

JSON

JSON (JavaScript Object Notation) is a lightweight data interchange format that's easy for humans to read and write and easy for machines to parse and generate. It is often used for transmitting data in web applications between a server and a client.

Here are some key points about JSON:

Syntax:

- JSON data is written as key/value pairs.
- Keys are strings enclosed in double quotes.
- Values can be strings, numbers, objects, arrays, true, false, or null.

Structure:

- An object is an unordered set of key/value pairs enclosed in curly braces { }.
- An array is an ordered collection of values enclosed in square brackets [].

Basic of JSON –

key/name value pairs

```
{ "name" : "value" }
```

Objects are comma separated

```
{ "name1" : "value" , "name2" : "value" , "name3" : "value" }
```

Arrays have square brackets with values separated by comma

```
{ "name" : [ { "name" : "value" }, { "name" : "value" } ] }
```

JSON Data Types

String:

A sequence of characters enclosed in double quotes.

Number:

Can be an integer or a floating-point number.

Boolean:

Represents a logical value, either true or false.

Object:

An unordered set of key/value pairs enclosed in curly braces {}.

Keys must be strings, and values can be of any JSON data type.

Array:

An ordered collection of values enclosed in square brackets [].

Values can be of any JSON data type.

Null:

Represents a null value.

```
{
  "string": "Hello, JSON!",
  "number": 123,
  "float": 45.67,
  "boolean": true,
  "nullValue": null,
  "object": {
    "nestedString": "Nested Hello",
    "nestedNumber": 456
  },
  "array": ["one", 2, false, null, {"key": "value"}]
}
```

JavaScript Reminder

See below code snippets & output –

Normal JS values:

```
let string = "hello";
let number = 14;
let Null = null;
let Undefined = undefined;
```

Output:

```
hello
14
null
undefined
```

JS Array Examples:

```
//normal array that can have multiples arrays inside it
let arr1 = [12,4,"hello",[5,"Hi"]];
//array can have JS objects
let arr2 = [{"name":"Titas"}, {"Role":"React"}];
```

Output:

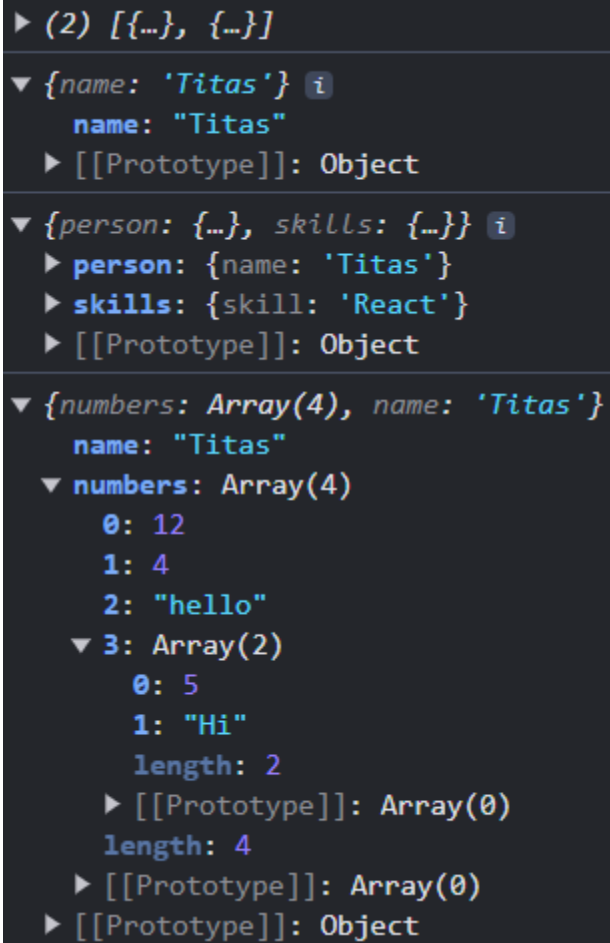
```
(4) [12, 4, 'hello', Array(2)]
  0: 12
  1: 4
  2: "hello"
  3: Array(2)
    0: 5
    1: "Hi"
    length: 2
  [[Prototype]]: Array(0)
length: 4
[[Prototype]]: Array(0)

(2) [{...}, {...}] ⓘ
  0: {name: 'Titas'}
  1: {Role: 'React'}
  length: 2
  [[Prototype]]: Array(0)
```

JS Object examples:

```
//normal JS object
let obj1 = {name:"Titas"};
//JS object can have JS objects inside it
let obj2 = {person:{name: "Titas"}, skills:{skill: "React"}};
//JS object can have array inside it
let obj3 = {numbers:arr1,name:"Titas"};
```

Output:



```
▶ (2) [{...}, {...}]
▼ {name: 'Titas'} ⓘ
  name: "Titas"
  ▶ [[Prototype]]: Object
▼ {person: {...}, skills: {...}} ⓘ
  ▶ person: {name: 'Titas'}
  ▶ skills: {skill: 'React'}
  ▶ [[Prototype]]: Object
▼ {numbers: Array(4), name: 'Titas'}
  name: "Titas"
  ▼ numbers: Array(4)
    0: 12
    1: 4
    2: "hello"
    ▼ 3: Array(2)
      0: 5
      1: "Hi"
      length: 2
    ▶ [[Prototype]]: Array(0)
    length: 4
    ▶ [[Prototype]]: Array(0)
    ▶ [[Prototype]]: Object
```

Above examples are reminder of JS so that we can have clear picture of it. Because in real scenario we have to convert JSON data coming from API call to the JS object & then we can use JS object into our frontend to show data.

JS array are written in [] braces & JS Object are written in { } braces.
JS Object have key-value pairs.

JSON format vs JS Object format

```
> let myJson = {"name":"Alex","Age":30};
let myJson2 = {name:"Alex2",Age:35};
let myJson3 = {name:'Alex3',Age:40};

< undefined

> console.log(myJson);
console.log(myJson2);
console.log(myJson3);

▶ {name: 'Alex', Age: 30}
▶ {name: 'Alex2', Age: 35}
▶ {name: 'Alex3', Age: 40}
```

Here is this example see we are creating JS objects. Here all the formats are valid & working correctly, but what is going to happen if we want to check if all JS object formats are valid JSON or not? See below -

```
1 {
2   "name": "Alex",
3   "Age": 30
4 }
```

JSON is valid!

```
1 {name:"Alex2",Age:35}|
```

Invalid JSON!

Error: Parse error on line 1:
{name:"Alex2",Age:35}
-^
Expecting 'STRING', '}', got 'undefined'

```
1 {name:'Alex3',Age:40}|
```

Invalid JSON!

Error: Parse error on line 1:
{name:'Alex3',Age:40}
-^
Expecting 'STRING', '}', got 'undefined'

```
1 {"name":'Alex3',Age:40}|
```

Invalid JSON!

Error: Parse error on line 1:
{"name":'Alex3',Age:40}
-----^
Expecting 'STRING', 'NUMBER', 'NULL', 'TRUE', 'FALSE', '{', '[', got 'undefined'

So we can see that for JSON object we can only use **" "**, also note that in JSON we also can't use function. But in JS object we can have function in object.

JSON example & operations

```
> let jsonObject = {  
  "name": "John",  
  "age": 30,  
  "isStudent": false,  
  "courses": ["Math", "Science"],  
  "address": {  
    "street": "123 Main St",  
    "city": ["Kolkata", "Chennai", "Pune"]  
  }  
};
```

This is a JSON data or we can say JS Object. Now we will do different types of operations to perform modification of JSON data or JS Object & conversion including data accessing & print data.

Access data :

```
> console.log(jsonObject.name);  
console.log(jsonObject.age);  
console.log(jsonObject.isStudent);  
console.log(jsonObject.courses);  
console.log(jsonObject.courses[0]);  
console.log(jsonObject.address.street);  
console.log(jsonObject.address.city);  
console.log(jsonObject.address.city[1]);
```

```
John  
30  
false  
▶ (2) ['Math', 'Science']  
Math  
123 Main St  
▶ (3) ['Kolkata', 'Chennai', 'Pune']  
Chennai
```

Modify data:

```
> jsonObject.age = 31;  
jsonObject.courses.push("History");  
jsonObject.address.street = "456 Elm St";
```

Again print all data –

```
John  
31  
false  
▶ (3) ['Math', 'Science', 'History']  
Math  
456 Elm St  
▶ (3) ['Kolkata', 'Chennai', 'Pune']  
Chennai
```

Convert JS Object to JSON string

Previously we had create a JSON object as a JS object literal. Now we need to modify this as a JSON string. See when we call an API we will get JSON data as a JSON string format. So for that first we need to modify JSON string data coming from API call into JS object using parse method. Then we can perform any JS operation & again if we need to modify database with new data then again we need to change the updated JS object data to JSON string & then we can modify change in database using API call. Databases stored data as a string or we can say JSON string. So to modify JS object into usable string format JSON data, we have to use JSON stringify method.

So previously we made a JSON object as a JS object literal directly into our code. Now we will change it to JSON string like this –

```
> let jsonString = JSON.stringify(jsonObject);  
console.log(jsonString);  
  
{"name":"John","age":30,"isStudent":false,"courses":["Math","Science"],"address":{"street":"123 Main St","city":["Kolkata","Chennai","Pune"]}}
```

JSON.stringify() is a method in JavaScript that converts a JavaScript object or value to a JSON string. This is useful for sending data to a web server, storing data in a file, or displaying the data as a string. Note that previously the console log shows the data as a object format, but this time after stringify we can only see the whole data as a string format.

Basic syntax of JSON is –

JSON.stringify(value[, replacer[, space]]);

Value – This is the main parameter. The data we want to convert into JSON string should be wrote here. It can be object, array, number, string, boolean, or null.

```
let obj = { name: "John", age: 30 };  
let jsonString = JSON.stringify(obj);  
console.log(jsonString); // Output: {"name":"John","age":30}
```

Replacer – This is the way to changing the transform properties of stringify method using any function or array. Using function -

```
let obj = { name: "John", age: 30 };

function replacer(key, value) {
  if (key === "age") {
    return undefined; // Exclude the "age" property
  }
  return value; // Otherwise, include the property
}

let jsonString = JSON.stringify(obj, replacer);
console.log(jsonString); // Output: {"name":"John"}
```

```
let filteredJsonString = JSON.stringify(jsonObject, (key, value) => {
  if (key === "age") {
    return undefined; // Omit the age property
  }
  return value;
});
console.log(filteredJsonString);
```

Using an array –

```
let obj = { name: "John", age: 30, city: "New York" };

let jsonString = JSON.stringify(obj, ["name", "city"]);
console.log(jsonString); // Output: {"name":"John","city":"New York"}
```

If we don't use anything in replacer or used NULL then stringify will convert the entire object into JSON string considering all the parent and child data sets.

Space –

This make JSON data easier to read by adding blank spaces. We can use number or string value to clarify how our spaces will add.

Numbers simply add the numbers of spaces for indentation –

```
let obj = { name: "John", age: 30, city: "New York" };

let jsonString = JSON.stringify(obj, null, 2);
console.log(jsonString);
/* Output:
{
  "name": "John",
  "age": 30,
  "city": "New York"
}
*/
```


String specifies the exact string to use as an indentation –

```
let obj = { name: "John", age: 30, city: "New York" };

let jsonString = JSON.stringify(obj, null, "\t");
console.log(jsonString);

/* Output:
{
    "name": "John",
    "age": 30,
    "city": "New York"
}
*/
```

Convert JSON string to JS Object

As we discussed earlier that if we want to use data coming from API call then we need to convert it into JS object. Because data comes as a JSON string.

```
> let jsonString = '{"name":"alex","age":"40","role":"react"}'
let jsonObject = JSON.parse(jsonString);
console.log(jsonObject);
```

Output-

```
▼ {name: 'alex', age: '40', role: 'react'} ⓘ
  age: "40"
  name: "alex"
  role: "react"
  ► [[Prototype]]: Object
```

Basic syntax of parse is –

JSON.parse(text[, reviver])

Text- this has to be the JSON string. Note the string must be a valid JSON format.

Reviver- This is a function that can be used to change or transform the output of parse method before it returns the final result based on operations performed inside function. **Note that this**

method calls every time for each key-value pair of JSON string & using this method we can only change **values** not **keys**. For changing keys we need to explicitly perform another operation.

```
> let jsonString = '{"name":"alex","age":"40","role":"react"}';
  function reviver(key, value) {
    if(key === "age")
    {
      return 30;
    }
    return value;
  }
  let jsonObj = JSON.parse(jsonString, reviver);
  console.log(jsonObj);
```

Output-

```
▼ {name: 'alex', age: 30, role: 'react'} ⓘ
  age: 30
  name: "alex"
  role: "react"
  ► [[Prototype]]: Object
```

Previously the age was 40. But using parse method & reviver function we modify age value & also change it from JSON string to JS object.

Error Handling

We have to consider error scenario while using parse method. It may happen that the data we want to parse is not correctly formatted as JSON format. So at that time parse method throws & exception. Now we have to handle that exception.

```
try {
  let jsonString = '{"name":"alex,"age":"40","role":"react"}';
  let jsonObj = JSON.parse(jsonString, reviver);
  console.log(jsonObj);
}
catch (e) {
  console.log(e.message);
}
```

Output-

```
Expected ',', or '}' after property value in JSON at position 15 (line 1 column 16)
```

See above I deleted " from "alex" & using try-catch block we are handling error.

Example of JSON using JS & HTML

```
<!DOCTYPE html>
<html>
  <head>
    <title>Input to Object</title>
  </head>
  <body>
    <label for="firstName">First Name:</label>
    <input type="text" id="firstName" name="firstName" />
    <br />
    <label for="lastName">Last Name:</label>
    <input type="text" id="lastName" name="lastName" />
    <br />
    <button id="addButton">Add</button>
    <button id="viewButton">View Object</button>

    <script>
      // Initialize an empty object to store the names
      let person = {};

      // Function to add names to the object
      function addNames() {
        // Get the values from the input fields
        const firstName = document.getElementById("firstName").value;
        const lastName = document.getElementById("lastName").value;

        // Add the names to the person object
        person.firstName = firstName;
        person.lastName = lastName;

        // Clear the input fields
        document.getElementById("firstName").value = "";
        document.getElementById("lastName").value = "";
      }

      // Function to view the object in the console
      function viewObject() {
        let person2 = JSON.stringify(person);
        console.log(person);
        console.log(person2);
      }

      // Attach event listeners to the buttons
      document.getElementById("addButton").addEventListener("click", addNames);
      document
        .getElementById("viewButton")
        .addEventListener("click", viewObject);
    </script>
  </body>
</html>
```

Fetch JSON data using fetch(API) call

```
<!DOCTYPE html>
<html>
  <head>
    <title>Input to Object</title>
  </head>
  <button id="firstName" name="firstName">click me to see JSON in console</button>

  <script>
    const Name = document.getElementById('firstName').addEventListener('click',JSONsee);
    function JSONsee() {
      console.log("Clicked");
      const url = "https://api.npoint.io/57ab289b387d233f0292";
      fetch(url)
        .then(function(response){
          console.log(response.status);
          return response.text();
        })
        .then(function(data){
          console.log(data);
        })
        .catch(function(e){
          console.log("there is an error in your fetch operation",e);
        })
    }
  </script>
</body>
</html>
```

Here when we click the button then in console we will get JSON data coming from API.

```
Clicked
200
{"age":30,"arr":[2,5,8,"Hii",10,"hello"],"Role":"React","name":"Alex"}
```

First we target button using id & then we handle click event and same time calling function **JSONsee()**.

In the function we store API link in the variable & then using **fetch** we initiate network request to the specify url. It returns a promise that resolves to the **response** object.

Now we handle promises by consuming it using **.then , .catch** . Inside these we declare functions that will **log** the operations data in the console.

JS Reminder

Key Concepts of Promises :

1. States:

- **Pending:** The initial state. The promise is neither fulfilled nor rejected.
- **Fulfilled:** The operation completed successfully.
- **Rejected:** The operation failed.

2. Settled:

- A promise is considered settled when it is either fulfilled or rejected.

3. Chaining:

- Promises can be chained together to handle sequences of asynchronous operations.

Consuming Promises :

Promises are consumed using the then, catch, and finally callback methods.

- **then():** Called when the promise is fulfilled.
- **catch():** Called when the promise is rejected.
- **finally():** Called when the promise is settled (either fulfilled or rejected).

Asynchronous Operation :

An asynchronous operation is a non-blocking operation that allows the program to continue executing other tasks while waiting for the operation to complete. This is essential in scenarios where operations might take time, such as network requests, file reading, or timers.

Synchronous Operation

```
console.log('Start');  
console.log('Middle');  
console.log('End');
```

Here first start & end will log in console & then after 1 sec middle will log in console because setTimeout() is a Asynchronous Operation.

```
console.log('Start');  
  
setTimeout(() => {  
    console.log('Middle');  
}, 1000);  
  
console.log('End');
```

Asynchronous Operation

Fetch multiple JSON data from API & render in React

```
import React, {useState, useEffect} from 'react';
import './styles.css';

export default function App() {
  const [setarr, resetarr] = useState([]);
  useEffect(() => {
    const url = "https://randomuser.me/api/?results=10";
    fetch(url)
      .then(response => {return response.text();})
      .then(data => {
        let arr = JSON.parse(data);
        resetarr(arr.results);
      })
      .catch(e => console.log("error in fetching data",e));
  }, []);

  return (
    <div className="App">
      <h1>Hello CodeSandbox</h1>
      <ul>{setarr.map((persons, index) => (<li key={index}>{persons.name.first}{persons.name.last}</li>))}</ul>
    </div>
  );
}
```

Following the upper code we can have a clear idea how we are using **fetch()** property & having JSON data and then finally render it in UI.

OUTPUT:

Hello CodeSandbox

- AllenWard
- FelixPark
- ApoorvaNair
- TimLecomte
- KarlaPoulsen
- BlaiseLefebvre
- EverettPatterson
- NoeliaIbáñez
- YaseminDurak
- LucilleColeman

Here one thing notice that we can do the upper code in 2 different ways however one process is much more efficient than another.

The other approach we can modify the code in these lines—

```
fetch(url)
  .then(response => response.json())
  .then(data => resetarr(data.results))
  .catch(e => console.log('error in fetching data', e));
}, []);
```

Look here we are converting the JSON string data into JS object in the first **.then()**

Now we simply go to the results array by searching **data.results** of the JS object from **response.json()**

Here if we look for the JSON data format then we can see that the main object that we have from **response.json()** has 2 arrays. We just only need 2nd array that have the valuables data.

So either we first return **text()** from 1st **then()** & then we convert the string data to JS object using **JSON.parse()** & then we can access JSON's results array data storing in a variable like this —

```
.then(data => {
  let arr = JSON.parse(data);
  resetarr(arr.results);
})
```

Or we can directly modify JSON data into JS object as shown in upper 1st image & then we can easily move to results array.

Below is the structure of JS object receive & converted from JSON by fetch API—

```
▼ {results: Array(10), info: {...}} i
  ► info: {seed: '22db4646cd82cb2d', results: 10, page: 1, version: '1.4'}
  ▼ results: Array(10)
    ► 0: {gender: 'male', name: {...}, location: {...}, email: 'miguel.candelaria@example.com', login: {...}, ...}
    ► 1: {gender: 'female', name: {...}, location: {...}, email: 'laurita.pereira@example.com', login: {...}, ...}
    ► 2: {gender: 'female', name: {...}, location: {...}, email: 'teresa.meyer@example.com', login: {...}, ...}
    ► 3: {gender: 'male', name: {...}, location: {...}, email: 'ernesto.delgado@example.com', login: {...}, ...}
    ► 4: {gender: 'female', name: {...}, location: {...}, email: 'emily.white@example.com', login: {...}, ...}
    ► 5: {gender: 'male', name: {...}, location: {...}, email: 'brajko.nemanjic@example.com', login: {...}, ...}
    ► 6: {gender: 'female', name: {...}, location: {...}, email: 'karina.limon@example.com', login: {...}, ...}
    ► 7: {gender: 'female', name: {...}, location: {...}, email: 'imogen.anderson@example.com', login: {...}, ...}
    ► 8: {gender: 'female', name: {...}, location: {...}, email: 'harper.bell@example.com', login: {...}, ...}
    ► 9: {gender: 'male', name: {...}, location: {...}, email: 'harry.wilson@example.com', login: {...}, ...}
    length: 10
    ► [[Prototype]]: Array(0)
  ► [[Prototype]]: Object
```

Now we will discuss about different HTTP request we use. Previously we saw that we are only using `fetch()` & JS promise, but when we actually call API in our React code, we use **async/await**. Inside **async/await** we use either **fetch()** or external library like **Axios**.

So why should we use **async/await** every time though it's not necessary?

- There are several points those have advantages over others, like –
 - **async/await** provides more cleaner code that is easily readable.
 - **async/await** omits **.then() callbacks**. So for that nesting callbacks will not necessary.
 - Using **async/await** we can handle errors easily using **try-catch** block. Using this method we can handle `fetch` error & `parse()` error in same block.
 - When we **await** a **promise** then the current **async function** will be paused until the promise resolves or rejects. Rest of the code will execute parallelly.

This part of code will not execute until the promise is being resolved

```
import React, { useState, useEffect } from "react";
import "./styles.css";

export default function App() {
  const [setData, resetData] = useState([]);

  useEffect(() => {
    const fetchdata = async () => {
      try {
        const response = await fetch("https://randomuser.me/api/?results=5");
        const data = await response.json();
        resetData(data.results);
      } catch (e) {
        console.log("there is an error with data fetching");
      }
    };
    fetchdata();
  }, []);

  return (
    <div className="App">
      <h1>Async/Await example</h1>
      <ul>
        {setData.map((person, index) => (
          <li key={index}>
            {person.name.first}
            {person.name.last}
          </li>
        ))}
      </ul>
    </div>
  );
}
```

This is the **async function**. When the promise is await for resolve, this whole function will be paused.

Async/Await example

- FortunataBarros
- EthanGinnish
- ToddTucker
- GerBrandon
- DianaMorgan

In the above example see we use async function & inside the async function we declare try-catch block. Note that we use await fetch. This means that until the first fetch promise is not resolve (either successful or rejected), the whole async function will be paused & rest of the code will also not execute (rest of the code means from 2nd await fetch data to console.log of catch block).

This means that await keyword ensures that the next line of code (awaited line) is not executed until the promise being awaited settles (resolves or rejects).

Now we will see what happened when we call 2 APIs parallelly or if we have to fetch 2 APIs then how should we do that?

By default, await does indeed pause execution until the awaited promise resolves. This ensures that each operation completes in sequence before moving to the next line of code.

So to fetch APIs parallelly, Option 1 is using **Promise.all()** method :

See the next page code & below is the output UI –

Async/Await example

- Sofia Nielsen
- Kaitlin Griffin
- Francielle Moura
- Charlie Fernandez

- Nikolaus Pfaff
- Edmund Myrstad
- Yashika Acharya
- Marlisa Ermens
- Rosie Davidson
- Undine Strasser

```

import React, { useState, useEffect } from "react";
import "./styles.css";

export default function App() {
  const [setData, resetData] = useState([]);
  const [setArr, resetArr] = useState([]);

  useEffect(() => {
    async function fetchData() {
      try {
        const [api1Data, api2Data] = await Promise.all([
          fetch("https://randomuser.me/api/?results=4").then((response) =>
            response.json()
          ),
          fetch("https://randomuser.me/api/?results=6").then((response) =>
            response.json()
          ),
        ]);

        resetData(api1Data.results);
        resetArr(api2Data.results);
      } catch (error) {
        console.error("Error fetching data:", error);
        throw error;
      }
    }
    fetchData();
  }, []);

  return (
    <div className="App">
      <h1>Async/Await example</h1>
      <ul>
        {setData.map((person, index) => (
          <li key={index}>
            {person.name.first} {person.name.last}
          </li>
        ))}
      </ul>

      <br />

      <ul>
        {setArr.map((person, index) => (
          <li key={index}>
            {person.name.first} {person.name.last}
          </li>
        ))}
      </ul>
    </div>
  );
}

```

Now we will see the another method which is using two async functions :

While await ensures sequential execution within a single async function, you can still achieve parallelism in fetching multiple APIs by starting multiple async functions concurrently and awaiting their results collectively.

```
import React, { useState, useEffect } from "react";
import "./styles.css";

export default function App() {
  const [setData, resetData] = useState([]);
  const [setArr, resetArr] = useState([]);
  useEffect(() => {
    const fetchData1 = async () => {
      try {
        const response = await fetch("https://randomuser.me/api/?results=6");
        const data = await response.json();
        resetData(data.results);
      } catch (error) {
        console.error("Error fetching data:", error);
        throw error;
      }
    };

    const fetchData2 = async () => {
      try {
        const response = await fetch("https://randomuser.me/api/?results=4");
        const data = await response.json();
        resetArr(data.results);
      } catch (error) {
        console.error("Error fetching data:", error);
        throw error;
      }
    };

    fetchData1();
    fetchData2();
  }, []);
```

See here we are defined two different async functions and then we call them. We fetch 2 APIs separately in the separate async functions. This method is more efficient than the previous one.

Rest of the code is in the next page -

```

return (
  <div className="App">
    <h1>Async/Await example</h1>
    <ul>
      {setData.map((person, index) => (
        <li key={index}>
          {person.name.first} {person.name.last}
        </li>
      ))}
    </ul>

    <br />

    <ul>
      {setArr.map((person, index) => (
        <li key={index}>
          {person.name.first} {person.name.last}
        </li>
      ))}
    </ul>
  </div>
);
}

```

Async/Await example

- Cristóbal Santiago
 - Sarah Francois
 - Gökhan Eronat
 - Lydie Koornstra
 - Marianne Ennis
 - Mélissa Leroux
-
- Ross Torres
 - Gerardo Rolón
 - Marion Durand
 - Peppi Ylitalo

Below we will discuss why multiple async function have better efficiency over Promise.all().

Here one thing is very important & noticeable. See in the 1st Promise.all() example, if any API call will get reject or occur any error while fetching json data then the other API will also not shows any output in the UI & the code flow simply goes to catch block. So in general we will not see any data in the UI. See below UI screenshot –

Async/Await example

The same will happen when we modify any of the API link. Note that remove **throw error;** line so that UI didn't shows error. It will only log in console like below -

```
► Error fetching data: TypeError: Failed to fetch
    at fetchData (/src/App.js:21:57)
    at eval (/src/App.js:28:5)
    at commitHookEffectListMount (/node_modules/react-dom.development.js:23189:26)
    at commitPassiveMountOnFiber (/node_modules/react-dom.development.js:24965:13)
    at commitPassiveMountEffects_complete (/node_modules/react-dom.development.js:24930:9)
    at commitPassiveMountEffects_begin (/node_modules/react-dom.development.js:24917:7)
    at commitPassiveMountEffects (/node_modules/react-dom.development.js:24905:3)
    at flushPassiveEffectsImpl (/node_modules/react-dom.development.js:27078:3)
    at flushPassiveEffects (/node_modules/react-dom.development.js:27023:14)
    at eval (/node_modules/react-dom.development.js:26808:9)
    at workLoop (/node_modules/scheduler.development.js:266:34)
    at flushwork (/node_modules/scheduler.development.js:239:14)
    at MessagePort.performWorkUntilDeadline (/node_modules/scheduler.development.js:533:21)
```

But using the different async function technique we can omit this kind of error, means that if any API fetching time error occurs or get rejected then other API call will work parallelly & it still shows it's data in the UI while other one log error in the console as per code. See below example –

```
try {
  const response = await fetch("https://randomuser.me/api?results=6");
  const data = await response.json();
  setData(data.results);
} catch (error) {
  console.error("Error fetching data:", error);
  setData([]);
}
```

See here I modify my code little bit.

- First I delete one letter from the 1st API link so that fetch promise will get resolve with error.
- Secondly I reset the state (setData) by empty array so that whenever we got the error we can see changes in the UI and remove **throw error;** line so that UI didn't shows error. It will only log in console.

Async/Await example

-
-
-
-

Frida Andersen
Claire Meunier
Bekim Dupuis
Adam Wang

```
► Error fetching data: TypeError: Failed to fetch
    at fetchData1 (/src/App.js:21:32)
    at eval (/src/App.js:38:5)
    at commitHookEffectListMount (/node_modules/react-dom.development.js:23189:26)
    at commitPassiveMountOnFiber (/node_modules/react-dom.development.js:24965:13)
    at commitPassiveMountEffects_complete (/node_modules/react-dom.development.js:24930:9)
    at commitPassiveMountEffects_begin (/node_modules/react-dom.development.js:24917:7)
    at commitPassiveMountEffects (/node_modules/react-dom.development.js:24905:3)
    at flushPassiveEffectsImpl (/node_modules/react-dom.development.js:27078:3)
    at flushPassiveEffects (/node_modules/react-dom.development.js:27023:14)
    at commitRootImpl (/node_modules/react-dom.development.js:26974:5)
    at commitRoot (/node_modules/react-dom.development.js:26721:5)
    at performSyncWorkOnRoot (/node_modules/react-dom.development.js:26156:3)
    at flushSyncCallbacks (/node_modules/react-dom.development.js:12042:22)
    at flushSync (/node_modules/react-dom.development.js:26240:7)
    at Object.scheduleRefresh (/node_modules/react-dom.development.js:27834:5)
    at eval (/node_modules/react-dom.development.js:300:17)
    at Set.forEach (<anonymous>)
    at Object.performReactRefresh (/node_modules/react-dom.development.js:289:26)
    at eval (/node_modules/cssbus-esh-helper.js:13:20)
```

See here that whenever the API call have error, it will go to the catch block & log the error in the console & also clear the first state by replacing empty array. Also in parallel the other API resolves correctly so we get the data. So now see that in UI we only have the data coming from 2nd async function.

The same will happen if we only modify 2nd API link & 1st link remain correct. The first data will be shown in the UI & the 2nd list of items will disappear because it will replace by empty array by the catch block.

Axios

Axios is a powerful library that simplifies HTTP request. We can use axios like this way –

```
import React, { useState, useEffect } from "react";
import axios from "axios";
import "./styles.css";

const example = () => {
  const [set, reset] = useState([]);
  useEffect(() => {
    const fetchdata = async () => {
      try {
        const response = await axios.get("https://randomuser.me/api/?results=10");
        reset(response.data.results);
      } catch (e) {
        console.log("there is an error with axios", e);
        throw e;
      }
    };
    fetchdata();
  }, []);

  return (
    <div className="App">
      <h1>Hello CodeSandbox</h1>
      <br />
      <ul>
        {set.map((persons, index) => (
          <li key={index}>{persons.name.first} {persons.name.last}</li>
        ))}
      </ul>
    </div>
  );
}

export default example
```

Remember that Axios is an external library & we need to install first in our project, then we can use that by importing it in the components.

1st we need to install using npm command – **npm install axios**

2nd we need to import axios in our component like – **import axios from “axios”;**

Axios have various methods in next page

1. **axios.get(url[, config])**

This method performs an HTTP GET request to the specified url.

Parameters:

url: The URL to which the request is sent.

config (optional): Configuration object for the request. It can include headers, parameters, authentication credentials, etc.

```
axios.get('https://api.example.com/data')
  .then(response => {
    console.log(response.data);
  })
  .catch(error => {
    console.error('Error fetching data:', error);
  });
```

2. **axios.post(url[, data[, config]])**

This method sends an HTTP POST request to the specified url.

Parameters:

url: The URL to which the request is sent.

data (optional): The data to be sent as the request body.

config (optional): Configuration object for the request.

```
const newData = { name: 'John Doe', age: 30 };

axios.post('https://api.example.com/users', newData)
  .then(response => {
    console.log('Data posted successfully:', response.data);
  })
  .catch(error => {
    console.error('Error posting data:', error);
  });
```

3. **axios.put(url[, data[, config]])**

This method sends an HTTP PUT request to the specified url.

Parameters:

url: The URL to which the request is sent.

data (optional): The data to be sent as the request body.

config (optional): Configuration object for the request.

```
const updatedData = { id: 1, name: 'Updated Name' };

axios.put('https://api.example.com/users/1', updatedData)
  .then(response => {
    console.log('Data updated successfully:', response.data);
  })
  .catch(error => {
    console.error('Error updating data:', error);
  });
```

4. **axios.delete(url[, config])**

This method sends an HTTP DELETE request to the specified url.

Parameters:

url: The URL to which the request is sent.

config (optional): Configuration object for the request.

```
axios.delete('https://api.example.com/users/1')
  .then(response => {
    console.log('Data deleted successfully:', response.data);
  })
  .catch(error => {
    console.error('Error deleting data:', error);
  });
```


In conclusion -

- Promises are straightforward but can become nested and harder to read with complex asynchronous code.
- `async/await` provides a cleaner and more readable way to write asynchronous code, making it easier to handle errors and maintain code.
- Axios is a powerful library that simplifies HTTP requests with features like request and response interceptors, automatic JSON data transformation, and more.

IMPORTANT LINKS

- <https://jsonlint.com/> - JSON validator
- <https://www.npoint.io/> - make your own JSON data & API link
- <https://randomuser.me/> -premade JSON data API creating random user details
 - [*Made by Titas*](#)