REACT

1. What is react, why we use react and where we use

What is React?

React is a **JavaScript library** used for **building user interfaces (UIs)** — especially for **single-page applications (SPAs)**.
It was developed by **Facebook (now Meta)** and is maintained by Meta and a large open-source community.

It helps developers build **fast**, **interactive**, and **dynamic** web apps by creating reusable UI components.

Why we use React

We use React because it makes web development:

1. **Faster and more efficient** — React updates only the parts of the page that change using the **Virtual DOM**, instead of reloading the entire page.
2. **Component-based** — You can create small, reusable pieces of UI (like buttons, cards, forms) and combine them to make big applications.
3. **High performance** — Virtual DOM improves rendering speed and makes apps more responsive.
4. **Reusable code** — Once a component is written, you can use it anywhere in the app.
5. **Strong community & ecosystem** — Many tools, libraries, and packages are available for React.
6. **Easy to integrate** — You can use React with other frameworks or backends like Node.js, Django, or Laravel.
7. **Declarative approach** — You describe *what* you want the UI to look like, and React handles *how* to render it.

Where we use React

React is mainly used in:

1. **Web applications**
    - Example: Facebook, Instagram, Netflix, and Airbnb use React for their frontends.
2. **Mobile applications (React Native)**
    - Using React Native, you can build Android and iOS apps using the same React codebase.
3. **Desktop applications**
    - With tools like **Electron + React**, you can build desktop apps (like Slack or Visual Studio Code).
4. **Progressive Web Apps (PWAs)**
    - React helps create web apps that behave like mobile apps.

2. What is the difference between Html using webpage and react using webpage

1. HTML Webpage (Traditional Web Development)

When you build a webpage using **HTML, CSS, and JavaScript** directly, you're working with **static or manually updated content**.

How it works:

- The **browser loads an entire HTML page** from the server.
- If something changes (like new data or user interaction), the **entire page refreshes** or you must manually modify the **DOM** using JavaScript.
- Structure, style, and behaviour are often mixed together (HTML + CSS + JS in separate files but manually connected).

2. React Webpage (Modern Web Development)

When you build a webpage using **React**, you create a **component-based**, **dynamic**, and **efficient** UI.

How it works:

- React uses a **Virtual DOM** (a copy of the real DOM in memory).
- When something changes, React updates **only the changed parts** — not the whole page.
- React components are **modular**, **reusable**, and use **JSX** (JavaScript + HTML syntax).
- Pages don't reload; React dynamically updates the UI inside a single page → called a **Single Page Application (SPA)**

Feature	HTML Webpage	React Webpage
Structure	Built using HTML, CSS, and JS separately	Built using components (JSX + JS)
Page Reload	Reloads entire page on update	Updates only changed parts (no reload)
Performance	Slower for dynamic content	Faster due to Virtual DOM
Reusability	Low – elements must be re-created	High – components can be reused
Logic Handling	Handled manually with JS	Handled with React state & hooks
App Type	Multi-page website	Single Page Application (SPA)
Maintenance	Harder as project grows	Easier due to modular components

HTML webpage = Static and manually updated.

React webpage = Dynamic, component-based, and automatically updates.

3. what is the difference between react and angular

Feature	React	Angular
Type	JavaScript **library** (focused on UI)	Full **framework** (complete solution)
Developed by	**Meta (Facebook)**	**Google**
Language used	JavaScript (or JSX)	TypeScript (a superset of JS)
Architecture	View layer only (MVC's "V")	Full MVC / MVVM architecture
Data Binding	One-way data binding	Two-way data binding
Components	Uses **functional & class components**	Uses **components, modules, and directives**
DOM Type	**Virtual DOM** (updates efficiently)	**Real DOM** (with change detection mechanism)
Performance	Very fast due to Virtual DOM	Slightly slower due to real DOM and heavy structure
Learning Curve	Easier (only the view layer to learn)	Steeper (covers routing, state, forms, etc.)
Flexibility	Highly flexible; choose your own tools (routing, state management, etc.)	Everything built-in (routing, forms, HTTP, etc.)
Size	Smaller and lighter	Larger in file size
Used for	SPAs, dynamic UIs, mobile (React Native)	Enterprise-level applications, large-scale SPAs
Updates	Library evolves quickly	Follows structured version releases

4. what is typescript

TypeScript is a superset of JavaScript developed by Microsoft that adds static typing and advanced features to the JavaScript language. It is designed to make web development more reliable and easier to maintain, especially for large-scale applications. In simple terms, TypeScript allows developers to specify data types for variables, functions, and objects, which helps catch errors during development rather than at runtime. This improves the quality and stability of the code.

Unlike JavaScript, which is dynamically typed, TypeScript uses compile-time type checking, meaning it checks for mistakes before the code is executed. It also supports object-oriented programming concepts such as classes, interfaces, and inheritance, making the code more structured and organized. TypeScript code is written in files with the extension .ts, and it must be compiled into plain JavaScript before running in a browser or Node.js environment.

Developers prefer TypeScript because it provides better error detection, cleaner syntax, and enhanced code readability. It also works seamlessly with modern frameworks like Angular, React, and Vue, where it helps manage complex logic with fewer bugs. Overall, TypeScript makes JavaScript development safer, smarter, and more efficient, allowing developers to build scalable and maintainable web applications.

5. what is variable, declaration, initialisation, lexical scope, block scope, functional scope, Hosting and closure

1. Variable

A variable is a named storage location in memory that holds data which can be used and changed during program execution. It acts as a container for storing values like numbers, strings, or objects. In JavaScript, variables are declared using var, let, or const. For example,

let name = "Siva";
Here, name is the variable, and "Siva" is the value stored in it.
Variables help make programs dynamic and reusable.

2. Declaration

Declaration means **creating a variable** by telling the JavaScript engine its name — but not necessarily giving it a value. When you declare a variable, space is reserved in memory for it.
Example:
let age;
Here, the variable age is declared but not assigned any value yet, so its value is undefined.

3. Initialization

Initialization means **assigning a value** to a declared variable. It can happen at the same time as the declaration or later.
Example:

```
let age = 21; // declared and initialized
age = 25;    // re-initialized
```

Here, age is both declared and initialized with a value of 21.
Initialization gives meaning to a variable by storing data in it.

4. Lexical Scope

Lexical scope means that the **scope (visibility)** of variables is determined by the **position of the variable in the source code**, not by how or where it is called.
In simple terms, **inner functions have access to variables defined in their outer functions**.
Example:

```
function outer() {
 let a = 10;
 function inner() {
  console.log(a); // can access 'a' because of lexical scope
 }
 inner();
```

```
}
outer();
```
Here, inner() can access variable a defined in outer() because it is **lexically inside** that function.

5. Block Scope

Block scope means that variables declared inside a block { } can be accessed **only within that block**.

This applies to variables declared with **let** and **const**, but **not** with var.

Example:

```
{
 let x = 5;
 console.log(x); // works
}
console.log(x); // ❌ Error — x is not defined
```
Here, x is **block-scoped**, meaning it exists only inside the { } where it was declared.

6. Functional Scope

Function scope means that a variable declared inside a function is accessible **only within that function**.

In JavaScript, variables declared with **var** are function-scoped.

Example:

```
function test() {
 var y = 10;
 console.log(y); // works
}
console.log(y); // ❌ Error — y is not accessible outside function
```
Here, y is **function-scoped**, meaning it's visible only inside the test() function.

7. Hoisting

Hoisting is a behavior in JavaScript where **variable and function declarations are moved to the top of their scope** before the code executes.

However, only **declarations** are hoisted — **initializations are not**.

Example:

```
console.log(a); // undefined (not an error)
var a = 10;
```

Here, the declaration var a is hoisted, but the value assignment a = 10 is not.

If you use let or const, they are hoisted too but stay in a **"temporal dead zone"** until declared — so using them before declaration causes an error.

8. Closure

A **closure** is created when a **function "remembers" the variables from its outer scope**, even after that outer function has finished executing.

In simple terms, **a closure gives a function access to variables outside its own scope**.

Example:

```
function outer() {
  let count = 0;
  return function inner() {
    count++;
    console.log(count);
  };
}

const counter = outer();
counter(); // 1
counter(); // 2
```

Here, the inner function inner() remembers the variable count from outer(), even though outer() has already returned. This is called a **closure** — one of JavaScript's most powerful features.

Variable: Container for storing data.
Declaration: Creating the variable name.
Initialization: Giving it a value.
Lexical Scope: Inner functions can access outer variables.
Block Scope: Exists only inside { }.
Function Scope: Exists only inside a function.
Hoisting: Declarations move to the top before execution.
Closure: Function remembers variables from its parent scope.

6. In web development process why node is coming, which one is first node, angular or react

Why Node.js Comes in Web Development
Node.js is a **JavaScript runtime environment** that allows developers to **run JavaScript on the server side**, not just in the browser.
Earlier, JavaScript was used **only for frontend (client-side)** tasks like animations, form validation, or DOM manipulation. But with Node.js, developers can now use JavaScript to build **backend (server-side)** applications too — such as handling databases, APIs, and server requests.
So, Node.js comes into web development to make **JavaScript a full-stack language** — usable for both frontend and backend. It helps developers build faster, scalable, and real-time web applications like chat apps, streaming platforms, or APIs.

Technology	Release Year	Developed By	Purpose
Node.js	**2009**	Ryan Dahl	Run JavaScript on the server (backend)
AngularJS	**2010**	Google	Build dynamic frontend web

Technology	Release Year	Developed By	Purpose
			apps (MVC framework)
React	**2013**	Facebook	Build UI components for frontend apps

How They Work Together in Web Development

- **Node.js** → Handles the **backend**, APIs, and server logic.
- **Angular / React** → Handle the **frontend** (the user interface).
  In modern web development:
- Developers often use **Node.js** to build the backend (like with Express.js).
- And use **React** or **Angular** for the frontend.
  Together, they form a **full-stack JavaScript environment** — for example, **MERN (MongoDB, Express, React, Node)** or **MEAN (MongoDB, Express, Angular, Node)** stacks.

7. VITE and CRA create and write the folders and file name