React Fundamentals (JSX, Components, Props, Hooks)

A practical mental model for building modern web app UIs

What is React? (One-sentence mental model)

React is a UI library for building the **view** of your app: you write **components** (functions) that describe **what the UI should look like for the current data/state**, and React updates the browser DOM efficiently when that data changes.

Core Idea: Components are functions that return UI

Component = Function

A React **component** is usually a JavaScript function that returns **JSX**. JSX *looks like HTML* but is actually JavaScript syntax that compiles into React function calls.

Key Concept Check

React does not "mix HTML and JS" at runtime. JSX is compiled into JavaScript first, then React renders it.

Simple Component (Static UI)

```
export default function ShoppingList() {
  return (
    <div className="shopping-list">
      <h1>Shopping List</h1>
      <ul>
        <li>Apples</li>
        <li>Bananas</li>
        <li>Grapes</li>
      </ul>
    </div>
  );
}
```

JSX Essentials: "JavaScript inside UI"

Curly braces { } mean "evaluate JavaScript here"

Inside JSX, wrap JS expressions in { } to embed values into the UI (strings, numbers, function results, etc.).

Rendering a list with `map()`

```
function ShoppingList() {
  const items = ["Apples", "Bananas", "Grapes"];

  return (
    <ul>
      {items.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  );
}
```

Must-know: list keys

When rendering arrays, provide a stable `key` so React can track items across re-renders. Avoid `index` as key if the list can be reordered/inserted/deleted (it can cause UI bugs).

Composition: building UI by nesting components

Composition = Components inside components

React UIs scale by composing small components into larger ones, like LEGO.

Why composition matters

- Reuse UI blocks across pages
- Keep each component small and understandable
- Separate concerns (layout vs data vs behavior)

Props: passing data from parent to child

Props = function arguments (data in)

Parents pass data to children using HTML-like attributes; children receive them as a JS object.

Passing props to a reusable `Card`

```
function Cards() {
  return (
    <div className="cards">
      <Card name="Alice" job="Frontend Developer" />
      <Card name="Bob" job="Backend Developer" />
      <Card name="Charlie" job="Full Stack Developer" />
    </div>
  );
}

function Card({ name, job }) {
  return (
    <div className="card">
      <h4><b>{name}</b></h4>
      <p>{job}</p>
    </div>
  );
}
```

Key Concept Check

Props are read-only. A child component should not mutate props; if something must change, lift state up to the parent.

Interactivity: events call functions

Events connect UI to behavior

In JSX, you attach handlers like `onClick` / `onChange` to run code when the user interacts.

Click handler inside a component

```
function Button() {
  function handleClick() {
    console.log("Hello!");
  }

  return <button onClick={handleClick}>Click</button>;
}
```

Reading input text from the event

```
function InputLogger() {
  function handleOnChange(text) {
    console.log(text);
  }

  return (
    <input
      type="text"
      onChange={(e) => handleOnChange(e.target.value)}
    />
  );
}
```

Must-know: handlers are functions, not strings

Write `onClick={handleClick}` not `onClick="handleClick()"` (that is plain HTML style).

State: variables that re-render the UI

useState creates state that updates the view

`useState` stores a value *and* gives an updater function. When state updates, React re-renders the component so the UI matches the new state.

Counter with useState

```
import { useState } from "react";

function Counter() {
  const [myNumber, setMyNumber] = useState(0);

  function increment() {
    setMyNumber(myNumber + 1);
  }

  return (
    <div>
      <p>{myNumber}</p>
      <button onClick={increment}>Increment!</button>
    </div>
  );
}
```

Must-know: state updates are batched/asynchronous

If you log right after `setState`, you may see the old value. Use the functional form when new state depends on old state:

Safe increment (functional update)

```
setMyNumber((prev) => prev + 1);
```

State with arrays/objects (immutability rule)

To update arrays/objects in state, create a **new** array/object (do not mutate in place).

Adding to an array in state

```
const [fruits, setFruits] = useState([]);

function addFruit(currentFruit) {
  setFruits((prev) => [...prev, currentFruit]);
}
```

Side Effects: running code when the component loads/changes

useEffect = "run this after render"

Use `useEffect` for **side effects**: fetching data, subscriptions, timers, manually touching the DOM, etc.

Fetch on page load (empty dependency array)

```
import { useEffect, useState } from "react";

function LoadDataFromServer() {
  const [data, setData] = useState("");
  const [loading, setLoading] = useState(false);

  async function loadData() {
    setLoading(true);
    const result = await apiCall();
    setData(result);
    setLoading(false);
  }

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

  return <div>{loading ? "Loading..." : data}</div>;
}
```

Dependency Array Rules (high-yield)

- `[]` runs once on mount (initial load)
- `[x]` runs whenever `x` changes
- Omit the array: runs after every render (often a mistake)

useRef: persistent values and DOM references

useRef has 3 common use cases

- Store a mutable value that **does not trigger re-render**
- Keep a value across renders with **synchronous** updates (`ref.current`)
- Get a direct reference to a **DOM element** (e.g., focus an input)

Must-know: changing a ref does NOT update the UI

If you want the UI to change, use `useState`. If you want to remember something without re-rendering, use `useRef`.

Ref counter (logs update, UI does not)

```
import { useRef } from "react";

function CounterWithRef() {
  const myNumber = useRef(0);

  function increment() {
    myNumber.current += 1;
    console.log(myNumber.current);
  }

  return (
    <div>
      <p>{myNumber.current}</p>
      <button onClick={increment}>Increment!</button>
    </div>
  );
}
```

DOM reference: focus an input on load

```
import { useEffect, useRef } from "react";

function InputFocus() {
  const inputRef = useRef(null);

  useEffect(() => {
    inputRef.current.focus();
  }, []);

  return <input ref={inputRef} type="text" />;
}
```

Putting it together: the React "flow"

How a React UI typically works (the loop)

- **Render**: component returns JSX based on current state/props
- **User interacts**: events call handlers
- **Update state**: `setState(...)` triggers re-render
- **Effects run**: `useEffect` handles side work (fetching, DOM, etc.)

Final Takeaways

- React UI is built from **components** (functions returning JSX)
- **Props** pass data down; **state** changes UI
- **Events** create interactivity
- **useEffect** handles side effects (fetching, subscriptions)
- **useRef** stores values without re-render and references DOM nodes

Rule of thumb: **UI that must update** → useState. **UI-irrelevant memory / DOM handle** → useRef.