**Chapter 06 - Exploring the world**

**1️⃣ What is Microservice architecture?**

Microservice architecture breaks an application into **small, independent services**, each responsible for a specific feature (e.g., Auth, Payments, Orders).

* They communicate via APIs.
* Easy to scale, deploy, and maintain.

**2️⃣ What is Monolith architecture?**

Monolith is a **single, large codebase** where all features (UI, business logic, database) are tightly coupled and deployed together.

* Harder to scale and update compared to microservices.

**3️⃣ Difference between Monolith and Microservice**

| **Feature** | **Monolith 🏛️** | **Microservices ⚡** |
| --- | --- | --- |
| **Structure** | One big codebase | Multiple small independent services |
| **Scalability** | Hard to scale parts separately | Easy to scale individual services |
| **Deployment** | Entire app redeployed on changes | Deploy/update one service at a time |
| **Tech Stack** | Single technology for all | Each service can use different tech |
| **Failure Impact** | One bug may crash whole app | Failure isolated to one service |

**4️⃣ Why do we need useEffect Hook?**

“We need useEffect to perform side effects in React functional components,

such as fetching data, subscriptions, or timers.

It ensures these actions run at the right time (on mount, update, or unmount) without causing unnecessary re-renders.”

A screenshot of a computer program

AI-generated content may be incorrect.

**Explanation –**

First, two things to know:

* React **renders UI** (what we see on screen).
* But sometimes we need to do things **outside UI rendering** → like fetching data, setting a timer, or listening to a key press.  
  These things are called **side effects**.

**🛑 Problem (without useEffect)**

If you directly put an API call or timer inside your component function → it would run **every time the component re-renders**, leading to:

* Multiple API calls,
* Memory leaks,
* Bugs.

**✅ Solution → useEffect**

The useEffect Hook lets us **control when side effects should run** (like on mount, update, or unmount).  
It makes functional components behave like class components with lifecycle methods (componentDidMount, componentDidUpdate, componentWillUnmount).

**5️⃣ What is Optional Chaining (?.)?**

Optional Chaining is a **safe way to access nested properties** without breaking the code if something is null or undefined.

A computer code with text

AI-generated content may be incorrect.

**6️⃣ What is Shimmer UI?**

* A **loading placeholder effect** shown while actual data (like images/text) is being fetched.
* Looks like a grey/animated skeleton.
* Improves user experience during API calls.

**7️⃣ Difference between JS Expression and JS Statement**

| **Term** | **Expression** | **Statement** |
| --- | --- | --- |
| **Definition** | Produces a value | Performs an action |
| **Example** | 2 + 3, user.name | if (x>0) { }, for(...) { } |
| **Usage in JSX** | Allowed inside { } | Not directly allowed |

✅ In JSX, only **expressions** work inside { }.

Example :

Statement : Expression:

const a=10; {(const a=10), console.log(a)}

console.log(a)

**8️⃣ What is Conditional Rendering?**

It means **rendering components/UI based on a condition**.

Example:

A screenshot of a computer screen

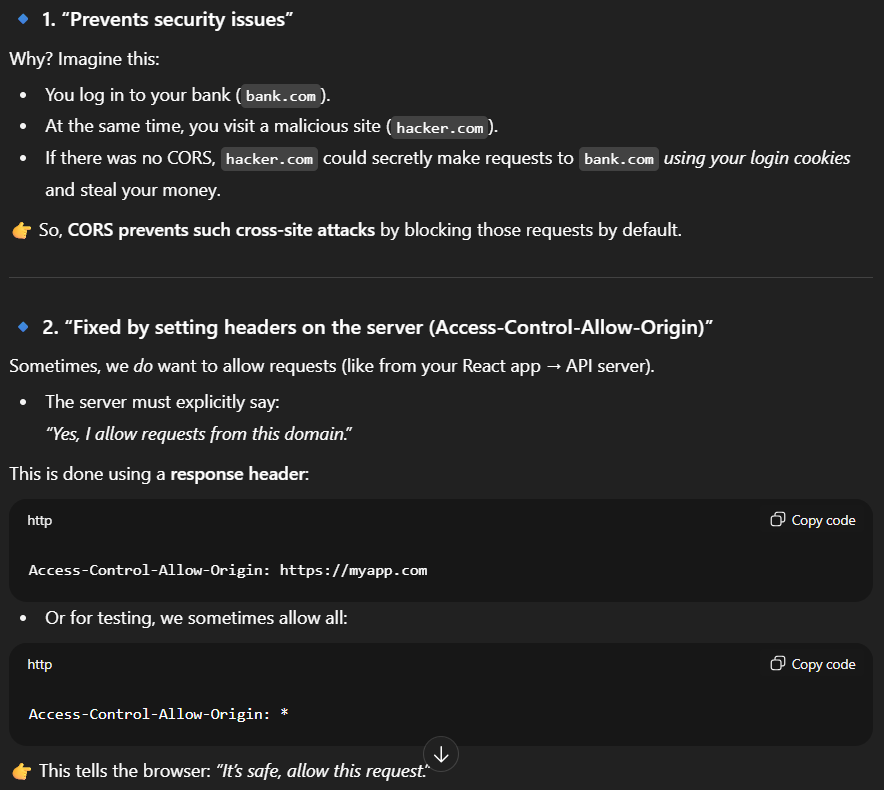
AI-generated content may be incorrect.

**9️⃣ What is CORS?**

**Cross-Origin Resource Sharing (CORS)** is a browser mechanism that controls whether a website can request resources (APIs) from a different domain.

* Prevents security issues.
* Fixed by setting headers on the server (Access-Control-Allow-Origin)[ACAO].

Explanation –

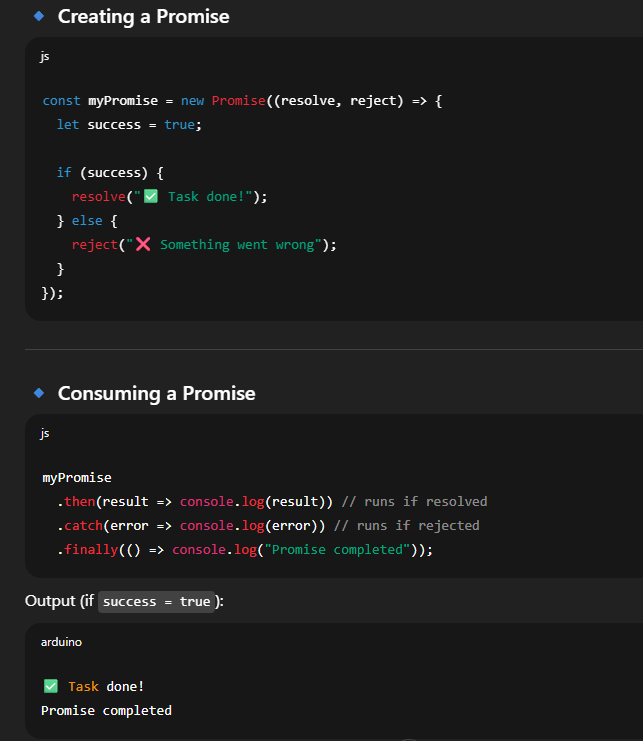


**🔹 What is a Promise?**

A **Promise** in JavaScript is an object that represents the **result of an asynchronous operation** (something that takes time, like API calls, DB query, file read, etc).

It has 3 states:

1. **Pending** → still running (not yet finished).
2. **Fulfilled** → completed successfully → returns a value (resolve).
3. **Rejected** → failed → returns an error (reject).

****

**🔹 Why Promises?**

Without Promises, we used **callbacks** → which caused **callback hell** (messy nested functions).  
Promises solve that by giving a clean way to handle async tasks.

**🔹 How async/await fits in?**

* A Promise **represents a future value**.
* await makes JavaScript wait for that future value before moving on.
* Basically, async/await is just **syntactic sugar** for Promises (a cleaner way of writing .then() and .catch()).

✅ **Interview One-liner:**  
*“A Promise is an object that represents the eventual success or failure of an asynchronous operation. It provides .then() for success, .catch() for errors, and .finally() for cleanup.”*

**🔟 What is async and await?**

* **async** → makes a function return a Promise.
* **await** → pauses code execution until the promise resolves.

Example:

A screen shot of a computer code

AI-generated content may be incorrect.

**1️⃣1️⃣ What is the use of const json = await data.json(); in getRestaurants()?**

* data.json() converts the **raw HTTP response** (stream) into a **JavaScript object** (JSON format).
* await ensures conversion is complete before assigning.

A screen shot of a computer code

AI-generated content may be incorrect.