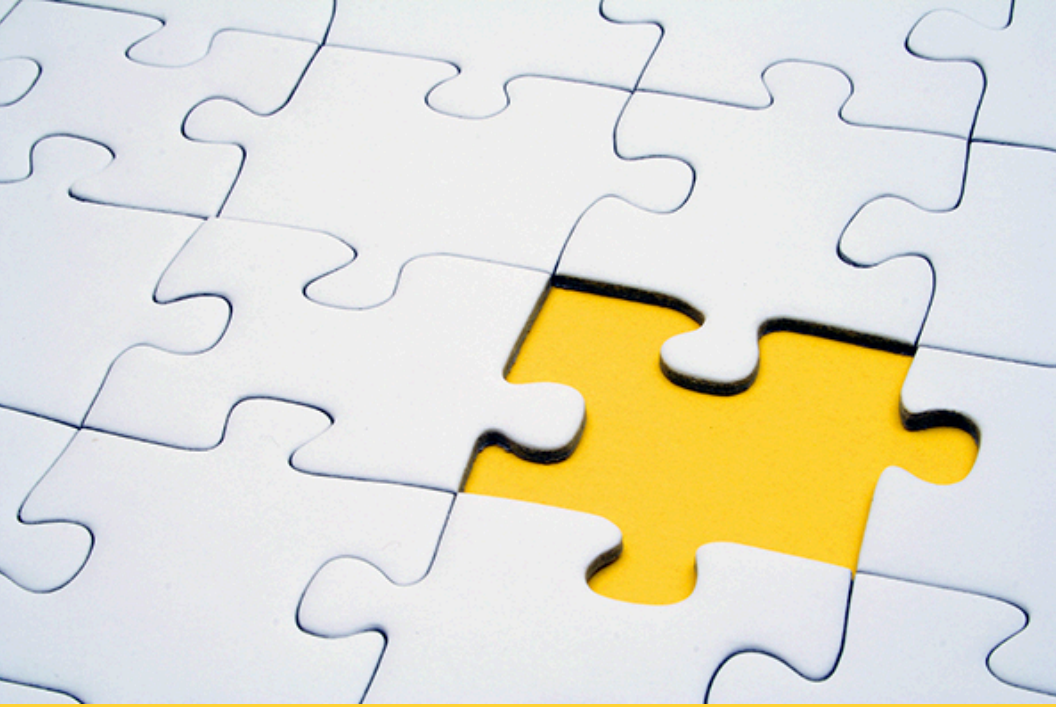


REACTJS

Simplified React Mastery

Unraveling The Power of ReactJs

By **Vivek Singh**

Simplified React Mastery

Unraveling the power of ReactJs

By **Vivek Singh**

Enthusiastic | Solution Hunter | Researcher

With over 14+ years of dedicated experience in the realm of software development and a passion for simplifying complex concepts, Vivek Singh brings a unique perspective to the world of ReactJs. Through extensive exploration and a keen eye for innovative solutions, Vivek Singh has mastered the art of breaking down intricate technical topics into easily digestible narratives.

As a seasoned practitioner of Java, Spring Framework, Spring Boot, and various other technologies, Vivek Singh combines a deep understanding of software engineering with a knack for effective communication. With a portfolio that spans from Micro-Services to Frontend technologies, Vivek Singh is adept at weaving together the intricate threads of modern development.

In this book, Vivek Singh channels their enthusiasm for both learning and teaching, offering readers a simplified journey into the world of ReactJs. By leveraging their expertise in ReactJs alongside a multitude of other tools, Vivek Singh is committed to empowering readers to grasp the foundations of this essential technology and embark on their own creative software endeavors.

Join Vivek Singh on this literary adventure, where enthusiasm meets expertise, and simplicity guides the way through the intricate landscape of ReactJs.

“Life is a canvas of endless possibilities, where every stroke of learning adds vibrant colors to our journey, and work becomes the masterpiece through which we share our soul with the world.”

Prologue

In the ever-evolving landscape of web development, where technologies rise and fall like tides, one constant remains– the desire to build remarkable digital experiences. Welcome to a journey that begins with a single step, a journey into the heart of React Project Setup.

As we embark on this exploration, we find ourselves at the crossroads of creativity and structure, where the dance of innovation is guided by the rhythm of meticulous preparation. This book is your compass, your guiding light through the intricate maze of setting up a React project.

In the following chapters, we'll traverse the vast expanse of tools, configurations, and methodologies that shape the foundation of every successful React application. From choosing the right tools for the job to architecting a resilient directory structure, our voyage will be marked by insights and revelations that empower you to master the art of project setup.

But this journey is not just about lines of code and configuration files; it's about understanding the philosophy that underpins modern web development. It's about honing your skills to craft applications that not only work flawlessly but also engage and delight users.

So, dear traveler, let us embark on this expedition together. Let curiosity be our compass, determination our guide, and this book our map to navigate the intricate terrain of React Project Setup. As you turn the pages and uncover the treasures within, may you discover a newfound

appreciation for the intersection of creativity and structure –the heart and soul of every remarkable digital creation.

With each chapter, you'll uncover the pieces of the puzzle that bring your projects to life. So let's begin, for beyond these words lies a world of possibilities waiting to be realized through your code, your creations, and your boundless imagination.

Table of Content

Prerequisites

HTML/CSS

- **HTML**: Understanding the basic structure of web documents, elements, attributes, and semantic HTML.
- **CSS**: Familiarity with styling, layout, responsive design, and CSS pre-processors like Sass or LESS.

JavaScript

React is a JavaScript library, so a strong grasp of JavaScript is essential.

- **ES6+ Features**: Understanding modern JavaScript features like arrow functions, classes, template literals, destructuring, and more.
- **Asynchronous Programming**: Concepts like promises and async/await.
- **DOM Manipulation**: Although React abstracts much of this away, understanding how JavaScript interacts with the DOM can be beneficial.

Version Control Systems

- **Git**: Basic knowledge of version control using Git, including commits, branches, merges, and working with remote repositories.

Node.js & npm

- **Node.js**: Familiarity with Node.js as it's used in the development environment for React.
- **npm**: Understanding the Node Package Manager (npm) for managing packages and dependencies.

Command Line Interface (CLI)

- Comfortable using the command line, as many React development tasks are performed using terminal commands.

Understanding of Web Development Concepts

- **Client-Server Architecture**: Basic understanding of how the client and server interact in a web environment.
- **HTTP/HTTPS Protocols**: Familiarity with how data is exchanged between the client and server.
- **Web Browsers and Developer Tools**: Experience with browser developer tools for testing and debugging.

Optional Concepts & Tools

- **State Management Concepts:** Familiarity with state management, either in general programming or specific to web development.
- **Pre-processors and Build Tools:** Understanding of tools like Babel, Webpack, or other bundlers and transpilers can be helpful but not required to get started.

Conclusion

While you can start learning React with just HTML, CSS, and JavaScript knowledge, a deeper understanding of the above topics will enable a smoother learning experience and more efficient development. Since you're already working as a full-stack developer, you likely have a solid foundation in many of these areas, which will certainly be advantageous as you explore React.

Overview of ReactJs

What is ReactJs

ReactJS, or simply React, is an open-source JavaScript library developed by Facebook and the community. It's used to build user interfaces for web applications by enabling developers to create reusable components and manage their state. This makes React a go-to tool for crafting dynamic and engaging web experiences.

Core Principals:

1. **Components:** React breaks down the UI into independent, reusable pieces called components. Each component can manage its state and render logic, enabling a modular and clean code structure.
2. **Virtual DOM:** Unlike traditional DOM manipulation, React creates a virtual representation of the UI in memory. Whenever there's a change in the component's state or data, React calculates the difference between the current Virtual DOM and the previous version and updates the real DOM efficiently. This process is known as reconciliation.
3. **Unidirectional Data Flow:** React promotes a one-way data flow, meaning that data within a component flows in a single direction. This makes the code easier to understand and debug.

Key Features:

1. **JSX:** JSX (JavaScript XML) allows developers to write HTML-like syntax within JavaScript. It is then trans-compiled into JavaScript code, enabling a more intuitive way to define components.
2. **State and Props:** State allows components to create and manage their local data, while Props enable data transfer between parent and child components.
3. **Hooks:** Introduced in React 16.8, hooks enable functional components to have state and lifecycle features without writing a class.
4. **Context API:** Provides a way to pass data through the component tree without having to pass props down manually at every level.

Ecosystem:

React's ecosystem includes various tools and libraries to enhance its capabilities:

1. **Redux:** A state management library.
2. **React Router:** A routing library for navigating between different views.
3. **Next.js:** A framework for server-rendered React applications.
4. **Styled-components:** A library for styling components.

Past of ReactJS

Inception

- **2013**: React was created by Jordan Walke, a software engineer at Facebook. It was initially used for Facebook's News Feed. Facebook made it open-source in May 2013, during the JSConf US.

Key Milestones

- **2015**: Introduction of React Native, allowing the development of native mobile apps using React.
- **2016**: Introduction of the functional programming concept in React with the release of stateless functional components.
- **2017**: The release of React 16, also known as React Fiber, which revamped the core algorithm. It also introduced features like Error Boundaries and Fragments.
- **2018**: Introduction of Hooks in React 16.8, allowing functional components to use state and other React features.

Adoption and Growth

- Over the years, React's popularity soared, and it became one of the leading libraries for building UI. It gained a broad community of developers, contributors, and supporters.

Future of ReactJS

React continues to evolve, and its future seems promising. Here are some trends and directions that could shape its future:

Continued Emphasis on Functional Programming

- Functional components and hooks are likely to remain central, as they promote cleaner and more maintainable code.

Improved Concurrent Mode

- Concurrent Mode, an experimental feature, enables more fluid user experiences by allowing the rendering process to be interrupted and prioritized. Its full implementation could significantly change how React apps are built and function.

Server Components

- React Server Components aim to enable modern UX with a server-rendered architecture. It is expected to reduce the amount of JavaScript sent to the client and improve performance.

Enhanced Integration with Other Technologies

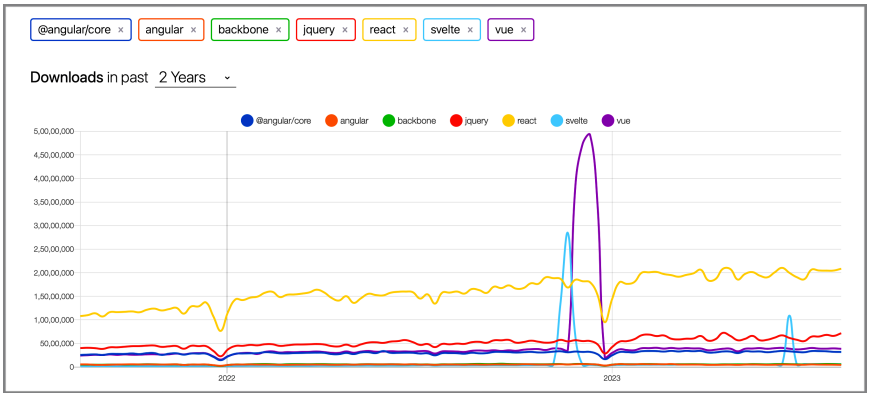
- React may see increased integration with emerging web technologies like WebAssembly, Progressive Web Apps (PWAs), and more.

Focus on Sustainability and Community-Driven Development

- The React team might continue to foster open collaboration, focusing on sustainability, accessibility, and community-driven enhancements.

More Tools and Libraries

1. Expect more tools, libraries, and frameworks to emerge around React, extending its capabilities and making development more straightforward.



Source: https://npmtrends.com/@angular/core-vs-angular-vs-backbone-vs-jquery-vs-react-vs-svelte-vs-vue

Conclusion

React is widely used in modern web development to create single-page applications (SPAs), interactive user interfaces, and complex front-end applications. It is often combined with other technologies like React Router for routing, Redux for state management, and various build tools for optimising and bundling code.

ReactJS has undergone significant changes and growth since its inception, driven by its community and the needs of modern web development. Its future appears to be on a path of continued innovation, focusing on performance, usability, and developer experience.

Given the ever-changing nature of technology, keeping up to date with React's official releases, community discussions, and key industry players will provide the most accurate insights into its ongoing evolution.

Installation & Setup

Starting with React is easy and doesn't take much time. All you need is a tool called Node.js and a few simple commands. In just a few steps, you'll have everything ready to begin building your very own React application. Let's take a look at how to get everything set up so you can start creating right away.

Your First React App

Step 1: Setting Up the Development Environment

Make sure you have Node.js and npm (Node Package Manager) installed on your system. If not, you can download and install them from the official Node.js website.

Step 2: Create a New React Project

Open your terminal or command prompt and run the following command to create a new React application using **`create-react-app`**, a tool that sets up a new React project with sensible defaults:

bash

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

Replace **my-first-react-app** with the name you want for your project.

Step 3: Navigate to the Project Folder

Once the command completes, navigate into the project folder:

```bash
cd my-first-react-app
```

Step 4: Start the Development Server

You can start the development server by running:

```bash
npm start
```

This command will start a local development server, and your new React app should open in your default web browser at **http://localhost:3000/**.

Step 5: Explore the Code

Open the project folder in your preferred code editor. You'll find the following main files and folders:

- **public/index.html**: The HTML file that hosts your React application.
- **src/index.js**: The JavaScript entry point where the React application is rendered.
- **src/App.js**: A sample React component that you can start modifying.

Step 6: Make Your First Changes

Try making some changes to the **src/App.js** file. For example, you can modify the content inside the ** ** to display a personalized welcome message:

Jsx

```
function App() {
  return (
    <div className="App">
      <h1>Welcome to Our Knowledge Session!</h1>
    <p>This is just the beginning of an exciting journey.</p>
    </div>
  );
}
```

Save the file, and the browser should automatically reload to show your changes.

Step 7: Build and Deploy (Optional)

Once you're ready to share your app, you can build it for production by running:

bash

```
npm run build
```

You can then deploy the **build** folder to a web server or hosting platform of your choice.

Your First React App Using Webpack

To Been Researched and Prepared

Your First React App Using Vite

To Been Researched and Prepared

Vite ESlint

To Been Researched and Prepared

Download Tools

To Been Researched and Prepared

Prettier

To Been Researched and Prepared

Conclusion

Congratulations! You've just set up and customized your first React application. From here, you can explore more features, create new components, and even integrate other libraries and tools like React Router or Redux.

How to get started in ReactJs

Thinking in React

When working with React, the thought process revolves around a component-based architecture. Here's how it usually works:

- **Break Down UI into Components**: Look at the design or layout of the application and divide it into individual components. Think of each part of the UI as a separate, self-contained piece.
- **Build a Static Version**: Start by building a static version of the UI without any interactivity or state management. Focus on defining and composing components.
- **Identify State and Data Flow**: Determine what parts of the UI need to be dynamic and figure out where the state should live within the component hierarchy.
- **Add Interactivity**: Implement interactive features by adding state and handling user inputs or events.
- **Consider Reusability**: Design components with reusability in mind, allowing them to be used in different parts of the application or even across different projects.

Why Components?

Components are fundamental to React for several reasons:

- **Modularity**: Components allow developers to break down complex UI into smaller, manageable pieces. This

modular approach makes it easier to develop, test, and maintain code.

- **Reusability**: Components can be reused across different parts of the application, leading to a more consistent UI and less repetitive code.
- **State Management**: Components can manage their state or receive data through props, giving developers control over how data flows through the application.
- **Separation of Concerns**: Components encourage a clear separation between logic, presentation, and behavior. This separation helps in maintaining a clean and organized codebase.
- **Efficient Updates**: React's Virtual DOM ensures that only the components that have changed are re-rendered, making updates efficient.

Conclusion

Thinking in terms of components aligns with modern software design principles and offers a systematic approach to building web applications. It encourages developers to create modular, reusable, and maintainable code, leading to more robust and scalable applications.

Components not only represent the building blocks of the UI but also encapsulate behavior and state, allowing for more cohesive and efficient development.

React Component

Declarative Vs Imperative

The terms "declarative" and "imperative" are commonly used to describe programming paradigms, especially in the context of UI development and React. Let's delve into what they mean:

Declarative Programming

In declarative programming, you express what you want to achieve without specifying how to achieve it. The focus is on the desired outcome rather than the step-by-step process.

Example in React:

Jsx

```
function App() {
  return (
    <div>
      <h1>Hello, World!</h1>
    </div>
  );
}
```

Here, you declare what the UI should look like, and React takes care of rendering it. You don't need to describe how to create or update the DOM elements.

Pros of Declarative Programming:

- More readable and maintainable code.
- Less concern with the underlying mechanics, allowing focus on the overall logic.
- Often leads to more concise code.

Imperative Programming

In imperative programming, you specify how to achieve something through explicit instructions. You describe the step-by-step process, detailing how to reach the desired outcome.

Example in Vanilla JavaScript (for comparison):

JavaScript

```
const container = document.createElement('div');
const header = document.createElement('h1');
header.textContent = 'Hello, World!';
container.appendChild(header);
document.body.appendChild(container);
```

Here, you provide a specific sequence of commands to create and append the elements to the DOM.

Pros of Imperative Programming:

- Offers more control over the exact logic and execution flow.
- Can be more performant in some cases, as you control every detail.

Declarative vs. Imperative in React:

- **Declarative Approach**: React embraces a declarative approach where you define what the UI should look like for different states, and React takes care of updating the UI when the state changes. This makes code more predictable and easier to debug.
- **Imperative Approach**: Traditional JavaScript DOM manipulation is more imperative, where you need to manage every detail of how elements are created, updated, and removed. This can become complex and error-prone, especially in large applications.

Where are they focus:

- **Declarative**: Focuses on the "what" - describes the desired end state without detailing the process.
- **Imperative**: Focuses on the "how" - provides a specific sequence of steps to achieve the end state.

React's declarative nature simplifies the process of building complex UIs, as developers can describe the desired state and rely on React to handle the updates. This leads to more maintainable code and allows developers to focus on higher-level logic rather than the intricacies of DOM manipulation.

JSX Basics

What is JSX?

JSX is a syntax extension for JavaScript, used predominantly with React. It provides a way to write components and elements using a syntax that resembles HTML. Though it looks like HTML, it is ultimately transpiled to JavaScript.

Basic Syntax

JSX looks similar to HTML but is actually closer to JavaScript. You can define elements and components using tags:

Jsx

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

Embedding Expressions in JSX

You can include JavaScript expressions inside JSX by wrapping them in curly braces ({}). This allows dynamic rendering of content:

Jsx

```
const user = { name: 'Vivek' };
const element = <h1>Hello, {user.name}!</h1>;
```

JSX Must Return a Single Element

JSX expressions must have a single parent element. If you need to return multiple elements, wrap them in a container like a or a fragment (<> ... </>):

Jsx

```
const element = (
  <>
    <h1>Title</h1>
    <p>Content</p>
  </>
);
```

Self-Closing Tags and Naming Conventions

JSX tags that have no children must be self-closed:

Jsx

```
const image = <img src="image.jpg" alt="description" />;
```

Attributes and component names in JSX follow the camelCase convention, and custom React components must start with a capital letter.

Specifying Attributes and Properties

JSX allows you to define props (properties) for components. String literals can be passed directly, while expressions must be enclosed in {}:

Jsx

```
const element = <MyComponent name="Vivek" age={30} />;
```

JSX Transpilation

JSX code is not understood by browsers, so it must be transpiled into regular JavaScript. Tools like Babel handle this transformation:

Jsx

```
const element = <h1>Hello</h1>;
// Transpiles to:
const element = React.createElement('h1', null, 'Hello');
```

Handling Children and Nested Components

JSX supports nested components, and you can define children components within parent components:

Jsx

```
function Welcome(props) {
  return <h1>Welcome, {props.name}!</h1>;
}
const element = (
  <div>
    <Welcome name="Vivek" />
    <p>Have a great day!</p>
  </div>
);
```

Accessibility Considerations

JSX supports ARIA (Accessible Rich Internet Applications) attributes, but they should be used in camelCase, such as aria-labelledby becoming ariaLabelledBy.

Jsx

```
function AccessibleButton(props) {
  return (
      <div role="button" tabIndex={0} ariaPressed={false}
onClick={props.onClick}>Click me</div>
  );
}
```

Conditional Rendering in JSX

JSX allows for conditional rendering using JavaScript logical operators, such as the ternary operator or short-circuit evaluation. Let's go through some examples demonstrating different ways you can achieve conditional rendering in JSX:

Using the Ternary Operator

A common approach to conditional rendering is using the ternary operator (**? :**). Here's an example that renders a greeting based on a user's login status:

Jsx

```
function Greeting(props) {
  return (
    <div>
          {props.isLoggedIn ? <h1>Welcome, Vivek!</h1> :
<h1>Please log in.</h1>}
    </div>
  );
}
```

Using Logical && Operator

If you want to render something based on a condition and do nothing if the condition is false, you can use the logical AND (**&&**) operator:

Jsx

```
function Notification(props) {
  return (
    <div>
          {props.hasNotifications && <span>You have new
notifications!</span>}
    </div>
  );
}
```

Using Functions for Conditional Rendering

You can also define functions inside components to encapsulate the conditional logic:

Jsx

```
function UserGreeting(props) {
  function renderGreeting() {
    if (props.isAdmin) {
      return <h1>Welcome, Admin!</h1>;
    } else if (props.isLoggedIn) {
      return <h1>Welcome, User!</h1>;
    } else {
      return <h1>Please log in.</h1>;
    }
  }

  return <div>{renderGreeting()}</div>;
}
```

Using IIFE (Immediately Invoked Function Expression)

An IIFE allows you to write more complex logic within your JSX:

Jsx

```
function UserProfile(props) {
  return (
    <div>
      {(() => {
        if (props.isLoading) {
          return <div>Loading...</div>;
        } else if (props.error) {
          return <div>Error loading profile</div>;
        } else {
          return <div>{props.username}'s Profile</div>;
        }
      })()}
    </div>
  );
}
```

Conditional rendering is a powerful feature in React, allowing you to create dynamic and responsive user interfaces. By leveraging JavaScript expressions, you can decide what parts of your components to render based on specific conditions, resulting in more flexible and maintainable code.

JSX provides a more natural and expressive way to define React components, offering a seamless blend of JavaScript functionality and HTML-like syntax. By understanding these core aspects, developers can create dynamic, reusable, and accessible React components efficiently.

Creating Components

In React, creating components is at the core of building applications. Components are reusable, self-contained pieces of code that represent part of a UI. Let's dive into how to create components in React:

Functional Components

Functional components are the simplest and most common way to define components in React. Here's an example:

Jsx

```
function Welcome(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

You can use this component by invoking it with a JSX tag:

Jsx

```
<Welcome name="Vivek" />
```

Class Components

You can also define components using ES6 classes. Class components are used when you need to have internal state or lifecycle methods:

Jsx

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

Component Composition

You can combine components to build more complex UIs. For example:

Jsx

```
function App() {
  return (
    <div>
      <Welcome name="Vivek" />
      <Welcome name="John" />
      <Welcome name="Jane" />
    </div>
  );
}
```

Components with State

Function component: It can have state using the "*useState*" hook:

Jsx

```
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>
  );
}
```

Class Component: It's state is handled using the **state** object and "*setState*" method:

Jsx

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }

  increment() {
    this.setState({ count: this.state.count + 1 });
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
          <button onClick={() => this.increment()}>Click me</
button>
      </div>
    );
  }
}
```

Understanding Props

Props allow you to pass data and even functions between components. They enable components to be dynamic and reusable, as they can render different content based on the props they receive.

Using Props in Functional Components

In functional components, props are accessed as an argument to the function. Here's an example:

Jsx

```
function Greeting(props) {
  return <h1>Hello, {props.name}! Your date of birth is {props.dob}</h1>;
}

<Greeting name="Vivek" dob="30th September" />
```

You can also destructure the props in the function argument for more concise code:

Jsx

```
function Greeting({ name, dob }) {
  return <h1>Hello, {name}! Your date of birth is {dob}</h1>;
}
```

Using Props in Class Components

In class components, props are accessed through **this.props**. Here's an example:

Jsx

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}!</h1>;
  }
}
```

Passing Various Types of Data

You can pass various types of data through props, including strings, numbers, objects, functions, and even other React elements.

Props Are Read-Only

One of the fundamental principles of props is that they are read-only. A component should never modify the props it receives. If you need to modify the data, you should manage it within the state of the component or use a state management library like Redux.

PropTypes

React allows you to define **propTypes** for a component, specifying the expected type of each prop. This helps with type checking and can prevent bugs in development.

Jsx

```
import PropTypes from 'prop-types';

function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}

Greeting.propTypes = {
  name: PropTypes.string.isRequired,
};
```

Default Props

You can define default values for props using the **defaultProps** property in class components or by using default parameters in functional components.

Jsx

```
Greeting.defaultProps = {
  name: 'Guest',
};
```

Component Lifecycle

The Component Lifecycle in React is a series of methods that get executed at different stages of a component's existence, providing developers with hooks to write code that reacts to changes in the component's state and properties.

Class Components with Lifecycle Methods

It can have lifecycle methods that will allows you to control what happens at different phases of a component's existence, from creation to removal from the DOM. Ex.

Jsx

```
class UserComponent extends React.Component {
  componentDidMount() {
    // Fetch user data when the component mounts
  }

  render() {
    return <div>{this.props.user.name}'s Profile</div>;
  }
}
```

1. Mounting Phase

The mounting phase is when the component is being inserted into the DOM.

- constructor(props): This method is called before the component is mounted. You can initialize state and bind event handlers here.
- static getDerivedStateFromProps(props, state): This method is called right before "render()", allowing you to update the state based on changes in props.
- render(): The only required method in a class component. It returns the JSX that represents the UI.
- componentDidMount(): This method is called after the component is inserted into the DOM. It's commonly used to initiate API calls or other side effects.

2. Updating Phase

The updating phase occurs when the component's props or state change, causing a re-render.

- static getDerivedStateFromProps(props, state): Similar to the mounting phase, this method is called whenever the component is re-rendered.
- shouldComponentUpdate(nextProps, nextState): Allows you to cancel the update by returning false. You can optimize performance by preventing unnecessary re-renders.
- render(): Called again to determine the changes in the virtual DOM.
- getSnapshotBeforeUpdate(prevProps, prevState): Called right before the changes from the virtual DOM are to be reflected in the real DOM. It returns a value that can be passed to componentDidUpdate.

- componentDidUpdate(prevProps, prevState, snapshot): Called immediately after updating occurs. You can perform side effects or update DOM in response to prop or state changes.

3. Unmounting Phase

The unmounting phase is when the component is being removed from the DOM.

- componentWillUnmount(): This method is called before the component is removed from the DOM. You can perform cleanup tasks, like invalidating timers or canceling network requests here.

4. Error Handling Phase

This phase handles errors that occur during rendering, in a lifecycle method, or in the constructor.

- static getDerivedStateFromError(error): Allows you to render a fallback UI by updating the state.
- componentDidCatch(error, info): You can log the error details and show a user-friendly message.

Functional Components with Lifecycle Methods using “useEffect”

It can use the useEffect hook for similar behavior:

1. Introduction to useEffect

The useEffect hook lets you perform side effects in functional components. Side effects could include API calls, subscriptions, manual DOM manipulations, and more. It serves a similar purpose to componentDidMount, componentDidUpdate, and componentWillUnmount in class components.

2. Basic Usage

The basic syntax of useEffect is:

Jsx

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

function UserComponent(props) {
  useEffect(() => {
      // Your side effect code here
  });

  return <div>{props.user.name}'s Profile</div>;
}
```

3. Handling Mounting and Unmounting

You can use useEffect to handle both mounting and unmounting logic:

Jsx

```
useEffect(() => {
  // Code to run on mount

  return () => {
    // Code to run on unmount
  };
}, []);
```

By passing an empty dependency array ([]), the effect will run once when the component mounts, and the return function will run once when it unmounts.

4. Running Effects After Updates

Without the dependency array, useEffect will run after every render:

Jsx

```
useEffect(() => {
  // Code to run after every render
});
```

5. Running Effects Based on Specific Dependencies

You can make the effect run when specific values change by providing a dependency array with those values:

Jsx

```
useEffect(() => {
  // Code to run when `prop` or `stateValue` changes
}, [prop, stateValue]);
```

6. Cleanup

Many effects need to be cleaned up to prevent memory leaks. You can return a function from your effect that will be called when the component is unmounted or before re-running the effect:

Jsx

```
useEffect(() => {
  // Subscribe to something
  const subscription = someService.subscribe();

  return () => {
    // Unsubscribe when the component is unmounted or dependencies change
    subscription.unsubscribe();
  };
}, [someDependency]);
```

7. Skipping Effects

If you want to skip an effect under certain conditions, you can handle that logic inside the effect itself:

```jsx
useEffect(() => {
  if (!shouldRun) return;

  // Rest of the effect
}, [shouldRun]);
```

When to Use Each

Class Components: If you're working on a legacy codebase that primarily uses class components or if you prefer the object-oriented paradigm.

Function Components: For new projects or when writing new components, it's often recommended to use function components, especially since the introduction of Hooks.

Class Comp... Lifecycle (Example)

Jsx

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

class UserProfile extends Component {
  constructor(props) {
    super(props);
    this.state = { user: null };
    console.log('Constructor: Component is being constructed');
  }

  static getDerivedStateFromProps(props, state) {
      console.log('getDerivedStateFromProps: Updating state based
on props');
    return null; // Returning null means no update to the state
  }

  componentDidMount() {
            console.log('componentDidMount: Component has been
mounted');
    // Simulate an API call to fetch user data
    setTimeout(() => {
      this.setState({ user: 'Vivek' });
    }, 1000);
  }
```

Jsx Continue...

```
shouldComponentUpdate(nextProps, nextState) {
        console.log('shouldComponentUpdate: Deciding whether to update');
   return true; // Returning true means continue with the update
  }

  componentDidUpdate(prevProps, prevState) {
          console.log('componentDidUpdate: Component has been updated');
  }

  componentWillUnmount() {
          console.log('componentWillUnmount: Component will be unmounted');
  }

  render() {
   console.log('render: Rendering component');
   return (
    <div>
     <h1>User Profile</h1>
                {this.state.user ? <p>Welcome, {this.state.user}!</p> : <p>Loading...</p>}
    </div>
   );
  }
}

export default UserProfile;
```

Here's a brief explanation of what's happening at each stage:

- **Constructor**: Initializes the state and binds event handlers.
- **getDerivedStateFromProps**: Can update the state based on changes in props.
- **render**: Renders the JSX markup.
- **componentDidMount**: Used to perform side effects like API calls after the component has been inserted into the DOM.
- **shouldComponentUpdate**: Can be used to optimize performance by preventing unnecessary updates.
- **componentDidUpdate**: Allows additional logic to be executed after an update occurs.
- **componentWillUnmount**: Used to clean up any subscriptions, timers, or other side effects before the component is removed from the DOM.

Functional Comp... Lifecycle (Example)

Jsx

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

function UserProfile(props) {
  const [user, setUser] = useState(null);

  // Equivalent to componentDidMount and componentDidUpdate
  useEffect(() => {
    console.log('Component has been mounted or updated');

    // Simulate an API call to fetch user data
    setTimeout(() => {
      setUser('Vivek');
    }, 1000);

    return () => {
      console.log('Component will be unmounted');
      // Equivalent to componentWillUnmount
    };
  }, []);
  // The empty array makes this effect run only on mount and unmount

// Equivalent to componentDidUpdate
  useEffect(() => {
    if (!user) return;
    console.log('User Updated');
  }, [user]);
  // This effect run only when user updated
```

Jsx Continue...

```
 return (
   <div>
     <h1>User Profile</h1>
     {user ? <p>Welcome, {user}!</p> : <p>Loading...</p>}
   </div>
 );
}
export default UserProfile;
```

In this example:

- **useState:** Initializes the state of the user. Equivalent to using the constructor in class components.
- **useEffect:** With an empty dependency array ([]), it acts similarly to componentDidMount, running the code inside the effect after the component is inserted into the DOM. It also returns a function that acts like componentWillUnmount, running when the component is about to be removed from the DOM.
- **Combining useEffect with Dependencies:** If you want to perform actions when specific props or state change, you can provide those values in the dependency array. This will make the effect act like componentDidUpdate for those specific values.

Basic Stateful Components

useState Hook

The **useState** hook allows functional components to maintain internal state without needing to convert them to class components. It's called with the initial value of the state and returns an array containing the current state value and a function to update that state.
The syntax is:

Jsx

```
const [stateValue, setStateValue] = useState(initialValue);
```

- **stateValue:** The current value of the state.
- **setStateValue:** A function to update the state value.
- **initialValue:** The initial value of the state.

Tips & Tricks

If we must update the state based on its existing value or when there are multiple states being set within a parent function, then we should proceed with the following method to update the state.

Jsx

```
setCount(count => count + 1);
```

Example:

Let's consider a simple counter component:

Jsx

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

function Counter() {
  // Declare a state variable named 'count' with an initial value of 0
  const [count, setCount] = useState(0);

  // Function to increment the count
  const incrementCount = () => {
    setCount(count + 1);
  };

  // Render the component
  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={incrementCount}>Increment</button>
    </div>
  );
}

export default Counter;
```

Here's how the code works:

- **Declare State:** const [count, setCount] = useState(0); declares a state variable called count with an initial value of 0.

- **Update State:** Inside the incrementCount function, setCount is used to update the state. Every time setCount is called, the component re-renders with the new state value.
- **Use State:** The current value of the count state is displayed inside the rendered JSX.

State In Class Components

While the **useState** hook is specific to functional components, you can achieve the same functionality using class-based components with the **this.state** object and **this.setState** method.

Example

Let's convert functional counter example to class component:

Jsx

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

class Counter extends Component {
  // Initialize the state with a 'count' property set to 0
  constructor(props) {
    super(props);
    this.state = { count: 0 };
  }
```

Jsx Continue...

```
// Method to increment the count
 incrementCount = () => {
  this.setState(prevState => ({
   count: prevState.count + 1
  }));
 };

 // Render the component
 render() {
  return (
   <div>
    <p>Count: {this.state.count}</p>
     <button   onClick={this.incrementCount}>Increment</
button>
   </div>
  );
 }
}

export default Counter;
```

Explanation:

- **Initialize State:** In the constructor, you initialize the state object with the initial state value. In this case, count is set to 0.

- **Update State:** The incrementCount method uses this.setState to update the state. The prevState parameter is provided to access the current state value, and you return a new object with the updated state.
- **Use State:** The value of the state is accessed using this.state.count and is displayed inside the rendered JSX.

Comparison with useState:

- In functional components, you use useState to declare and manage state, while in class-based components, you use this.state and this.setState.
- While useState returns the current state value and a function to update that state, class-based components rely on the object-oriented pattern with this.state to access state values and this.setState to update them.

The class-based pattern for managing state has been the traditional way to handle state in React before the introduction of hooks. Both approaches are valid, but functional components with hooks (including useState) are now considered more modern and often preferred for their conciseness and readability, aligning with current best practices in React.

Input Event Listeners

Input event listeners are crucial for handling user interactions within forms or other input fields in a React application. They allow you to execute specific functions when users perform actions like typing, focusing, or interacting with input elements.

Here's an example of how you might use input event listeners in functional components. Similar approach will be applied to class-based components:

Functional Component:

Jsx

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

function TextInput() {
  const [text, setText] = useState('');

  const handleInputChange = (e) => {
    setText(e.target.value);
  };

  return (
    <input type="text" value={text} onChange={handleInputChange} />
  );
}

export default TextInput;
```

In this example, we use the onChange event listener to call the handleInputChange function whenever the user types into the input field. The function then updates the text state with the current input value.

Other Common Input Event Listeners:

- onFocus: Triggered when the input field gains focus.
- onBlur: Triggered when the input field loses focus.
- onKeyPress, onKeyDown, onKeyUp: Triggered when keys are pressed, held down, or released.

These event listeners help create responsive and interactive input fields that can handle various user actions, leading to a more engaging user experience. By understanding how to utilize these event listeners, you can create complex forms and input handling mechanisms within your applications, an essential aspect of front-end development.

Array State Project

To Been Researched and Prepared

Counter Project (Example)

To Been Researched and Prepared

Other Useful Information

Virtual DOM

The Virtual DOM (VDOM) is a fundamental concept in React that allows for efficient updates and rendering. Let's dive into what it is and how it works, along with an example.

What is Virtual DOM?

The Virtual DOM is a lightweight in-memory representation of the actual DOM elements. It's a virtual copy of the real DOM, with the same properties and methods.

How Virtual DOM Works:

- Initial Render: React builds a Virtual DOM to represent the UI. This Virtual DOM is a tree structure, mirroring the actual DOM.
- State or Props Change: When there's a change in state or props, React creates a new Virtual DOM tree representing the updated UI.
- Diffing Algorithm: React compares the new Virtual DOM tree with the old one (reconciliation) to figure out exactly what changed. This process is known as "diffing."
- Update the Real DOM: React then updates only the real DOM elements that were changed, rather than re-rendering the entire tree. This minimizes direct

manipulation of the real DOM, which is slow, thus enhancing performance.

Example:

Let's say we have a counter application, and our component looks like this:

Jsx

```
class Counter extends React.Component {
 constructor() {
  super();
  this.state = { count: 0 };
 }
 increment = () => {
  this.setState({ count: this.state.count + 1 });
 };
 render() {
  return (
   <div>
    <p>Count: {this.state.count}</p>
          <button  onClick={this.increment}>Increment</
button>
   </div>
  );
 }
}
```

- Initial Render: React creates a Virtual DOM tree representing this UI.
- On Increment Click: When the "Increment" button is clicked, the state changes, and React creates a new Virtual DOM tree representing the updated count.
- Diffing: React compares the new and old Virtual DOM trees and identifies that only the <p> element displaying the count has changed.
- Update Real DOM: React updates only this <p> element in the real DOM, leaving the rest of the DOM untouched. This is far more efficient than rebuilding the entire DOM.

Why StrictMode

"StrictMode" is a special component in React that can be wrapped around any part of your application to enforce specific checks and warnings during the development build. Here's why StrictMode is valuable:

1. Identifying Unsafe Lifecycles:

StrictMode will warn about legacy lifecycle methods like componentWillMount, componentWillReceiveProps, and componentWillUpdate. These methods are considered unsafe for asynchronous React rendering and will be deprecated in future React versions. By alerting developers about their usage, StrictMode encourages the use of safer alternatives.

2. Detecting Side Effects:

Components should be pure functions of their state and props. If a component's render method has side effects, it

can lead to inconsistencies in the UI. StrictMode detects and warns about unexpected side effects during the rendering phase.

3. Warning About Legacy String Refs:

The use of string refs is considered legacy and can lead to bugs. StrictMode warns about the usage of string refs and encourages the use of callback refs or the React.createRef API.

4. Detecting Unexpected Mutations:

StrictMode helps in finding problems where state or props are being mutated directly instead of using setState or returning a new object, respectively.

5. Checking for Deprecated APIs:

It will also warn you about the usage of deprecated methods and APIs, encouraging developers to update their code to use the latest and recommended approaches.

6. Helping with Concurrent Mode:

As React moves towards more concurrent rendering capabilities, StrictMode ensures that your code follows best practices and doesn't rely on patterns that might break in concurrent rendering.

Usage Example:

You can wrap any part of your application with StrictMode:

Jsx

```jsx
import React from 'react';

function App() {
  return (
    <React.StrictMode>
      <MyComponent />
    </React.StrictMode>
  );
}
```

StrictMode doesn't render any visible UI; it activates additional checks and warnings only in the development build. It's a valuable tool for making your code more robust, maintainable, and future-proof, aligning with modern React practices.

Advanced Components

Rendering Lists

To Been Researched and Prepared

Render Raw HTML

To Been Researched and Prepared

Simple Todo List Project (Example)

To Been Researched and Prepared

Basic Hooks

Hook Rules

To Been Researched and Prepared

useRef Hook

To Been Researched and Prepared

Refs In Class Components

To Been Researched and Prepared

useMemo Hook

To Been Researched and Prepared

useCallback Hook

To Been Researched and Prepared

Custom Hooks

To Been Researched and Prepared

useFetch Custom Hook Project (Example)

To Been Researched and Prepared

useArray Custom Hook Project (Example)

To Been Researched and Prepared

useLocalStorage Custom Hook Project (Example)

Jsx

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

export default function useLocalStorage(key, initialValue) {
  // Get the stored value from localStorage
  const storedValue = localStorage.getItem(key);

  // Check if the stored value is null, if so, use the initial value
    const initial = storedValue !== null ? JSON.parse(storedValue) : initialValue;

  // Define a state variable
  const [value, setValue] = useState(initial);
  // Use useEffect to update localStorage whenever the value changes
  useEffect(() => {
    localStorage.setItem(key, JSON.stringify(value));
  }, [key, value]);
  return [value, setValue];
}
```

Forms

Form Basics

To Been Researched and Prepared

One Way Data Flow

To Been Researched and Prepared

useState vs useRef

To Been Researched and Prepared

Basic Form Validation Project (Example)

To Been Researched and Prepared

Advanced Stateful Components

useReducer Hook

To Been Researched and Prepared

useContext Hook

To Been Researched and Prepared

Context In Class Components

To Been Researched and Prepared

Local State Is Best

To Been Researched and Prepared

Never Store Derived State

To Been Researched and Prepared

Environment Variables

To Been Researched and Prepared

Advanced Todo List (Example)

To Been Researched and Prepared

Routing

Routing Without A Library

To Been Researched and Prepared

React Router Basics

To Been Researched and Prepared

Nested Routes

To Been Researched and Prepared

Dynamic Routes

To Been Researched and Prepared

Loaders

To Been Researched and Prepared

Basic Routing Project (Example)

To Been Researched and Prepared

Actions

To Been Researched and Prepared

Advanced Routing Project (Example)

To Been Researched and Prepared

Useful Library

Form Libraries

To Been Researched and Prepared

React Hook Form Implementation

To Been Researched and Prepared

TanStack Query

To Been Researched and Prepared

Test Cases

To Been Researched and Prepared

About the Author

To Been Write