MODULE: 4 (New Request)

1. What is JSON

- **JSON (JavaScript Object Notation)** is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is primarily used for transmitting data between a server and a web application, often in web APIs.
- **Key Features of JSON:**
- **Text-based format**: JSON is a plain-text format that can be read and written by both humans and machines.
- **Data representation**: It represents data in a structured way using key-value pairs (similar to objects in JavaScript).
- **Language-independent**: Although JSON originated in JavaScript, it is language-independent, with parsers available for many programming languages like Python, Java, C#, etc.
- **JSON Syntax:**
- **Data is represented as key-value pairs**.
- **Data is enclosed in curly braces {}** for objects and square brackets [] for arrays.
- **Basic Structure:**
- Json
- {
- "key": "value",
- "key2": "value2"
- }
- **JSON Data Types:**
- **String**: Text enclosed in double quotes. Example: "name": "John"
- **Number**: Numeric values (no quotes). Example: "age": 30
- **Boolean**: true or false. Example: "is Student": true
- **Array**: An ordered list of values. Example: "colors": ["red", "green", "blue"]
- **Object**: A collection of key-value pairs. Example: "address": {"city": "New York", "zip code": "10001"}
- **Null**: Represents an empty value. Example: "middle Name": null
- **Example of JSON:**
- Json
- {
- "name": "Alice",
- "age": 25,
- "is Student": false,
- "skills": ["JavaScript", "React", "CSS"],
- "address": {
- "city": "New York",
- "zip code": "10001"

-   }
- }
- **Explanation:**
- **Objects**: Represented using curly braces {}. Example: "address": { "city": "New York", "zip code": "10001" }
- **Arrays**: Represented using square brackets []. Example: "skills": ["JavaScript", "React", "CSS"]
- **Keys**: Always in double quotes.
- **Values**: Can be a string, number, array, object, or null.
- **Why JSON is popular:**
- **Human-readable**: The syntax is simple and easy to understand.
- **Lightweight**: It is compact compared to other data formats like XML.
- **Widely supported**: JSON is supported by most programming languages, making it a popular choice for APIs and data transfer.
- **Common Usage:**
- **API responses**: Web APIs often send data in JSON format between a server and a client.
- **Configuration files**: JSON is used in many configuration files, such as package. Json in Node.js or settings. Json in applications.
- **Example in JavaScript:**
- To parse JSON data in JavaScript:
- JavaScript
- const Json String = '{"name": "Alice", "age": 25}';
- const parsed Data = JSON. Parse(Json String); // Converts JSON string to a JavaScript object
- console.log(parsedData.name); // Output: Alice
- To convert an object to a JSON string:
- JavaScript
- const user = { name: "Bob", age: 30 };
- const Json String = JSON.stringify(user); // Converts JavaScript object to a JSON string
- console.log(jsonString); // Output: {"name":"Bob","age":30}
- In summary, JSON is a widely used data format that facilitates easy data exchange between systems and applications, especially in web development.

2. What is promises

- **Promises** in JavaScript are a way to handle asynchronous operations. A promise represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It allows you to handle asynchronous operations more effectively than using callbacks, especially when chaining multiple asynchronous tasks.
- **Key Concepts of Promises:**
- **Pending**: The initial state of the promise. It means the operation is not yet complete.
- **Resolved (Fulfilled)**: The state when the asynchronous operation completes successfully.

- **Rejected**: The state when the asynchronous operation fails, usually due to an error.
- **Basic Syntax:**
- A promise is created using the new Promise() constructor, and you pass a function (called the "executor") to it, which has two parameters:
- resolve: A function that is called when the promise is fulfilled successfully.
- reject: A function that is called when the promise is rejected (fails).
- **Example of Creating a Promise:**
- javascript

```
let myPromise = new Promise((resolve, reject) => {
  let success = true; // Simulate success or failure

  if (success) {
    resolve("The operation was successful!");
  } else {
    reject("The operation failed!");
  }
});
```

- **Handling a Promise:**
- After creating a promise, you can use .then() to handle the resolved value, and .catch() to handle any errors if the promise is rejected.
- **Example:**
- javascript

```
myPromise
  .then((result) => {
    console.log(result); // Output: "The operation was successful!"
  })
  .catch((error) => {
    console.log(error); // Output: "The operation failed!" if rejected
  });
```

- **States of a Promise:**
- **Pending**: Initial state, neither fulfilled nor rejected.
- **Fulfilled**: The operation was successful, and the promise resolves with a value.
- **Rejected**: The operation failed, and the promise rejects with an error.
- **Example with Asynchronous Operation (Simulated Timeout):**
- javascript

```
function myAsyncFunction() {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      let success = true;

      if (success) {
        resolve("Operation completed successfully!");
```

- } else {
- reject("Something went wrong.");
- }
- }, 2000); // Simulate 2 seconds delay
- });
- }
-
- myAsyncFunction()
- .then((message) => {
- console.log(message); // Output: "Operation completed successfully!" after 2 seconds
- })
- .catch((error) => {
- console.log(error); // If rejected, output the error message
- });
- **Chaining Promises:**
- You can chain multiple .then() calls to handle sequential asynchronous tasks, where each .then() returns a new promise.
- **Example:**
- javascript
- function firstTask() {
- return new Promise((resolve) => {
- setTimeout(() => {
- resolve("First task complete");
- }, 1000);
- });
- }
-
- function secondTask() {
- return new Promise((resolve) => {
- setTimeout(() => {
- resolve("Second task complete");
- }, 1000);
- });
- }
-
- firstTask()
- .then((result) => {
- console.log(result); // "First task complete"
- return secondTask(); // Return the next promise
- })
- .then((result) => {

-     console.log(result); // "Second task complete"
-   })
-   .catch((error) => {
-     console.log("Error:", error); // Handle any errors
-   });
- **Using Promise.all:**
- Promise.all() allows you to run multiple promises in parallel and wait for all of them to complete.
- **Example:**
- javascript
- const promise1 = new Promise((resolve) => setTimeout(() => resolve("First"), 1000));
- const promise2 = new Promise((resolve) => setTimeout(() => resolve("Second"), 500));
- const promise3 = new Promise((resolve) => setTimeout(() => resolve("Third"), 2000));
-
- Promise.all([promise1, promise2, promise3])
-   .then((results) => {
-     console.log(results); // ["First", "Second", "Third"]
-   })
-   .catch((error) => {
-     console.log("Error:", error);
-   });
- **Conclusion:**
- Promises are a powerful tool for handling asynchronous code, improving readability, and preventing callback hell. They allow you to:
- Handle success and error cases separately using .then() and .catch().
- Chain multiple asynchronous operations together.
- Wait for multiple asynchronous operations to complete with Promise.all().
- Promises are essential for modern JavaScript development, especially for handling tasks like data fetching, timers, or any asynchronous operation.

3. Write a program of promises and handle that promises also

- Here's a simple example of using **Promises** in JavaScript, with promise creation, handling with .then() and .catch(), and some basic asynchronous tasks.
- **Program: Simulating Asynchronous Operations with Promises**
- javascript
- // Simulate an asynchronous operation using a Promise
- function asyncOperation(success) {
-   return new Promise((resolve, reject) => {
-     setTimeout(() => {
-       if (success) {
-         resolve("Operation was successful!");

```
      } else {
        reject("Operation failed!");
      }
    }, 2000); // Simulating a 2-second delay
  });
}

// Calling the asyncOperation function
asyncOperation(true)
  .then((result) => {
    console.log(result);  // If successful, this will run
  })
  .catch((error) => {
    console.log(error);  // If failed, this will run
  });

// Calling the asyncOperation function with failure
asyncOperation(false)
  .then((result) => {
    console.log(result);  // This will not run because the promise is rejected
  })
  .catch((error) => {
    console.log(error);  // This will run because the promise is rejected
  });
```

- **Explanation:**
- **Function asyncOperation(success)**: This function simulates an asynchronous operation. It returns a **Promise** that resolves if success is true and rejects if success is false. The operation is simulated with setTimeout(), which delays the operation for 2 seconds.
- **.then()**: This method is called if the promise resolves successfully. The result from the resolve() function is passed to the .then() block.
- **.catch()**: This method is called if the promise is rejected. The error message from the reject() function is passed to the .catch() block.
- **Output:**
- When asyncOperation(true) is called:
- After 2 seconds, it logs: "Operation was successful!"
- When asyncOperation(false) is called:
- After 2 seconds, it logs: "Operation failed!"
- **Conclusion:**
- **Promises** allow us to handle asynchronous operations in a more manageable way.
- **.then()** handles successful results.
- **.catch()** handles errors or rejections.

- This pattern makes asynchronous programming in JavaScript cleaner and more readable.

4. Use fetch method for calling an api https://fakestoreapi.com/products

- Here's how you can use the **fetch()** method to call the API https://fakestoreapi.com/products and handle the response with promises in JavaScript:
- **Program: Fetch Data from an API using fetch() and Promises**
- javascript

```
// Fetch data from the API
fetch('https://fakestoreapi.com/products')
  .then((response) => {
    // Check if the response is successful
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    // Parse the JSON data from the response
    return response.json();
  })
  .then((data) => {
    // Handle the fetched data here
    console.log('Fetched Products:', data);
  })
  .catch((error) => {
    // Handle any errors that occurred during the fetch operation
    console.error('There was an error with the fetch operation:', error);
  });
```

- **Explanation:**
- **fetch()**: This method is used to make a request to the given URL (https://fakestoreapi.com/products). By default, it performs a GET request.
- **.then(response => response.json())**: The first .then() block checks if the response is successful using the response.ok property. If the response is successful, the data is parsed from JSON format using response.json().
- **.then(data => { ... })**: The second .then() block receives the parsed JSON data, and in this case, it logs the data to the console (which is an array of products).
- **.catch(error => { ... })**: If any error occurs during the fetch request (e.g., network errors, invalid URL, or failure to parse the response), the .catch() block is triggered, and the error is logged.
- **Output:**
- The output will be an array of product objects fetched from the API and logged to the console. Each product object contains details like id, title, price, description, etc.
- **Example of Output:**

- json
- [
- {
-   "id": 1,
-   "title": "Product 1",
-   "price": 29.99,
-   "description": "Description of product 1",
-   "category": "electronics",
-   "image": "https://example.com/image1.jpg"
- },
- {
-   "id": 2,
-   "title": "Product 2",
-   "price": 19.99,
-   "description": "Description of product 2",
-   "category": "electronics",
-   "image": "https://example.com/image2.jpg"
- },
- ...
-]
- This way, you can fetch data from the API and handle the promise with .then() and .catch() to deal with success and failure cases.
- 4o mini

5. Display all the product from the api in your HTML page

- To display all the products from the API (https://fakestoreapi.com/products) on an HTML page, you can use the fetch() method to retrieve the data and then dynamically generate HTML to display the products. Below is a complete example:
- **HTML and JavaScript to Fetch and Display Products**
- html
- <!DOCTYPE html>
- <html lang="en">
- <head>
- <meta charset="UTF-8">
- <meta name="viewport" content="width=device-width, initial-scale=1.0">
- <title>Product List</title>
- <style>
-   body {
-    font-family: Arial, sans-serif;
-    display: flex;
-    justify-content: center;
-    align-items: center;

```
•       flex-direction: column;
•       padding: 20px;
•     }
•     .product-container {
•       display: flex;
•       flex-wrap: wrap;
•       gap: 20px;
•       justify-content: center;
•     }
•     .product {
•       border: 1px solid #ddd;
•       padding: 10px;
•       width: 200px;
•       text-align: center;
•       box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
•       border-radius: 5px;
•     }
•     .product img {
•       width: 100%;
•       height: auto;
•       border-radius: 5px;
•     }
•     .product h3 {
•       font-size: 18px;
•       margin: 10px 0;
•     }
•     .product p {
•       font-size: 14px;
•       color: #555;
•     }
•     .product .price {
•       font-size: 16px;
•       color: green;
•       font-weight: bold;
•     }
•   </style>
• </head>
• <body>
•
•   <h1>Product List</h1>
•   <div class="product-container" id="product-container">
•     <!-- Products will be displayed here -->
```

```
  </div>

  <script>
    // Fetch data from the API
    fetch('https://fakestoreapi.com/products')
      .then((response) => {
        if (!response.ok) {
          throw new Error('Network response was not ok');
        }
        return response.json();  // Parse the response as JSON
      })
      .then((products) => {
        // Get the container where products will be displayed
        const productContainer = document.getElementById('product-container');

        // Loop through the products and create HTML elements to display them
        products.forEach((product) => {
          const productElement = document.createElement('div');
          productElement.classList.add('product');

          // Set the HTML content of the product element
          productElement.innerHTML = `
            <img src="${product.image}" alt="${product.title}">
            <h3>${product.title}</h3>
            <p>${product.description}</p>
            <p class="price">$${product.price}</p>
          `;

          // Append the product element to the container
          productContainer.appendChild(productElement);
        });
      })
      .catch((error) => {
        console.error('There was an error with the fetch operation:', error);
      });
  </script>

</body>
</html>
```

- **Explanation:**
- **HTML Structure**:

- The body contains an <h1> for the page title and a div with the ID product-container where all the products will be displayed.
- The <style> tag contains basic CSS to make the page look clean, with the products arranged in a responsive grid.
- **JavaScript**:
- The fetch() method is used to get the product data from the API (https://fakestoreapi.com/products).
- Once the data is fetched successfully, we parse the response as JSON using .json().
- The products array is looped through, and for each product, we create a new div element with the class product containing the product's image, title, description, and price.
- These div elements are then appended to the product-container div on the page.
- **CSS Styling:**
- The product-container uses Flexbox to arrange the product cards in a responsive manner.
- Each product card (.product) has some basic styling, including borders, padding, and shadow effects.
- Images are displayed with a fixed width, and product descriptions and prices are styled.
- **Output:**
- This code will display a grid of product cards on the web page, each containing:
- An image of the product.
- The product's title.
- The product's description.
- The product's price.
- When you load the HTML page in the browser, it will fetch data from the API, display each product, and handle any errors in case the fetch fails.
- 4o mini
-