

# react

---

## browser architecture

- network component
  - responsible for sending request and receiving response from server
  - communicated with server over network
- rendering engine or layout engine
  - converts the html/css to JS
  - it used html and css parsers
  - parser is a program which converts one form into another form
- javascript engine
  - core of any browser
  - responsible for executing JS and producing the output
  - all the JS engines today are written in C++
- UI component
  - responsible for showing the output to the user on the browser's window
- web storage
  - responsible for maintaining the history/sessions inside browser
  - uses JS objects like sessionStorage and localStorage

## server

- process used to serve certain type of requests
- types
  - web server
    - used to serve http or https requests
    - e.g. apache2, nginx, IIS (Internet Information Service), express
  - database server
    - RDBMS: MySQL, MariaDB, Oracle, MS SQL Server, PostgreSQL, IBM DB2 etc.
    - No SQL: MongoDB, redis, firebase, couch db, cocroach db etc.
  - file server
    - used to share files with many users at a same time
    - e.g. ftp (file transfer protocol), nfs (network file system), smb (server message block)
  - mail server
    - used to send and receive emails
    - e.g. Postfix (SMTP server), DoveCot (IMAP and PoP3)
- ways for client to communicate with server
  - REST
    - stands for REpresentational State Transfer
    - a design pattern (not a protocol) developed on top on HTTP (web) architecture
    - it contains
      - http methods

- http request
  - http response
- used to send the data (generally in the form of JSON) to the client
- can be used to send the data in different format as well (like XML)
- GraphQL
  - stands for graph query language
  - developed on top of REST
  - uses a POST method to send the query or mutation to perform the communication
- gRPC
- SOAP: Simple Object Access Protocol

## JS pre-requisites

- map()
  - used to transform a value from one form to another
  - use it only when there is no condition required to select the data
  - represents projection
- filter()
  - used to filter (select) data from an array based on a condition
  - use it when a condition is required to select the data
  - represents selection
- combination of map and filter
  - use it when both projection and selection is required
  - always use filter before map
- functional programming language
  - function is considered as first class citizen
    - function is treated as an object of type function
  - a function can be passed as an argument to another function
    - e.g. map(), filter()
  - a function can be returned as a return value of another function
    - e.g. closer in JS

## SPA

- single page application
  - uses only one html page which gets rendered / loaded initially
  - once it gets render it never (re)loads again
    - SPA application does NOT send new request for UI for every other page (component)
  - it updates the internal components (sections) without loading the entire page
- advantages
  - SPA application can cache the contents for offline use
  - faster than multi-page application
  - provides component-based architecture which promotes the re-usability
- disadvantages
  - the first time loading is bit slower than multi-page application
    - it has to load all the components(sections) while loading the application for the first time
  - these applications are memory-heavy

- frameworks or libraries used to develop SPAs
  - react: developed by Facebook in 2013
  - angular: developed by Google in 2011
  - preact: light weight react
  - svelte: faster than react but has less ecosystem
- tooling
  - language: react (JS), angular (TS), preact (JS), svelte (JS)
  - bundler:
    - tool to create a bundle (archive of all the required pages of a web application)
    - e.g. Webpack, vite
  - platform: node.js

## angular

- angular is a framework used to develop SPA type application
- developed by Google in 2011
- developed in TypeScript
- TS is required to develop angular application
- bit slower than react as it requires more memory
- framework
  - contains different modules/libraries
  - most of the times, a frame is built using multiple languages
  - e.g. angular: TS + HTML + CSS + JS + others

## react

- react is a library used to develop SPA type application
- developed by Facebook (Meta) in 2013
- uses component-based architecture
- developed in JavaScript
- JS or TS can be used to develop react application
- since it is a library, it requires less memory compared to Angular
- it is bit faster than Angular
- free to use
- library
  - single file developed using only one language and solves only one problem
  - library requires less memory than a framework
- can be used
  - using CDN links

- add the following CDN links to the application

- 
- 

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0"
    />
    <title>Document</title>
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
    ></script>
    <script
      crossorigin
      src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"
    ></script>
  </head>
  <body></body>
</html>
```

- react.development.js
  - adds the react foundation
- react-dom.development.js
  - adds the virtual DOM (manged by react) support in react application

- using bundlers like vite

- uses JSX to create the UI
- react basic application using CDN links

```
<!DOCTYPE html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta
      name="viewport"
      content="width=device-width, initial-scale=1.0"
    />
    <title>Document</title>
    <script
      crossorigin
      src="https://unpkg.com/react@18/umd/react.development.js"
```

```

</script>
<script
  crossorigin
  src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"
></script>
</head>
<body>
  <div id="root"></div>

  <script>
    // get the object of div having id as root
    const root = document.getElementById('root')

    // create element to render
    // param1: type of element to be created
    // param2: properties of element
    // param3: contents of the element
    const h1 = React.createElement('h1', {}, 'welcome to react')

    // render the h1 created by react
    ReactDOM.createRoot(root).render(h1)
  </script>
</body>
</html>

```

- Virtual DOM
  - in-memory copy of browser or real DOM
  - real DOM is created by the browser to render the UI in the browser's window
  - when real DOM changes, the entire real DOM gets re-rendered
    - this is very memory-heavy operation
    - this may slow down the UI generation
  - react create a in-memory of real DOM called as virtual DOM to solve the re-rendering of web page when there is a change in the page
  - how virtual DOM works
    - when react application loads, react library creates a copy of real DOM and renders the entire web page when it gets loaded initially (only for the first time)
    - when there is a change required, the change will be performed on the virtual DOM first
    - it will updates the virtual DOM with new copy of changed element
    - then react finds the difference between old DOM object and new DOM object (diffing algorithm)
    - then react creates a patch of new changes to be applied on the real DOM
    - finally the patch will be applied on real DOM to change the browser UI (re-concilation)
    - because of this process, the entire real DOM never reloads which is why the react application is faster than MPA

simple examples

**render single element**

```
const root = document.getElementById('root')
const element = React.createElement('h2', {}, 'welcome to react')
ReactDOM.createRoot(root).render(element)
```

### render multiple elements

```
// get root element
const root = document.getElementById('root')

// create required element(s)
const h1 = React.createElement('h1', {}, 'this is a header1')
const h2 = React.createElement('h2', {}, 'this is a header2')
const h3 = React.createElement('h3', {}, 'this is a header3')

// render the elements by using an array of elements
ReactDOM.createRoot(root).render([h1, h2, h3])
```

### render hierarchy of element

```
// create root element
const root = document.getElementById('root')

// create a child element
const h1 = React.createElement('h1', {}, 'Child element of div')

// create parent div
const div = React.createElement('div', {}, h1)

// render the element
ReactDOM.createRoot(root).render(div)
```

## html

- hyper text markup language
- markup language: language made with tags and contents(data)
- uses XML syntax to design web pages
- the tags are standardized by W3C

## XML

- eXtensible markup language
- used as a serialization format
  - making raw data into information

- e.g.
  - meaningless raw fact: 10, person1, pune
  - meaningful information
    - - 10 - person1 -  
pune
    -

## JSX

- using XML syntax in JavaScript
- used to design user interface in react
- react uses babel behind the scene to convert the JSX syntax to `React.createElement()` function

```
// jsx syntax
const h1 = <h1>this is header</h1>
```

## babel

- is a JavaScript compiler used to compile one form of JS to another
- can be used to convert higher version (which browser can NOT understand) of JS to lower version (which browser can understand)
- babel is responsible for converting the JSX into `React.createElement()`
- can be added in a page using CDN link
  -

```
const root = document.getElementById('root')

// JSX syntax
const h1 = <h1>this is header</h1>

// babel converts the above line to the the line below
// const h1 = React.createElement('h1', {}, 'this is header')

ReactDOM.createRoot(root).render(h1)
```

## conventions to write JSX

- use xml (html) syntax in JS
- the properties are using camelCase: `className`
- to read value of a variable use string interpolation (`{}`)
- array
  - to read values of an array use map function
  - in array, every element must have a unique key to identify each element uniquely
    - these keys are used by diffing algorithm to track the changes from older to news state of virtual DOM

- execute the JS expression or function using {}

### rendering single element

```
const root = document.getElementById('root')
const h1 = <h1>this is header</h1>
ReactDOM.createRoot(root).render(h1)
```

### rendering hierarchy of elements

```
const root = document.getElementById('root')

// create element hierarchy
const div = (
  <div>
    <h1>this is header 1</h1>
    <h2>this is header 2</h2>
  </div>
)

ReactDOM.createRoot(root).render(div)
```

### string interpolation

- read variable's value in content of an element (string)

```
const root = document.getElementById('root')

// variable having a value
const userName = 'bill gates'

// string interpolation
const h1 = <h1>Welcome {userName}</h1>

ReactDOM.createRoot(root).render(h1)
```

### using expression inside {}

```
const root = document.getElementById('root')
const numbers = 10
const h1 = (
  <h1>
    number = {number}, cube = {number ** 3}
  </h1>
)
```

```
)  
ReactDOM.createRoot(root).render(h1)
```

### calling JS function inside JSX syntax

```
function cube(number) {  
  return number ** 3  
}  
  
const root = document.getElementById('root')  
const numbers = 10  
  
// invoke function cube()  
const h1 = (  
  <h1>  
    number = {number}, cube = {cube(number)}  
  </h1>  
)  
ReactDOM.createRoot(root).render(h1)
```

### rendering an array of strings

```
const root = document.getElementById('root')  
  
// array of strings  
const countries = ['india', 'uk', 'japan', 'bhutan', 'russia', 'Germany']  
  
// render the array  
const div = countries.map((country, index) => {  
  return <div key={index}>{country}</div>  
})  
  
ReactDOM.createRoot(root).render(div)
```

## react architecture

- react has a component-based architecture
- it flexible and reusable architecture

### component

- reusable entity in react which contains
  - user interface: created using JSX syntax
  - state (optional): used to store some data for internal use
  - event handlers (optional): used to handle events generated by the component UI

- props (optional): parameters (properties) of a component
- mandatory to return only one element as a return value of a component
  - if you have multiple elements, then wrap them inside a parent element
- types
  - class component
    - a class derived from React.Component class
    - this class implements method named render()
    - used a lot before react hooks to create stateful component
    - stateful means a component which is maintaining its state for internal use
    - using class component is not required as the react hook can maintain the state inside a functional component

```
const root = document.getElementById('root')

// class component
class Dummy extends React.Component {
  render() {
    return <div>this is a class component </div>
  }
}

// here the <Dummy /> is class component
ReactDOM.createRoot(root).render(<Dummy />)
```

- functional component
  - javascript function which returns a JSX user interface
  - optionally maintains its own state (using react hooks)
  - earlier it was used to create only stateless component but after introduction of react hooks today it is also used to create a stateful component as well
  - stateless means a component which is not maintaining its state
  - it is faster than class component
  - it is light weight component

```
const root = document.getElementById('root')

// functional component
function Dummy() {
  return <div>this is a functional component </div>
}

// here the <Dummy /> is function component
ReactDOM.createRoot(root).render(<Dummy />)
```

## conditional rendering

- rendering an element when a condition is true
- 

## export and import of functions

- default export
  - used to export only one entity (function/const/variable) from a file
  - uses `export default` keyword to export an entity

```
// Person.jsx
function Person() {
  return <h1>person</h1>
}

export default Person

// App.jsx
// do not use the extension
import Person from './components/Person'

// App.jsx
// here MyPerson will act as reference to existing function named
Person
import MyPerson from './components/Person'
```

- export or non-default export
  - can be used to export multiple entities from a file
  - use keyword `export` for all the exported entities

```
// Person.jsx
export function Person() {
  return <h1>Person</h1>
}

// App.jsx
// import the non-default Person
import { Person } from './components/Person'

// import the non-default Person with an alias
import { Person as MyPerson } from './components/Person'
```

## props

- represents properties of a component

- a component may receive a parameter named props to collect all the properties pass to the component
- is an object which has all the properties as key-value pairs
- is a way for parent to pass the information with child component
- is a readonly object
  - child is not supposed to change the props
  - even if child changes the props object, the parent will not receive the updated data

```
const root = document.getElementById('root')

// functional component
function Person(props) {
  return <div>name: {props['name']}</div>
}

// here the <Dummy /> is function component
ReactDOM.createRoot(root).render(<Person name='person1' />)
```

```
const root = document.getElementById('root')

// functional component
function Person(props) {
  // props destructuring
  const { name } = props
  return <div>name: {name}</div>
}

// here the <Dummy /> is function component
ReactDOM.createRoot(root).render(<Person name='person1' />)
```

```
const root = document.getElementById('root')

// functional component
function Person({ name }) {
  return <div>name: {name}</div>
}

// here the <Dummy /> is function component
ReactDOM.createRoot(root).render(<Person name='person1' />)
```

## state

- is an object maintained by a component
- is read-writable object
- gets stored in browser's memory

- is a collection of key-value pairs
- is a trigger to re-render the component's UI
  - when state of a component is updated, the component re-renders its UI with updated values of state
- to maintain a state object inside a component
  - class component
    - provides state object implicitly (this.state)
  - functional component
    - use react hook named useState()

## vite

- a build tool used to build multiple types of project
  - e.g. react, angular, svelte, preact etc.
- build
  - compile the project: minifying (reducing the file size) the js files
  - add the external referenced packages
  - create a bundle (collection of required files) for deployment
- create a vite project using npm

```
# create vite project
# > npm create vite <project name>
> npm create vite app1

# install all required modules
> npm install

# run the project
> npm run dev

# build the project
> npm run build

# test the project
> npm run test

# lint the project (find the errors/warnings inside the project)
> npm run lint
```

- create a vite project using yarn

```
# install yarn
> npm install -g yarn
```

```

# create vite project
> yarn create vite <project name>

# install the dependencies
> yarn

# start the project
> yarn dev

# build the project
> yarn build

# test the project
> yarn test

# lint the project (find the errors/warnings inside the project)
> yarn lint

```

- project hierarchy
  - node\_modules
    - directory which contains all the dependencies of a react application
    - dependencies
      - the packages required to run the application
      - when project is built, these dependencies will be included in the dist directory
      - e.g.
        - react: used to create react elements
        - react-dom: used to manage virtual DOM
    - devDependencies
      - these packages are required to develop the application
      - these packages are NOT required to run the application
      - when project is built, these dependencies will NOT be included in the dist directory
      - e.g.
        - vite: used to build the project
        - @vitejs/plugin-react: plugin required to build react application using vite
  - public
    - directory which contains public resources like css files and images
  - src
    - directory which contains the source code of the application
    - contains
      - assets: directory which contains the assets like images, audio or video files
      - main.jsx
        - entry point for the react application
        - starts the react application by rendering the root
        - it renders the first component of the application name App
      - index.css

- contains the styles which can be used across different components of the project
    - contains the global (for the project) styles
  - App.jsx
    - contains the application's first component
    - when react application starts, it loads the first component named App
    - all other components of the application get loaded by App component
  - App.css
    - contains the styles only used by the App component
- it may contain following directories
  - components
    - directory which contains reusable components
  - pages
    - directory which contains the pages (components) of the application
  - features/slices
    - directory which contains the sliced required for redux toolkit
  - services
    - directory which contains services (axios function) to connect to the backend
  - providers
    - directory with the context providers used to share the information with multiple components of the application
- .gitignore
  - used by git in order to ignore files while committing the changes
- eslint.config.js
  - contains eslint configuration
  - linter is a tool which is used to find the syntactical errors/warning in the code
  - eslint configuration is stored in this file
- index.html
  - only html file in the project
  - gets loaded in the browser when the application starts
  - never gets reloaded implicitly
- package.json
  - contain the meta information of the project
  - contains
    - name
    - version
    - description
    - dependencies
    - devDependencies
    - script
- package-lock.json or yarn.lock
  - contains list of all the dependencies installed from package.json
- README.md
  - contains the readme information about the project
- vite.config.js

- contains the configuration of vite
  - contains the configuration of vite plugins
- project dev booting
  - npm run dev command gets fired
  - vite starts a lite webserver on port 5173
  - lite server loads index.html file
  - index.html loads the main.jsx from src directory
  - main.jsx starts the first component named App
- project execution

## handling events in react

- define a event handler inside a component
- event handler
  - function which handles a required event
  - e.g. onClick() => click event of a button
- use event names provided by react
  - e.g. click => onClick

```
import React from 'react'

function Counter() {
  // event handler
  const onClick = () => {
    alert('button clicked')
  }

  return (
    <div>
      <button onClick={onClick}>test click</button>
    </div>
  )
}

export default Counter
```

## react hooks

- react hook is a special function in react which starts with **use** prefix
- it is mandatory to call the react hook function inside the component body directly
  - hooks can NOT be called inside the inner functions of a component
- e.g.
  - provided by react
    - useState()
    - useContext()
    - useRef()

- useCallback()
- useMemo()
- useReducer()
- useEffect()
- useLayout()
- provided react router
  - useNavigate()
  - useLocation()
- provided by redux
  - useSelector()
  - useDispatch()

## useState()

- used to add a member to component's state object
- accepts a initial value of the member
- returns an array with 2 members
  - 0th position (getter): reference to the state member to read the current value
  - 1st position (setter): reference to a function to update the value

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

function Counter() {
  // counter: getter to read the current value of state member
  // setCounter: setter to update value of state member
  const [counter, setCounter] = useState(0)

  const onIncrement = () => {
    setCounter(counter + 1)
  }

  return (
    <div>
      <h2>value of counter = {counter}</h2>
      <button onClick={onIncrement}>increment</button>
    </div>
  )
}

export default Counter
```

## useContext()

- used to use context in an application
- context
  - information which is required to shared with multiple components without props drilling
  - solution for props drilling
- props drilling

- sharing information from one component with multiple (other or child components) components
- try to avoid it as it will introduce many bugs
- note
  - the components with which you want to share the information, must be declared inside the Providers' list
  - if a component is not in the Providers list, the useContext() returns undefined

```
// App.jsx
import { createContext, useState } from 'react'
import Counter1 from './components/Counter1'
import Counter2 from './components/Counter2'

// create a context to share the information
// this will create an empty context object
export const CounterContext = createContext()

function App() {
  const [counter, setCounter] = useState(0)

  return (
    <div>
      {/* share the counter and setCounter with both the child components */}
      <CounterContext.Provider value={{ counter, setCounter }}>
        <Counter1 />
        <Counter2 />
      </CounterContext.Provider>
    </div>
  )
}

export default App
```

```
// Counter1.jsx

import React, { useContext } from 'react'
import { CounterContext } from '../App'

function Counter1() {
  // use the context and get counter and setCounter
  const { counter, setCounter } = useContext(CounterContext)

  const onIncrement = () => {
    setCounter(counter + 1)
  }

  return (
    <div>
      <h2>Counter1</h2>
    </div>
  )
}
```

```

        <div>counter: {counter}</div>
        <button onClick={onIncrement}>increment</button>
      </div>
    )
  }

  export default Counter1

```

```

// Counter2.jsx

import React, { useContext } from 'react'
import { CounterContext } from '../App'

function Counter2() {
  // use the context and get counter and setCounter
  const { counter, setCounter } = useContext(CounterContext)

  const onIncrement = () => {
    setCounter(counter + 1)
  }

  return (
    <div>
      <h2>Counter2</h2>
      <div>counter: {counter}</div>
      <button onClick={onIncrement}>increment</button>
    </div>
  )
}

export default Counter2

```

## useEffect()

- hook used to handle the component life cycle events
- accepts two parameters
  - 1st: function reference to get called in different events
  - 2nd: dependency array
- life cycle events
  - component mounted/started/loaded
    - this event occurs only once when the component gets loaded/mounted
    - dependency array must be empty

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

```
function Properties() {
  // handle component loaded/mounted event
  useEffect(() => {
    // this function gets executed immediately after the
    component is loaded
    console.log(`Properties component is loaded`)
  }, [])

  return (
    <div>
      <h1>Properties</h1>
    </div>
  )
}

export default Properties
```

- component unmounted/finished/unloaded

- this event is fired only once when the component is unmounted/unloaded
- dependency array must be empty

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

function Properties() {
  // handle component loaded/mounted event
  useEffect(() => {
    return () => {
      // this function is called when component is unmounted
    }
  }, [])

  return (
    <div>
      <h1>Properties</h1>
    </div>
  )
}

export default Properties
```

- component state changed

- required to perform an action when component state is changed
- dependency array must not be provided

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

function Counter() {
```

```

const [counter1, setCounter1] = useState(0)
const [counter2, setCounter2] = useState(0)

useEffect(() => {
  // this function gets called when there is change in state
  console.log(`state changed`)
})

return (
  <div>
    <h1>Counter</h1>
    <div>counter1: {counter1}</div>
    <div>counter2: {counter2}</div>

    <button onClick={() => setCounter1(counter1 + 1)}>
      update counter1
    </button>
    <button onClick={() => setCounter2(counter2 + 1)}>
      update counter2
    </button>
  </div>
)
}

export default Counter

```

- component state member changed
  - used to handle event when one state member is changed
  - add the member inside the dependency array

```

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

function Counter() {
  const [counter1, setCounter1] = useState(0)
  const [square, setSquare] = useState(0)

  useEffect(() => {
    // when counter1 changes, this function gets called
    console.log('counter1 is changed')
    setSquare(counter1 ** 2)
  }, [counter1])

  return (
    <div>
      <h1>Counter</h1>
      <div>counter1: {counter1}</div>
      <div>square of counter = {square}</div>

      <button onClick={() => setCounter1(counter1 + 1)}>
        update counter1
      </button>
    </div>
  )
}

```

```

        </button>
      </div>
    )
  }

  export default Counter

```

## VS Extensions

- Auto Import
  - <https://marketplace.visualstudio.com/items?itemName=NucleaR.vscode-extension-auto-import>
- Auto rename tag
  - <https://marketplace.visualstudio.com/items?itemName=formulahendry.auto-rename-tag>
- Code spell checker
  - <https://marketplace.visualstudio.com/items?itemName=streetsidesoftware.code-spell-checker>
- React snippets
  - <https://marketplace.visualstudio.com/items?itemName=rodrigovallades.es7-react-js-snippets>

## third party packages

- react router
  - used to add routing feature in react application
  - routing feature will enable the react application to switch between the components dynamically
  - provides router which does the routing
  - installation
    - `npm install react-router-dom`
  - BrowserRouter
    - used to add routing feature for website (browser)
    - must be the parent of in order to add the routing feature
  - Routes
    - collection of Route objects
    - contains all the possible routes of an application
  - Route
    - mapping between path (to launch the component) and element (component to be launched)

```

// main.jsx
import { createRoot } from 'react-dom/client'
import App from './App.jsx'
import { BrowserRouter } from 'react-router-dom'

createRoot(document.getElementById('root')).render(
  <BrowserRouter>
    <App />
  </BrowserRouter>
)

```

```

import { Route, Routes } from 'react-router-dom'
import './App.css'
import Login from './pages/Login/Login'
import Register from './pages/Register/Register'

function App() {
  return (
    <div>
      <Routes>
        <Route
          path='login'
          element={<Login />}
        />
        <Route
          path='register'
          element={<Register />}
        />
      </Routes>
    </div>
  )
}

export default App

```

- to navigate to another component

- static navigation

- there is no need to write any JS code to navigate to another component
- static navigation always loads the same component every time
- can be performed by using `<Link></Link>`

```

import { Link, Route, Routes } from 'react-router-dom'

import AboutUs from './pages/AboutUs/AboutUs'
import Home from './pages/Home/Home'

function App() {
  return (
    <div>
      <h1>App Component</h1>
      <ul>
        <li>
          <Link to='/home'>Home</Link>
        </li>
        <li>
          <Link to='/about-us'>About Us</Link>
        </li>
      </ul>
    </div>
  )
}

```

```

    <Routes>
      <Route
        path='home'
        element={<Home />}
      />
      <Route
        path='about-us'
        element={<AboutUs />}
      />
    </Routes>
  </div>
)
}

export default App

```

- dynamic navigation
  - navigation with JS code
  - also called as conditional navigation
  - can be achieved by a react hook named `useNavigate()`

```

import { Link, Route, Routes, useNavigate } from 'react-router-dom'

import AboutUs from './pages/AboutUs/AboutUs'
import Home from './pages/Home/Home'

function App() {
  // get reference of navigate function
  const navigate = useNavigate()

  const gotoHome = () => {
    // go to home
    navigate('/home')
  }

  const gotoAboutUs = () => {
    // go to about us page
    navigate('/about-us')
  }

  return (
    <div>
      <h1>App Component</h1>
      <button onClick={gotoHome}>Home</button>
      <button onClick={gotoAboutUs}>About Us</button>

      <Routes>
        <Route

```

```

        path='home'
        element={<Home />}
      />
      <Route
        path='about-us'
        element={<AboutUs />}
      />
    </Routes>
  </div>
)
}

export default App

```

◦ nested routing

- routing with parent-child relationship
- used to share contents with all child components
- `<Outlet />` is a placeholder to load the child component based on the route or path
- the child component can be loaded using the path format `parent-path/child-path`

```

// App.jsx
import { Link, Route, Routes, useNavigate } from 'react-router-dom'

import AboutUs from './pages/AboutUs/AboutUs'
import Home from './pages/Home/Home'

function App() {
  // get reference of navigate function
  const navigate = useNavigate()

  const gotoHome = () => {
    // go to home
    navigate('/home')
  }

  const gotoAboutUs = () => {
    // go to about us page
    navigate('/about-us')
  }

  return (
    <div>
      <h1>App Component</h1>
      <button onClick={gotoHome}>Home</button>
      <button onClick={gotoAboutUs}>About Us</button>

      <Routes>
        <Route
          path='home'

```

```

        element={<Home />}
      >
        <Route
          path='about-us'
          element={<AboutUs />}
        />
      </Route>
    </Routes>
  </div>
)
}

export default App

```

```

// Home.jsx
import React from 'react'
import { Outlet } from 'react-router-dom'

function Home() {
  return (
    <div>
      <h1>Home</h1>
      <Outlet />
    </div>
  )
}

export default Home

```

- react toastify
- used to show nice looking popups
- install
- npm install react-toastify
- code

```

// App.jsx
import React from 'react'

import { ToastContainer, toast } from 'react-toastify'

function App() {
  const notify = () => toast('Wow so easy!')

  return (
    <div>
      <button onClick={notify}>Notify!</button>

```

```
    <ToastContainer />
  </div>
)
}
```

- axios

- used to call the REST APIs
- install
  - npm install axios

- redux

- redux is a javascript library which is used for global state management
- a state which will be shared with all the components of an application will be managed by the redux library
- used in the application where considerable data/information need to be shared with all components
- it can be used in any JS application (react/angular/vue/vanilla JS)
- alternatives to redux
  - flux, ngrx (angular), zustand

- redux toolkit

- a wrapper library provided to implement redux global store easily in different frameworks
- uses redux behind the scene
- concepts
  - action
    - name of the reducer function which is used to invoke the reducer function
  - reducer
    - function which is used to perform some operation on a state stored inside a slice
  - store
    - collection of slices
    - used to store key-value pairs
- architecture
- installation
  - npm install @reduxjs/toolkit react-redux
  - yarn add @reduxjs/toolkit react-redux
- implementation

- create an empty global store

```
// src/store.jsx

import { configureStore } from '@reduxjs/toolkit'

// global store
export const store = configureStore({
  reducer: {},
})
```

- add the store inside the application, so the application can access the store for storing the global state
  - use Provider provided by react-redux to use the store for the entire application

```
// main.jsx

import { createRoot } from 'react-dom/client'
import App from './App.jsx'
import { Provider } from 'react-redux'
import { store } from './store.js'

createRoot(document.getElementById('root')).render(
  <Provider store={store}>
    <App />
  </Provider>
)
```

- add a slice to store a counter to the global store
  - slice is a part of store which is used to store a state member

```
// src/features/counterSlicer.js
import { createSlice } from '@reduxjs/toolkit'

// create a slice
const counterSlice = createSlice({
  name: 'counter',
  initialState: {
    value: 0,
  },
  reducers: {
    // increment the state value
    increment: (state) => {
      state.value += 1
    },
  },
})
```

```

    // decrement the state value
    decrement: (state) => {
      state.value -= 1
    },
  },
})

// export the counterSlice actions
export const { increment, decrement } = counterSlice.actions

// export default reducer object of counterSlice
// the reducer object contains the reducer function
// references
export default counterSlice.reducer

```

- add the slice to the store

```

// src/store.js

import { configureStore } from '@reduxjs/toolkit'
import counterSlice from './features/counterSlice'

// global store
export const store = configureStore({
  reducer: {
    // counterSlice's reducer is added to the global store
    counter: counterSlice,
  },
})

```

- read value from store (slice)

- useSelector()

- used to read a state of a slice from the global state
    - returns a state of selected slice

```

function Counter() {
  // the store has a slice named counter, and counter
  // slice has
  // initial state with key as 'value'
  const value = useSelector((store) =>
    store.counter.value)
}

```

- update the global store (slice)

- useDispatch()

- used to update state of selected slice
- returns an object of dispatcher which dispatches the action to the selected slice
- once the slice receives the action, it executes the required reducer