**Assignment 6 – Exploring the world**

Q1: What is a Microservice?

Microservice architecture is an architectural style used in software development where an application is composed of multiple loosely coupled services. Each service is designed to perform a specific business function and can be developed, deployed, and scaled independently. These services communicate with each other through well-defined APIs, often over lightweight protocols like HTTP or messaging queues.

Key characteristics of Microservice architecture:

1. Decomposition: small, focused services each responsible for a specific task.
2. Independence: independent development without affecting other parts of the app.
3. Scalability: Horizontal scaling by adding more instances of the service in demand.
4. Resilience: Services are designed to bounce back after failures. Since the services are isolated, failures in one service are less likely to impact the overall system.
5. Autonomy: Development teams have autonomy over the design, development and deployment of individual microservice. This ensures faster development. Can use different technology stacks.
6. Polyglot Persistence: Each microservice can have its own data storage technology.
7. Continuous Delivery: Microservices often employ CI/CD practices to automate the deployment pipeline.

Q2: What is Monolith architecture?

Monolithic architecture is an older, traditional architectural style used in software development, where the entire application is built as a single, self-contained unit. In a monolithic architecture, all functionality is tightly integrated and deployed together as a single executable or deployment artifact.

Key characteristics of Monolithic architecture:

1. Single Codebase – All layers of the app are within the same codebase.
2. Tight Coupling – Changes in one part can have ripple effects in another.
3. Monolithic deployment – deployed as one single unit usually on a single server. Change in single file requires re-deployment of the entire app.
4. Scalability – scaled by replicating the entire app across multiple servers. Inefficient resource usage.
5. Development Process -Entire team works on the same codebase and use of version controlling systems.
6. Technology Stack-Entire application is built using the same tech stack thus developers have limited flexibility in choosing the best tools and technologies.

Q3: What is the difference between Monolith and Microservice?

All points covered in Q1 and Q2

1. **Architecture Style**:
   * **Monolithic Architecture**: In monolithic architecture, the entire application is built as a single, self-contained unit. All functionality, including the user interface, business logic, and data access layers, is tightly integrated and deployed together.
   * **Microservice Architecture**: In microservice architecture, the application is decomposed into small, independent services, each responsible for a specific business function. These services are loosely coupled and can be developed, deployed, and scaled independently.
2. **Size and Complexity**:
   * **Monolithic Architecture**: Monolithic applications tend to be larger and more complex, as all functionality is contained within a single codebase or deployment artifact.
   * **Microservice Architecture**: Microservices are smaller and more focused, with each service responsible for a specific aspect of the overall functionality. This decomposition simplifies development and maintenance, as teams can work on smaller, more manageable services.
3. **Deployment Unit**:
   * **Monolithic Architecture**: In monolithic architecture, the entire application is deployed as a single unit. Updates or changes to the application require redeploying the entire monolith.
   * **Microservice Architecture**: Microservices are deployed independently, allowing updates or changes to be deployed to individual services without affecting the entire application. This enables faster iteration and deployment.
4. **Scalability**:
   * **Monolithic Architecture**: Scaling monolithic applications can be challenging, as all components are tightly coupled and need to be scaled together. This can lead to inefficient resource usage.
   * **Microservice Architecture**: Microservices can be scaled independently based on demand, allowing resources to be allocated more efficiently. This fine-grained scalability enables better performance and cost optimization.
5. **Technology Stack**:
   * **Monolithic Architecture**: Monolithic applications are typically built using a single technology stack, limiting flexibility and innovation.
   * **Microservice Architecture**: Microservices can be developed using different programming languages, frameworks, or technologies, allowing teams to choose the best tools for each service. This promotes innovation and flexibility.
6. **Development and Maintenance**:
   * **Monolithic Architecture**: Monolithic applications can become difficult to maintain and extend as they grow larger and more complex. Changes to one part of the application can have unintended consequences on other parts.
   * **Microservice Architecture**: Microservices are easier to maintain and extend, as changes to one service have minimal impact on other services. This promotes agility and enables faster iteration.

Q4: Why do we need a useEffect Hook?

useEffect is a React hook that lets you synchronize a component with an external system. An external system means any piece of code that is not controlled by React such as:

1. A 3rd-party API like the Swiggy API
2. A timer managed with setInterval() and clearInterval()
3. An event subscription using window.addEventListener() and window.removeEventListener()
4. A 3rd-party animation library with an API like animation.start() and animation.reset() .

For example, you might want to control a non-React component based on the React state, set up a server connection, or send an analytics log when a component appears on the screen. Effects let you run some code after rendering so that you can synchronize your component with some system outside of React.

1. Import useEffect from React - import { useEffect } from 'react';
2. Call it at the top-level of your component.

function MyComponent() {

useEffect(() => {

// Put your code here – called Side effect. Code here will run after \*every\* render

return ()=>{

//optional – clean up code.Disconnecting from DB, clearing timers etc

}

});

return <div />; //JSX for rendering the component

}

1. **The code that is written to connect to the external system is called Side effect**. In our application the side effect is fetching data from Swiggy.
2. Optionally return a cleanup function from useEffect(). This cleanup function runs when the Component is removed from the page(unmounted) or before running the effect again(if any dependencies change).
3. If your Effect only adjusts some state based on other state, you might not need an Effect.

Q5: What is Optional Chaining?

The Optional Chaining (?.) operator is a feature introduced in ES11, that accesses an object's property or calls a function. If the object accessed or function called using this operator is undefined or null, the expression short circuits and evaluates to **undefined** instead of throwing an error.

const obj = {

property1: {

property2: {

property3: ‘Hello’

}

}

};

// Without optional chaining

const value = obj.property1 && obj.property1.property2 && obj.property1.property2.property3;

// With optional chaining

const valueWithOptionalChaining = obj.property1?.property2?.property3; // Hello

------------------------------------------------------------------------------------------------

const arr = [1, 2, 3];

// Without optional chaining

const element = arr[5] !== undefined ? arr[5] : 'Element does not exist';

// With optional chaining

const elementWithOptionalChaining = arr?.[5]; // undefined

-------------------------------------------------------------------------------------------------

const obj = {

func: () => 'Hello'

};

// Without optional chaining

const result = obj.func ? obj.func() : undefined;

// With optional chaining

const resultWithOptionalChaining = obj.func?.(); // 'Hello'

Q6: What is Shimmer UI?

Shimmer UI is a technique used in web applications where the user can visualize that there is something happening in the background while data is being fetched asynchronously (and takes some time).

Earlier a blank screen or a loading spinning wheel were displayed. This showcases bad UX. Instead, Shimmer UI provides visual feedback while the content is being loaded. CSS animations or libraries/frameworks (react-loading-skeleton or react-content-loader) provide ready-to-use components for creating shimmer effects.

Q7: What is the difference between JS expression and JS statement.

JS Expression is **any valid unit of code that resolves to a value**. It can be a single value, a variable, or a combination of values, variables, operators, and function calls that evaluate to a single value. Expressions can appear where Javascript expects a value – assignment statement, function argument (function invocation), within other expressions.

5 + 3

myVar

myFunction()

10 > 5

JS Statement : is a larger piece of code **that performs some action**. It typically consists of one or more expressions and is executed sequentially. They don’t produce a value directly (though may produce indirectly )

var x = 5; // Variable declaration and assignment statement

if (x > 0) { // Conditional statement

} else { }

for (var i = 0; i < 5; i++) { // Loop statement }

**Hence, Expressions produce a value, statements perform some action.**

**In React, JSX expressions are used to define the structure and content of UI components, while JavaScript statements are used to define the behavior and logic within those components.**

JSX expressions are JS expressions that are used within JSX syntax to represent the structure and content of UI components. These include variables, function calls, and JS expressions enclosed in curly braces {}.

Example of JSX expression:

const name = "John";

const greeting = <p>Hello, {name}</p>;

{handleClick}

In the above example, {name} is a JSX expression that evaluates to the value of the ‘name’.

React Component Statements: React components are typically defined as classes or functions. Statements within React components include variable declarations, function declarations, conditional statements, loops, and method calls.

Example of React component statement (using functional components):

function MyComponent() {

const handleClick = () => console.log('Button clicked'); //statement

return (

<div>

<button onClick={handleClick}>Click Me</button>

</div>

);

}

In this example, the handleClick function declaration is a statement within the functional component MyComponent.

<https://react-cn.github.io/react/tips/if-else-in-JSX.html>

Q8: What is Conditional Rendering, explain with a code example.

Often the Component will display different things based on some condition. In React, rendering JSX based on a condition is Conditional Rendering.

{isLoggedIn ? (

    <button className="logout-btn" onClick={() => setLoggedIn(false)}>

      Logout

    </button>

  ) : (

    <button className="login-btn" onClick={() => setLoggedIn(true)}>

      Login

    </button>

  )}

Here, if the value of isLoggedIn is true Logout button is displayed otherwise Login button.

Q9: What is CORS?

To understand CORS – take the example of our application. From <http://localhost:1234> we are making a fetch request to <https://www.swiggy.com/dapi/restaurants/list/v5?lat=12.9351929&lng=77.62448069999999&page_type=DESKTOP_WEB_LISTING>

Two URLS have the same origin if the protocol(or scheme), host, and port are same for both. In this case, all the 3 values are different. Hence, ‘same-origin policy’ is not being followed here. [http:// is port 80 by default]

For security reasons, browsers follow the same-origin policy, thereby restricting cross-origin HTTP requests initiated from scripts. If our APIs were from localhost, there wouldn’t have been any problem. So, to get the response from other servers, we need to provide the appropriate CORS headers.

According to MDN - Cross Origin Resource sharing (CORS) is an HTTP-header based mechanism that allows a server to indicate origins(protocol(or Scheme), domain, port) other than its own to permit loading resources.

In other words, CORS works by adding new HTTP header(in the response) that lets the destination server describe which origins are permitted to send request to fetch information from it.

Example of access control scenarios:

1. Simple request:

A simple request is a request without any options method.

For example, suppose web content at https://foo.example wishes to fetch JSON content from domain <https://swiggy.other>

const fetchPromise = fetch("https://swiggy.other");

fetchPromise

.then((response) => response.json())

.then((data) => console.log(data));

This operation performs a simple data exchange between the server and the client, using CORS headers to handle the privileges:



The request header has a lot of info – Origin shows where the invocation is coming from. The response returns header with Status:200 along with “Access-Control-Allow-Origin: \*”, which means that the invoking resource can be from any origin.

Note: if the server wants to restrict requests only from “https://foo.example “, then the server would send the response header as Access-Control-Allow-Origin: https://foo.example

1. Preflight request:

Unlike simple request, where the request is sent with the actual data, here in preflight request, first an HTTP request is sent using the OPTIONS method to the server, to determine if the actual request is safe to send.

Example, requests containing passwords to bank accounts, money transactions etc.

const fetchPromise = fetch("https://bar.other/doc", {

method: "POST",

mode: "cors",

headers: {

"Content-Type": "text/xml",

"X-PINGOTHER": "pingpong",

},

body: "<person><name>Arun</name></person>",

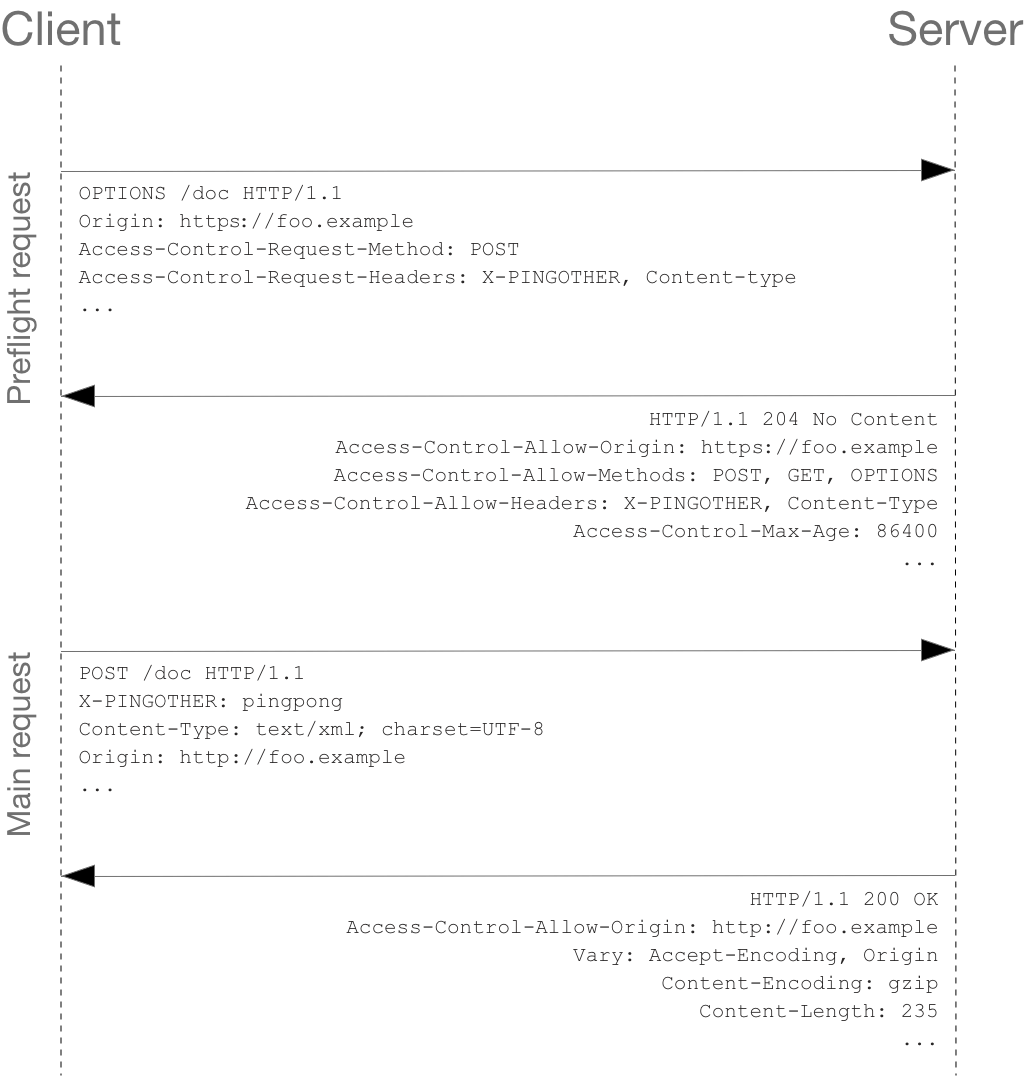
});

fetchPromise.then((response) => {

console.log(response.status);

});

Since the code above has a custom header, this request is preflighted.



OPTIONS is an HTTP/1.1 method that is used to determine further information from servers, and is a safe method, meaning that it can't be used to change the resource.

In the Preflight request, we send headers that we going to make a POST request from client origin. Check the OPTIONS response header.

After this the main request is sent.

Note: the actual POST request does not include the Access-Control-Request-\* headers.

Refer <https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS>

Q10: What is async and await?

The async/await syntax is a special syntax created to work with Promise objects. It makes our code simpler, and clearer.

When handling a Promise, you need to call the function that returns a Promise and then chain with then/catch methods. In case there are subsequent promises, the chaining increases.

fetch(SWIGGY\_APP\_URL)

    .then(response=>response.json())

    .then(json=> {

console.log(json)

//…some other code})

    .catch(err=>console.log(err))

The async keyword gives you a simpler way to work with asynchronous promise-based code. Adding async at the start of a function makes it an async function. It operates via the event-loop.

Inside an async function, you can use the await keyword before a function call that returns a promise. This makes the code wait at that point until the promise is settled, at which point the fulfilled value of the promise is treated as a return value, or the rejected value is thrown.

const getRestaurants = async () => {

const response = await fetch(SWIGGY\_APP\_URL);

const json = await response.json();

}

Q11: What is the use of `const json = await response.json();` in getRestaurants()?

The data object, returned by the await fetch(), is a generic placeholder for multiple data formats. so we can extract the JSON object from a fetch response by using await data.json(). data.json() is a method on the data object that lets you extract a JSON object from the data or response. The method returns a promise because we have used await keyword. So data.json() returns a promise resolved to a JSON object.

const getRestaurants = async () => {

    const response = await fetch(SWIGGY\_APP\_URL);

    const json = await response.json();

}

Notes:

Dependency Array: The role of dependency array is to control when the effect should run.

Reactive values include props and all variables and functions declared directly inside of your component.

1. If you specify the dependencies, your Effect runs **after the initial render** *and* **after re-renders with changed dependencies.**

useEffect(() => {

// ...

}, [a, b]); // Runs again if a or b are different

1. An Effect with empty dependencies doesn’t re-run when any of your component’s props or state change.

If your Effect truly doesn’t use any reactive values, it will only run after the initial render.

useEffect(() => {

// ...

}, []); // Does not run again (except once in development)

1. If you pass no dependency array at all, your Effect runs after every single render (and re-render) of your component.

useEffect(() => {

// ...

}); // Always runs again and again