# **Getting Started with react**

React is a JS LIBRARY to create UI. (not a framwork)

Essentially, we describe a webpage using reusable components, and react will take care of efficiently creating and updating DOM elements. (DOM-->Duc Object Model)

Vite: is a modern frontend **Build** tool that provides a fast development environment and optimized production builds. It is particularly designed for Vue.js, but it also supports React, Svelte, and other frameworks.  
 npm create vite@latest

To run our web server:   
 npm run dev

There are two ways to create a react components, js class and functions. Function base component is getting more popular. We should use PascalCasing for naming our functions.

**JSX = Javascript XML**: is a syntax extension for JavaScript that allows you to write HTML-like code inside JavaScript. It makes writing React components easier and more readable.

When our application starts, React takes a tree of components and builds a JavaScript data structure called the **Virtual DOM:** The VDOM is a lightweight copy of the Real DOM that React keeps in memory.

1. React creates a virtual representation of the UI.
2. When data changes, React compares (diffs) the new VDOM with the previous one.
3. React finds what changed and updates only that part in the Real DOM.
4. This reduces unnecessary DOM manipulations, making React fast and efficient.

In browser-based apps, updating the DOM is done by a companion library called ReactDOM. In mobile apps, React Native uses native components to render the user interface.

React is not limited to web applications. It can be used across different platforms with different renderers:

React Library: Platform:

React DOM Web (Browsers)

React Native Mobile (iOS & Android)

React Three Fiber 3D Graphics (WebGL)

React Native Web Converts React Native to Web

Since React only manages the UI state and rendering logic, it can be adapted to any platform with the appropriate renderer. And Since React is just a library and not a framework like Angular or Vue, we often need other tools for concerns such as routing, state management, internationalization, form validation, etc.

# **Building Components**

In react, a component cannot render more than one element. For having several element (like a list with images, h1 and ...) in one component, we should wrap our elements with a **<></>**, which actually we are telling react to use a **Fragment** to wrap all his children.

In JSX we can only write HTML elements, to write logic like using if statement, we should use **{}**. To render a list in JSX, we use the ‘*array.map()*’ method. When mapping items, each item **must have a unique key**, which can be a string or a number.

**Conditional rendering:**

A very common technique to render contents dynamically is to use && operator:

{ true && 1} --> 1

{ true && "hamed" } --> "hamed"

{ false && "hamed" } --> false

So instead of using ternary operator like *{(a Boolean statement) ? <p>true</p> : null}* we can use the above implementation which is better and shorter.

## **Handling Events:**

React actually use synthetic events, which are cross-browser wrappers around native events.

For handling the click event, we need to specify the **onClick** attribute on the html element, and pass a function. The function can be written there if it is a small logic (arrow func), but a better way is to define the function somewhere else and simply reference it to the onClick attribute.

**Note:**

onClick is **an event listener** that triggers a function when an element is clicked.  
The function that runs when onClick fires is called an **event handler**.

**Using Arrow Functions vs. Function References:**

Use function reference when no arguments are needed:

<button onClick={handleClick}>Click me</button>

function handleClick() {

console.log("Button clicked!");}

Use an arrow function when passing arguments **to prevent immediate execution**:

<li onClick={() => handleClick(index)}> {item} </li>

function handleClick(index) {

console.log("Clicked item index:", index);}

Wrong: *onClick={setSelectedIndex(index)}* ❌ (Runs immediately on render)

Correct: *onClick={() => setSelectedIndex(index)}* ✅ (Runs only when clicked)

The event handler functions are by convention **named** like this: handleClick, handleChange, handle...

Here are some other event listener:

Click Events onClick

Form Events onChange, onSubmit

Keyboard Events onKeyDown, onKeyUp

Mouse Events onMouseEnter, onMouseLeave

Prevent Bubbling e.stopPropagation()

Prevent Default e.preventDefault()

## **Managing State:**

State management in React controls how data changes and updates the UI. React provides multiple ways to manage state based on the complexity of our app.

For small, isolated state changes, we **use Local State (useState)** which stores state inside a single component. And different component will have separated different state.

There are different management approaches base on more complex uncases like:

Small UI state useState()

Share state across components Context API

Large-scale global state Redux, Zustand, Jotai

Server-side data & caching React Query, SWR

## **Hooks:**

We use the state hook to define state (data that can change over time) in a component. A hook is a function that allows us to tap into built-in features in react. It is a special function in React that allows functional components to use state and lifecycle features without needing class components (older way).

For example, **useState** is a state hook which by using it we can tell the react that this component can have data or state that will change over time:

*const arr = useState(-1)*

*arr[0]* // variable

*arr[1]* // updater function: using this, react will be notified the state of this component is changed and then it will re-render our component which causes the DOM to be updated under the hood.

There are also other hooks ...

## **Passing Data via Props:**

Props (short for "properties") are a way to pass data from a parent component to a child component. In another word, we can just use them to pass data to our components.

Props are read-only, meaning that a child cannot modify props received from a parent.

### **Passing Functions via Props:**

Sometimes we need a mechanism to notify the parent of a component that an item is selected, that’s where we use a **Call Back Function**. The parent passes a function as a prop to the child, and the child calls it when an event occurs.

### **Passing Children:**

In React, **Children** is a special prop that allows components to pass nested content, such as other components or elements, to their child components. if we want to define an interface for it, we need to specify it to a **ReactNode** type.

## **Props vs. State:**

**Props:**

* Input passed to a component.
* Similar to function args.
* Immutable

**State:**

* Data managed by a component.
* Similar to local variables.
* Mutable.

What do both of them have in common is that they both cause a **re-render** by changing.

# **Styling Components**

Overall, there a lot of CSS frameworks that we can use, but to style component manually we can only use two ways:

* **CSS Modules**: which we specify CSS roles in a separate file, import it in the component and name it. Syles them using dot or [] we pass the roles to the className element like this:

<ul> className={[styles.list, styles.listrgroup].join(" ")} </ul>

**Note**: the CSS file should be named like this: ListGroup.module.css, then create a folder and store both the component and the style in one file.

* CSS-in-JS: I didn’t like this approach, visit its video.

## **CSS frameworks/libraries:**

* . **Bootstrap** – A popular CSS framework that provides pre-styled components (buttons, cards, modals, etc.) and a responsive grid system. It is easy to use but can lead to websites looking similar if not customized.
* **Tailwind CSS** – A utility-first CSS framework that allows you to style elements directly in HTML using classes (e.g., bg-blue-500, text-lg). It offers flexibility and customization but has a learning curve.
* **Material UI (MUI)** – A React-based UI library that follows Google’s Material Design guidelines. It provides ready-made, customizable React components (buttons, cards, typography, etc.), making it great for modern web apps.
* **Chakra UI** – is a component-based library that provides accessible, customizable, and developer-friendly UI components for React. It focuses on simplicity and flexibility while maintaining great design principles.

|  |  |  |
| --- | --- | --- |
| Use Case | Best Option | Why? |
| Fast prototyping & ready-made components | Bootstrap or Material UI | Quick, pre-styled, and easy to use |
| Fully customized UI with flexibility | Tailwind CSS | No design restrictions, lightweight |
| React-based UI with built-in components | Chakra UI | Clean API, flexible, built for React |
| Modern web apps following Material Design | Material UI (MUI) | Google's Material Design, polished look |
| Performance & minimal styles | Tailwind CSS | No extra styles, only what you use |

For adding icons, we will use **React Icons**. They are essentially a react component which we can also customize them.

## Managing Component State **State Hook:**

* React updates state **asynchronously** to minimize unnecessary re-renders, it batches the updates, applies them all at once and then re-renders a component. This means that if you try to read the state immediately after setting it, you might still see the old value.
* State is stored **outside of the component**: React does not store the state inside the component function itself. Instead, it keeps state in an internal memory structure that persists across re-renders. That is why we cannot declare a variable like let count = 0; inside a React component and try to update it in an onClick, because React re-runs the component function every time state changes. So A normal variable the count above resets on each render.
* Use hooks at the top level of our component: Hooks must not be used inside loops, conditions, or nested functions.

## **Choosing the state structure:**

* Avoid redundant state variables: anything that can be computed from existing variables.
* Group related variables inside an object.
* When using objects, avoid deeply nested structures.

Pure Function: Means that given the same input, always returns the same result.

## **Updating objects:**

When we dealing with objects and arrays, we should remember to treat them as immutable or read only. So, we cannot modify the property and set them to the object, react wont detect any changes.

In React, when managing objects with useState, you must update the entire object rather than modifying properties directly. This is because React does not merge nested state updates automatically. So Never modify state directly (use the spread operator { ... } to create a new object and set them).

### **Updating Nested objects:**

When updating nested objects, you need to update each level manually:

const [profile, setProfile] = useState({

name: "Hamed",

address: { city: "Tehran",

zip: 12345 },

});

const updateCity = () => {

setProfile(prevProfile => ({

...prevProfile,

address: { ...prevProfile.address, city: "Milan" },

})); };

### **Updating arrays:**

const [tags, setTags] = useState (['happy', 'cheerful']);

const handleClick = () => {

//Add

setTags([...tags, 'exciting']);

//Remove

setTags([tags.filter(tag => tag !== 'happy')]);

//update

setTags(tags.map(tag => tag === 'happy' ? 'happiness' : tag))

}

## **Simplifying update logic with immer:**

**Immer** is a JavaScript library that simplifies immutable state updates by allowing you to write mutative-looking code while ensuring immutability under the hood. In React, updating objects and arrays immutably (without modifying the original state) can be verbose and error-prone. Immer makes this process easier.

import { produce } from "immer";

...

const [tags, setTags] = useState(["happy", "cheerful"]);

const handleClick = () => {

setTags(

produce(draft => {

// Add a new tag

draft.push("exciting");

// Remove "happy" tag

const index = draft.indexOf("happy");

if (index !== -1) draft.splice(index, 1);

// Update "happy" to "happiness"

draft.forEach((tag, i) => {

if (tag === "happy") draft[i] = "happiness";

});

})

);

};

## **Sharing State between components:**

If two sibling components need to share state, move the state to their common parent and pass it down as props. As a role of thumb, **the component that hold the state is responsible for updating it.**

# **Building Forms**

Accessing Input fields:

**UseRef Hook**: The useRef hook in React is used to create a mutable reference that persists across renders without causing a re-render when updated. It is commonly used for:

* Accessing DOM Elements – useRef allows direct interaction with a DOM element.
* Storing Mutable Values – It helps store values that don’t trigger re-renders when changed.
* Keeping Previous Values – It can be used to track previous state values.

UseRef returns an object: {***current****: initialValue*} and unlike useState, updating a useRef value does not trigger a component re-render.

Note: we have to always initiate useRef with **null**.

We can use useState instead of useRef in some cases, but it depends on what we are trying to achieve.

Here’s a comparison:

Feature useRef useState

Triggers re-render? ❌ No ✅ Yes

Holds a persistent value across renders? ✅ Yes ✅ Yes

Best for DOM manipulation? ✅ Yes ❌ No

Useful for tracking values without re-render? ✅ Yes ❌ No

👉 Use useRef when you **don’t need re-renders** and just want to persist a value.

👉 Use useState when **the UI should update** when the value changes.

## **Creating and Validating Forms with React-Hook-Form:**

**React Hook Form** is a popular library for handling forms in React using hooks. It provides an efficient and minimalistic way to manage form validation, data handling, and submission.

Common Hooks & Functions:

Hook/Function Description

useForm () Initializes form control, validation, and submission.

register () Registers input fields to React Hook Form.

handleSubmit () Handles form submission.

reset () Resets form values.

setValue () Manually sets field values.

watch () Watches field values in real-time.

getValues () Retrieves form values programmatically.

formState.errors Contains validation errors.

Often, we need to have several form validations, this is where we better to use **Schema** validation libraries like:

**Zod**: Zod is a TypeScript-first schema validation library used to define and validate data structures in a simple and type-safe way. It is widely used in form validation, API request validation, and TypeScript applications to enforce data integrity.

# **Connecting to Back-end**

Understanding the Effect Hook:

The useEffect hook in React is used to handle **side effects** in functional components. Side effects include things like:

* Fetching data from an API
* Updating the DOM
* Subscribing to events
* Setting up and cleaning up timers

The effect hook takes a function that **performs the side effect** and an optional array of dependencies. Whenever the dependencies change, the effect hook runs again.

## **Fetching data:**

For making HTTP Requests we can use fetch or **Axios library**: Axios is a popular library for making HTTP requests with more features.

we need to use *axios.get('url')* for fetching data, and this method return an object called **Promise**.

A Promise is a JavaScript object that represents a value that may be available now, later, or never. It's used to handle **asynchronous** operations.

An asynchronous operation is a process that runs in the **background** without **blocking** the main execution thread. This is useful for tasks that take time, such as:

* Fetching data from an API
* Reading/writing files
* Waiting for user input

As a best practice, when we fetch data in an effect, we should also provide a cleanup function for cancelling the fetch request in case the data is no longer needed. (AbortController ()).  
  
The communication between the front-end and the back-end happens over HTTP, the same protocol that powers the web. The front-end sends an HTTP request to the back-end, and the back-end sends an HTTP response back.  
  
Each HTTP request and response contains a header and a body. The header provides metadata about the message, such as the content type and HTTP status code, while the body contains the actual data being sent or received.   
  
When sending HTTP requests, we must handle errors properly. This can be done using try-catch blocks or by handling the error in the promise chain using *.catch()*.   
  
**Custom hooks** are a way to reuse code logic between multiple components. By encapsulating logic in a custom hook, we can create reusable pieces of code that can be shared across components without duplicating the code. Custom hooks can be used to handle common tasks, such as fetching data, and can help to make our code more organized and easier to maintain.