**Namaste React**

**Ep03: Laying the Foundation**

**Revision of Ep02**

* Command we used to initialize npm?

npm init

* After giving above command it will create package.json configuration file which will be used by npm to manage dependencies.
* there are 2 dependencies in config file or in general

1. devDependencies: we need it in development env. only.

2. dependencies: these dependencies that are required by our project.

* browserlist: browserlist helps us to transform our code and make our code compatible with different type of browser versions

**Polyfills**

A polyfill is a piece of code (usually JavaScript on the Web) used to provide modern functionality on older browsers that do not natively support it.

Babel is responsible for converting the new source to older version of source code.

What is Babel?

Bavel is JavaScript package, it takes our code and converts it to some other code, here the input code will be the newer version of js and output will be the older version to support old browsers.

Babel is a toolchain that is mainly used to convert ECMAScript 2015+ code into a backwards compatible version of JavaScript in current and older browsers or environments. Here are the main things Babel can do for you:

* Transform syntax
* Polyfill features that are missing in your target environment (through a third-party polyfill such as [core-js](https://github.com/zloirock/core-js))
* Source code transformations (codemods)

Whenever we want build our project, we need to give the command along with the starting point.

npx parcel index.html

npx parcel build index.html

but this is not the feasible way of doing, usually in projects we will build a script in the package.json which runs the command in easy way.

  "scripts": {

    "start": "parcel index.html",

    "build": "parcel build index.html",

  },

After adding the script we can start the build by just npm run start and npm run build command

**Difference between npm and npx**

* npx is exactly same as npm but it is executing a package without downloading it.
* npm commands are used when we want things to be installed in our project
* we can assume that npx = npm run
* shortcut for npm run start is npm start.

Console log removel using parcel.

To remove the console log from our build files we need to use some package named as babel-plugin-transform-remove-console which will remove our console.log.

npm install babel-plugin-transform-remove-console --save-dev

as this command mentioned about dev env we can find this dependency under the devDependencies in package.json file.

Once the package is install we need to configure it to use in our project as below.

* Create config file named as .babelrc
* Add below configuration

*// without options*

{

"plugins": ["transform-remove-console"]

}

*// with options*

{

"plugins": [ ["transform-remove-console",

{ "exclude": [ "error", "warn"] }] ]

}

* Take the build again and you can see the removal of the console messages we configured here.

index.js:1 Warning: Each child in a list should have a unique "key" prop.

To solve the above warning, we need to give keys to every child/sibling in same div element.

e.g.

const heading2 = React.createElement(

  "h1",

  { id: "title1", **key: "t1"** },

  "This is Heading 2 from the react CreateElement Tag"

);

**What is the DOM?**

The Document Object Model (DOM) is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM represents the document as nodes and objects; that way, programming languages can interact with the page.

A web page is a document that can be either displayed in the browser window or as the HTML source. In both cases, it is the same document but the Document Object Model (DOM) representation allows it to be manipulated. As an object-oriented representation of the web page, it can be modified with a scripting language such as JavaScript.

**What is the Virtual DOM?**

The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM. This process is called **reconciliation**.

This approach enables the declarative API of React: You tell React what state you want the UI to be in, and it makes sure the DOM matches that state. This abstracts out the attribute manipulation, event handling, and manual DOM updating that you would otherwise have to use to build your app.

Since “virtual DOM” is more of a pattern than a specific technology, people sometimes say it to mean different things. In React world, the term “virtual DOM” is usually associated with React elements since they are the objects representing the user interface. React, however, also uses internal objects called “fibers” to hold additional information about the component tree. They may also be considered a part of “virtual DOM” implementation in React.

**Why do we need a virtual DOM when there is an actual DOM?**

Well, DOM operations are expensive, and updating the whole DOM on every prop/state change is very inefficient. Here’s how the virtual DOM deals with this inefficiency:

* A component props/state changes
* React triggers a rerender
* React compares the virtual DOM (virtual DOM before the update) with the updated virtual DOM (virtual DOM after update)
* React determines the best possible way to reflect the changes in the UI with minimal operations on the real DOM.

So its virtual DOM helps React update the UI to match the most recent tree.

**Reconciliation**

1. The process of keeping virtual DOM in sync with the real DOM is called reconciliation.
2. Reconciliation is the process through which React updates the DOM. When a component’s state changes, React has to calculate if it is necessary to update the DOM. It does this by creating a virtual DOM and comparing it with the current DOM. In this context, the virtual DOM will contain the new state of the component.

**The Diffing Algorithm**

When diffing two trees, React first compares the two root elements. The behavior is different depending on the types of the root elements.

**Elements Of Different Types**

Whenever the root elements have different types, React will tear down the old tree and build the new tree from scratch. Going from <a> to <img>, or from <Article> to <Comment>, or from <Button> to <div> - any of those will lead to a full rebuild.

When tearing down a tree, old DOM nodes are destroyed. Component instances receive componentWillUnmount(). When building up a new tree, new DOM nodes are inserted into the DOM. Component instances receive UNSAFE\_componentWillMount() and then componentDidMount(). Any state associated with the old tree is lost.

Any components below the root will also get unmounted and have their state destroyed. For example, when diffing:

<div>

<Counter />

</div>

<span>

<Counter />

</span>

This will destroy the old Counter and remount a new one.

Note:This method is considered legacy and you should avoid it in new code: UNSAFE\_componentWillMount()

**DOM Elements Of The Same Type**

When comparing two React DOM elements of the same type, React looks at the attributes of both, keeps the same underlying DOM node, and only updates the changed attributes. For example:

<div className="before" title="stuff" />

<div className="after" title="stuff" />

By comparing these two elements, React knows to only modify the className on the underlying DOM node.

When updating style, React also knows to update only the properties that changed. For example:

<div style={{color: 'red', fontWeight: 'bold'}} />

<div style={{color: 'green', fontWeight: 'bold'}} />

When converting between these two elements, React knows to only modify the color style, not the fontWeight.

After handling the DOM node, React then recurses on the children.

**Component Elements Of The Same Type**

When a component updates, the instance stays the same, so that state is maintained across renders. React updates the props of the underlying component instance to match the new element, and calls UNSAFE\_componentWillReceiveProps(), UNSAFE\_componentWillUpdate() and componentDidUpdate() on the underlying instance.

Next, the render() method is called and the diff algorithm recurses on the previous result and the new result.

Note: These methods are considered legacy and you should avoid them in new code:

UNSAFE\_componentWillUpdate()

UNSAFE\_componentWillReceiveProps()

**Recursing On Children**

By default, when recursing on the children of a DOM node, React just iterates over both lists of children at the same time and generates a mutation whenever there’s a difference.

For example, when adding an element at the end of the children, converting between these two trees works well:

<ul>

<li>first</li>

<li>second</li>

</ul>

<ul>

<li>first</li>

<li>second</li>

<li>third</li>

</ul>

React will match the two <li>first</li> trees, match the two <li>second</li> trees, and then insert the <li>third</li> tree.

If you implement it naively, inserting an element at the beginning has worse performance. For example, converting between these two trees works poorly:

<ul>

<li>Duke</li>

<li>Villanova</li>

</ul>

<ul>

<li>Connecticut</li>

<li>Duke</li>

<li>Villanova</li>

</ul>

React will mutate every child instead of realizing it can keep the <li>Duke</li> and <li>Villanova</li> subtrees intact. This inefficiency can be a problem.

**Keys**

In order to solve this issue, React supports a key attribute. When children have keys, React uses the key to match children in the original tree with children in the subsequent tree. For example, adding a key to our inefficient example above can make the tree conversion efficient:

<ul>

<li key="2015">Duke</li>

<li key="2016">Villanova</li>

</ul>

<ul>

<li key="2014">Connecticut</li>

<li key="2015">Duke</li>

<li key="2016">Villanova</li>

</ul>

Now React knows that the element with key '2014' is the new one, and the elements with the keys '2015' and '2016' have just moved.

In practice, finding a key is usually not hard. The element you are going to display may already have a unique ID, so the key can just come from your data:

<li key={item.id}>{item.name}</li>

When that’s not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. The key only has to be unique among its siblings, not globally unique.

As a last resort, you can pass an item’s index in the array as a key. This can work well if the items are never reordered, but reorders will be slow.

Reorders can also cause issues with component state when indexes are used as keys. Component instances are updated and reused based on their key. If the key is an index, moving an item changes it. As a result, component state for things like uncontrolled inputs can get mixed up and updated in unexpected ways.

**React.createElement()**

whenever we are doing React.createElement() it is doing below.

* React.createElement() give us an object
* this object is converted into the HTML code
* HTML code is put upon the DOM

If we are using so many createElement for creating multiple HTML elements then it will be very difficult read and do the code.

We can reduce this pain by using something known as JSX.

**What is JSX?**

JSX is a html like syntax but it is not html inside javascript.

e.g. const element = <h1>Hello, world!</h1>;

**Difference between HTML and JSX**

1. **You need to return a single parent element in JSX**

One of the major differences between HTML and JSX is that in JSX, you must return a single parent element, or it won't compile.

A lot of developers use <div>...</div>, but a better one that many people use is “fragment”, <>...</> which makes the code more readable.

In HTML, you are free to do whatever you want as you don’t have to return a single parent element.

1. **Use of className instead of class attribute**

In JSX we use className attribute whereas in HTML we use the class attribute. This is because JSX is transpiled into JavaScript and class is a reserved word in JavaScript.

JSX

<div className = "container"></div>

HTML

<div class = "container"></div>

1. **Self-closing tags**

Self-closing tags in JSX must have the forward slash whereas the forward slash is optional in the HTML self-closing tags.

JSX

<img src="#" />

<br/>

HTML

<img src="#" >

<br>

1. **Event listeners**

You need to write all HTML attributes and event listeners in camelCase while writing JSX. So, onclick becomes onClick, onmouseover becomes onMouseOver

**What is Tabindex**

tabindex is a global attribute that allows an HTML element to receive focus. It needs a value of zero or a negative number in order to work in an accessible way.

When tabindex’s value is set to zero or a positive number, the element can be navigated to via the keyboard’s Tab key. When it is set to a negative number, its element can be programmatically focused via JavaScript.

**How JSX execution happens?**

Our JS engine doesn’t understand the JSX code and here the Babel comes into the picture.

Babel is next generation javascript compiler, it takes the JSX code and converts it into the normal javascript code.

e.g.

JSX

const element = <h1>Hello, world!</h1>;

Babel output

const element = React.createElement("h1", null, "Hello, world!");

Where does babel come from?

Babel comes along with bundler in our case it is parcel, it came as a dependency along with parcel.

*Read more about babel on babel documentation*

[*https://babeljs.io/docs/en/*](https://babeljs.io/docs/en/)

React component

Components are independent and reusable bits of code. They serve the same purpose as JavaScript functions, but work in isolation and return HTML.

Component is a JavaScript class or function that optionally accepts inputs i.e. properties(props) and returns a React element that describes how a section of the UI (User Interface) should appear.

*https://reactjs.org/docs/components-and-*[*props*](https://reactjs.org/docs/components-and-props.html)*.html*

Components come in two types, Class components and Function components

* Function components

The simplest way to define a component is to write a JavaScript function:

function Welcome(props) {

return <h1>Hello, {props.name}</h1>;

}

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “**function components**” because they are literally **JavaScript functions**.

These functional components will return JSX.

When we want to render react component we will do like root.render(<HeaderComponent/>); whereas for react element we will do like root.render(header);.

React Element==React object

Functional component==React function

We should start the name of the component by capital letter, it is not mandatory but it is a good practice.

**How to use react element inside of react component?**

We need to use {} these curly brackets to use any react element inside the JSX.

Similarly, for adding react components into another react components we can use like a tag <HeaderComponent/>

As we know functional components are nothing but just a javascript functions we can also add them inside {} like {HeaderComponent()}

We can write any javascript code inside JSX, trick is we need to write the code inside the curly brackets {} and will work.

**JSX is XSS safe**

Javascript JSX is very safe that it saves our code from most common vulnerable attack known as XSS attack.

Whenever we give any javascript to JSX first it will do sanitization on that data.

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that’s not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

**Component composition**

When we want to use one component inside another component is known as component composition.

Component composition is the name for passing components as props to other components, thus creating new components with other components.