1. What is a Microservice?

In a large-scale application, there are multiple components such as the user interface (UI), backend services, messaging systems, email services, etc. Traditionally, all of this functionality was bundled together and hosted in a single environment — a pattern known as **monolithic architecture**. In such systems, even a small change in one part of the application required rebuilding and redeploying the entire system, which made development and maintenance slow and error-prone.

Microservices architecture solves this problem by breaking down the application into smaller, independent services. Each component of the application — such as the UI, authentication, database interaction, or email handling — is developed, deployed, and maintained as an independent service with its own codebase and sometimes even its own repository. These independently managed components are called **microservices**.

This architectural style improves scalability, allows teams to work in parallel on different services, and makes the application more resilient and easier to maintain.

2. What is Monolith architecture?

In a large-scale application, there are many components such as the user interface (UI), backend logic, messaging systems, email services, and more. In a monolithic architecture, all these components are bundled together into a single codebase and deployed as one unit in a single environment.

This means that even a small change in any part of the application requires rebuilding and redeploying the entire application. This can slow down development, increase the risk of bugs, and make scaling more difficult.

Monolithic architecture was traditionally common but is increasingly being replaced by microservices for greater flexibility and maintainability.

3. Why do we need and use the useEffect hook ?

The useEffect hook is one of the most important **side-effect hooks** in React. It allows us to perform side effects in function components — such as fetching data from an API, setting up a subscription, or updating the DOM.

We use useEffect when:

- We want to run a piece of code **once** when the component is first rendered (on mount).
- We want to re-run that code **only when specific dependencies change**, rather than on every render.

- We want to clean up something (like removing an event listener or canceling a timer) when the component unmounts.

The useEffect hook takes two parameters:

1. A **callback function** that contains the side-effect logic.
2. A **dependency array** — an array of values that the effect depends on.

The effect runs:

- **After the initial render**, and
- **Every time** any value in the dependency array changes.

If the dependency array is empty ([]), the effect runs **only once**, when the component mounts.

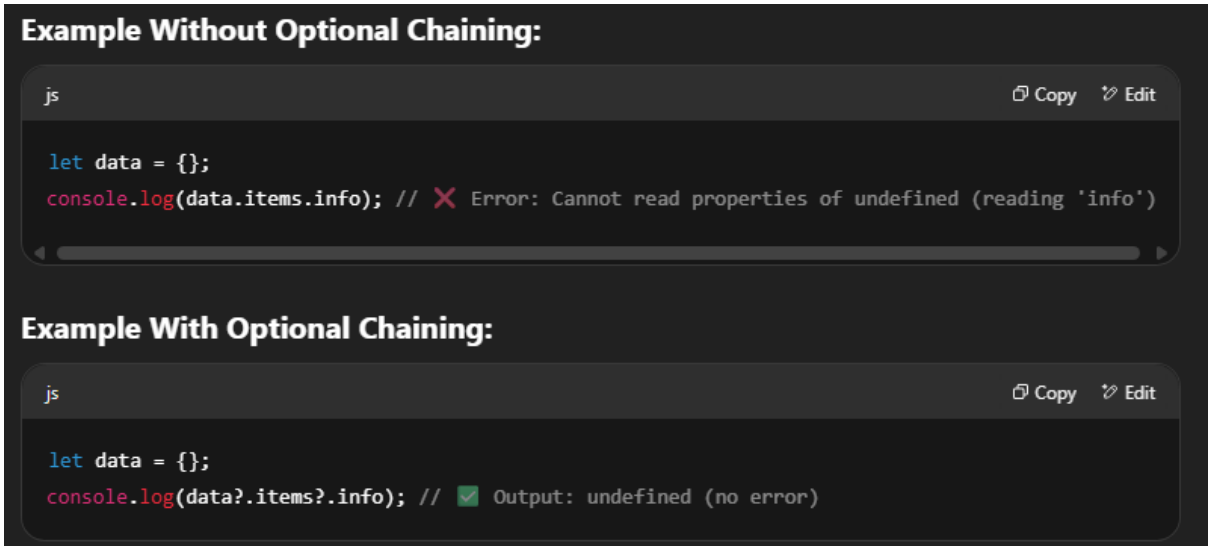
4. What is the difference between Monolith and Microservice?

Aspect	Monolithic Architecture	Microservices Architecture
Structure	All components (UI, backend, database, etc.) are part of a single codebase.	Application is divided into multiple independent services, each handling a specific feature.
Development	Teams work on the same codebase, which can lead to dependency conflicts.	Teams can work independently on different services.
Deployment	Entire application must be rebuilt and redeployed even for small changes.	Only the modified service needs to be redeployed.
Scalability	Difficult to scale individual components.	Each service can be scaled independently based on demand.
Fault Isolation	A failure in one part can bring down the whole application.	Failures are isolated; one service going down doesn't affect others.
Technology Stack	Typically uses a single technology stack for the whole application.	Different services can use different tech stacks best suited for their job.
Code Management	Single repository and codebase.	Separate repositories or codebases for each service.

Aspect	Monolithic Architecture	Microservices Architecture
Example Use Case	Small or simple applications.	Large, complex, or enterprise-level applications.

5. What is Optional Chaining?

Optional chaining (?.) is a JavaScript feature that allows you to safely access deeply nested object properties **without causing an error** if one of the intermediate properties is null or undefined. Normally, if you try to access a property on undefined or null, JavaScript throws a **TypeError**. Optional chaining prevents that by **short-circuiting** the evaluation and returning undefined instead of throwing an error.



6. What is Shimmer UI ?

Shimmer UI is a loading placeholder effect used to enhance the user experience while data is being fetched and the actual content is still loading. It displays a skeleton version of the UI with a shimmering animation, simulating the layout of the final content. This technique provides **visual feedback** to users that content is loading, which helps reduce perceived waiting time and improves overall UX.

7. What is the difference between JS expression and JS statement

JavaScript Statement: A statement is a complete instruction that performs an action. It doesn't always produce a value directly and is executed for its side effects.

```
let a = 5;              // Variable declaration statement
if (a > 3) { ... }      // Conditional statement
function greet() { ... } // Function declaration (statement)
```

JavaScript Expression: An **expression** is any valid unit of code that **produces a value**. Expressions can be assigned to variables or used inside other expressions/statements.

```
5 + 10                // Arithmetic expression → 15
"Hello" + " World"    // String expression → "Hello World"
x > 3                 // Boolean expression → true or false
let sum = (a, b) => a + b;  // Function expression
```

8. What is CORS and why is it needed ?

CORS (Cross-Origin Resource Sharing) is a browser security feature that allows or blocks web applications from making requests to a different origin (domain, protocol, or port) than the one from which the application was loaded. It is enforced due to the Same-Origin Policy, which restricts cross-origin HTTP requests for security. When a request is made to a different origin, the browser may send a preflight OPTIONS request to check if the target server allows the actual request. The server must respond with specific headers like Access-Control-Allow-Origin to indicate which origins are permitted. If the headers are valid, the browser proceeds with the request; otherwise, it blocks it, even if the server responds successfully.

9. What is async-await in JS ?

async and await are JavaScript keywords used to handle asynchronous operations in a cleaner, more readable way. Declaring a function with async makes it return a **promise**, and using await inside it pauses the function's execution *only within that function*. When await is encountered, the async function is **suspended and moved out of the call stack**, allowing the rest of the program to continue executing — JavaScript itself never pauses. Once the awaited promise is resolved or rejected, the function is queued back onto the call stack to **resume execution**. This behavior creates the illusion of synchronous flow while still being non-blocking under the hood. await must always be used inside an async function.

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

In getRestaurants(), the line const json = await data.json(); is used to **extract the response body as JSON** after making a fetch() request. The data.json() method itself returns a **promise**, because parsing the response body is an asynchronous operation. If we don't use await, we'll get a **pending promise**, not the actual parsed data. By using await, we **pause execution within the async function** until the JSON is fully parsed and can be assigned to json, allowing us to process it further.