooks up the HTML code in **index.html** with the React code in **main.jsx**.
- The **script** is imported in **index.html**.

- The entire React application (All components) is rendered inside the **root** element of the **index.html** file.

- In React, a function that starts with a capital letter and returns JSX code is called a **component**.

## JSX

- We have to put the JSX code inside the **return** statement in React.js.

- When some HTML keywords such as **for**, **class**, etc clash with identical React keywords, we need to use equivalents such as **htmlFor**, **className**, etc.

- We can only return a **single top-level HTML / HTML-Like element** from a component. If we need to have multiple Top-Level elements in that component, then we can enclose all those elements in a **fragment** (<> </>), which is an **empty element**.

- We will also use a declarative approach to update the form. So, instead of telling React how to change the value and when to change the value, **we will change the JSX and tell React the value**.

## React State

- **State** is the memory of a component. It is local to a component instance on the screen.

- useState() is a React **Hook**. They are special functions that're only available while React is rendering.

- The statement **const [newItem, setNewItem] = useState("");** always returns exactly 2 values: newItem and setNewItem. Here, newItem is the **state variable** and setNewItem is the **state setter**. The **state setter** can change the value of the **state variable** and make React re-render the component.

- The value passed in **useState()** is the default value of the **state variable**.

- If we just call **useState("some value");** with a value, then React will keep rendering the entire component into HTML over and over again (Infinite loop). This is because React sees that useState has a value, "some value" and thus, it needs to render the component.

- To prevent this from happening, we need to ensure that React only renders the component when we want it to, so we call useState() only when we want to render the component.

- So, here we use the useState() function to update our input when it changes.

- The **onChange** event fires on every keystroke.

- First, we create an input with 2 attributes, **value** which contains the **state variable**'s current value and an **onChange** event listener that fires on every keystroke and calls the **state setter** inside which we have passed the current input. The **state setter** takes the current input and assigns it to the **state variable**, and after doing that renders the component. All instances of the **state setter** will have the same value throughout the component.

- The **e.target.value** value passed inside the **state setter** is the current input value from the **input**.

## TODOS LOGIC

- In order to add an item to the list, we need to add an event listener to the **submit** attribute of the form. This is the **onSubmit** event listener.

- The **onSubmit** event listener has a handler function called **handleSubmit()** which takes an event e. Inside this handler function, we have the **e.preventDefault()** which prevents the form from submitting by default.

- If we want to use the current value of the **state variable**, then we need to pass it as an argument in a function. This is illustrated by the following example. This example uses **const [todos, setTodos] = useState([ ]);**

- Suppose we have an array and we want to add members to it using the spread operator and we try to do it directly using the **todos** state variable as **[...todos, {id: crypto.randomUUID(), title: newItem, completed: false}];** then what will happen is that this will add just 1 item to the array. This is because the value of **todos** is the value that we rendered on our last render, which in this case is **[ ]**, since that is its default value.

- However, if we try to add another item to that array, then it will overwrite **todos** and still only 1 item will be present. This is because every time we try to use the above statement, **todos** is being set to its default value and the array item is added to that.

- To be able to use **todos' current value**, we need to pass a function like **currentTodos** inside **useState**. This parameter will contain the updated value of **todos**. After this, we need to put the logic of adding the item to **currentTodos**. This logic will be enclosed in a **return** statement. This logic will always allow us to keep adding items over subsequent renders.

- In the **handleSubmit** function, we call the **setNewItem("")** to ensure that the input clears when we press the Add button. This happens as it sets **newItem** to an empty string.

- JavaScript code can be used in **{ }** inside the JSX code.

- If we're returning an array of HTML (Actually JSX) items, then each of those items should have a unique key. This is because ReactJS needs to know which element in that list we want to change / refer to.

- Each **<li>** in the **list** class has the **onChange** event set to the **toggleTodo(todo.id, e.target.checked)** function which takes the todo's id (**<li>'s** id) and checked value.

- In the **toggleTodo** function, we use the **setTodos** function with the **currentTodos** argument. Here, we use **map** to check **if a todo we want to check is checked**, if it isn't it gets checked by creating a new state object with **completed** added to it (This is because state variables are immutable and we can mutate them only by creating a new state and not directly). Otherwise, the todo is returned as it is.

- The **deleteTodos(id)** function takes the id id of the todo to be deleted and then filters out that todo from the list of all todos and returns the filtered list.

- When we assign a function to be called to an event listener, then we should code it as **() => {function_name(argument)}**. Here, the function that is to be called is assigned to the event listener. When the event listener fires, only then is this function called and its result obtained.

- But if we code it as **{function_name(argument)}** , then the function is called and the result of calling that function is assigned to the event listener. This is undesirable and will cause the function to be executed regardless of whether the event listener fires or not.

- We use a short-circuited AND operator (&&) to check the length of the todos list and display "No Todos" if the list length is 0.

## BREAK APP INTO COMPONENTS

- The **NewTodoForm** function has the code with the **input** field and the button.

- It also has the **handleSubmit** function.

- It has the **newItem and setNewItem** states.