**Namaste React**

**Ep05 Let’s get hooked**

**React File structure**

-React doesn’t have opinions on how you put file into folders.

-some developers group their files by features.

-while moving files into folders we need to export it so that we can import it whenever necessary.

Whenever we are writing a component name of the component should always starts with capital letters, because if we are not then react will think that component as a HTML tag.

**There are 2 ways of exporting a file or Component**

1. export default
   1. this is the default way
   2. this means we want to export only one value or component from the file.

[A module is a self-contained unit that can expose assets to other modules using export and acquire assets from other module by using import statement]

* 1. There can be only one default export
  2. Default export is the value that will be imported from the module, if we use simple import statement

e.g. import Title from “./components/Title”

Title is the name that will be given locally to the variable assigned to contain the values, and it doesn’t have to be named like the origin export.

1. Named export
   1. e.g. export const Title = () => {…}
   2. This is a named import with the name “Title”
   3. It can be imported like: import {Title} from “./components/Title”
   4. If we have created a new file which have 2 components and we want to export both of these components then,
      1. we can wrap these components into a single object and can export
      2. or we can export it separately.
   5. If Header.js have two components, Header and Title then, either we can export each component using names or wrap them into a single component and use default export.

When we are using the component in same file, then we don’t have to export

**One way data binding**

const Body = () => {

const searchText = "KFC";

return (

<>

<div>

<input

type="text"

placeholder="Search"

value={searchText}>

</input>

</div>

</>

);

};

-Here in above example, we have a variable “searchText” and if we put that in value attribute of input then the value KFC will go inside input box.

-We are not able to edit the value “KFC” because it is a hardcoded value.

-To change the value in the input box, we need to modify the variable searchText, but in input box if we write something it won’t change the searchText. This is called as **One-way-data-binding**.

**How can we change the value of searchText?**

-Write a onChange method

onChange = {(e) => onChangeInput}

Create a onChangeInput function, it takes a function (which is a callback function) which have a **e** event.

So whenever input is changed this function will be called.

-If we need to maintain a variable that changes itself, then we need to maintain a **React kind** of variable.

**React Variable**

React variable is just like a state variable

Every component in React maintains a state so, we can put some variables onto that state.

Every time we have to create a local variable we can use state in it.

In react, if we want to create a local variable like searchText, we will create it using **useState Hook**

**useState Hook**

**What is Hook?**

Hooks are the new feature introduced in the React 16.8 version. It allows you to use state and other React features without writing a class. Hooks are the functions which "hook into" React state and lifecycle features from function components. **It does not work inside classes**.

**When would I use a Hook?** If you write a function component and realize you need to add some state to it, previously you had to convert it to a class. Now you can use a Hook inside the existing function component. We’re going to do that right now!

In the end hooks are just a normal JS functions.

**Rules of Hooks**

Hooks are similar to JavaScript functions, but you need to follow these two rules when using them. Hooks rule ensures that all the stateful logic in a component is visible in its source code. These rules are:

1. Only call Hooks at the top level

Do not call Hooks inside loops, conditions, or nested functions. Hooks should always be used at the top level of the React functions. This rule ensures that Hooks are called in the same order each time a components renders.

1. Only call Hooks from React functions

You cannot call Hooks from regular JavaScript functions. Instead, you can call Hooks from React function components. Hooks can also be called from custom Hooks.

**useState Hook**

useState hook is the new way of creating variables,

const [searchText]=useState();

if we have to create local variable in react we need to use state variables.

State variables are created using useState hook.

**Whats is State?**

State is a plain JavaScript object used by React to represent an information about the component’s current situation.

It’s managed in the component (just like any variable declared in a function). The difference is while a “normal” variable “disappears” when their function exits, the state variables are preserved by React.

**What is the difference between props and state?**

The main responsibility of a Component is to translate raw data into rich HTML. With that in mind, the props and the state together constitute the raw data that the HTML output derives from.

You could say props + state is the input data for the render() function of a Component, so we need to zoom in and see what each data type represents and where does it come from.

Because we also use Cosmos where props can contain an initial state, getting this straight is crucial.

**Common ground**

Before separating props and state, let's also identify where they overlap.

* Both props and state are plain JS objects
* Both props and state changes trigger a render update
* Both props and state are deterministic. If your Component generates different outputs for the same combination of props and state then you're doing something wrong.

**Does this go inside props or state?**

tl;dr: If a Component needs to alter one of its attributes at some point in time, that attribute should be part of its state, otherwise it should just be a prop for that Component.

**props**

props (short for properties) are a Component's configuration, its options if you may. They are received from above and immutable as far as the Component receiving them is concerned.

A Component cannot change its props, but it is responsible for putting together the props of its child Components.

**state**

The state starts with a default value when a Component mounts and then suffers from mutations in time (mostly generated from user events). It's a serializable\* representation of one point in time—a snapshot.

A Component manages its own state internally, but—besides setting an initial state—has no business fiddling with the state of its children. You could say the state is private.

\* We didn't say props are also serializable because it's pretty common to pass down callback functions through props.

**Changing props and state**

|  |  |  |
| --- | --- | --- |
|  | ***props*** | ***state*** |
| Can get initial value from parent Component? | Yes | Yes |
| Can be changed by parent Component? | Yes | No |
| Can set default values inside Component?\* | Yes | Yes |
| Can change inside Component? | No | Yes |
| Can set initial value for child Components? | Yes | Yes |
| Can change in child Components? | Yes | No |

\* Note that both props and state initial values received from parents override default values defined inside a Component.

**Should this Component have state?**

state is optional. Since state increases complexity and reduces predictability, a Component without state is preferable. Even though you clearly can't do without state in an interactive app, you should avoid having too many Stateful Components.

**Component types**

* **Stateless Component —** Only props, no state. There's not much going on besides the render() function and all their logic revolves around the props they receive. This makes them very easy to follow (and test for that matter). We sometimes call these dumb-as-f\*ck Components (which turns out to be the only way to misuse the F-word in the English language).
* **Stateful Component —** Both props and state. We also call these state managers. They are in charge of client-server communication (XHR, web sockets, etc.), processing data and responding to user events. These sorts of logistics should be encapsulated in a moderate number of Stateful Components, while all visualization and formatting logic should move downstream into as many Stateless Components as possible.

**What is the role of the useState variable?**

It is use to create state variables.

Const [searchText, updateSearchText] = useState();

Here first element searchText is a variable that we need to get updated while execution.

Second element of the array updateSearchText is the function that will update the searchText variable.

Right side useState(“”) here we will give the initial/defailt value of the local state variable searchText. e.g. const [searchText]=useState(“KFC”);

**Two-way binding**

In above example we are reading as well as writing the searchText variable.

**We have local variables then why do we need state variables?**

* Because react has no idea what’s happening to our local variables, so react won’t re-render any updates happening on that variable. Every time, the variable wants to be in sync with the UI we need to use State variables. This is because react keeps track of the state variables.
* Whenever the variable is updated, whole component re-renders again, that is React destroys the component and create it again

Here reconciliation is happening behind the scenes (Diff Algorithm).

* React only renders that updated portion and it is very quick.