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

🗐 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 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)

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-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

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

🐞 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.

👉 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
`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`.

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

Feature	CommonJS Syntax	ES6 Module Syntax	Notes
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
```

👉 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>;
}
```

➡️ 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.**