ReactJS

Elite
IT Academy

What is React?

- React is a JavaScript library for web and native user interfaces
- React lets you build user interfaces out of individual pieces called components.
- Create your own React components like Headers, Footers, Menu’s etc. Then combine them into entire screens, pages, and apps.
- Write components with code and markup
- React components receive data and return what should appear on the screen. You can pass them new data in response to an interaction, like when the user types into an input. React will then update the screen to match the new data.
- React is also an architecture. Frameworks that implement it let you fetch data in asynchronous components that run on the server or even during the build. Read data from a file or a database, and pass it down to your interactive components.

A Web Page with JavaScript

App state	Data definition, organization, and storage
User actions	Event handlers respond to user actions
Templates	Design and render HTML templates
Routing	Resolve URLs
Data fetching	Interact with server(s) through APIs and AJAX

Installing React & Visual Studio Code

- Install NodeJs
  - https://nodejs.org/en/download/
- Create a REACT project using the create-react-app program
  - npx create-react-app first-app
- The application is created, and navigated to the directory using
  - cd first-app
- Start the application using
  - npm start

Creating a React App & Code Walkthrough

Elite
IT Academy

The REACT project

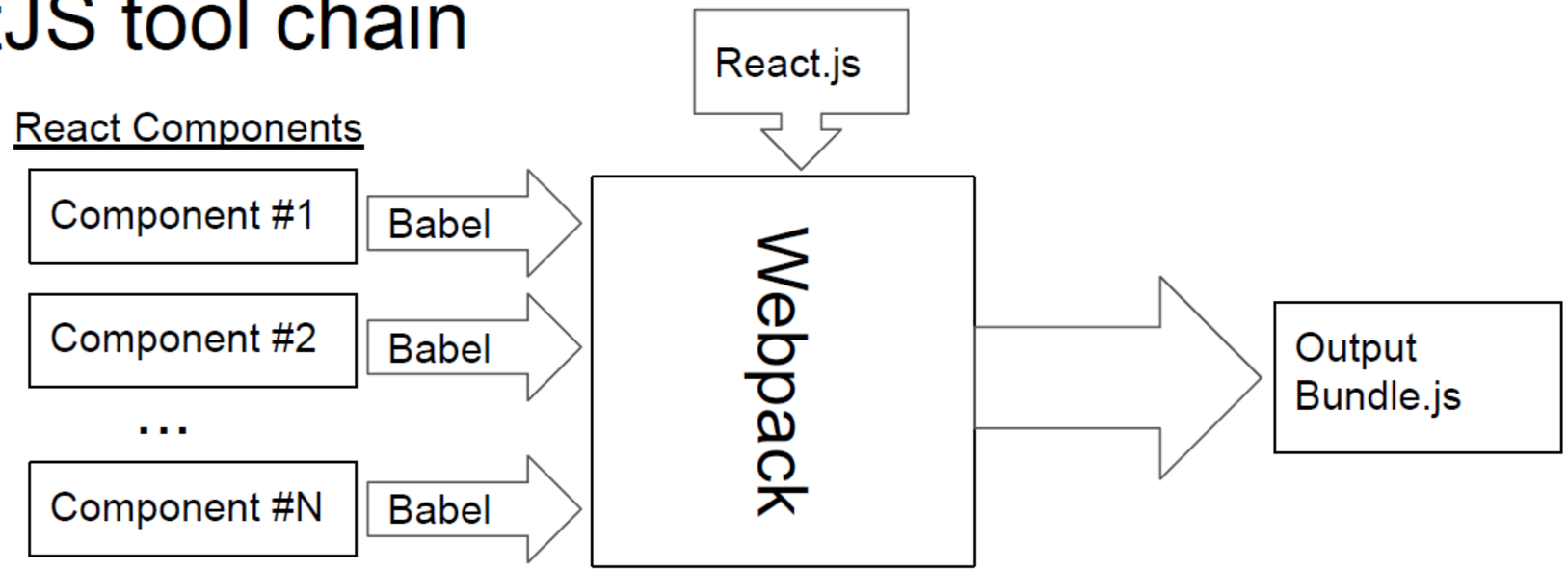
- Project Folders
  - **node_modules** - The node_modules folder is used to save all downloaded packages from NPM on your computer for the REACT project
  - **src** - Contains the source code of the application
    - App.js
    - App.css
    - index.css
    - index.js
  - **package.json -** is used to store the metadata associated with the project as well as to store the list of dependency packages

Webpack and Babel

- Tools like Webpack and Babel ensure that our React apps are efficient, optimized, and compatible across different environments.
- With the rise of frameworks and libraries like React, where applications are composed of many modules, there’s a necessity to manage these modules in an efficient manner.
    - **Module Bundling**
    - **Code Splitting**
    - **Asset Management**
    - **Optimizations**
- Babel is a powerful JavaScript compiler that has become an indispensable tool for developers, especially those building React applications.
- At its core, Babel’s primary function is to transform cutting-edge JavaScript code into versions of JavaScript that can be interpreted by older browsers.

Webpack and Babel

ReactJS tool chain

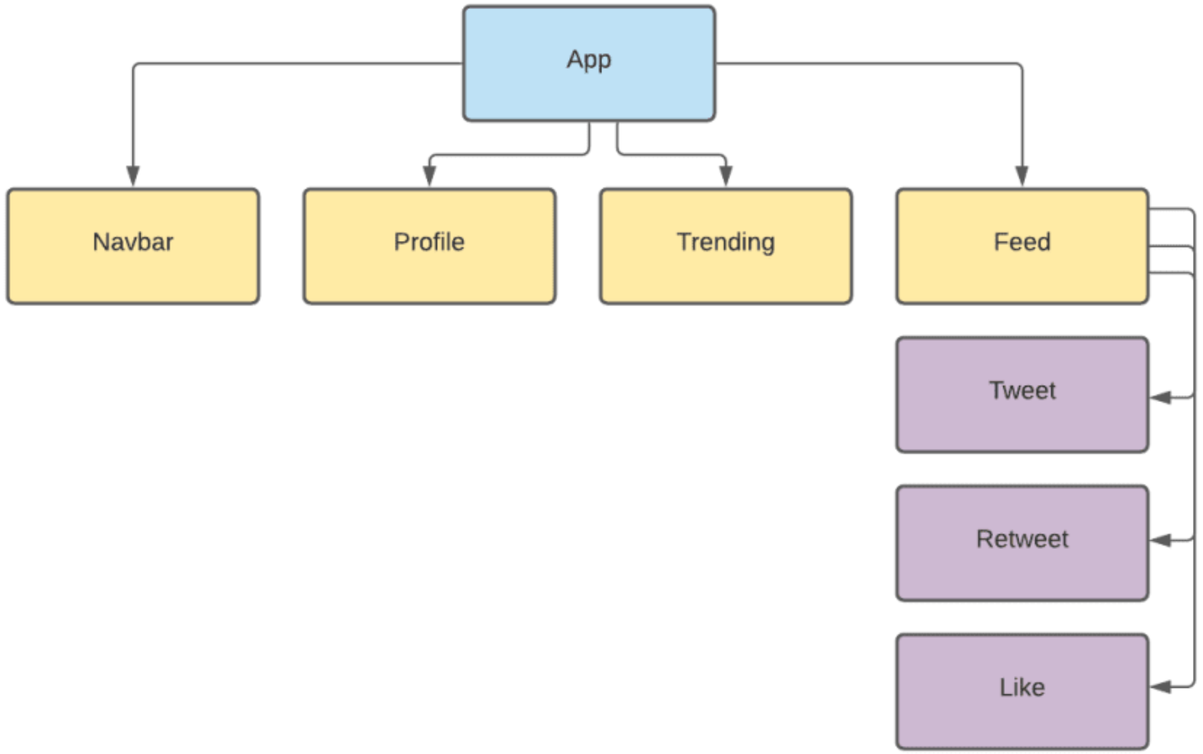


Babel - Transpile language features (e.g. ECMAScript, JSX) to basic JavaScript

Webpack - Bundle modules and resources (CSS, images)

Output loadable with single `script` tag in any browser

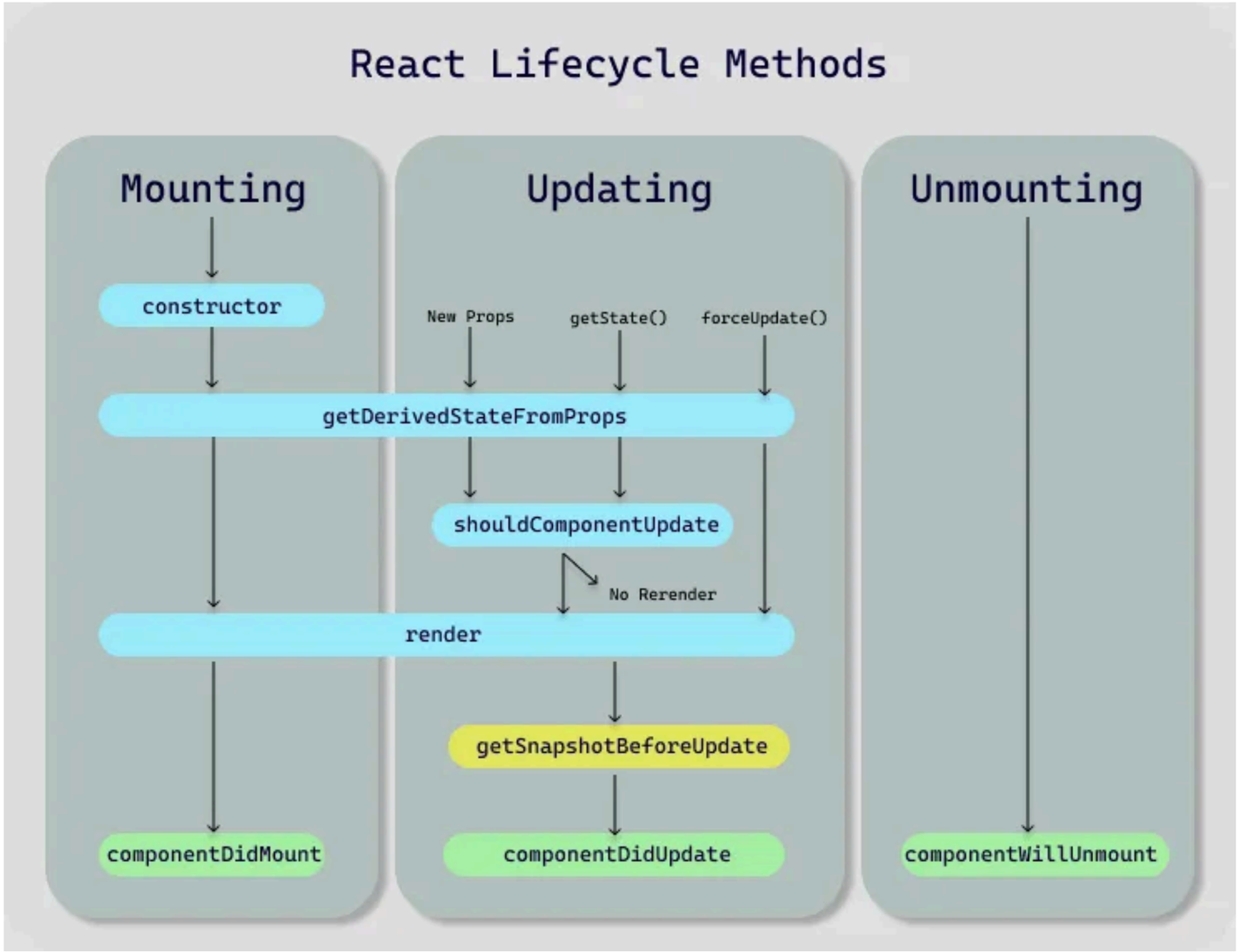
React Components



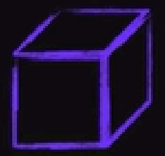
Lifecycle of a Functional Component

There are 3 phases in the React Component LifeCycle:

- Mounting Phase - During the mounting phase, a functional component is created and added to the DOM. In this phase, you typically initialize state and perform any setup that's needed when the component is first rendered.
- Updating Phase - In the updating phase, the functional component is re-rendered due to changes in its props or state.
- Un-Mounting Phase - In the unmounting phase, the functional component is being removed from the DOM.



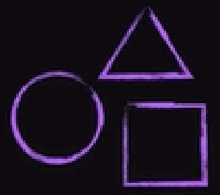
Why Components?



Reusable Building Blocks

Create **small building blocks** & **compose** the UI from them

If needed: **Reuse** a building block in different parts of the UI (e.g., a reusable button)



Related Code Lives Together

Related HTML & JS (and possibly CSS) code is **stored together**

Since JS influences the output, storing JS + HTML together makes sense

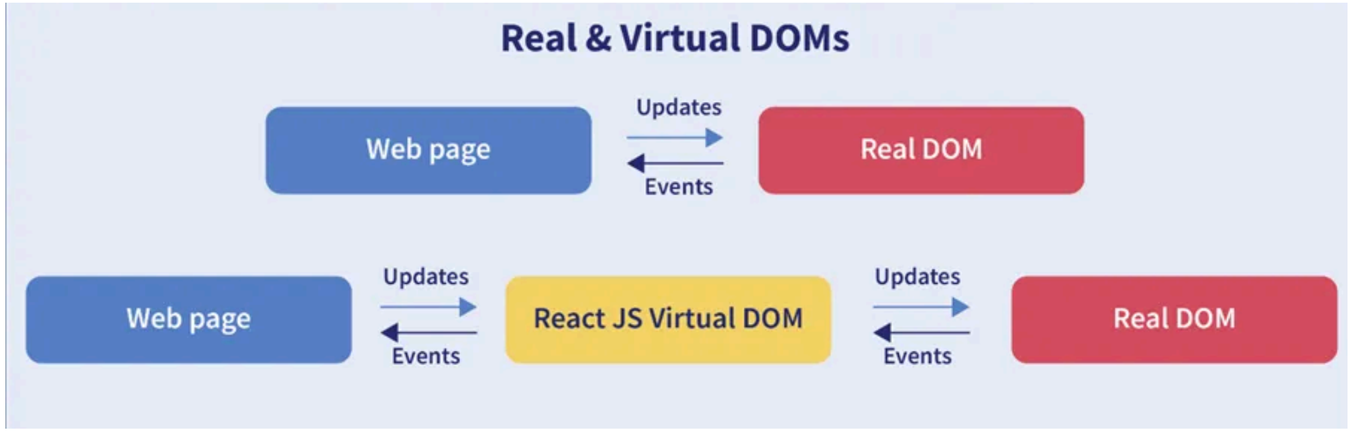
Separation of Concerns

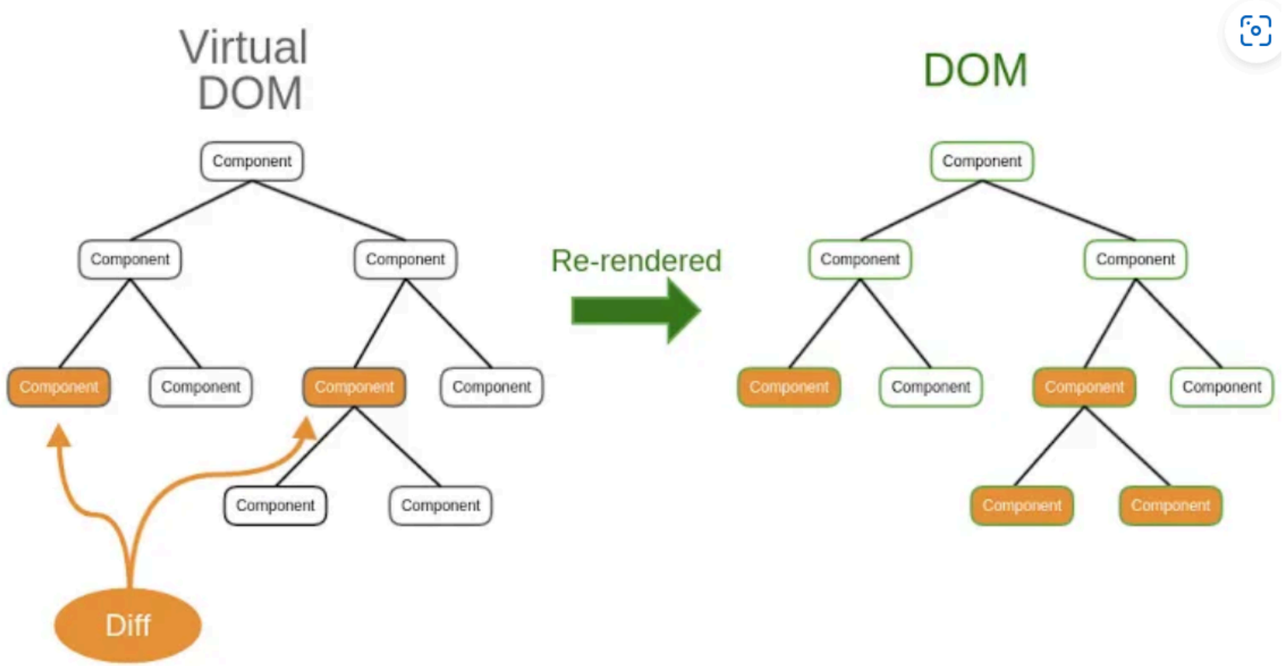
Different components handle different data & logic

Vastly **simplifies** the process of working on complex apps

What is Virtual DOM

- The virtual DOM (VDOM) is a programming concept where an ideal, or “virtual”, representation of a UI is kept in memory and synced with the “real” DOM by a library such as ReactDOM.





Java Script Xtension (JSX)

Elite
IT Academy

JSX - JavaScript Syntax Extension

- XML-like syntax extension of JavaScript
- Transpiles to JavaScript
- Lowercase tags are treated as HTML/SVG tags, uppercase are treated as custom components

JSX - JavaScript Syntax Extension

Elite IT Academy

Without JSX

```
const textHere = React.createElement('h1', {}, 'This is an example without using JSX.');
const container = React.createElement('div','{}', textHere);
ReactDOM.render(container,rootElement);
```

With JSX

```
const container = (
 <div>
  <h1>This is an example without using JSX.</h1>
 </div>
);
ReactDOM.render(container,rootElement);
```

React.createElement - Examples

Elite IT Academy

```
import React from 'react';
import ReactDOM from 'react-dom';
const title = React.createElement('h1', {}, 'My First React Code');
const container = React.createElement('div', {},title);
ReactDOM.render(container,document.getElementById('global'));
```

React.createElement - Examples

```
import React from 'react';
import ReactDOM from 'react-dom';
const list = React.createElement('div', {},
              React.createElement('h1', {}, 'My favorite ice cream flavors'),
              React.createElement('ul', {},
              [React.createElement('li', { className: 'brown' }, 'Chocolate'),
               React.createElement('li', { className: 'white' }, 'Vanilla'),
               React.createElement('li', { className: 'yellow' }, 'Banana')
               ])) ;

ReactDOM.render(list,document.getElementById('global'));
```

React.createElement

Every JSX code is converted as the browser understands the React.createElement function call.

The syntax for React.createElement is as follows:

```
React.createElement (type, [props], [...children])
```

createElement function's parameters.

- type can be an HTML tag like h1 or a React component like div.
- children can include other HTML elements or be a component.
- props are the qualities you wish the element to have.

JSX Nested Elements

You must enclose each element with a container element if you want to use more than one element.

```
import React, { Component } from 'react';
class Application extends Component{
    render(){
        return(
            <div>
                <h1>Coding Ninjas</h1>
                <p>Welcome! Let us know about JSX in depth.</p>
            </div>
        );
    }
}
```

Here, we're using div as a container element for two nested components.

Like HTML, JSX elements can be nested inside other JSX elements.

JSX Attributes

Although the 'class' attribute is commonly used in HTML, you cannot use it in JSX since it is rendered as JavaScript and the 'class' keyword is a reserved word in JavaScript. Instead, use the className attribute.

```
const greetPeople = "Welcome to React";
<h1 className="header">{greetPeople}</h1>
```

When we compile this, we'll get the following

```
<h1 class="header">Welcome to Coding Ninjas</h1>
```

Self Closing Tags in JSX

- When it comes to rendering HTML, browsers are generally chill.
- JSX, on the other hand, needs that all tags to be closed.
- This means that certain components, including <input> and <img>, must be closed with'/'. JSX will throw an error if a tag is not self-closed.

```
/* This code shows error */
<div>
    <br >
    <p>Hello! How was your day?</p>
    <p>Make sure to close all the self-closing tags!</p>
    <br >
</div>
```

```
<div>
    <br/>
    <p>Hello! How was your day?</p>
    <br/>
    <img src={imgLink} />
</div>
```

JSX-Adding Dynamic Content

- We've just used HTML tags as part of JSX, though it becomes more useful when we add JavaScript code to JSX.
- To include JavaScript code in JSX, wrap it in curly brackets as follows

```
const Application = () => {
    const numberTaken = 35;
    return (
        <div>
            <p>Number: {numberTaken}</p>
        </div>
    );
};
```

- We can only write a JSX expression that evaluates to some value inside the curly brackets. As a result, the use of curly brackets is sometimes referred to as JSX Expression Syntax.

JSX-Expression

- In a JSX expression, you can include the following elements:
  - String examples like "Hello World".
  - A number such as 40.
  - [1, 2, 3, 4, 5, 6] is an example of an array.
  - The property of an object that will be evaluated to a value.
  - A function call that returns a value that could be JSX.
  - A map method that returns a new array every time.

- The items listed below are invalid and cannot be used in a JSX expression:
  - A while loop or for loop, or any other loop
  - A declaration of a variable
  - A declaration of a function
  - An object
  - An if condition statement

JSX- Setting Attributes & Event Listeners

The curly braces can also be used to set 'attributes' and 'event listeners' in JSX with JavaScript.

```
const altImageDescription = "Let us look at a picture of a dog.";
function randomEvent(e){}
const dogImage = (
  <img
      src="images/dog.jpg"
      alt={altImageDescription}
      onClick={randomEvent}
  />
);
```

- Like HTML elements, JSX elements can have event listeners.
- The name of the event listener attribute should be something like onClick or onMouseOver: the word 'on' followed by the sort of event you're monitoring.
- The value of an event listener attribute should be a function.

JSX- Conditionals

you can use if statements outside of your JSX to determine which components should be used

```
var loginButton;

if (loggedIn) {
   loginButton = <LogoutButton />;
} else {
   loginButton = <LoginButton />;
}

return (<nav><Home />{loginButton}</nav>);
```

JSX- Conditionals

Using a ternary operator

```
const printDog (
 <p>
  {random() === 'dog' ? 'print dog' : 'print cat'}
 </p>
);
```

The && operator is identical to the ? ternary operator, however, it does not include the second fallback condition. Basically, if the situation is true, take some action. Don't do anything if the condition doesn't apply.

```
const printDog (
 <p>
  {random() === 'dog' && 'print dog' }
 </p>
);
```

JSX- Key Takeaways

- Every JSX tag is changed into a React component called React.createElement.
- Only items that evaluate to some value, such as string, number, array map method, and so on, are allowed in JSX Expressions, which are written inside curly brackets.
- When adding classes to an HTML element in React, we must use className instead of class.
- In React, all attribute names are written in camelCase.
- When used inside JSX, undefined, null, and boolean are not displayed on the UI.

How REACT Works

```
function App() {
 return (
 <div>
 <p> Hello World <p>
 </div>
 );
}
```

```
const para = document.createElement('p');
para.textContent = 'Hello World';
document.getElementById('root').append(para);
```

Creating a new component

```
function SubTitle() {
 const name = "My Name";
 return (
 <h2> Hello {name} </h2>
 );
}

export default SubTitle;
```

```
import SubTitle from './components/SubTitle'

function App() {
 return (
 <div>
 <p> Hello World <p>
 <SubTitle />
 </div>
 );
}

export default App
```

Adding Styles

```
.subtitle {
 display: flex;
 justify-content: space-between;
 align-items: center;
 box-shadow: 0 2px 8px rgba(0, 0, 0, 0.25);
 padding: 0.5rem;
 margin: 1rem 0;
 border-radius: 12px;
 background-color: #4b4b4b;
}
```

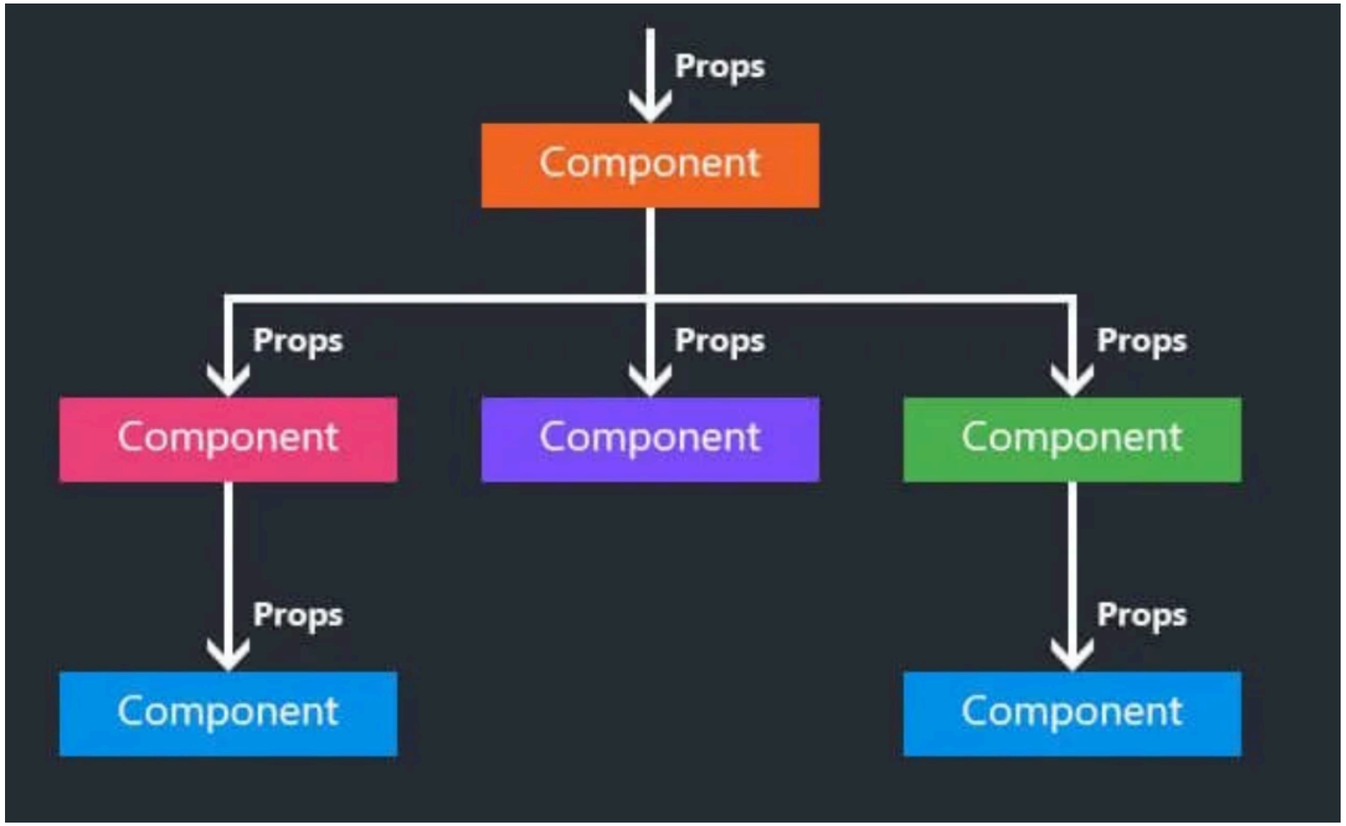
```
import "./SubTitle.css";

function SubTitle() {
 const name = "My Name";
 return (
  <div className="subtitle">
   <h2> Hello {name} </h2>
   <p className="subtitle-para">
      This is a paragraph of data in containing a long line
   </p>
  </div>
 );
}

export default SubTitle;
```

Props, State & Events

Elite
IT Academy

Props in ReactJS

- Passed as an object to a component and used to compute the returned node
- Changes in these props will cause a recomputation of the returned node (“render”)
- Unlike in HTML, these can be any JS value



Passing Parameters to components

```
function App() {
 return (
  <div>
   <h1> Hello World </h1>
   <SubTitle title="Hello Mayank" para="This is para for Mayank" />
   <SubTitle title="Hello Nitin" para="This is para for Nitin" />
  </div>
 );
}

function SubTitle(props) {
 return (
  <div className="subtitle">
   <h2> {props.title} </h2>
   <p className="subtitle-para">{props.para}</p>
  </div>
 );
}
```

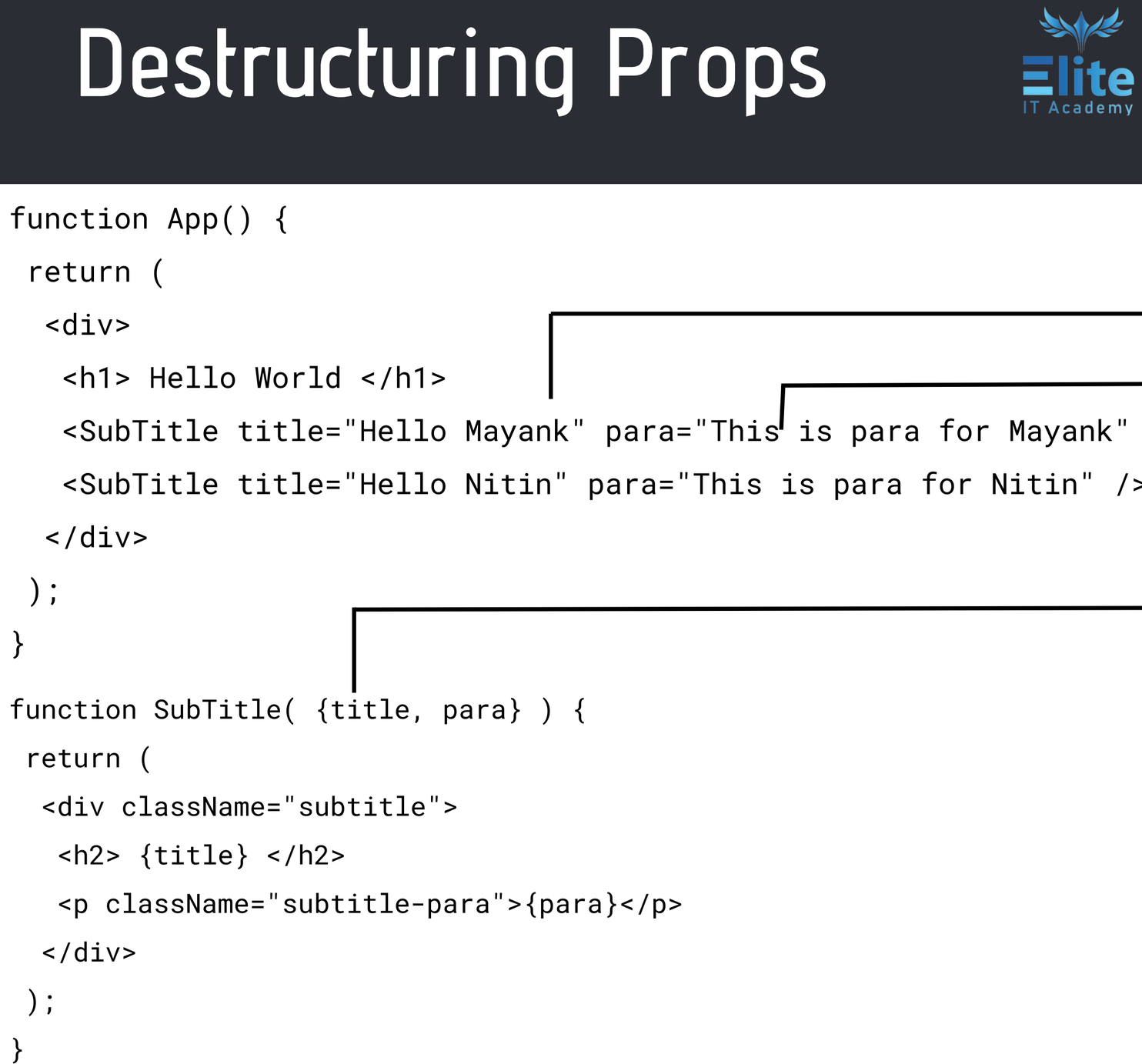
{

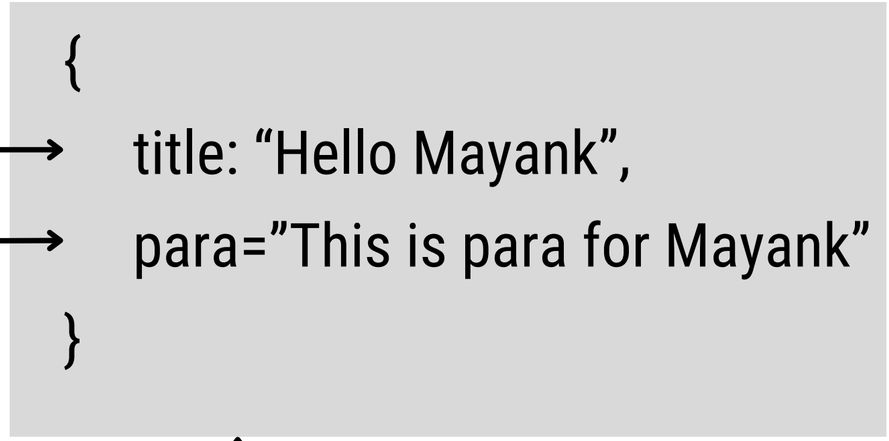
title: “Hello Mayank”,

para=”This is para for Mayank”

}

Destructuring Props





Splitting Components

SubTile.js

```
import SubTitleName from "./SubTitleName";
import SubTitlePara from "./SubTitlePara";

function SubTitle(props) {
 return (
  <div className="subtitle">
   <SubTitleName title={props.title} />
   <SubTitlePara para={props.para} />
  </div>
 );
}
```

SubTitleName.js

```
function SubTitleName(props) {
 return <h2> {props.title} </h2>;
}

 export default SubTitleName;
```

SubTitlePara.js

```
import "./SubTitle.css";

export default function SubTitlePara(props) {
 return <p className="subtitle-para">{props.para}</p>;
}
```

Creating an Event Handler

SubTitleName.js

```
function SubTitleName(props) {
 const clickHandler = () => {
  console.log("Clicked!!");
 };

 return <h2 onClick={clickHandler}>
           {props.title}
        </h2>;
}

export default SubTitleName;
```

Event handler

Registering an Event Hander

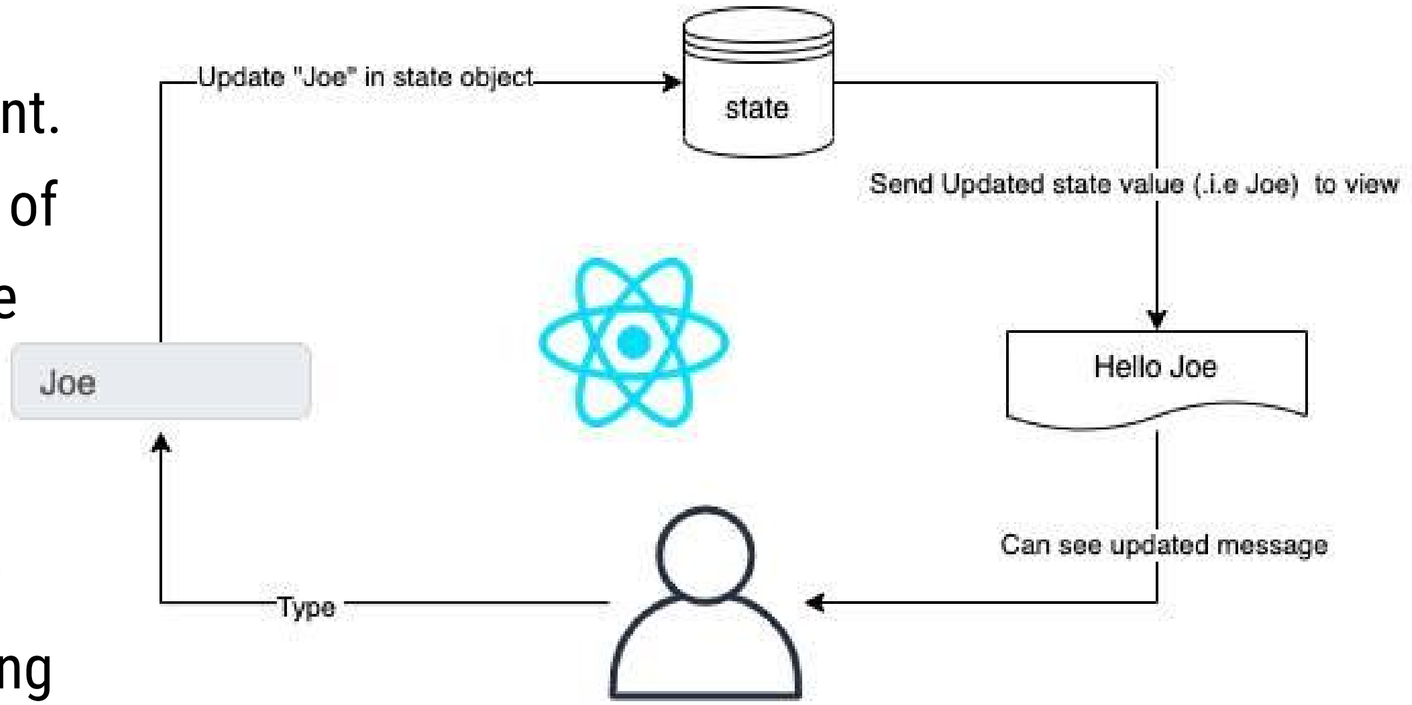
```
onClick={clickHandler}
onClick={ ()=> {clickHandler()}
onClick={console.log(“clicked”)}
```

React Event Handling

- Event Binding: To handle events in React, you start by defining event handlers as methods within your component. These methods are typically named according to the event they handle, such as `handleClick` for a click event.
- JSX Event Syntax: In your JSX code, attach event handlers to elements using a specific syntax. For example, to handle a button click, you would add `onClick={this.handleClick}` to the button element.
- Event Handling Method: Inside your event-handling method, you can access event properties such as `event.target.value` for input fields. You can also use the `setState` method to update the component's state based on the event.
- Prevent Default Behavior: When necessary, prevent the default behavior of certain events using `event.preventDefault()`.
- Event Delegation: React's virtual DOM allows for efficient event delegation, where a single event listener can handle multiple elements, improving performance.
- Asynchronous Event Handling: Keep in mind that React event handlers are asynchronous. If you need to access the updated state immediately after an event, use the callback function of `setState`.

What is Component State?

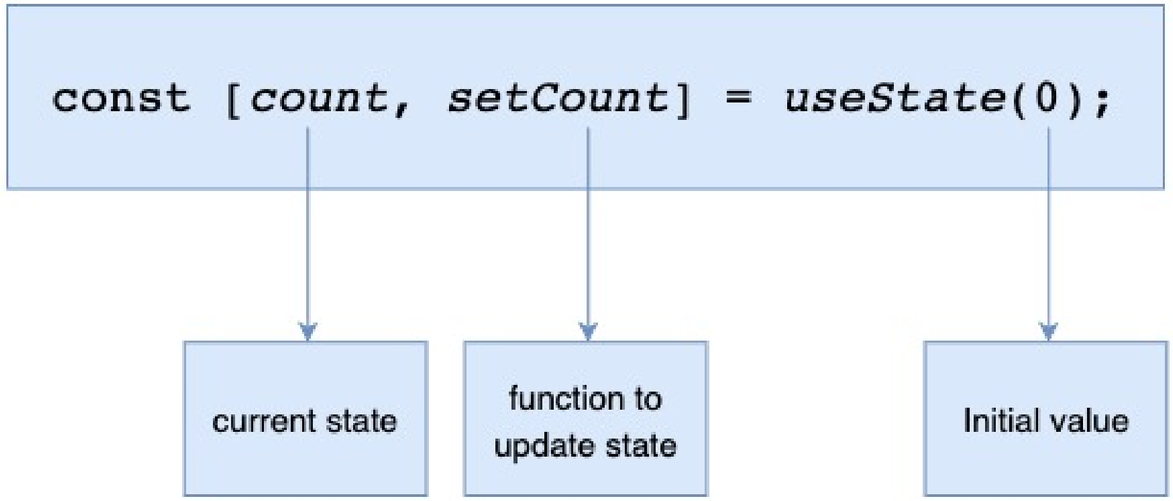
- In React, state is a JavaScript object that holds data that can be used to influence the rendering of a component.
- In simple terms, consider it any piece of information that can change over time and impact how your component appears or behaves.
- The state allows developers to create dynamic and interactive UIs by enabling components to respond to user input, API responses, or other events.



The useState hook

SubTitleName.js

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

function SubTitleName(props) {
  const [title, setTitle] = useState(props.title);

   const clickHandler = () => {
     setTitle("Title Changed");
      console.log(title);
  };

  return <h2 onClick={clickHandler}> {title} </h2>;
}

export default SubTitleName;
```



Using a single state & previous state

InputForm.js

```
const [formData, setFormData] = useState({
 title: "",
 para: "",
});

const titleChangedHandler = (event) => {
 setFormData({
  ...formData,
  title: event.target.value,
 });
 console.log(event.target.value);
};
```

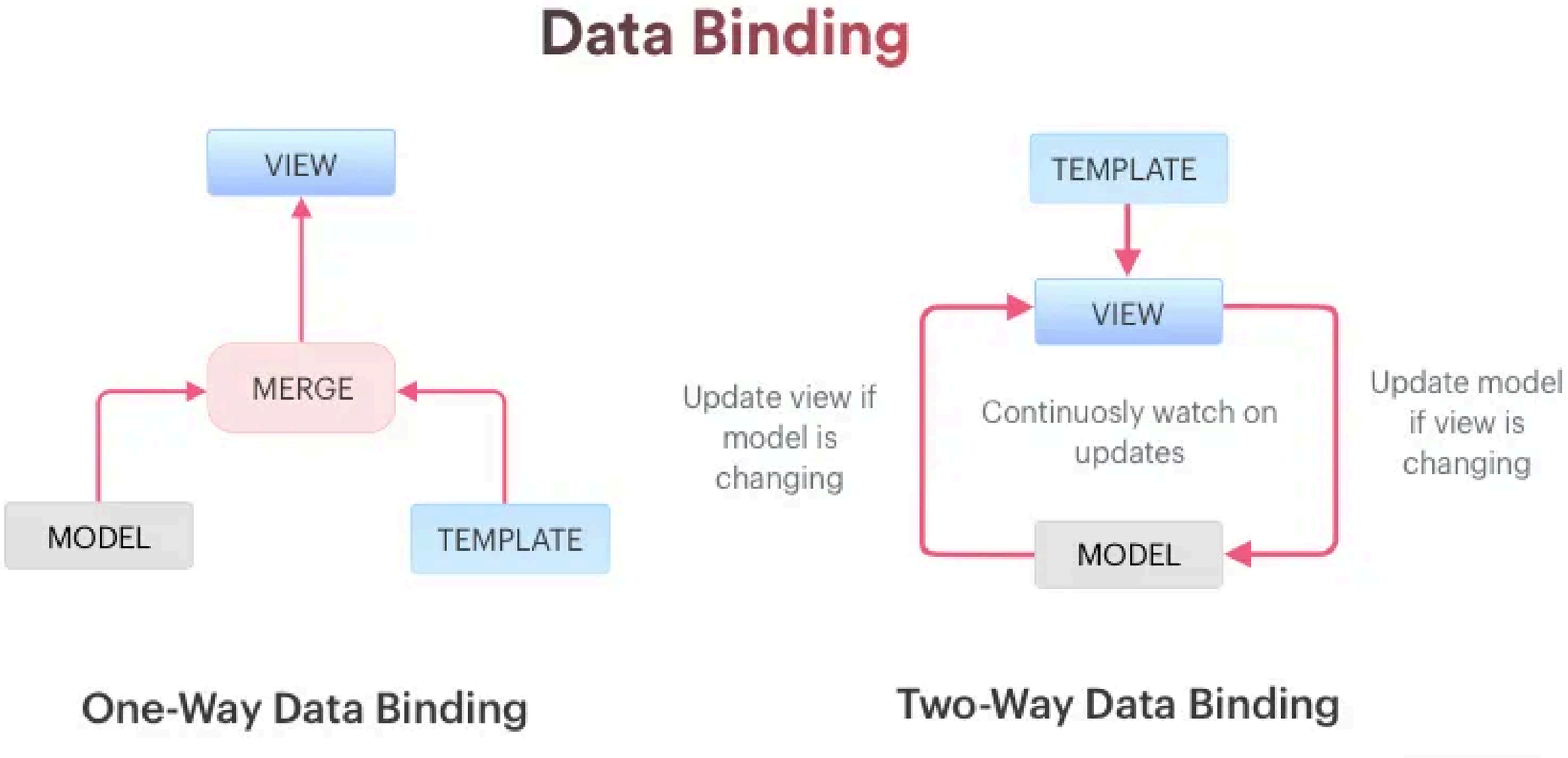
How is state updated?

```
const paraChangedHandler = (event) => {
 setFormData((prevState) => {
  return {
   ...prevState,
   para: event.target.value,
  };
 });
 console.log(event.target.value);
}
```

State vs Props

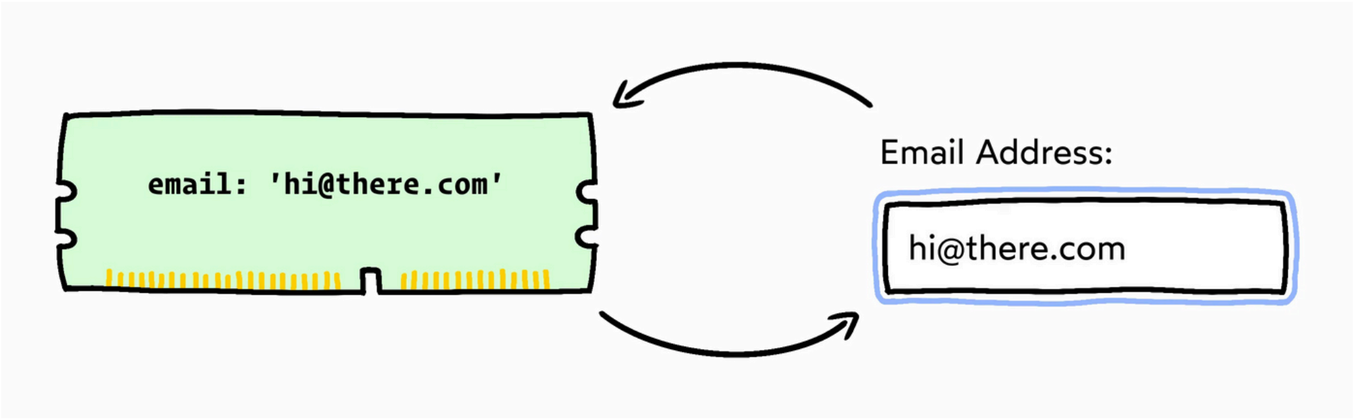
State	Props
State is managed within the component	Props gets passed to the component
State can be changed(mutable)	Props are read only and cannot be changed (immutable)
State can be accessed using the use state hooks in functional components and in-class components can be accessed using this. State	Props can be accessed in functional component using props parameter and in-class component, props can the accessed using this.props
State changes can be asynchronous	Props are read only
State is controlled by react components	Props are controlled by whoever renders the components
State can used to display changes with the component	Props are used to communicate between components

Two way data binding



Two Way data Binding

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

const ControlledComponentExample = () => {
 const [email, setEmail] = useState('');

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

 return (
 <div>
 <input type="email" value={email} onChange={handleInputChange} />
 <p>Email Address: {email}</p>
 </div>
 );
};
```



Child to Parent Communication

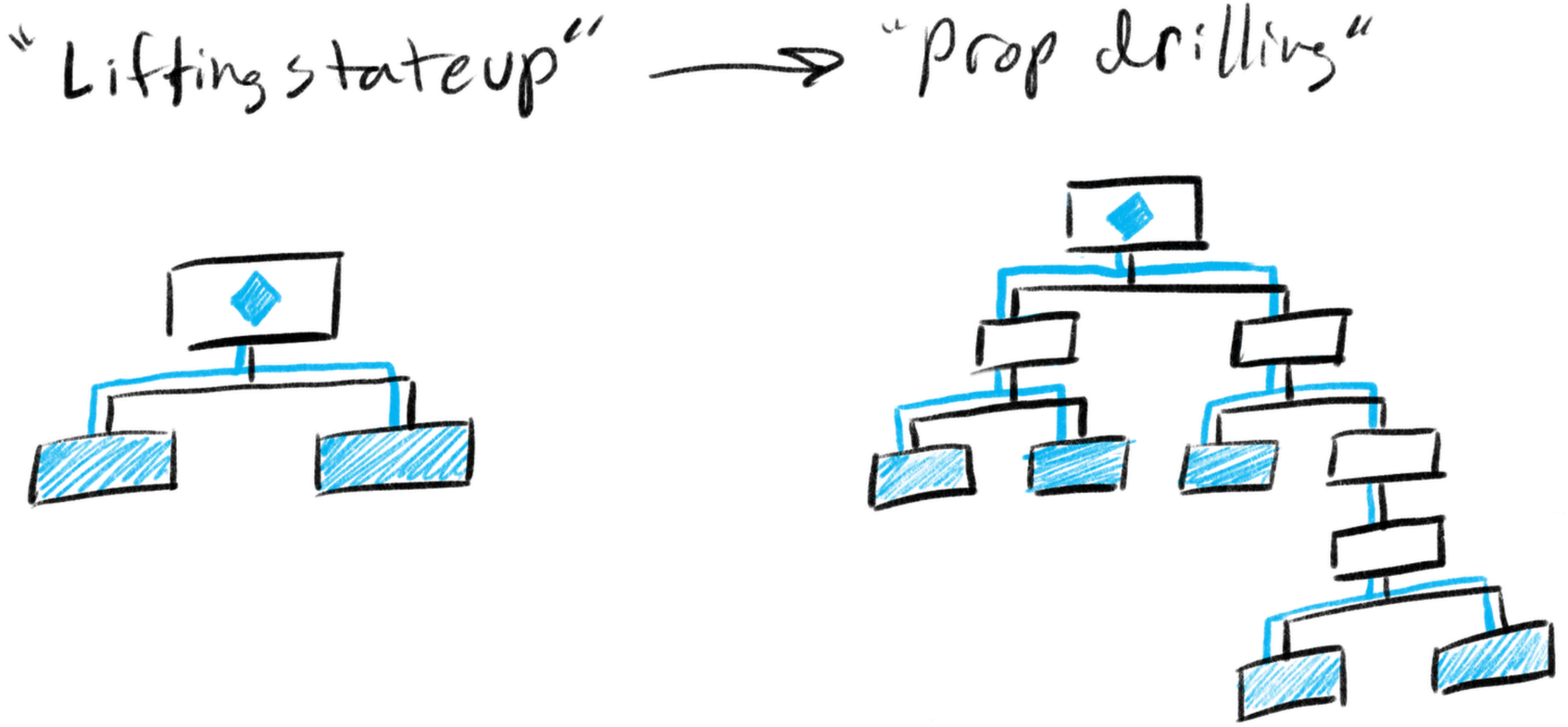
App.js

```
// add handler to call in child
const onChangedForm = (newData) => {
 console.log(newData);
};

// Pass the handler to the child as a prop
 <InputForm onChangedData={onChangedForm} />
```

InputForm.js

```
// In child call the handler defined in the props.

const submitHandler = (event) => {
  event.preventDefault();
  let enteredData = {
   title: title,
   para: para,
  };
  props.onChangedData(enteredData);
  setTitle("");
  setPara("");
 };
```

Prop Drilling vs Lifting State up



Styling React Components

Elite
IT Academy

Styling in React

- Styling in React refers to applying visual design and layout to components and elements within a React application.
- It involves using CSS or other styling libraries to customize the appearance of these components, making them visually appealing and consistent with the application’s overall design.
- You can style components using different options
  - **Inline Styling**
  - **CSS frameworks**
  - **Component-based Styling**

Inline Styling

- Every JSX element has a style property you can add to its opening tag. This means you can add inline styling to JSX in a React component like in traditional HTML.
- The primary difference is that you must specify inline styles as objects. In this object, the keys are the CSS properties written in camelCase, and the values are strings corresponding to valid CSS values.
- And because the stylings have to be an object, you have to add them inside two curly braces if you're passing them directly to the element:
    - **<h1 style={{ textAlign: 'center', marginTop: '2rem', color**
- You can define the same styles as a separate object and pass it into the style property

```
export default function Home() {
   const myStyles = {
      textAlign: 'center',
      marginTop: '2rem',
      color: '#F43596',
    };
   return (
      <main>
        <h1 style={myStyles}>
            Hello
        </h1>
      </main>);
}
```

Conditional Styling

- to handle conditional CSS with inline styling in React, you can use a combination of the useState hook and ternary operator

```
import { useState } from 'react';
const ConditionalInlineStyling = () => {
     const [isActive, setIsActive] = useState(false);
     const buttonStyle = {
           margin: '0 auto',
           backgroundColor:
           isActive ? 'green' :
          'gray',cursor: isActive ? 'pointer' :
          'not-allowed',color:
          'white',padding: '0.6rem 1.2rem',
          border: 'none',
          };
```

```
return (
    <div style={
       { textAlign: 'center',
         marginTop: '2rem'
       }}>
        <button style={buttonStyle}>
           {isActive ? 'Active' :
                       'Inactive'}
        </button>
     </div>);
};

export default ConditionalInlineStyling;
```

CSS Stylesheets

- import the stylesheet using the import keyword and then specify the relative path of the stylesheet,

```
import { useState } from 'react';
import './styles/styles.css'; // Import the CSS file
const ConditionalStyledComponent = () => {
    const [isActive, setIsActive] = useState(false);
    return (
       <div className="container">
            <button className={`button ${isActive ? 'button-active' : 'button-inactive'}`}>
                  {isActive ? 'Active' : 'Inactive'}
            </button>
       </div>
      );
    };

export default ConditionalStyledComponent;
```

Using CSS Modules

- CSS Modules offer a powerful solution for writing component-specific styles in React.
- They let you scope styles to individual components, thereby letting you avoid naming conflicts and simplifying style maintenance.
- To use CSS modules in React, create a file with the .module.css extension. For example, ***styles.module.css***.
- You then need to import the file into your component with the import keyword and a name you want, for example, styles or classes, followed by the relative path of the CSS modules file
    - ***import styles from 'relative-path-to-css-modules-file.module.css'***;
- The name you choose is now an object, the keys are the classes in the CSS modules file and the values are the respective properties in the class

Using CSS Modules

```
import { useState } from 'react';
import styles from './styles/styles.module.css'; // Import the CSS modules file
const ConditionalStyledComponent = () => {
    const [isActive, setIsActive] = useState(false);
    return (
       <div className={styles.container}>
            <button className={`${styles.button} ${isActive ? styles.buttonActive :
                                styles.buttonInactive }`}>
                {isActive ? 'Active' : 'Inactive'}
            </button>
       </div>
      );
};

export default ConditionalStyledComponent;
```

Using CSS-in-JS with StyledComponents

- CSS-in-JS is a styling approach in which CSS is written in JavaScript, allowing you to style your components using JavaScript syntax.
- This method offers a more dynamic and modular way to manage styles, making it easier to handle themes, scoped styles, and both pseudo-classes and pseudo-elements.
- styled-components is the most Popular CSS-in-JS library. Others are Emotion and styled-jss.
- To use styled-components, you first need to add it to your project by installing it with NPM or any other package manager
    - npm install styled-components
- You then need to import styled from styled-components:
    - import styled from 'styled-components'
- To define styles for an element or a whole component, you have to define a component and assign it to styled.element-name, then define the styles you want inside the backticks.

Using CSS-in-JS with StyledComponents

Elite IT Academy

```
import { useState } from 'react';
import styled from 'styled-components';
const Button = styled.button`
 background-color: ${(props) =>
      (props.isActive ? 'green' :
          'gray')};
 color: white;
 padding: 0.6rem 1.2rem;
 border: none;
 cursor: ${(props) => (props.isActive
          ? 'pointer' : not-allowed')};
 transition: background-color 0.3s;
`;
const Container = styled.div`
 display: flex;
 justify-content: center;
 align-items: center; `;
```

```
const ConditionalStyledComponent = () =>
{
   const [isActive, setIsActive] = useState(false)
   return (
     <Container>
         <Button isActive={isActive}>{isActive ?
              'Active' : 'Inactive'}
         </Button>
     </Container>);
    };

export default ConditionalStyledComponent;
```

Rendering Components in React

Elite
IT Academy

Conditional Rendering

- Better User Experience: It'll help you create dynamic interfaces that adapt to user actions and data changes.
- Enhanced Performance: To prevent performance issues, you must avoid rendering unnecessary components in larger applications.
- Streamlined Code: Using conditional statements can enhance code readability and simplicity. You can decide what content should be rendered and avoided.
- Flexibility: It'll build flexible and customizable components by rendering different content based on the application state. In this, components can also adapt to various contexts and user interactions.

Display a Component Conditionally

```
import { useState } from 'react'
import ModalWrapper from 'ui/ModalWrapper'
import QuizScreen from 'screens/QuizScreen'

const App = () => {
  const [showResultModal, setShowResultModal] = useState<boolean>
(false)

  return (
    <div>
      <QuizScreen />
      {showResultModal && <ModalWrapper title="SHOW RESULT" />}
    </div>
  )
}
export default App
```

Using a short-circuiting AND operation

Toggle between two components

```
const Dashboard = () => <p>Dashboard Screen</p>
const Login = () => <p>Login Screen</p>

const App = () => {
  const [isLoggedIn, setIsLoggedIn] = useState < boolean > false

  return <div>{isLoggedIn ? <Dashboard /> : <Login />}</div>
}

export default App
```

Using ternary operator to display components conditionally

Conditionally rendering multiple components

```
const { currentScreen, setCurrentScreen }
                              = useState(ScreenTypes.SplashScreen)
  switch (currentScreen) {
    case ScreenTypes.SplashScreen:
      return <SplashScreen />
    case ScreenTypes.QuizTopicsScreen:
      return <QuizTopicsScreen />
    case ScreenTypes.QuizDetailsScreen:
      return <QuizDetailsScreen />
    case ScreenTypes.QuestionScreen:
      return <QuestionScreen />
    case ScreenTypes.ResultScreen:
      return <ResultScreen />
    default:
      return <SplashScreen />
  }
```

Displaying a list of items

Elite IT Academy

The "React Way" to Render a List

- Use `Array.map`
- Not a `for` loop
- Give each item a unique key
- Avoid using array index as the key

Displaying a list

```
function IdiomaticReactList(props) {
    return (
        <div> {
            props.items.map((item, index) => ( <Item key={index} item={item} /> ))}
        </div> );
}
```

- React relies on the key to identify items in the list.
- The first time the component is rendered, React will create DOM nodes for them.
- The next time that component renders, React will avoid recreating DOM nodes if it can tell that the items are the same.
- React can look at the key and know that though this <Item> is not strictly === to the old <Item>, it is the same because the keys are the same.
- This leads to a couple rules for keys. They must be:
  - Unique – Every item in the list must have a unique key.
  - Permanent – An item's key must not change between re-renders, unless that item is different.

Rendering Components in React

Elite
IT Academy

Component Composition

Elite
IT Academy

What is Component Composition?

- React application all about components. They're the building blocks of any React application.
- The idea is to break down complex problems into smaller, manageable components.
- Each component in React has a specific task.
    - It encapsulates the logic required for this task, and it may contain its own state or props.
    - The component's code is responsible for rendering some parts of the UI.
- When we talk about composing components, we're essentially talking about taking these independent components and combining them to form a new component.

What is Component Composition?

```
// Define a Header Component
function Header() {
  return (
    <div> Header </div>
  );
}

// Define a Footer Component
    function Footer() {
      return (
        <div> Footer </div>
      );
    }
```

```
// Define a Container Component
    function App() {
      return (
        <div className="App">
          <Header />
          <Footer />
        </div>
      );
}
```

- **The App Component is composed of the Header Component and the Footer Component.**
- **Each of these components is responsible for rendering a specific part of the UI.**
- **The App Component composes these components to form a new component that represents the entire app.**

Why Component Composition?

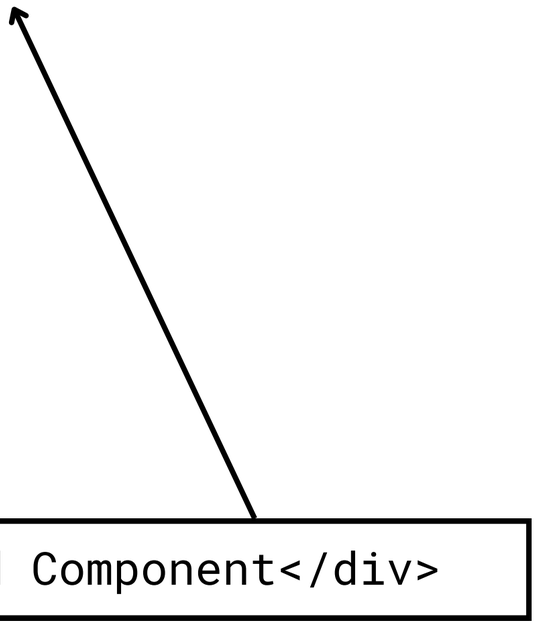
- **Code Reusability and Maintainability**: Composing components promotes code reusability. We can create a component once and reuse it in different parts of our application. This leads to less code, which is easier to maintain and test.
- **Separation of Concerns:** Each component in React is responsible for a single piece of functionality. This separation of concerns makes our code more understandable and easier to debug.
- **Performance:** React can optimize rendering when we compose components. React only re-renders components when their state or props change. By breaking our app into smaller components, we can minimize unnecessary renders and improve performance.
- **Flexibility:** Composing components gives us a lot of flexibility. We can easily swap out components or change their order without affecting the overall functioning of our app.

Props and Children

- Props
  - In React, props are the parameters we pass from a parent component to a child component.
  - They allow us to pass data and event handlers down the component tree.
  -
- Children
  - When composing components, we often pass components as props.
  - The children prop is a special prop in React that allows us to pass components as data to other components.
  - It's like a placeholder for where the child components go.

Props and Children

```
function Container({ children }) {
      return (
        <div className="container">
          {children}
        </div>
      );
    }
function App() {
      return (
        <Container>
          <div>Child Component</div>
        </Container>
      );
    }
```



- The Container Component receives a children prop from the App Component.
- This children prop is a div element, but it could be any valid React element, including custom components.

Higher Order Components

- Higher-order components allow developers to reuse code logic in their projects.
- As a result, this means less repetition and more optimized, readable code.
- HOCs in React offer a versatile way to enhance the functionality and behavior of components.
- React’s Higher Order Component is a pattern that stems from React’s nature that privileges composition over inheritance.

- ***A higher-order component is a function that takes in a component and returns a new component.***

Higher Order Components

Simple Text Component

```
const TextComponent = (props) =>
    <p>{props.text}</p>;
```

Higher Order Component

```
const withFancyText = (WrappedComponent) => {
 return (props) => {
 const fancyStyle =
   { fontStyle: "italic", fontWeight: "bold"
};
 return (
 <div style={fancyStyle}>
 <WrappedComponent {...props} />
 </div>
 );}; };
```

```
import TextComponent from “TextComponent”
import withFancyText from “withFancyText”

const FancyText = withFancyText(TextComponent)

const App = () => {
    return ( {
      <div> <FancyText /> </div>
      })

}
```

Using HOC's in Applications

- Authentication
  - In order to authenticate users before accessing a route, , you can create an HOC called withAuth that checks if the user is authenticated and redirects them to the login page if not. Then, you can wrap the specific components or routes that need authentication with this HOC, reducing duplication and enforcing consistent authentication behavior.
- Logging
  - By wrapping the relevant components with withLogger, you can achieve consistent logging across those components.
- Styling and Theming
  - You might have a design system with reusable styles and themes. You can create an HOC named withTheme that provides the necessary theme-related props to a component.
  - This way, the wrapped component can easily access and apply the appropriate styles based on the provided theme

Forms in React

Elite
IT Academy

Forms in React

- Forms play an essential role in modern web applications. They enable users to share information, complete tasks and provide feedback.
- React provides several features and techniques for creating and managing forms, including state management, event handling, and form validation.

Controlled and Uncontrolled components

- Controlled Components: In this approach, form data is handled by React through the use of hooks such as the useState hook.
- Uncontrolled Components: Form data is handled by the Document Object Model (DOM) rather than by React. The DOM maintains the state of form data and updates it based on user input.

Controlled Components

- In React, a controlled component is a component where form elements derive their value from a React state.
- When a component is controlled, the value of form elements is stored in a state, and any changes made to the value are immediately reflected in the state.
- To create a controlled component, you need to use the value prop to set the value of form elements and the onChange event to handle changes made to the value.
- The value prop sets the initial value of a form element, while the onChange event is triggered whenever the value of a form element changes. Inside the onChange event, you need to update the state with the new value using a state update function.

```
import {useState} from 'react';
export default function  ControlledComponent()  {
   const  [inputValue, setInputValue] =  useState('');
   const  handleChange = (event) => {
          setInputValue(event.target.value);
   };

   return  (
      <form>
          <label>Input Value:
            <input  type="text"  value={inputValue}
                 onChange={handleChange} />
          </label>
          <p>Input Value: {inputValue}</p>
      /form>)
};
```

Form Validation

- Form validation can take various forms, depending on the type and complexity of the data being collected. Common types of form validation include:
  - Required field validation: Checking that required fields are not left empty.
  - Format validation: Ensuring that input data is in the correct format (for example, email addresses, phone numbers, and so on).
  - Length validation: Checking that input data is within a certain length range.
  - Pattern validation: Checking that input data matches a specific pattern.
- Common methods for form validation include using built-in HTML validation attributes like required, minlength, and maxlength, as well as using React to perform custom validation logic.

Uncontrolled Components

- Uncontrolled components in React refer to form elements whose state is not managed by React. Instead, their state is handled by the browser's DOM.
- An uncontrolled component allows the browser to handle the form elements' state. When a user enters text into a text input field or selects an option from a select box, the browser updates the DOM's state for that element automatically.
- To get the value of an uncontrolled form element, you can use a feature called "ref". "Refs" provide a way to access the current value of DOM elements.

useRef

- Ref's help you get access to DOM elements in your React program
- Set's up a connection between the HTML Element and Javascript code
- import {useRef} from 'react'
- within the component
  - const nameInputRef = useRef();
  - const addrInputRef = useRef(); etc.
- In the Input tag for the HTML element add a ref prop
  - ref={nameInputRef}
- In SubmitHandler use the nameInputRef to access the current object value
  - nameInputRef.current.value

useRef Hook

- useRef creates a mutable JavaScript object that is useful and unique because it keeps the same reference to an object through multiple renders.
- The way to use useRef is to mutate its { current: ... } property.
- Refs have been a concept in React for a while, but their main usage has been to refer to a specific DOM node in order to make mutations to that DOM node with any available functions on that node.
- This is still basically the case when using the useRef hook, as it is possible to mutate DOM nodes, but useRef has even more capabilities because it can also reference any value that needs to be accessed through any re-renders for the lifetime of the component.
- This makes it similar to the instance values that are commonly used in classes.

```
function TextInputWithBlurButton() {
        const inputRef = useRef(null);
        const blurOnButtonClick = () => {
        // `current` points to the mounted text input
            inputRef.current.blur();
        };
        return (<>
            <input ref={inputRef} type="text" />
            <button onClick=
                    {blurOnButtonClick}>
                    Click to blur input</button>
            </>);
}
```

JavaScript Form Data

- JavaScript FormData is an interface that allows you to construct and manipulate form data in a way that closely mimics how HTML forms work.
- It enables you to easily gather data from various input elements, such as text fields, checkboxes, and file inputs.
- FormData organizes this data into key-value pairs, making it perfect for sending structured data to your backend server.
- The FormData object doesn't include data from the fields that are disabled or the fieldsets that are disabled.

JavaScript Form Data

- The formdata object lets you compile a set of key/value pairs to send using XMLHttpRequest.
- In a simply term formdata provide an alternative of bundling data(object),and sending it to the server without using basic html form tag.
- Think of it as a way to replicate what a html form does.

Appending data to a form object without using html form tag

```
const appendFunction=()=>{
    let myFormData = new FormData()
    myFormData.append('city','Lagos');
    myFormData.append('address','no 33 Golden Street');
    //formdata is an array of array we can iterate it..
    for(let obj of myFormData){
        console.log(obj)
    }
}
```

JavaScript Form Data

- When sending data to the server using formdata
- Do ensure that(Content-Type: multipart/form-data)on the request header is set,this to enable the server to know which multi media types and handle it properly(blobs,.csv, etc).

Appending large data to the server using html form tag.

```
const cityAddressForm = () => {
    const [citiesAddress, setCitiesAddress] = useState(null)
    const form = useRef(null)

    const submit = e => {
      e.preventDefault()
      const data = new FormData(form.current)
      //please take note of the Content-Type: multipart/form-data send along in the request header.
      fetch('/api', { method: 'POST', body: data ,'Content-Type': 'multipart/form-data'})
        .then(res => res.json())
        .then(json => setCitiesAddress(json.citiesAddress))
    }

    return (
      <form ref={form} onSubmit={submit}>
        <input type="file" name="citiesAddress"  multiple />

        <input type="submit" name="Submit" />
      </form>
    )
  }
```

React Hook Form

- The number of re-renders in the application is smaller compared to the alternatives because it uses refs instead of state.
- The amount of code you have to write is less as compared to formik, react-final-form and other alternatives.
- react-hook-form integrates well with the yup library for schema validation so you can combine your own validation schemas.
- Mounting time is shorter compared to other alternatives.

React Hook Form

- React Hook Form is a lightweight library for managing forms in React applications. Whether you need to create a simple contact form or a complex multi-step form, React Hook Form can help simplify your form creation process.
- you'll need to install the library in your project. You can do this using npm by running the following command
  - npm install react-hook-form
- you need to import the useForm hook from the react-hook-form package in your component.
  - import { useForm } from "react-hook-form";
- The useForm hook provides several functions and properties that you can use to manage your form:
  - register: This function is used to register form fields with React Hook Form.
  - handleSubmit: This is used to handle form submissions. It takes a callback function that is called when the form is submitted.
  - errors: This represents an object containing any validation errors that occur when a form is submitted.
  - watch: This function is used to watch for changes to specific form fields. It takes an array of form field names and returns the current value of those fields.

React Hook Form -AddingValidations

- To add validation we can pass an object to the register function as a second parameter like this:

```
<input
  type="text"
  name="email"{...register("email", {required: true})}/><input
  type="password"
  name="password"{...register("password", {required: true,minLength: 6})}/>
```

- If there is any error for any of the input field, the errors object will be populated with the type of error which we're using to display our own custom error message like this

```
{errors.email && errors.email.type === "required" &&
                              (<p className="errorMsg">Email is required.</p>)}
{errors.email && errors.email.type === "pattern" &&
                               (<p className="errorMsg">Email is not valid.</p>)}
```

React Hook Form -DefaultValues

- The useForm hook accepts a list of options, one of which is defaultValues.
- Using defaultValues we can set initial values for the form elements and re-set them when moving from one page to another like this

```
const { user } = props;
const {
    register,
    handleSubmit,formState: { errors }} = useForm({

defaultValues: {first_name: user.first_name,last_name: user.last_name
    }});// JSX<Form.Control
    type="text"{...register("first_name")}/><Form.Control
    type="text"{...register("last_name")}/>
```

React Hook Form -ResettingValues

- we can call the reset function returned by the useForm hook to clear the form data

```
const { reset } = useForm();

reset({username: "Alex",email: "alex@example.com",password: "Test@123"});
```

React Hooks

Elite
IT Academy

Hooks in React

- Hooks allow function components to have access to state and other React features.
- Hooks allow us to "hook" into React features such as state and lifecycle methods.
- Rules for Hooks
  - Hooks can only be called inside React function components.
  - Hooks can only be called at the top level of a component.
  - Hooks cannot be conditional
- Built-in React Hooks
  - State Hooks - useState, useReducer
  - Context Hooks - useContext
  - Ref Hooks - useRef, useImperativeHandle
  - Effect Hooks - useEffect, useLayoutEffect, useInsertionEffect
  - Performance Hooks - useMemo, useCallback, useTransition, useDefferredValue

useEffect Hook

- useEffect is a React Hook that lets you synchronize a component with an external system
- The useEffect Hook allows you to perform side effects in your components.
- Some examples of side effects are: fetching data, directly updating the DOM, and timers.
- Uses
  - Connecting to an external system
  - Wrapping Effects in custom Hooks
  - Fetching data with Effects
  - Specifying reactive dependencies
  - Updating state based on previous state from an Effect
  - Removing unnecessary object dependencies
  - Removing unnecessary function dependencies
  - Reading the latest props and state from an Effect
  - Displaying different content on the server and the client

useEffect Hook - Usage

- usage: useEffect(setup, dependencies?)

```
import { useEffect } from 'react';
import { createConnection } from './chat.js';

function ChatRoom({ roomId }) {
     const [serverUrl, setServerUrl] = useState('https://localhost:1234');

     useEffect(() => {
          const connection = createConnection(serverUrl, roomId);
          connection.connect();
          return () => {
               connection.disconnect();
          };
       }, [serverUrl, roomId]);
}
```

useEffect Parameters

- setup:
  - The function with your Effect’s logic.
  - Your setup function may also optionally return a cleanup function.
  - When your component is added to the DOM, React will run your setup function. After every re-render with changed dependencies,
  - React will first run the cleanup function (if you provided it) with the old values, and then run your setup function with the new values.
  - After your component is removed from the DOM, React will run your cleanup function.
- **optional** dependencies:
  - The list of all reactive values referenced inside of the setup code.
  - Reactive values include props, state, and all the variables and functions declared directly inside your component body.
  - The list of dependencies must have a constant number of items and be written inline like [dep1, dep2, dep3].
  - If you omit this argument, your Effect will re-run after every re-render of the component.
  - if you have an empty array of dependencies, it will be called only once on first render

useEffect Hook

- useEffect is a Hook, so you can only call it at the top level of your component or your own Hooks. You can’t call it inside loops or conditions. If you need that, extract a new component and move the state into it.
- If some of your dependencies are objects or functions defined inside the component, there is a risk that they will cause the Effect to re-run more often than needed. To fix this, remove unnecessary object and function dependencies. You can also extract state updates and non-reactive logic outside of your Effect.
- If your Effect wasn’t caused by an interaction (like a click), React will generally let the browser paint the updated screen first before running your Effect. If your Effect is doing something visual (for example, positioning a tooltip), and the delay is noticeable (for example, it flickers), replace useEffect with useLayoutEffect.
- If your Effect is caused by an interaction (like a click), React may run your Effect before the browser paints the updated screen. This ensures that the result of the Effect can be observed by the event system. Usually, this works as expected. However, if you must defer the work until after paint, such as an alert(), you can use setTimeout.
- Even if your Effect was caused by an interaction (like a click), React may allow the browser to repaint the screen before processing the state updates inside your Effect. Usually, this works as expected. However, if you must block the browser from repainting the screen, you need to replace useEffect with useLayoutEffect.
- Effects only run on the client. They don’t run during server rendering.

Custom Hooks

- A custom Hook is a JavaScript function whose name starts with ”use” and that may call other Hooks.
- Unlike a React component, a custom Hook doesn’t need to have a specific signature. We can decide what it takes as arguments, and what, if anything, it should return.
- every time you use a custom Hook, all state and effects inside of it are fully isolated.
-

React Router

Elite
IT Academy

React Router

- Routing is the capacity to show different pages to the user.
- That means the user can move between different parts of an application by entering a URL or clicking on an element.
- React comes without routing. And to enable it in our project, we need to add a library named react-router.
- To enable routing in our React app, we first need to import BrowserRouter from react-router-dom.
- if we need routing in our entire app, we must wrap our higher component with BrowserRouter.
- To render routes, we have to import the Route and the Routes component from the router package.
- To add links to our project, we will use the React Router again.
    - import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom"
- After importing Link, we have to update our navigation bar a bit. Now, instead of using a tag and href, React Router uses Link and to to, well, be able to switch between pages without reloading it.

React Router - Passing Parameters

- Passing parameters to routes in a React router enables components to access the parameters in a route's path.
- These parameters can be anything from a user object to a simple string or number, which can be very helpful when, for example, you want to pass data related to a specific user from one component to another.
- It can be achieved using the 'props object' or the 'useParams hook' provided by React Router Dom.
- Parameters, often referred to as **params**, are dynamic parts of the URL that can change and are set to a specific value when a particular route is matched.
- In React Router, parameters can be used to capture values from the URL, which can then be used to further personalize or specify what to render in a particular view.
- By passing parameters, we're making routes dynamic, allowing us to reuse the same component for different data based on the parameter value.

React Router - Passing Parameters

- Every route in the React Router is associated with a path.
- This path is a string or a regular expression, which React Router matches with the current URL.
- If the current URL matches the path of a route, the associated component or render prop function of that route gets rendered.
- The 'Route path' is where we can specify parameters that we want to pass to our components.

```
<Link to={`/about/${name}/${id}`}>About</Link>
<Route path="/about/:name/:id" element={<About />} />
```

- React Router provides a hook called useParams() that we can use to access the match params of the current route. This hook makes it easy to extract params from the URL.

```
function BlogPost() {
    let { postId } = useParams();
    return <div>Post ID: {postId}</div>;
}
```

Context API

Elite
IT Academy

The Context API

- The Context API provides a means to share values like state, functions, or any data across the component tree without passing props down manually at every level.
- This is particularly useful for global data that many components need to access.
- The Context API is versatile and can be used in various scenarios where managing state and sharing data across multiple components is necessary.
    - **Global state management in medium to large apps**: the Context API can handle global state management like cart items in an e-commerce app or the currently playing song in a music app.
    - **Authentication management**: using the Context API and other solutions like it to manage the auth state is a common use case for it.
    - Other use cases are localization, user preferences like notification settings, open, close, and toggle states of a modal, API request management, breadcrumbs navigation, step progress, and any other point where the state is involved.

React Contexts

- **Used to share state across multiple components.**

Step1: Create the context object

```
import React from 'react'

const MyContext = React.createContext({
... State objects
}
export default MyContext
```

Step 2: Wrap components that should have access to the context

```
import MyContext from "./store/MyContext"

<MyContext.Provider> value = {{
                    ... initialize context
                    }}
... enclosed Components that will have
access to the state.
</MyContext.Provider>
```

React Contexts

Step 3: Listen to the context

Method 1: Using Consumer

```
return(
  <MyContext.Consumer>
  {(ctx) => {
      return (
      ... JSX - ctx.state can be accessed
       )
   }})
```

method 2: useContext hook

```
import React, {useContext} from 'react'
import MyContext from '../../store/MyContext'

// in functional componenent
const ctx = useContext(MyContext);
```

useContext Hook

- The useContext hook is just another way to access the context previously defined except it can now be used in a functional component.
- This still requires a Provider of this specific context type to be somewhere above the component that useContext is called in up the component tree.
- You need to use the actual MyContext object initialized with React.createContext to as the parameter in useContext—not MyContext.Provider or MyContext.Consumer.

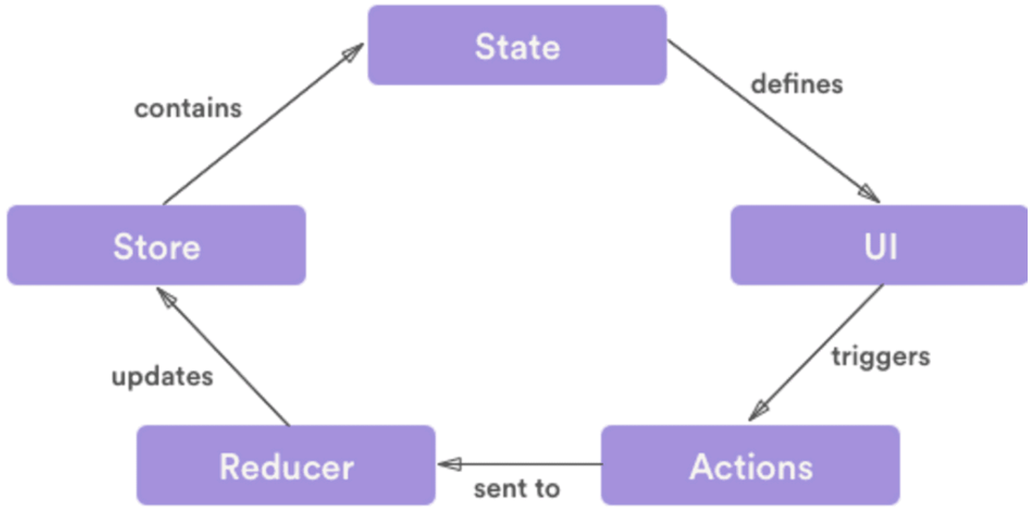
```
function functionThatUsesContext() {
   const value = useContext(MyContext);
   // read or subscribe to any of the values of MyContext object
}
```

React Redux

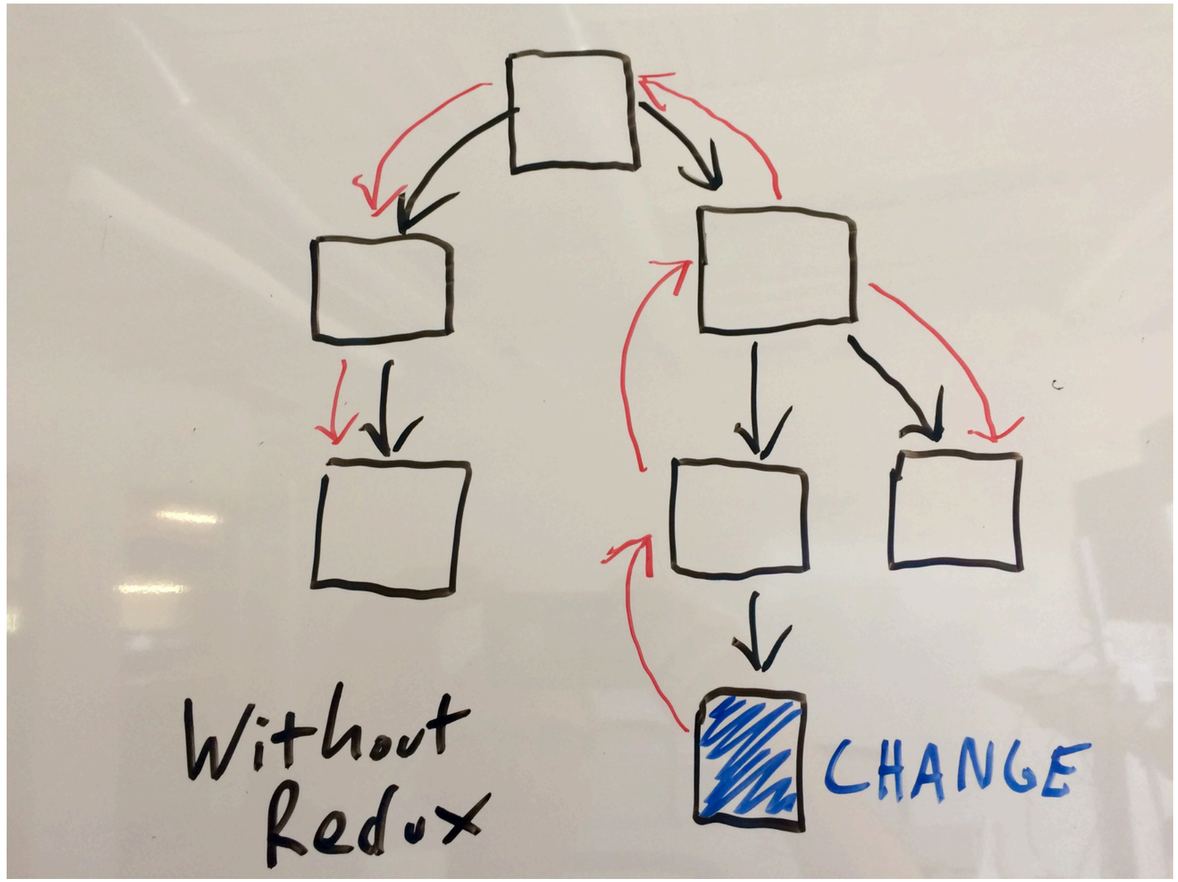
Elite
IT Academy

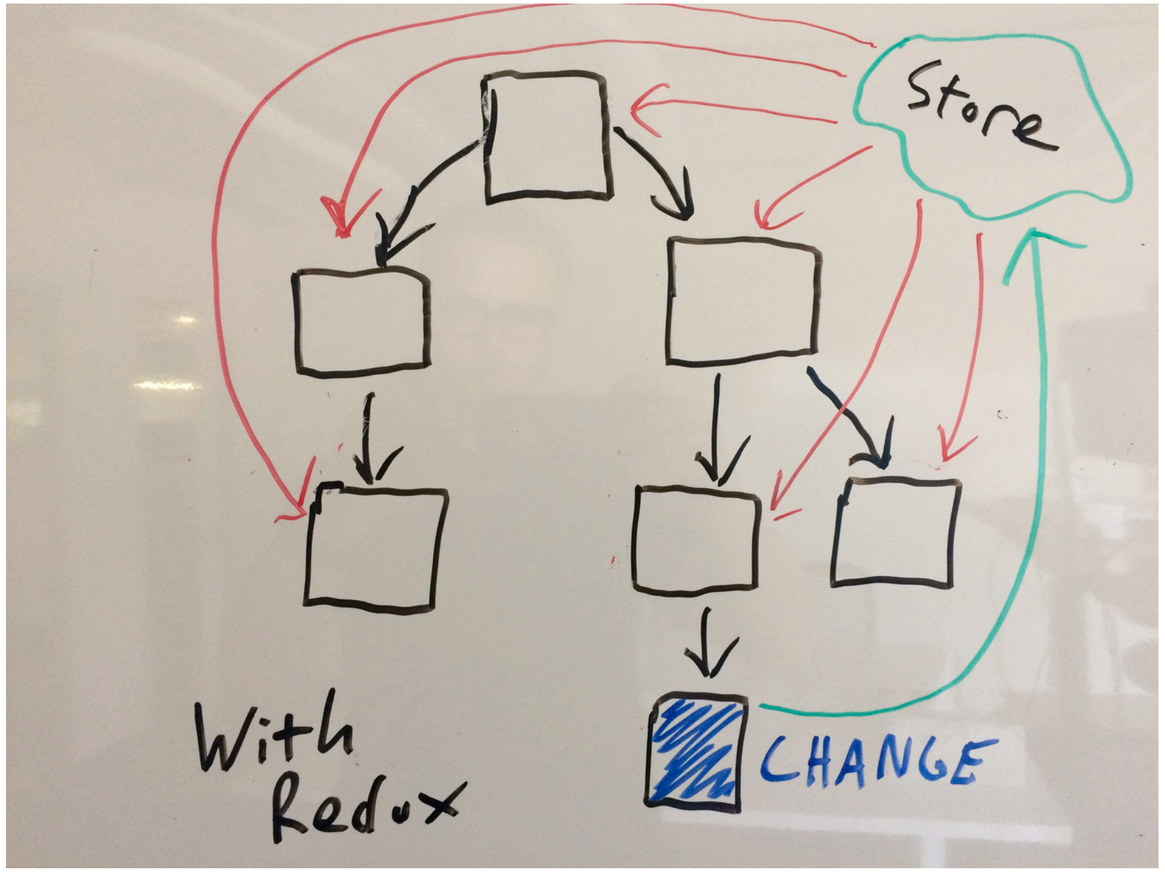
What is REDUX?

- As your React app grows, state generally becomes more and more spread out amongst different class-based components.
- This is messy. Plus, state is tightly coupled with the view logic.
- Redux provides a central store that holds the state of your application.
- Each component can access the stored state directly, without sending it from one component to another.
- Redux helps to mitigate those issues by managing state in a single place outside of React, promoting-
  - A single source of truth
  - Seperation of concerns
- There are three building parts:
  - actions,
  - store,
  - and reducers.



What is REDUX?





What is REDUX?

- Redux is a tool based on the Flux pattern for managing application state.
- Principles
  - Single source of truth: The state of your whole application is stored in an object tree within a single store.
  - State is read-only: The only way to change the state is to emit an action, an object describing what happened.
  - Changes are made with pure functions: To specify how the state tree is transformed by actions, you write pure reducers.

Action -> Reducer -> State -> View

Using Redux for app-wide state

- Kinds of State:
  - Local State
  - Cross-Component State
  - App-Wide State
- Redux is used to create a Central store of data (State)
- Components can subscribe to the store, and are notified when the data changed
- Components cannot directly manipulate data in the store
- Reducer functions are used to update the store data
- Components trigger a change in the data store by dispatching Actions
- Redux forwards Actions to the reducer which performs the Actions which changes the state, and then subscribing components are notified about the change

npm install --save redux react-redux

What is a State in Redux

Step 1: Create a State object

```
// Define an initial state value for the app
const initialState = {
 value: 0,
 ... Other state values
}
```

- Redux apps normally have a JS object as the root piece of the state, with other values inside that object.

Reducer function in Redux

- The reducer receives two arguments, the current state and an action object describing what happened.
- When the Redux app starts up, we don't have any state yet, so we provide the initialState as the default value for this reducer

Create a Reducer Function

```
function counterReducer(state = initialState, action) {
 // Reducers usually look at the type of action that happened
 // to decide how to update the state
 switch (action.type) {
      case 'counter/incremented': return { ...state, value: state.value + 1 }
      case 'counter/decremented'  return { ...state, value: state.value - 1 }
      default:                    return state
 }
}
```

Action Objects

- Action objects that should have two properties, one describing the type of action, and one describing what should be changed in the app state.
- Based on the type of the action, we either need to return a brand-new object to be the new state result, or return the existing state object if nothing should change.
- The state is updated immutably by copying the existing state and updating the copy, instead of modifying the original object directly.

Create a Action Object

example:

```
export const startAction = {type: "rotate",payload: true};
```

Creating a Store

Step 1: Create a Store

```
import { createStore } from 'redux';

const store = createStore(counterReducer);
```

- We defined a reducer, which takes the previous state and an action (which we have yet to define) and returns a new state .
- The reducer is then passed as an argument into createStore, setting the initial state of the app.

Provide the Redux Store to React

Step 1: Create a Store

```
import App from './App'
import store from './app/store'
import { Provider } from 'react-redux'

// As of React 18
const root =
ReactDOM.createRoot(document.getElementById('root'))

root.render(
 <Provider store={store}>
     <App />
 </Provider>
)
```

- Once the store is created, we can make it available to our React components by putting a React Redux <Provider> around our application in src/index.js.
- Import the Redux store we just created, put a <Provider> around your <App>, and pass the store as a prop

Connect to the Redux Store

```
import { connect } from "react-redux";
export default connect(mapStateToProps,
                       mapDispatchToProps)(App);

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

const mapDispatchToProps = dispatch => ({
startAction: () => dispatch(startAction),
stopAction: () => dispatch(stopAction)
});
```

- We need to connect our component to our redux store so we import connect from react-redux.

- We need to retrieve the state from our store
- mapStateToProps: this is used to retrieve the store state

- And to specify that we want the start and stop actions to be used for changing the state.
- mapDispatchToProps: this is used to retrieve the actions and dispatch them to the store

Connect to the Redux Store

- mapStateToProps function should return a plain object that contains the data the component needs:
  - Each field in the object will become a prop for your actual component
  - The values in the fields will be used to determine if your component needs to re-render

```
function mapStateToProps(state) {
 return {
    a: 42,
    todos: state.todos,
    filter: state.visibilityFilter,
   }
}
```

	`(state) => stateProps`	`(state, ownProps) => stateProps`
`mapStateToProps` runs when:	store `state` changes	store `state` changes or any field of `ownProps` is different
component re-renders when:	any field of `stateProps` is different	any field of `stateProps` is different or any field of `ownProps` is different

Connect to the Redux Store

- The second argument passed in to connect, mapDispatchToProps is used for dispatching actions to the store.
- With React Redux, your components never access the store directly - connect does it for you. React Redux gives you two ways to let components dispatch actions:
  - By default, a connected component receives props.dispatch and can dispatch actions itself.
  - connect can accept an argument called mapDispatchToProps, which lets you create functions that dispatch when called, and pass those functions as props to your component.

```
render() {
 return <button onClick={
   () => this.props.toggleTodo(
            this.props.todoId)} />
  }

const mapDispatchToProps = dispatch => {
 return {
    toggleTodo: todoId =>
         dispatch(toggleTodo(todoId))
 }
}
```

Using Redux Hooks (useSelector & useDispatch)

In the component where the store needs to be accessed

```
import {useSelector, } from 'react-redux';
```

In functional component, subscribe to the store by using the useSelector hook

```
   const myData = useSelector(state => state.myData);
```

Now the store is subscribed to any changes in myData will reflect in the component via the variable myData.

Dispatching actions to the Store

In the same component where the store needs to be accessed, add the dispatch hook

```
import {useSelector, useDispatch} from 'react-redux';
```

In the functional component,

```
   const dispatch = useDispatch();
```

In handler to access the store

```
   const saveHandler = () => {
       dispatch({type: 'savemydata', value: data});
    }
```

Using Redux Toolkit

npm install @reduxjs/toolkit

Step 1: Creating a Slice

```
import {createSlice} from '@reduxjs/toolkit'

const mySlice = createSlice({
    name: "mySlice",
    initialState: {myState: ""},
    reducers: {
       saveMyState(state, action) {
          state.myState = action.payload;
          }
      }
});
```

Using Redux Toolkit

Step 2: Configure the slice

```
import {createSlice, configureStore} from '@reduxjs/toolkit'

const store = configureStore({
      reducer: {
         myState: mySlice.reducer
         }
      });
```

Using Redux Toolkit

Step 3: Export action for each slice in the store

```
export const mySliceActions = mySlice.actions;
```

Step 4: In the component using the slice

```
import {mySliceActions} from "./store/store'
const myData = useSelector((state) => state.myState.myData);
```

Step 5: In the dispatch use the action object to dispatch a call to the method in the action

```
dispatch(mySliceActions.saveMyState(data));
```

React Hooks

Elite
IT Academy

What are React Hooks?

Elite IT Academy

- React hooks were introduced in React v16.8.
- Their most basic purpose is to give functions the ability to manage (and share) state, effects, and much more.
- Simultaneously simplifying expected behavior, increasing the potential for extensibility, and improving performance.
- React hooks reduce the number of components in the tree, significantly, making it easier to get a handle on the layout of the component tree and debug without having to wade through tons of Higher Order Components.
- Hooks also continue with the general pattern that React has been edging towards for a while in that hooks allow, and encourage, the use of "vanilla" JavaScript by taking (more) advantage of closures and they allow more avenues to implement native JavaScript instead of adding more framework-specific abstractions.

useState Hook

- The useState hook is similar to its counterpart with classes this.setState
- In general, is meant for more granular state management and can contain any type, for managing larger state objects, arrays, etc., useReducer is a better-suited hook.
- The most common pattern used is array destructuring but if you are unfamiliar with that pattern, you can always just use the first two elements in the array.
- The naming of both the state holder and the state changer are entirely up to you, but just as the convention with hooks is to start them with use, the convention for the state setter is to start it with set.

```
import React, { useState } from 'react';
function Counter() {
      const [count, setCount] = useState(0);
      return (<div>
       <p>The current count is {count}</p>
       <button onClick={() =>
             setCount(count + 1)}>
             }>
             Increment Counter
       </button>
       </div>);
}
```

useEffect Hook

- The useEffect hook deals with side effects and runs after every completed render
- It gives you the opportunity to manage any side effects
- There are a few other features that are important to notice with different implementations of useEffect.
- useEffect can have a dependency array as the last parameter where the props, state or other kinds of values can be placed.
- This prevents unnecessary re-renders, as useEffect will only run if one of its dependent values changes.
- useEffect can emulate the functionality of componentWillUnmount by simply returning a function that cleans up after itself.
    - In this returned function, you can unsubscribe from events that were subscribed to inside useEffect or unreferenced anything that is no longer required.
- Dependencies argument of useEffect(callback, dependencies) lets you control when the side-effect runs. If dependencies are:
    - Not provided: the side-effect runs after every rendering.
    - An empty array []: the side-effect runs once after the initial rendering.
    - Has props or state values [prop1, prop2, ..., state1, state2]: the side-effect runs once after initial rendering and then only when any dependency value changes.

useEffect Hook

```
import React, { useState, useEffect } from 'react';
function FriendStatus(props) {
   const [isOnline, setIsOnline] = useState(null);
    useEffect(() => {
      function handleStatusChange(status) {
          setIsOnline(status.isOnline);}
          ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
          // Specify how to clean up after this effect:
          return function cleanup() {
                 ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
                 };
       }, [props.friend.id]); // Only re-subscribe if props.friend.id changes
       if (isOnline === null) {return 'Loading...';}
       return isOnline ? 'Online' : 'Offline';
}
```

useReducer Hook

- While useState is an effective way to manage one or more simple state structures
- useReducer is designed to handle larger state structures that can be multiple levels deep.
- The actions contained within the reducer are dispatched by name with the dispatch function and the reducer uses that information to perform an action on the state to create the new state.
- It can, of course, get more complicated the deeper your state structure goes you will need to remember to return the previous state along with the changes.
- This is usually done with Object Spread Syntax.

useReducer Hook

```
const initialState = {count: 0};

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

```
function Counter() {
   const [state, dispatch] =
         useReducer(reducer, initialState);
   return (<>
      Count: {state.count}
     <button onClick=
           {() => dispatch({type: 'increment'})}>+
     </button>
     <button onClick=
           {() => dispatch({type: 'decrement'})}>-
     </button></>
     );
}
```

useReducer Hook. Spread Syntax

```
const initialState = {count: 0,
           someOtherStateVar: [1,2,3]};

function reducer(state, action) {
   switch (action.type) {
      case 'increment':
          return {...state,     // spreads the entire Object state into this object
                  count: state.count + 1 // overrides the count part of the previous state
                 };
      case 'decrement':
           return {...state,
             count: state.count + 1};
      default:throw new Error();
     }
}
```

File Upload

Elite
IT Academy

Uploading files to the server

- Uploading files in a React app, such as images, documents, or any other file types, typically follows a structured approach:
  - **User File Selection:** The journey begins with allowing the user to select a file. In React, this is commonly achieved by utilizing the <input> element with its type attribute set to “file”. When a file is chosen, the event handler listens to any changes or interactions with the file input and updates the application’s state with the selected file’s information.
  - **Server Communication:** Once the file information is captured and stored in the application’s state, the next step is sending it over to a server. This could be for processing, storage, or any other backend operation. Tools like axios or the native fetch API aid in making asynchronous HTTP requests to servers. It’s crucial to wrap the selected file in a FormData object, ensuring the data is properly formatted for transmission.
  - **Feedback & Response Handling:** Upon initiating communication with the server, always anticipate two outcomes: success or failure. Implementing feedback mechanisms like success messages, error alerts, or even displaying the uploaded file helps improve user experience.
  - **Error Handling:** Issues might arise during the upload process, be it network glitches, file format mismatches, or server-side errors. Informative error messages and alternative solutions can steer users in the right direction.
  - **External Libraries and Tools:** While React provides a solid foundation, sometimes external libraries or tools can expedite the development process. Tools like axios simplify HTTP communications.