React ⚛️ 💗

☑️ Also known as React.js/ React.JS

Free & Open-source JavaScript library for building user interfaces (UIs).

Created by Meta (Facebook) & is now maintained by Meta & community of developers.

> HTML/CSS/JS code refreshes the whole page unexpectedly upon interaction (like clicking a button, submitting a form, etc.)

- React is used to build *SPAs (Single Page Applications)*
- Creates *reusable UI components*
- Develops dynamic, fast & scalable front end applications.

👉 ReactJS uses a Virtual DOM mechanism to efficiently update the HTML DOM.

Instead of reloading the entire DOM, the virtual DOM updates only the specific elements that have changed, resulting in faster performance.

🥷 React follows a Component-Driven Architecture

The user interface is built by composing independent, reusable pieces called components. Each component manages its own logic and rendering, making development more modular, scalable, and easier to maintain.

A Component is a reusable UI element.

React Element : an object that describes the properties of an actual DOM node which will be created by React.

Simple Structure of a React Elements:

```
{
    type: "button"
    props: {
        classname: "btn"
    }
}
```

✸ SPAs(Single Page Application)

A web application that loads a single HTML page and dynamically updates content as users interact with it is known as a Single Page Application (SPA).

In an SPA, the entire page doesn't reload with every interaction; only the necessary parts of the page are updated, providing a smoother and faster user experience.

- It uses client side rendering (CSR) [Javascript Updates the DOM]
- Communicates to server via APIs (e.g. REST, GraphQL etc.)

Example: Facebook, Gmail, Twitter

☑ Pros

- Faster Navigation (No full page reloads)
- Smoother UX
- Easier to build complex, interactive UIs
- Good for real time updates

❌ Cons

- Slower initial loads (Heavy JS Bundles)
- Search engines historically struggled with Javascript heavy SPAs. But solutions like Next.JS do help.
- Browser history management needs extra efforts (e.g React Router)

🌟 MPAs(Multi Page Application)

A traditional web app where each page is a seperate HTML document loaded from the server.

- It uses Server side rendering (SSR) [Server generats HTML for each request]

Example: Amazon, Wordpress, Old school sites

☑ Pros

- Better SEO out of the box as each page is a seperate HTML page.
- faster initial loads (Less Javascript code, server handles rendering)
- Easier to scale as pages can be cached independently.

❌ Cons

- Slower navigation (full page reloads upon each interaction)
- Less interactivity (more clunky UX)
- Harder to maintain state (e.g Shopping Cart across pages)

👉 Where to use SPA & MPA**

- If you need a dynamic, app alike experience you need to use SPA (e.g SaaS Tool, Social Network Site)
- If you prioritize SEO, Simplicity, Server-Driven content you need to use MPA (e.g Blogs, News etc.)

👓 Hybrid Approach

Modern frameworks like **Next.JS (React), Nuxt.JS (Vue)** allows **SSR + SPA** hybrid models for a better SEO & Optimization.

This approach is best for both cases.

🥷 React application is made up of Multiple Components

- Each of them are **responsible for outputting a small, reusable piece of HTML**.
- **Components can be nested within other components to allow complex applications** to be built out of simple building blocks.

👉 Key Features of React

1. **Component Based Architecture :** React apps are built using reusable components, making code easier to manage & scale
2. **Virtual DOM :** React uses a Virtual DOM to efficiently update only the parts of the page that change, improving performance.
3. **Declare Syntax :** Instead of manually updating the DOM like Vanilla JS, you describe how the UI should look & React handle updates automatically.
4. **Strong Ecosystem :** React has a massive community, tons of libraries (like Redux, Next.JS & Extensive documentation)
5. **Works with other frameworks :** React can be integrated with backend/mobile apps. (Node.JS, Django) [React Native]

🗃️ Library

A collection of pre-written functions/modules that you can call as needed. It only provides the boilerplate for the project.

Key Idea : You can control the application's flow & structure. The library will provide you utility.

Analogy : Like a toolbox, you'll pick which tools (functions) to use & when to use.

Example: React (UI Library), Loadash (Utility Function), JQuery (DOM Manipulation)

Charecteristics :

1. Flexibility : Uses only what I need
2. Less Opinionated : No Strict rules on application structure
3. More Responsibility : You decide how to integrate everything together.

🖼️ Framework

A fully featured structure that dictates how to build an application (includes library, tools, rules)

Key Idea : It controls the application's flow. The library will provide you utility.

Analogy : Like a blueprint, you build within it's rules.

Example: Angular, Django, Ruby on Rails

Charecteristics :

1. Built in tools like tools for routing, state management etc. is already included.
2. Scalability : Enforces best practices for large teams.
3. Less flexibility : Must follow the framework's convention.

👉 React A *Framework* or *Library*

React is a library (for UI components), but it's an ecosystem (React Router, Redux, Next.JS) can feel like a framework.

Next.JS is a framework built on React as it adds routing, SSR and other opinionated features.

➡️ Why does Library/Framework matter?

Libraries give you freedom but requires more decisions. While Frameworks speed up the development but limit flexibility.

- We use library if we need lightweight control (e.g React for dynamic UI)
- We use a framework if we want to Structure/Reliability (e.g Angular for Enterprise Applications)

Features	Library (e.g React)	Framework (e.g Angular)
Control Flow	You call the library	Framework calls your code
Flexibility	High (Pick & Choose)	Low (Follow Conventions)
Use Case	Add functionality to an app	Build fullstack apps
Learning Curve	Easier to Start	Steaper
Use Case	Lightweight	Often heavier

☑️ Virtual DOM

The Virtual DOM (VDOM) is a lightweight, in-memeory represenation of the real Document Object Model (DOM) used by the libraries like React to optimize UI updates.

It's a core reason why react is so fast.

Working Procedure

1. Initial Render : React creates a Virtual DOM tree (basically a JavaScript Object) that mirrors the real DOM.

HTML

```
<div>
    <h1> Hello React </h1>
</div>
```

React DOM

```
{type: 'div',
  props: {
    children: {
      type: 'h1',
      props: {
        children: "Hello React!"}}}}
```

Every HTML element becomes an object with:

- type: The tag name as a string
- props: An object containing all properties (including children)

The children prop is special:

- Can be a string (for text nodes)
- Can be another object (for nested elements)
- Can be an array (for multiple children)

2. State/Props Change : When data changes (e.g `setState`), React creates a new Virtual DOM tree.

3. Diffing (Reconciliation) : React compares the new VDOM with the previous DOM (Diffing Algorithm) to find out miimal changes that are needed.

4. Batch Update to Real DOM : Only the changed parts are updated in real DOM (avoiding full renders, which is time & memory consuming)

Why do we use Virtual DOM?

- It minimizes direct DOM manipulations (slowest parts of the web application)
- Batches multiple updates into a single render cycle.
- Cross Platform : React Native can also use similar concepts for making mobile UIs

Features	Virtual DOM	Real DOM
Speed	Fast (JS Objects)	Slow (Browser regenerates code everytime)
Updates	Batched & optimized	Immediate & memory costly
Memory	Lightweight (Less memory)	Heavy (Browser Rendered)

👉 Reconciliation

React's algorithm for efficiently upgrading the DOM when state/props change.

- It's the diffing process that compares the new Virtual DOM with the previous one to determine the minimal changes needed for the real DOM.

➡️ How does it work?

- Trigger : A component's state/props change
- New Virtual DOM tree : React creates a new virtual DOM tree.
- Diffing : React compares the new tree with the old one. (reconciliation)
- Update : Only the changed parts are applied to the real DOM.

🔁 Reconciliation v/s Virtual DOM?

Virtual DOM is a lightweight copy of real DOM while Reconciliation is an algorithm that compares Virtual DOM snapshot with the old DOM to update the real DOM efficiently.

- Reconciliation minimizes costly DOM operations.
- Gives smooth UI & updates feel instantaneous.
- Complex diffing can slow down a lot of large scale applications.

👉 JSX

JavaScript XML/ JSX is a syntax extension for JavaScript, primarily used with React to describe UI components.

Till now we've seperately been using HTML, CSS, JS

Through JSX we can write HTML like code in JavaScript, which later on will be transformed into JavaScript code by React library.

- JSX & React are different. We can write React without JSX.

React.createElement => Object => HTMLElement(render)

```
const Heading = React.createElement ("h1", {id: "heading"}, "This is React")
```

JSX - HTML like/ XML like syntax

```
const jsxHeading = <h1 id="heading"> This is React </h1>;
const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(heading);
```

As a better example:

```
<!DOCTYPE html>
<html>
  <body>
    <div id="root"></div>

    <script src="https://unpkg.com/react@18.3.1/umd/react.development.js">
```

```
</script>
    <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js">
</script>

    <script>
      function App() {
        return React.createElement(
          "div",
          {},
          React.createElement("h1", {}, "Hello World")
        );
      }

      const rootContainer = document.getElementById("root");
      const root = ReactDOM.createRoot(rootContainer);
      root.render(App());
    </script>
  </body>
</html>
```

- A basic HTML element with `id="root"` — this is the mount point for the React app.
- `App` is a React component. It returns a React element (`<div><h1>Hello World</h1></div>`) using `React.createElement`.
- No JSX is used here — it's all vanilla JS.
- `rootContainer` gets the `div#root`.
- `ReactDOM.createRoot()` creates a root for the React app (React 18 style).
- `root.render(App())`; renders the result of calling the App function.

JSX is not pure JS as JS engine dosen't understand JSX. JSX code gets transpilled (converted into the code browser can understand) prior going into the JSX engine.

Component Structure + JSX Structure:

```
import React from 'react';

function MyComponent() {
    return (
        <div>
            <h1>Hello World!</h1>
            <p>This is Pratik Speaking</p>
        </div>
    );
}

export default MyComponent;
```

In XML we can create our custom tags which we call user defined tags

JSX => React.createElement => ReactElement- JS Object => HTML Element (at the time of Render)

We need to use cammel case while declaring attributes in JSX.

Key Features of JSX

- Embeded Expressions : You can embed JS expressions inside JSX using `{}`

```
const name = 'Alice'
const greeting = <h1> Hello, {name}!</h1>
```

- Self closing tags like `<img/>`
- JSX use `className` instead of `class`
- Inline CSS styles are written in Javascript objects:

```
<div style = {{color: 'red', fontSize: '20px'}}>
        Styled
    </div>
```

- We can use React components like HTML tags:

```
function Welcome (props){
    return <h1> Hello, {props.name} </h1>
}
```

How JSX Works:

- Welcome is a functional component that accepts props as its parameter
- It returns JSX that renders an `<h1>` heading
- The component uses the name props inside curly braces `{}` to display dynamic content.

Using the Component:

```
const element = <Welcome name="John Doe" />;
```

What Happens When This Renders:

- React creates an instance of the Welcome component
- It passes the props object { name: 'John Doe' } to the component
- The component returns: `<h1>Hello, John Doe</h1>`
- This JSX gets converted to actual DOM elements

React converts this JSX to something like:

```
{type: Welcome,
  props: {
    name: 'John Doe'}}
```

Why Use JSX

- It's easier to visualize UI structure.
- It's optimized to React.

> Babel compiles JSX into `React.createElement()` calls:

JSX:

```
<div id='app'>
    Hello World!
</div>
```

JavaScript:
`React.createElement("div", {id : "app"}, "Hello World")`

Babel is a JavaScript transpiler. Transpiles modern JavaScript (ES6+) into backward-compatible versions (ES5)

Converts newer JS syntax into older syntax that can run in older browsers. (Different developers writting different versions of JavaScript, henceforth Babel is used)

Example: Arrow functions () => {} → function() {}

☑ Installing React Boilerplate through `npm`

What's NPM?

NPM is a Node Package Manager. It's the default package manager for the Node.js runtime environment. `npm` dosen't have a full form. Here's a quick overview:

◇ NPM is a tool to manage JavaScript packages/libraries.

- Used to install, share, and version-control packages.
- Comes pre-installed with Node.js.
- It's a package manager but it's abbreviation is not node package manager.

It comes bundled with Node.js and is the default tool for managing dependencies in JavaScript/Node.js projects.

- World's largest software registry and package manager for JavaScript. A standard repository for all the packages.

`npm` Commands Cheat Sheet

Essential npm commands for JavaScript/React development.

📦 Package Installation

Command	Description
`npm install`	Install all dependencies from `package.json`
`npm install <package>`	Install a specific package (e.g., `npm install react`)
`npm install <package> --save-dev`	Install as dev dependency
`npm install -g <package>`	Install globally
`npm install <package>@<version>`	Install specific version

🚀 Development & Scripts

Command	Description
`npm start`	Start development server
`npm run build`	Create production build
`npm test`	Run tests
`npm run <script>`	Run custom script
`npm run dev`	Start dev mode (Vite/Next.js)

🛠️ Project Setup

Command	Description
`npm init`	Create new `package.json`
`npm init -y`	Quick init with defaults
`npm init <initializer>`	Scaffold project (e.g., `npm create vite@latest`)

- npm uses `package.json` & `node_modules`
- All the configuration will be inside `package.json` file for the project which is a configuration for `npm`.
- Our projects are dependent on a lot of packages. These packages are called dependencies to the project. `npm` manages these packages.

Bundler

A bundler helps you to clean your code prior sending to production. It bundles/packs your app so it can be sent to production.

Example : *webpack,vite,parcel etc.*
-> `create-React-app` uses babel & webpack.

◇ **Why Do We Need Bundlers?**

- Dependency Management: It combines imported modules into a single file.
- Resolves `import`/`require` statements.
- Code Optimization: Minifies, compresses, and tree-shakes (removes unused code).
- Improves load time.
- Supports TypeScript, SCSS, and modern JS (ES6+).

◇ **Bundler vs. Compiler vs. Transpiler**

Tool	Purpose
Bundler	Combines files, manages dependencies
Compiler	Converts code to machine code (e.g., TypeScript → JS)
Transpiler	Converts modern JS to older syntax (e.g., Babel)

◇ **When to Use a Bundler?**

- SPAs (React, Vue, Angular)
- Libraries (Publishing to npm)
- Legacy Browser Support (via polyfills)

◇ **Popular JavaScript Bundlers**

Bundler	Key Features	Used With
Webpack	Highly configurable, plugin system	React, Vue, Angular
Vite	Ultra-fast (ESM + Rollup), HMR	Next.js, Svelte
Parcel	Zero-config, fast builds	Small to medium apps

Example: `npm install -D parcel` [Dev dependency installed of parcel bundler]

🤖 Installing React Boilerplate through `Vite` Npm

1. `Installing gitbash`
2. `npm create vite@latest`
3. Need to install the following packages:
   `create-vite@6.5.0 Ok to proceed? (y) Y`

```
> npx
> create-vite

│
◇  Project name:
│  Pratik-Portfolio
│
◇  Package name:
```

```
│  pratik-portfolio
│
◇  Select a framework:
│  React
│
◇  Select a variant:
│  JavaScript
│
◇  Scaffolding project in C:\A Storage\02. Front End Tech\LiveCohort\Pratik-
Portfolio...
│
└  Done.
```

Now we need to install NPM. I've boilerplate installed but I need to code that'll be used in boilerplate :

```
Now run:

  cd Pratik-Portfolio
  npm install
  npm run dev

PS C:\A Storage\02. Front End Tech\LiveCohort> cd Pratik-Portfolio
>>   npm install
>>   npm run dev

added 152 packages, and audited 153 packages in 47s

33 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

> pratik-portfolio@0.0.0 dev
> vite
```

Upon Each Command on Terminal :

- `npm install` -> will go to `src/package.json` [Registry File]
- this file will consist of **dependencies & dev-dependencies**
- These two files will check what are the things inside them. It'll install the related code from there.
- After running the `npm install` we'll get new files `node_modules` inside main file.

Inside `package.json` file there will be info about your library/packages which are installed inside `node_modules`.

Now to run our project: `npm run dev`

package.json

- It stores all the metadata regarding the project we're working on.

```
"name": "/* project name  */"
"type": " module/commonjs "             /*  ES6/CommonJS  */

"scripts" : {               /*  We can chnage & edit these scripts  */
  "dev": "vite",           [npm run `dev`] → runs the command vite → vite runs
the algo that starts local server
  "build": "vite build"   [npm run `build`] → build project
}
```

☑ Frontend Tool `vite` also provides us a JS package alongside & if we want to execute a package in my terminal ➡ `npx vite` ➡ Localhost starts

Code Moduling

Breaking the whole codebase into small & reusable code modules.

There are two types of moduling setups -

- Commonjs
- ES6 Moduling

Now inside `package.json` → we find `"scripts"` where we see `"dev" : "vite"`

- *From this section we get the command* `npm run dev`.
- We can also use `npx vite` to run our localhost.

`npx` is used to execute Node.js packages directly from the terminal, without needing to install them globally.

☑ CommonJS vs ES6 Modules

Feature	CommonJS	ES6 Modules
Syntax	`require()` / `module.exports`	`import` / `export`
Usage	Node.js (pre-ES modules)	Modern JS (browser + Node.js)
TypeScript	Supported, but with compatibility quirks	Native support preferred

Using CommonJS for Type Moduling

```
const dummyUser = {
  id: 1,
  name: "Pratik",
  email: "pratik@example.com"
};
```

```
module.exports = { dummyUser };
```

```
// Import using CommonJS
const { dummyUser } = require('../types/user');

/**
 * @param {import('../types/user').User} user
 */
function showUser(user) {
  console.log(`Hello, ${user.name}`);
}

showUser(dummyUser);
```

📊 CommonJS Export & Import Syntax Comparison Chart

Feature	CommonJS Syntax	ES6 Module Syntax	Notes
Default Export	`module.exports = myFunction;`	`export default myFunction;`	You export one main value/function
Import Default	`const myFunction = require('./file');`	`import myFunction from './file.js';`	`require()` grabs the whole export
Named Export	`exports.myFunc = myFunction;` *or* `module.exports = { myFunc };`	`export const myFunc = () => {}`	Exports as properties of an object
Import Named	`const { myFunc } = require('./file');`	`import { myFunc } from './file.js';`	Destructure the returned object
Both (Mix)	❌ Not recommended in CommonJS	☑ Allowed (`export default` + `named`)	CommonJS doesn't cleanly support mixing

Default Export (CommonJS)

file: greet.js

```
function greet(name) {
  return `Hello, ${name}`;
}

module.exports = greet; // 👈 Default export
```

file: app.js

```
const greet = require('./greet'); // 👈 Default import
console.log(greet('Pratik'));
```

Named Exports (CommonJS)

file: math.js

```
const add = (a, b) => a + b;
const sub = (a, b) => a - b;

// Option 1
exports.add = add;
exports.sub = sub;

// OR Option 2 (preferred for cleaner syntax)
module.exports = { add, sub };
```

file: app.js

```
const { add, sub } = require('./math');
console.log(add(2, 3)); // 5
```

🔄 Dependencies

Dependencies are external packages/libraries that your project relies on to function properly.

Types of Dependencies

1. **Regular Dependencies (dependencies)**
   Packages required for your application to run in production

- Installed with `npm install <package>`
- Listed in `package.json` under `dependencies`

Example: json

```
"dependencies": {
  "react": "^18.2.0",
  "express": "^4.18.2"}
```

2. **Development Dependencies (devDependencies)**
   Packages only needed during development (testing, building, etc.)

- Installed with `npm install <package> --save-dev`
- Not neede while production
- Not included in production builds

Example: json

```
"devDependencies": {
  "jest": "^29.5.0",
  "webpack": "^5.76.0"}
```

npm uses Semantic Versioning (SemVer):

- `^1.2.3`: [Caret] Allow patch and minor updates (1.x.x) [Safe]
- `~1.2.3`: [Tilde] Allow only patch updates (1.2.x)
- `1.2.3`: Exact version
- `*`: Latest version (not recommended)

🕵️ `package-lock.json`

The Dependency Version Lockfile. Keeps the record of every version of the dependencies/packages that are getting installed (including nested sub-dependencies).

📥 Intergrity

Inside `package-lock.json` if we search "node_modules/react" we'll see an `"integrity"` key.

It's an unique fingerprint (checksum) of the package's contents. Generated using SHA-512 (common) or SHA-1 (older).

Integrity hash ensures that the package installed on your machine is bit-for-bit identical to the one deployed in production (or used by your teammates).

Format : json

```
"integrity": "sha512-9a1b2c3d4e5f6...=="
```

- `sha512`: Hashing algorithm
- `9a1b2c3d...`: The actual hash digest.

📝 Transitive Dependencies

When your project depends on `vite`, and `vite` itself depends on other packages (which may depend on even more packages), this creates a chain called **transitive dependencies (or dependency tree)**.

🌳 Dependency Tree Example

```
your-project
└── parcel@2.9.3 (direct dependency)
    ├── @parcel/core@2.9.3
    │   ├── @parcel/fs@2.9.3
    │   └── @parcel/utils@2.9.3
    └── postcss@8.4.21
        └── nanoid@3.3.6 (transitive dependency)
```

- Every packages have it's own dependency hence for that there'll be seperate 'package.json' files inside it's folder in `node_modules`.
- Every package in `node_modules` has its own `package.json`, defining its dependencies, scripts, and metadata.

☑ Every package manages its own dependencies (via its `package.json`).

☑ `node_modules` can be nested or flat (depends on npm/yarn/pnpm).

☑ Conflicts occur if two packages need incompatible versions of the same dependency.

☑ Use `npm ls <package>` to see where a dependency is installed.

We don't push all of our code in the production/github. We'll put some of the code into `.gitignore`.

- If there's not a file we should create it & inside the file type `\node_modules`
- If we've a `package.json` & `package-lock.json` file (outside) we can recreate `node_modules` again. (prompt: `npm install`) That's why it's not important to push `node_modules` inside github.
- Files like `node_modules/` are recreated when you `run npm install` or `yarn install`.
- Since these files are recreated identically on any machine (or server), storing them in `Git` is redundant.

Configure Browsers List

`browserslist` configuration, typically found in `package.json` or a `.browserslistrc` file.

- json

```
"browserslist": [
    "last 2 versions"]
```

❓ Questions

Q: Why should I not modify `package-lock.json`?

A: It is a generated file and is not designed to be manually edited. Its purpose is to track the entire tree of dependencies (including dependencies of dependencies) and the exact version of each dependency. You

should commit `package-lock.json` to your code repository

You should avoid updating the `package.json` manually since it could break the synchronization between `package.json` and `package-lock. json`.

Q: What is `node_modules`? Is it a good idea to push that on git?

A: The `node_modules` folder contains generated code. This is not code you've written and you should never make any updates to the files inside Node modules because there's a pretty good chance they'll get overwritten next time you install some modules.

It is better to not commit the `node_modules` folder, and instead add it to your `.gitignore` file.

Here are all the reasons why you shouldn't commit it: The node_modules folder has a massive size (up to Gigabytes). It is easy to recreate the node_modules folder via packages. json

Q: What is the `dist` folder?

A: The `/dist` stands for distributable. The /dist folder contains the minimized version of the source code. The code present in the /dist folder is actually the code which is used on production web applications.

Parcel's default directory for your output is named dist . The --dist-dir public tag defines the output folder for your production files and is named public to avoid confusion with the dist default directory.

Q: What is `browserlists`?

A: Browserslist defines and shares the list of target browsers between various frontend build tools.

👉 How to create an NPM Script?

> NPM Scripts are needed to build our project. We'll be creating the script in our `package.json` file.

- We can use these scripts for multiple works. (e.g Start our project in Dev Mode, production ready application)

let's create a script for starting our project in Dev Mode.
inside `package.json`

```
"scripts": {
    "test": "jest",
    "start": "parcel index.html"}
```

This script will be used for dev build : `"start": "parcel index.html"`

To run these scripts : `npm run <script_name>` [`npm run start`]

💝 What is `npx`?

`npx` stands for Node Package Executer.

It's a tool that comes bundled with NPM (v5.2.0 and above), used to execute Node packages directly without installing them globally.

🔧 Why Use npx?

- Run packages without global install
- Avoid cluttering your system with global packages
- Always runs the latest version
- Great for running CLI tools once or temporarily

📂 Why we don't push automatically created files like `node_modules/parcel-cache/dist` to the `git`?

Reason we don't push automatically created files like `node_modules/parcel-cache/dist` to the `git` as all the commands that we run like `npx/npm`, we're running them in `server`. Hence the server will be automatically creating the deleted files once again which will be similar to the previously generated local files in our code editor.

🥷 JSON (JavaScript Object Notation)

A lightweight data interchange format that's easy for humans to read and write, and easy for machines to parse and generate.

◇ JSON Structure Basics:

- Data is in key-value pairs
- Data is separated by commas
- Curly braces {} hold objects
- Square brackets [] hold arrays

👉 Import & Export

abc.jsx	**xyz.jsx**
`let a = 5`	I want to use `a` from abc.jsx
	We need to make it export ready to send to xyz.jsx
`export default a`	`import a from abc.jsx` [Path]
We can use `default export` once in a file.	Now `a+5=10`
	while we import we can change the name of the variable.
	`import b from abc.jsx` [b=5]
We can use multiple `export const` in a single file.	

x.jsx	**actions**	**y.jsx**

x.jsx	actions	y.jsx
`a=5, b=6, c=7`	Import	`import {e,f,g} from x.jsx`
export const e=a	Export	
export const f=b		
export const g=c		

At the times of `import` we've to import the `export const` in a curly bracket {} & the name of the variables must be same.

👉 Folder Updatation

- `public` folder's files will be accessed globally.
- We need to delete `assets`, `app.css`, `index.css` files.
- Delete all codes from `app.jsx`.
- Delete `import './index.html'` from `main.jsx`.
- Put `rafce` (ReactArrowFunctionComponentExport) [in app.jsx]

```
const App() => {
    return <div>Application</div>
}
export default app;
```

Imports will be done in `main.jsx`

```
import Abc from "app.jsx"
```

We can change the name to something else but the file that we default exported remians same.

```
import {x} from "./App.jsx"
```

Important VS Code Extension : ES7 + React/Redux/React-Native...

Inside `main.jsx`

```
createRoot(document.getElementById("root").render(
  <StrictMode>
    <App/> -> Whatever we've written in app.jsx
  </StrictMode>
))
```

`createRoot(document.getElementById("root")).render(<App />)`

- `createRoot` - Lets you create a root to display React components inside a browser DOM mode.
- `document.getElementById("root")` - Selects the div id `root` from HTML.
- `.render(<App />)` - This self closing tag renders the React functional components of `App.jsx` inside root.

`<App />` is an XML creates user defined tags.

➡️ Inside `App.jsx`

```
const App = () => {
  return <div>App</div>
}
export default App;
```

- As you can see we've passed an annoymous arrow function which is passed in `const App`.
- Through `export default` now the function component is ready to get rendered inside `root`.

👉 React Functional Components

Everything in a React is a **Component.** Breaking the whole codebase in smaller pieces of code.

Class Based Component (Old Way)

Functional Component (New Way):
Functional components are the modern way to write React components using JavaScript functions (often with arrow functions).

A JavaScript Function which returns a React Element is called Functional Element. It will always return HTML.

```
const HeadingComp = () => {
  <div id="container">
    return <h1 className="heading">Pratik Das It's Calling</h1>
  </div>}
```

React Element Rendering & **React Component Rendering** are not same.

- React Component rendering : `root.render(<HeadingComponent />)`
- React Element rendering : `root.render(parent)`

Now, suppose there are two **Function components** like this:

```
const HeadingComp = () =>
  <div id="container">
    <h1 className="heading">Pratik Das It's Calling</h1>
  </div>
```

```
  const HeadingComp2 = () =>
  <div id="container">
    **<HeadingComp/>**
    <h1 className="heading">Pratik Das It's Him</h1>
  </div>
```

If I want to now render my `HeadingComp` component inside `HeadingComp2` container I need to add `<HeadingComp/>` inside the container.

All the code of `HeadingComp` will come to `HeadingComp2` & will render in HTML if we use `<HeadingComp/>`

Component Composition

Basically composing two components inside one another. A fundamental React pattern through which we'll build complex UIs by combining smaller, reusable components.

If I use any direct function I've to use `return`.

```
const HeadingComp2 = function() {
  return(
  <div id="container">
    <h1 className="heading">Pratik Das It's Him</h1>
  </div>)}
```

If I had to put a React element inside React component/use normal JS code inside a React Component I can write it like this.

```
const num = 10000

const HeadingComp = () =>
  <div id="container">
    {ANY JAVASCRIPT CODE}**********
    <h1>{num}</h1>
    <h1 className="heading">Pratik Das It's Calling</h1>
  </div>
```

☑ Any Code Written after `return` won't work. The code will be unreachable.
Example:

```
const App = () => {
  return <div> Pratik </div>
  console.log ("Hello World")     // will give us an error
}
```

☑ We can only return single data/entity/variable/value.

```
const App = () => {
  return <div> Pratik </div> <div> Pratik </div> <div> Pratik </div>
  // will give us an error
}
```

Instead we can make a parent div & put all of the the divs inside it. But this div has no function except wrapping up the divs.

```
const App = () => {
    return <div>
              <div>App</div>
              <div>School</div>
           </div>;
}
```

➡ React Fragment Tag

To improve the issue of wrapping up the divs inside one main div, we can wrap these divs using the `<Fragment></Fragment>` tag from React. This tag doesn't appear in the Chrome Elements panel — only the `root` element will be visible, along with the `App` and `School` divs.

```
const App = () => {
    return <Fragment>
              <div>App</div>
              <div>School</div>
           </Fragment>;
}
```

Now this Fragment tag can be written in this way also. We call it empty tags also -

```
const App = () => {
    return <>
              <div>App</div>
              <div>School</div>
           </>;
}
```

☑ **A function should have a single return statement, and it must be the last statement in the function.**

More JSX

If we want to describe or return our component in multiple lines, we use parentheses () to wrap the JSX.

```
const MyComponent = () => (
  <div>
    <h1>Hello</h1>
    <p>This is a multi-line component.</p>
  </div>
);
```

Props

In React, props (short for "properties") are a fundamental concept for passing data between components.

What are Props?

Read-only data passed from parent to child components. Similar to function arguments in JavaScript. Enable component reusability with dynamic data

childern + attribute that we pass

Goal
You have a `button`, and instead of hardcoding its text or style every time, you want to reuse it with different:

- text (e.g., "Buy Now", "Add to Cart", "Learn More")
- colors
- `onClick` actions

Example
Reusable Button Component (Button.js)

```
function Button(props){    // Object
  return(
    <button
      onClick={props.onClick}
      style={{
        backgroundColor: props.bgColor,
        border: "none",
        borderRadius: "5px",
        color: props.textColor
      }}
    >
    {props.label}
    </button>
  )
}
```

```
export default Button;
```

🐞 Using the Button Component:

```
import Button from './Button';

function App() {
  const handleClick = () => {
    alert("Button clicked!")
  }

  return (
    <div>
      <Button
        label="Buy Now"
        bgColor="#ff4757"
        textColor="#fff"
        onClick={handleClick}
      />

      <Button
        label="Learn More"
        bgColor="#1e90ff"
        textColor="#fff"
        onClick={() => console.log("Learn More clicked")}
      />
    </div>
  );
  }
```

When you use a component in JSX, you pass props like this

```
<Button label="Buy Now" bgColor="red" />
```

This is exactly like saying:

```
props = {
  label: "Buy Now",
  bgColor: "red"
}
```

So each `key=value` becomes a property inside the props object.

📥 Event Listener & Event Handling in React

Event handling in React is the process of responding to user interactions like:

- Clicking a button 🖱️
- Typing into an input field ⌨️
- Submitting a form 📄
- Hovering over an element 🖥️

Instead of traditional addEventListener, React allows us to attach events directly in the JSX using camelCase syntax.

```
const App = () => {
  const handleclick = () => {
    alert ("Button Clicked!")
  }

    return (<>
              <div>App</div>
              <div>School</div>
              <button onClick={handleClick}>Click Here</button>
          </>);
}
```

- The event is named onClick (camelCase), not onclick.
- You pass a function reference `{handleClick}`, not a function call. `{handleClick( )}`

Now the previous example is for **Non-Parameterized function**. If the function is parameterized like this -

```
const App = () =>{
  const handleClick = (message) =>{
    alert (message)
  }
}

return (<>
  <button
  onClick={handleClick("Kolkata")}
  >
    Click Here
  </button>
</>)
```

We must pass the argument when using a **parameterized function** in React, because React doesn't automatically know what argument to pass into your custom function.

But in this case without user's interaction the parameterized function will run immediately.
So instead we can add a wrapper handler function (fake function) so it doesn't run immediately — it only

runs when user interacts.

```
<button onClick={() => handleClick("Kolkata")}>Click (param)</button>
```

☑ The Container Presentation Pattern

In React it's is a design pattern that helps separate presentation logic from business (or container) logic, making components cleaner, reusable, and easier to test.

In the Presenter Pattern, a component is split into:

- Container (Smart) Component: Handles business logic, data fetching, and state.
- Presenter (Dumb/UI) Component: Focuses only on rendering UI based on props.

🎮 Hangman Game

- Add a New Folder named `components` inside `src/components`
- Another folder inside that named Button → `src/components/Button`
- Another file named `Button.jsx`
- Component Function name should start with an uppercase letter.
- Next we'll be defining a function named Button →

```
const Button = () => {
  return (
    <button>
    </button>
  );
};

export default Button;
```

- Now we want to put inputs in our component through parameters. `(props)`

```
const Button = (props) => {
return (
    <button>
      {props.text}        // JSX Curlies {} → Evaluates Valid JS Expression
    </button>
  );}
```

- After specifying the name of the props by destructuring -

```
const Button = ({ text }) => {
  return (
```

```
    <button>
      {text}
    </button>
  );
};
```

- Now we want to define how each of the button will work on clicking. We're also gonna evaluate the props that we're passing though `onClick={onClickHandler}`

```
const Button ({text, onClickHandler}){
  return(
    <>
      <button
      onClick={onClickHandler}
      >
        {text}
      </button>
    </>
  )
}
```

- in the `App.jsx` We'll pass `onClickHandler` & inside we'll call an annonymous function that'll return the event that'll get performed on clicking of the button.

```
return (
    <>
      <Button text="Click me" onClickHandler={() => console.log("Button
clicked")}/>
    </>
  )
```

- Adding style after evaluation. First {} is to pass objects in style & second {} to evaluate

```
    <button
    onClick={onClickHandler}
    style={{
      backgroundColor: 'blue',
      color: 'white',
      padding: '10px 20px',
      border: 'none',
      borderRadius: '5px',
      cursor: 'pointer'
    }}
```

- After TailwindCSS

```
<button
    onClick={onClickHandler}
    className="bg-blue-500 text-white py-2 px-4 border-none rounded cursor-
pointer"
    >
      {text}
    </button>
```

- Now we need to make the button more reusable. So we'll be passing a props named `styleType` & default value is set to `primary`. We've also added a switch case function `getButtonStyling` where we're passing the `styleType` as props to check the colour of the button based on the value we give.

☑ Using default parameter syntax means you don't have to explicitly pass `primary` every time.

Dynamic Styling with `getButtonStyling()`
This function takes styletype and returns the appropriate Tailwind CSS class for that style.

You're switching over different styleType values like `primary`, `secondary`, `error`, etc.

```
const Button = ({ text, onClickHandler, styletype = "primary" }) => {
  return (
    <button
    onClick={onClickHandler}
    className={`${getButtonStyling(styletype)} text-white py-2 px-4 border-none
rounded cursor-pointer`}
    >
      {text}
    </button>
  );
};

function getButtonStyling(styletype) {
  switch (styletype) {
    case "primary":
      return "bg-blue-500 text-white";
    case "secondary":
      return "bg-gray-500 text-white";
    case "error":
      return "bg-red-500 text-white";
    case "success":
      return "bg-green-500 text-white";
    case "warning":
      return "bg-yellow-500 text-white";
    default:
      return "bg-blue-500 text-white";
  }
}
```

- **Single Responsibility Principle :** In React, the Single Responsibility Principle (SRP) means that each component should have one clear job and handle only one reason to change.

Key Idea

A React component should:

Do one thing (render a part of the UI, handle a specific piece of logic, etc.).

Change for one reason only (e.g., UI structure changes, not because of unrelated state changes).

```
Create a new file named `getButtonStyling.js` & add `function
getButtonStyling(styletype)`
```

- Now we'll make a dedicated folder named `pages`
- We'll add the `StartGame.jsx` page.

```
const StartGame = () => {
  return (
    <div>
      <h1>Welcome to Hangman!</h1>
      <p>Get ready to guess the word!</p>
    </div>
  );
};

export default StartGame;
```

- Also will create another folder in components named `TextInput` & will add `TextInput.jsx`

```
const TextInput = ({ type ="text", label, onChangeHandler, placeholder = "Enter
Your Input Here" }) => {
  return (
    <label>
      <span className="text-gray-700">{label}</span>
      <input
        type={type}
        className="border border-gray-500 px-4 py-2 rounded-md w-full"
        onChange={onChangeHandler}
        placeholder={placeholder}
      />
    </label>
  );
};

export default TextInput;
```

- props are passed `type ="text"` [default value text], `label`, `onChangeHandler`, `placeholder = "Enter Your Input Here"` [default value]
- Now we're gonna see the `App.jsx` structure where we'll be declaring all the values:

```
import TextInput from './components/TextInput/TextInput'
import './App.css'
import Button from './components/Button/Button'

function App() {
  return (
    <>
      <Button text="Click me" onClickHandler={() => console.log("Button
clicked")} styletype='primary'/>
      <Button text="Click me" onClickHandler={() => console.log("Warning
clicked")} styletype="warning"/>
      <Button text="Click me" onClickHandler={() => console.log("Error
clicked")} styletype="error"/>
      <Button text="Click me" onClickHandler={() => console.log("Success
clicked")} styletype="success"/>

      <TextInput label="Your Guess" onChangeHandler={(e) =>
console.log(e.target.value)} />
    </>
  )
}
export default App
```

- Now I Want - A password text field, A Show / Hide toggle button, A Submit button. I'll make a new folder inside component - `TextInputForm` & will create `TextInputForm.jsx`

```
import TextInput from "../TextInput/TextInput";
import Button from "../Button/Button";

const TextInputForm = () => {
  return (
    <form>
        <div className="mb-4">
          <TextInput
          label="Enter Your Word/Phrase Here"
          placeholder="Your Word/Phrase"
          onChangeHandler={(e) => console.log(e.target.value)} />
        </div>

        <div>
            <Button
            styletype="warning"
            text="Show/Hide"
            onClickHandler={() => console.log("Show/Hide clicked")} />
        </div>
```

```
        <div>
            <Button
            type="submit"
            styleType="primary"
            text="Submit"
            onClickHandler={() => console.log("Form submitted")} />
        </div>
    </form>
  );
};
```

- Now to prevent the default behaviour of the whole form getting refreshed we'll add a function & pass that in Form.

```
const TextInputForm = () => {
  const handleFormSubmit = (e) => {
    e.preventDefault();
    console.log("Form submitted");
  };

  return (
    <form onSubmit={handleFormSubmit}>...)...}
```

- Also We'll add another props to target & fetch the input that we're putting inside the Input box

```
const handleTextInputChange = (e) => {
    console.log("Text input changed:", e.target.value);
  };

  return (
    <form onSubmit={handleFormSubmit}>
        <div className="mb-4">
          <TextInput
          label="Enter Your Word/Phrase Here"
          placeholder="Your Word/Phrase"
          onChangeHandler={handleTextInputChange} />
        </div>...)
```

- Now our target is to seperate the Logical layer with the UI. We'll be shifting the `const TextInputForm` & `const handleTextInputChange` to a new file named `TextInputFormContainer.jsx`.

Also we'll be passing these attributes as props.

```
import TextInputForm from "./TextInputForm";

const TextInputFormContainer = () => {
    const handleFormSubmit = (e) => {
    e.preventDefault();
    console.log("Form submitted");
  };

  const handleTextInputChange = (e) => {
    console.log("Text input changed:", e.target.value);
  };
  return (
    <div>
      <TextInputForm
        handleFormSubmit={handleFormSubmit}
        handleTextInputChange={handleTextInputChange}
      />
    </div>
  );
};
```

- Now we'll be changing the `Show/Hide` button on click & it'll be interacting & changing UI in Input Box.
- Inside `TextInputForm.jsx` we'll be adding a props named `handleShowHideClick`

```
  <div>
    <Button
      styletype="warning"
      text="Show/Hide"
      onClickHandler={handleShowHideClick} />
  </div>
```

- Just like rest of the two functions we'll add another function in `TextInputFormContainer.jsx`

```
const handleShowHideClick = () => {
    console.log("Show/Hide clicked");
  };

  return (
    <div>
      <TextInputForm
        handleFormSubmit={handleFormSubmit}
        handleTextInputChange={handleTextInputChange}
        handleShowHideClick={handleShowHideClick}
      />
    </div>
  );
```

- Now we'll be passing a prop named `inputType` in `TextInputForm.jsx`.

```
const TextInputForm = ({inputType, handleFormSubmit, handleTextInputChange,
handleShowHideClick }) => {
  return (
    <form onSubmit={handleFormSubmit}>
        <div className="mb-4">
          <TextInput
          type={inputType}
          label="Enter Your Word/Phrase Here"
          placeholder="Your Word/Phrase"
          onChangeHandler={handleTextInputChange} />
        </div>...)...}
```

- While calling it through `TextInputFormContainer.jsx` we'll put the default value as `type ="text"`

```
return (
    <div>
      <TextInputForm
        inputType="text"
        handleFormSubmit={handleFormSubmit}
        handleTextInputChange={handleTextInputChange}
        handleShowHideClick={handleShowHideClick}
      />
    </div>
  );
```

- Now I'll make a variable named `inputType` & will assign the value `"passowrd"` to it. i'll also call that in `TextInputForm`. Alongside Will run a if else loop that'll toggle based on the current state.

```
const TextInputFormContainer = () => {

    let inputType = "password";...
    ...
    const handleShowHideClick = () => {
    console.log("Show/Hide clicked");
    if (inputType === "password") {
        inputType = "text";
    } else {
        inputType = "password";
    }
  };

  return (
    <div>
```

```
        <TextInputForm
          inputType={inputType}
          ...)}
```

If we write variables in this way in our components, React dosen't track the changes after interactions. It guesses that we're using these variables for our own usage.

Also when I change variable React calls the whole function & inside that it again gets `let inputType = "password"`

Now we'll be making a type of varibale that React will track. upon changes React will manage the memeory. We'll calling these `state variables`

React Hooks

Special functions introduced in React 16.8 that let you use state and other React features in functional components — without writing class components.

They essentially let functional components do things like:

- Store and update state (useState)
- Perform side effects (useEffect)
- Share logic across components (custom hooks)
- Access React's internal context and refs (useContext, useRef)

☑ Why Hooks?

Before hooks, only class components could use lifecycle methods, state, and certain advanced features. Hooks solved:

- Logic Reuse – Avoided "wrapper hell" with higher-order components (HOCs) or render props.
- Cleaner Code – No need for verbose class syntax.
- Better Separation of Concerns – Logic grouped by functionality instead of lifecycle methods.

Commonly Used Hooks

Hook	Purpose	Example
`useState`	Manage local component state	`const [count, setCount] = useState(0)`
`useEffect`	Run side effects (API calls, event listeners)	`useEffect(() => { document.title = count; }, [count])`
`useContext`	Consume values from `React.createContext()`	`const value = useContext(MyContext)`
`useRef`	Store mutable values, DOM references	`const inputRef = useRef(null)`

Hook	Purpose	Example
`useReducer`	Manage complex state logic	`const [state, dispatch] = useReducer(reducer, initialState)`
`useMemo`	Memoize values for performance	`const memoValue = useMemo(() => compute(a, b), [a, b])`
`useCallback`	Memoize functions to prevent re-renders	`const fn = useCallback(() => doSomething(), [])`

`useState` Hook

`useState` hook in React lets functional components have their own state — something that was only possible in class components before hooks came along.

Syntax

```
const [stateVariable, setStateFunction] = useState(initialValue);
```

- `stateVariable` → The current state value.
- `setStateFunction` → Function to update the state.
- `initialValue` → The starting value of your state (number, string, object, array, etc.).

Now we're gonnna use it in our current game to manage the variable changes.

```
const TextInputFormContainer = () => {

    const [inputType, setInputType] = useState("password");

    ...

    const handleShowHideClick = () => {
    console.log("Show/Hide clicked");
    if (inputType === "password") {
        setInputType("text");
    } else {
        setInputType("password");
    }
  };

  ...}
```

`useState` hook returns an Array & we destructure it inside `const [inputType, setInputType]` → First Element is state variable & Second Element is updater function.

- Also in `TextInputForm.jsx` we'll use this state to toggle button Among `Show` & `Hide`.

```
...
<div>
<Button
  styletype="warning"
  text={inputType === "password" ? "Show" : "Hide"}
  onClickHandler={handleShowHideClick} />
</div>
...
```

- Now I want to collect the hidden data user 1 is typing in the box & want to send it to the next page upon clicking on submit. We'll use a state variable to achieve it.

Whatever is written in input tag that'll be stored inside the `value` variable of `useState`.

Inside `handleTextInputChange` function I'm getting all the changes that I'm doing in input tag. in `e.target.value` I'll get the changed/updated value.

```
const [value, setValue] = useState("");

    const handleFormSubmit = (e) => {
    e.preventDefault();
    console.log("Form submitted:", value);
  };

  const handleTextInputChange = (e) => {
    setValue(e.target.value);
    console.log("Text input changed:", e.target.value);
  };
```

This way I'm tracking my changes & storing the value that I'm putting inside the input tag, inside state variable.

- Now I want to get redirected to a new page upon clicking on submit.
- If I use `window.object.href` the whole page will be refreshed. We're making a SPA so obviously we don't want that to happen.

React Router

React Router is the standard library for handling routing in React applications — it lets you create single-page apps (SPAs) that can have multiple "pages" without actually reloading the browser.

- Instead of the server serving different HTML files for each URL, React Router changes the UI based on the URL path while keeping the app running in the browser.

Why React Router?

- Enables navigation between components based on the URL.
- Keeps the UI in sync with the browser's address bar.

- Avoids full page reloads → faster navigation.
- Can handle nested routes, dynamic routes, and redirects.

Installation

```
npm install react-router-dom
```

Enabling React Router in your project.

- Go to `main.jsx` & `import {BrowserRouter} from 'react-router-dom'`
- Wrap your like this

```
<BrowserRouter>
      <App />
</BrowserRouter>
```

- Now We'll make a new page in our game named `PlayGame.jsx`
- We'll introduce `<Routes>` in `App.jsx` →

```
import {Route, Routes} from 'react-router-dom'
...
return (
    <Routes>
      <Route path="/" element={<TextInputFormContainer />} />
      <Route path="/button" element={<Button />} />
      <Route path="/text-input" element={<TextInputForm />} />
    </Routes>
  )
```

- On each route we'll define which component will render.
- Now if we want to redirect to any page inside we can use `Link to =" "` which is provided from React Router.

In `PlayGame.jsx` →

```
return (
    <>
      <h1>Play Game</h1>
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
```

In `StartGame.jsx` →

```
return (
    <div>
      <h1 className="text-2xl font-bold">Welcome to Hangman!</h1>
      <p className="text-lg">Get ready to guess the word!</p>
      <TextInputFormContainer />
      <Link to="/play" className="text-blue-600">Start Game</Link>
    </div>
  );
```

- Now we want that upon clicking on Submit button we should get redirected to the PlayGame page after 3 seconds.

It gives a programatic way where I can embed my logics also.
We'll use Navigate hook to control this.

Inside `TextInputFromContainer` →

```
import { useNavigate } from "react-router-dom";
...
const navigate = useNavigate();

    const handleFormSubmit = (e) => {
    e.preventDefault();
    console.log("Form submitted:", value);
    if (value) {
      // Navigate to the play page if the value is valid.
        setTimeout(() => {
            navigate("/play");
        }, 3000);
    }
  };
...
```

- Now I want to transfer the Data that I've been putting inside the input tag.
  `StartPage.jsx` → `PlayPage.jsx`

If we check the `TextInputFormContainer` We'll see a state which holds `value` & our taget is to send that value to `PlayPage.jsx`

Now Let's talk about `URL Structure` →

- Constant Part of the URL

```
/blog
```

- **Path Params**: Only value & it just becomes part of the main URL.

```
/blog/4
```

- **Query Params**: Key Value pair (`?___=___&___=___`) Representation of data:

```
/blog?date=24_08_25&author=pratik
```

We'll try to give the `/play` from `TextInputFormContainer` a path param in this way.

```
const handleFormSubmit = (e) => {
    e.preventDefault();
    console.log("Form submitted:", value);
    if (value) {
      // Navigate to the play page if the value is valid
        setTimeout(() => {
            navigate(`/play?text=${value}`);
        }, 3000);
    }
  }
```

in the URL bar query params will show like this:

```
http://localhost:5173/play?text=pratik
```

To fetch the value in `PlayGame.jsx` react-router Dom will give us a hook called `useSearchParams()`

```
const PlayGame = () => {
  const params = useSearchParams();
  return (
    <>
      <h1>Play Game</h1>
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
};
```

`useSearchParams()` returns an array & we'll be destructuring the array & will find `searchParams`

```
const [searchParams] = useSearchParams()
```

Up next we'll use `.get("text")` to fetch the value of `text` key which is basically our written word.

```
const [searchParams] = useSearchParams();
  console.log("Search params:", searchParams.get("text"));
```

Now as we can see this solution is not good for the Game we're making as we can see the hidden word in the URL already.

We can try doing this through Path Params -
Unlinke query params this type of params are integral part of URL. So we need to define it in our `App.jsx` file through `:text`. `:` means this part of the URL is variable.

```
<Route path="/play/:text" element={<PlayGame />} />
```

Now We've registered our path params & our `TextInputFormContainer` will look like this:

```
setTimeout(() => {
            navigate(`/play/${value}`);
        }, 3000);
```

We'll be able to access the value in `PlayGame.jsx` using `useParams()` hook given by React Router. It gives us an object & if we remember that in our `App.jsx` we gave the variable name `:text`.

```
const PlayGame = () => {
  const { text } = useParams();
  return (
    <>
      <h1>Play Game {text}</h1>
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
};
```

Multiple Path Params are also possible:

`App.jsx`

```
<Route path="/play/:text/:id" element={<PlayGame />} />
```

`TextInputFormContainer`

```
setTimeout(() => {
            navigate(`/play/${value}/2`);
        }, 3000);
```

PlayGame.jsx

```
const { text, id } = useParams();
  return (
    <>
      <h1>Play Game {text} {id} </h1>
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
```

Even this solution is not good for our game as it'll show the data on URL

3rd way: Using React-Router we can directly access the value from one page to another using another argument in Navigate with state property.

TextInputFormContainer

```
setTimeout(() => {
            navigate(`/play`, { state: { wordSelected: value } });
        }, 3000);
```

We can pass any value in that state. I've passed an object with key `wordSelected`. Whatever I assign inside `state`, can fetch it to the page I'm navigating to. → `\play`

We'll use `useLocation()` hook in `PlayGame.jsx` that'll return an object. If we destructure that - we'll get state property. Inside state we created the key `wordSelected` which we can access directly.

PlayGame.jsx

```
const PlayGame = () => {
  const {state} = useLocation();
  return (
    <>
      <h1>Play Game {state.wordSelected}</h1>
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
};
```

Now as we've seen in Hangman-Game, we need kind of interface where the recieved word gets masked/hidden in this way - `_ _ _ _ _ _`

- We'll be making a new Folder named `MaskedText` in componenets & then I'll be generating files named → `MaskedText.jsx` & `MaskingUtility.js`

inside `MaskingUtility.js` we'll be taking two arguments → `originalWord` (given input in the first place & expected to be guessed), `guessedLetters` (Letters which are guessed by the player.)

- I'll at first put all the letters to uppercase & then will make a new `.set` in `MaskingUtility.js`.

```
export function getAllCharacters(originalWord, guessedLetters) {
  guessedLetters = guessedLetters.map(letter => letter.toUpperCase());
  const guessedLettersSet = new Set(guessedLetters);
}
```

- `.set` will add/update a value for a given key.
- Now we'll take `originalWord` → Will change it to all `uppercase` → then will `split` it into an array & then use `map` to go over each `char`. Will check if `guessedLettersSet`'s char are matching with original word. If yes it'll show that `char` or else it'll show `_`

```
export function getMaskedString(originalWord, guessedLetters) {
  guessedLetters = guessedLetters.map(letter => letter.toUpperCase());
  const guessedLettersSet = new Set(guessedLetters);

  const result = originalWord.toUpperCase().split("").map(char => {
    if (guessedLettersSet.has(char)) {
      return char;
    } else {
      return "_";
    }
  });

  return result;
}
```

Now if we copy this code to console & call

```
getMaskedString ("humble", ["h","m","e"])
```

then we'll get this →

```
H_M__E
```

- Now we'll be moving to `MaskedText.jsx` file where we'll make the UI for this part.
- We'll take two arguments named `text` & `guessedLetters` in `MaskedText` function.
- Next we'll bring `getMaskedString`

```
const maskedString = getMaskedString(text, guessedLetters);
```

- Whatever we'll pass in `const MaskedText = ({text, guessedLetters})` will automatically get inside `const maskedString = getMaskedString(text, guessedLetters);`

Rendering Lists:

React says put any array inside `{ }` & we'll render it.

`PlayGame.jsx`

```
const arr = ["hello", "world"]
return (
  <>
    <h1> Play Game {state.wordSelected}</h1>
    {arr}
    <Link to="/start" className="text-blue-600">Back to Start</Link>
  </>
)
```

It'll render `helloworld`

- Now to render a tag as `h1` we've to map each of the elements.

```
return (
    <>
      <h1>Play Game {state.wordSelected}</h1>
      {arr.map((element) => (
        <h1>{element}</h1>
      ))}
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
```

This was if I render a list/array in my React Components, it's important to pass a `key` prop so that React can identify each items uniquely.

```
return (
    <>
      <h1>Play Game {state.wordSelected}</h1>
      {arr.map((element, idx) => (
```

```
        <h1 key={idx}>{element}</h1>
      ))}
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
```

Now we'll be using the same logic to `MaskedText.jsx`

```
import { getMaskedString } from "./MaskingUtility";
const MaskedText = ({text, guessedLetters}) => {

  const maskedString = getMaskedString(text, guessedLetters);
  return (
    <>
      {maskedString.map((letter, index) => (
        <span key={index} className="mx-1">
          {letter}
        </span>
      ))}
    </>
  );
};

export default MaskedText;
```

- We'll use this `MaskedText` components inside `PlayGame.jsx`

```
import { Link, useLocation } from "react-router-dom";
import MaskedText from "../components/MaskedText/MaskedText";

const PlayGame = () => {
  const {state} = useLocation();
  // const { text } = useParams();
  // const [searchParams] = useSearchParams();
  // console.log("Search params:", searchParams.get("text"));
  return (
    <>
      <h1>Play Game {state.wordSelected}</h1>
      <MaskedText text={state.wordSelected} guessedLetters={[]} />
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
};

export default PlayGame;
```

- As of now the section UI will show blank as the `guessedLetters={[]}` is initially none.

Now We'll make the keyboard of Hangman App.

- Create a Component folder named `LetterButtons` & create a jsx named `LetterButtons.jsx`
- Inside that we'll create a `const` named `alphabates` which will store `"QWERTYUIOPASDFGHJKLZXCVBNM".split("")`
- Create a set of original letters from the text & Create a set of guessed letters for quick lookup

```
const alphabets = "QWERTYUIOPASDFGHJKLZXCVBNM".split("");
const LetterButtons = ({ text, guessedLetters, onLetterClick }) => {

  // Create a set of original letters from the text
  const originalLetters = new Set(text.toUpperCase().split(""));
  // Create a set of guessed letters for quick lookup
  const guessedLettersSet = new Set(guessedLetters);
};

export default LetterButtons;
```

- Now we'll make an Array of `buttons` through map function. In alphabates all the letters are unique so we will be using that as a key. A custom string is made as `button - ${letter}`. We'll Also print the letter

```
const buttons = alphabets.map((letter) => {
    return (
      <button
      key={`button - ${letter}`}
      >
          {letter}
      <button/>)})
```

- We'll be returning `const buttons` also.

```
return <>{buttons}</>;
```

- We'll be adding some props to the `<button>`. First we'll be passing `onClick`. Parent component will tell us what'll happen on click through `onLetterClick`

```
<button
        key={`button - ${letter}`}
        onClick={onLetterClick}>
```

- Next property we'll be keeping as `disabled. Buttons I've clicked will be disabled through this.

```
return (
      <button
        key={`button - ${letter}`}
        onClick={onLetterClick}
        disabled={guessedLettersSet.has(letter)}
      >
        {letter}
      </button>
    );
```

For the letter current button is been made, if that letter has been guessed then disable it.

- Next we'll be adding a component to figure out the style. In that button I'll be passing this function `${buttonsStyle(letter)` which will define the colors based on the click on buttons. If the guessed letter is correct then It should be present inside original letters. if it's wrong then it'll be shown red.

```
const buttonsStyle = function (letter) {
    if (guessedLettersSet.has(letter)) {
      return `${originalLetters.has(letter) ? "bg-green-500" : "bg-red-500"};
    }
    return "bg-blue-500 hover:bg-blue-600";
  };

  const buttons = alphabets.map((letter) => {
    return (
      <button
        key={`button - ${letter}`}
        onClick={onLetterClick}
        disabled={guessedLettersSet.has(letter)}
        className={`h-12 w-12 m-1 rounded-md text-white font-bold
${buttonsStyle(letter)}`}
      >
        {letter}
      </button>
    );
  });
```

- Now inside `PlayGame.jsx` we'll be putting this with the prop values.

```
<div>
      <LetterButtons
        text={state.wordSelected}
        guessedLetters={[]}
        onLetterClick={() => {}}
      />
    </div>
```

- Now we'll write the logic of guessing inside `PlayGame.jsx`. upon changes on the guessed letter - the UI should change. Dosen't matter if it's a right guess or a wrong guess. To achieve that we need to rerender our componenets so we'll use `useState` hook. I want to track my variables & UI Changes.

```
const [guessedLetters, setGuessedLetters] = useState([]);
```

- We'll be adding a function - `function handleLetterClick(letter)`
- For updation of Array we need to approach it differently to make a new array from this - `useState([])`. We'll be destructuring all the elements of the old array by `...guessedLetters`& then will add new `letter`

```
function handleLetterClick(letter) {
    setGuessedLetters([...guessedLetters, letter]);
  }
```

- Now we'll pass `handleLetterClick` inside `onLetterClick` of `<LetterButtons/>` & inside `guessedLetters` of both `MaskedText` & `LetterButtons` we'll be passing our state variable `guessedLetters`

```
const PlayGame = () => {
  const {state} = useLocation();
  const [guessedLetters, setGuessedLetters] = useState([]);
  function handleLetterClick(letter) {
    setGuessedLetters([...guessedLetters, letter]);
  }
  // const { text } = useParams();
  // const [searchParams] = useSearchParams();
  // console.log("Search params:", searchParams.get("text"));
  return (
    <>
      <h1>Play Game</h1>
      <MaskedText text={state.wordSelected} guessedLetters={guessedLetters} />
    <div>
      <LetterButtons
        text={state.wordSelected}
        guessedLetters={guessedLetters}
        onLetterClick={handleLetterClick}
      />
    </div>
      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
};
```

After writing this code the whole UI will go blank after clicking on any Button in the webpage.

Reason :

- In console It'll show that inside `MaskingUtility.js` `letter.toUpperCase()` function is not available.
- If we console.log - `guessedLetters` we'll see empty arrray inside. Then if we click on any of the letter button it'll add a event mistakenly. This has happened because of a mistake we've done in page `PlayGame.jsx`
- `const buttonsStyle = function (letter)` is just a event handler & we've mistakenly passed the letter inside it. We should be passing an event instead.
- We'll get event object inside event handler. When I'll get event object we'll be fetching value from that event object.
- Now we'll go to `LetterButtons.jsx` & we passed `onLetterClick` directly last time in `onClick`. This time we'll create a function.

```
function handleLetterClick(event) {}
```

- To extract character from the `event` object we'll be giving a prop named `value` & will pass `{letter}`.

```
function handleLetterClick(event) {
  const characterOfTheLetter = event.target.value;
    onLetterClick(characterOfTheLetter);
  }

  const buttons = alphabets.map((letter) => {
    return (
      <button
        key={`button - ${letter}`}
        value={letter}
        onClick={handleLetterClick}
        disabled={guessedLettersSet.has(letter)}
        className={`h-12 w-12 m-1 rounded-md text-white font-bold
${buttonsStyle(letter)}`}
      >
        {letter}
      </button>
    );
  });
```

- Now if from parent component there's no callback been passed for `onLetterClick`
  For that we can use `onLetterClick?.`

What's `?.` operator

We call it `Optional Chaining`. The variable you're gonna put optional chaining operator - if that variable is of a non-undefined value then it'll call the object's property/will call the function.

How it works:

When the `?.` operator is used in an expression like `obj?.prop` or `obj?.method()`, if `obj` is `null` or `undefined`, the entire expression immediately "short-circuits" and evaluates to `undefined` instead of throwing a `TypeError`. This eliminates the need for explicit null or undefined checks at each level of nesting.

So, In our project it would check if `onLetterClick` is working as a valid function then it'll get called with `characterOfTheLetter` argument. If a valid function is not passed from the parents to `LetterButtons ({onLetterClick})` & Letter Buttons are clicked in the webpage it'll check that this function is not valid. So It'll not execute the rest of the part.

The whole codebase for `LetterButtons.jsx` looks like this now.

```
const alphabets = "QWERTYUIOPASDFGHJKLZXCVBNM".split("");
const LetterButtons = ({ text, guessedLetters, onLetterClick }) => {

  // Create a set of original letters from the text
  const originalLetters = new Set(text.toUpperCase().split(""));
  // Create a set of guessed letters for quick lookup
  const guessedLettersSet = new Set(guessedLetters);

  const buttonsStyle = function (letter) {
    if (guessedLettersSet.has(letter)) {
      return `${originalLetters.has(letter) ? "bg-green-500" : "bg-red-500"}`;
    }
    return "bg-blue-500 hover:bg-blue-600";
  };

  function handleLetterClick(event) {
    const characterOfTheLetter = event.target.value;
    onLetterClick?.(characterOfTheLetter);
  }

  const buttons = alphabets.map((letter) => {
    return (
      <button
        key={`button - ${letter}`}
        value={letter}
        onClick={handleLetterClick}
        disabled={guessedLettersSet.has(letter)}
        className={`h-12 w-12 m-1 rounded-md text-white font-bold
${buttonsStyle(letter)}`}
      >
        {letter}
      </button>
    );
  });
```

```
  return <>{buttons}</>;
};

export default LetterButtons;
```

Now the whole code would work perfectly & upon repeatative letter word like `apple` upon clicking in `P` it would fill up two spaces.

- Now We'll be saving Hangman images (from Excalidraw) as `1.svg`, `2.svg` etc. in our `src/images`. Next we'll be creating a file named `HangMan.jsx`. Will pass a prop named `step`.

```
import Level1 from '../../../public/images/1.svg'
import Level2 from '../../../public/images/2.svg'
import Level3 from '../../../public/images/3.svg'
import Level4 from '../../../public/images/4.svg'
import Level5 from '../../../public/images/5.svg'
import Level6 from '../../../public/images/6.svg'

function HangMan({ step }) {
}

export default HangMan;
```

- The value of `step` will let us know in which Level we're currently.
- Now we'll create an array of images. Will use it inside a logic. If step is bigger than/equals to the length of the images array, then show the last image or render the image of the step.

```
function HangMan({ step }) {
    const images = [Level1, Level2, Level3, Level4, Level5, Level6];
    return(
        <div className="w-[300px] h-[300px]">
            <img
            src={step>=images.length ? images[images.length - 1] :
images[step]}
            alt={`Hangman step ${step}`} />
        </div>
    )
}
```

- Now I want to step up whenever i guess somet wrong letter. So for that we'll be adding a state variable in `PlayGame.jsx`

```
const [step, setStep] = useState(0);
```

- If the word we got through input matches with the `letter` that we've typed then `console.log("Correct guess:", letter)` or else `setStep(step + 1);` increase step counter by one.

```
const [step, setStep] = useState(0);
  function handleLetterClick(letter) {
    if (state.wordSelected.toUpperCase().includes(letter)) {
      console.log("Correct guess:", letter);
    } else {
      setStep(step + 1);
    }
    setGuessedLetters([...guessedLetters, letter]);
  }
```

🗺️ React Component Lifecycle

In React, every component goes through a lifecycle — from being **created, updated, and finally removed from the DOM**. This lifecycle is divided into three main phases:

🔑 React Component Lifecycle Phases

1. Mounting Phase (When component is created & inserted into the DOM)

Happens when the component is rendered for the first time.

2. Updating Phase (When component is re-rendered due to changes)

Triggered when props or state changes.

3. Unmounting Phase (When component is removed from the DOM)

Cleanup before the component disappears.

Now, this can be the scenario that **I want to track the event changes & want to do something on basis of that**. To achieve that we'll be using a hook called `useEffect()`.

👉 `useEffect()`

It will help you to control instructions to be executed during different lifecycle events of a component in React.

- `useEffect()` in React is a Hook that lets you run side effects in function components.

It acts like a combination of `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in class components.

Syntax: It takes two parameters. 1. Callback, 2. Array

```
useEffect(() => {
    console.log("Component Loaded");
  });
```

- **First Execution :** When the component first mounts/ attaches in our DOM → Whatever logic I write inside Callback `() ("Component Loaded")` will get executed.
- **Second Execution :** Whenever the component re-renders/gets updated.

Now whenever we click on `show/hide` button it'll give us `"Component Loaded"` on console as everytime the state is changing/getting updated.

Now If I need a **More Granular Control** -

- No Dependency Array ([] missing) → Runs after every render.

```
useEffect(() => {
    console.log("Component Loaded");
  });
```

- Empty dependency array ([]) → Runs only once (like componentDidMount). The first time component renders.

```
useEffect(() => {
    console.log("Component Loaded");
  },[]);
```

- With dependencies ([value1, value2]) → Runs whenever those dependencies change.

```
  useEffect(() => {
    console.log("Component Loaded");
  }, [value]);
```

Cleanup function (return () => { ... }) → Runs before component unmount OR before the effect runs again.

```
useEffect(() => {
    console.log("Temp component mounted");

    return () => {
      console.log("Temp component unmounted");
    };
    // Cleanup logic
  }, []);
```

`<StrictMode>`

A devloper friendly mode. Mostly when we run our app in strict mode it's tend to get loaded twice.

- You can think of `useEffect` as a way to run side effects in a React component. Fetching data from an API is one of the most common side effects.

We want to make a One-Person Game now. So we'll be watching example of this property of `useEffect` through this.

👉 Making a JSON-Server made Backend

- Make a seperate folder for backend named `WordServer`.

```
mkdir WordServer
```

- Go To `WordServer`

```
cd WordServer
```

- creatinng `package.json` file

```
npm init -y
```

- install `json-server`

```
npm install json-server
```

- make a new file in folder named `db.json` & add these types of data in it.

```
{
    "words": [
        {
            "wordValue": "MANGO",
            "wordHint": "A tropical fruit"
        },
        {
            "wordValue": "APPLE",
            "wordHint": "A common fruit"
        },
        {
```

```
            "wordValue": "BANANA",
            "wordHint": "A yellow fruit"
        },
        ...
    ]}
```

- `npx json-server db.json` in terminal to run `json-server`
- Run the `Endpoint` [Endpoints: http://localhost:3000/words]
- You'll see in browser it'll automatically give a unique id to each of the objects

```
{
    "wordValue": "MANGO",
    "wordHint": "A tropical fruit",
    "id": "f30a"
}
```

- Now if we write `http://localhost:3000/words/f30a` in the URL section we'll only see the object of this id in the browser screen.

Now we'll try to call these details through `useEffect`

- add a new page in `pages` folder named `Home.jsx`

```
import { Link } from "react-router-dom"
import Button from "../components/Button/Button";

const Home = () => {
  return (
    <>
      <Link to="/play">
        <Button text="Single Player Game" styletype="error" />
      </Link>
      <br />
      <Link to="/start">
        <div className="mt-4" >
          <Button text="Multiplayer Game" styletype="success" />
        </div>
      </Link>
    </>
  );
};

export default Home;
```

- Connect it to `App.jsx`

```
```

```
<Route path="/" element={<Home />} />
```

But Here as `<Link to="/play">` play page is not recieving any data from the start page but still is getting redirected to the page, it'll show a blank page.

So somehow we need to send some data to play page to start the Single Player Game.

- If we send it in this way then it'll start the game with the word `"example"`.
- Using link tag when I'm redirecting to the `PlayGame.jsx` page, then I'm sending some data in `state` property & that data is getting fetched by `useLocation` in `PlayGame.jsx`

```
<Link to="/play" state={{ wordSelected: "example" }}>
```

- Now I don't want my feature to work like this obviously. So first of all delete `state={{ wordSelected: "example" }}`
- We should handle our `PlayGame.jsx` in such a way that if `wordSelected` is passing a null value then it should be handled accordingly.
- Inside `PlayGame.jsx` add optional chaining for all of the `wordSelected`

```
if (state?.wordSelected?.toUpperCase().includes(letter))
...
<MaskedText text={state?.wordSelected}
...
<LetterButtons
        text={state?.wordSelected}
...
```

To make the logic easier we'll be using short circuting through a conditional -

```
return (
    <>
      <h1>Play Game</h1>
      {state?.wordSelected && (
        <>
          <MaskedText text={state?.wordSelected} guessedLetters=
{guessedLetters} />
          <div>
      <LetterButtons
        text={state?.wordSelected}
        guessedLetters={guessedLetters}
        onLetterClick={handleLetterClick}
      />
```

```
    </div>
    <div>
      <HangMan step={step} />
    </div>
        </>
      )}

      <Link to="/start" className="text-blue-600">Back to Start</Link>
    </>
  );
```

- Now inside `Home.jsx` page we'll add state property `state={{ wordSelected: word }}` & we'll be making a state variable to pass the word.

```
const Home = () => {
    const [word, setWord] = useState("");
  return (
    <>
      <Link to="/play" state={{ wordSelected: word }}>
        <Button text="Single Player Game" styletype="error" />
      </Link>
...
)}
```

- We're sending this `state={{ wordSelected: word }}` useState variable word to `state` property & we're accessing that value in `PlayGame.jsx` `useLocation()`
- Now I want that from backend a word randomly come & updates the `useState("")` here. When I'm redirecting to the homepage then my data gets downloaded (mounting).

```
useEffect(() => {
        fetchWords()
    }, []);
```

Now, we'll use `async` & will add the `fetchWords()`. JS dosen't know how to raise a Network Request but our Browser gives us a functionality where we can raise a network request using `fetch` API. We can call a neywork through this.

```
async function fetchWords() {
        // Fetch words from the API
        const response = await fetch("http://localhost:3000/words/");
        // Convert the fetched data to JSON
        const data = await response.json();
        // Assuming the API returns an array of words, pick one at random
        const randomIndex = Math.floor(Math.random() * data.length);
        setWord(data[randomIndex]);
```

```
    }

    useEffect(() => {
        fetchWords()
    }, []);
```

Whenever my data is gonna be downloaded, It'll pick a random value & whatever value is in that random index value that'll be saved in my `useState` variable.

```
const Home = () => {
    const [word, setWord] = useState("");

    async function fetchWords() {
        // Fetch words from the API
        const response = await fetch("https://random-word-api.vercel.app/api?
words=100");
        // Convert the fetched data to JSON
        const data = await response.json();
        // Assuming the API returns an array of words, pick one at random
        const randomIndex = Math.floor(Math.random() * data.length);
        setWord(data[randomIndex]);
        console.log("Fetched word:", data[randomIndex]);
    }

    useEffect(() => {
        fetchWords()
    }, []);

  return (
    <>
      <Link to="/play" state={{ wordSelected: word }}>
        <Button text="Single Player Game" styletype="error" />
      </Link>
      <br />
      <Link to="/start">
        <div className="mt-4" >
          <Button text="Multiplayer Game" styletype="success" />
        </div>
      </Link>
    </>
  );
};
```

- In `PlayGame.jsx` we'll add this link tag

```
<Link to="/" className="text-red-600">Back to Home</Link>
      <Link to="/start" className="text-blue-600">Back to Start</Link>
```

Upon clicking on `Back to Home` it'll select a new word everytime. In UI first there was `Home.jsx` componenet - from there you went to `PlayGame.jsx` so Home Componenet got removed.

- From `PlayGame.jsx` when you'll come to `Home.jsx` it'll mount back again & again `useEffect` will get called as it'll always get called on first load.