RESEARCH ARTICLE OPEN ACCESS

ReactJS: The Latest Front-end Web Development Framework

Aayush Kumar Shaw[1], Rishij Manna[2], Anirban Bhar[3], Soumya Bhattacharyya[4]

[1](B. Tech Student, Department of Information Technology, Narula Institute of Technology, Kolkata, India
Email: as94321038@gmail.com)

[2] (B. Tech Student, Department of Information Technology, Narula Institute of Technology, Kolkata, India
Email: rishijmanna2018@gmail.com)

[3] (Assistant Professor, Department of Information Technology, Narula Institute of Technology, Kolkata, India
Email: anirban.bhar@nit.ac.in)

[4] (Assistant Professor, Department of Information Technology, Narula Institute of Technology, Kolkata, India
Email: soumya.bhattacharyya@nit.ac.in)

Abstract:

To create user interfaces for single-page applications, developers often turn to the open-source toolkit ReactJS. Developers now have the tools they need to create massive web applications that are data-driven and dynamic without requiring frequent page reloads thanks to ReactJS. Therefore, React employs a clever diffing computation to only recover the necessary data in its DOM hub, while keeping everything else indefinitely. The use of modular components makes it easy to construct our software. The innovative concept of React also makes UI arrangement trustworthy, relieving a substantial burden from programmers so that they may concentrate on more substantial limits and business reasoning. Similarly, there is no required action or sequence of actions in a respond task. Users can pick and choose from a wide variety of libraries to do any number of tasks.

React.js is a game-changer for front-end development, opening up exciting new possibilities for app creation. This article discusses the benefits of using ReactJS for the front end of these applications and how it is aiding in their development. The primary principles, characteristics, features, development procedures, architecture, and a few dependencies that make up this library, as well as their advantages over competing frameworks, will be the emphasis of this article.

Keywords — **User interface, Open-source toolkit, ReactJS, Front-end development.**

I. INTRODUCTION

Internet has become a hive of activity for searching for information and performing various duties that were previously performed manually. Numerous mobile and web applications have facilitated the completion of a variety of duties. In the modern era, a significant portion of our daily tasks can be completed online. quicker internet and devices necessitate quicker application performance.

The migration of software and applications from desktop computers to the web is gaining momentum. There are numerous web and mobile device-compatible applications. Various frameworks and libraries based on JavaScript are used to develop various applications. Currently in production are ReactJS, AngularJS, EmberJS, MeteorJS, VueJS, and KnockoutJS, among others. ReactJS is one of these front-end application development frameworks.

Facebook developed the renowned open-source front-end JavaScript library React. The popularity of React among developer communities is largely attributable to its simplicity and straightforward yet effective development process. React simplifies the

creation of interactive user interfaces. It changes the application's data efficiently by rendering the precise components to the view of each state and updating the application's view.

In ReactJS, each component is responsible for managing its own state and composing user interfaces. This concept of components as opposed to templates in JavaScript allows a great deal of data to be readily passed to the application while keeping the state outside of the DOM. React can also be rendered on the server using Node.js. React Native can be used to create mobile applications in addition to web applications.

The purpose of the thesis is to conduct in-depth research on the JavaScript-based ReactJS library. The thesis will cover the fundamental concepts, characteristics, features, development processes, essential architecture, market research, and compatibility. The objective is to impart a comprehensive comprehension of the ReactJS library.

II. FEATURE

Eight years ago, React was shown to the public for the first time, and in that time, it has experienced remarkable expansion both inside and outside of Facebook. At Facebook, the development of brand-new online projects often takes place with some flavour of the framework known as React, which is also experiencing widespread adoption in the wider industry. Developers and engineers are opting for React as their preferred framework since it enables them to spend more time concentrating on the creation of the product and less time struggling with and learning how to use the framework.

An application written in React is made up of a collection of separate components, each of which represents a single view. Iterating on product development is made more simpler by the concept of every individual view component. This is because it is not necessary to take into consideration the complete system in order to make modifications to a single view or component. Because React surrounds the mutative, imperative API of the DOM with a declarative API, the level of abstraction is increased while the programming paradigm is simplified. This makes the code for an application that was built with React generally predictable. In addition to this, the application that is designed with React is simpler to grow.

The use of React in conjunction with the rapid iteration cycle of the web has resulted in the creation of several outstanding products, including a large number of components for Facebook. On top of React, the fantastic JavaScript framework known as Relay has also been developed, and it helps to make the process of retrieving data on a wide scale more straightforward.

III. RELATED WORK

ReactJS is a widely utilised open-source JavaScript toolkit that serves as a fundamental framework for the creation of single-page or mobile applications. However, React primarily focuses on presenting data to the Document Object Model (DOM), which means that developing React apps often requires the use of additional frameworks for managing state and handling routing. Flux [1], as a form of engineering, is employed by Facebook in conjunction with React. React is primarily focused on the View layer inside the Model-View-Controller (MVC) design pattern, hence limiting its scope to the V component. [2] Flux is the architectural pattern responsible for facilitating the creation of data layers in JavaScript applications and constructing client-side web applications. Flux supplements the reactivity of composable view components by facilitating the flow of information through its unidirectional data stream. React provides developers with a layout language and a set of function hooks that facilitate the creation of HTML elements [3]. It might be argued that Flux can be characterised more as a pattern rather than a framework, consisting of four primary components: Dispatcher, Stores, Views (React components), and Actions. The concept of flux adheres to the notion of a one-way flow of information, hence facilitating a more streamlined approach to pinpointing errors. The data undergoes a significant bottleneck within your application's process. React and Flux are widely recognised frameworks that adhere to the concept of unidirectional data flow.

Lifecycle methods [4] are a set of predefined methods in object-oriented programming that are invoked automatically at various stages of an object's lifecycle. These methods serve specific purposes and allow developers to perform necessary actions during different phases of every component in React goes through a series of lifecycle events. In a similar vein, React Lifecycle systems encompass a series of operations that occur within the lifespan of a React component, spanning from its initialization to its termination. Every element in the React framework undergoes a series of operations throughout its lifecycle, including mounting, updating, and unmounting. The render() method facilitates the allocation of fragments to the user interface (UI) throughout the process of resurrecting and mounting a component. If there is no information to be communicated in the section, the render() function will restore an incorrect state. The componentDidMount() function is a lifecycle method that is invoked when a component is mounted and rendered. In the event that data is required from a remote endpoint, the componentDidMount() method is employed to initiate API calls. The componentDidUpdate() lifecycle method is employed when there is a need to revive the Document Object Model (DOM) in response to changes in properties or state. In order to properly dispose of any activities and prevent harm to the component, it is necessary to invoke the componentWillUnmount() method before the component is unmounted. Altering the condition of the component is deemed inconceivable within the framework of this technique. The tasks involved in this technique encompass the disposal of timers, storage management, and the cessation of API calls. The utilisation of the shouldComponentUpdate() lifecycle method is advantageous in situations where it is preferred to abstain from utilising React to express prop or state updates. In this particular methodology, the constituent element of the course is once again delivered. This particular approach is employed infrequently and distinctively for certain performance enhancements. The getSnapshotBeforeUpdate() is a recently introduced lifecycle method that can be utilised as an alternative to the componentWillUpdate() method.This process is commonly referred to as pre-instantiating the Document Object Model (DOM). The value returned by the getSnapshotBeforeUpdate() method is used as a parameter in the componentDidUpdate() method. This system is hardly utilised or remains largely unused.

IV. STATE MANAGEMENT

When developing more complicated applications, state management in React becomes increasingly important. In React, you can choose from a number of alternative approaches to managing state that are optimised for various scenarios. Here is a rundown of some of the most popular:

A. Local State

Using the useState hook, local state refers to the process of handling data locally, within a single component. This works well for moderately large programmes when state doesn't need to be shared among a lot of moving parts. For components without numerous dependent children, its simplicity and low weight make it a viable option.

Example:

```
import React, { useState } from 'react';
function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

B. Context API

Using the Context API, you can avoid passing props through the entire component tree in order to communicate state between different components. It's an excellent choice for medium-sized projects when several components need access to a specific state without resorting to prop drilling.

Example:

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

const MyContext = createContext();
```

```
function MyProvider({ children }) {
  const [state, setState] = useState(initialState);

  return (
    <MyContext.Provider value={{ state, setState }}>
      {children}
    </MyContext.Provider>
  );
}

function MyComponent() {
  const { state, setState } = useContext(MyContext);

  return <div>{state.value}</div>;
}
```

C. Redux

Redux is a container for JavaScript applications that provides a predictable state. It is suited for use in large applications that require intricate control of their states. Your application's state will be centralised by Redux, which will make it much simpler to maintain and troubleshoot.

Example:

```
// actions.js
const increment = () => ({ type: 'INCREMENT' });

// reducer.js
const initialState = { count: 0 };
const reducer = (state = initialState, action) => {
  switch (action.type) {
    case 'INCREMENT':
      return { ...state, count: state.count + 1 };
    default:
      return state;
  }
};

// store.js
import { createStore } from 'redux';
const store = createStore(reducer);

// component.js
import React from 'react';
import { connect } from 'react-redux';
import { increment } from './actions';

function MyComponent(props) {
  return (
    <div>
      <p>You clicked {props.count} times</p>
      <button onClick={props.increment}>Click me</button>
    </div>
  );
}

const mapStateToProps = (state) => ({
  count: state.count
});

export default connect(mapStateToProps, { increment })(MyComponent);
```

D. MobX

MobX is a library for managing states that, through the transparent application of functional reactive programming (TFRP), makes managing states both straightforward and scalable. In comparison to Redux, it offers a more straightforward method of maintaining application state and may be used for applications of varying sizes.

Example:

```
import { observable, action, makeObservable } from 'mobx';

class CounterStore {
  count = 0;

  constructor() {
    makeObservable(this, {
      count: observable,
      increment: action
    });
  }

  increment() {
    this.count++;
  }
}

const counterStore = new CounterStore();

// Component
import React from 'react';
import { observer } from 'mobx-react';

const MyComponent = observer(() => {
  return (
    <div>
      <p>You clicked {counterStore.count} times</p>
      <button onClick={() => counterStore.increment()}>Click me</button>
    </div>
  );
});
```

Each of these approaches to state management comes with a unique set of benefits and drawbacks; selecting one is typically determined by the

particular specifications of the application you are working on. Local state is fantastic for simplicity, Context API is beneficial for avoiding unnecessary prop drilling, Redux is excellent for managing huge state in a predictable fashion, and MobX gives an approach to state management that is more straightforward and intuitive. In the end, the decision is going to be determined by the size, complexity, and particular requirements of your React application.

V. PERFORMANCE OPTIMIZATION

Virtual DOM is a programming concept in React that maintains an ideal or "virtual" representation of the user interface in memory. This synthetic representation corresponds to the DOM (Document Object Model) that is rendered on the web page. The virtual DOM serves as an intermediary between the state/data changes in a React component and the actual rendering of the user interface in the browser.

Here's how it operates and why it's important for optimising React applications:

When the state or data of a React component changes, the entire component tree is re-rendered in the virtual DOM, not the actual DOM. React then compares the newly created virtual DOM to a snapshot of the virtual DOM prior to the update. This procedure is known as "diffing."

Since manipulating the actual DOM is performance-intensive and time-consuming, the virtual DOM provides a means to reduce direct interaction with the DOM. By utilising a virtual representation, React can combine updates and reduce the number of DOM manipulations, resulting in significant performance enhancements.

React groups multiple updates and applies them all at once, reducing the number of times the actual DOM is consulted and modified. This approach to updating the DOM in batches is more efficient than making individual modifications to the DOM whenever the state changes.

The reconciliation algorithm utilised by React is intelligent. When the state changes, the entire actual DOM is not updated. Instead, it calculates the difference between the old and new virtual DOMs. On the actual DOM, only the differences (updates) are then applied. This process, known as reconciliation, ensures that the minimum number of DOM manipulations are performed, resulting to optimised performance.

Virtual DOM also enables the effective reuse of components. When the state of a component changes, React only needs to update the portion of the virtual DOM that corresponds to that component, rather than the entire DOM. This reusability is central to the effectiveness of React.

React facilitates a more seamless user experience by optimising DOM updates. Updates are efficiently computed and implemented, resulting in faster rendering times and a more responsive application experience.

In conclusion, React's Virtual DOM is an essential optimisation technique. By minimising direct interaction with the actual DOM, React applications become more efficient, responsive, and seamless for the user. This approach to rendering updates is a significant reason why React has become so popular among developers for creating high-performance web applications.

Improving the efficacy of a React application is essential for providing a positive user experience. You have mentioned a number of outstanding techniques, including code splitting, memoization, and lazy loading. Here is a more thorough explanation of these and additional techniques:

A. Code Splitting:

Code splitting is the process of dividing a JavaScript bundle into smaller pieces that can be launched on demand. This is particularly beneficial for large applications, where loading everything at once could be time-consuming.

React has a built-in method for separating code using dynamic imports. Libraries such as React Loadable and React.lazy() can also aid in code separation implementation.

B. Memoization:

Memoization is an optimization technique used to store the results of expensive function calls and return the cached result when the same inputs occur again.

React provides a useMemo hook which can be used to memoize values and prevent unnecessary

re-computations. Similarly, the useCallback hook can be used to memoize functions.

C. Lazy Loading:

Lazy loading entails loading only the necessary components and assets when they are required, as opposed to loading everything in advance.

React provides the delayed component loading React.lazy() function. Additionally, React Suspense can be used to manage loading conditions while components are lazily loaded.

VI. CONCLUSION

The purpose of the thesis was to investigate and evaluate ReactJS, an open-source, JavaScript-based front-end library. Facebook created ReactJS for their own use and later released it as open source. Since its inception, ReactJS has rapidly acquired immense popularity among developers and the tech industry.

In conclusion, this document provides clear instructions on how to get started with React, React's features and functionalities with examples, when to choose React over other alternatives, and what data architecture management system to consider along with its future prospects.

Since React is a difficult and essential technology to master, it would be advantageous to acquire more knowledge about it through additional research. With this in mind, the topic was selected. The development of a React application would be beneficial in terms of enhancing practical skills, but due to certain limitations, a comprehensive review and evaluation was produced. During the last few months of studying React, a firm concept has already been developed.

Finally, it can be stated that ReactJS is a significant technology to learn and should be considered for production use. It has added a new dimension to the development of web applications. The fast rendering library increases the application's efficacy, and it is evident that React has a promising future, so it is worthwhile to learn React.

REFERENCES

[1] Naimul Islam Naim, “ReactJS: An Open Source JavaScript Library for Front-end Development,”.

[2] Archana Bhalla, Shivangi Garg, Priyangi Singh, “Present day web-development using reactjs,” Volume: 07 Issue: 05, May 2020.

[3] Dave Carlsson, David Ko, “System and methods for JavaScript based HTML website layouts,” US 8,959.427 B1.

[4] Minati Biswal, “React lifecycle methods,” Researchgate Article, December 2019.