Introduction to React: Building Dynamic User Interfaces

By: Sina Sedighi

Prerequisites

To embark on the journey of React, it's essential to have a solid foundation in the following technologies:

- **HTML:** Hyper Text Markup Language, the standard for creating the structure of web pages.
- **CSS:** Cascading Style Sheets, used for presenting and styling HTML documents.
- **JavaScript:** A lightweight, cross-platform programming language, pivotal for dynamic web page development.

HTML

HTML, or Hyper Text Markup Language, is the backbone of web development. It defines the structure of a web page through elements that label content. These elements, such as headings, paragraphs, and links, are encapsulated by start and end tags.

CSS

Cascading Style Sheets (CSS) enrich the presentation of HTML documents. By describing the visual style of web pages, CSS enhances user experience and allows for a clear separation of content and design.

JavaScript

JavaScript, a versatile scripting language, plays a crucial role in web development. Known for its lightweight nature, cross-platform compatibility, and interpretation, JavaScript empowers developers to create dynamic and interactive web pages.

ES6 (ECMAScript 6)

ES6, or ECMAScript 6, represents the sixth edition of the ECMAScript standard, aiming to standardize JavaScript. Published in 2015, ES6 introduces significant enhancements, including improved variable declarations and the introduction of arrow functions.

Variables in ES6

Before ES6, variables were declared using the **var** keyword. ES6 offers more options with **let** and **const**, providing flexibility and block-scoping:

```
let myVariable = 'Hello, ES6!'; const piValue = 3.14;
```

Arrow Functions

Arrow functions in ES6 allow for a more concise syntax in function declarations:
JS code:

```
const add = (a, b) => a + b;
```

Array Methods

There are many JavaScript array methods. One of the most useful in React is the .map() array method. The .map() method allows you to run a function on each item in the array, returning a new array as the result.

```
const myArray = ['apple', 'banana', 'orange'];

myArray.map((item) => console.log(item))
```

JSX: JavaScript Syntax Extension

JSX stands for JavaScript syntax extension. It is a JavaScript extension that allows us to describe React's object tree using a syntax that resembles that of an HTML template. It is just an XML-like extension that allows us to write JavaScript that looks like markup.

With jsx :

```
const myElement = <h1>I Love JSX!</h1>;
```

Without jsx:

```
const myElement = React.createElement('h1', {}, 'I do not use JSX!');
```

What is React?

React, often termed a JavaScript library for building user interfaces, is more accurately described as a library for constructing single-page applications. React facilitates the creation of reusable UI components, managing their state independently and enabling the composition of intricate user interfaces.

Benefits of Using React

1. **Reusability:** React components are designed for reusability, reducing redundancy and simplifying code maintenance.
2. **Efficiency with Virtual DOM:** React's virtual DOM efficiently updates the actual DOM, enhancing performance and user experience.
3. **One-Way Data Binding:** React follows a unidirectional data flow, enhancing understanding of data changes and their impact on application state.
4. **Community and Ecosystem:** React boasts a large and active community, providing abundant resources, libraries, and tools for developers.

Getting Started with React

To kickstart your React journey, ensure you have Node.js (version 14 or higher) installed. Create a new React app using the following method:

```
npx create-react-app my-app
```

Components in React

Components in React are independent and reusable pieces of code, resembling JavaScript functions. Two types of components, Class components and Function components, serve distinct purposes. This tutorial emphasizes Function components.

Function Components

A function component in React is a JavaScript function that defines a reusable piece of user interface. Unlike class components, function components are simpler and more concise, making them a popular choice for many React developers. These components are primarily used to encapsulate a specific piece of UI logic or functionality.

Function Components Example

In this example, MyComponent is a function component defined using an arrow function. It simply returns a JSX structure that represents the UI of the component. This component can then be used and composed within other parts of a React application.

```
const MyComponent = () => {
  return (
    <>
      <h2>Hello from MyComponent!</h2>
      <p>This is a simple function component</p>
    </>
  );
};
export default MyComponent;
```

State and useState Hook in Function Components

Functional components in React utilize the **useState** hook to manage state. This hook adds state to functional components, providing a pair of values: the current state and a function to update it.

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

Components Lifecycle

In React, the lifecycle of a component can be broken down into three main phases: Mounting, Updating, and Unmounting. With function components, these phases are managed using the useEffect hook and other hooks introduced in React.

1. Mounting Phase:

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

const MyComponent = () => {
  useEffect(() => {
    // This code runs after the component has mounted
    console.log('Component is mounted');

    // Cleanup function (optional)
    return () => {
      console.log('Component will unmount');
    };
  }, []); // Empty dependency array means this effect runs once after the
initial render

  return (
    <>
      {/* Component rendering code */}
    </>
  );
};
export default MyComponent;
```

Explanation:

Mounting occurs when an instance of a component is being created and inserted into the DOM.

The useEffect hook with an empty dependency array ([]), running the effect once after the initial render.

Any cleanup logic, such as removing event listeners, can be performed in the cleanup function returned by useEffect.

2. Updating Phase:

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

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

  useEffect(() => {
    // This code runs after every render (including the initial render)
    console.log('Component is updated');

    // Cleanup function (optional)
    return () => {
      console.log('Cleanup on component update');
    };
  }, [count]); // Dependency array includes the variables that trigger the
effect

  return (
    <div>
      {/* Component rendering code */}
      <p>Count: {count}</p>
      <button onClick={() => setCount(count + 1)}>Increment</button>
    </div>
  );
};

export default MyComponent;
```

Explanation:

Updating occurs when a component is re-rendered as a result of changes to its state.

The useEffect hook with a dependency array ([count]), running the effect after every render.

The effect can include cleanup logic or actions to be performed when specific dependencies (e.g., count) change.

3. Unmounting Phase

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

const MyComponent = () => {
  useEffect(() => {
    // Cleanup function (optional)
    return () => {
      console.log('Component will unmount');
    };
  }, []); // Empty dependency array means this effect runs once after the
initial render

  return (
    <div>
      {/* Component rendering code */}
    </div>
  );
};

export default MyComponent;
```

Explanation:

Unmounting occurs when a component is being removed from the DOM.

The cleanup function inside the useEffect, running just before the component is unmounted.

This is the place to perform cleanup tasks such as clearing intervals or canceling subscriptions to prevent memory leaks.

Joke App

Let's combine the **useState** and **useEffect** hooks to create a simple Joke App:

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

const JokeApp = () => {
    const [joke, setJoke] = useState('Loading...');

    useEffect(() => {
      fetch('https://api.chucknorris.io/jokes/random')
        .then((response) => response.json())
        .then((data) => setJoke(data.value));
    }, []);

    return (
      <div>
        <h1>Joke:</h1>
        <p>{joke}</p>
      </div>
    );
};

export default JokeApp;
```

This code defines a React functional component called JokeApp. The purpose of this component is to fetch a random Chuck Norris joke from the Chuck Norris Jokes API (https://api.chucknorris.io) and display it on the screen. Let's break down the code:

Import Statements:

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

The code imports React along with the useState and useEffect hooks from the 'react' library. These hooks are essential for managing state and performing side effects in functional components.

Component Definition:

```
const JokeApp = () => {
```

The JokeApp component is defined as an arrow function.

State Initialization:

```
const [joke, setJoke] = useState('Loading...');
```

The component uses the useState hook to declare a state variable joke and its corresponding updater function setJoke. The initial value of joke is set to 'Loading...'.

Effect Hook for Data Fetching:

```
useEffect(() => {
  fetch('https://api.chucknorris.io/jokes/random')
    .then((response) => response.json())
    .then((data) => setJoke(data.value));
}, []);
```

The useEffect hook is utilized for side effects. In this case, it's used to fetch data from the Chuck Norris Jokes API when the component mounts ([] as the dependency array).

The fetch function is used to make an HTTP request to the Chuck Norris Jokes API, and the response is processed using the json method.

The retrieved joke is then set in the component's state using the setJoke updater function.

Render Function:

```
return (
  <div>
    <h1>Joke:</h1>
    <p>{joke}</p>
  </div>
);
```

The render function returns JSX that represents the structure of the component.

It displays a heading "Joke:" and the content of the joke state variable within a paragraph element.The retrieved joke is then set in the component's state using the setJoke updater function.

Export Statement:

```
export default JokeApp;
```

The JokeApp component is exported as the default export of the module, making it available for use in other parts of the application.

Result:

In summary, this React component fetches a Chuck Norris joke from an API using the fetch function and displays it on the screen. The loading state is shown initially, and once the joke is fetched, it replaces the loading state with the actual joke content.