# React Fundamentals

React JSX (JavaScript XML) allows you to write HTML elements directly in JavaScript, making it easier to create and manage the UI components in React applications. Here is a simple example to illustrate how JSX works in a React component:

## Basic Example of React JSX

1. **Create a new React app** (if you haven't already):

   ```
   npx create-react-app my-jsx-example
   cd my-jsx-example
   ```

2. **Modify the `App.js` file** to use JSX:

```
import React from 'react';

function App() {
  // Using JSX to define the UI
  return (
    <div className="App">
      <header className="App-header">
        <h1>Welcome to React JSX Example</h1>
        <p>
          This is a simple example of using JSX in a React
component.
        </p>
      </header>
    </div>
  );
}

export default App;
```

3. **Run the app** to see the output:

   ```
   npm start
   ```

## Detailed Breakdown

- **Import React**: Even though importing `React` is not strictly necessary for JSX as of React 17, it's still a good practice and may be required if you are using older versions.

  ```
  import React from 'react';
  ```

- **Define a Functional Component**: We define a functional component named `App`.

  ```
  function App() {
    // Component logic goes here
  }
  ```

- **Return JSX**: Inside the component, we return JSX which represents the UI.
  - Elements**: JSX elements are written using XML-like syntax.**

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

  - **Attributes**: JSX attributes are similar to HTML attributes, but some attributes are written in camelCase (e.g., className instead of class).

    ```
    const element = <div className="container">Content</div>;
    ```

  - **Embedding Expressions**: You can embed JavaScript expressions inside JSX by wrapping them in curly braces {}.

    ```
    const name = 'Amit';
    const element = <h1>Hello, {name}!</h1>;
    ```

In this example, the JSX includes a `div` with a class of "App", a `header`, an `h1` element, and a `p` element.

```
return (
  <div className="App">
    <header className="App-header">
      <h1>Welcome to React JSX Example</h1>
      <p>This is a simple example of using JSX in a React
component.</p>
    </header>
  </div>
);
```

## Adding Some Style

To make it look better, you can also add some styles in the `App.css` file:

```
/* App.css */
.App {
  text-align: center;
}

.App-header {
  background-color: #282c34;
  padding: 20px;
  color: white;
}
```

## React Hooks

React Hooks are functions that let you use state and other React features without writing a class. They enable you to manage component state, handle side effects, and access context in a more concise and readable way. Introduced in React 16.8, Hooks have become an essential part of modern React development. Here, we'll explore some of the most commonly used hooks in detail with examples.

## 1. `useState`

The `useState` hook lets you add state to functional components. It returns an array with the current state value and a function to update that state.

**Example**:

```
import React, { useState } from 'react';

function App() {
  const [count, setCount] = useState(0); // Initialize state with useState
hook

  const handleClick = () => {
    setCount(count + 1); // Update state when button is clicked
  };

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={handleClick}>
        Click me
      </button>
    </div>
  );
}
```

## 2. `useEffect`

The useEffect hook lets you perform side effects in function components. It runs after the first render and after every update, but you can control when it runs by specifying dependencies.

**Example**:

```
import React, { useState, useEffect } from 'react';

function App() {

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

  useEffect(() => {
    alert(`You clicked ${count} times`)
  });

  const handleClick = ()=> {
    setCount (count + 1)
  }

  return (
    <div>
      <div>You have clicked {count} times</div>
      <button onClick={ handleClick} >
        Click me
      </button>
    </div>
  );
}
```

## 3. `useContext`

The `useContext` hook lets you access the value of a context directly.

**Example**:

```
import React, { useContext, createContext } from 'react';

const MyContext = createContext();

function Example() {
  const value = useContext(MyContext);
  return <div>{value}</div>;
}

function App() {
  return (
    <MyContext.Provider value="Hello from context!">
      <Example />
    </MyContext.Provider>
  );
}
export default App;
```

## 4. `useReducer`

The `useReducer` hook is usually preferable to `useState` when you have complex state logic that involves multiple sub-values or when the next state depends on the previous one.

```
const [state, dispatch] = useReducer(reducer, initialState);
```

- reducer: A function that specifies how the state should be updated based on actions.
- initialState: The initial state value.
- state: The current state.
- dispatch: A function to send actions to the reducer.

**Example**:

```
import React, { useReducer } from 'react';

const initialState = { count: 0 };

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function App() {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })}>-</button>
    </div>
  );
}

export default App;
```

## 5. `useRef`

The `useRef` hook can be used to persist values between renders and to directly access DOM elements.

**Example**:

```
import React, { useRef } from 'react';

function FocusInput() {
  const inputRef = useRef(null);

  const handleClick = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={handleClick}>Focus the input</button>
    </div>
  );
}

export default FocusInput;
```

## 6. useMemo

The `useMemo` hook returns a memoized value. It only recomputes the memoized value when one of the dependencies has changed. This can improve performance.

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- memoizedValue: The memoized value.
- computeExpensiveValue: The function to compute the value.
- [a, b]: An array of dependencies that, when changed, will cause the function to recompute the value.

**Example**:

```
import React, { useState, useMemo } from 'react';

function ExpensiveCalculation({ num }) {
  const compute = (num) => {
    console.log('Computing...');
    return num * 2;
  };

  const memoizedValue = useMemo(() => compute(num), [num]);

  return <div>Computed Value: {memoizedValue}</div>;
}

function App() {
  const [number, setNumber] = useState(0);

  return (
    <div>
      <input
        type="number"
        value={number}
        onChange={(e) => setNumber(parseInt(e.target.value))}
      />
      <ExpensiveCalculation num={number} />
    </div>
  );
}

export default App;
```