# **Fetch API in JavaScript**

**🔷 What is it?**

The Fetch API provides a modern way to make **HTTP requests** in JavaScript, like getting data from a server or sending data to one.

It is promise-based, replacing older methods like XMLHttpRequest.

**🔷 Why was it introduced?**

* Simplify and modernize **asynchronous requests**
* Provide cleaner, more readable syntax using Promises (and async/await)
* Easier error handling and chaining
* Integrates better with modern APIs and frameworks

**🔷 Rules and Syntax**

// Basic fetch

fetch('https://api.example.com/data')

.then(response => response.json()) // Converts response to JSON

.then(data => console.log(data)) // Use the data

.catch(error => console.error('Error:', error));

* fetch(url, options):
  + url: the API endpoint
  + options: optional object for methods like POST, headers, body, etc.
* For POST requests:

fetch('https://api.example.com/data', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify({ name: 'John', age: 30 })

});

**🔷 When and Where to Use**

* When you need to:
  + Fetch data from REST APIs (e.g., weather, news, user data)
  + Submit form data to a server
  + Load content dynamically on a webpage
  + Build single-page applications (SPAs)

**🔷 Real World Example**

Fetching a list of users and displaying on a webpage.

**🌐 Sample Webpage with HTML, CSS, JavaScript and Fetch**

<!-- index.html -->

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8" />

<meta name="viewport" content="width=device-width, initial-scale=1.0"/>

<title>Fetch API Example</title>

<link rel="stylesheet" href="styles.css" />

</head>

<body>

<h1>User List</h1>

<ul id="userList"></ul>

<script src="script.js"></script>

</body>

</html>

css

CopyEdit

/\* styles.css \*/

body {

font-family: Arial, sans-serif;

padding: 20px;

}

h1 {

color: #0077cc;

}

ul {

list-style: none;

padding: 0;

}

li {

background: #f0f0f0;

margin: 8px 0;

padding: 10px;

border-radius: 5px;

}

javascript

CopyEdit

// script.js

fetch('https://jsonplaceholder.typicode.com/users')

.then(response => response.json())

.then(users => {

const userList = document.getElementById('userList');

users.forEach(user => {

const li = document.createElement('li');

li.textContent = `${user.name} (${user.email})`;

userList.appendChild(li);

});

})

.catch(err => console.error('Failed to load users:', err));

Absolutely! Let's explore more **real-world examples of JavaScript fetch** with:

* URL parameters (query string)
* POST, PUT, DELETE methods
* Handling **different HTTP status codes**

**✅ 1. GET Request with Query Parameters**

**🎯 Use Case: Search users by name from an API**

const name = 'john';

fetch(`https://api.example.com/users?name=${encodeURIComponent(name)}`)

.then(response => {

if (!response.ok) throw new Error(`Status: ${response.status}`);

return response.json();

})

.then(data => console.log('User data:', data))

.catch(err => console.error('Error:', err));

**🧠 Key Notes:**

* encodeURIComponent() ensures special characters don't break the URL.
* Good for search, filters, pagination, etc.

**✅ 2. POST Request to Submit Form Data**

**🎯 Use Case: User submits contact form**

const formData = {

name: 'Alice',

email: 'alice@example.com',

message: 'Hello, I need help!'

};

fetch('https://api.example.com/contact', {

method: 'POST',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(formData)

})

.then(response => {

if (response.status === 201) {

console.log('Message submitted successfully!');

} else {

console.warn('Submission failed:', response.status);

}

})

.catch(error => console.error('Network error:', error));

**🧠 Key Notes:**

* Use Content-Type: application/json
* 201 = Created successfully

**✅ 3. PUT Request to Update User Profile**

**🎯 Use Case: User updates their profile info**

const updatedUser = {

id: 123,

name: 'Alice Updated',

bio: 'Full Stack Dev'

};

fetch(`https://api.example.com/users/${updatedUser.id}`, {

method: 'PUT',

headers: {

'Content-Type': 'application/json'

},

body: JSON.stringify(updatedUser)

})

.then(response => {

if (response.ok) {

return response.json();

} else {

throw new Error(`Update failed with status ${response.status}`);

}

})

.then(data => console.log('User updated:', data))

.catch(err => console.error('Error:', err));

**🧠 Key Notes:**

* PUT = full update (use PATCH for partial)
* 200 OK or 204 No Content expected

**✅ 4. DELETE Request to Remove a Resource**

**🎯 Use Case: Admin deletes a user account**

const userId = 123;

fetch(`https://api.example.com/users/${userId}`, {

method: 'DELETE'

})

.then(response => {

if (response.status === 204) {

console.log('User deleted successfully');

} else {

console.warn('Failed to delete user:', response.status);

}

})

.catch(err => console.error('Delete error:', err));

**🧠 Key Notes:**

* 204 = success, no content returned
* Check response.status to handle server response

**✅ 5. Handling Different HTTP Status Codes**

fetch('https://api.example.com/data')

.then(res => {

switch (res.status) {

case 200:

return res.json(); // success

case 404:

throw new Error('Not Found (404)');

case 500:

throw new Error('Server Error (500)');

default:

throw new Error(`Unexpected status: ${res.status}`);

}

})

.then(data => console.log('Data received:', data))

.catch(error => console.error('Request failed:', error.message));

**🧠 Key Notes:**

* Handle different codes for a better UX
* Add custom error messages based on res.status

**🚀 Real APIs You Can Practice With**

| **API** | **URL** |
| --- | --- |
| JSONPlaceholder (Mock) | https://jsonplaceholder.typicode.com |
| OpenWeatherMap (Weather) | https://openweathermap.org/api |
| Reqres (Mock User API) | <https://reqres.in> |
| GitHub API | <https://api.github.com> |

**✅ Basic Understanding**

1. **What is the purpose of the fetch() method in JavaScript?**

a) To create HTML elements  
b) To parse JSON data  
c) To make HTTP requests  
d) To handle browser events

✅ **Answer:** c) To make HTTP requests  
📘 **Explanation:** fetch() is used to perform HTTP requests to servers.

1. **Which value does the fetch() method return?**

a) Object  
b) JSON  
c) Promise  
d) Array

✅ **Answer:** c) Promise  
📘 **Explanation:** fetch() returns a **Promise** that resolves to the Response object.

1. **Which of the following is the correct syntax for using fetch()?**

a) fetch.get("url")  
b) fetch("url")  
c) fetch => "url"  
d) fetch.url("get")

✅ **Answer:** b) fetch("url")  
📘 **Explanation:** The correct syntax is fetch(url).

1. **Which method is used to extract JSON from the fetch response?**

a) response.toJSON()  
b) response.parseJSON()  
c) response.text()  
d) response.json()

✅ **Answer:** d) response.json()  
📘 **Explanation:** The json() method is used to parse the response as JSON.

1. **Which HTTP method is used by default in fetch()?**

a) POST  
b) GET  
c) PUT  
d) DELETE

✅ **Answer:** b) GET  
📘 **Explanation:** By default, fetch() uses the GET method.

**✅ Error Handling and Status**

1. **How do you handle network errors in fetch?**

a) Using .json()  
b) Using .catch()  
c) Using .status()  
d) Using .finally()

✅ **Answer:** b) Using .catch()  
📘 **Explanation:** .catch() is used to handle **network errors** or rejections.

1. **Which property of the Response object contains the HTTP status code?**

a) code  
b) statusCode  
c) httpStatus  
d) status

✅ **Answer:** d) status  
📘 **Explanation:** The status property contains the HTTP status code like 200, 404, etc.

1. **Which response status code indicates success?**

a) 100  
b) 200  
c) 300  
d) 400

✅ **Answer:** b) 200  
📘 **Explanation:** HTTP 200 means OK — the request succeeded.

1. **How can you manually reject a fetch promise for error status codes like 404?**

a) Check response.ok  
b) Use response.error()  
c) Use throw new Error()  
d) Both a and c

✅ **Answer:** d) Both a and c  
📘 **Explanation:** Use if (!response.ok) throw new Error(...) to reject based on status.

**✅ Advanced Usage**

1. **What does the mode option in fetch configure?**

a) Request caching  
b) Cross-origin request mode  
c) Timeout duration  
d) Response parsing

✅ **Answer:** b) Cross-origin request mode  
📘 **Explanation:** mode controls how CORS is handled: cors, no-cors, or same-origin.

1. **How do you send data in the body of a POST request using fetch?**

a) fetch("url", { method: "POST", body: data })  
b) fetch.post("url", data)  
c) fetch("url").send(data)  
d) fetch("url", { data: body })

✅ **Answer:** a) fetch("url", { method: "POST", body: data })  
📘 **Explanation:** Use the body field in the fetch options for POST requests.

1. **Which header must be set to send JSON data with fetch?**

a) 'Content-Encoding': 'json'  
b) 'Accept': 'application/xml'  
c) 'Content-Type': 'application/json'  
d) 'Authorization': 'json'

✅ **Answer:** c) 'Content-Type': 'application/json'  
📘 **Explanation:** This header tells the server the content is JSON.

1. **Which of the following will NOT trigger the .catch() block in fetch?**

a) Network failure  
b) DNS resolution failure  
c) 500 Internal Server Error  
d) No internet connection

✅ **Answer:** c) 500 Internal Server Error  
📘 **Explanation:** HTTP error status (like 500) does **not** cause .catch() unless thrown manually.

1. **Which method is used to read the response as plain text?**

a) response.toText()  
b) response.read()  
c) response.text()  
d) response.string()

✅ **Answer:** c) response.text()  
📘 **Explanation:** Use response.text() to read the body as plain text.

1. **Which option enables credentials like cookies to be sent in a cross-origin fetch request?**

a) credentials: "omit"  
b) credentials: "same-origin"  
c) credentials: "include"  
d) cookies: true

✅ **Answer:** c) credentials: "include"  
📘 **Explanation:** Use credentials: "include" to send cookies/auth headers with cross-origin requests.

**Async programming quiz:**

1. **What does a Promise represent in JavaScript?**  
   a) A loop  
   b) An event  
   c) An asynchronous operation  
   d) A function

✅ **Answer:** c) An asynchronous operation  
📘 **Explanation:** A Promise represents the eventual completion (or failure) of an asynchronous operation.

1. **Which of the following is NOT a valid state of a Promise?**  
   a) pending  
   b) fulfilled  
   c) settled  
   d) rejected

✅ **Answer:** c) settled  
📘 **Explanation:** settled is not a formal state — it's a term describing a promise that's either fulfilled or rejected.

1. **What does Promise.resolve(value) do?**  
   a) Rejects the promise  
   b) Creates a pending promise  
   c) Returns a fulfilled promise  
   d) Pauses execution

✅ **Answer:** c) Returns a fulfilled promise  
📘 **Explanation:** It creates a Promise that is resolved with the given value.

1. **What is the purpose of then() in a Promise?**  
   a) Handle errors  
   b) Cancel the promise  
   c) Chain synchronous functions  
   d) Handle resolved values

✅ **Answer:** d) Handle resolved values  
📘 **Explanation:** .then() is used to handle the result when a Promise is fulfilled.

1. **Which method is used to catch errors in a Promise?**  
   a) .error()  
   b) .fail()  
   c) .then()  
   d) .catch()

✅ **Answer:** d) .catch()  
📘 **Explanation:** .catch() is used to handle rejected Promises.

**✅ Async / Await Basics**

1. **What does the async keyword do?**  
   a) Runs code in parallel  
   b) Pauses code execution  
   c) Automatically returns a Promise  
   d) Makes a function synchronous

✅ **Answer:** c) Automatically returns a Promise  
📘 **Explanation:** Any async function returns a Promise implicitly.

1. **What is the purpose of await?**  
   a) Rejects a Promise  
   b) Cancels a Promise  
   c) Waits for a Promise to settle  
   d) Converts a function to async

✅ **Answer:** c) Waits for a Promise to settle  
📘 **Explanation:** await pauses the execution of an async function until the Promise settles.

1. **Which of the following is valid syntax with await?**  
   a) await 10  
   b) await "text"  
   c) await Promise.resolve(5)  
   d) All of the above

✅ **Answer:** d) All of the above  
📘 **Explanation:** await works on any value; non-Promise values are auto-wrapped as resolved Promises.

1. **Can await be used outside an async function?**  
   a) Yes  
   b) No  
   c) Only in Node.js  
   d) Only for resolved Promises

✅ **Answer:** b) No  
📘 **Explanation:** You cannot use await at the top level unless the environment supports top-level await.

1. **What happens when you await a rejected Promise without a try...catch?**  
   a) The function continues  
   b) The rejection is ignored  
   c) An error is thrown  
   d) It becomes undefined

✅ **Answer:** c) An error is thrown  
📘 **Explanation:** Rejections must be handled with try/catch or .catch().

**✅ Advanced Concepts**

1. **Which of the following is TRUE about Promises?**  
   a) They can be used synchronously  
   b) They are cancellable  
   c) They handle asynchronous operations  
   d) They block the event loop

✅ **Answer:** c) They handle asynchronous operations  
📘 **Explanation:** Promises provide a clean way to handle async logic.

1. **How do you run multiple Promises in parallel and wait for all of them?**  
   a) Promise.race()  
   b) Promise.chain()  
   c) Promise.join()  
   d) Promise.all()

✅ **Answer:** d) Promise.all()  
📘 **Explanation:** Promise.all() waits for all Promises to resolve.

1. **Which method returns the result of the first settled Promise (resolved or rejected)?**  
   a) Promise.first()  
   b) Promise.race()  
   c) Promise.any()  
   d) Promise.resolve()

✅ **Answer:** b) Promise.race()  
📘 **Explanation:** Promise.race() returns the result of the first settled (fulfilled or rejected) Promise.

1. **Which method returns the first successfully resolved Promise, ignoring rejections?**  
   a) Promise.all()  
   b) Promise.any()  
   c) Promise.race()  
   d) Promise.resolve()

✅ **Answer:** b) Promise.any()  
📘 **Explanation:** Promise.any() returns the first **fulfilled** Promise and ignores rejections.

1. **What does await return?**  
   a) Always a Promise  
   b) The resolved value of a Promise  
   c) The rejected reason  
   d) Nothing

✅ **Answer:** b) The resolved value of a Promise  
📘 **Explanation:** await extracts and returns the resolved value.

**✅ Practical & Interview-Focused**

1. **What will this code output?**

async function test() {

return 42;

}

console.log(test());

a) 42  
b) undefined  
c) Promise { 42 }  
d) Error

✅ **Answer:** c) Promise { 42 }  
📘 **Explanation:** An async function always returns a Promise.

1. **Which keyword can pause code execution in async function until Promise resolves?**  
   a) yield  
   b) pause  
   c) await  
   d) stop

✅ **Answer:** c) await  
📘 **Explanation:** await pauses execution until the Promise resolves.

1. **What is a common mistake when chaining Promises?**  
   a) Not returning a Promise inside .then()  
   b) Using await inside .then()  
   c) Calling catch() before then()  
   d) Wrapping await in a setTimeout

✅ **Answer:** a) Not returning a Promise inside .then()  
📘 **Explanation:** If you forget to return a Promise, the chain won't wait properly.

1. **How can you make an async function throw a custom error?**  
   a) async throw Error("...")  
   b) return Error(...)  
   c) await throw new Error(...)  
   d) throw new Error("...")

✅ **Answer:** d) throw new Error("...")  
📘 **Explanation:** You can throw errors in async functions like normal functions.

1. **Which of these is BEST for sequentially awaiting multiple asynchronous calls?**  
   a) Promise.all()  
   b) Multiple await calls one after another  
   c) setTimeout  
   d) await Promise.race()

✅ **Answer:** b) Multiple await calls one after another  
📘 **Explanation:** Sequential operations should be awaited one by one to preserve order.

**✅ Event Loop – Interview MCQs**

1. **What is the main responsibility of the Event Loop in JavaScript?**

a) Executing all functions in parallel  
b) Managing variable declarations  
c) Handling asynchronous operations and executing callback queues  
d) Running JavaScript code faster

✅ **Answer:** c) Handling asynchronous operations and executing callback queues  
📘 **Explanation:** The Event Loop constantly checks the call stack and the callback/task queue to process asynchronous code.

1. **Which of the following is NOT part of the JavaScript concurrency model?**

a) Call Stack  
b) Web APIs  
c) Task Queue (Callback Queue)  
d) Garbage Collector

✅ **Answer:** d) Garbage Collector  
📘 **Explanation:** The event loop involves the Call Stack, Web APIs, and Task Queue, not the Garbage Collector.

1. **What happens when the Call Stack is empty and the Task Queue has callbacks?**

a) The browser crashes  
b) The Event Loop pushes the first task to the Call Stack  
c) All tasks execute simultaneously  
d) Tasks are discarded

✅ **Answer:** b) The Event Loop pushes the first task to the Call Stack  
📘 **Explanation:** The Event Loop checks the call stack, and when it's empty, it dequeues a task from the Task Queue.

1. **Which of the following queues has higher priority in modern JavaScript engines?**

a) Task Queue  
b) Callback Queue  
c) Microtask Queue  
d) Message Queue

✅ **Answer:** c) Microtask Queue  
📘 **Explanation:** Microtasks (e.g., Promise.then) are executed before tasks like setTimeout.

1. **Which of the following is executed first in the Event Loop?**

a) setTimeout(() => console.log("Timeout"), 0)  
b) console.log("Sync")  
c) Promise.resolve().then(() => console.log("Promise"))

✅ **Answer:** b) console.log("Sync")  
📘 **Explanation:** Synchronous code runs first, then microtasks (like Promise callbacks), then macrotasks (like setTimeout).

1. **What is the difference between Macrotasks and Microtasks?**

a) Macrotasks are only used by Promises  
b) Microtasks are executed after all macrotasks  
c) Microtasks are executed before the next macrotask  
d) Microtasks cannot be created manually

✅ **Answer:** c) Microtasks are executed before the next macrotask  
📘 **Explanation:** After each task, the Event Loop drains the microtask queue before continuing.

1. **What is the result of the following code?**

console.log("Start");

setTimeout(() => console.log("Timeout"), 0);

Promise.resolve().then(() => console.log("Promise"));

console.log("End");

a) Start → Timeout → Promise → End  
b) Start → End → Promise → Timeout  
c) Start → Promise → Timeout → End  
d) Start → End → Timeout → Promise

✅ **Answer:** b) Start → End → Promise → Timeout  
📘 **Explanation:** Synchronous code runs first (Start, End), then microtasks (Promise), then macrotasks (Timeout).

1. **What is the role of the Web APIs environment in the Event Loop?**

a) To execute synchronous code  
b) To block other async tasks  
c) To register and handle async operations like timers, fetch, and events  
d) To optimize code

✅ **Answer:** c) To register and handle async operations like timers, fetch, and events  
📘 **Explanation:** Web APIs handle async tasks like setTimeout, fetch, etc., and pass callbacks to the Task Queue.

1. **Which of the following uses the Microtask queue in JavaScript?**

a) setInterval  
b) setTimeout  
c) Promise.then()  
d) addEventListener

✅ **Answer:** c) Promise.then()  
📘 **Explanation:** Promises and queueMicrotask() use the microtask queue.

1. **What will be the order of output?**

setTimeout(() => console.log("timeout"), 0);

Promise.resolve().then(() => console.log("promise"));

console.log("sync");

a) timeout → promise → sync  
b) sync → promise → timeout  
c) sync → timeout → promise  
d) promise → sync → timeout

✅ **Answer:** b) sync → promise → timeout  
📘 **Explanation:** Sync code (console.log("sync")) executes first, followed by the microtask (promise), then the macrotask (timeout).

**🧩 1. What is Deployment in Web Development?**

**Deployment** is the process of making your web application or website **accessible on the internet**. After you've written your HTML, CSS, JavaScript, etc., you deploy it so that **users can view and interact with it online**.

**❓ 2. Why Do We Need Deployment Platforms?**

* So others can **see your app** in action
* To **host portfolios, business sites, APIs, or static sites**
* For **version control**, **CI/CD**, and **easy domain mapping**
* To showcase **JavaScript apps** (e.g., weather app, to-do list, APIs)

**🚀 3. Deployment Platforms Explained**

**🔹 GitHub Pages**

* **Type**: Static site hosting (HTML, CSS, JS only)
* **Hosted by**: GitHub
* **Ideal for**: Simple sites, documentation, portfolios, or JS apps that don’t need a backend

**✅ Pros:**

* Free with GitHub
* Directly deploy from repo
* Easy to use for beginners

**❌ Limitations:**

* No server-side code (e.g., Node.js, Python)
* Not for dynamic data without APIs

**🛠 How to Deploy:**

1. Create a GitHub repo.
2. Push your website code (index.html, etc.) to it.
3. Go to **Settings → Pages**.
4. Choose a branch (usually main) and root directory.
5. GitHub will host your site at:  
   https://<username>.github.io/<repo-name>/

**🔹 Netlify**

* **Type**: Static site and serverless functions
* **Hosted by**: Netlify
* **Ideal for**: Frontend frameworks (React, Vue, Svelte) and static JS projects

**✅ Pros:**

* Drag & drop OR connect GitHub
* Auto-deploy on git push
* Supports custom domains, HTTPS, forms, and lambda functions

**❌ Limitations:**

* Basic backend support only via functions
* Free tier has bandwidth/storage limits

**🛠 How to Deploy:**

1. Sign in to [Netlify](https://netlify.com)
2. Connect your GitHub repository or upload folder
3. Set build command if using framework (e.g., npm run build)
4. Deploy and access your live URL

**🔹 Render**

* **Type**: Full-stack (static + backend)
* **Hosted by**: Render.com
* **Ideal for**: Static websites, APIs, full backend (Node.js, Flask, Django)

**✅ Pros:**

* Deploy frontend + backend from GitHub
* Free static hosting, easy backend deployment
* Great for full apps (e.g., Node.js + MongoDB)

**❌ Limitations:**

* Slight learning curve for beginners
* Cold start time for free plans

**🛠 How to Deploy:**

1. Create a free account on [Render](https://render.com)
2. Choose “Static Site” or “Web Service” depending on app type
3. Connect to GitHub and select your repo
4. Set build command (npm run build) and publish directory (dist, build, etc.)
5. Render will give you a live site URL

**🕰 4. When and Where to Use Each**

| **Platform** | **Best For** | **When to Use It** |
| --- | --- | --- |
| GitHub Pages | Static HTML/CSS/JS, docs, portfolios | When you want a quick, free deployment for static websites |
| Netlify | React, Vue apps, small dynamic forms | When you need CI/CD, custom domain, and serverless functions |
| Render | Full-stack apps (frontend + backend) | When you need a backend (Node.js, Django, Flask) + frontend |

**🌍 5. Real-World Examples**

**🌐 GitHub Pages Example**

You create a **JavaScript Portfolio** using HTML + CSS + JS and deploy it via GitHub Pages to share on your resume.

bash

CopyEdit

project/

│

├── index.html

├── style.css

└── script.js

* Push to GitHub
* Enable GitHub Pages
* Done! Publicly available site

**🌐 Netlify Example**

You build a **React-based To-Do App** with create-react-app.

* Run: npm run build
* Drag & drop build/ folder to Netlify
* Or connect GitHub repo for CI/CD
* Live in seconds, with custom domain

**🌐 Render Example**

You build a **weather app** using:

* Frontend: HTML/CSS/JS
* Backend: Node.js + Express
* API: OpenWeatherMap

You deploy:

* Frontend as static site
* Backend as web service (Node.js app)

**🔍 6. Comparison Table**

| **Feature** | **GitHub Pages** | **Netlify** | **Render** |
| --- | --- | --- | --- |
| Static Sites | ✅ | ✅ | ✅ |
| Custom Domain | ✅ | ✅ | ✅ |
| Auto Deploy (Git) | ✅ | ✅ | ✅ |
| Serverless Functions | ❌ | ✅ (limited) | ✅ (full backend) |
| Backend Hosting | ❌ | ❌ | ✅ (Node.js, Flask etc) |
| CI/CD Support | ✅ | ✅ | ✅ |
| Use Case | Docs, portfolios | SPAs, frontend apps | Full-stack web apps |

**📌 Summary**

* **GitHub Pages** = best for pure front-end or documentation
* **Netlify** = great for modern frontend projects (React/Vue) with basic backend logic
* **Render** = full-featured for frontend + backend hosting

**✅ New Project: Public GitHub User Profile Viewer**

**🔍 Features:**

* Enter a GitHub username
* Fetch public user info (avatar, name, followers, location, etc.)
* Uses fetch() and **no API key required**

<!DOCTYPE html>

<html>

<head>

<title>GitHub Profile Viewer</title>

<style>

body { font-family: Arial, sans-serif; background: #f4f4f4; padding: 20px; }

input, button { padding: 10px; font-size: 16px; margin-right: 10px; }

#profile { margin-top: 20px; padding: 20px; background: #fff; border-radius: 8px; max-width: 400px; }

img { width: 100px; border-radius: 50%; }

</style>

</head>

<body>

<h2>👤 GitHub Profile Viewer</h2>

<input type="text" id="username" placeholder="Enter GitHub username">

<button onclick="getProfile()">Get Profile</button>

<div id="profile"></div>

<script>

async function getProfile() {

const username = document.getElementById('username').value.trim();

const profile = document.getElementById('profile');

profile.innerHTML = 'Loading...';

if (!username) {

profile.innerHTML = '❌ Please enter a username.';

return;

}

try {

const res = await fetch(`https://api.github.com/users/${username}`);

if (!res.ok) throw new Error("User not found");

const data = await res.json();

profile.innerHTML = `

<img src="${data.avatar\_url}" alt="Avatar"><br><br>

<strong>${data.name || data.login}</strong><br>

📍 ${data.location || 'Unknown'}<br>

👥 Followers: ${data.followers}<br>

📦 Public Repos: ${data.public\_repos}<br>

🔗 <a href="${data.html\_url}" target="\_blank">Visit Profile</a>

`;

} catch (e) {

profile.innerHTML = `❌ Error: ${e.message}`;

}

}

</script>

</body>

</html>

**🔹 2. Blog Reader App**

**📰 Uses JSONPlaceholder Fake REST API**

<!DOCTYPE html>

<html>

<head>

<title>Blog Reader</title>

<style>

body { font-family: sans-serif; padding: 20px; }

.post { border: 1px solid #ccc; margin: 10px 0; padding: 10px; border-radius: 6px; }

.delete { color: red; cursor: pointer; }

</style>

</head>

<body>

<h2>📝 Blog Reader</h2>

<div id="posts">Loading posts...</div>

<script>

async function loadPosts() {

const res = await fetch('https://jsonplaceholder.typicode.com/posts?\_limit=5');

const posts = await res.json();

const container = document.getElementById('posts');

container.innerHTML = '';

posts.forEach(post => {

const div = document.createElement('div');

div.className = 'post';

div.innerHTML = `

<h4>${post.title}</h4>

<p>${post.body}</p>

<span class="delete" onclick="deletePost(${post.id}, this)">🗑 Delete</span>

`;

container.appendChild(div);

});

}

async function deletePost(id, el) {

await fetch(`https://jsonplaceholder.typicode.com/posts/${id}`, { method: 'DELETE' });

el.closest('.post').remove();

}

loadPosts();

</script>

</body>

</html>

**Real-Time Crypto Price Tracker (No API Key Needed)**

<!DOCTYPE html>

<html>

<head>

<title>Crypto Tracker</title>

<style>

body { font-family: Arial, sans-serif; background: #111; color: #eee; padding: 20px; }

input, button { padding: 10px; font-size: 16px; margin-right: 10px; }

#output { margin-top: 20px; background: #222; padding: 20px; border-radius: 8px; }

table { width: 100%; border-collapse: collapse; margin-top: 10px; }

th, td { padding: 10px; border: 1px solid #444; }

</style>

</head>

<body>

<h2>💰 Live Crypto Price Tracker</h2>

<input type="text" id="symbol" placeholder="e.g. bitcoin, ethereum, dogecoin" />

<button onclick="getCrypto()">Get Price</button>

<div id="output"></div>

<script>

async function getCrypto() {

const symbol = document.getElementById("symbol").value.toLowerCase().trim();

const output = document.getElementById("output");

if (!symbol) {

output.innerHTML = "❌ Please enter a crypto name (e.g., bitcoin)";

return;

}

output.innerHTML = "Loading...";

try {

const res = await fetch(`https://api.coingecko.com/api/v3/simple/price?ids=${symbol}&vs\_currencies=usd,inr`);

const data = await res.json();

if (!data[symbol]) throw new Error("Invalid symbol or not supported");

output.innerHTML = `

<h3>📊 ${symbol.toUpperCase()} Price</h3>

<table>

<tr><th>Currency</th><th>Price</th></tr>

<tr><td>USD</td><td>$${data[symbol].usd}</td></tr>

<tr><td>INR</td><td>₹${data[symbol].inr}</td></tr>

</table>

`;

} catch (err) {

output.innerHTML = `❌ Error: ${err.message}`;

}

}

</script>

</body>

</html>

**🧪 Try These Cryptos:**

* bitcoin
* ethereum
* dogecoin
* ripple
* solana