Assignment 5 - Exploring the World

1. What is a Microservice?

In software development, a microservice is an **architectural style** for building applications as a collection of small, independent services. These services are:

- **Independently deployable**: They can be developed, tested, and deployed individually, without affecting other services.
- **Loosely coupled**: They communicate with each other through well-defined interfaces, typically APIs (Application Programming Interfaces), and have minimal dependencies on each other.
- **Focused on business capabilities**: Each service typically handles a specific, well-defined function, like user authentication, product management, or order processing.

Benefits of using Microservices:

- **Increased agility and flexibility**: Makes it easier to make changes and add new features to the application, without impacting other parts.
- **Improved scalability**: Allows individual services to be scaled independently based on their specific needs.
- **Enhanced fault tolerance**: If one service fails, it doesn't bring down the entire application.
- **Technology independence**: Different services can be written in different languages and technologies, allowing teams to choose the best tools for the job.

However, there are also some drawbacks:

- **Increased complexity**: Requires more planning and coordination to develop and manage many separate services.
- **Distributed complexity**: Issues can be harder to identify and debug due to the distributed nature of the application.
- **Testing challenges**: Integration testing between multiple services can be complex.

2. What is Monolith architecture?

In software development, Monolithic architecture describes the traditional method of building applications as **single, unified units**. Unlike microservices, which break down functionalities into independent services, monoliths contain **all components** (like user interface, business logic, and data storage) **tightly coupled within a single codebase**.

Key characteristics of Monolithic architecture:

- **Single codebase**: All functionalities are housed within one large codebase, making it easier to understand and manage initially.
- **Tight coupling**: Components are heavily dependent on each other, often sharing resources and libraries.
- **Centralized deployment**: The entire application needs to be deployed and updated as a whole.
- **Single technology stack**: Typically uses a single programming language and set of tools for the entire application.

Pros of Monolithic architecture:

- **Simple to develop and understand**: Easier to get started as everything is in one place.
- **Fast initial development**: Faster initial development due to tight coupling and familiarity with the single technology stack.
- **Easy debugging**: Troubleshooting issues is often simpler since dependencies are well-defined within the single codebase.

Cons of Monolithic architecture:

- **Limited scalability**: Scaling becomes difficult as the application grows, as changes require updating the entire codebase.
- **Reduced agility**: Making changes to one part of the application can impact other parts due to tight coupling.
- **Maintenance challenges**: Maintaining and updating a large codebase can be time-consuming and error-prone.
- **Single point of failure**: An issue in one part can potentially bring down the entire application.

3. **What is the difference between Monolith and Microservice?**

- **Architecture:**

Monolithic:
Single, tightly integrated unit.
All components and modules run within the same process space.
Components are tightly interconnected and share the same database.

Microservices:
Composed of small, independent services.
Each service runs in its own process space.
Services communicate with each other through well-defined APIs.

- **Deployment:**
  Monolithic:
  Deployed as a single unit.
  Updates or changes require redeploying the entire monolith.

  Microservices:
  Independently deployable services.
  Each service can be updated and deployed independently without affecting the entire system.

- **Scaling:**
  Monolithic:
  Scaling involves replicating the entire monolith, which can lead to inefficiencies.

  Microservices:
  Can scale individual services independently based on demand.

- **Technology Stack:**
  Monolithic:
  Consistent technology stack throughout the application.

  Microservices:
  Each service can use its own technology stack, chosen based on specific requirements.

- **Development and Maintenance:**
  Monolithic:
  Can become challenging for large and complex applications.
  Changes to one part may impact the entire application.

Microservices:
Easier to develop and maintain due to smaller, focused codebases.
Changes to one service do not necessarily impact others.

- **Fault Isolation:**
  Monolithic:
  Failure in one component may affect the entire application.

  Microservices:
  Faults are isolated to individual services, limiting the impact on the overall system.

- **Communication:**
  Monolithic:
  Components communicate directly within the same process.

  Microservices:
  Services communicate through well-defined APIs, often using lightweight protocols like HTTP/REST or message queues.

- **Database:**
  Monolithic:
  Typically shares a single database.

  Microservices:
  Each service can have its own database, allowing for more autonomy.

4. **Why do we need useEffect hook?**

- **Lifecycle Management:** useEffect mirrors lifecycle methods (e.g., componentDidMount) for functional components.
- **Declarative Side Effects:** Express side effects declaratively, specifying post-render tasks without worrying about execution timing.
- **Avoid Memory Leaks:** Ensures cleanup of resources (e.g., subscriptions) to prevent memory leaks upon unmounting or updates.
- **Data Fetching:** Facilitates asynchronous data fetching post-render, improving component interaction.

- **Dependency Tracking:** Tracks dependencies, re-executing the effect only when specified values change, optimizing performance.
- **Separation of Concerns:** Encourages a clean separation of logic related to side effects from the component's rendering logic

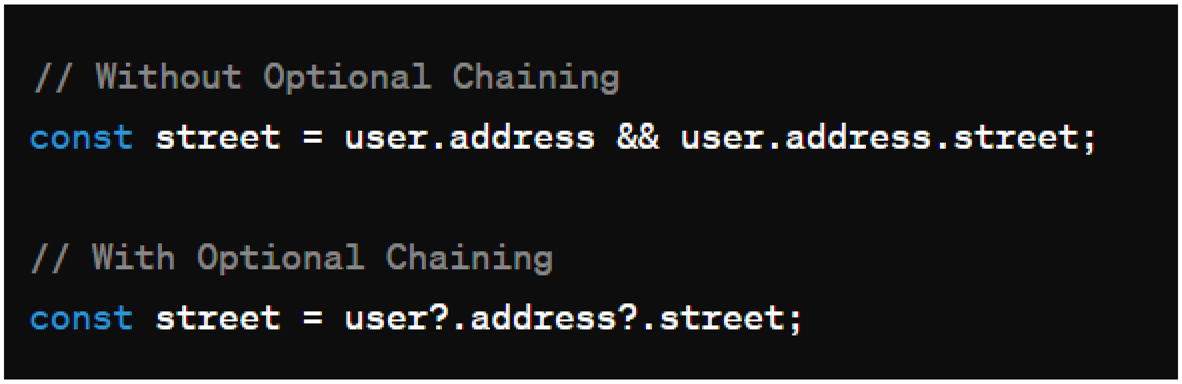
5. **What is Optional Chaining?**

Optional Chaining is a JavaScript feature that simplifies accessing nested properties of objects, even when intermediate properties might be null or undefined. It uses the ?. syntax to safely navigate through object properties without causing errors if a property along the chain is null or undefined.

Key Points:

- **Safe Property Access:** Enables safe access to nested properties without explicitly checking each level for existence.

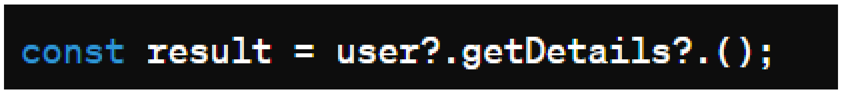
- **Short-Circuit Behavior:** If a property is null or undefined, the expression short-circuits, and the overall result is undefined.

- 

```
// Without Optional Chaining
const street = user.address && user.address.street;

// With Optional Chaining
const street = user?.address?.street;
```

- **Function Calls:** It can also be used with functions, preventing errors if the function is not present.



```
const result = user?.getDetails?.();
```

- **Browser Compatibility:** It's a modern JavaScript feature **(ECMAScript 2020)**, so ensure compatibility or use transpilers (like Babel) for broader support.

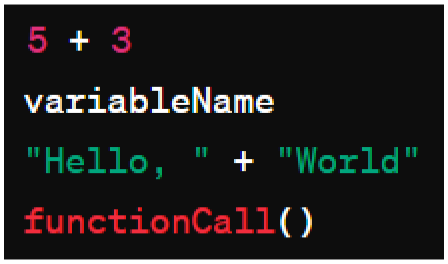
6. What is Shimmer UI?

Shimmer UI is a design technique used to indicate loading states in applications. It uses animated placeholders that mimic the layout of the actual content, creating a “shimmering” effect. This improves perceived performance, provides visual cues, and enhances user satisfaction by creating a smooth transition from loading to loaded state. Examples include Facebook, LinkedIn, and Netflix.

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

Expression:

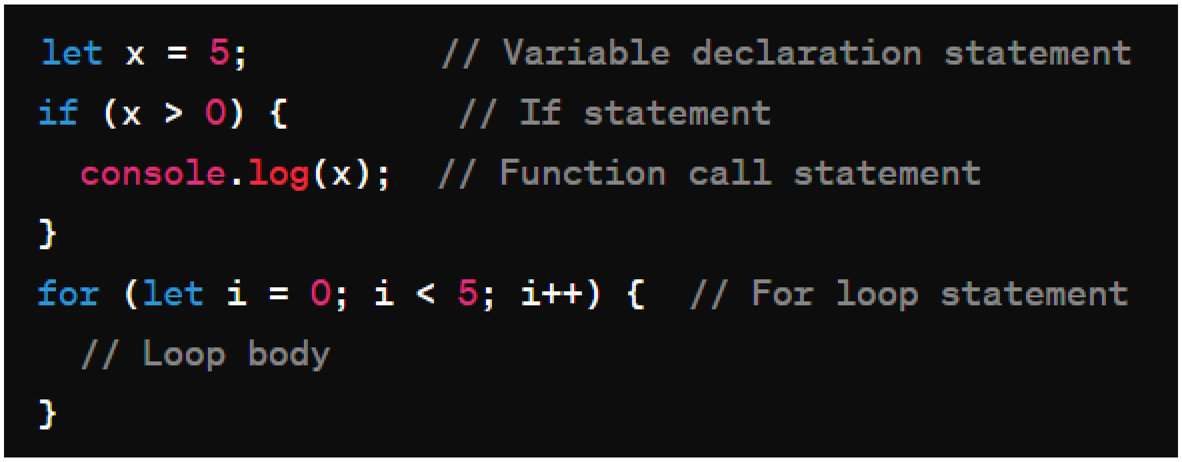
An expression is a piece of code that produces a value. It can be a combination of variables, operators, literals, and function calls that results in a single value.
Examples of expressions include:



```
5 + 3
variableName
"Hello, " + "World"
functionCall()
```

Statement:

A statement is a larger unit of code that performs a specific action. It doesn't necessarily produce a value. Statements are executed one after the other, and they often include expressions.
Examples of statements include:

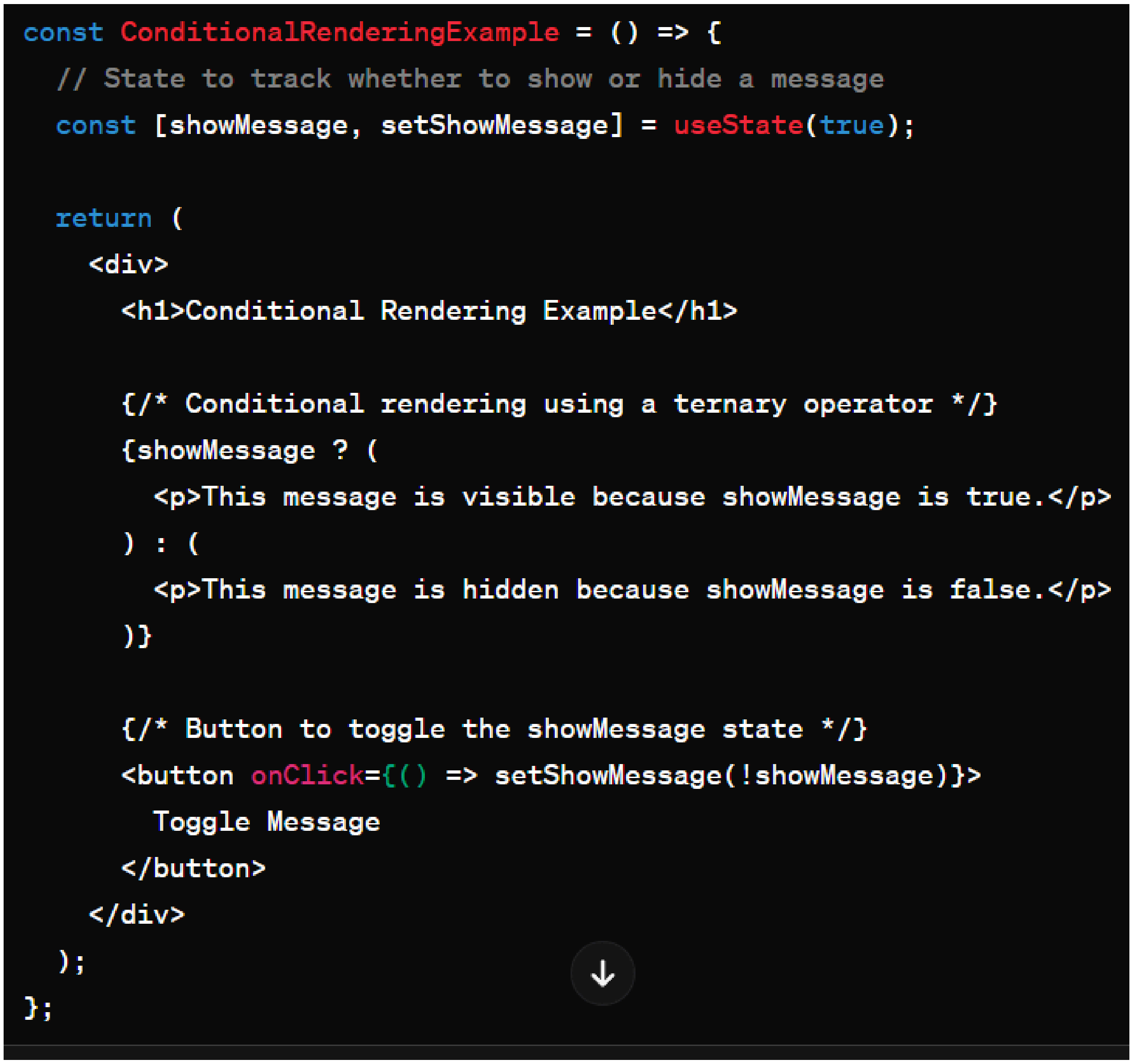


```
let x = 5;          // Variable declaration statement
if (x > 0) {         // If statement
  console.log(x);  // Function call statement
}
for (let i = 0; i < 5; i++) {  // For loop statement
  // Loop body
}
```

Therefore, expressions produce values, and statements are larger units of code that perform actions. In many cases, expressions are used within statements to perform specific tasks.

8. **What is Conditional Rendering, explain with a code example?**

Conditional rendering refers to the process of displaying different content or components in a user interface based on certain conditions or criteria. In JavaScript and frameworks like React, conditional rendering is often achieved using conditional statements or ternary operators.



```
const ConditionalRenderingExample = () => {
  // State to track whether to show or hide a message
  const [showMessage, setShowMessage] = useState(true);

  return (
    <div>
      <h1>Conditional Rendering Example</h1>

      {/* Conditional rendering using a ternary operator */}
      {showMessage ? (
        <p>This message is visible because showMessage is true.</p>
      ) : (
        <p>This message is hidden because showMessage is false.</p>
      )}

      {/* Button to toggle the showMessage state */}
      <button onClick={() => setShowMessage(!showMessage)}>
        Toggle Message
      </button>
    </div>
  );
};
```

9. **What is CORS?**

CORS, or Cross-Origin Resource Sharing, is a security feature implemented by web browsers to control how web pages in one domain can request and interact with resources hosted on another domain. It is a mechanism that enables or restricts web applications running at one origin (domain) to make requests for resources from a different origin.

When a web page makes a cross-origin HTTP request (such as an API request) using XMLHttpRequest or the Fetch API, the browser enforces the Same-Origin Policy, which prevents web pages from making requests to a different domain than the one that served the web page. CORS is designed to relax these restrictions selectively.

CORS works by allowing the server to include specific HTTP headers in its response to indicate which origins are permitted to access the resources.

The main CORS headers include:

- **Access-Control-Allow-Origin**: Specifies which origins are allowed to access the resource. It can be a specific origin, a comma-separated list of origins, or the wildcard * (allowing any origin).

- **Access-Control-Allow-Methods**: Indicates the HTTP methods (e.g., GET, POST) that are permitted when accessing the resource.

- **Access-Control-Allow-Headers**: Lists the HTTP headers that can be used when making the actual request.

- **Access-Control-Allow-Credentials**: Indicates whether the browser should include credentials (e.g., cookies, HTTP authentication) when making the actual request.

- **Access-Control-Expose-Headers**: Specifies which headers are safe to expose to the response of the actual request.

To enable CORS on a server, these headers need to be configured correctly in the server's HTTP responses. It's important to note that CORS is a security feature implemented by browsers, and it doesn't affect non-browser clients (e.g., server-to-server communication).

CORS is essential for securing web applications while allowing them to interact with APIs and resources hosted on different domains.

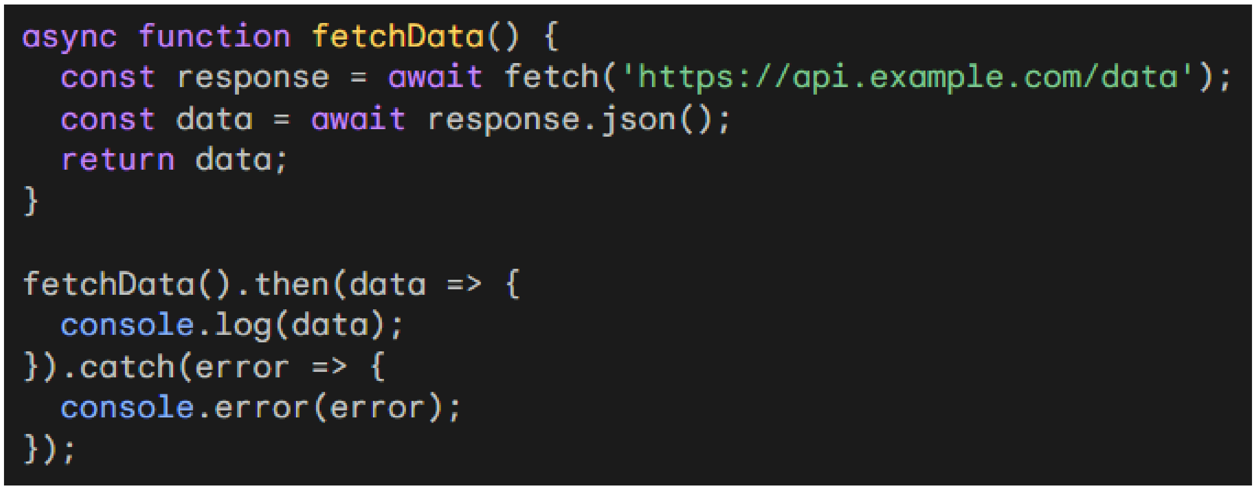
10. What is async and await?

Async and await are keywords in JavaScript that simplify working with asynchronous code. They provide a cleaner and more readable way to handle promises, which are used to represent the eventual result of an asynchronous operation. They were introduced in ECMAScript 2017 (ES8) to work with Promises and provide a more synchronous-like way of handling asynchronous operations.

async Function:
The async keyword is used to declare a function that will always return a Promise.
An async function implicitly returns a Promise that resolves to the value returned by the function or rejects with an exception thrown from within the function.

await Operator:
The await keyword is used inside an async function to wait for a Promise to resolve before continuing with the execution of the function.
It can only be used within an async function.

The await expression pauses the execution of the async function until the Promise is resolved or rejected. It effectively "waits" for the asynchronous operation to complete.



```javascript
async function fetchData() {
  const response = await fetch('https://api.example.com/data');
  const data = await response.json();
  return data;
}

fetchData().then(data => {
  console.log(data);
}).catch(error => {
  console.error(error);
});
```

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

Converting the response to JSON format:

- **data** represents the response object obtained from the fetch request to the api.
- This response is initially in a binary format that's not directly usable in JavaScript.

- The .json() method of the response object parses the response and converts it into a JavaScript Object Notation (JSON) structure.
- Assigning the result to json makes it accessible for further processing.

Making the function asynchronous:

- The await keyword before data.json() pauses the execution of the fetchData function until the promise returned by .json() resolves.
- This ensures that the code below await data.json(); only executes after the conversion to JSON is complete.
- By using await, you essentially avoid writing nested .then() and .catch() callbacks, making the code cleaner and more readable.