JSX

JSX is an extension to XML, that's why the syntax of JSX looks extremely similar to XML and HTML (HTML is also a derivative of XML only). Meta prepared JSX for making the UI implementation more easy, because using plain JS functions that React and other technologies provide can make the code unreadable. And with JSX, because it is a very HTML like syntax it becomes easy.

Features

- Looks and feels like HTML
- Has some minor differences compared to HTML
- Writing complex UI logic is easy with JSX
- Get's easily embedded with any JS code as well i.e. in the same file we can write JS and JSX together.

Meta didn't intent to get JSX compatibility added in the browser or in Ecmascript but they intend that there should be a layer of transpilation that can convert JSX to normal JS code.

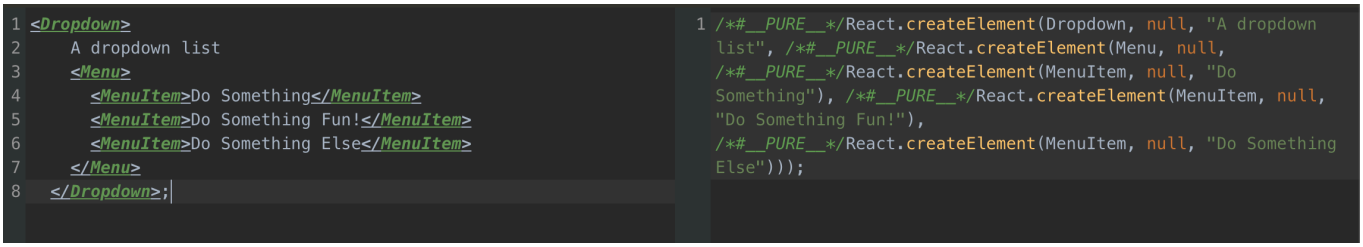
Example code for JSX

```
<Dropdown>
    A dropdown list
    <Menu>
      <MenuItem>Do Something</MenuItem>
      <MenuItem>Do Something Fun!</MenuItem>
      <MenuItem>Do Something Else</MenuItem>
    </Menu>
  </Dropdown>;
```

Transpilation

Transpilation is the process of converting source code from one high-level programming language to another.

We can use transpilers like `Babel` to convert JSX to compatible JS.



```jsx
<Dropdown>
    A dropdown list
    <Menu>
      <MenuItem>Do Something</MenuItem>
      <MenuItem>Do Something Fun!</MenuItem>
      <MenuItem>Do Something Else</MenuItem>
    </Menu>
  </Dropdown>;
```

```jsx
/*#__PURE__*/React.createElement(Dropdown, null, "A dropdown
list", /*#__PURE__*/React.createElement(Menu, null,
/*#__PURE__*/React.createElement(MenuItem, null, "Do
Something"), /*#__PURE__*/React.createElement(MenuItem, null,
"Do Something Fun!"),
/*#__PURE__*/React.createElement(MenuItem, null, "Do Something
Else")));
```

Note:

To use JSX in your code base, we need to make a file with `.jsx` extension.

Components

As we know that components are reusable UI elements. And in normal programming, functions help us to put reusable piece of code at one place.

Now In a react code base if we have a function that returns some kind of JSX, we will call that a component or more specifically a functional component. Why a functional component ?
Because this component has been made using functions.
Is there any other way to make components ? Yes, we can make class components also.

Functional Component

Writing functional component is easy, we use the function keyword and then name the function. Name of the function is technically name of the component. This component must return some JSX to be called as component.

Now we know how to define it, but how to call it ?

Calling a component

Component are functions and generally we call functions like this:

```
App( ... );
```

But because we are using JSX, it provides an alternative JSX compatible way to call out functional components.

```
<App />
```

The above JSX syntax calls the App component.

Note

Name of a component must start with a capital letter else it will not work properly. The custom components must be in PascalCase and inbuilt HTML element based components like `div`, `h1`, `h2` etc should be in lowercase.

Exampe:

The below code will not work:

```
import customComponent from "./CustomComponent";

function App() {

return (

        <div>

                <h1>Hello, world!</h1>

                <customComponent />

                <customComponent />

                <customComponent />

                <customComponent />

                <customComponent />

        </div>

);

}


export default App; // To make sure other files can access our APp function
```

We will get the following error in the browser:

```
Warning: <customComponent /> is using incorrect casing. Use PascalCase for
React components, or lowercase for HTML elements.
```

How to setup tailwind with vite react project ?

1. Setup a new vite project
2. Inside the project install tailwindcss, postcss and autoprefixer

```
npm install -D tailwindcss postcss autoprefixer
```

3. Setup tailwindcss config and postcss config file

```
npx tailwind init -p
```

4. The above command will create the config files.
5. Go to your index.css file and add tailwind directives to enable tailwind styling classes in the project

```
/* index.css */*
@tailwind base;

@tailwind components;

@tailwind utilities;
```

6. The above code will add tailwind styling everywhere in our project because `index.css` is the root css styling that's applied everywhere
7. And that's it, tailwind is ready to use.
8. Now to use tailwind css in any component we need to add tailwind classes. But JSX is a part of JS, hence we cannot use `class` keyword in the JSX syntax to mimic html attribute `class`. So, in JSX instead of class we write `className` attribute to html element components.

```
function App() {

        return (
```

```
                <div>

                        <h1 className="font-semibold text-3xl">Hello, world!
</h1>

                </div>

        );

}
```

More Details on JSX

- In a JSX based component, if we have a single JSX to be returned, then that is directly done, but for a multiline component, we need to wrap the JSX in a pair of parenthesis and then return it.

```
function Button() {
        return (

                <button>

                        Click me

                </button>

        );
}
export default Button;
```

- In a JSX component, there should be only one Parent HTML element returned, i.e. we can only return one single element from a JSX component, if we want to return multiple elements then we need to wrap those multiple elements in a single parent element.
  The below code will not work as it is returning multiple elements:

```
function Button() {
        return (

                <button>

                        Click me
```

```
                </button>


                <p>click</p>
        );
}
```

But if we wrap the button and p tag in a common parent and then return it, it will work.

```
function Button() {
        return (
                <div>
                        <button>
                                Click me
                        </button>
                        <p>click</p>
                </div>
        );
}

export default Button;
```

React fragments

- Because of the constraint that we can only return a single parent from JSX component, we end up adding more wrapper elements on the ui, which might not be required. For example, the `div` wrapping button and p tag is an extra element. For a very complex UI, there can be many many components, and if each component return something irrelevant then we have a lot of non-required elements.
- To stop this we can use React fragments, fragments can be used to wrap the elements in a single parent, without adding any extra element on the html.

How to create fragment ?

To create a fragment the simplest way is to write empty opening and closing tags.

```
function Button() {
        return (
                <>
                        <p>Hello</p>
                        <button>btn</button>
                </>

        );
}
```

Another way to add fragment is to use `React.Fragment` component.

```
import React from 'react';
function Button() {
        return (
                <React.fragment>
                        <p>Hello</p>
                        <button>btn</button>
                <React.fragment>

        );
}
```

And with any one of the two used, we will still get no extra wrapper elements on the UI.

JSX curlies

In JSX we can write some JS, which will be evaluated on the runtime and the return value is showed on the UI. To do this, we have to wrap out JS expression in a pair of curly braces called as JSX curlies.

```
function Button() {
        return (
                <>
                        {10*3}
                </>

        );
```

```
}
```

This is going to show 30 on the UI because thats the return value after evaluating `10*3`.

How to make functional component more alive??

Currently our components haver some static ui, they are not expecting any inputs. But in real function we have the logic confined and then we pass some inputs which is processed on the function and we get an output.

To pass some inputs to our components which they can consume we use something called as `props`.

```
function Button({ buttonType, text }) {
        return (

                <button

                        type={buttonType}

                        className="px-4 py-2 bg-blue-500 border border-blue-
700 text-white hover:bg-blue-700 hover:border-blue-900 rounded-md
transition-all"

>

                        {text}

                </button>

        );
}
export default Button;
```

So, here in the Button component, we have destructured the props object and fetched text and button type prop. We can use these props by wrapping them in JSX curlies as they need to be evaluated during runtime.
If we don't want to destructure the props, then we can expect a props object directly as parameter and then use properties inside it.

```
function Button(props) {
        return (

                <button

                        type={props.buttonType}

                        className="px-4 py-2 bg-blue-500 border border-blue-
700 text-white hover:bg-blue-700 hover:border-blue-900 rounded-md
transition-all"

>

                        {props.text}

                </button>

        );
}
export default Button;
```

But how do we pass the values of these props ? We can pass the values from the call site of component in the form of `key=value` format which is very similar to that of HTML attreibutes but with just JSX.

```
<Button text="Secondary" buttonType="submit" />
```

So here text prop gets a `Secondary` value and type prop gets a `submit value`.