Master Asynchronous javascript

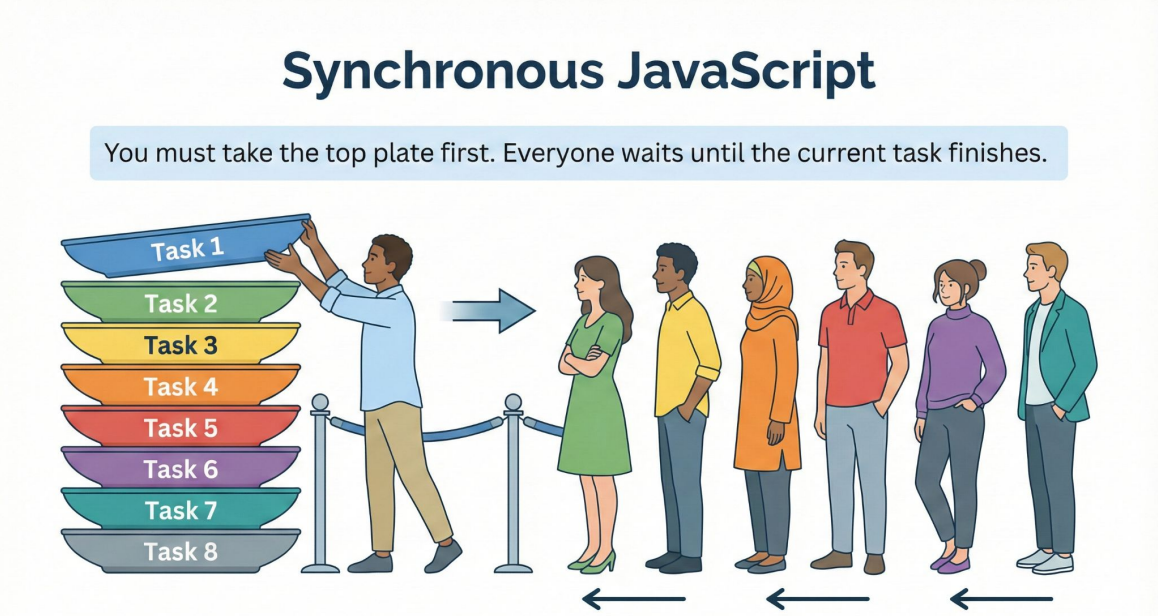
Synchronous and Asynchronous javascript

Courses Overview

- Synchronous JavaScript
- Asynchronous JavaScript
- Why async matters in real-world apps
- Callbacks
- Promises
- Async/Await
- Fetch API & HTTP requests
- Error handling
- Real-world projects

Synchronous javaScript /blocking

- Code runs **step by step**
- Each line must finish before the next line starts
- If one task is slow, everything stops and waits



Synchronous /blocking

So the javaScript Code

```
console.log("Task 1");

console.log("Task 2");

console.log("Task 3");

console.log("task 4");
```

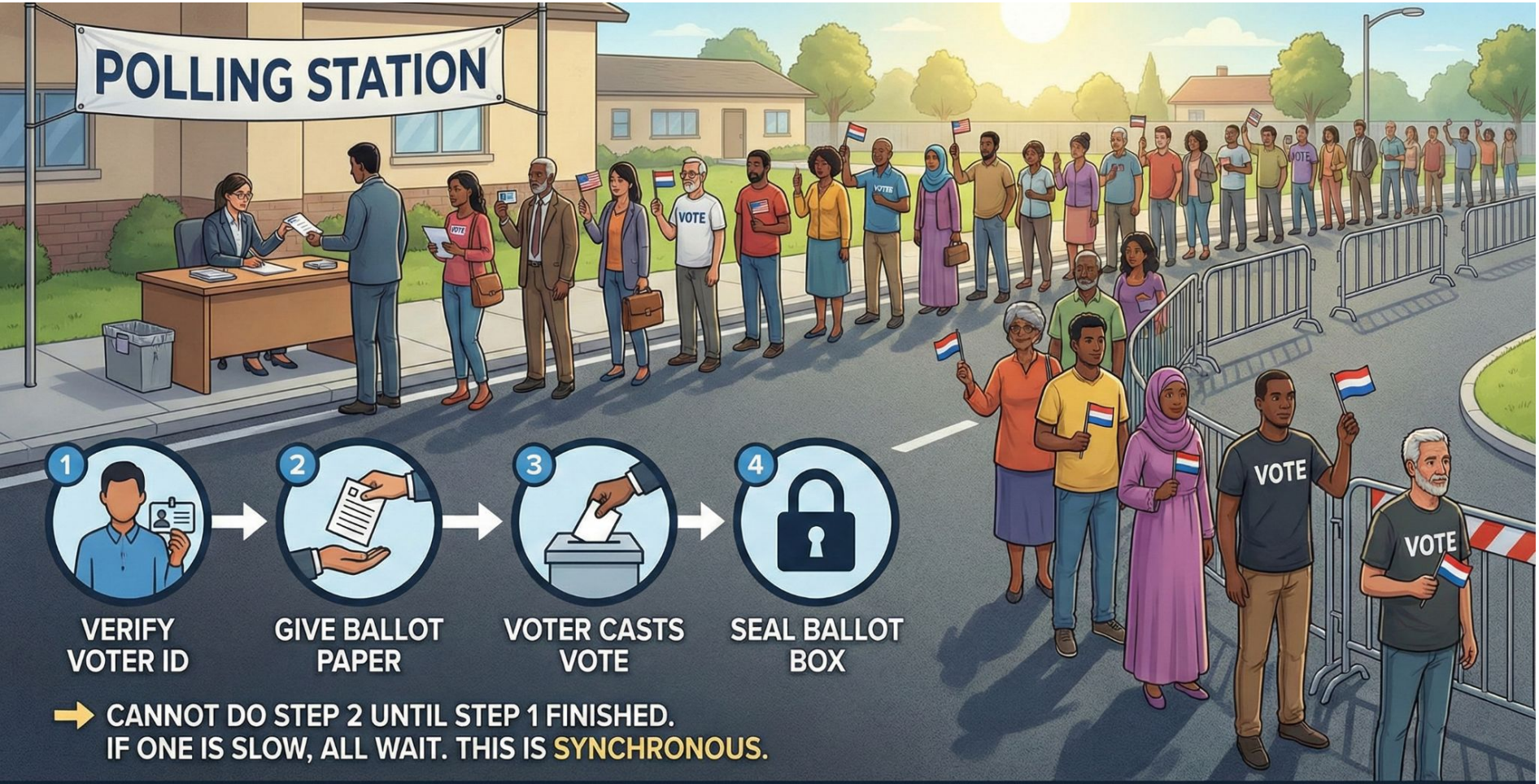
Synchronous /blocking

Example : **Election Process**

imagine you are an election officer helping **one voter at a time**:

1. Check voter ID
2. Give ballot paper
3. Voter marks vote
4. Seal ballot box

You cannot jump to step 2 until step 1 is complete.
Everything happens in a strict order.



SYNCHRONOUS ELECTION PROCESS CONCEPT

Synchronous javaScript /blocking

```
console.log("Check voter ID");

console.log("Give ballot paper");

console.log("Voter marks vote");

console.log("Seal ballot box");
```

This will **always** run in the same exact order.

Problems with Synchronous javaScript /blocking

If one voter is slow, everyone behind them must wait.
This causes:

- Freezing
- Delays
- Poor user experience

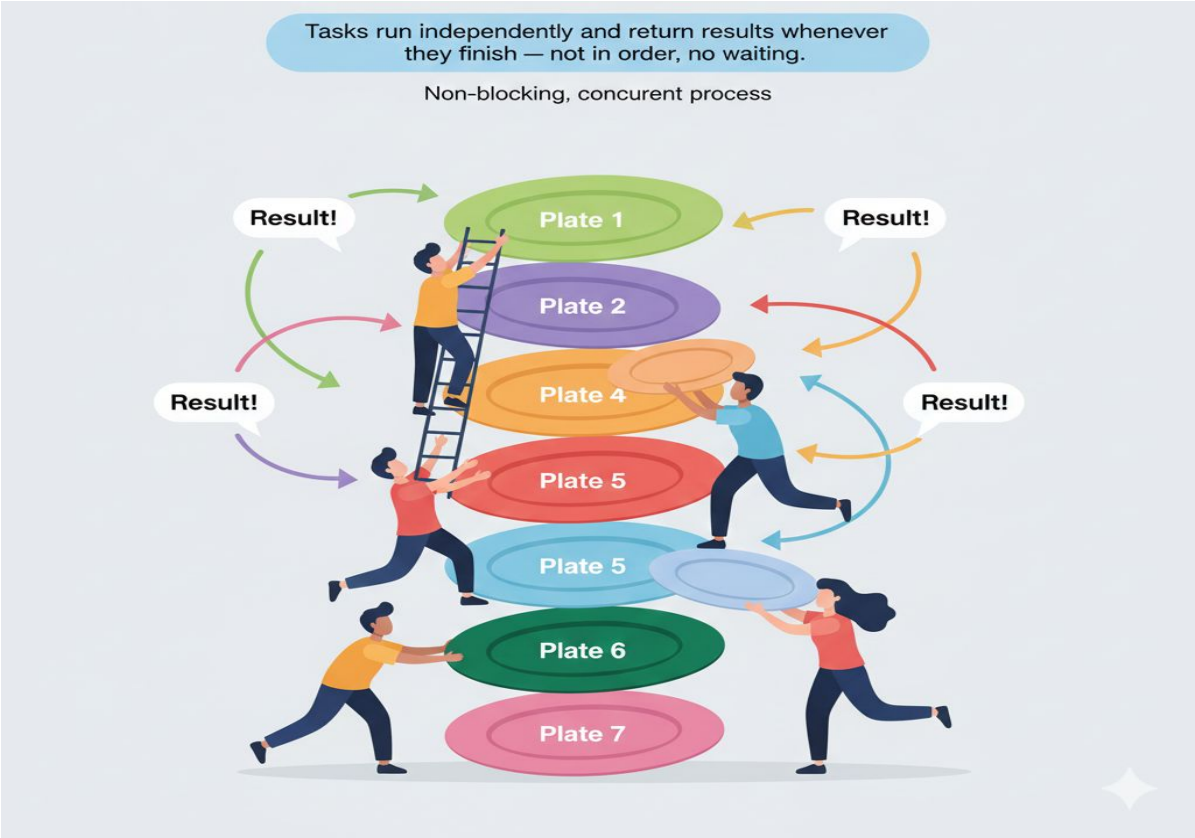
This is why JavaScript uses **Asynchronous behavior.**

Asynchronous javaScript

JavaScript can **do many things without waiting** for one task to finish.

Like **multiple chefs** cooking different foods at the same time.
While one chef is waiting for food to boil, another chef can fry something.

What is Asynchronous javaScript /non-blocking



Why Async Matters in Real-World Apps

Modern apps rely heavily on internet, servers, and waiting operations.

📝 Real Examples

- YouTube fetching recommended videos
- Instagram loading photos (HTTP request)
- Uber tracking driver location (continuous API calls)

What is Asynchronous javaScript /non-blocking

`setTimeout` goes to **Web APIs**, not JS's main thread.
JS continues running.

```
console.log("1");

setTimeout(() => console.log("2"), 2000);

console.log("3");
```

What is Asynchronous javaScript /non-blocking

Asynchronous = code that **does not block** the main thread.
The browser/Web APIs handle the waiting.

Asynchronous JavaScript means:

- JS starts a task
- JS does **not** wait
- JS continues with other tasks
- JS returns later when the result is ready

This makes apps fast and responsive.

What is Asynchronous javaScript /non-blocking

```
console.log("1. JS starts a task");

setTimeout(() => {
  console.log("4. JS returns later when the result is ready");
}, 2000);

console.log("2. JS does NOT wait");

console.log("3. JS continues with other tasks");
```

What is Asynchronous javaScript /non-blocking

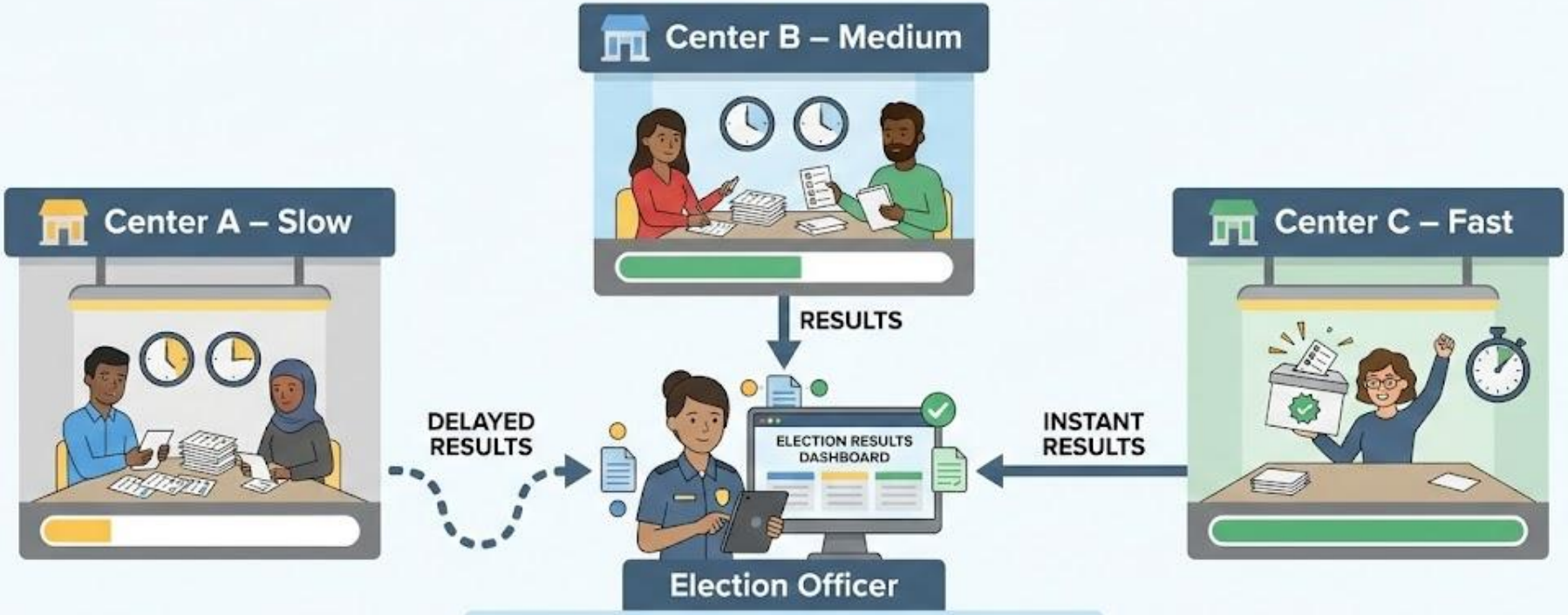
Election Example (Asynchronous)

During vote counting:

- Center A finishes in 3 seconds
- Center B finishes in 1 second
- Center C finishes instantly without waiting

You don’t wait for A before asking B and C.
They report when they are ready.
Results come in **any order**, depending on time.

Asynchronous Election Example



Each center works independently.
Results return when ready, not in order.

Asynchronous Workflow: Non-blocking process.

What is Asynchronous javaScript /non-blocking

```
console.log("Start counting...");
// 1. Center A (Written First - Slowest)
setTimeout(() => {
 console.log("Center A finished");
}, 3000);
// 2. Center B (Written Second - Medium)
setTimeout(() => {
 console.log("Center B finished");
}, 1000);
```

What is Asynchronous javaScript /non-blocking

```
// 3. Center C (Written Last - Instant!)
setTimeout(() => {
    console.log("Center C finished");
}, 0);


console.log("Waiting for results...");
```

Callbacks

A callback is:

A function that runs **later**, after an asynchronous task finishes.

Election example:
“Call me when your counting center finishes.”

```
setTimeout(() => {
    console.log("Center A is done!");
}, 2000);
```

Callbacks

A callback is a function executed later.

```
// SIMPLE CALLBACK EXAMPLE
// Wash clothes first, then dry them (callback runs after the task)
function washClothes(callback) {
  console.log("Washing clothes...");
  setTimeout(() => {    console.log("Clothes are washed!");
    callback(); // run later   }, 1000);}
washClothes(() => {  console.log("Now we dry the clothes!");
});
```

Callbacks Hell

Callbacks get messy when tasks depend on each other.

Callback Hell Example

```
// Many steps depend on the previous one → becomes deeply nested

function wash(callback) {

  setTimeout(() => {    console.log("1. Clothes washed");

    callback();

  }, 500);}
```

Callbacks Hell

```
function dry(callback) {  setTimeout(() => {    console.log("2. Clothes dried");
    callback();   }, 500);}
function fold(callback) {
  setTimeout(() => {    console.log("3. Clothes folded");
    callback();  }, 500);}
function store(callback) {
  setTimeout(() => {    console.log("4. Clothes stored in closet");
    callback();  }, 500);}
```

Callback Hell

```
// NESTED CALLBACK HELL

wash(() => {

  dry(() => {    fold(() => {

      store(() => {

        console.log("Done — but this is callback hell!");      });

    });   }); });
```

This becomes: hard to read hard to debug hard to scale

This problem forced the world to create **Promises.**

Promises

It helps to solve callback hell

A Promise represents a value that will **arrive later**.

- It may come soon
- It may come late
- JS will run the code once the Promise is fulfilled

A **Promise** in JavaScript is like a *container* that says:

> “I will finish this task later. When I finish, I will tell you whether it worked or failed.”

You can think of a Promise like a **school homework**:

- When you get the homework → you promise to finish it later.
- When you finish → you tell your teacher.

Promises

A Promise has **3 states**:

Promise States

- **Pending** → still working
- **Fulfilled** → success
- **Rejected** → error

⭐ **Promise States (very important!)**

1. PENDING

- The task has started.
- It is still working.
- It didn’t finish yet.

Promises

2. FULFILLED (RESOLVED)

- The task **finished successfully**.
- Promise gives a result.

3. REJECTED

- When the task **failed** the Promise gives an error.

Promises

2. Basic Promise Example (simple)

```
let homework = new Promise((resolve, reject) => {
  let done = true;
  if (done) {     resolve("I finished my homework!");
  } else {
    reject("I couldn't do my homework.");
  } });
```

- `resolve()` → success
- `reject()` → failure

Promises

```
// ---- 1. Wash function (returns a Promise) ----

function wash() {

  return new Promise((resolve) => {

    setTimeout(() => {      console.log("1. Clothes washed");

      resolve(); // Success → move to next

    }, 500);  }); }
```

Promises

```
// ---- 2. Dry function ----

function dry() {

  return new Promise((resolve) => {

    setTimeout(() => {      console.log("2. Clothes dried");

      resolve();

    }, 500);  });}
```

Promises

```
// ---- 3. Fold function ----

function fold() {

  return new Promise((resolve) => {

    setTimeout(() => {

      console.log("3. Clothes folded");

      resolve();

    }, 500);  });}
```

Promises

```
// ---- 4. Store function ----

function store() {

  return new Promise((resolve) => {

    setTimeout(() => {

      console.log("4. Clothes stored in closet");

      resolve();

    }, 500);   });}
```

Promises

```
// ---- RUNNING EVERYTHING IN ORDER (NO CALLBACK HELL) ----

wash()    .then(dry)     // After washing, dry

  .then(fold)    // After drying, fold

  .then(store)   // After folding, store

  .then(() => {    console.log("All done — no more callback hell!");  })

  .catch((error) => {    console.log("Something went wrong:", error);  });
```

The creates then Promise Chaining Hell

Promises

Cleaner than callback hell.

❌ **Problem with Promises**

They are cleaner but still:

- Verbose
- Too many `.then()`
- Harder with try/catch

Solution → **Async/Await**

Async/await

Solving Promise chaining Hell

Async Makes a function "special" — now it can use `await`.

Await Means: “Stop here and wait until this Promise finishes.”

It pauses the function until the step is done.
This makes your code run **top to bottom like real steps**.

```
// ---- 1. Wash function that returns a Promise ----
function wash() {  return new Promise((resolve) => {
    setTimeout(() => {      console.log("1. Clothes washed");
      resolve();     }, 500);  });}
```

Async/await

```
// ---- 2. Dry function ----
function dry() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("2. Clothes dried");
      resolve();
    }, 500);
  }); }
```

Async/await

```
// ---- 3. Fold function ----
function fold() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("3. Clothes folded");
      resolve();
    }, 500);
  });}
```

Async/await

```
// ---- 4. Store function ----
function store() {
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log("4. Clothes stored in closet");
      resolve();
    }, 500);
  });}
```

Async/await

```
// ---- 5. The async function that runs everything in order ----
async function runLaundry() {
  await wash();   // Wait for washing to finish
  await dry();    // Then dry
  await fold();   // Then fold
  await store();  // Then store
  console.log("All done — async/await makes it super clean!");}
// ---- START the laundry process ----
runLaundry();
```

Fetch API and HTTP Requests

```
async function loadCountries() {
  try {
    // 1 Send HTTP GET request
    const res = await fetch(

"https://restcountries.com/v3.1/all?fields=name,capital,population,languages,flags"
    );
    // 2 Check if request was successful
    if (!res.ok) throw new Error(`HTTP error! status: ${res.status}`);

    // 3 Parse JSON response
    const countries = await res.json();
```

Fetch API and HTTP Requests

```
    // 4 Do something with data
    console.log(countries);
    // console.log(countries[0]);
    // console.log(countries[0].name.common)
    // print all countries
    countries.forEach((country) => {
      console.log("Country:", country.name.common);
      console.log("Capital:", country.capital?.[0]);
      console.log("Population:", country.population);
      console.log("Flag:", country.flags.png);
      console.log("----------------------");
    });   } catch (error) {     console.error("Error fetching countries:", error);   }
}
 loadCountries();
```

Error Handling

Problem: Fetch won’t throw errors by itself

Even 404 is NOT an automatic error.

Proper Error Handling

```
// Error Handling
async function load() {
  const res = await fetch("https://wrong-url.com/users");

  if (!res.ok) {
    throw new Error("Failed: " + res.status);
  }
```

Error Handling

```
  return "success";
}

load()
  .then(result => console.log(result))
  .catch(error => console.error("Error:", error.message));
```

Real World Projects

realproject/project.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>project</title>
    <link rel="stylesheet" href="./project.css">
</head>
<body>
```

Real World Projects

```
<h1>Coutries Dashoard Real project</h1>
<input type="text" id="searchInput" placeholder="Search by Country">
<table id="countriesTable">
    <thead>
        <tr>
            <th>Flag</th>
            <th>Country Name</th>
            <th>Capital</th>
            <th>Population</th>
            <th>Languages</th>          </tr>       </thead>
```

Real World Projects

```
        <tbody>


        </tbody>
    </table>
    <script src="./project.js"></script>


</body>
</html>
```

Real World Projects

realproject/project.css

```
*{    margin: 0;
    padding: 0;}
body{    font-family: Arial, Helvetica, sans-serif;    padding: 20px;
}

input{    margin-bottom: 20px;
    padding: 20px;
    width: 300px;
    font-size: 16px; }
```

Real World Projects

```
table{    border-collapse: collapse;
    width: 100%;}
th, td {    border: 1px solid #333;
    padding: 8px;
    text-align: left;}
th{    background-color: #f2f2f2;}

img{    width: 40px;}
```

Real World Projects

realproject/project.js

```
let allCountries = [];

async function loadCountries() {
  try {
    const res = await fetch(

"https://restcountries.com/v3.1/all?fields=name,capital,population,languages,flags"
    );
```

Real World Projects

```
    if (!res.ok) throw new Error(`HTTP error! status: ${res.status}`);

    const countries = await res.json();

    // Sort alphabetically
    allCountries = countries.sort((a, b) =>
      a.name.common.localeCompare(b.name.common)
    );

    displayCountries(allCountries);
  } catch (error) {
    console.error("Failed to load countries:", error);
  }
}
```

Real World Projects

```
function displayCountries(countries) {
  const tbody = document.querySelector("#countriesTable tbody");
  tbody.innerHTML = ""; //clear previous data

  countries.forEach((country) => {
    const row = document.createElement("tr");

    row.innerHTML = `
      <td>
        <img src="${country.flags.png}" alt="Flag of ${country.name.common}">
      </td>
      <td>${country.name.common}</td>
      <td>${country.capital?.[0] || "N/A"}</td>
```

Real World Projects

```
      <td>${country.population.toLocaleString()}</td>
      <td>${
        country.languages ? Object.values(country.languages).join(", ") : "N/A"
      }</td>
    `;

    tbody.appendChild(row);
  }); //
}
```

Real World Projects

```
const searchInput = document.getElementById("searchInput");

searchInput.addEventListener("input", (e) => {
  const searchTerm = e.target.value.toLowerCase();

  const filtered = allCountries.filter((c) =>
    c.name.common.toLowerCase().includes(searchTerm)
  );

  displayCountries(filtered);
});

loadCountries();
```