
ReactJS

It relies on reusable components, not templates. for UI dev, allowing developers to render views where data changes over time.

Cmd - node -v	Cmd - npm -V
- V12.19.0	- 6.14.8

Cmd - npx Create - React - App react - essentials

- npx: Package runner
- Create - React - App: tool to Build the Project
- react - essentials: Name of the Project

Cmd - npm start - starts the development server.

cmd - npm run build - Bundles the app into static files 4 Prod.

cmd - npm test - Starts the test runner.

Cmd - code . - opens the Visual Studio code.

Lifecycle Management

Lifecycle management in React refers to understanding and effectively managing the stages or phrases that a component goes through during its existence. This understanding is crucial for handling tasks like initialization, updates, and cleanup of resources.

1. Mounting

This phase occurs when a component is being initialized and inserted into the DOM for the first time.

Key methods / hooks: 'constructor', 'render', 'componentDidMount', 'UseEffect' (with empty dependency array).

Methods / Functions:

- **Constructor () (class components only):** used for initializing state and binding method.
- **render ():** Responsible for rendering JSX elements to the DOM.

- `ComponentDidMount()` (class components only): Invoked immediately after the component is mounted (inserted into DOM). It's often used for initialization tasks that require DOM nodes or data fetching.

- `UseEffect(() => {}, [])`: Hook that allows performing side effects (like data fetching or setting up subscription) in function components. When used with an empty dependency array, it mimics componentDidMount.

2. Updating

- This phase occurs when a component re-renders due to changes in its props or state.

Key methods / hooks :- shouldComponentUpdate, render, componentDidUpdate, useEffect (with dependency array).

Methods / Functions :-

shouldComponentUpdate (nextProps, nextState) (class components only) :-
Allows optimization by determining if the component should re-render based on changes in props or state.

render () : Renders updated JSX based on new props / state.

componentDidUpdate (prevProps, prevState) (class components only) :-
Invoked immediately after an update occurs. It's used for performing additional updates or interacting with DOM based on changes.

3. Unmounting

- This phase occurs when a component is being Removed from the DOM.
- Key methods/Hooks: `componentWillUnmount`, `useEffect` (clean up function).

`componentWillUnmount()` (class components only) Invoked immediately before a component is unmounted and destroyed. It's used for clean up tasks like canceling network requests, clearing timers, or unsubscribing from subscriptions.

`useEffect(() => { return () => {} }, [])`: The clean up function returned by useEffect runs before the component is removed from the DOM. It's used for clean up tasks in Function Components.

- Component is a function that returns a UI.

```
function App() {
    return (
        <div className="App">
            <Header />
            <h2> Main </h2>
            <h3> Footer </h3>
        </div>
    );
}

function Header () {
    return (
        <header>
            <h1> Eve's kitchen </h1>
        </header>
    );
}
```

Component here is a function

Below

Functional vs class components

Functional Components and Class Components are two types of components in React, each with its own syntax and capabilities.

1. Functional Components

- Defined as javascript functions.
- Use useState and useEffect, and other Hooks for state management and side effects since React Hooks were introduced in React 16.8.
- Concise syntax with arrow functions or regular function declarations.

```
import React from 'React';

const FunctionalComponent = () => {
    return <div> Hello, I'm a functional component </div>;
};

export default FunctionalComponent;
```

2. Class Components

- Defined as ES6 classes that extend React.Component
- Use `this.state` and `this.setState` for state management.
- Use lifecycle methods (componentDidMount, componentDidUpdate) for managing side effects and updating the UI.

```
Import React, {Component} from 'React';

Class ClassComponent extends Component {
    render () {
        return <div> Hello, I'm a class component! </div>
    }
}

export default classComponent;
```

Industry standard

In the Realm of React development, several pratices and patterns have emerged as industry standards. These standards are based on best pratices, community consensus, and the evolution of React itself. Here are some key aspects that constitute industry standard in React development :

1. Functional Components with Hooks

- Fonctional components : using Functional components for their simplicity and readability.
- Hooks : Utilizing Hooks (useState, useEffect, useContext, etc) for managing state, side effects, context and more. Hooks have become the preferred way to write React components since React 16.8.

2. State Management

- Local state : Using useState for local component state management with functional components.
- Global state : Employing Context API (useContext) for managing global state when needed, or using state management libraries like Redux or Mobx for more complex state management scenarios.

3. Component Lifecycle

- Lifecycle Methods: prioritizing the use of useEffect Hook for managing side effects and lifecycle events (componentDidMount, componentDidUpdate, componentWillUnmount).

- cleanup: Ensuring proper cleanup of resources using cleanup functions in useEffect.

4. Component Composition and reusuability

- Composition: Emphasizing component composition to create reusable and maintainable UI components
- High-Order Components (HOCs): Using HOCs or Render Props for sharing behavior between components
- Component libraries: Leveraging component libraries like Material UI, Ant Design, or others for consistent UI design + functionality.

5. Code organization and Architecture

- Folder Structure: organizing code into modular components, containers, services, and utils for better maintainability and scalibility.

- Separation of Concerns: Separating UI components from business logic and state management concerns.

6. Performance Optimization

- Memoization :- Using React.memo or useMemo for memoizing expensive computations.
- Virtualizations : Implementing virtualization techniques (like using react-virtualized or react-window) for

What is useEffect ?

useEffect is a function in React that lets you perform actions [called side effects] in your components

- Fetching data from an API.
- Updating the DOM.
- Setting up subscriptions or timers.

What is dependency array?

The dependency array is an optional second argument you pass to 'useEffect'. It tells React when to run your side effect.

Empty dependency []

When you pass an empty array to useEffect, it means :-

- Run this side Effect only once after the first Render of the component.
- Don't run it again on subsequent re-renders.

Why use an Empty Dependency Array?

jsx Copy

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

function FetchDataComponent() {
  useEffect(() => {
    // This code runs only once when the component first appears
    console.log('Component mounted');

    // Simulate fetching data from an API
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => console.log(data));

    // Cleanup function runs when the component is about to disappear
    return () => {
      console.log('Component will unmount');
    };
  }, []);  // Empty array means run only once

  return <div>Check the console for lifecycle logs and fetched data.</div>;
}

export default FetchDataComponent;
```

Empty Dependency Array ([ ]) Tells React to run the effect only once when the component is first Rendered.

No Array Runs the effect after every Render (not Recommeded for operations like data fetching).

Array with variable [var1, var2] Runs the effect when any of the listed variable change.

React Hooks

React Hooks are functions that lets you use State and other React features in functional components.

They were introduced in React 16.8 to allow functional components to have features previously only available in Class components, such as local components state and lifecycle methods.

Basic Hooks

1. useState
2. useEffect
3. useContext

Additional Hooks

4. useReducer
5. useCallBack
6. useMemo
7. useRef
8. useImperativeHandle
9. useLayoutEffect
10. useDebugValue

useHooks.com

hook is just a function that we can use for any sort of repeatable code

a custom hook is a function and this function is going to always start with the keyword use.

```
function useInput(initialValue) {
    const [value, setValue] = useState(initialValue);
    return [
        {
            value,
            onChange: (e) => setValue(e.target.value)
        },
        () => setValue(initialValue);
    ]
```

element 1

element 2

```
Function useInput(initialValue) {
    Const [Value, setValue] = useState(initialValue);
    return [
        {
            value,
            onChange: (e) => setValue(e.target.value)
        },
        () => setValue(initialValue);
    ]
```

element 1

element 2

```
Function App() {
    const [titleProps, resetTitle] = useInput("");
    const [colorProps, resetColor] = useState("#0000");
    const submit = (e) => {
        e.preventDefault();
        alert(`${titleProps.value}, ${colorProps.value}`)

        resetTitle();
        resetColor();
    };
```

UseState

purpose: Adds State to Functional Components.

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

function Counter() {
  const [count, setCount] = useState(0); // Initialize state with 0

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

export default Counter;
```

UseState returns an array with two elements: the current State value (count) and a Fenction to update it.

You can use SetCount to update the State, and React will re-render the component with new state.

Another example of useState

```
Import { FaStar } from "react-icon/fa"; // react-icons library.
Const createArray = (length) => [...Array(length)];

Function Star ({ selected = false, onSelect }) {
  return <FaStar color={ selected ? "red" : "gray" }
               onClick={ onselect } />
  );
}

Function StarRating ({ totalStars = 5 }
  Const [selectedStars, setselectedStars] = useState(0);
  <>
    return createArray(totalStars).map((n,i) => (
      <Star key={i} selected={selectedStars > i}
        onselect={() => SetSelectedStars(i+1)} />
  </>
  ));
}

Function App() {
  return < StarRating />
```

UseEffect

purpose :- Handle side Effects in Functional Components

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

function DataFetcher() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('https://api.example.com/data')
      .then(response => response.json())
      .then(data => setData(data));

    return () => {
      console.log('Cleanup if necessary'); // Cleanup function
    };
  }, []); // Empty dependency array means run only once

  return (
    <div>
      {data ? <p>{data.someValue}</p> : <p>Loading...</p>}
    </div>
  );
}

export default DataFetcher;
```

- UseEffect runs the code inside it after the component renders.

The optional dependency array ([]) determines when the effect runs.

- Empty array ([]) : Runs once.
- No array : Runs after every render.
- Array with dependencies ([dept1, dept2]) : Runs when any dependency changes.

UseContext

purpose :- provides a way to share values (like themes, User data) between components without passing props manually at every Level.

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

const ThemeContext = React.createContext('light');

function ThemedComponent() {
  const theme = useContext(ThemeContext); // Use context value

  return <div>The current theme is {theme}</div>;
}

export default ThemedComponent;
```

UseContext takes a Context Object (created with React.createContext) and Returns the current context value.

Context API

The context API in React is a feature that allows you to share state and other data across your component tree without having to pass props down manually at every level. It provides a way to create global variables that can be passed around within a React application, which is useful for themes, user authentication, and other global states.

How the Context API works

1. Creating a Context

You create a context using `React.createContext()`. This returns a context object which you can use to create a provider and a consumer.

2. Provider

The provider component is used to wrap the part of your application where the context should be available. It accepts a value prop which can be any data you want to share across your components.

UseReducer

purpose :- Manages more complex state logic compared to 'useState'.

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

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, { count: 0 });

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })}>Increment</
button>
      <button onClick={() => dispatch({ type: 'decrement' })}>Decrement</
button>
    </div>
  );
}

export default Counter;
```

- UseReducer : takes a reducer function and an initial state.
- It returns the current state and a dispatch function to update the state.
- The reducer function specifies how the state updates based on actions.

```
import { useReducer } from "react";

function App() {
  const [checked, setChecked] = useReducer((checked) => !checked, false);
  );
  return (
    <div className="App">
      <input
        type="checkBox"
        value={checked}
        onChange={setChecked}
      />
      <label>
        {checked ? "checked" : "not checked"}
      </label>
    </div>
  );
```

Arg 1.

Arg 2 → false

UseCallBack

5. UseCallBack :- Memoizes a function to prevent it from being recreated on every Render.

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

function Button({ onClick }) {
  return <button onClick={onClick}>Click me</button>;
}

function Counter() {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    setCount(count + 1);
  }, [count]); // Dependencies: recreate only if count changes

  return (
    <div>
      <p>Count: {count}</p>
      <Button onClick={handleClick} />
    </div>
  );
}

export default Counter;
```

- UseCallBack returns a memoized version of the callBack function that only changes if one of the dependencies has changed.

- Useful to optimize performance, especially when passing callBacks to child components.

UseMemo

6. UseMemo :- Memoizes a value to prevent recalculating it on every render.

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

function ExpensiveCalculation({ count }) {
  const computedValue = useMemo(() => {
    // Some expensive calculation
    return count * 2;
  }, [count]); // Dependencies: recalculate only if count changes

  return <div>Computed value: {computedValue}</div>;
}

function Counter() {
  const [count, setCount] = useState(0);

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

export default Counter;
```

- UseMemo returns a memoized value that only recomputes if the dependency change.
- Useful for expensive calculations or operations.

UseRef

To UseRef is Access and Interact with DOM element or keep a mutuable object that persists across renders.

jsx Copy

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

function TextInputWithFocusButton() {
  const inputEl = useRef(null);

  const onButtonClick = () => {
    inputEl.current.focus();
  };

  return (
    <div>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </div>
  );
}

export default TextInputWithFocusButton;
```

- UseRef returns a mutuable object whose .current property is initialized to the passed argument.
- Can be Used to access DOM elements directly or persist a value across Renders without causing re-renders.
- UseRef as being a hook that is going to reach out to some sort of UI element and gets its value.
- Uncontrolled Component, We're saying create this little container, give us whatever that value is and Unlike UseState where the Component will re-render if there's some sort of change, UseRef is not going to re-render. We always are going to have to reach out to the current value, and get its value.

```
Import { useRef } From "react";

Function App() {
  Const txtTitle = useRef();
  Const hexColor = useRef();

  Const submit = (e) => {
  e.preventDefault(); // Prevent the page from refreshing.
  Const title = txtTitle.current.value;
  const color = hexColor.current.value;
  alert( title, color );
  txtTitle.current.value = "";
  hexColor.current.value = "";
return (
  <form onSubmit = {submit}>
    <input
      ref = {txtTitle}
      type = "text"
      placeholder = "color title"
    />
);
```

8. UseImperativeHandle

purpose :- customizes the instance value that is exposed when using ref with a parent Component.

jsx Copy

```jsx
import React, { useRef, forwardRef, useImperativeHandle } from 'react';

const FancyInput = forwardRef((props, ref) => {
  const inputRef = useRef();

  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));

  return <input ref={inputRef} type="text" />;
});

function ParentComponent() {
  const inputRef = useRef();

  return (
    <div>
      <FancyInput ref={inputRef} />
      <button onClick={() => inputRef.current.focus()}>Focus the input</
button>
    </div>
  );
}

export default ParentComponent;
```

9. UseLayoutEffect

9. useLayoutEffect

similar to useEffect, but runs synchronously after all DOM mutations. useful for reading layout from the DOM & synchronously re-rendering.

jsx Copy

```jsx
import React, { useLayoutEffect, useRef } from 'react';

function LayoutEffectExample() {
  const divRef = useRef();

  useLayoutEffect(() => {
    const { height } = divRef.current.getBoundingClientRect();
    console.log('Height:', height);
  });

  return <div ref={divRef} style={{ height: '100px' }}>Hello</div>;
}

export default LayoutEffectExample;
```

10. UseDebugValue

10. UseDebugValue: Customizes the label shown in React DevTools for custom Hooks.

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

// Custom hook to simulate fetching user status
function useUserStatus(userID) {
  const [isOnline, setIsOnline] = useState(null);

  // Use useDebugValue to provide a meaningful label in React DevTools
  useDebugValue(isOnline ? 'Online' : 'Offline');

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    // Simulate a subscription to a user's online status
    const subscription = {
      subscribe: (id, callback) => {
        console.log(`Subscribed to user ${id}`);
        // Simulate a status change
        setTimeout(() => callback({ isOnline: Math.random() > 0.5 }),
1000);
      },
      unsubscribe: (id) => {
        console.log(`Unsubscribed from user ${id}`);
      }
    };

    subscription.subscribe(userID, handleStatusChange);
    return () => subscription.unsubscribe(userID);
  }, [userID]);

  return isOnline;
}

export default useUserStatus;
```

```
import React from 'react';
import useUserStatus from '../hooks/useUserStatus';

function UserStatus({ userID }) {
  const isOnline = useUserStatus(userID);

  if (isOnline === null) {
    return <div>Loading...</div>;
  }

  return (
    <div>
      User {userID} is {isOnline ? 'Online' : 'Offline'}
    </div>
  );
}

export default UserStatus;
```

```
import React from 'react';
import UserStatus from './components/UserStatus';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <h1>React Hooks Demo with useDebugValue</h1>
      </header>
      <main>
        <UserStatus userID={1} />
      </main>
    </div>
  );
}

export default App;
```

- UseDebugValue helps with debugging by labeling the labeling the hook value in React DevTools.

- Useful for custom Hooks to provide more context.

→ A reducer function's most simple definition is it takes in the current state & it returns a new state.

```
const [checked, toggle] = useReducer(
    (checked) => !checked,   ← function
    false                    ← initial state.
);
```

→ How to type React Props.

Since React props are used to send transmit data between one React component to another, there are many types that you can use to type React props.

To write the types of your props, you need to add a colon & the object literal notation (: {}) next to the destructuring assignment of the children prop at the component declaration.

Here's an example of typing a string & a number props:

```
const App = ({ title, score }: { title: string, score: number }) => {
    <h1>{title} = {score}</h1>
)
```

```
function App() {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch('https://api.github.com/users/${login}')
    .then((response) => response.json())
    .then(setData);
  }, []);

  if (data) {
    return <div>{JSON.stringify(data)}</div>;
  }

  return <div>No User Available.</div>
}
```

→ Any file that ends in test.js → that is a test file.

Useful commands

`npm init`

This utility will walk you through creating package.json file. It only covers the most common items, and tries to guess the defaults.

`npm install express` - installs the express (Backend)

`npm i mongoDB` - installs mongoDB drivers

`npm i react react-dom` - installs react libraries.

- `npm i -D eslint babel-eslint` eslint-plugin-react

← `npm i prop-types` Helps improve the React App

- `npm i -D nodemon` restarts the node anytime files are changed.

- `npm i -D babel-loader @babel/cli @babel/core @babel/preset-env.`

`@babel/preset-react @babel/plugin-proposal-class-properties.`

↗ Loads the Babel & it libraries

The `--save-dev` option in npm (node package manager) is used to add a package to the 'devDependencies' section of your "package.json" file. This indicates that the package is only needed during the development phase and not in the production build of your App.

What `--save-dev` Does

Installs the package: The specified package is installed & added to your project's 'node_modules' directory.

Updates 'package.json': The package is added to the 'devdependencies' section of your 'package.json' file.

dependencies:- These are the packages required for your Application to run in production. They are installed when someone installs your project (e.g. `npm install` without any flags).

devdependencies:- These are the packages required only for development purposes, such as testing, building, or compiling your code. They are not installed when the 'NODE_ENV' environment variable is set to 'production'.

`Install eslint --save-dev`

Add `'eslint'` to the `'devdependencies'` section of your `'package.json'`

```
{
    "devdependencies" : {
        "eslint" : "^7.32.0"
    }
}
```

`npm install axios` // Installing a package as a Regular Dependency.

1. Installs the axios.
2. Add `'axios'` to the `'dependencies'` section of your package.json.

```
{
    "dependencies" : {
        "axios" : "^0.21.1"
    }
}
```

When to use '--save-dev'

Use `--save-dev` when installing packages that you only need during development, such as:

Linters & formatters: 'eslint', 'prettier'

Testing libraries: 'jest', 'mocha', 'chai'

Build tools: 'webpack', 'gulp', 'babel'

Type definitions (when using Typescript):
@types/react, @types/node

Learning package.json

The **BrowserList** key in your **package.json** file specifies which Browser your Application supports. This helps tools like Autoprefixer and Babel to only add the neccessary polyfills & prefixes for the specified Browsers, optimizing your bundle size & ensuring compatibility.

production :- specifies the Browsers to support in the production Build. The example configuration supports :

- Browsers with more than 0.2% market share (`'>0.2%'`)
- Browsers that are not officially dead (`'not dead'`)
- All versions of opera mini (`'not op_mini all'`)

development :- Specifies the browsers to support during development.

- The last version of chrome (`'last 1 chrome version'`)

Type Definitions

Type definitions are declarations that describe the types of variables, functions, and objects in your code. They provide a way for Typescript (and other typed languages) to understand the types of data being used, which helps with type checking, autocomplete, & other development features.

Type definitions are like a **cheat sheet** for Typescript. They tell Typescript what kind of data (types) to expect when you use certain Functions, Objects, or Variables. This helps prevent mistakes & makes your code easier to understand.

When using popular javascript libraries in Typescript, you often need extra Files that describe the types used in those libraries. These files are called type definitions files and usually have a **.d.ts** extension.

npm install lodash ↙ For typescript:

npm install @types/lodash --save -dev

Using Type Definitions

with type definitions installed, Typescript can help you write correct code:

without type definitions

Typescript doesn't know that the _.shuffle function does.

```
Import _ from 'lodash'
Let arr = [1,2,3,4];
Let shuffledarr = _.shuffle(arr); // Typescript has no clue what _.shuffle is.
```

```
Import _ from 'lodash'
Let arr: number[] = [1,2,3,4];
Let shuffledarr: number[] = _.shuffle(arr);
// Typescript understands _.shuffle
```

Writing your own Type Definitions

Sometimes, you have your own Javascript code & want Typescript to understand it. you can write a `.d.ts` File to describe the types.

```
// mylibrary.js
Function greet (name) {
    return 'hello, ${name}!';
}
module.exports = {greet}
```

create a type definition file mylibrary.d.ts

```
declare module "mylibrary" {
    export Function greet (name: string): string;
}
```

input ↑ output ↑

```
Import {greet} from "mylibrary";
Let greeting: string = greet ("world");
Console.log (greeting);
```

Explain "Import _ from lodash"

In javascript and Typescript 'import _ from 'lodash';' is a statement used to import the entire lodash library & assign it to the variable '_'. This allows you to use Lodash's utility functions within your code by calling them on '_'.

When developing a React project without Typescript, the project structure remains fundmentally similar to a typescript project. The primary difference is the use of javascript (.js) files instead of Typescript (.ts/.tsx) files.

public/

- **index.html :** The main HTML file This is the file that the React Application mounts to. Contains a `<div id="root"></div>` which acts as the mount point for the React components.

src/

assets/ : Contains static assets such as images and global styles.

images/ : stores images file.

styles/ : contains css files, like global.css for global styles.

- **Components / :** Contains reusable React components.
  - Header.js
  - Footer.js
- **Pages / :-** contains components that represents different pages of the application.
  - Homepage.JS
  - About Page.JS
- **App.JS :-** The root component that includes the main structure & routing of the application.
- **Index.js :-** The entry point of the React application. It renders the 'App' component into the DOM.

→ When you create a React project, Below files gets created.

src/index.js

javascript Copy

```
import React from 'react';
import ReactDOM from 'react-dom';
import { BrowserRouter as Router } from 'react-router-dom';
import App from './App';
import './assets/styles/global.css';

ReactDOM.render(
  <React.StrictMode>
    <Router>
      <App />
    </Router>
  </React.StrictMode>,
  document.getElementById('root')
);
```

src/App.js

javascript Copy

```
import React from 'react';
import { Route, Switch } from 'react-router-dom';
import HomePage from './pages/HomePage';
import AboutPage from './pages/AboutPage';
import ContactPage from './pages/ContactPage';
import Header from './components/Header';
import Footer from './components/Footer';

const App = () => {
  return (
    <div className="App">
      <Header />
      <main>
        <Switch>
          <Route path="/" exact component={HomePage} />
          <Route path="/about" component={AboutPage} />
          <Route path="/contact" component={ContactPage} />
        </Switch>
      </main>
      <Footer />
    </div>
  );
};

export default App;
```

Initial Setup with Create React App

To set up a new React project without TypeScript using Create React App, you can use the following commands:

.ts vs .tsx

.ts : For regular Typescript files (no JSX).

.tsx : for Typescript files with JSX (used in React Component).

Using Bootstrap

Adding Bootstrap or Any other Library

To add a library like Bootstrap for styling, you can install it via npm & import it in your `index.js` or `App.js`.

1. npm install Bootstrap
2. in src/index.js or src/index.tsx
   - Add below Line
     - Import 'bootstrap/dist/css/bootstrap.min.css';

React Hook Form

React Hook Form (RHF) is a library for managing forms in React applications. It aims to make form handling simple, efficient, and performant by leveraging React hooks. RHF provides a minimal API to register inputs, manage form state, perform validation, and handle form submissions.

Key features of React Hook form

1. **performance :-**

RHF minimizes re-renders, making it highly performant even with large forms.

2. **Ease of use :-**

provides a simple API that is easy to understand & use.

3. **Flexible Validations :-**

- Supports synchronous and asynchronous validation.
- Integrates well with validation libraries like Yup.

4. **Minimal Boilerplate :**

- Reduces the amount of boilerplate code compared to other form libraries.

5. **Tiny Bundle size :**

- Lightweight with a small bundle size.

6. **Typescript support**

- provides excellent Typescript support for type-safe forms.

<Controller>

The <Controller> Component in React Hook Form (RHF) is used to wrap controlled inputs, such as custom components or third-party components that don't directly integrate with RHF. It allows you to use these components w/ React Hook Form's API seamlessly.

The <Controller> component takes care of the registration and validation of these inputs, making it easier to work with complex or custom input elements.

npm init npm init -y

npm run Build :- it'll take all of the developer friendly code that we've been working with so far and turn it into browser friendly code and basically all of that code is going to be put into the build folder.

Step 1: Install React Hook Form and dependencies.

```
npm install react-hook-form
```

Step 2: custom input components

```
// customInput.tsx
import React from 'React';
Interface CustomInputProps {
    value: string;
    onChange: (value: string) => void;
}

Const customInput;
Preact.FC<customInputProps> = ({ value, onChange }) => {
return (
 <input type="text" value={value}
   onchange={(e) => onChange(e.target.value)} className="form-control"
   />
 );
};

export default Custom input;
```

UseForm

In React, UseForm typically refers to a custom hook that you might create or use from library to manage form state and validation more easily. one popular library that provides such functionality is 'react-Hook-Form'. This libray allows you to manage form state and validation with minimal re-renders and very simple syntax.

```
Import React From 'react';
Import { useForm } from 'react-Hook-Form';

Const MyForm = () => {
Const { register, handleSubmit, Watch, formState: { errors } } = useForm();
Const onSubmit = data => console.log(data);

return (
  <Form onSubmit = {handleSubmit(onSubmit)} />
    <Input defaultValue = "test" {... register ("example")} />
    <p> { errors. example?. message } </p>

    <input {...register("exampleRequired", {This Field is Required"})} />
    <p> { errors. exampleRequired 2. message } </p>

    <input type = "submit" />
  </Form>
);
};

  export default MyForm;
```

1. **useForm Hook:** The useForm hook is used to create the form instance, It returns several properties and methods to manage the form.

- **register:** A function to register an input.
- **handleSubmit:** A function to handle form submission.
- **watch:** A function to watch the value of an input.
- **formState: {errors}:** An object containing form errors.

2. **register:** The register function is used to register each input field with the form. It can also take validation rules.

3. **handleSubmit:** This function is used to wrap the form's submit event. It takes a callback function to handle the form data.

4. **watch:** This function allows you to watch the value of an input field.

5. **formState.errors:** Contains any validation errors for the form fields.

Setting Up Routes in the main App Component

In React Router, you typically setup your routes in the main application component (App.js or similar) to define how different URLs should Render different components. However, whether you absolutely must set up routes in the main app component depends on your application's architecture and routing needs.

Setting up routes in the main app component is a common approach because it centralizes the routing logic and ensures that routing is managed consistentcy across the entire App.

```
Import React from 'react';
Import { BrowserRouter as Router, Routes, Switch } From 'react-router-dom';

Const App = () => {
    <Router>
        <div>
        <nav>
        <ul>
        <li> <Link to = "/" > Home <Link> </li>
        </ul>
        </nav>

    <Routes>
        <Route Path = "/" element = {<Home />} />
        <Route path = "/dashboard/*" element = {<Dashboard/>} />
    </Routes>
    </div>
    </Router>
);

    export default app;
```

V6 before that, it was component

Link component

In React Router, the **Link component** is used for declarative navigation around your application. It renders an `<a>` (anchor) tag in HTML with an 'href' attribute that supports client-side navigation without a full page reload.

Key features of 'Link' component:

1. **Declarative Navigation:** Instead of manually setting 'href' attributes on anchor tags (`<a>`), you use the Link component to define navigation links in a declarative way.

2. **prevents Full page Reloads:** When a user clicks on a 'Link' component, React Router intercepts the click event & uses the HTML5 history API (pushState) to update the URL in the browser's address bar without reloading the entire page.

3. **Maintains Single-Page Application (SPA) Behavior:** Helps maintain the SPA behavior by allowing navigation between different routes defined by '`<Route>`' components without requesting new HTML pages from the server.

BrowserRouter :- Wraps the entire application and provides the context for React Router to work with client-side navigation.

Link Component : Used within the <nav> element to create navigation links (<a> tags) to different routes (to="/", to = "/about"). These links are rendered as <a> tags in the HTML output.

to prop : Specifies the destination URL or path where the 'Link' should navigate when clicked.

Additional props:

The Link component accepts additional props beyond 'to' for more advanced use cases:

- **className :** Adds custom css classes to the Rendered <a> tag.
- **Style :** Adds custom css classes to the rendered <a> tag.
- **Other Props:** Any other props like 'onClick', 'target', etc, that are valid for <a> tags can be passed to the Link Component.

React Router Version

To check the version of React Router that you are using in your project, you can typically find this information in your package.json file.

1. Using npm.

```
npm show react-router-dom version
```

This command will display the latest version of react-router-DOM available on npm. To check the version installed in your project:

```
npm list react-router-dom
```

React Router History

React Router uses HTML5 history API under the hood (pushState, replaceState, and popState events) for navigation, which allows for smooth, client-side routing experience without full page reloads. This API enables React Router to manage navigation states and URLs in a single-page application.

If you want to access the history object directly in a React component, you can use the 'useHistory' hook provided by React Router DOM.

purpose of useHistory

The useHistory hook provides a convenient way to interact with the history API and manage navigation within your React components without needing to rely on props being passed down from parent components. It encapsulates the history object and its methods, simplifying the process of navigating between different routes or modifying the browser's history stack.

Here's a simple example demonstrating how to use the Usehistory hook in React component

```
Import React from 'React';
import {useHistory} from 'React-rocter-dom';

Const MyComponent = () => {
    const history = usehistory();

    const handleClick = () => {
    // Navigate to a different route programmatically.
        history.push('/another-route');
    };

    return (
        <div>
        <button onClick={handleClick}>Go to Another Route</button>
        </div>
    );
};

    export default MyComponent;
```

Explanation

1. Importing 'useHistory' : The useHistory hook is imported from 'react-router-dom'.

2. Accessing 'history' object: By calling useHistory(), the component gains access to the history object provided by React Router.

3. Navigating programatically : Inside the component, history.push('/another-route') when the button is clicked.

4. Rendering : When the button is clicked ('onClick={handleClick}'), the handleClick function updates the URL and triggers the Rendering of the component for '/another-route'.

key methods of history object

push(path: string, state?: any): pushes a new entry onto the history stack and navigates to the specified path.

replace(path: string, state?: any): Replaces the current entry on the history stack with a new one and navigates to the specified path.

go(n: number): Moves forward or backward in the history stack by 'n' entries.

goBack(): Equivalent to history.go(-1), which navigates back to the previous entry in the history stack.

goForward(): Equivalent to history.go(1), which forwards to the next entry in the history stack.

Context API

The **context API** in React is a feature that allows you to share state and other data across your component tree without having to pass props down manually at every level. It provides a way to create global variables that can be passed around within a React application, which is useful for themes, user authentication, and other global states.

How the context API works

1. **Creating a context :**

You create a context using **React.createContext()**. This returns a context object which you can use to create a provider and a consumer.

a. **Provider :**

The provider component is used to wrap the part of your application where the context should be available. It accepts a value prop which can be any data you want to share across your components.

```
Const App = () => {
  return (
  <Router>
    <Routes>
      <Route path="/" element={<Home />} />
      <Route path="/about" element={<About />} />
      <Route path="/user/:id" element={<UserProfile />} />
      <Route path="/product/:category/:id" element={<ProductDetail />} />
      <Route path="/blog/:year?/:month?" element={<BlogArchive />} />
```

Nested Routes

```
      <Route path="/dashboard" element={<dashboard />} />
        <Route path="profile" element={<Profile />} />
        <Route path="setting" element={<setting />} />
      </Route>
      <Route path="/home" element={<Navigate to="/" />} />
      <Route path="*" element={<NotFound />} />
```

Changes And Improvements in V6

1. element 'instead' of 'component': In v6, you use the element prop with JSX elements instead of the component prop.

2. Nested Routes :- Nested Routes are now defined within the parent route's element prop, improving readability & component encapsulation.

3. useNavigate : Hook : Replaces useHistory for programmatic Navigation.

Programmatic Navigation

In React Router V6, you use the useNavigate hook to Navigate programmatically.

```
Import React from 'React';

Const MyComponent = () => {
    const navigate = useNavigate();

    Const handleClick = () => {
        navigate('/another-route');
    };
    return (
        <div>
            <button onClick={handleClick}> Go to another Route
            </button>
        </div>
    );
);
```

React Fragment or <>... </>

In React, the <> and </> syntax is a shorthand for React.Fragment. Fragments lets you group a list of children elements without adding extra nodes to the DOM. This is useful for returning multiple elements from a component's render method without wrapping them in an additional HTML element, such as div.

```
Import React from 'react';

Const MyComponent = () => {
  return (
    <div>
      <div>
        <h1> Title 1 </h1>
        <p> Content </p>
      </div>
      <div>
        <h1> Title 2 </h1>
        <p> Content </p>
      </div>
    </div>
  );
);
```

In this example, the div elements are used solely to the group the h1 and p elements together.

- Example With Fragment

```
Import React from 'React';

const MyComponent = () => {
    return (
        <>
            <h1> Title 1 </h1>
            <p> Content 1 </p>
            <h1> Title 2 </h1>
            <p> Content 2 </p>
        </>
    );
};

export default MyComponent;
```

Explanation

Without Fragments : The DOM contains additional div elements that serve no purpose other than grouping child elements.

With Fragments : The unnecessary div elements are removed, and only the h1 and p elements are rendered in the DOM.

Benefits

1. **Cleaner DOM :-** By removing unnecessary wrapper elements, the DOM remains cleaner and more readable.

2. **Simplified CSS :** Styling becomes simpler because there are fewer elements to target and override.

3. **Performance :** Reducing the number of elements can improve performance, especially in large applications with many components

3. **Consumer :** Component is used to access the context value. It can be used in any component that needs to access the shared data.

Code Analysis

```
Const renderStars = (rating) => {

// create an array of 10 filled star icons.
Const filledStars = Array.from ({length: 10}, (_, index) =>
( <StarRateIcon key={index} style={{color: '#19762'}} />

// create an array of empty star icons, the number of which
   is (10 - Rating)
const emptyStars = Array.from ({length: 10 - Rating}, (_, index) =>
( <StarRateIcon key={index} style={{color: '#e0e0e0'}} />

// combine the filled stars up to the given rating and the
   remaining empty stars.
   return [...filledStar (0, rating), ... emptyStars];
```

```
Const filledStars = Array.from({length: 10}, (_, index) =>
(<starRateIcon key={index} style={{color: '#19762'}} />
```

`Array.from({length:10})` creates an array with 10 elements

The second parameter of `Array.from` is a mapping function that fills each element with a starRateIcon Component.

Each starRateIcon is given by a key attribute based on the index and a style attribute to set its color to blue.

```
const emptyStars = Array.from({ length: 10 - Rating }, (_, index) => (
  <StarRateIcon key={index} style={{ color: '#e0e0e0' }} />
));
```

`Array.from({ length: 10 - rating })` creates an array with a number of elements equal to `10 - rating`.

Each element in this array is filled with a `StarRateIcon` component styled with a grey color.

object object.

```
return [ ...filledStars(0, Rating), ...emptyStars ];
```

combining two objects & put it in array