C:\vs code\GitHub\learn\WEB DEV\FRONT-END\5-React\new_learn\works\my_app\res.jsx

~~~
// start a react app : --[]
```
npm run start
```
// install all libraries from the package.json file :
```
npm install
```
// {for managing course content } : --[]

//  clone the main branch
```
git clone https://github.com/Yarob50/Tarmeez-React-Course.git
```
// git all branches list :
```
git branch -a
```

// create new react app :  (app_name) can not contain capital letters :----------[]
```
    npx create-react-app app_name;
```

// app.js : it's represented the root component :


// File Structure :----------[]
/*
    1. **index.html**:
    - This file is the entry point of your React application.
    - It contains the HTML structure of your web page.
    - Usually includes a `<div>` element with an id where React will render the application.

    2. **index.js**:
    - This file is the entry point of your React application where you bootstrap your React application.
    - It typically imports necessary dependencies and renders the root component of your application
        into the DOM.

    3. **App.js**:
    - This file defines the root component of your React application.
    - It's where you structure your application layout and manage the overall state and behavior.

    4. **Component files (e.g., Header.js, Footer.js, etc.)**:
    - These files contain individual components of your application.
    - Each component typically represents a reusable UI element or a logical part of your application.
    - They encapsulate HTML structure, styles, and behavior related to that specific UI element.

~~~

```
    5. **Stylesheets (e.g., styles.css, App.css)**:
    - These files contain stylesheets for your components.
    - They provide CSS rules for styling your components and ensuring a consistent look and feel
      across the application.

    6. **Other utility files (e.g., utils.js, constants.js)**:
    - These files contain utility functions, constants, or configurations used throughout your application.
    - They help keep your code organized and facilitate code reuse.

    7. **Package.json**:
    - This file contains metadata about your project and its dependencies.
    - It lists all the dependencies required for your project to run, along with their versions.
    - It also includes scripts for running various tasks such as starting the development server
      or building the production bundle.

    8. **Node_modules**:
    - This directory contains all the dependencies installed for your project.
    - It's managed by npm (Node Package Manager) and typically not version-controlled.

    9. **Webpack.config.js / Babel.config.js**:
    - These files contain configurations for bundling and transpiling your React code.
    - Webpack is a module bundler that bundles your JavaScript files and their dependencies into a single file for
      the browser.
    - Babel is a JavaScript compiler that transforms your modern JavaScript code (ES6/ES7) into a backward-compatible
    version for older browsers.

    10. **.gitignore**:
      - This file specifies intentionally untracked files that Git should ignore.
      - It usually includes directories like `node_modules` and files like `.DS_Store` that you don't want
      to include in version control.

    11 ** .public**:
      - static content

    This structure provides a foundation for organizing a React application, allowing for scalability, maintainability,
    and reusability of code. It separates concerns by breaking down the
    application into smaller, manageable components and provides a clear separation of HTML, CSS, and JavaScript logic.
*/


// App.js Main Structure  :----------[]
import React, { Component } from 'react';
import ReactDOM from 'react-dom/client';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(
<React.StrictMode>
```

```
  <App />
</React.StrictMode>
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
/*
Sure, let's break down the code `App.js`:----------[]

1. **Imports**:
- `import React from 'react';`: Imports the React library, which is required for building React components.
- `import ReactDOM from 'react-dom/client';`: Imports the `ReactDOM` module from the `react-dom` package.
It's used for rendering React components into the DOM.
- `import './index.css';`: Imports a CSS file for styling. This is typically used to apply global
 styles to the application.
- `import App from './App';`: Imports the `App` component from the file `App.js`. This is the main
 component of the application.
- `import reportWebVitals from './reportWebVitals';`: Imports a function called
 `reportWebVitals` from a file named `reportWebVitals.js`.

2. **Root Element**:
- `const root = ReactDOM.createRoot(document.getElementById('root'));`: Creates a root element
 using `ReactDOM.createRoot()`. This method is used to create a root for the new concurrent
  React mode.
- `root.render()`: Renders the `App` component into the root element.

3. **Strict Mode**:
- `<React.StrictMode>`: Wraps the `App` component with `React.StrictMode`. This is a development
 mode that helps identify potential problems in your code. It enables additional checks and warnings
  for potential issues.
- `</React.StrictMode>`: Closes the `React.StrictMode` component.

4. **Performance Measurement**:
- `reportWebVitals()`: Calls the `reportWebVitals` function. This function can be used to measure various
 performance metrics in the application, such as rendering performance, network performance, etc. You can pass
 a callback function to log the results or send them to an analytics endpoint.

*/
const reportWebVitals = onPerfEntry => {
if (onPerfEntry && onPerfEntry instanceof Function) {
  import('web-vitals').then(({ getCLS, getFID, getFCP, getLCP, getTTFB }) => {
    getCLS(onPerfEntry);
    getFID(onPerfEntry);
    getFCP(onPerfEntry);
    getLCP(onPerfEntry);
    getTTFB(onPerfEntry);
  });
}
```

```
};

export default reportWebVitals;

// ReportWebVitals.js Explain code : ----------[]

/*

1. **Function Definition**:
- `const reportWebVitals = onPerfEntry => {`: Defines a constant named `reportWebVitals` which is a
   function that takes a parameter `onPerfEntry`. This parameter is expected to be a callback function
   that will be invoked with performance data.

2. **Conditional Check**:
- `if (onPerfEntry && onPerfEntry instanceof Function) {`: Checks if `onPerfEntry` is provided and if
   it's a function. This ensures that the function is callable and has been provided before proceeding.

3. **Dynamic Import**:
- `import('web-vitals').then(...`: Dynamically imports the `web-vitals` module using the `import()`
   function. This is a dynamic import syntax introduced in ES6. It allows importing modules on demand.
- `({ getCLS, getFID, getFCP, getLCP, getTTFB }) => { ... }`: Destructures the imported module to
   extract specific functions `getCLS`, `getFID`, `getFCP`, `getLCP`, and `getTTFB`.

4. **Performance Metric Collection**:
- `getCLS(onPerfEntry);`, `getFID(onPerfEntry);`, `getFCP(onPerfEntry);`, `getLCP(onPerfEntry);`,
   `getTTFB(onPerfEntry);`: Calls each of the imported functions (`getCLS`, `getFID`, etc.) with
   `onPerfEntry` as an argument. These functions are responsible for collecting specific
   performance metrics like Cumulative Layout Shift (CLS), First Input Delay (FID), etc.

5. **Export**:
- `export default reportWebVitals;`: Exports the `reportWebVitals` function as the default export
   of this module. This allows other parts of the codebase to import and use this function.

Overall, this code dynamically imports the `web-vitals` module, retrieves specific
 performance metric functions from it,
and invokes them with the provided callback function (`onPerfEntry`).
It provides a generic way to collect and report
web performance metrics in a React application.

The functions `getCLS`, `getFID`, `getFCP`, `getLCP`, and `getTTFB` are part of a web
 performance API called [Web Vitals](https://web.dev/vitals/), which provides key metrics
 to help measure the performance and user experience of a website. Each of these functions
 captures a specific aspect of web performance. Here's an explanation of each:

### 1. **`getCLS(onPerfEntry)` - Cumulative Layout Shift (CLS)**
  - **CLS** measures **visual stability** and how much the content on the page shifts during loading.
  - A low CLS score means the elements on the page don't move around unexpectedly, providing
  a better user experience. High CLS often occurs when images, ads, or fonts load asynchronously
   and cause layout shifts.
  - **Good CLS score**: A score below 0.1 is considered good.

  **Example use case**: You visit a page, and suddenly an ad appears, causing text to shift down.
  CLS measures this unexpected shift.

### 2. **`getFID(onPerfEntry)` - First Input Delay (FID)**
  - **FID** measures the **time from when a user first interacts** with your page
  (e.g., clicking a button, tapping on a link) to the time when the browser is actually
   able to begin processing that interaction.
  - This metric is important for measuring **interactivity** and responsiveness.
  - **Good FID score**: A score below 100ms is considered good.

  **Example use case**: You click a button, but the page doesn't respond immediately
  due to JavaScript tasks or layout calculations. FID measures the delay between your
  click and the page responding.

### 3. **`getFCP(onPerfEntry)` - First Contentful Paint (FCP)**
  - **FCP** measures the time from when the page starts loading to the point when
  **any part of the page's content** is rendered on the screen. This can include text,
  images, or other DOM elements.
  - This metric is crucial for measuring **perceived load speed**, giving users a sense
   that the page is loading.
  - **Good FCP score**: A score below 1.8 seconds is considered good.

  **Example use case**: When you load a page, FCP is the moment when the first visible
   part of the webpage (such as text or an image) appears in the viewport.

### 4. **`getLCP(onPerfEntry)` - Largest Contentful Paint (LCP)**
  - **LCP** measures the **render time of the largest visible content element**
  (such as an image or large text block) within the viewport.
  - It's a critical metric for measuring how long it takes for the main content
  to become visible to the user, reflecting **perceived loading performance**.
  - **Good LCP score**: A score below 2.5 seconds is considered good.

  **Example use case**: On a blog page, the LCP might be an image or large
  text header. LCP measures when the largest visible content element is fully loaded.

### 5. **`getTTFB(onPerfEntry)` - Time to First Byte (TTFB)**
  - **TTFB** measures the time it takes for the browser to receive
   the **first byte of content** from the server after the user requests the page.
  - This metric is important for measuring **backend performance** and
  the responsiveness of the server.
  - **Good TTFB score**: A score below 200ms is considered good.

  **Example use case**: When you request a page, TTFB measures how long it takes the server
   to send the first byte of data to the browser after the request is made.

---


*/
```

```

// React linking index.html with index.js :----------[]
/*
When you run a React application using tools like Create React App,
a development server is launched that handles linking the JavaScript files with the
 `index.html` file automatically.

During development:

1. **Development Server**:
- When you run `npm start` or `yarn start`, Create React App starts a development server.
- This development server serves your React application and provides hot reloading,
allowing you to see changes in real-time as you develop.
- The development server takes care of linking the JavaScript files (typically named
   something like `main.js` or `bundle.js`) with the `index.html` file.

2. **Automatic Injection**:
- As you make changes to your React components and save your files, Create React App
automatically rebuilds your application and updates the browser.
- The development server injects the updated JavaScript code into the `index.html`
file, so you don't need to manually refresh the page to see your changes.

During production:

1. **Build Process**:
- When you build your React application for production using `npm run build` or `yarn build`,
Create React App generates optimized production-ready files.
- This includes a bundled JavaScript file containing your React components and logic.

2. **Injection Mechanism**:
- Create React App uses a build script to inject the bundled JavaScript file into the `index.html` file.
- It replaces a placeholder comment in the `index.html` file with a `<script>` tag linking to the bundled
 JavaScript file.
- This ensures that when you open the `index.html` file in a browser or deploy it to a server, the necessary
   JavaScript code is linked and executed properly.

In both development and production, Create React App takes care of linking the JavaScript files with the `index.html`
file, making it easier for you to focus on building your React components and features without worrying
 about the underlying setup and configuration.
*/
// hot Reload : ----------[]
/*
Auto reload (not refresh) the page when you save you work  be caution
 that logs errors wouldn't disappear from the console til you refresh the page
*/
// create new component  : ----------[]
/*
   1- new js file
   2- write a function that return the component structure
   3- export the function as default
```

~~~
  4- import the js module(new js fie )  in the target file (App.js)
*/
 // example :
  // content.js :
  ```
    export default function MyFirstComponent() {
      return (
        <div className="content">
          <h1>hello world</h1>
          <h2>ayoub majid</h2>
        </div>
      );
    }
  ```
  // App.js File :
    import Content from  './content'

// use a component  : ----------[]
  ```
    <compName></compName>
  ```
  //or : self-closing element
  ```
    <compName/>
  ```
// use js into xml structure : ----------[]
```
  {jsCode}
```
 // example 1 :
  ```
    export default function content() {
      const title="hello World";
      return (
        <div className="content">
          <h1>{title}</h1>
          <h2>ayoub majid</h2>
        </div>
      );
    }
  ```
 //example 2 :
  ```
    export default function content() {
      return (
        <div className="content">
          <h1>{contentObj.title}</h1>
          <h2>{contentObj.userFullName}</h2>
        </div>
      );
~~~

```
        }
        let  contentObj= {
        title:"Hello world",
        userFullName :"Ayoub Majd"
        };
        ```

// add attribute : ----------[]
    // add event :
    ```
        <element onEvent={functionName} ></element>
    ```
    //example :
    ```
        function btnClick(){
            console.log("You Clicked At me ")
        }
        <button onClick={btnClick} >Click Here</button>

    ```
    // add styling :
    /*
        you have to set the styling in camelCase format : (ex : backgroundColor)
    */
    ```
        <element style={{styling}} ></element>
    ```

    // example :
    ```
        const elemStyling={
            backgroundColor:"green",
            fontWeight:""
        }
        <header className="App-header" style={elemStyling}>
    ```
// add class to an element :
```
    <element className="cls1 cls2" ></element>
```
//add class to an element using a variable :
```
<element className={variableName} ></element>
```
//add class to an element using variables and string (Template Literals) :
 ```
<element className={` className ${variableName}`} ></element>

-- or  :
<element className={ 'className' + variableName} ></element>
```

~~~

```
// example :
    import "./myButton.css";

    export default function MyButton() {
        const { title, githubLink, class: buttonClass } = ManageByButton.myButtonInfo;

        return (
            <div className={`content ${buttonClass}`}>
                <h1>{title}</h1>
                <a href={githubLink}>my github account</a>
            </div>
        );
    }

    class ManageByButton {
        static myButtonInfo = {
            title: "Hello world",
            githubLink: "https://github.com",
            class: "contentButton",
        };

        static getCurrentDate() {
            return new Date().toString();
        }
    }

    const title = "Hello world";


    const myButtonInfo = {
        title: title,
        githubLink: "https://github.com",
        class :"contentButton"
    };



// add styling using a css file : ----------[]
/*
    1- create new css file :
    2- add  styling
    3- import css file in the component.js file: import "./FileName.css"
*/
  // example :
        // content.css
        ```
            .content {

~~~

~~~
                padding: 20px;
                margin-bottom: 10px 0;

            }
            .content button {
                padding: 10px 20px;
                font-size: 1em;
                border-radius: 12px;
                border: none;
                background-color: aqua;
            }
        ```
        // content.js
        ```
            import "./content.css";

            export default function content() {
                return (
                    <div className="content">
                        <h1>Hello world</h1>
                        <button onClick={btnClick}>Click Here</button>
                    </div>
                );
            }
        ```

// Scoped Styling (CSS Modules or Inline Styles)
/*
    CSS Modules: When using CSS Modules, the styles are scoped to the component. This means that the
    class names are locally scoped, and there is no risk of them affecting other parts of the
    application.

    Inline Styles: Applying styles directly to elements using the style attribute ensures
    that those styles are specific to that element only.
*/
// example :
    // MyButton.module.css
    .button {
        background-color: blue;
        color: white;
    }

    // MyButton.jsx
    import styles from './MyButton.module.css';

    export default function MyButton() {
        return <button className={styles.button}>Click Me</button>;
    }


~~~

```
//  project structure :
/*
/your-app-name
│
├── public
│   ├── index.html
│   ├── manifest.json
│   ├── robots.txt
│   └── (any other static assets like icons, images, etc.)
│
├── src
│   ├── assets
│   │   ├── images
│   │   ├── styles
│   │   └── (any other static assets like fonts, etc.)
│   │
│   ├── components
│   │   ├── (ComponentName)
│   │   │   ├── (ComponentName).jsx
│   │   │   ├── (ComponentName).css
│   │   │   └── (ComponentName).test.js
│   │   └── (more components...)
│   │
│   ├── hooks
│   │   └── useCustomHook.js
│   │
│   ├── pages
│   │   ├── Home
│   │   │   ├── Home.jsx
│   │   │   ├── Home.css
│   │   │   └── Home.test.js
│   │   └── (more pages...)
│   │
│   ├── services
│   │   └── api.js
│   │
│   ├── utils
│   │   └── helpers.js
│   │
│   ├── App.js
│   ├── index.js
│   └── index.css
│
├── .gitignore
├── package.json
├── README.md
└── (other config files like .eslintrc, .prettierrc, etc.)


*/
```

~~~jsx
// props : --[]
/*
parameters that you pass when you create or call a component
*/
// in App.js :
```
  <ComponentName paremeter1="value" paremeter2="value" paremeterN="value" />
```
// in ComponentName.js
```
  export default function({paremeter1,paremeter2,paremeterN}){
  return(
    // code

  );
  }
```

// example :
// App.js :

  import "./App.css";
  import Content from "./content";
  function App() {
    return (
      <div className="App">
        <Content name="ayoub" email="ayoub@gmail.com" phone="05332"></Content>
        <Content email="nasim@gmail.com" phone="3322"></Content>
        <Content name="majid" email="majid@gmail.com" phone="2544"></Content>
      </div>
    );
  }
  export default App;

//Content.js  :

  import './content.css'
  export default function ({name,email,phone}) {
    return (
        <div className='content'>
          <h1>{name}</h1>
          <h3>{email}</h3>
          <h4>{phone}</h4>
        </div>
    );
  }


// set default props Values :
  ComponentName.defaultProps = {
~~~

~~~
    prop1: "Default prop1",
    prop2: "default prop2",
  };

  // or:
  export default function ComponentName({prop1="default prop1",prop2="default prop2"}){


  }

// example :
  import "./Article .css";
  export default function Article({ name, email, birthDate }) {
    console.log(name, email, birthDate);

    return (
      <article className="articleComponentClass">
        <h2>{name}</h2>
        <h2>{email}</h2>
        <h2>{birthDate}</h2>
      </article>
    );
  }
  Article.defaultProps = {
    name: "Default Name",
    email: "default@example.com",
    birthDate: "01-01-1970",
  };

// add xml content :
// in App.js :
```
  <ComponentName paremeter1="value" paremeter2="value" paremeterN="value" >
   // xml content
  </ComponentName>
```
// in ComponentName.js :  the children key contain value the xml content passed to the componentName
```
  export default function({paremeter1,paremeter2,paremeterN,children}){
  return(
    // code
    {children}

  );
  }
```

// example :
// App.js :

~~~

~~~
    import "./App.css";
    import Content from "./content";
    function App() {
        return (
            <div className="App">
                <Content name="ayoub" email="ayoub@gmail.com" phone="05332" />

                <Content email="nasim@gmail.com" phone="3322" >
                    <h1 style={{ backgroundColor: "black", color: "white", padding: "10px" }}>nice to meet you</h1>
                </Content>
                <Content name="majid" email="majid@gmail.com" phone="2544"></Content>
            </div>
        );
    }
    export default App;

// Content.js :

    import "./content.css";
    import { Component } from 'react';
export default function ({ name = "no name", email = "no email", phone = "no phone", children }) {
        return (
            <div className="content">
                <h1>{name}</h1>
                <h3>{email}</h3>
                <h4>{phone}</h4>
                {children}
            </div>
        );
    }


// convert from string to jsx :
```
    export default function ComponentName(){
    return (
        <div dangerouslySetInnerHTML={{ __html:content }} />
    );
    }
```
/*
dangerouslySetInnerHTML: This is used to render the HTML content inside the children. Be careful with this
 approach to avoid injecting any untrusted content, as it can lead to security issues.
*/
// example :
    import React from "react";
    import Button from "./Button";

    export default function Tags() {

~~~

```

  const tagText ="javaScript"
  const  children= "<i class='fab fa-js'></i><img src='path/to/javascript.png' alt='JavaScript' />",


  return (
    <div className="tags">
      <Button text={tagText}>
      <div dangerouslySetInnerHTML={{ __html:children }} />
      </Button>




    </div>
  );
  }

// conditional Rendering :--[]

// example using ternary operator ? :
return (
  <div className="App">
    {showChildren ? (
      <Content email="nasim@gmail.com" phone="3322">
        <h1 style={{ backgroundColor: "black", color: "white", padding: "10px" }}>nice to meet you</h1>
      </Content>
    ) : null}
  </div>
);

// example using  if statement :
  import "./App.css";
  import Content from "./content";
  function App() {
    let showChildren = true;

    return (
      <div className="App">
        <Content name="ayoub" email="ayoub@gmail.com" phone="05332" />
        <Content name="majid" email="majid@gmail.com" phone="2544" />
        <LoadChildren childrenStatus={showChildren} />
      </div>
    );
  }
  function LoadChildren({ childrenStatus = true }) {
    console.log(Boolean(false));
    if (childrenStatus) {
      return (
        <Content email="nasim@gmail.com" phone="3322">
```

```
                    <h1 style={{ backgroundColor: "black", color: "white", padding: "10px" }}>nice to meet you</h1>
                </Content>
            );
        } else {
            return null;
        }
    }
    export default App;

// stat :
/*
    state in React allows components to manage and maintain their internal data, enabling
    them to be dynamic and interactive. Understanding how to effectively use state is
    crucial for building robust and scalable React applications.
*/

// without using stat :

    import "./MyButton.css";

    export default function MyButton() {
        let name = "ayoub";
        return (
            <div className="MyButtonComponentClass">
                <button
                    onClick={() => {
                        name = parseInt(Math.random() * 10) +"-amina" ;
                        console.log(name);
                    }}
                >
                    My button
                </button>
                <h1>{name}</h1>
            </div>
        );
    }

/*
    The UI did not change when you modified the `name` variable because React does not
    automatically re-render the component when you directly modify local variables. Here's a
    detailed explanation of why this happens and how you can fix it:

    ### Why the UI Did Not Update

    1. Local Variables and React Rendering:
    - React's rendering cycle is based on its state and props. Local variables (like `name` in your example)
        are not part of React's state management system. React does not track or react to changes in these variables.

    2. State Management:
    - To trigger a re-render in React, you need to use state. Local variables are not reactive, so changing
```

~~~jsx
    them will not cause the component to re-render. React only re-renders components when the state or props
        change.

    3. Direct Variable Mutation:
    - The `name` variable is changed on a button click, but since it's not part of React's state, React
    is unaware of this change. The component will only re-render if there is a state change.

    ### How to Fix It

    To ensure that changes to `name` trigger a re-render, you should use React's `useState` hook. The `useState`
    hook allows you to manage state in functional components, and updating the state will automatically trigger
    a re-render.

*/


// useStat hook :

    import { useState } from "react";
    export default function Btn({ title }) {
        if (!title) title = "user title";
        const [name, setName] = useState(title);

        function changeName() {
            if (name == title) setName("Ayoub");
            else setName("Majid");
        }
        return (
            <div className="btnContainer">
                <button onClick={changeName}>Click Me</button>
                <h1>{name}</h1>
            </div>
        );
    }

    /*
        In this code, the `useState` hook plays a crucial role in managing state
        within the functional component
        `Btn`. Let's break down its role and how it triggers re-renders
        when the state of `title` changes:
    */

    //* 1. **Initializing State**:
    ```javascript
    const [name, setName] = useState(title);
    ```
    /*
            The `useState` hook is used to declare a state variable `name` and its corresponding
            setter function `setName`. The initial value of the `name` state variable is set to
            the value of the `title` prop passed to the component.
~~~

~~~
*/

//* 2. **Rendering Initial UI**:
```javascript
<h1>{name}</h1>
```
/*
        The `<h1>` element in the JSX renders the current value of the `name` state variable. Initially,
        it displays the value of the `title` prop passed to the component.
*/

//* 3. **Updating State on Button Click**:
```javascript
function changeName() {
    if (name === title) {
        setName("Ayoub");
    } else {
        setName("Majid");
    }
}
```
/*
    The `changeName` function is called when the button is clicked. It checks if
    the current value of the `name` state variable is equal to the `title` prop.
    If they are equal, it updates the `name` state variable to `"Ayoub"`, otherwise,
    it updates it to `"Majid"`.
*/

//* 4. **Re-rendering on State Change**:
/*
    When the `setName` function is called to update the state variable `name`,
    React re-renders the component. This is because React detects the change
    in state and automatically triggers a re-render of the component to reflect
    the updated state. Consequently, the `<h1>` element displaying the value of
    the `name` state variable is re-evaluated and updated with the new value.
*/

//* 5. **Re-rendered UI**:
/*
After the state is updated, React re-renders the component with the new value of the `name`
state variable. As a result, the UI is updated to display the new name ("Ayoub" or "Majid")
depending on the current state.
*/
/*
    In summary, the `useState` hook manages the state of the `name` variable in the `Btn` component.
    When the state changes (due to button click), React automatically triggers a re-render of the
    component to reflect the updated state, resulting in the UI being updated accordingly.
    This mechanism enables React to efficiently handle state changes and keep the UI in sync
    with the underlying data.
*/
~~~

```

// how does react detect  the change of state : -- []
    /*
        React detects changes in state using a mechanism called reconciliation.
        When a component's state changes, React compares the previous state with
        the new state. It then determines which parts of the component's UI need
        to be updated to reflect the changes in state.
    */

    //* 1. **Virtual DOM**:
    /*
        React maintains a virtual representation of the DOM (Document Object
        Model), known as the Virtual DOM. This virtual representation mirrors
        the actual DOM but is lightweight and exists entirely in memory.
    */

    //* 2. **Rendering**:
    /*
        When a component's state changes, React re-renders the component and
        updates the Virtual DOM accordingly. React compares the new Virtual DOM
        with the previous Virtual DOM to identify the differences.
    */

    //* 3. **Differential Algorithm**:
    /*
        React employs a highly optimized algorithm called the Reconciliation
        Algorithm to perform a "diffing" process between the new Virtual DOM
        and the previous Virtual DOM. This process involves efficiently
        identifying the minimal set of changes needed to update the actual DOM.
    */

    //* 4. **Identifying Changes**:
    /*
        During the diffing process, React identifies which elements in the
        Virtual DOM have changed between the previous and new states. It compares
        elements based on their type, attributes, and content.
    */

    //* 5. **Batching Updates**:
    /*
        React batches multiple state updates into a single re-render operation
        for performance optimization. This means that if multiple state changes
        occur within the same event handler or lifecycle method, React combines
        them into a single update, reducing unnecessary re-renders.
    */

    //* 6. **Updating the DOM**:
    /*
        Once React identifies the changes needed to update the actual DOM, it
        applies these changes in a batched manner to minimize the number of DOM
```

~~~
        manipulations. This process ensures that the UI is efficiently updated
        to reflect the changes in state.
    */

    /*
        In summary, React detects changes in state by comparing the Virtual DOM
        before and after a component's re-rendering. It uses an efficient
        reconciliation algorithm to identify the minimal set of changes required
        to update the actual DOM, ensuring optimal performance and UI responsiveness.
    */


// example with input :
```
    import "./Inp.css";
    import { useState } from "react";
    export default function Inp() {
        let [content, setContent] = useState("");
        function changeName(e) {
            setContent(e.target.value);
        }
        return (
            <div className="inputContainer">
                <input placeholder="enter your name" onChange={changeName} />
                <h2>{content}</h2>
            </div>
        );
    }
```

// From submission : --[]
// Method 1 :
```
    import "./Frm.css";
    import { useState } from "react";
    export default function Frm() {
        let [name, setName] = useState("");
        let [age, setAge] = useState(0);
        age = Number(age);
        return (
            <form
                onSubmit={(event) => {
                    event.preventDefault();
                    console.log("name :", name);
                    console.log("age :", age);
                }}
            >
                <label for="name">name:</label>
                <input
                    id="name"
~~~

~~~
                    onChange={(event) => {
                        setName(event.target.value);
                    }}
                />
                <label for="age">age:</label>
                <input
                    id="age"
                    type="number"
                    onChange={(event) => {
                        setAge(event.target.value);
                    }}
                />

                <button>submit</button>
            </form>
        );
    }

```
// Method 2 :
```
    import "./Frm.css";
    import { useState } from "react";
    export default function Frm() {
        let [formInfo, setFromInfo] = useState({ name: "", age: 0 });

        function setValue(event) {
                setFromInfo({...formInfo,[event.target.id]:event.target.value});
        }
        return (
            <form
                onSubmit={(event) => {
                    event.preventDefault();
                    console.log(formInfo);
                }}
            >
                <label >name:</label>
                <input id="name" onChange={setValue} />
                <label >age:</label>
                <input id="age" type="number" onChange={setValue} />

                <button>submit</button>
            </form>
        );
    }
```
// advance Example  Form Stat:
```
    import "./Frm.css";
    import { useState } from "react";
~~~

```

  export default function Frm() {
    const [formInfo, setFormInfo] = useState({ name: "", age: 0, generalInfo: "", isStudent: false, userCountry: "MR", gender: "" });

    function handleChange(event) {
      const { id, type, value, checked } = event.target;
      setFormInfo((prevState) => ({
        ...prevState,
        [id]: type === "checkbox" ? checked : value,
      }));
    }

    return (
      <form
        onSubmit={(event) => {
          event.preventDefault();
          console.log(formInfo);
        }}
      >
        <label htmlFor="name">Name:</label>
        <input type="text" id="name" onChange={handleChange} />

        <label htmlFor="age">Age:</label>
        <input id="age" type="number" onChange={handleChange} />

        <label htmlFor="generalInfo">General Info:</label>
        <textarea id="generalInfo" onChange={handleChange} />

        <div className="checkedBoxes">
          <input type="checkbox" id="isStudent" onChange={handleChange} />
          <label htmlFor="isStudent">Is Student</label>
        </div>

        <select id="userCountry" onChange={handleChange}>
          <option>MR</option>
          <option>KSA</option>
          <option>UAI</option>
          <option>US</option>
        </select>

        <label>Gender:</label>

        <div>
          <label>Male</label>
          <input type="radio" id="gender" name="gender" value="Male" onChange={handleChange} checked={formInfo.gender === "Male"} />
        </div>

        <div>
```

~~~
                    <label>Female</label>
                    <input type="radio" name="Female" id="gender" value="Female" onChange={handleChange} checked={formInfo.gender === "Female"} />
                </div>

                <button type="submit">Submit</button>
            </form>
        );
    }

```
// the complete setForm function :
    function setFormInput(event) {
        let { id, value, checked, type, name } = event.target;

        value = id === "age" ? parseInt(value, 10) : value;

        setFormInputs({ ...formInputs, [type === "radio" ? name : id]: type === "checkbox" ? checked : value });
    }

// Array Stat Example :
import "./App.css";
import { useState } from "react";

function App() {
    //  const devices=["Iphone","Mac","Samsung","Windows"];

    let [devices, setDevices] = useState([]);
    let [deviceInput, setDeviceInput] = useState("");

    const devicesList = devices.map((device, index) => {
        return (
            <div key={index} style={{ display: "flex", gap: "20px", width: "60%", minWidth: "250px", alignItems: "center", border: "1px solid black", justifyContent: "space-between", padding: "7px 10px", borderRadius: "10px" }}>
                <li>{device}</li>

                <div style={{ display: "flex", gap: "10px" }}>
                    <button
                        onClick={() => {
                            deleteDevice(index);
                        }}
                    >
                        Delete
                    </button>

                    <button
                        onClick={() => {
                            updateDevice(index);
                        }}
                    >
~~~

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

  function deleteDevice(deviceIndex) {
    let newDeviceList = [...devices];
    newDeviceList.splice(deviceIndex, 1);
    setDevices(newDeviceList);
  }
  function updateDevice(deviceIndex) {
    let newDevices = [...devices];

    newDevices[deviceIndex] = prompt("enter the new Name : ", newDevices[deviceIndex]);

    if (newDevices[deviceIndex]) setDevices(newDevices);
  }
  function changeStat(event) {
    setDeviceInput(event.target.value);
  }

  function addDevice() {
    setDevices((prevState) => [...prevState, deviceInput]);
    setDeviceInput("");
  }

  return (
    <div className="App">
      <div className="addDeviceContainer" style={{ margin: "30px" }}>
        <input placeholder="add new device" style={{ marginRight: "6px" }} onChange={changeStat} value={deviceInput} />
        <button onClick={addDevice}>Add</button>
      </div>

      <ul style={{ marginTop: "20px", display: "flex", flexDirection: "column", gap: "10px", alignItems: "center" }}>{devicesList}</ul>
    </div>
  );
}

// oop version : -----[]
import style from "./Test.module.css";

import { useState } from "react";
export default function TesT() {
  const [newDeviceFormInfo, setNewDeviceFormInfo] = useState({ deviceName: "", isDisabled: false });

  const [devices, setDevices] = useState([]);

```

```
class utile {
  static handelChangeFormInputs(event, setStatForm) {
    let { id, value, checked, type, name } = event.target;

    value = id === "age" ? parseInt(value, 10) : value;

    setStatForm((formInputs) => {
      return { ...formInputs, [type === "radio" ? name : id]: type === "checkbox" ? checked : value };
    });
  }
}
class clsHandelAddNewDevice {
  static ChangeFormInputs(event) {
    utile.handelChangeFormInputs(event, setNewDeviceFormInfo);
  }
  static #changeSubmitButtonStat(newStat) {
    setNewDeviceFormInfo((prevFormInfo) => {
      return { ...prevFormInfo, isDisabled: newStat };
    });
  }
  static #clearAddFormStat() {
    setNewDeviceFormInfo((prevFormInfo) => {
      return { deviceName: "", isDisabled: false };
    });
  }
  static handelFormSubmit(event) {
    event.preventDefault();
    // true => disabled = true
    clsHandelAddNewDevice.#changeSubmitButtonStat(true);

    clsHandelCurdDevices.addDevice(newDeviceFormInfo["deviceName"]);

    clsHandelAddNewDevice.#clearAddFormStat();
  }
}
class clsHandelCurdDevices {
  static addDevice(deviceName) {
    setDevices((prevDevicesInfo) => [...prevDevicesInfo, deviceName]);
  }
  static editDevice(deviceIndex) {
    let targetDeviceName = devices[deviceIndex];

    let newName = prompt(`Enter the  new name   [index  ${deviceIndex}] : `, targetDeviceName);

    if (newName && newName !== targetDeviceName) {
      let tempDevices = [...devices];
      tempDevices[deviceIndex] = newName;

      setDevices(tempDevices);
    }
```

```jsx
    }
    static deleteDevice(deviceIndex) {
      let newDeviceList = [...devices];
      newDeviceList.splice(deviceIndex, 1);
      setDevices(newDeviceList);
    }
  }

  return (
    <>
      <div className="header">
        <h1>Devices</h1>
        <form id="addDeviceForm" onSubmit={clsHandelAddNewDevice.handelFormSubmit}>
          <input type="text" id="deviceName" placeholder="Enter the name of the device" value={newDeviceFormInfo["deviceName"]} onChange={clsHandelAddNewDevice.ChangeFormInputs} required />
          <button disabled={newDeviceFormInfo["isDisabled"]}>Add device</button>
        </form>
      </div>
      <div className="devicesContainer">
        {devices.map((deviceName, index) => {
          return (
            <div key={index} className="deviceItem">
              <h1 className="deviceName">
                {" "}
                {index}-{deviceName}
              </h1>
              <div className="controlSection">
                <button onClick={() => clsHandelCurdDevices.editDevice(index)}>Edit Device</button>
                <button onClick={() => clsHandelCurdDevices.deleteDevice(index)}>delete Device</button>
              </div>
            </div>
          );
        })}
      </div>
    </>
  );
}


//  updating states several Times :--[]
function App() {
  let [count, setCount] = useState(0);

  function increaseCounter() {
    setCount(count + 1);
    setCount(count+ 1);
  }


  return (
```

~~~
        <div className="App">
            <h1 >The count is : <span onClick={increaseCounter}  >{count}</span> </h1>
        </div>
    );
}
/*
    The reason the counter doesn't increment by 2 each time you click is because
     the `setCount` function in React's `useState` hook doesn't immediately update
     the state. Instead, it schedules  an update, and React
    may batch multiple `setState` calls together for performance reasons.

    In your `increaseCounter` function, you're calling `setCount`
    twice with the same value of `count`. Both calls essentially set the same
    value. React sees these two calls and batches them together
    into a single update, resulting in only one increment by 1.

    If you want to increase the count by 2 each time you click,
    you should use the functional update form of `setCount`, which
    takes the current state as an argument and returns the
    new state. This ensures that the state updates are applied
    one after the other:
*/
    ```javascript
    function increaseCounter() {
        setCount(prevCount => prevCount + 1);
        setCount(prevCount => prevCount + 1);
    }
    ```
/*
    With this change, the count will indeed increase by 2
    each time you click. Each call to  `setCount` is now using
    the previous state to calculate the new state, so you're  effectively
    incrementing by 1 twice.
*/
/*
    In JavaScript, code execution generally follows a single-threaded event loop model.
    This means that JavaScript code is executed in a sequence, and only one operation
    can be processed at a time. However, React's reconciliation process and state updates
    are asynchronous operations, meaning they don't happen immediately when you call
    `setState` or `useState` setter functions.

    When you call `setState` or `useState` setters in React, React schedules the state
    updates for processing. React then decides when to apply these updates based on
    its internal mechanism, which aims to optimize performance by batching updates
    and minimizing unnecessary re-renders.

    While you can't precisely determine whether two state updates will be batched
    together or not in a given scenario, you can rely on React's behavior that it
    will batch updates when possible to improve performance.

~~~

```
    Here's a simplified explanation of how React typically handles state updates:

    1. When you call `setState` or `useState` setter functions, React records the
    state update requests.
    2. React batches multiple state updates that occur within the same event loop iteration.
    3. Before the next repaint, React reconciles the state updates and performs
    a single re-render of the component.

    In your specific case, calling `setCount` twice within the same event loop iteration
    is likely to result in React batching these updates together,
    leading to a single re-render of the component with the combined state update.
    However, React's exact behavior may vary depending on factors such as the React version,
        the environment (development vs. production), and the complexity of your component tree.

    While you cannot directly observe the batching behavior of React's state updates
    , you can rely on React's efficient handling of state updates to optimize performance
    in your application.
*/

// Prop Drilling : is pass props From a parents to a hierarchal children List
// useContext :
/*
    useContext is a React Hook that allows functional components to consume values from the Context API.
    Context provides a way to pass data through the component tree without having to pass props down
    manually at every level.
*/
// How useContext Works:
/*
    Create a Context: First, you need to create a context using the React.createContext()
    function. This creates a new context object.

    Provide the Context: You then provide the context to the component tree using a Context.Provider
    component. This component wraps the part of the tree where you want to make the context available.

    Consume the Context: Finally, you consume the context value in any descendant component using
    the useContext hook. This hook takes the context object as an argument and returns
    its current value.
*/
// steps :
/*
    1- define the provider  :
    2- define the consumer  :
    3- value to pass :
 */

    // Main Features of useContext:
 /*
    Simplicity: useContext provides a simple way to consume context values in
    functional components without the need for render props or higher-order components.

    Avoids Prop Drilling: It helps in avoiding prop drilling by allowing components
     to access context values directly, no matter how deeply nested they are in the component tree.

    Dynamic Context Updates: Components consuming context with useContext will
     re-render whenever the context value changes.

    Performance Optimization: React optimizes the context value retrieval with useContext,
     ensuring that components only re-render when necessary based on changes to the context value.

    Multiple Contexts: You can consume multiple contexts within a single component by
    calling useContext multiple times with different context objects.

    Static Type Checking: useContext can be easily used with static type checking libraries
     like TypeScript for type-safe context consumption.

    In summary, useContext is a powerful tool in React for managing state and sharing data
    across components in a more concise and efficient way compared to traditional prop drilling methods.
*/
//  Example  1: --[]

// --------- App.js ------------- :
    import "./App.css";
    import Form from "./LoanForm/LoanForm";



    export default function App() {




        return (
            <div className="App">
                    <Form />

            </div>
        );
    }

 // ------------- LoanFormInputsContext.js ------------- :
    import { createContext } from "react";
    export let LoanInputsContext = createContext({
        type: "text",
        value: "",
        id: "", handelChange: null,
        placeholder: "",
        IsRequired: true }
        );


```

```
// --------- LoanForm.js ------------- :
    import "./LoanForm.css";
    import { useState } from "react";
    import Alter from "../Alter/Alter";
    import Input from "../Input/Input";
    import { LoanInputsContext } from "../contexts/LoanFormInputsContext";

    export default function Form() {
        let initialFormInfo = {
            name: "",
            phone: "",
            age: "",
            salary: "Less Than 500$",
            isEmployee: false,
            isSubmitted: false,
        };
        let initialErrors = {
            isValidPhone: true,
            isValidAge: true,
        };

        let [formInfo, setFormInfo] = useState(initialFormInfo);
        let [errors, setErrors] = useState(initialErrors);
        document.addEventListener("click", (event) => {
            if (formInfo.isSubmitted && !event.target.classList.contains("alterDivText")) {
                setFormInfo((prevState) => ({
                    ...prevState,
                    isSubmitted: false,
                }));
            }
        });

        function changeFormInfo(event) {
            const { id, type, value, checked } = event.target;
            setFormInfo((prevState) => ({
                ...prevState,
                [id]: type === "checkbox" ? checked : value,
            }));
        }
        function handelFormSubmission(event) {
            event.preventDefault();
            setFormInfo((prevState) => ({
                ...prevState,
                isSubmitted: true,
            }));

            const isValidPhone = formInfo.phone.length >= 10 && formInfo.phone.length <= 12;
            const isValidAge = formInfo.age >= 18 && formInfo.age <= 100;

            setErrors({
```

```
        isValidPhone,
        isValidAge,
      });
    }

    function GeneraleAlterMsg() {
      return formInfo.isSubmitted && <>{errors.isValidAge && errors.isValidPhone ?
        <Alter msg="The Form Has Been submitted Successfully" /> : !errors.isValidPhone ?
         <Alter msg="Phone Number Format is incorrect" msgColor="red" />
         : <Alter msg="Age is Not allowed" msgColor="red" />}</>;
    }

    return (
      <>
        <form onSubmit={handelFormSubmission}>
          <div className="header">
            <h2>Requesting a Loan</h2>
            <hr />
          </div>

          <div className="nameContainer inputContainer">
            <LoanInputsContext.Provider value={{ value: formInfo.name, id: "name",
            handelChange: changeFormInfo, placeholder: "Enter Your Name" }}>
              <Input />
            </LoanInputsContext.Provider>
          </div>
          <div className="phoneContainer inputContainer">
            <LoanInputsContext.Provider value={{ value: formInfo.phone, id: "phone",
            handelChange: changeFormInfo, placeholder: "Enter Your Phone " }}>
              <Input />
            </LoanInputsContext.Provider>
          </div>

          <div className="ageContainer inputContainer">
            <LoanInputsContext.Provider value={{ type: "number", value: formInfo.age, id: "age",
            handelChange: changeFormInfo, placeholder: "Enter Your Age " }}>
              <Input />
            </LoanInputsContext.Provider>
          </div>

          <input type="submit" />
        </form>

        {GeneraleAlterMsg()}
      </>
    );
  }

// --------- Input.js ------------- :
  import "./Input.css";

  import { useContext } from "react";

  import { LoanInputsContext } from "../contexts/LoanFormInputsContext";
  import { UserContext } from "../contexts/UserContext";

  export default function Input() {
    const inputContext = useContext(LoanInputsContext);
    const userContextInfo = useContext(UserContext);

    return (
      <div className="inputContent">
        <h1>The User {userContextInfo.username}</h1>
        <h2>This is the header of the Component</h2>
        <input type={inputContext.type} placeholder={inputContext.placeholder}
        id={inputContext.id} value={inputContext.value} onChange={inputContext.handelChange}
         required={inputContext.IsRequired} />
      </div>
    );
  }

// --------- Alter.js ------------- :
  import "./Alter.css";
  import { useContext } from "react";
  import { UserContext } from "../contexts/UserContext";

  export default function Alter({ msg = "Message To Show", msgColor = "green", children }) {
  let userData = useContext(UserContext);
  return (
    <div className="alterDiv">
    <h2 style={{ color: msgColor }} className="alterDivText">
      {msg}
      {msgColor === "green" && (
      <h3>Welcome Mes {userData.username}</h3>
      )}
    </h2>
    {children}
    </div>
  );
  }


// Example 2 :  -- []

// --------- context.js ------------- :
import { createContext } from "react";
export let imgSizeContext=createContext(0);
export let  placeContext=createContext({})

// --------- App.js ------------- :
```

```
import { useState } from 'react';
import { places } from './data.js';
import { getImageUrl } from './utils.js';
import {imgSizeContext,placeContext} from "./Context.js"

export default function App() {
const [isLarge, setIsLarge] = useState(false);
const imageSize = isLarge ? 150 : 100;
return (
  <>
  <label>
    <input
    type="checkbox"
    checked={isLarge}
    onChange={e => {
      setIsLarge(e.target.checked);
    }}
    />
    Use large images
  </label>
  <hr />
  <imgSizeContext.Provider value={imageSize}>
  <List />
  </imgSizeContext.Provider>

  </>
)
}

  function List() {
  const listItems =
    places.map(place =>
    <li key={place.id}>
    <placeContext.Provider value={place}>
    <Place />
    </placeContext.Provider>
    </li>
  );
  return (
    <ul>{listItems}</ul>
      );

  }

  import { useContext } from "react";
  function Place() {
  let place=useContext(placeContext)
  return (
    <>
    <PlaceImage
```

```
      />
      <p>
        <b>{place.name}</b>
        {': ' + place.description}
      </p>
      </>
    );
    }

    function PlaceImage() {
    let imgSize=useContext(imgSizeContext);
      let place=useContext(placeContext)
    return (
      <img
      src={getImageUrl(place)}
      alt={place.name}
      width={imgSize}
      height={imgSize}
      />
    );
    }

// --------- data.js ------------- :
  export const places = [{
    id: 0,
    name: 'Bo-Kaap in Cape Town, South Africa',
    description: 'The tradition of choosing bright colors for houses began in the late 20th century.',
    imageId: 'K9HVAGH'
  }, {
    id: 1,
    name: 'Rainbow Village in Taichung, Taiwan',
    description: 'To save the houses from demolition, Huang Yung-Fu, a local resident, painted all 1,200 of them in 1924.',
    imageId: '9EAYZrt'
  }, {
    id: 2,
    name: 'Macromural de Pachuca, Mexico',
    description: 'One of the largest murals in the world covering homes in a hillside neighborhood.',
    imageId: 'DgXHVwu'
  }, {
    id: 3,
    name: 'Selarón Staircase in Rio de Janeiro, Brazil',
    description: 'This landmark was created by Jorge Selarón, a Chilean-born artist, as a "tribute to the Brazilian people."',
    imageId: 'aeO3rpI'
  }, {
    id: 4,
    name: 'Burano, Italy',
    description: 'The houses are painted following a specific color system dating back to 16th century.',
    imageId: 'kxsph5C'
  }, {
    id: 5,
    name: 'Chefchaouen, Marocco',
    description: 'There are a few theories on why the houses are painted blue, including that the color repels mosquitos or that it symbolizes sky and heaven.',
    imageId: 'rTqKo46'
  }, {
    id: 6,
    name: 'Gamcheon Culture Village in Busan, South Korea',
    description: 'In 2009, the village was converted into a cultural hub by painting the houses and featuring exhibitions and art installations.',
    imageId: 'ZfQOOzf'
  }];

  // --------- utils.js ------------- :
  export function getImageUrl(place) {
    return (
    'https://i.imgur.com/' +
    place.imageId +
    'l.jpg'
    );
  }

//* context management :
/*
  In React, **context** is used to share values across components without having
  to pass props manually through every level of the component tree.
  However, one of the potential downsides of using context
  is that components that consume the context will re-render whenever
  the value in the context changes.

  Here’s a detailed explanation of how and why this happens,
  and how it can lead to unnecessary re-renders:
*/
//### How Context Works with Re-rendering:
/*
  When you use React's `Context.Provider`, it provides a value to all
  components that consume it. Any time that the value provided by the
  `Provider` changes, all components that are consuming this context
  will re-render, even if the part of the value they rely on hasn’t changed.
*/

//* Example:
/*
  - In this example, both `ChildComponentA` and `ChildComponentB` are consuming
    `MyContext`

  - If `state` changes (e.g., `setState({ value1: 3, value2: 2 })`), **both**
   `ChildComponentA` and `ChildComponentB` will re-render.

  - Even if `ChildComponentB` is only using `value2` (which hasn’t changed),
    it will still re-render because the context object has changed.
```

```

  - This is considered an **unnecessary re-render** for `ChildComponentB`.
*/
  const MyContext = React.createContext();

  function ParentComponent() {
  const [state, setState] = useState({ value1: 1, value2: 2 });

  return (
    <MyContext.Provider value={state}>
    <ChildComponentA />
    <ChildComponentB />
    </MyContext.Provider>
  );
  }

  function ChildComponentA() {
  const context = useContext(MyContext);
  return <div>{context.value1}</div>;  // Only uses value1
  }

  function ChildComponentB() {
  const context = useContext(MyContext);
  return <div>{context.value2}</div>;  // Only uses value2
  }

//###  Why Unnecessary Re-renders Can Be a Problem:
/*
  1. Performance Impact: When components re-render unnecessarily,
    it can slow down the app, especially if the app is large, or if the
    component tree is deep and complex.

  2. **State Synchronization**: If child components perform complex
    calculations or side effects on re-render, unnecessary re-renders
    might lead to inefficiency, causing redundant work.

  3. **Component Bloat**: If many components are consuming the same
    context, it becomes harder to optimize re-renders, especially
    when only a small part of the context changes.

*/
// 1. **Memoizing Context Value**:
/*
  - **Problem**: Passing a new object or function as context value on every
            render causes all consuming components to re-render.

  - **Solution**: Use `useMemo` to memoize the context value so that
             it only changes when necessary.
*/
//*    Example:
```

```
/*
   **Benefit**: With `useMemo`, the context value is only recalculated
   and changed when `value1` or `value2` changes, avoiding unnecessary
   re-renders of `ChildComponent`.
*/
  const MyContext = React.createContext();

  function ParentComponent() {
    const [value1, setValue1] = useState(1);
    const [value2, setValue2] = useState(2);

    const contextValue = useMemo(() => ({ value1, value2 }), [value1, value2]);

    return (
      <MyContext.Provider value={contextValue}>
        <ChildComponent />
      </MyContext.Provider>
    );
  }

// 3. **Selector Functions**:
/*
  - **Problem**: Context consumers might re-render even when they don’t
              need all the context data.

  - **Solution**: Create a custom hook or use a selector function to only
               extract the part of the context that the component needs.
*/
  Example:
/*
  **Benefit**: This allows you to tightly control which parts of the context
             each component subscribes to, helping reduce unnecessary re-renders.
*/
  function useValue1() {
    const context = useContext(MyContext);
    return context.value1;
  }

  function ChildComponentA() {
    const value1 = useValue1();
    return <div>{value1}</div>;
  }

//*  Conclusion:
/*
   When using React's context, it’s important to be mindful of how context changes
   affect re-rendering in consuming components. Splitting contexts, memoizing values,
   and using selector functions are effective strategies for avoiding unnecessary
   re-renders and keeping your application performance optimal.
*/
```

```

// React Router :  --- []

// install the react router dom library :
```
    npm install react-router-dom --save
```

// in the index.js :
    import React from 'react';
    import ReactDOM from 'react-dom/client';
    import './index.css';
    import App from './App';
    import reportWebVitals from './reportWebVitals';
    import {BrowserRouter} from "react-router-dom"

    const root = ReactDOM.createRoot(document.getElementById('root'));
    root.render(
    <React.StrictMode>

        <BrowserRouter>
        <App />
        </BrowserRouter>

    </React.StrictMode>
    );

// App.js : create new route Example :
    import "./App.css";
    import { Route, Routes } from "react-router-dom";
    function App() {
        return (
            <>
                <div className="App">
                    <Routes>
                        <Route path="/Home" element={<h1>Hello From Home</h1>} />
                        <Route path="/" element={<h1>Hello From Home</h1>} />
                    </Routes>
                </div>
            </>
        );
    }
    export default App;

// create a link Example :
    import "./navBar.css";
    import { Link } from "react-router-dom";

    export default function NabBar() {
        return (
      <nav>
        <h1>
          <span>M</span>ajid
        </h1>
        <ul>
          <li>
            <Link to="/home">Home</Link>
          </li>

          <li>
            <Link to="/services">Services</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
        </ul>
      </nav>
    );
  }

//  Dynamic routing :
// ----App.js :----
```
  <Route path="/pageName/:dynamicEndPoint" element={<ServiceDetails/>} ></Route>
```

// ----ServiceDetails.js :----
import { useParams } from "react-router-dom";

import Service from "../../Service/Service";
import { servicesListContext } from "../../../contexts/ServicesContext";
import { useContext } from "react";
import ErrorPage from "./../ErrorPage/ErrorPage";
export default function ServiceDetails({ title, description }) {
  const { serviceId } = useParams();
  const servicesList = useContext(servicesListContext);

  const targetService = servicesList.find((serviceItem) => {
    return serviceItem.id == serviceId;
  });

  return (
    <>
      <h1>Welcome to the service details page </h1>;<h1> Service id : {serviceId}</h1>
      {targetService ? <Service id={targetService.id} name={targetService.name} description={targetService.description}></Service> : <ErrorPage />}
    </>
  );
}

```

~~~
// add error page :
```
     <Route path="*" element={<ErrorPage />} />
```

// routes group:
```
    <Route path="/services" >
        <Route path=":serviceId" element={<ServiceDetails />} />
        <Route index element={<services/>}>
        <Route path="new" element={<NewService />} />
        <Route path="delete" element={<DeleteService />} />
    </Route>
```

// add layout to a  routes group
// ----App.js :----
```
<Route path="/services" element={<ServiceLayout />}>
        <Route index element={<services/>}>
        <Route path="new" element={<NewService />} />
        <Route path="delete" element={<DeleteService />} />
    </Route>
```
// ----ServicesLayout.js:----
    import { Outlet } from "react-router-dom";
    export default function ServiceLayout() {
        return (
            <div>
                <h1 style={{ width: "100vw", background: "red", color: "white" }}>Services</h1>

                <Outlet />
            </div>
        );
    }

// download the material ui library :
```
    npm install @mui/material @emotion/react @emotion/styled --save
```
// create a theme example :
    import "./App.css";

    import { createTheme, ThemeProvider} from "@mui/material/styles";

    import Button from "@mui/material/Button";

    import { orange, green } from "@mui/material/colors";

    import { Chip } from "@mui/material";
~~~

~~~
  const theme = createTheme({
    palette: {
      primary: {
        main: orange[500],
      },
      secondary: {
        main: green[500],
      },
    },
  });
  function App() {
    return (
      <ThemeProvider theme={theme}>
        <div className="App">
          <Button color="primary" variant="outlined">
            Click me
          </Button>
          <Chip label="primary" color="primary" variant="outlined" />

        </div>
      </ThemeProvider>
    );
  }

  export default App;

// install material icons :
```
  npm install @mui/icons-material  --save
```

// using uuid library to generate unique ids ---[] :
  // install the library :
  ```
  npm install uuid --save
  ```

  // use the library to generate a unique id :
  import { v4 as uuidv4 } from 'uuid';
  uuidv4(); // ⇨ '9b1deb4d-3b7d-4bad-9bdd-2b0d7b3dcb6d'



// use effect :-- [-]
/*
  `useEffect` is one of React's most powerful hooks. It allows you to perform side effects
  in functional components. Side effects are operations like data fetching, manual
  DOM manipulation, subscribing to services, timers, etc. The `useEffect`
  hook is React's way of handling such operations in a declarative, clean way,
  while also supporting lifecycle management (like mounting, updating, and unmounting).
~~~

```
*/

//  Basic Syntax
useEffect(() => {
    // side effect logic (e.g., data fetching, setting up event listeners)
    return () => {
        // cleanup logic (e.g., removing event listeners)
    };
    }, [dependencies]);

// Key Concepts of `useEffect`
/*
1. **Side Effects**: These are operations that are not purely related to rendering
the UI (like API calls, updating the document title, setting up listeners, etc.).

2. **Dependencies Array**: This controls when the effect runs. `useEffect`
    runs whenever the component renders, but you can control when it re-runs
    by specifying certain dependencies in the array. These dependencies are
    the variables or states that the effect depends on.

   - No dependency array: If you don’t pass any array, the effect runs
     after every render (mount and every update).
   - Empty array `[]`: If you pass an empty array, the effect runs only once,
     after the initial render (componentDidMount behavior).
   - With dependencies `[dep1, dep2]`: The effect runs only when any of the values
     in the array change. It acts as a watcher for those dependencies
     (componentDidUpdate behavior).

3. **Cleanup Function**: Sometimes, side effects need to be cleaned up
    (e.g., removing an event listener, clearing a timer, aborting an API request).
    You return a function inside `useEffect` that will be executed when the component
    is unmounted or before the effect is re-executed (if the dependencies change).
*/

//  Example 1: Basic Use Case (Data Fetching)
/*
    - This example fetches user data when the component mounts.
    The empty array `[]` ensures the effect runs only once.
*/
    import { useState, useEffect } from "react";

    function UserList() {
    const [users, setUsers] = useState([]);

    useEffect(() => {
        // Side effect: fetch data
        fetch('https://jsonplaceholder.typicode.com/users')
        .then((response) => response.json())
        .then((data) => setUsers(data));
    }, []); // Empty array: effect runs only once after initial render
```

```

  return (
    <ul>
    {users.map((user) => (
      <li key={user.id}>{user.name}</li>
    ))}
    </ul>
  );
  }

  export default UserList;

// Example 2: Effect with Dependencies
/*
  - This effect updates the document's title every time `count` changes.
  The dependency array `[count]` ensures the effect only runs when `count`
  is updated.
*/
  import { useState, useEffect } from "react";

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;

    // Cleanup is not required here
  }, [count]); // Effect runs every time the 'count' value changes

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

// Example 3: Cleanup Effect (Event Listener)
/*
  In this example, the event listener for `mousemove` is added when
  the component mounts. The cleanup function removes the event listener
  when the component unmounts to prevent memory leaks.
*/
  import { useState, useEffect } from "react";

  function MouseTracker() {
  const [position, setPosition] = useState({ x: 0, y: 0 });

  useEffect(() => {
    const updateMousePosition = (e) => {
```

```
    setPosition({ x: e.clientX, y: e.clientY });
    };

    window.addEventListener("mousemove", updateMousePosition);

    // Cleanup function to remove event listener
    return () => {
    window.removeEventListener("mousemove", updateMousePosition);
    };
  }, []); // Empty array: effect runs only on mount and cleanup on unmount

  return (
    <div>
    Mouse position: {position.x}, {position.y}
    </div>
  );
  }

// When Does `useEffect` Run?
/*
- On Mount: If you pass an empty array `[]`, it behaves like `componentDidMount`
        and runs only once after the component renders for the first time.

- On Update: If you pass a list of dependencies `[dep1, dep2]`, the effect
         runs again whenever any of those dependencies change, similar
         to `componentDidUpdate`.

- On Unmount: The cleanup function returned by `useEffect` behaves like
       `componentWillUnmount`, executing just before the component is removed
        from the DOM or before the effect is rerun due to changes in dependencies.
*/

//  Example 4: Conditional Side Effects
/*
  You can conditionally run effects based on state or props:
*/

  function FetchDataOnToggle({ shouldFetch }) {
  const [data, setData] = useState(null);

  useEffect(() => {
    if (!shouldFetch) return;

    fetch('https://jsonplaceholder.typicode.com/posts')
    .then((res) => res.json())
    .then((data) => setData(data));
  }, [shouldFetch]); // Effect only runs if 'shouldFetch' changes

  return (
    <div>
```

~~~
        {shouldFetch ? <p>Fetched data!</p> : <p>No fetch initiated.</p>}
      </div>
    );
    }

//  Common Mistakes with `useEffect`
/*
1. **Forget to pass the dependency array**: Without the array,
    the effect will run on every render, which can cause performance issues.

2. **Using state setters inside effects without dependency**:
     If you're setting state in an effect and don't include it in the
     dependencies, it can lead to stale closures or infinite loops.

3. **Wrong dependencies**: Always ensure that any external values used inside
    `useEffect` are listed in the dependency array. This includes props, states,
     or any other variables. Omitting necessary dependencies may cause bugs or
     stale values to be used.
*/

// Conclusion
/*
- Mounting, Updating, and Unmounting: `useEffect` can handle all phases
                                       of the component lifecycle.

- Dependency management: React automatically handles when the effect
                         should run by observing the values in the dependency array.

- Cleanup: You can ensure proper memory management by using the
           cleanup function.

By understanding `useEffect`, you can manage side effects efficiently
in React functional components!
*/

// Run React project in production environment:
/*
This command will:
1. Convert your React project to pure HTML, CSS, and JavaScript code (compiling).
2. Bundling: Collect all code and libraries into one directory for production.
*/
```
    npm run build
```

//  2. useMemo:
/*
    - The `useMemo` hook allows you to memoize expensive calculations, preventing
    them from being recalculated  on every re-render.
*/
~~~

```
/*
  - This ensures that the `computeHeavyTask` function is only recalculated when
   `input` changes, reducing unnecessary work.
*/
const expensiveCalculation = useMemo(() => {
  return computeHeavyTask(input);
}, [input]);

// Example :
  import { Stack } from "@mui/material";
  import TodoListItem from "../TodoListItem.js/TodoListItem";
  import "./TodoList.css";
  import { useMemo } from "react";
  export default function TodoList({ arrTasksStat, setDeleteModalStat, deleteModalStat, editModalStat, setEditModalStat, completeTask, filterStat }) {
    let filteredTasks = useMemo(() => {
      return arrTasksStat.filter((taskItem) => {
        if (filterStat == "completed") return taskItem.isCompleted;
        if (filterStat == "not completed") return !taskItem.isCompleted;

        return true;
      });
    }, [arrTasksStat, filterStat]);

    const memoizedTasks = useMemo(() => {
      return filteredTasks.map((taskItem) => <TodoListItem key={taskItem.id} taskItem={taskItem} deleteModalStat={deleteModalStat} setDeleteModalStat={setDeleteModalStat} editModalStat={editModalStat} setEditModalStat={setEditModalStat} completeTask={completeTask} />);
    }, [filteredTasks,deleteModalStat,editModalStat]);
    return (
      <Stack spacing={1} className="TodoListComponentClass">
        {memoizedTasks}
      </Stack>
    );
  }

// #### 3. useCallback:
/*
  - The `useCallback` hook memoizes callback functions, which is useful for avoiding
  re-creating functions every time a component re-renders.
*/
const handleClick = useCallback(() => {
  console.log("Button clicked");
}, []);

/// Turning the context into  a provider
/*
  To turn a context into a provider in React, you create a context using `React.createContext()`
  and wrap your component tree with the `Provider` component that comes with the context.
  The `Provider` component will allow you to pass down values (like state or functions)
  to any descendant component that consumes the context.
```

```
*/
//Here’s a step-by-step guide on how to set up and use a context provider in React:
// */ ### 1. Create the Context ---[]

//In a separate file (e.g., `MyContext.js`), create a context using `React.createContext()`.


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

    // Create the context
    export const MyContext = createContext();


// ### 2. Create a Provider Component --[]
/*
    Inside the same file (or a new one), create a provider component that
    uses `MyContext.Provider`. This component will hold any shared state and
    functions and pass them as values to the provider.
*/

    // Create a provider component
    export const MyProvider = ({ children }) => {
    const [state, setState] = useState("Hello from context!");

    // Value to be provided to consumer components
    const contextValue = {
        state,
        updateState: (newState) => setState(newState),
    };

    return (
        <MyContext.Provider value={contextValue}>
        {children}
        </MyContext.Provider>
    );
    };


// ### 3. Wrap Your App (or Part of Your App) with the Provider --[]
/*
    To make the context available in your component tree, wrap
    your application (or a specific part of it) with the provider.
*/

// In `App.js`:

    import React from 'react';
    import { MyProvider } from './MyContext';
    import SomeComponent from './SomeComponent';

```

```jsx
  function App() {
  return (
    <MyProvider>
    <SomeComponent />
    </MyProvider>
  );
  }

  export default App;


// ### 4. Consume the Context in a Component --[]
/*
  Now you can access the context values in any descendant component
  using the `useContext` hook.
*/

// In `SomeComponent.js`:
  import React, { useContext } from 'react';
  import { MyContext } from './MyContext';

  function SomeComponent() {
  const { state, updateState } = useContext(MyContext);

  return (
    <div>
    <p>{state}</p>
    <button onClick={() => updateState("New value from SomeComponent!")}>
      Update State
    </button>
    </div>
  );
  }

  export default SomeComponent;


// ### Summary

// - **Create** the context using `React.createContext()`.
// - **Wrap** your app with the context provider component.
// - **Consume** the context in any component using `useContext(MyContext)`.
/*
  This approach makes it easy to share state and functions across your component
  tree without prop drilling, making your code more organized and scalable.
*/

// ## Documentation: `useReducer` Hook in React with Example

// ### Overview
```

~~~
/*
The `useReducer` hook is a React function that provides an alternative to `useState` for managing complex state logic in functional components. It is especially useful when:
- The state transitions are complex.
- State updates depend on previous states.
- You want a centralized place to handle multiple state transitions.
*/
// The `useReducer` hook works similarly to `Redux` reducers, encapsulating state logic in a single reducer function to handle dispatched actions.

// ### Syntax

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

// #### Parameters:
/*
- **`reducer`**: A function that determines the next state based on the current state and an action.
- **`initialState`**: The initial value for the state when the component is first rendered.
*/

// #### Returns:
/*
- **`state`**: The current state managed by the reducer.
- **`dispatch`**: A function to send actions to the reducer, triggering a state update.
*/


// ### Example: Calculator with `useReducer`
/*
This example demonstrates how `useReducer` can manage a simple calculator that performs basic arithmetic operations.
It uses a reducer function to handle operations like addition, subtraction, multiplication, and division.
*/

// #### File Structure
```
src
├── App.js
├── reducers
│   └── resultReducer.js
└── App.css
```

// ### Step 1: Define the Reducer Function
/*
In `src/reducers/resultReducer.js`, define the `resultReducer` function. This function manages the state logic based
on the action type. Each action type corresponds to a basic arithmetic operation.
*/
```javascript
~~~

```
// src/reducers/resultReducer.js

export function resultReducer(state, action) {
  const { firstValue, secondValue } = action.payload;
  switch (action.type) {
    case "addition":
      return firstValue + secondValue;
    case "subtraction":
      return firstValue - secondValue;
    case "multiplication":
      return firstValue * secondValue;
    case "division":
      return secondValue !== 0 ? firstValue / secondValue : "Cannot divide by zero";
    default:
      return state;
  }
}
```

// In this reducer:
/*
- The `state` represents the current result.
- Each `case` in the `switch` statement handles a specific action type by performing the corresponding arithmetic operation on `firstValue` and `secondValue`.
*/

// ### Step 2: Set Up `useReducer` in the Component
/*
In `src/App.js`, use `useReducer` to initialize the state and manage the dispatch function. Define helper functions to handle input and dispatch actions.
*/

// src/App.js

import React, { useReducer, useRef } from "react";
import { resultReducer } from "./reducers/resultReducer";
import { Stack, Button, FilledInput } from "@mui/material";
import { blue } from "@mui/material/colors";

function App() {
  const firstInputRef = useRef();
  const secondInputRef = useRef();

  // Initialize useReducer with the reducer function and an initial result of 0
  const [reducerResult, resultDispatch] = useReducer(resultReducer, 0);

  // Function to retrieve the input values, converting them to floats or defaulting to 0
  function getInputsValues() {
    const firstValue = parseFloat(firstInputRef.current.value) || 0;
    const secondValue = parseFloat(secondInputRef.current.value) || 0;
```
```

```jsx
    return { firstValue, secondValue };
  }

  // Dispatch functions for each arithmetic operation
  function handleAddition(e) {
    e.preventDefault();
    resultDispatch({ type: "addition", payload: getInputsValues() });
  }

  function handleSubtraction(e) {
    e.preventDefault();
    resultDispatch({ type: "subtraction", payload: getInputsValues() });
  }

  function handleMultiplication(e) {
    e.preventDefault();
    resultDispatch({ type: "multiplication", payload: getInputsValues() });
  }

  function handleDivision(e) {
    e.preventDefault();
    const { firstValue, secondValue } = getInputsValues();
    if (secondValue === 0) {
      alert("Cannot divide by zero");
      return;
    }
    resultDispatch({ type: "division", payload: { firstValue, secondValue } });
  }

  return (
    <div className="App">
      <h1>Calculator Result</h1>
      <h2>{reducerResult}</h2>

      <form>
        <Stack alignItems="center" spacing="10px">
          <FilledInput
            type="number"
            inputRef={firstInputRef}
            sx={{ width: "70%", borderLeft: `4px solid ${blue[500]}` }}
            placeholder="Enter the first number"
          />
          <FilledInput
            type="number"
            inputRef={secondInputRef}
            sx={{ width: "70%", borderLeft: `4px solid ${blue[500]}` }}
            placeholder="Enter the second number"
          />
        </Stack>

```

```
          <Stack alignItems="center" sx={{ marginTop: "30px" }} spacing="15px">
            <Button variant="outlined" onClick={handleAddition} sx={{ width: "200px" }}>Add</Button>
            <Button variant="outlined" onClick={handleSubtraction} sx={{ width: "200px" }}>Subtract</Button>
            <Button variant="outlined" onClick={handleMultiplication} sx={{ width: "200px" }}>Multiply</Button>
            <Button variant="outlined" onClick={handleDivision} sx={{ width: "200px" }}>Divide</Button>
          </Stack>
        </form>
      </div>
  );
}

export default App;


// ### Explanation of Key Parts

// 1. **`useReducer` Initialization**:
/*
  Here, `useReducer` initializes the `reducerResult` state to `0` and provides `resultDispatch` to handle actions.
*/
  const [reducerResult, resultDispatch] = useReducer(resultReducer, 0);

// 2. **Dispatching Actions**:
/*
   Each button in the form triggers an arithmetic operation by dispatching an action with a
   specific type (`"addition"`, `"subtraction"`, etc.) and `payload` containing the input
   values.
*/
//    For example:

  function handleAddition(e) {
    e.preventDefault();
    resultDispatch({ type: "addition", payload: getInputsValues() });
  }


// 3. **Helper Function `getInputsValues`**:
/*
   This function retrieves the numeric values from the input fields,
   defaulting to `0` if the input is empty or invalid.
*/
// ### Summary of Benefits of Using `useReducer` Here
/*
- **Centralized Logic**: All arithmetic operations are handled in one place (`resultReducer`),
   making the logic easier to read and maintain.
- **Predictability**: State changes happen only through `resultReducer`, ensuring predictable
   transitions based on `action.type`.
- **Scalability**: Adding more operations (like exponentiation) requires just adding another
   case to `resultReducer` and creating a corresponding dispatch function.
*/
```

```
// ### Pros and Cons of `useReducer`

//  Pros |
/*
    1- Centralized and modular logic for complex state updates
    2- Easily traceable actions, making debugging easier
    3- Facilitates future scalability in managing state transitions
*/
// Cons |
/*
   1-  Can be overkill for simple state updates
   2-  Might feel more complex than `useState`
   3-  Not ideal for small, isolated state values
*/

// ### Best Practices
/*
    - Use `useReducer` when managing complex state logic or when updates
      involve multiple actions or dependencies.
    - Keep the reducer function pure. Avoid side effects (like API calls or asynchronous logic)
      inside the reducer.

    - Use constants for action types to avoid typos and make updates easier.
*/
// ### When to Use `useReducer` vs `useState`
/*
    - Use **`useState`** for simple state needs, like toggling a boolean or managing a small counter.
    - Use **`useReducer`** when:
    - The state has multiple sub-values or complex transitions.
    - State updates depend on previous state values.
    - The state transitions require explicit handling for clarity and scalability.
*/

// ### Final Notes
/*
    This example demonstrates how `useReducer` can simplify handling complex state updates
    in a React functional component. It’s particularly useful for modularizing logic in
    more advanced components or applications. By encapsulating state transitions in the
    reducer, the application remains organized, predictable, and easy to expand in functionality.
*/

// redux :
/*
    Redux: A Comprehensive Guide
    Redux is a predictable state management library for JavaScript applications,
    commonly used with React but can be used with any JavaScript framework or library.
    It helps manage the state of an application in a consistent way, making debugging
    and testing easier.
*/

```

```
// ### **What is Redux Toolkit (RTK)?**
/*
Redux Toolkit is the official, opinionated way to write Redux applications. It simplifies
 common Redux tasks like:
- Configuring the store.
- Creating reducers and actions.
- Handling complex state updates.
- Managing side effects with tools like `createAsyncThunk`.

---

### **Core Features of Redux Toolkit**

1. **`configureStore`**:
   - Simplifies store setup with built-in support for middleware and DevTools.

2. **`createSlice`**:
   - Combines actions and reducers into a single "slice" of the state.

3. **`createAsyncThunk`**:
   - Simplifies handling asynchronous logic (e.g., API calls).

4. **DevTools Integration**:
   - Automatically integrates Redux DevTools without extra configuration.

---
*/
// ### **Modern Redux Workflow**

// #### 1. **Install Redux Toolkit and React-Redux**:
    npm install @reduxjs/toolkit react-redux



// #### 2. **Create a Slice**:
// The `createSlice` method allows you to define the reducer and actions in one place.


// features/counterSlice.js
    import { createSlice } from '@reduxjs/toolkit';

    const counterSlice = createSlice({
        name: 'counter',
        initialState: { value: 0 },
        reducers: {
            increment: (state) => {
                state.value += 1; // RTK allows mutating state via Immer
            },
            decrement: (state) => {
                state.value -= 1;
```

```
      },
      incrementByAmount: (state, action) => {
        state.value += action.payload;
      },
    },
  });

  export const { increment, decrement, incrementByAmount } = counterSlice.actions;
  export default counterSlice.reducer;




// #### 3. **Configure the Store**:
// Use `configureStore` to set up your store, middleware, and reducers.

// app/store.js
  import { configureStore } from '@reduxjs/toolkit';
  import counterReducer from '../features/counterSlice';

  export const store = configureStore({
    reducer: {
      counter: counterReducer, // Add slices here
    },
  });

  export default store;



// #### 4. **Connect Redux to React**:
// Wrap your app with the `Provider` component to make the Redux store available throughout the component tree.


// index.js
  import React from 'react';
  import ReactDOM from 'react-dom';
  import { Provider } from 'react-redux';
  import { store } from './app/store';
  import App from './App';

  ReactDOM.render(
    <Provider store={store}>
      <App />
    </Provider>,
    document.getElementById('root')
  );


// #### 5. **Use Redux State and Dispatch in Components**:
```

```
// Use `useSelector` to access the state and `useDispatch` to dispatch actions.


// Counter.js
    import React from 'react';
    import { useSelector, useDispatch } from 'react-redux';
    import { increment, decrement, incrementByAmount } from './features/counterSlice';

    const Counter = () => {
        const count = useSelector((state) => state.counter.value);
        const dispatch = useDispatch();

        return (
            <div>
                <h1>{count}</h1>
                <button onClick={() => dispatch(increment())}>Increment</button>
                <button onClick={() => dispatch(decrement())}>Decrement</button>
                <button onClick={() => dispatch(incrementByAmount(5))}>Increment by 5</button>
            </div>
        );
    };

    export default Counter;



// ### **Handling Async Logic with Redux Toolkit**
/*
For asynchronous operations (e.g., fetching data from an API),
you can use `createAsyncThunk`.
*/
// #### 1. **Create an Async Thunk**:

// features/userSlice.js
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';

// Async thunk to fetch users
    export const fetchUsers = createAsyncThunk('users/fetchUsers', async () => {
        const response = await fetch('https://jsonplaceholder.typicode.com/users');
        return response.json();
    });

    const userSlice = createSlice({
        name: 'users',
        initialState: { data: [], status: 'idle', error: null },
        reducers: {},
        extraReducers: (builder) => {
            builder
                .addCase(fetchUsers.pending, (state) => {
                    state.status = 'loading';
        })
        .addCase(fetchUsers.fulfilled, (state, action) => {
          state.status = 'succeeded';
          state.data = action.payload;
        })
        .addCase(fetchUsers.rejected, (state, action) => {
          state.status = 'failed';
          state.error = action.error.message;
        });
    },
  });

  export default userSlice.reducer;


// #### 2. **Use the Thunk in a Component**:

  import React, { useEffect } from 'react';
  import { useSelector, useDispatch } from 'react-redux';
  import { fetchUsers } from './features/userSlice';

  const UserList = () => {
    const dispatch = useDispatch();
    const users = useSelector((state) => state.users.data);
    const status = useSelector((state) => state.users.status);

    useEffect(() => {
      if (status === 'idle') {
        dispatch(fetchUsers());
      }
    }, [status, dispatch]);

    return (
      <div>
        {status === 'loading' && <p>Loading...</p>}
        {status === 'succeeded' && (
          <ul>
            {users.map((user) => (
              <li key={user.id}>{user.name}</li>
            ))}
          </ul>
        )}
        {status === 'failed' && <p>Error loading users</p>}
      </div>
    );
  };

  export default UserList;

// ### **Advantages of Redux Toolkit**
```

```
/*
    1. **Simplified Boilerplate**:
    - Reduces the need to write separate action types, action creators, and switch statements.

    2. **Built-In Middleware**:
    - Automatically includes `redux-thunk` for handling async logic.

    3. **Immer for Immutability**:
    - Allows writing mutable-looking code that is internally immutable.

    4. **Developer-Friendly Defaults**:
    - Pre-configured Redux DevTools support.

    5. **Scalable Structure**:
    - Organizes your code into "slices," making it easy to manage large-scale applications.


*/
// ### **When to Use Redux Toolkit**
/*
    - Your app requires complex state management.
    - State is shared across multiple components.
    - You need to handle asynchronous operations (e.g., API calls).
*/
/*
For smaller applications, consider alternatives
like React Context API if Redux feels too heavy.
*/
```