SUM.JS

```
// I can work with many css and html and js files in the same time :

// error handling :

// try to do something :
try {
  // code
} catch (argument) {
  // failed try :
  //code
} finally {
  //always executed :
  //code
}

//api ____________________________________________________:
// create an object from XMLHttpRequest class :
//  let obj= new className()
let request=new XMLHttpRequest();

//  open function :  to  Prepare a request.: obj.open("type","url");
request.open("get","https://jsonplaceholder.typicode.com/posts");
// specify  the type of  response :
 request.responseType="json";
// send the request :
request.send();

// I need to await the response from the server :
// so I need to check if the response arrived  I'll get it and  display it on the console:

// run a function when the  stat of request change
request.onreadystatechange=function(){
console.log("done");
document.querySelector("h2").innerHTML+=" change stat<br>";
};

// run function when the response arrived :

request.onload=function(){

// get response and push it to dom (h1)
// for string we use request.responseText :
// for json we use request.response
let ArrResponse=this.response
 // the default  type of response is a string :
  // to convert the you need to use request.responseType="type";
 console.log( typeof this.response);

 // now the response is an array :
 // we can handel with  easily  and access to data from anywhere :
 console.log(ArrResponse[0].userId);

 // print all title in the dom :
```

```

 for(post of ArrResponse){
document.querySelector('h1').innerHTML+=`${post.title} <br><br>`;

 }



}


//example :
```
html code
<h1 style="margin-bottom: 60px;" > json placeholder</h1>

    <div style="padding: 10px;
    box-shadow: 1px 1px 10px rgba(0, 0, 0, 0.609); width: 60%;">
        <h2>wait the response ...</h2>
        <progress max="100" value="0">

    <pre></pre>
```
// js
const progress = document.querySelector("progress");
const WaitTitle = document.querySelector("h2");
const responseDom = document.querySelector("center>pre");

// create a object from XmlHttpRequest  class :
request = new XMLHttpRequest();

// prepare the request :
request.open("get", "https://jsonplaceholder.typicode.com/posts/1");
request.responseType = "json";

// send the request :
request.send();

// check stats :

request.onreadystatechange = function () {
  let computer = 0;
  let EditProgress = setInterval(function () {
    progress.value = ++computer;
    if (computer == 50) clearInterval(EditProgress);
  }, 50);
};
// get  json when the response arrived :
request.onload = function () {
  setTimeout(() => {
    for (let i = 50; i <= 110; i += 20) {
      let computer = 50;
      setTimeout(() => {
        progress.value = i;
```

```
      }, 200);
    }

    setTimeout(() => {
      document.body.style.background = "black";
      document.body.style.color = "white";
      WaitTitle.style.color = "red";
      WaitTitle.innerText = "the response arrived";

      let Response = request.response;
      console.log(Response);
      responseDom.innerHTML = `

 userId : ${Response.userId} <br>

 id : ${Response.id} <br>

 title :${Response.title} <br>

`;
    }, 200);

    //userId: 1, id: 1, title
  }, 2500);
};


/*
In JavaScript,you can work with network requests using the `XMLHttpRequest` object or the more
 modern `fetch` API
. When dealing with network requests, you typically encounter various states in the lifecycle
of the request. Here's
a detailed explanation of the states for both `XMLHttpRequest` and `fetch`:

**Using XMLHttpRequest:**

1. **Uninitialized (State 0):**
   - This is the initial state when the `XMLHttpRequest` object is created but not
    yet initialized.
   - You can set the request method, URL, and other properties in this state.

2. **Opened (State 1):**
   - After calling the `open` method on the `XMLHttpRequest` object,
   it enters this state.
   - You can set request headers in this state.

3. **Headers Received (State 2):**
   - When the `send` method is called, the request is sent to the server,
    and the `XMLHttpRequest` object enters this state.
   - At this point, you can access response headers using the `getResponseHeader` method.

4. **Loading (State 3):**
   - In this state, the `XMLHttpRequest` object is actively downloading the response
    data from the server.
   - You can monitor the progress of the download using event listeners like `onprogress`.
```

```
5. **Done (State 4):**
   - The request is complete, and the `XMLHttpRequest` object enters this state.
   - You can access the response data and check the `status` property to determine
   the HTTP status code.
   - Handle the response and perform any necessary actions based on the
   server's response.

Here's an example of how you can use `XMLHttpRequest` to make a network request
 and monitor its state changes:
*/
javascript
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data', true);

xhr.onreadystatechange = function () {
  if (xhr.readyState === 4) { // State 4: Done
    if (xhr.status === 200) { // HTTP status code 200 (OK)
      // Process the response data
      console.log(xhr.responseText);
    } else {
      // Handle errors or non-200 status codes
      console.error('Request failed with status:', xhr.status);
    }
  }
};

xhr.send(); // Initiate the request

// add headers params :
xhr.setRequestHeader("HeaderName1", "HeaderValue1");
xhr.setRequestHeader("HeaderName2", "HeaderValue2");

// send data :

// Set the Content-Type header if you're sending JSON data
/*
Yes, setting the Content-Type header to "application/json" using xhr.setRequestHeader("
Content-Type",
"application/json"); is a common practice when you're sending JSON data in the body of an
XMLHttpRequest.
 This header informs the server that the request body contains JSON-formatted data.
*/
xhr.setRequestHeader("Content-Type", "application/json");

// Create an object with the data you want to send
var data = {
    key1: "value1",
    key2: "value2"
};

// Convert the object to a JSON string
var jsonData = JSON.stringify(data);

// Send the JSON data in the body of the request
xhr.send(jsonData);

```

```
/*
**Using the Fetch API:**

The Fetch API provides a more modern and promise-based way to
work with network requests, making it easier to handle various states.
However, it doesn't expose explicit state changes like `XMLHttpRequest`.
 Instead, it returns a promise that resolves when the request is complete.

Here's an example of how you can use the Fetch API to make a network request:
*/
//javascript
fetch('https://api.example.com/data')
  .then((response) => {
    if (!response.ok) {
      throw new Error(`Request failed with status: ${response.status}`);
    }
    return response.json(); // Parse the response data as JSON
  })
  .then((data) => {
    // Process the response data
    console.log(data);
  })
  .catch((error) => {
    // Handle errors
    console.error(error);
  });


//In the Fetch API example, you handle the request completion and errors through
//promises, making it a more concise and modern approach to
//working with network requests in JavaScript.


// class : (OOP)

//1- declaration :
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }

  sayHello() {
    console.log(`Hello, my name is ${this.name}`);
  }
}


// 2- Constructors:
/*

 Classes have a special method called constructor that is automatically
 called when an object of the class is created. It's used for initializing object
 properties.

 */
```

```
276
277 // 3-  Creating Instances:
278 const person1 = new Person('Alice', 30);
279 const person2 = new Person('Bob', 25);
280
281 //4. Inheritance:
282
283 /*
284 Classes in JavaScript support inheritance using the extends keyword.
285 You can create a subclass that inherits properties and methods from a parent class.
286 */
287 class Student extends Person {
288   constructor(name, age, studentId) {
289     super(name, age); // Call the parent class constructor
290     this.studentId = studentId;
291   }
292
293   study() {
294     console.log(`${this.name} is studying.`);
295   }
296 }
297
298 //5. Super Keyword:-----------------------
299 /*
300 In JavaScript classes, the super keyword is used to call methods or constructors of a
301 parent class (also known as the superclass) within a subclass. It allows you to access
302 and invoke functions or constructors defined in the parent class from the child class.
303  Here's how super is commonly used:
304
305 */
306   //a= Calling the Parent Constructor::
307         class Parent {
308           constructor(name) {
309             this.name = name;
310           }
311         }
312
313         class Child extends Parent {
314           constructor(name, age) {
315             super(name); // Call the constructor of the parent class
316             this.age = age;
317           }
318         }
319
320         const child = new Child('Alice', 25);
321         console.log(child.name); // Accessing property from the parent class
322         console.log(child.age);  // Property specific to the child class
323
324
325 // b-Calling Parent Methods:
326
327       class Parent {
328         sayHello() {
329           console.log('Hello from Parent');
330         }
331       }
```

```
      class Child extends Parent {
        sayHello() {
          super.sayHello(); // Call the method of the parent class
          console.log('Hello from Child');
        }
      }

      const child1 = new Child();
      child1.sayHello();

  //c-Accessing Parent Properties:
      class Parent {
        constructor(name) {
      this.name = name;
        }
      }

      class Child extends Parent {
        constructor(name, hobby) {
          super(name); // Call the constructor of the parent class
          this.hobby = hobby;
        }

        getDetails() {
          return `${super.name} enjoys ${this.hobby}`;
        }
      }

      const child2 = new Child('Alice', 'painting');
      console.log(child2.getDetails());

//6- Static Methods:
class MathUtils {
  static square(x) {
    return x * x;
  }
}

const result = MathUtils.square(5); // Calling a static method

// 7 -Getters and Setters in JavaScript Classes:
/*
Getters and setters are special methods in JavaScript classes that allow you to control
 access to the properties of objects. They are used to get and set the values of object
 properties while providing an interface for controlling how those values are retrieved
 and modified.

*/
class Circle {
  constructor(radius) {
    this._radius = radius;
  }

  get area() {
    return Math.PI * this._radius * this._radius;
```

```
  }

  set radius(newRadius) {
    if (newRadius >= 0) {
      this._radius = newRadius;
    } else {
      console.error('Radius cannot be negative.');
    }
  }
}


// set response type json in postman

// headers : =>"Accept:"application/json"

// set headers params:
request.setRequestHeader('key','value');

// set response type :
request.setRequestHeader("Accept","application/json");
// set request type :
request.setRequestHeader("Content-Type","application/json");

// send request in  post mode

let bodyParams={
"key1":"value1",
"key2":"value2",
"key3":"value3"

}
// if you send json you will get  stat=500
request.send(bodyParams);

// you need to convert it to string :
 request.send(JSON.stringify(bodyParams));

 // return the status of the response :
 request.status
 // 404 url error :
 // 500 > server error :
 // 200<request.status <300 good response

// put type : updated all  information :
// patch : updated specific  information :


// get with filtering :
request.open("GET","https://jsonplaceholder.typicode.com/posts/?userId=1");

//✨ example : ----------------------------------[-]
function getPosts() {
  let request = new XMLHttpRequest();

  request.open("GET", "https://jsonplaceholder.typicode.com/posts");
```

```
  request.responseType = "json";
  request.send();

  request.onload = function () {
    if (request.status < 200 || request.status > 300) alert("server error");
    else {
      let posts = request.response;

      for (element of posts) {
        document.getElementById(
          "content"
        ).innerHTML += `<h2>${element.title}</h2>`;
      }
    }
  };
}

//getPosts();

function createNewPost() {
  let request = new XMLHttpRequest();
  request.open("POST", "https://jsonplaceholder.typicode.com/posts");
  request.setRequestHeader("Accept", "application/json");
  request.setRequestHeader("Content-Type", "application/json");
  request.responseType = "json";
  let bodyParams = {
    title: "my task",
    body: "go sleep",
    userId: 1,
  };

  // send the request :
  request.send(JSON.stringify(bodyParams));

  request.onload = function () {
    if (request.status < 200 || request.status > 300) alert("server error");
    else {
      let post = this.response;
      console.log(post);
      alert("the post has been created successfully ");
    }
  };
}

//createNewPost();

function updatePost() {
  let request = new XMLHttpRequest();
  request.open("PUT", "https://jsonplaceholder.typicode.com/posts/1");
  request.setRequestHeader("Accept", "application/json");
  request.setRequestHeader("Content-Type", "application/json");
  request.responseType = "json";
  let bodyParams = {
    title: "hello world",
    body: "bar",
    userId: 1,
```

```
  };

  // send the request :
  request.send(JSON.stringify(bodyParams));

  request.onload = function () {
    if (request.status < 200 || request.status > 300) alert("server error");
    else {
      let post = this.response;
      console.log(post);
      alert("the post has been updated successfully ");
    }
  };
}
//updatePost();

function deletePost() {
  let request = new XMLHttpRequest();
  request.open("DELETE", "https://jsonplaceholder.typicode.com/posts/1");
  request.setRequestHeader("Accept", "application/json");

  // send the request :
  request.send();

  request.onload = function () {
    if (request.status < 200 || request.status > 300) alert("server error");
    else alert("the post has been deleted successfully ");
  };
}

//deletePost();

function getPostsWithFiltering() {
  let request = new XMLHttpRequest();

  request.open("GET", "https://jsonplaceholder.typicode.com/posts/?userId=1");
  request.responseType = "json";
  request.send();

  request.onload = function () {
    if (request.status < 200 || request.status > 300) alert("server error");
    else {
      let posts = request.response;

      for (element of posts) {
        document.getElementById(
          "content"
        ).innerHTML += `<h2>${element.title}</h2>`;
      }
    }
  };
}

getPostsWithFiltering();


```

```
556
557 // PROMISES :
558
559 /*
560 one of   the raison of foundation of promises it's the problem
561 of callback hell
562
563 anything return promise it will  be able to you use then after it
564
565
566 the code in then will call when the resolve in promise will  call
567
568 */
569
570 // promise it 's a class you need to create a new instant to using it :
571 new Promise((resolve, reject) => {
572
573 })
574
575 //  the  code written in the promise is an  asynchronism code :
576 new Promise((resolve, reject) => {
577   // async code :
578
579   if(isSecureContext){
580       resolve();
581   }else{
582
583       reject();
584   }
585
586
587 })
588
589
590
591 // create new promise :
592
593 //  1-part : create  the async code :
594 let p = new Promise((resolve, reject) => {
595     // async code :
596
597     if (isSecureContext) {
598     // if  condition valid the resolve function will be called :
599         resolve();
600     } else {
601     // else the reject function will be called :
602         reject();
603     }
604 });
605
606 // part 2 : handel with response  :
607
608 // after this if the resolve function called  then the then function will call
609 p.then(function () {
610     console.log("calling then");
611 });
```

```
612
613 // else (the promise rejected ) the catch function will be calling :
614 p.catch(function () {
615     console.log("error ");
616 });
617
618
619 //EXAMPLE :
620 let h3 = document.querySelector("h3");
621
622 let p1 = new Promise((resolve, reject) => {
623     let flag = true;
624
625     if (flag) {
626         setTimeout(() => {
627             h3.style.visibility = "visible";
628         }, 1000);
629
630         resolve("the visibility of h3 changed with successfully : ");
631     } else {
632         reject("error");
633     }
634 });
635
636 p.then((successMsg) => {
637     console.log(successMsg);
638 });
639 p.catch((ErrorMsg) => {
640     console.error(ErrorMsg);
641 });
642
643 /*
644
645 Hello world is printed first because the JavaScript engine executes code synchronously,
646  line by line. When the engine reaches the `console.log("hello world");` line,
647  it executes it immediately and printsm "hello world" to the console.
648
649 The `Promise` object is asynchronous, meaning that the code inside of the `then()`
650 and `catch()`
651 callbacks will not be executed until the promise is fulfilled or rejected.
652  In this case, the promise , is fulfilled immediately, but the engine does not execute
653   the `then()` callback until after it has finished executing
654    the rest of the code in the script.
655
656 To see this in action, you can add a `setTimeout()` function to the `then()` callback:
657 */
658 ```javascript
659 let p = new Promise((resolve, reject) => {
660
661         resolve("the visibility of h3 changed with successfully : ");
662 });
663
664 p.then((successMsg) => {
665     setTimeout(() => {
666         console.log(successMsg);
667     }, 1000);
```

```
});
p.catch((ErrorMsg) => {
    console.error(ErrorMsg);
});

console.log("hello world");
```
/*

Now, when you run the script, you will see "hello world" printed to the
 immediately, followed by
 the success message from the promise one second later.

This is the behavior of the JavaScript engine because it is designed to be efficient.
By executing code synchronously, the engine can avoid the overhead of switching back
and forth between different contexts. However, this can also lead to some unexpected
behavior,
as in this case.

If you need to ensure that code is executed after a promise is fulfilled or rejected,
you can use the `await` keyword. The `await` keyword will cause the engine to pause
execution
 of the current function until the promise is fulfilled or rejected. This can be useful for
 ensuring that code is executed in a specific order, or for handling errors.
*/

// example : Promises Chain :
let Headers = document.querySelectorAll("h3");

new Promise((resolve, reject) => {
    setTimeout(() => {
        console.log("first header ");

        Headers[0].style.color = "red";
        resolve("second header :");
    }, 1000);
})
    .then((UserMsg) => {
        console.log(UserMsg);
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                Headers[1].style.color = "red";
                resolve("third header :");
            }, 1000);
        });
    })
    .then((UserMsg) => {
        console.log(UserMsg);
        return new Promise((resolve, reject) => {
            setTimeout(() => {
                Headers[2].style.color = "red";
            }, 1000);
        });
    });
```
```

```
  // fetch function :
  ```
  fetch is a JavaScript function that allows you to make network requests
   (typically HTTP requests) to
  fetch resources from a network, such as JSON data from a REST API, HTML
  from a website, or other types
  of data. It's widely used in modern web development for making asynchronous
  requests to web servers.
  ```
  fetch(url, options)
/*
✨url: The URL of the resource you want to fetch.
options (optional): An object containing various options for the request,
including the HTTP method, headers, request body, and more.
✨Creating a Request:
When you call fetch, it creates and returns a Promise that represents the future response to the request.
 However, the request is not sent immediately; it's only prepared at this stage.

✨Configuring the Request:
You can specify various options in the options object to configure the request:

     ✨method: The HTTP method (e.g., 'GET', 'POST', 'PUT', 'DELETE') to use for the request.
        headers: An object containing the HTTP headers for the request, such as 'Content-Type' and
         'Authorization'.

     ✨body: The request body, typically used for sending data in POST or PUT requests.
        It should be a string or a FormData object.

     ✨mode: The request mode (e.g., 'cors', 'no-cors', 'same-origin') that defines how cross-origin
       requests are handled.

     ✨credentials: Indicates whether to include cookies or credentials with the request
       ('same-origin', 'include', 'omit').

     ✨cache: The caching mode for the request ('default', 'no-store', 'reload', etc.).

     ✨redirect: How to handle redirects ('follow', 'error', 'manual').
     And more.
*/


//✨Sending the Request:
/*
To actually send the request, you need to call .then() or use async/await on the returned Promise.
This initiates the network request to the specified URL with the provided options.
*/
fetch(url, options)
  .then(response => {
    // Handle the response here
  })
  .catch(error => {
    // Handle errors here
```

```
  });

//✨Handling the Response:
/*
Once the request is sent, the fetch function returns a Promise that resolves
with a Response object representing the response from the server.
 You can then use methods and properties of this Response object to handle the response
data.

Common methods and properties of the Response object include:

   .json(): Parses the response body as JSON.
   .text(): Reads the response body as text.
   .blob(): Returns the response body as a binary Blob.
   .headers: Access to the response headers.
   .status: HTTP status code (e.g., 200 for OK, 404 for Not Found).
   .statusText: HTTP status message (e.g., "OK", "Not Found").
*/


//✨Handling Errors:
/*
If the network request fails or encounters an error (e.g., due to a network issue,
invalid URL, or server error), the Promise is rejected, and you can catch the error using
.catch().

Here's an example of using fetch to make a GET request and handle the response:

*/
fetch('https://jsonplaceholder.typicode.com/posts/1')
  .then(response => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json(); // Parse response body as JSON
  })
  .then(data => {
    console.log(data); // Process the JSON data
  })
  .catch(error => {
    console.error('Fetch error:', error);
  });


// The `fetch` API is structured around a set of classes and objects that allow you to
interact with
// network requests and responses. Understanding the structure of these classes can help you
work
// with the API more effectively. Here are some of the key classes and objects in the
`fetch` API:


"1. **`fetch` Function**:"
/*

The `fetch` function is the entry point to making network requests. It returns a
    `Promise` that resolves to a `Response` object representing the response to the request.
```

```
*/
"2. **`Request` Class**:"
  //  The `Request` class represents a network request that you can create and configure before
  //  passing it to the `fetch` function. It has a constructor that takes a URL and an optional
  //  options object to configure the request.

   ```javascript
   const request = new Request(url, options);
   ```

"3. **`Response` Class**:"
  // The `Response` class represents the response to a network request. It provides
  //   methods and properties to access various aspects of the response, including
  //   the response body, headers, status code, and more.

   fetch(url)
     .then(response => {
       // response is an instance of the Response class
       // you can use methods like response.json(), response.text(), etc.
     });

  // Common methods and properties of the `Response` class include:
  // - `.json()`: Parses the response body as JSON.
  // - `.text()`: Reads the response body as text.
  // - `.blob()`: Returns the response body as a binary Blob.
  // - `.headers`: Access to the response headers.
  // - `.status`: HTTP status code (e.g., 200 for OK, 404 for Not Found).
  // - `.statusText`: HTTP status message (e.g., "OK", "Not Found").

"4. ** Headers` Class**:"
  // The `Headers` class represents a collection of HTTP headers associated with a
  // request or response. You can use it to manipulate headers before sending a request
  //   or after receiving a response.

   const headers = new Headers();
   headers.append('Content-Type', 'application/json');
   headers.set('Authorization', 'Bearer Token');

"5. **`FormData` Class**:"
  // The `FormData` class allows you to create and manipulate form data that can be sent in a
  // network request. You can use it to build and send form data in a POST request.

   const formData = new FormData();
   formData.append('username', 'john_doe');
   formData.append('password', 'secure_password');

"6. **`URL` and `URLSearchParams` Classes**: "
  // The `URL` class represents a URL, and the `URLSearchParams` class is used for working
  // with URL query parameters. You can use these classes to parse and manipulate URLs.
```

```
881
882
883     const url = new URL('https://example.com/api');
884     url.searchParams.append('param1', 'value1');
885     url.searchParams.append('param2', 'value2');
886
887 //example of fetching data from placeholder API :
888 let posts = "https://jsonplaceholder.typicode.com/posts?userId=";
889 let Users = "https://jsonplaceholder.typicode.com/users";
890
891 function GetUsers() {
892     return new Promise((resolve, reject) => {
893         fetch(Users)
894             .then((response) => {
895                 if (!response.ok) throw new Error("there has been an Error in Fetching
    UsersData from the Sever ");
896
897                 return response.json();
898             })
899             .then((Users) => {
900                 console.log("Users ", Users);
901                 resolve();
902             })
903             .catch((ErrorMsg) => {
904                 reject(ErrorMsg);
905             });
906     });
907 }
908
909 function GetPosts(UserId) {
910     return new Promise((resolve, reject) => {
911         fetch(posts+UserId)
912             .then((response) => {
913                 if (!response.ok) throw new Error("there has been an Error in Fetching
    PostsData from the Sever ");
914
915                 return response.json();
916             })
917             .then((Posts) => {
918                 console.log("Posts : ", Posts);
919                 resolve();
920             })
921             .catch((ErrorMsg) => {
922                 reject(ErrorMsg);
923             });
924     });
925 }
926
927 // Define the URL for the API endpoint
928 const Url = 'https://api.example.com/endpoint';
929
930 // Create an object with the data you want to send in the request body
931 const data = {
932   param1: 'value1',
933   param2: 'value2'
934 };
935
```

```
// Create the request options, including method, headers, and body
const requestOptions = {
  method: 'POST', // or 'GET', 'PUT', 'DELETE', etc.
  headers: {
    'Content-Type': 'application/json' // specify the content type if sending JSON data
    // Add any other headers if needed
  },
  body: JSON.stringify(data) // Convert the data object to a JSON string
};

// Use the fetch function to make the request
fetch(Url, requestOptions)
  .then(response => {
    // Check if the request was successful (status code 200-299)
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    // Parse the response JSON
    return response.json();
  })
  .then(data => {
    // Do something with the data returned from the API
    console.log(data);
  })
  .catch(error => {
    // Handle errors
    console.error('Fetch error:', error);
  });


//axios in js :  get Users Using axios library :
const axios = require("axios"); // Import Axios in a Node.js environment

// Example: Making a POST request with request body parameters
axios.post("https://example.com/api/resource", {
  key1: "value1",
  key2: "value2",
}, {
  headers: {
    // Define your custom headers here
    "Content-Type": "application/json", // Set the appropriate content  type
    Authorization: "Bearer your-access-token", // Optional: Include an authorization header
  },
})
  .then((response) => {
    console.log(response.data); // Process the data from the response
  })
  .catch((error) => {
    console.error("Axios error:", error);
  });


// auto catch of errors :
function getUsersAxios() {
    return new Promise((resolve, reject) => {
      axios
```

```
        .get(Users)
        .then((response) => {
          return response.data;
        })
        .then((Users) => {
          console.log(Users);
          resolve();
        })
        .catch((error) => {
          reject(error);
        });
    });
  }

  function getPostsAxios(UserId) {
    let PostUrl = posts + UserId;

        axios
          .get(PostUrl)
          .then((response) => {
            return response.data;
          })
          .then((Posts) => {
            console.log(Posts);

          })
          .catch((error) => {
            reject(error);
          });

  }

  getUsersAxios()
    .then(() => {
      return getPostsAxios(1);
    })
    .catch((error) => {
      console.log("Error From :", error);
    })



// npm  node package manager :

//✨download node js :  node -v to know the version

//✨initialize the project => npm init

//✨ install library : npm install libraryName --save(save in in package.json)
//✨package.json : information about library that you have installed :
//✨node modules : contain the code of  all library
//✨lock.json specify version  of  libraries :

// ✨important information :
/*
  if you use just  write :  npn install
```

```
  the npm will take information library
  in the package file then it will install it and add  to your
  project
  very helpful when you work in a team and you want to install the library
  of  your team to work with,just  you  need to get the package file
  then write the npm install command then the npm will install
  all library  in the package.json with the same specification :

*/

//✨last step  import axios from node modules  ✔
/**
 *
 you use the require just when you  working with frameworks :(Angular,React...)

 due to we work just with pure js we need to import the axios.js
 manually
 like this :
 <script src="../node_modules/axios/axios.js"></script>
 or :
 <script src="../node_modules/axios/dist/axios.min.js"></script>
*/


// await and async :

// simple way to get Users and Posts using simple fetch
// without apply await ans async keyWord

/*
but firstly let's introduce the two fundamentals :

await: to keep the js await until an async code finished
it's veery useful when you handel with api settimeout ..
and many foundations in js :
// important notion about it :

be carefully because  you can use await only in an async function :



let's   freaking out the second concept :

async : this keyword using the define an async function :
and it's provide as to use the async keyword into  functions
and make the function automatically return   Promise
and the return keyword Represent the Resolve() function in the promise

async functionName(){
// async code

return  Anything   // resolve(Anything)
}

*/
```

```
// version 1:
/*
in this version bellow  : the code is writing
just with fetch functions without using  await and async
keyword

 how can see that's the implementation of it

 it's a little bit difficult to Read an maintain :

 specially  when you work on multiple then in the same promise :


and for that the await an async founded to solve this problem exactly

features of await and async  :

1-more readability of code :
2- easy the maintain
*/
let usersUrl = "https://jsonplaceholder.typicode.com/users";
let postsUrl = "https://jsonplaceholder.typicode.com/posts/?userId=";

/*
now let's develop  this code :
and make it more useful
by  Provide  to getUsers first then get  Posts after it :
following the order :

*/
function getUsers() {
    // fetch the data from api placeholder api :
    return new Promise((resolve, reject) => {
        fetch(usersUrl)
            .then((response) => {
                // check status if ok or not :

                if (!response.ok) throw new Error(response.statusText);

                return response.json();
            })
            .then((Users) => {
                console.log("All Users : ", Users);
                resolve("Success to get User Response : ");
            })
            .catch((error) => {
                console.log(error);
                reject("failed to get User Response there has been an error");
            });
    });
}

function getPosts(UserId) {
    // fetching data from basic PostsUrl +UserId;
    return new Promise((resolve, reject) => {
        fetch(postsUrl + UserId)
```

```
            .then((response) => {
                // check status if ok or not :

                if (!response.ok) throw new Error(response.statusText);

                return response.json();
            })
            .then((posts) => {
                console.log(`Posts  Related to User[${UserId}]`, posts);
                resolve("Success to get the  API response  ");
            })
            .catch((error) => {
                console.log(error);
                reject("there has been an error during  fetching the API Response");
            });
    });
}

getUsers()
    .then(() => getPosts(1))
    .catch((error) => {
        console.log("Error :", error);
    });

 // In the provided code:


  async function fetchData() {
    try {
      let response = await fetch('https://api.example.com/data');
      let data = await response.json();
      return data;
    } catch (error) {
      console.error('Error fetching data:', error);
      throw error;
    }
  }


// version 2 with await  and async functions  :
async function getUsers() {
    let response = await fetch('https://jsonplaceholder.typicode.com/users');

    if (!response.ok)  return response.statusText;
    let Users = await response.json();

    console.log('Users',Users);
    return;
}

let PostUrl = "https://jsonplaceholder.typicode.com/posts/?userId=";

async function getPosts(UserId) {
    let response = await fetch(PostUrl + UserId);

    if (!response.ok)  return response.statusText;
```

```
    let Posts = await response.json();

    console.log(`Posts[${UserId}]`, Posts);
    return;
}

async function getData(){

  await  getUsers();
    getPosts(1);

}

getData();


  /*

  The `throw error;` statement inside the `catch` block is throwing the error again
  after it has been logged. When an error is
   thrown within a `catch` block, it propagates the error up the call stack. In the
   context of an `async` function like `fetchData()`,
    if you call `fetchData()`, and an error occurs during the execution of `fetchData()`,
    the function will reject with the thrown error.

  Here's how it works:

  1. The `fetch` API is used to make an HTTP request to 'https://api.example.com/data'.
  2. If the request fails (for example, due to network issues or an invalid URL),
   `fetch` will reject with an error.

  3. The `await fetch(...)` expression inside the `try` block will throw an error.
  4. The code inside the `catch` block will execute, logging the
   error to the console using `console.error('Error fetching data:', error);`.

  5. After logging the error, `throw error;`
  re-throws the error, causing the `fetchData()` function to reject with this error.

  When you call `fetchData()`, you can handle the rejection
  by using `.catch()` or `try/catch` blocks in the calling code. For example:

  */
  ///javascript
  fetchData()
    .then(data => {
      // Handle successful data retrieval
      console.log('Data:', data);
    })
    .catch(error => {
      // Handle the error from fetchData() here
      console.error('Error in fetchData():', error);
    });


  /*
  In this case, if there's an error during the execution of `fetchData()`,
```

```
  it will be caught in the `.catch()` block, where you can handle
  it appropriately.
  */

  // example 2:
  let titles = document.querySelectorAll("h1");

function changeVisibility(index) {
    return new Promise((resolve) => {
        setTimeout(() => {
            titles[index].style.visibility = "visible";
            resolve(index);
        }, 1000);
    });
}

async function ChangeTitlesVisibility() {
    for (let i = 0; i < titles.length; i++) {
        await changeVisibility(i);

    }
}

ChangeTitlesVisibility();

//Authentication :

//https://reqres.in/

//type of token :

// Bearer token : (headers)  Authorization =Bearer token
// normal token : (headers)  Authorization = token


//login :
let loginUrl = "https://reqres.in/api/login";
let UserUrl = "https://reqres.in/api/users";

let loginToken= localStorage.getItem("userToken") || "";
function login() {
    axios
        .post(loginUrl, {
            email: "tracey.ramos@reqres.in",
            password: "cityslicka",
        })
        .then((response) => response.data)

        .then((Token) => {
            console.log(Token);
            loginToken = Token.token;
            localStorage.setItem("userToken",loginToken);
            createNewUser();
        })
        .catch((error) => {
            alert(error);
```

```
        });
}


function createNewUser() {
    let config = {
        headers: {
            "Authorization": "Bearer " + loginToken,
        },
    };

    axios
        .post(
            UserUrl,
            {
                name: "majid",
                job: "leader",
            },
            config
        )
        .then((response) => response.data)
        .then((newUserInfo) => {
            console.log(newUserInfo);
        })
        .catch((error) => {
            console.log(error);
        });
}


// using await and async  function :

// let loginUrl = "https://reqres.in/api/login";
// let UserUrl = "https://reqres.in/api/users";
let registerUrl = "https://reqres.in/api/register";

// let loginToken = localStorage.getItem("userToken") || "";

let bodyPrams = {
    email: "tracey.ramos@reqres.in",
    password: "cityslickda",
};

async function login() {
    try {
        let response = await axios.post(loginUrl, bodyPrams);

        let token = response.data;
        console.log(token);
        localStorage.setItem("userToken", token.token);
    } catch (error) {
        console.log("Error : ", error.message);
    }
}

// let headers = {
```

```
//  Authorization: "Bearer " + loginToken,
// };

let UserINfo = {
    name: "majid",
    job: "leader",
};
async function createNewUser() {
    try {
        let response = await axios.post(UserUrl, UserINfo, headers);

        let newUserInfo = response.data;
        console.log(newUserInfo);
    } catch (error) {
        console.log("Error : ", error.message);
    }
}

async function register() {
    try {
        let response = await axios.post(registerUrl, bodyPrams);

        let registerInfo = response.data;
        console.log(registerInfo);
        console.log("token : ", registerInfo.token);
        localStorage.setItem("userToken", registerInfo.token);
    } catch (error) {
        console.log("Error : ", error.message);
    }
}

async function main() {
    console.log("\nRegister new user : ");
    await register();

    console.log("\nlogin to my created account  : ");
    await login();

    console.log("\ncreate a new user using my token : ");
    await createNewUser();
}

main();


 login();

// delay function :
function delay(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}

 // CHANGE PAGE :
 window.location.assign("NewPageName.html");

/*
```

by using the login function we send request to api then api
generate a token that we received then save in local Storage

to provide to user enter directly to his account

without the need to login again by using CreateUser function :
that take Token from local Storage then created a new user

*/
/*
To navigate to another page using JavaScript, you can use the `window.location` object.
 Here are a few common ways to achieve this:

1. Using `window.location.href`:
You can set the `window.location.href` property to the URL of the page you want to navigate to. For example:

```javascript
// Navigate to a new page
window.location.href = "https://www.example.com/newpage.html";
```

2. Using `window.location.assign()` method:
The `assign()` method of the `window.location` object is another way to navigate to a new page:

```javascript
// Navigate to a new page
window.location.assign("https://www.example.com/newpage.html");
```

3. Using `window.location.replace()` method:
The `replace()` method of the `window.location` object can be used
to navigate to a new page and replace the current
 page in the browser history. This means the user cannot navigate
  back to the original page using the browser's back button.

```javascript
// Navigate to a new page and replace the current page in the browser history
window.location.replace("https://www.example.com/newpage.html");
```

Choose the appropriate method based on your specific use case and whether you want the new
 page to be added to the browser history or replace the current page.

Authentication and tokens are fundamental concepts in the realm of security
 and identity management, especially in the context of web applications and
 APIs. Let's explore these concepts in detail:

Authentication:

Authentication is the process of verifying the identity of a user, application,
 or system. It ensures that the entity trying to access a resource is who it claims to be
 . There are various methods of authentication, each with its own strengths and use cases:

1. **Username/Password:**

```
   - The user provides a username and password.
   - Common for web applications and traditional login systems.
   - Vulnerable to various attacks, such as phishing.

2. **Multi-Factor Authentication (MFA):**
   - Requires multiple forms of identification (e.g., password + SMS code or fingerprint).
   - Enhances security by adding an additional layer of verification.

3. **Token-Based Authentication:**
   - Uses tokens (e.g., JSON Web Tokens) for authentication.
   - Reduces the need to store sensitive credentials on the client.
   - Often used in modern web applications and APIs.

4. **OAuth and OpenID Connect:**
   - Delegated authorization and authentication protocols.
   - OAuth allows secure delegated access to resources.
   - OpenID Connect is an identity layer on top of OAuth, providing authentication.

### Tokens:

Tokens play a crucial role in modern authentication and authorization systems.
They are used to represent the authenticated user and provide secure access
to protected resources. The most common types of tokens are:

1. **Access Tokens:**
   - Grants access to specific resources on behalf of the user.
   - Short-lived and specific to the user and application.
   - Used in OAuth for authorization.

2. **Refresh Tokens:**
   - Used to obtain a new access token.
   - Longer-lived than access tokens.
   - Stored securely on the client.

3. **JSON Web Tokens (JWT):**
   - A compact, URL-safe means of representing claims between two parties.
   - Self-contained, containing information about the user or system.
   - Often used as access tokens in token-based authentication.

### Token-Based Authentication Flow:

1. **User Authentication:**
   - The user provides credentials (e.g., username/password) to the authentication server.

2. **Token Issuance:**
   - Upon successful authentication, the authentication server generates an access token
   (and optionally a refresh token).

3. **Token Storage:**
   - The access token is stored securely on the client (e.g., in a cookie or local storage).

4. **Token Usage:**
   - The client includes the access token in the headers of API requests to access protected
resources.

5. **Token Expiry and Refresh:**
```

```
   - Access tokens have a limited lifespan. If they expire, the client can use the refresh
   token to obtain a new access token without requiring the user to re-enter credentials.

### Benefits of Token-Based Authentication:

1. **Statelessness:**
   - No need to store user sessions on the server.
   - Each request contains the necessary authentication information.

2. **Scalability:**
   - Stateless nature simplifies scaling, as there's no need to synchronize session state
    across multiple servers.

3. **Security:**
   - Tokens can be encrypted and signed to ensure integrity and confidentiality.
   - Reduced risk of Cross-Site Request Forgery (CSRF) and session hijacking.

4. **Decoupling Frontend and Backend:**
   - The frontend and backend can be developed independently, as long as they adhere to the
token contract.

In summary, authentication verifies the identity of a user or application, while tokens play
a crucial
role in securely representing and granting access to resources. Token-based authentication,
 especially using technologies like OAuth and JWTs, has become a standard in modern web
development
 due to its security, scalability, and flexibility.


 */
```