React JS

By

Praveen Oruganti

Blog: https://praveenorugantitech.blogspot.com
Facebook Group: https://www.facebook.com/groups/268426377837151
Github repo: https://github.com/praveenoruganti
Email: praveenorugantitech@gmail.com

React.js is a JavaScript library created by Facebook and it generally uses jsx(js and html) for writing the code and uses Virtual DOM concept.

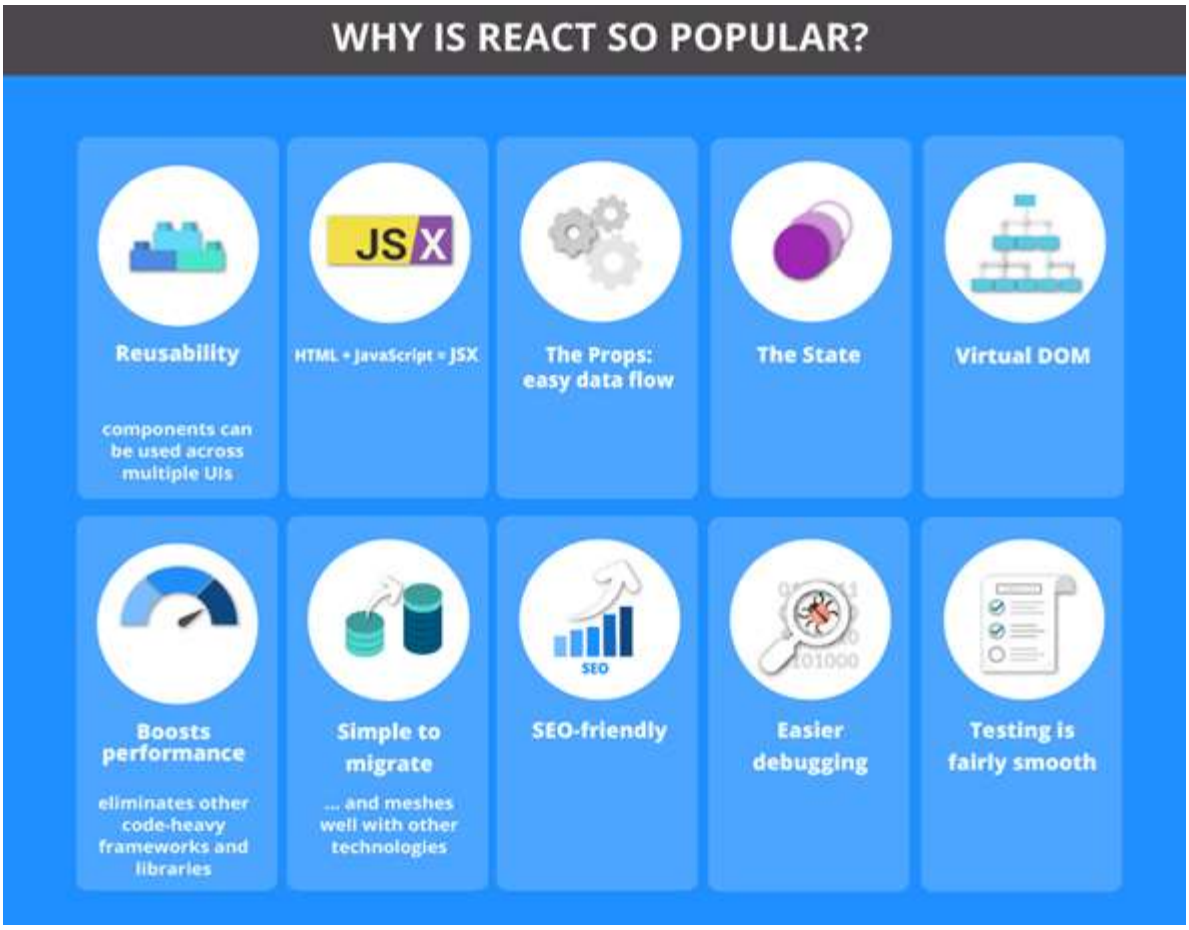
Initial release of react.js happened on May 29, 2013.

React.js is an Open Sourced and used to develop interactive User Interfaces.

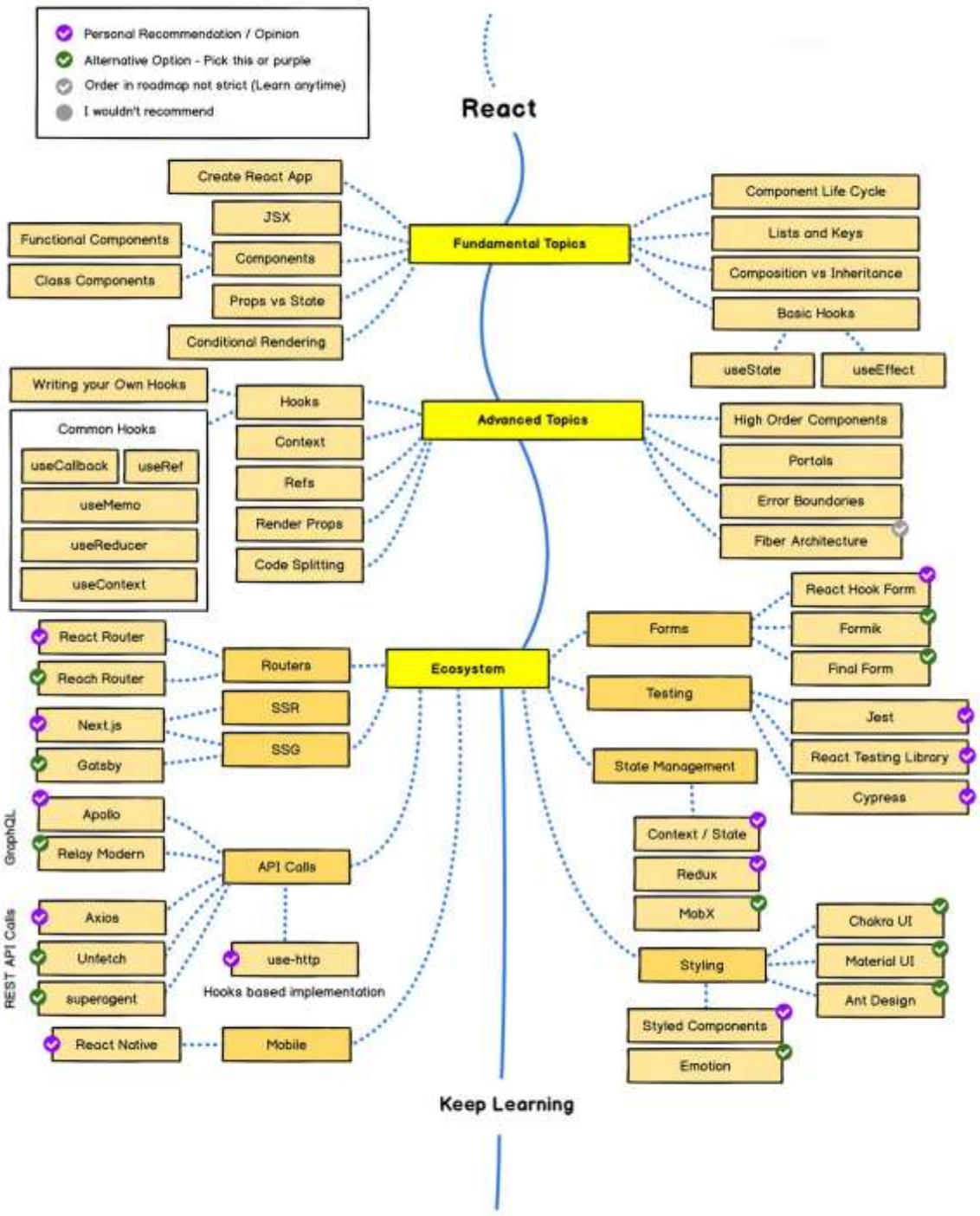
Prerequisites for learning React.js:

1. Knowledge of HTML & CSS
2. Knowledge of JavaScript and ES6 standards(https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript)
3. Some knowledge about the DOM
4.Some knowledge about Node & npm (and installation)

Generally we will use React.js for building single paged application (spa)



Let’s see the React JS roadmap and go deeper into the concepts



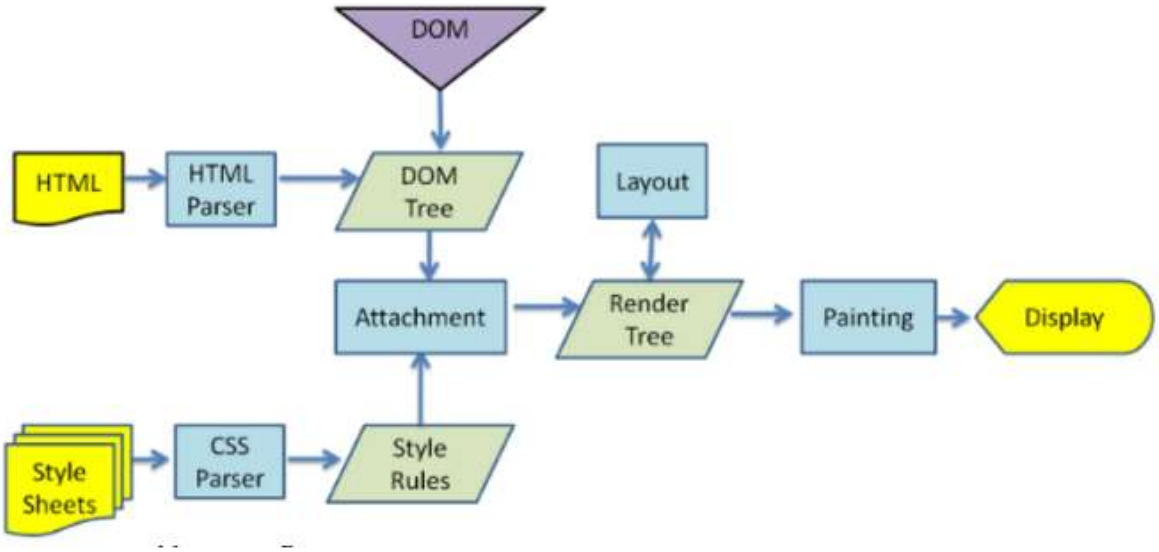
Blog: https://praveenoruganti tech.blogspot.com
Facebook Group: https://www.facebook.com/groups/268426377837151
Github repo: https://github.com/praveenoruganti
Email : praveenorugantitech@gmail.com

Aspects of react.js

1. Virtual DOM
2. Data binding
3. Server side rendering

What is meant by Virtual DOM?

Let's see how exactly renders the webpage,



Rendering engines which is responsible for displaying or rendering the webpage on the browser screen parses the HTML page to create DOM.

If you see above screenshot, updating a Real DOM does not involves just updating the DOM but, it involves a lot of other process.

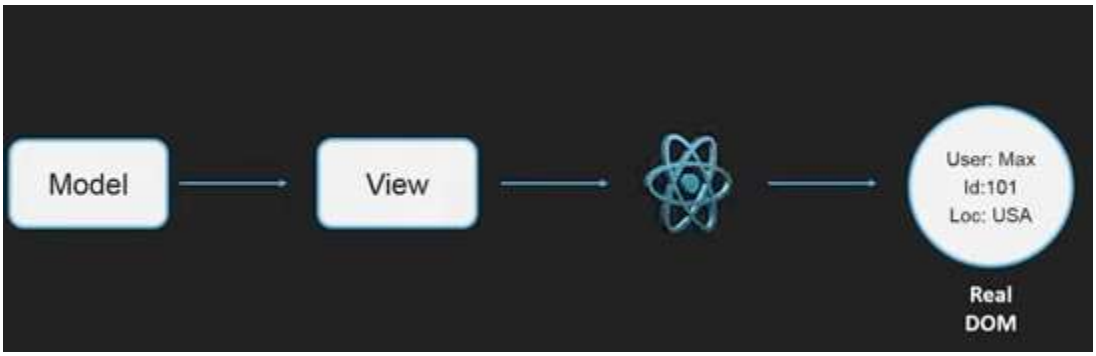
Also, each of the above steps runs for each update of the real DOM i.e. if we update the Real DOM 10 times each of the above step will repeat 10 times. This is why updating Real DOM is slow.

To solve the above problem, Virtual DOM came into the picture.

Virtual DOM is in-memory representation of Real DOM. It is lightweight JavaScript object which is copy of Real DOM.

Like an actual DOM, Virtual DOM is a node tree that lists the elements and their attributes and content as Objects and their properties.

Model gives data to view, if the DOM is empty then React will create a DOM for it.



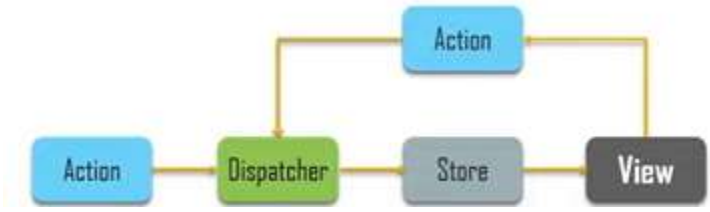
Updating virtual DOM in ReactJS is faster because ReactJS uses

- ✓ Efficient diff algorithm
- ✓ Batched update operations
- ✓ Efficient update of sub tree only
- ✓ Uses observable instead of dirty checking to detect change

What is data binding?

React.js follows unidirectional data flow or one way data binding.

Throughout the application the data flows in a single direction which gives you better control over it.



What is Server side rendering?

Server side rendering allows you to pre-render the initial state of react components at the server side only.

React.js installation

We can use online editors like

1. https://codesandbox.io/
2. https://jsfiddle.net/reactjs/

3. https://codepen.io/

Now let's see what are all the dependencies needed for react.js installation in your environment.

- ✓ Download and install npm : https://nodejs.org/en/
- ✓ Download and install editor https://code.visualstudio.com/docs/?dv=win

Main components to build a react application are

webpack.config.js -> contains information about the dependencies and the files from where browser start rendering form

HTML file -> contains the html template which is used by browser to render the elements on the webpage

JSX file -> contains description off what all components we want to display on our webpage and how they will behave.

Create React App

For creating application, use the below command

npx create-react-app praveenoruganti-reactjs-samples

npm start

once you run the above command you will see the below

All About JSX

Consider this variable declaration

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

This funny tag syntax is neither a string nor HTML.

It is called JSX, and it is a syntax extension to JavaScript. We recommend using it with React to describe what the UI should look like. JSX may remind you of a template language, but it comes with the full power of JavaScript.

JSX produces React “elements”.

Why JSX?

React embraces the fact that rendering logic is inherently coupled with other UI logic: how events are handled, how the state changes over time, and how the data is prepared for display.

Instead of artificially separating technologies by putting markup and logic in separate files, React separates concerns with loosely coupled units called “components” that contain both. We will come back to components in a further section, but if you’re not yet comfortable putting markup in JS, this talk might convince you otherwise.

React doesn’t require using JSX, but most people find it helpful as a visual aid when working with UI inside the JavaScript code. It also allows React to show more useful error and warning messages.

Embedding Expressions in JSX

In the example below, we declare a variable called name and then use it inside JSX by wrapping it in curly braces:

```
const name = 'Praveen Oruganti';

const element = <h1>Hello, {name}</h1>;

ReactDOM.render(element, document.getElementById('root'));
```

You can put any valid JavaScript expression inside the curly braces in JSX. For example, 2 + 2, user.firstName, or formatName(user) are all valid JavaScript expressions.

In the example below, we embed the result of calling a JavaScript function, formatName(user), into an <h1> element.

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}
const user = {
  firstName: 'Praveen',
  lastName: 'Oruganti'
};
const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);
ReactDOM.render( element,document.getElementById('root'));
```

JSX is an Expression Too

After compilation, JSX expressions become regular JavaScript function calls and evaluate to JavaScript objects.

This means that you can use JSX inside of if statements and for loops, assign it to variables, accept it as arguments, and return it from functions:

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
```

}

Specifying Attributes with JSX

You may use quotes to specify string literals as attributes:

```
const element = <div tabIndex="0"></div>;
```

You may also use curly braces to embed a JavaScript expression in an attribute:

```
const element = <img src={user.avatarUrl}></img>;
```

Don’t put quotes around curly braces when embedding a JavaScript expression in an attribute. You should either use quotes (for string values) or curly braces (for expressions), but not both in the same attribute.

Warning:

Since JSX is closer to JavaScript than to HTML, React DOM uses camelCase property naming convention instead of HTML attribute names.

For example, class becomes className in JSX, and tabindex becomes tabIndex.

Specifying Children with JSX

If a tag is empty, you may close it immediately with />, like XML:

```
const element = <img src={user.avatarUrl} />;
```

JSX tags may contain children:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX Prevents Injection Attacks

It is safe to embed user input in JSX:

```
const title = response.potentiallyMaliciousInput;

// This is safe:

const element = <h1>{title}</h1>;
```

By default, React DOM escapes any values embedded in JSX before rendering them. Thus it ensures that you can never inject anything that's not explicitly written in your application. Everything is converted to a string before being rendered. This helps prevent XSS (cross-site-scripting) attacks.

JSX Represents Objects

Babel compiles JSX down to React.createElement() calls.

These two examples are identical:

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);

const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

React.createElement() performs a few checks to help you write bug-free code but essentially it creates an object like this:

```
// Note: this structure is simplified

const element = {

  type: 'h1',

  props: {

    className: 'greeting',

    children: 'Hello, world!'

  }

};
```

These objects are called “React elements”. You can think of them as descriptions of what you want to see on the screen. React reads these objects and uses them to construct the DOM and keep it up to date.

Rendering Elements

Elements are the smallest building blocks of React apps.

An element describes what you want to see on the screen:

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

Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.

Rendering an Element into the DOM

Let’s say there is a somewhere in your HTML file:

```
<div id="root"></div>
```

We call this a “root” DOM node because everything inside it will be managed by React DOM.

Applications built with just React usually have a single root DOM node. If you are integrating React into an existing app, you may have as many isolated root DOM nodes as you like.

To render a React element into a root DOM node, pass both to ReactDOM.render():

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

ReactDOM.render(element, document.getElementById('root'));
```

It displays “Hello, world” on the page.

Updating the Rendered Element

React elements are immutable. Once you create an element, you can’t change its children or attributes. An element is like a single frame in a movie: it represents the UI at a certain point in time.

With our knowledge so far, the only way to update the UI is to create a new element, and pass it to ReactDOM.render().

Consider this ticking clock example:

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}

setInterval(tick, 1000);
```

It calls ReactDOM.render() every second from a setInterval() callback.

In practice, most React apps only call ReactDOM.render() once.

React Only Updates What's Necessary

React DOM compares the element and its children to the previous one, and only applies the DOM updates necessary to bring the DOM to the desired state.

Even though we create an element describing the whole UI tree on every tick, only the text node whose contents have changed gets updated by React DOM.

In our experience, thinking about how the UI should look at any given moment, rather than how to change it over time, eliminates a whole class of bugs.

React Components

Components let you split the UI into independent, reusable pieces, and think about each piece in isolation.

Conceptually, components are like JavaScript functions. They accept arbitrary inputs (called "props") and return React elements describing what should appear on the screen.

Function and Class Components

Functional Components

- ✓ Functional components are basic JavaScript functions. These are typically arrow functions but can also be created with the regular function keyword.
- ✓ Sometimes referred to as "dumb" or "stateless" components as they simply accept data and display them in some form; that is they are mainly responsible for rendering UI.
- ✓ React lifecycle methods (for example, componentDidMount) cannot be used in functional components.
- ✓ There is no render method used in functional components.
- ✓ These are mainly responsible for UI and are typically presentational only (For example, a Button component).
- ✓ Functional components can accept and use props.
- ✓ Functional components should be favored if you do not need to make use of React state.

The simplest way to define a component is to write a JavaScript function

```
import React from 'react';

const Greeting = (props) => {
```

```
    return (<h1>Hello, {props.name}</h1>)
}
export default Greeting;
```

This function is a valid React component because it accepts a single “props” (which stands for properties) object argument with data and returns a React element. We call such components “function components” because they are literally JavaScript functions.

Class Components

- ✓ Class components make use of ES6 class and extend the Component class in React.
- ✓ Sometimes called “smart” or “stateful” components as they tend to implement logic and state.
- ✓ React lifecycle methods can be used inside class components (for example, componentDidMount).
- ✓ You pass props down to class components and access them with this.props

You can also use an ES6 class to define a component

```
import React from 'react';

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

The above two components are equivalent from React’s point of view.

Note: Always start component names with a capital letter.

React treats components starting with lowercase letters as DOM tags. For example, represents an HTML div tag, but <Welcome /> represents a component and requires Welcome to be in scope.

State and Props

What are props?

Props is short for properties and they are used to pass data between React components. React's data flow between components is uni-directional (from parent to child only).

How do you pass data with props?

Here is an example of how data can be passed by using props:

```
class ParentComponent extends Component {

   render() {

      return (

         <ChildComponent name="First Child" />

      );

   }

}

const ChildComponent = (props) => {

   return <p>{props.name}</p>;

};
```

Firstly, we need to define/get some data from the parent component and assign it to a child component's "prop" attribute.

```
<ChildComponent name="First Child" />
```

"name" is a defined prop here and contains text data. Then we can pass data with props like we're giving an argument to a function:

```
const ChildComponent = (props) => {

  // statements

};
```

And finally, we use dot notation to access the prop data and render it:

return {props.name} ;

What is state?

React has another special built-in object called state, which allows components to create and manage their own data. So unlike props, components cannot pass data with state, but they can create and manage it internally.

Here is an example showing how to use state:

```
class Test extends React.Component {
    constructor() {
        this.state = {
            id: 1,
            name: "test"
        };
    }
    render() {
        return (
            <div>
              <p>{this.state.id}</p>
              <p>{this.state.name}</p>
            </div>
        );
    }
}
```

How do you update a component's state?

State should not be modified directly, but it can be modified with a special method called setState( ).

```
this.state.id = "2020"; // wrong

this.setState({        // correct

   id: "2020"

});
```

What happens when state changes?

OK, why must we use setState( )? Why do we even need the state object itself? If you're asking these questions, don't worry – you'll understand state soon :) Let me answer.

A change in the state happens based on user-input, triggering an event, and so on. Also, React components (with state) are rendered based on the data in the state. State holds the initial information.

So when state changes, React gets informed and immediately re-renders the DOM – not the whole DOM, but only the component with the updated state. This is one of the reasons why React is fast.

And how does React get notified? You guessed it: with setState( ). The setState( ) method triggers the re-rendering process for the updated parts. React gets informed, knows which part(s) to change, and does it quickly without re-rendering the whole DOM.

In summary, there are 2 important points we need to pay attention to when using state:

- ✓ State shouldn't be modified directly – the setState( ) should be used
- ✓ State affects the performance of your app, and therefore it shouldn't be used unnecessarily

Can I use state in every component?

Another important question you might ask about state is where exactly we can use it. In the early days, state could only be used in class components, not in functional components.

That's why functional components were also known as stateless components. However, after the introduction of React Hooks, state can now be used both in class and functional components.

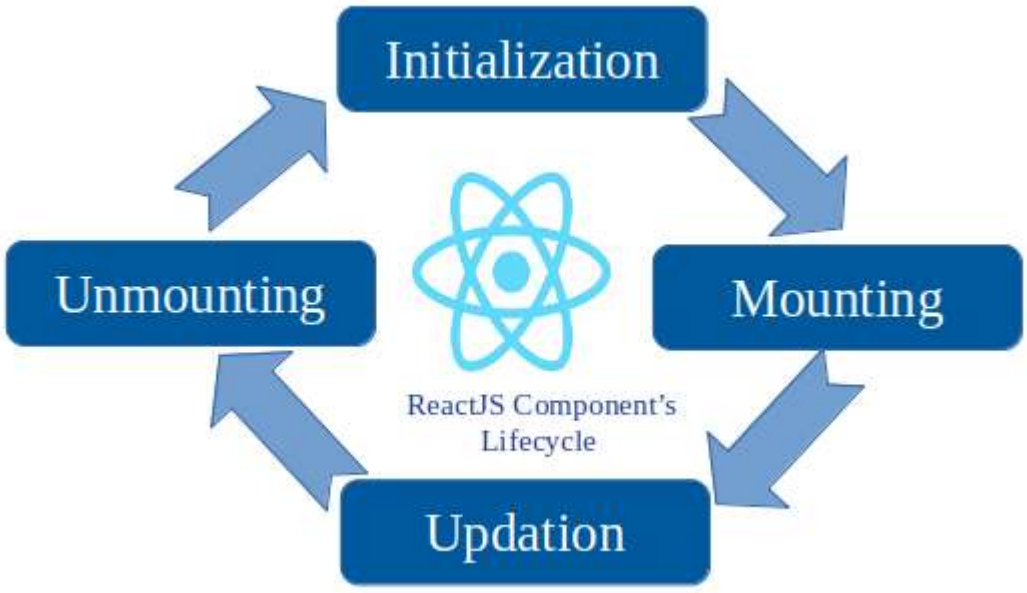
If your project is not using React Hooks, then you can only use state in class components.

What are the differences between props and state?

Finally, let's recap and see the main differences between props and state:

- ✓ Components receive data from outside with props, whereas they can create and manage their own data with state
- ✓ Props are used to pass data, whereas state is for managing data
- ✓ Data from props is read-only, and cannot be modified by a component that is receiving it from outside
- ✓ State data can be modified by its own component, but is private (cannot be accessed from outside)
- ✓ Props can only be passed from parent component to child (unidirectional flow)
- ✓ Modifying state should happen with the setState ( ) method

Now let's see React Component's lifecycle.



Components are created (mounted on the DOM), grow by updating, and then die (unmount on DOM). This is referred to as a component lifecycle.

There are different lifecycle methods that React provides at different phases of a component's life. React automatically calls the responsible method according to the phase in which the component is. These methods give us better control over our component and we can manipulate them using these methods.

Lifecycle Methods

A component's lifecycle is broadly classified into four parts:

- ✓ initialization
- ✓ mounting
- ✓ updating
- ✓ unmounting

Let's discuss the different lifecycle methods that are available at these different phases (i.e., initialization, mounting, updating & unmounting).

Initialization

This is the phase in which the component is going to start its journey by setting up the state (see below) and the props. This is usually done inside the constructor method (see below to understand the initialization phase better).

```
class Initialize extends React.Component {

   constructor(props)   {

   // Calling the constructor of

   // Parent Class React.Component

   super(props);

   // initialization process

   this.state = {

      date : new Date(),

      clickedStatus: false

    };

}
```

Mounting

The name is self-explanatory. Mounting is the phase in which our React component mounts on the DOM (i.e., is created and inserted into the DOM).

This phase comes onto the scene after the initialization phase is completed. In this phase, our component renders the first time. The methods that are available in this phase are:

1. componentWillMount()

This method is called just before a component mounts on the DOM or the render method is called. After this method, the component gets mounted.

Note: You should not make API calls or any data changes using this.setstate in this method because it is called before the render method. So, nothing can be done with the DOM (i.e. updating the data with API response) as it has not been mounted. Hence, we can't update the state with the API response.

2. componentDidMount()

This method is called after the component gets mounted on the DOM. Like componentWillMount, it is called once in a lifecycle. Before the execution of this method, the render method is called (i.e., we can access the DOM). We can make API calls and update the state with the API response.

Have a look to understand these mounting methods:

```
class LifeCycle extends React.Component {

  componentWillMount() {

     console.log('Component will mount!')

   }

  componentDidMount() {

     console.log('Component did mount!')

     this.getList();

   }
```

```
  getList=()=>{

   /*** method to make api call***

  }

  render() {

      return (

         <div>

            <h3>Hello mounting methods!</h3>

         </div>

      );

   }

}
```

Updating

This is the third phase through which our component passes. After the mounting phase where the component has been created, the update phase comes into the scene. This is where component's state changes and hence, re-rendering takes place.

In this phase, the data of the component (state & props) updates in response to user events like clicking, typing and so on. This results in the re-rendering of the component. The methods that are available in this phase are:

1.shouldComponentUpdate()

This method determines whether the component should be updated or not. By default, it returns true. But at some point, if you want to re-render the component on some condition, then shouldComponentUpdate method is the right place.

Suppose, for example, you want to only re-render your component when there is a change in prop — then utilize the power of this method. It receives arguments like nextProps and nextState which help us decide whether to re-render by doing a comparison with the current prop value.

2. componentWillUpdate()

Like other methods, its name is also self-explanatory. It is called before the re-rendering of the component takes place. It is called once after the 'shouldComponentUpdate' method. If you want to perform some calculation before re-rendering of the component and after updating the state and prop, then this is the best place to do it. Like the 'shouldComponentUpdate' method, it also receives arguments like nextProps and nextState

3. ComponentDidUpdate()

This method is called just after the re-rendering of the component. After the new (updated) component gets updated on the DOM, the 'componentDidUpdate' method is executed. This method receives arguments like prevProps and prevState.

Have a look to understand the updating methods better:

```
class LifeCycle extends React.Component {

    constructor(props)

    {

      super(props);

       this.state = {

          date : new Date(),

          clickedStatus: false,

          list:[]

       };

    }

    componentWillMount() {

        console.log('Component will mount!')

     }

    componentDidMount() {
```

```
    console.log('Component did mount!')
    this.getList();
  }
 getList=()=>{
  /*** method to make api call***
  fetch('https://api.mydomain.com')
    .then(response => response.json())
    .then(data => this.setState({ list:data }));
 }
  shouldComponentUpdate(nextProps, nextState){
    return this.state.list!==nextState.list
   }
  componentWillUpdate(nextProps, nextState) {
    console.log('Component will update!');
  }
  componentDidUpdate(prevProps, prevState) {
    console.log('Component did update!')
  }
 render() {
    return (
      <div>
        <h3>Hello Mounting Lifecycle Methods!</h3>
```

```
            </div>
        );
    }
}
```

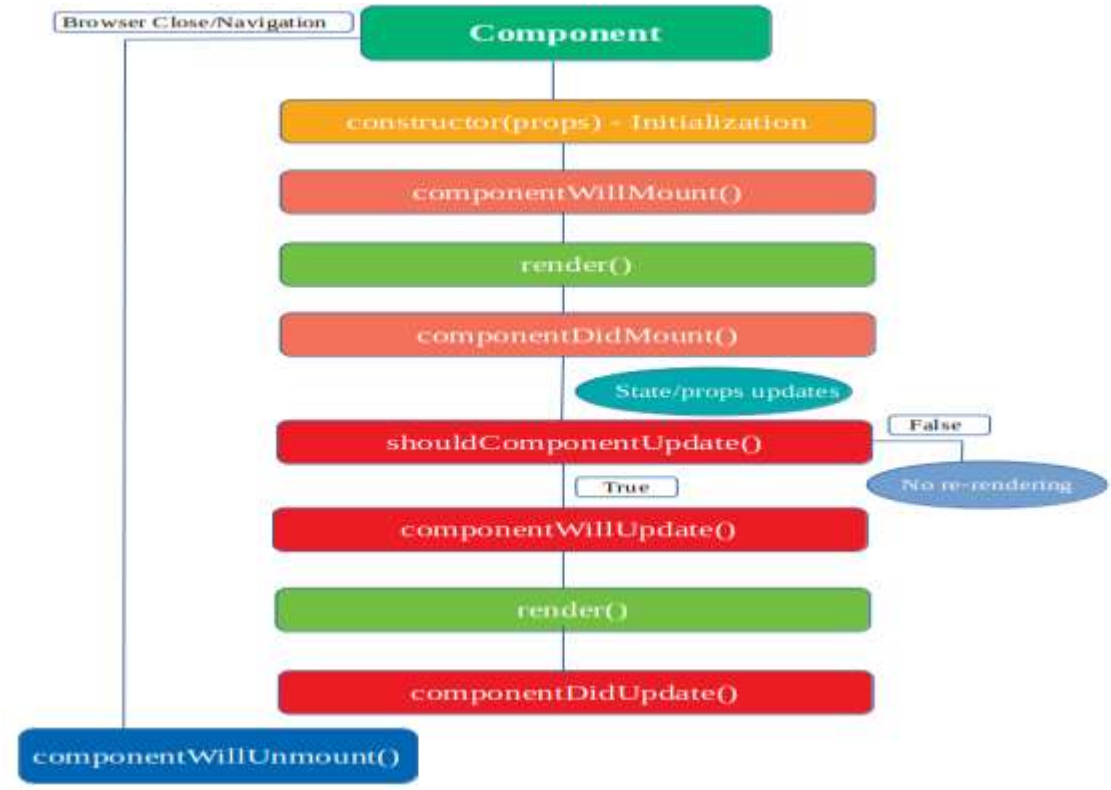
Unmounting

This is the last phase in the component's lifecycle. As the name clearly suggests, the component gets unmounted from the DOM in this phase. The method that is available in this phase is:

1. componentWillUnmount()

This method is called before the unmounting of the component takes place. Before the removal of the component from the DOM, 'componentWillUnMount' executes. This method denotes the end of the component's lifecycle.

Here is a flowchart representation of lifecycle methods:



Handling Events

Handling events with React elements is very similar to handling events on DOM elements. There are some syntax differences:

- ✓ React events are named using camelCase, rather than lowercase.
- ✓ With JSX you pass a function as the event handler, rather than a string.

For example, the HTML:

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

is slightly different in React:

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

Another difference is that you cannot return false to prevent default behavior in React. You must call preventDefault explicitly. For example, with plain HTML, to prevent the default link behavior of opening a new page, you can write:

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

In React, this could instead be:

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('The link was clicked.');
```

```
  }

  return (

    <a href="#" onClick={handleClick}>

      Click me

    </a>

  );

}
```

Here, e is a synthetic event. React defines these synthetic events according to the W3C spec, so you don't need to worry about cross-browser compatibility. See the SyntheticEvent reference guide to learn more.

When using React, you generally don't need to call addEventListener to add listeners to a DOM element after it is created. Instead, just provide a listener when the element is initially rendered.

When you define a component using an ES6 class, a common pattern is for an event handler to be a method on the class. For example, this Toggle component renders a button that lets the user toggle between "ON" and "OFF" states:

```
class Toggle extends React.Component {

  constructor(props) {

    super(props);

    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback

    this.handleClick = this.handleClick.bind(this);

  }

handleClick() {

    this.setState(state => ({
```

```
      isToggleOn: !state.isToggleOn
    }));
  }
  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

You have to be careful about the meaning of this in JSX callbacks. In JavaScript, class methods are not bound by default. If you forget to bind this.handleClick and pass it to onClick, this will be undefined when the function is actually called.

This is not React-specific behavior; it is a part of how functions work in JavaScript. Generally, if you refer to a method without () after it, such as onClick={this.handleClick}, you should bind that method.

If calling bind annoys you, there are two ways you can get around this. If you are using the experimental public class fields syntax, you can use class fields to correctly bind callbacks:

```
class LoggingButton extends React.Component {
```

```
  // This syntax ensures `this` is bound within handleClick.
  // Warning: this is *experimental* syntax.
  handleClick = () => {
    console.log('this is:', this);
  }
  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

This syntax is enabled by default in Create React App.

If you aren't using class fields syntax, you can use an arrow function in the callback:

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
```

```
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

The problem with this syntax is that a different callback is created each time the LoggingButton renders. In most cases, this is fine. However, if this callback is passed as a prop to lower components, those components might do an extra re-rendering. We generally recommend binding in the constructor or using the class fields syntax, to avoid this sort of performance problem.

Passing Arguments to Event Handlers

Inside a loop, it is common to want to pass an extra parameter to an event handler. For example, if id is the row ID, either of the following would work:

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>

<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

The above two lines are equivalent, and use arrow functions and Function.prototype.bind respectively.

In both cases, the e argument representing the React event will be passed as a second argument after the ID. With an arrow function, we have to pass it explicitly, but with bind any further arguments are automatically forwarded.

Conditional Rendering

In React, you can create distinct components that encapsulate behavior you need. Then, you can render only some of them, depending on the state of your application.

Conditional rendering in React works the same way conditions work in JavaScript. Use JavaScript operators like if or the conditional operator to create elements representing the current state, and let React update the UI to match them.

Consider these two components:

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

We'll create a Greeting component that displays either of these components depending on whether a user is logged in:

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

ReactDOM.render(
  // Try changing to isLoggedIn={true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

This example renders a different greeting depending on the value of isLoggedIn prop.

Element Variables

You can use variables to store elements. This can help you conditionally render a part of the component while the rest of the output doesn't change.

Consider these two new components representing Logout and Login buttons:

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}
```

In the example below, we will create a stateful component called LoginControl.

It will render either <LoginButton /> or <LogoutButton /> depending on its current state. It will also render a <Greeting /> from the previous example:

```
class LoginControl extends React.Component {
  constructor(props) {
```

```
  super(props);
  this.handleLoginClick = this.handleLoginClick.bind(this);
  this.handleLogoutClick = this.handleLogoutClick.bind(this);
  this.state = {isLoggedIn: false};
}

handleLoginClick() {
  this.setState({isLoggedIn: true});
}

handleLogoutClick() {
  this.setState({isLoggedIn: false});
}

render() {
  const isLoggedIn = this.state.isLoggedIn;
  let button;
  if (isLoggedIn) {
    button = <LogoutButton onClick={this.handleLogoutClick} />;
  } else {
    button = <LoginButton onClick={this.handleLoginClick} />;
  }
```

```
    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

ReactDOM.render(
  <LoginControl />,
  document.getElementById('root')
);
```

While declaring a variable and using an if statement is a fine way to conditionally render a component, sometimes you might want to use a shorter syntax. There are a few ways to inline conditions in JSX, explained below.

Inline If with Logical && Operator

You may embed expressions in JSX by wrapping them in curly braces. This includes the JavaScript logical && operator. It can be handy for conditionally including an element:

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
```

```
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}
const messages = ['React', 'Re: React', 'Re:Re: React'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

It works because in JavaScript, true && expression always evaluates to expression, and false && expression always evaluates to false.

Therefore, if the condition is true, the element right after && will appear in the output. If it is false, React will ignore and skip it.

Inline If-Else with Conditional Operator

Another method for conditionally rendering elements inline is to use the JavaScript conditional operator condition ? true : false.

In the example below, we use it to conditionally render a small block of text.

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
```

```
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

It can also be used for larger expressions although it is less obvious what's going on:

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}
```

Just like in JavaScript, it is up to you to choose an appropriate style based on what you and your team consider more readable. Also remember that whenever conditions become too complex, it might be a good time to extract a component.

Preventing Component from Rendering

In rare cases you might want a component to hide itself even though it was rendered by another component. To do this return null instead of its render output.

In the example below, the <WarningBanner /> is rendered depending on the value of the prop called warn. If the value of the prop is false, then the component does not render:

```
function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
```

```
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

ReactDOM.render(
  <Page />,
  document.getElementById('root')
);
```

Returning null from a component's render method does not affect the firing of the component's lifecycle methods. For instance componentDidUpdate will still be called.

React Hooks

Hooks are a new addition in React 16.8. They let you use state and other React features without writing a class.

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

export default function SampleCountHook() {
    // Declare a new state variable, which we'll call "count"
    const [count, setCount] = useState(0);
    return (
        <div>
            <p>You clicked {count} times</p>
            <button onClick={() => setCount(count + 1)}>
                Click me
            </button>
        </div>
    );
}
```

This example renders a counter. When you click the button, it increments the value:

Here, useState is a Hook (we'll talk about what this means in a moment). We call it inside a function component to add some local state to it. React will preserve this state between re-renders. useState returns a pair: the current state value and a function that lets you update it. You can call this function from an event handler or somewhere else. It's similar to this.setState in a class, except it doesn't merge the old and new state together. (We'll show an example comparing useState to this.state in Using the State Hook.)

The only argument to useState is the initial state. In the example above, it is 0 because our counter starts from zero. Note that unlike this.state, the state here doesn't have to be an object — although it can be if you want. The initial state argument is only used during the first render.

Declaring multiple state variables

You can use the State Hook more than once in a single component:

function ExampleWithManyStates() {

// Declare multiple state variables!

```
  const [age, setAge] = useState(42);

  const [fruit, setFruit] = useState('banana');

  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);

  // ...

}
```

But what is a Hook?

Hooks are functions that let you “hook into” React state and lifecycle features from function components. Hooks don’t work inside classes — they let you use React without classes. (We don’t recommend rewriting your existing components overnight but you can start using Hooks in the new ones if you’d like.)

Hooks are JavaScript functions, but they have two additional rules:

- ✓ Only call Hooks at the top level. Don’t try to call Hooks inside loops, conditions, or nested functions.
- ✓ Only call Hooks from React function components. Don’t try to call Hooks from regular JavaScript functions.

React provides a few built-in Hooks like useState. You can also create your own Hooks to reuse stateful behavior between different components. We’ll look at the built-in Hooks first.

1. useState, as the name describes, is a hook that allows you to use state in your function. We define it as follows:

```
const [ someState, updateState ] = useState(initialState)
```

Let’s break this down:

- ✓ someState: lets you access the current state variable, someState
- ✓ updateState: function that allows you to update the state — whatever you pass into it becomes the new someState
- ✓ initialState: what you want someState to be upon initial render

2.useEffect

useEffect is another hook that handles componentDidUpdate, componentDidMount, and componentWillUnmount all in one call. If you need to fetch data, for example, you could useEffect to do so, as seen below.

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

const HooksExample = () => {
    const [data, setData] = useState();

    useEffect(() => {
        const fetchGithubData = async (name) => {
            const result = await axios(`https://api.github.com/users/${name}/repo
s`)
            setData(result.data)
        }
        fetchGithubData('praveenoruganti')
    }, [data])

    return (
        <div className="App">
            {data && (
                data.map(item => <p>{item.name}</p>)
            )}
        </div>
    )
}

export default HooksExample;
```

Taking a look at useEffect we see:

- ✓ First argument: A function. Inside of it, we fetch our data using an async function and then set data when we get results.
- ✓ Second argument: An array containing data. This defines when the component updates. As I mentioned before, useEffect runs when componentDidMount, componentWillUnmount, andcomponentDidUpdate would normally run. Inside the first argument, we've set some state, which would traditionally cause componentDidUpdate to run. As a result, useEffect would run again if we did not

have this array. Now, useEffect will run on componentDidMount, componentWillUnmount, and if data was updated, componentDidUpdate. This argument can be empty— you can choose to pass in an empty array. In this case, only componentDidMount and componentWillUnmount will ever fire. But, you do have to specify this argument if you set some state inside of it.

3.useReducer

For those of you who use Redux, useReducer will probably be familiar. useReducer takes in two arguments — a reducer and an initial state. A reducer is a function that you can define that takes in the current state and an "action". The action has a type, and the reducer uses a switch statement to determine which block to execute based on the type. When it finds the correct block, it returns the state but with the modifications you define depending on the type. We can pass this reducer into useReducer, and then use this hook like this:

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

You use dispatch to say what action types you want to execute, like this:

dispatch({ type: name})

useReducer is the best solution in React for handling complex state interactions so let's look at how we can convert a component from useState to useReducer.

useReducer is normally used when you have to manage complex states — such as the signup form below.

```
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();
  }
}
export default function CounterReducer() {
  const [state, dispatch] = useReducer(reducer, initialState);
```

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

This hook has a lot of additional applications, including allowing us to specify a few reducers throughout our application and then reusing them for each of our components, changing based on what happens in those components. On a high level, this is similar to Redux's functionality — so we may be able to avoid using Redux for relatively simpler applications.

Note

React guarantees that dispatch function identity is stable and won't change on re-renders. This is why it's safe to omit from the useEffect or useCallback dependency list.

4.useContext

React released the Context API as a much needed solution for state that spans across multiple nested components. Unfortunately, the API for context was a bit bulky and difficult to use in class components. With the release of hooks, the React team decided to re-think how you interact with context and drastically simplified the code through the use of the useContext hook**.**

What Is The Context API?

As you already know, React uses state to store data and props to pass data between components. This works well for handling local state and for passing simple props between parent/child components. This system breaks down when you start to have global state or props that need to be passed to deeply nested components. With just props and state you end up having to resort to prop drilling which is when you pass down props through a bunch of different components so they can get to one single component far down the hierarchy.

This is where the Context API comes in. With the context API you can specify certain pieces of data that will be available to all components nested inside the context with no need to pass this data through each component. It is essentially semi-global state that is available anywhere inside the context.

In order to use context in a function component you no longer need to wrap your JSX in a consumer. Instead all you need to do is pass your context to the useContext hook and it will do all the magic for you. Here is an example.

```
import React, { useState } from 'react';
import App from './App';

export const multiStepContext = React.createContext();
const StepContext = () => {
    const [currentStep, setStep] = useState(1);
    const [userData, setUserData] = useState([]);
    const [finalData, setFinalData] = useState([]);

    function submitData(){
        setFinalData(finalData => [...finalData,userData]);
        setUserData('');
        setStep(1);
    }
    return (
        <div>
            <multiStepContext.Provider value={{ currentStep, setStep, userData, s
etUserData, finalData, setFinalData, submitData }}>
                <App />
            </multiStepContext.Provider>
        </div>
    )
}

export default StepContext;
```

```
import React, { useContext } from 'react';
import './App.css';
import FirstStep from './components/FirstStep';
import SecondStep from './components/SecondStep';
import ThirdStep from './components/ThirdStep';
import { Stepper, StepLabel, Step } from '@material-ui/core';
import { multiStepContext } from './StepContext';
import DisplayData from './components/DisplayData';
import AppBar from '@material-ui/core/AppBar';
import Toolbar from '@material-ui/core/Toolbar';
```

```
import Typography from '@material-ui/core/Typography';

function App() {
  const { currentStep, finalData } = useContext(multiStepContext);
  function showStep(step) {
    switch (step) {
      case 1: return <FirstStep />
      case 2: return <SecondStep />
      case 3: return <ThirdStep />
    }
  }
  return (
    <div className="App">
      <div className="App-header">
        <AppBar>
          <Toolbar>
            <Typography variant="h6">Praveen Oruganti Multi Step Form</Typography
>
          </Toolbar>
        </AppBar><br/><br/><br/><br/>
        <div className="center-stepper">
          <Stepper style={{ width: '18%' }} activeStep={currentStep - 1} orientat
ion='horizontal'>
            <Step>
              <StepLabel></StepLabel>
            </Step>
            <Step>
              <StepLabel></StepLabel>
            </Step>
            <Step>
              <StepLabel></StepLabel>
            </Step>
          </Stepper>
        </div>
        {showStep(currentStep)}
        <br />
        {finalData.length > 0 ? <DisplayData /> : ''}

      </div>
    </div>
  );
}
```

```
export default App;
```

With the help of useContext we were able to cut out all the consumer portion of the context and remove all the complex nesting. Now context works just like a normal function where you call the context and it will give you the values inside of it for you to use later in the code. This drastically simplifies code related to context and makes working with context so much more enjoyable.

Also, setting up a context provider for use with the useContext hook is exactly the same as you would do for a normal context consumer, so you can use all the same code for the context provider portion of the class component example at the start of the article.

In the end the useContext hook is very simple. All it does is provide a nice interface for consuming context, but that interface is so much better than the original context consumer interface. Next time you are working with context in your application make sure to give useContext a try.

5. useRef

In order to work with refs in React you need to first initialize a ref which is what the useRef hook is for. This hook is very straightforward, and takes an initial value as the only argument.

useRef(initialValue)

This hook then returns a ref for you to work with.

const myRef = useRef(null)

In the above example we have created a ref called myRef and set its default value to null. This means that myRef is now equal to an object that looks like this.

{ current: null }

This is because a ref is always an object with a single .current property which is set to the current value of the ref. If we were to instead create a ref with a default value of 0 it would look like this.

const myRef = useRef(0)

console.log(myRef)

```
// { current: 0 }
```

Now this seems like a lot of work in order to save a single value, but what makes refs so powerful is the fact that they are persisted between renders. I like to think of refs very similarly to state, since they persist between renders, but refs do not cause a component to re-render when changed.

Imagine that we want to count the number of times a component re-renders. Here is the code to do so with state and refs.

```
function State() {

  const [rerenderCount, setRerenderCount] = useState(0);

  useEffect(() => {

    setRerenderCount(prevCount => prevCount + 1);

  });

  return <div>{rerenderCount}</div>;

}

function Ref() {

  const rerenderCount = useRef(0);

  useEffect(() => {

    rerenderCount.current = rerenderCount.current + 1;

  });

  return <div>{rerenderCount.current}</div>;

}
```

Both of these components will correctly display the number of times a component has been re-rendered, but in the state example the component will infinitely re-render itself since setting the state causes the component to re-render. The ref example on the other hand will only render once since setting the value of a ref does cause any re-renders.

How To Use Refs

Now that we understand refs are just an object for storing a value that persists between renders, let's talk about when you would need to use a ref.

The most common use case for refs in React is to reference a DOM element. Because of how common this use case is every DOM element has a ref property you can use for setting a ref to that element. For example, if you wanted to focus an input element whenever a button was clicked you could use a ref to do that.

```
function Component() {

  const inputRef = useRef(null)

  const focusInput = () => {

    inputRef.current.focus()

  }

  return (

    <>

      <input ref={inputRef} />

      <button onClick={focusInput}>Focus Input</button>

    </>

  )

}
```

As you can see in the code above we use the ref property on the input element to set the current value of inputRef to the input element. Now when we click the button it will call focusInput which uses the current value of the inputRef variable to set the focus on the input element.

Being able to access any DOM element directly with a ref is really useful for doing things like setting focus or managing other attributes that you cannot directly control in React, but it can be easy to abuse this power. I often see newer React developers using refs to dynamically add and remove elements (appendChild, removeChild, etc.) in a

component instead of having React do that for you. This leads to inconsistencies between the actual DOM and the React virtual DOM which is very bad.

Using Refs Beyond The DOM

While most use cases for refs lie with referencing DOM elements, refs can also be used for any form of storage that is persisted across component renders. A very common use case for this would be storing the previous value of a state variable.

```
function Component() {

  const [name, setName] = useState('Kyle')

  const previousName = useRef(null)

  useEffect(() => {

    previousName.current = name

  }, [name])

  return (

    <>

      <input value={name} onChange={e => setName(e.target.value)} />

      <div>{previousName.current} => {name}</div>

    </>

  )

}
```

The above code will update the previousName ref every time the name changes so that it always has the previous value of the name variable stored in it.

Refs in React are incredibly useful for accessing and manipulating DOM elements directly. Refs are also amazing at persisting data between renders which is makes it possible to store persisted component data without causing a re-render when it is changed.