1. **What is Microservice?**

* Microservices - also known as the microservice architecture - is an architectural style that structures an application as a collection of services that are
  + Independently deployable
  + Loosely coupled
  + Organized around business capabilities
  + Owned by a small team
  + Highly maintainable and testable
* The microservice architecture enables the rapid, frequent and reliable delivery of large, complex applications. It also enables an organization to evolve its technology stack.

1. **What is Monolith architecture?**

* Monolithic Architecture is like a big container, wherein all the software components of an app are assembled and tightly coupled, i.e., each component fully depends on each other.
* Monolith means composed all in one piece.
* The **Monolithic** application describes a single-tiered **software** application in which different components combined into a single program from a single platform.

1. **What is difference between Monolithic and Micro services?**

* **Monolithic** –
  + Benefits:
    - Simple to develop — At the beginning of a project it is much easier to go with Monolithic Architecture.
    - Simple to test. For example, you can implement end-to-end testing by simply launching the application and testing the UI with Selenium.
    - Simple to deploy. You have to copy the packaged application to a server.
    - Simple to scale horizontally by running multiple copies behind a load balancer.
  + Drawbacks:
    - Maintenance — If Application is too large and complex to understand entirely, it is challenging to make changes fast and correctly.
    - The size of the application can slow down the start-up time.
    - You must redeploy the entire application on each update.
    - Monolithic applications can also be challenging to scale when different modules have conflicting resource requirements.
    - Reliability — Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug impact the availability of the entire application
    - Regardless of how easy the initial stages may seem, Monolithic applications have difficulty to adopting new and advance technologies. Since changes in languages or frameworks affect an entire application, it requires efforts to thoroughly work with the app details, hence it is costly considering both time and efforts.
* **Microservices** –
  + Benefits:
    - Microservices Enables the continuous delivery and deployment of large, complex applications.
    - Better testability — services are smaller and faster to test.
    - Better deployability — services can be deployed independently.
    - It enables you to organize the development effort around multiple teams. Each team is responsible for one or more single service. Each team can develop, deploy and scale their services independently of all of the other teams.
    - Each microservice is relatively small
    - Comfortable for a developer to understand
    - The IDE is faster making developers more productive
    - The application starts faster, which makes developers more productive, and speeds up deployments
    - Improved fault isolation. For example, if there is a memory leak in one service then only that service is affected. The other services continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
    - Microservices Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack.
  + Drawbacks:
    - Developers must deal with the additional complexity of creating a distributed system.
    - Developer tools/IDEs are oriented on building monolithic applications and don’t provide explicit support for developing distributed applications.
    - Testing is more difficult as compared to Monolith applications.
    - Developers must implement the inter-service communication mechanism.
    - Implementing use cases that span multiple services without using distributed transactions is difficult.
    - Implementing use cases that span multiple services requires careful coordination between the teams.
    - Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.
    - Increased memory consumption. The microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its Container, which is usually necessary to isolate the instances, then there is the overhead of M times as many Containers.

1. **Why do we need a useEffect Hook?**

* You’ve likely performed data fetching, subscriptions, or manually changing the DOM from React components before.
* We call these operations “side effects” (or “effects” for short) because they can affect other components and can’t be done during rendering.
* The Effect Hook, useEffect, adds the ability to perform side effects from a function component.
* It serves the same purpose as componentDidMount, componentDidUpdate, and componentWillUnmount in React classes, but unified into a single API.
* When you call useEffect, you’re telling React to run your “effect” function after flushing changes to the DOM.
* Effects are declared inside the component so they have access to its props and state.
* By default, React runs the effects after every render — *including* the first render.

1. What is Optional Chaining?

* The optional chaining ?. is a safe way to access nested object properties, even if an intermediate property doesn’t exist.
* The optional chaining ?. stops the evaluation if the value before ?. is undefined or null and returns undefined.
* In other words, value?.prop:
  + works as value.prop, if value exists,
  + otherwise (when value is undefined/null) it returns undefined.

1. What is Shimmer UI?

* Shimmer is a temporary animation placeholder for when a service call takes time to return data and we don't want to block rendering the rest of the UI.

1. What is the difference between JS statements and expressions?

* Any unit of code that can be evaluated to a value is an expression.
* Since expressions produce values, they can appear anywhere in a program where JavaScript expects a value such as the arguments of a function invocation.
* A statement is an instruction to perform a specific action.
* Such actions include creating a variable or a function, looping through an array of elements, evaluating code based on a specific condition etc.
* JavaScript programs are actually a sequence of statements.

1. What is conditional rendering? Explain with one example.

* In React, you can create distinct components that encapsulate behavior you need.
* Then, you can render only some of them, depending on the state of your application.
* Conditional rendering in React works the same way conditions work in JavaScript.
* Use JavaScript operators like [if](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/if...else) or the [conditional operator](https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Operators/Conditional_Operator) to create elements representing the current state, and let React update the UI to match them.
* Ex: function Greeting(props) {

const isLoggedIn = props.isLoggedIn;

if (isLoggedIn) { return <UserGreeting />; }

return <GuestGreeting />;}

const root = ReactDOM.createRoot(document.getElementById('root'));

// Try changing to isLoggedIn={true}:

root.render(<Greeting isLoggedIn={false} />);

}

1. What is CORS?

* “CORS” stands for **C***ross*-**O***rigin* **R***esource* **S***haring*.
* It allows you to make requests from one website to another website in the browser, which is normally prohibited by another browser policy called the Same-Origin Policy (SOP).
* **Cross-Origin Resource Sharing** ([CORS](https://developer.mozilla.org/en-US/docs/Glossary/CORS)) is an [HTTP](https://developer.mozilla.org/en-US/docs/Glossary/HTTP)-header based mechanism that allows a server to indicate any [origins](https://developer.mozilla.org/en-US/docs/Glossary/Origin) (domain, scheme, or port) other than its own from which a browser should permit loading resources.
* CORS also relies on a mechanism by which browsers make a "preflight" request to the server hosting the cross-origin resource, in order to check that the server will permit the actual request.
* In that preflight, the browser sends headers that indicate the HTTP method and headers that will be used in the actual request.

1. What is async and await?

* The async function declaration declares an async function where the await keyword is permitted within the function body.
* The async and await keywords enable asynchronous, promise-based behavior to be written in a cleaner style, avoiding the need to explicitly configure promise chains.
* The async keyword before a function has two effects:
  + Makes it always return a promise.
  + Allows await to be used in it.
* The await keyword before a promise makes JavaScript wait until that promise settles, and then:
  + If it’s an error, an exception is generated — same as if throw error were called at that very place.
  + Otherwise, it returns the result.
* Together they provide a great framework to write asynchronous code that is easy to both read and write.

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

* Data.json() is a method on the Data object that lets you extract a JSON object from the response. The method returns a promise, so you have to wait for the JSON: await data.json() .