# **Introduction to Asynchronous Programming**

**What is it?**

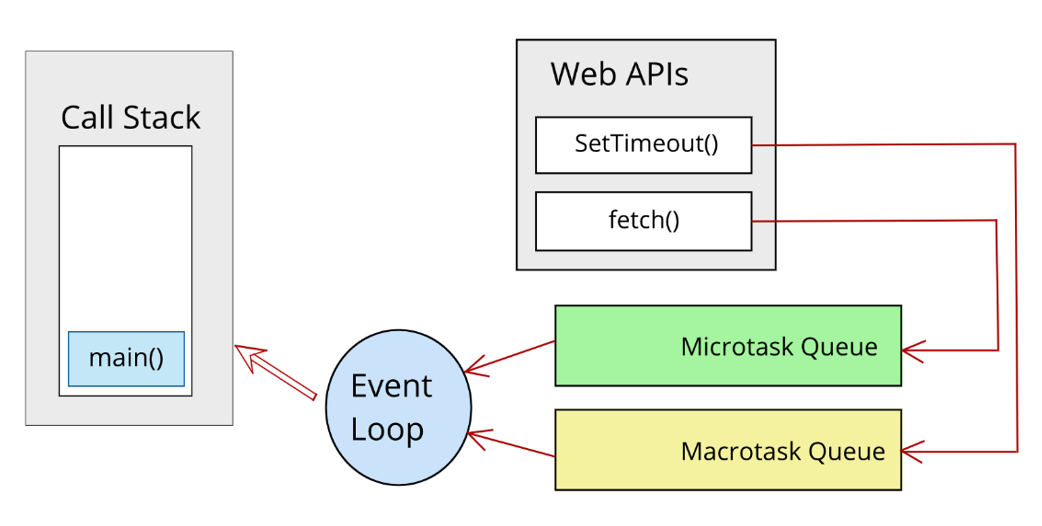
Asynchronous programming lets you start a potentially slow operation (e.g. network request, disk I/O, timer) and move on—without blocking the single JavaScript thread—so your app remains responsive.

**Why did it come about?**

* **Single‑threaded nature of JS:** Browsers and Node.js run JS on one thread. If you did heavy I/O synchronously, the UI would freeze or the server could not handle other requests.
* **Need for responsiveness:** Users expect smooth UIs and servers expect to handle many concurrent connections.

**How it works under the hood: the Event Loop**

1. **Call Stack:** where JS executes functions.
2. **Web/API or Node API:** where async operations are off‑loaded.
3. **Callback (Task) Queue:** completed async callbacks wait here.
4. **Event Loop:** when the Call Stack is empty, it pushes the next callback from the queue.

****

**Rules & Best Practices**

* **Never block the main thread.** Always off‑load heavy or I/O tasks.
* **Keep callbacks/Promises short.** Delegate complex logic to separate functions.
* **Handle errors!** Always provide error‐handling paths (callbacks: check error arg; Promises: .catch).
* **Clean up timers/listeners.** Avoid memory leaks.

**Where & When to Use**

* **Network Requests:** fetch‑ing APIs in the browser or Node.
* **File I/O:** reading/writing files in Node.js.
* **Timers:** setTimeout, setInterval.
* **Event Handling:** user clicks, keyboard events, etc.

**2. Callbacks**

**Overview**

A *callback* is simply a function you pass into another function to be invoked later—once an async operation completes.

// Function to simulate placing a food order

function placeOrder(order, callback) {

console.log(`Order placed for: ${order}`);

// Simulate some delay using setTimeout (e.g., food preparation time)

setTimeout(() => {

console.log(`Preparing your ${order}...`);

// Once preparation is done, trigger the callback

callback(order);

}, 2000); // 2 seconds delay

}

// Callback function to deliver food

function deliverOrder(order) {

console.log(`🚚 Order ready! Delivering your ${order} now.`);

}

// Call the function with a callback

placeOrder("Veg Biryani", deliverOrder);

**Rules**

1. **Error‑first signature:** (err, result) ⇒ { … }.
2. **Invoke exactly once.**
3. **Keep them short** (or delegate to named functions) to avoid deeply nested code.

**Why Callbacks?**

* **Legacy:** Introduced in early JS/Node for non‑blocking I/O and event handling.
* **Simplicity:** Very minimal API surface—just functions.

**Drawbacks**

* **“Callback Hell”:** deeply nested callbacks become hard to read and maintain.
* **Inversion of Control:** caller cedes control to the callback callee.
* **Error handling is clunky:** must check err at every level.

**Real‑World Example: XMLHttpRequest with Callbacks**

<button id="btn">Load Data</button>

<script>

document.getElementById('btn').addEventListener('click', () => {

const xhr = new XMLHttpRequest();

xhr.open('GET', 'https://api.example.com/data');

xhr.onload = () => {

if (xhr.status === 200) {

console.log('Data:', JSON.parse(xhr.responseText));

} else {

console.error('Error fetching:', xhr.statusText);

}

};

xhr.onerror = () => console.error('Network error');

xhr.send();

});

</script>

**3. Promises**

**Overview**

A **Promise** represents a value that may be available now, later, or never. It lets you attach callbacks via .then() and .catch() **without** nesting.

function orderIceCream(flavor) {

return new Promise((resolve, reject) => {

console.log(`📦 Order received for ${flavor} ice cream.`);

setTimeout(() => {

const inStock = Math.random() > 0.3; // 70% chance it's in stock

if (inStock) {

resolve(`✅ Your ${flavor} ice cream is ready! 🍨`);

} else {

reject(`❌ Sorry, ${flavor} is out of stock.`);

}

}, 2000); // 2-second delay to simulate processing

});

}

// Usage of the Promise

orderIceCream("Chocolate")

.then((message) => {

console.log(message);

})

.catch((error) => {

console.error(error);

});

**States**

1. **Pending:** initial state, neither fulfilled nor rejected.
2. **Fulfilled:** completed successfully.
3. **Rejected:** failed with an error.

Once settled, a Promise cannot change state.

**Rules & Best Practices**

* **Always return** Promises in functions so you can chain.
* **Chain instead of nest:** each .then() returns a new Promise.
* **Use .catch() at the end** to handle any upstream errors.
* **Avoid mixing callbacks** and Promises in the same API.

**Why Promises?**

* **Flatten your code:** no deep nesting.
* **Better error propagation:** a single .catch() handles errors anywhere in the chain.
* **Composability:** methods like Promise.all, Promise.race let you coordinate multiple async ops.

**Where & When**

* **Modern HTTP requests:** the Fetch API returns Promises.
* **Node.js APIs:** new Promise‐based fs.promises methods.
* **Any custom async logic** you want to expose in a clean, chainable way.

**Real‑World Example: Fetching Data with Promises**

<!DOCTYPE html>

<html>

<head>

  <title>Fetch API with Promises</title>

</head>

<body>

  <h2>User List</h2>

  <ul id="userList">Loading...</ul>

  <script>

    function fetchUsers() {

      // Using Fetch API which returns a Promise

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

        .then(response => {

          if (!response.ok) {

            throw new Error('Network response was not OK');

          }

          return response.json(); // parse JSON from the response

        })

        .then(users => {

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

          list.innerHTML = ''; // Clear "Loading..." text

          users.forEach(user => {

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

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

            list.appendChild(listItem);

          });

        })

        .catch(error => {

          document.getElementById('userList').innerHTML = `❌ Failed to load users: ${error.message}`;

        });

    }

    // Trigger the function

    fetchUsers();

  </script>

</body>

</html>

**Coordinating Multiple Requests**

<!DOCTYPE html>

<html>

<head>

  <title>Promise.all Example</title>

</head>

<body>

  <h2>📋 Data from Multiple API Calls</h2>

  <h3>Users:</h3>

  <ul id="userList">Loading users...</ul>

  <h3>Posts:</h3>

  <ul id="postList">Loading posts...</ul>

  <script>

    const usersURL = 'https://jsonplaceholder.typicode.com/users';

    const postsURL = 'https://jsonplaceholder.typicode.com/posts';

    // Function to fetch both users and posts

    function fetchData() {

      const usersPromise = fetch(usersURL).then(res => res.json());

      const postsPromise = fetch(postsURL).then(res => res.json());

      // Wait for both Promises to complete

      Promise.all([usersPromise, postsPromise])

        .then(([users, posts]) => {

          // Populate user list

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

          userList.innerHTML = '';

          users.slice(0, 5).forEach(user => {

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

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

            userList.appendChild(li);

          });

          // Populate post list

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

          postList.innerHTML = '';

          posts.slice(0, 5).forEach(post => {

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

            li.textContent = `${post.title}`;

            postList.appendChild(li);

          });

        })

        .catch(error => {

          document.getElementById('userList').textContent = '❌ Failed to load users';

          document.getElementById('postList').textContent = '❌ Failed to load posts';

          console.error('Error fetching data:', error);

        });

    }

    // Call the function

    fetchData();

  </script>

</body>

</html>

**1. Macro‑tasks (Tasks)**

**What are they?**

Macro‑tasks represent higher‑level events like:

* **Timers**: setTimeout, setInterval
* **I/O callbacks**: e.g. network responses, file system events in Node
* **UI rendering**: painting/updating the screen in browsers
* **User interactions**: click, keydown, etc.

Whenever one of these events fires, its callback is placed into the **macro‑task queue**.

**Macro Tasks**: Long-running tasks like setTimeout, setInterval, fetch, and DOM events.

**Execution model**

1. Pull one callback from the macro‑task queue.
2. Execute it to completion (run to the end, including any synchronous work).
3. **Then**, drain **all** pending micro‑tasks (see next section).
4. **Then**, render/update the UI (in browsers).
5. Go back to step 1.

**Real‑world example**

<!DOCTYPE html>

<html>

<body>

<h2>🍲 Fetch Recipe (Macro Task Demo)</h2>

<script>

console.log("🍳 Script Start"); // Synchronous

setTimeout(() => {

console.log("⏲️ setTimeout (Macro Task)");

}, 0);

fetch("https://jsonplaceholder.typicode.com/posts/1")

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

.then(data => {

console.log("📦 fetch then() (Micro Task)");

console.log("✅ Recipe title:", data.title);

});

Promise.resolve().then(() => {

console.log("🧩 Promise.then (Micro Task)");

});

console.log("🧾 Script End"); // Synchronous

</script>

</body>

</html>

🧠 Output Order (Demonstrates Macro + Micro Task Behavior)

🍳 Script Start

🧾 Script End

🧩 Promise.then (Micro Task)

📦 fetch then() (Micro Task)

⏲️ setTimeout (Macro Task)

**🔍 Explanation:**

| **Line** | **Task Type** | **Why?** |
| --- | --- | --- |
| Script Start/End | Synchronous | Runs immediately |
| Promise.then() | Micro Task | Added to microtask queue |
| fetch() then | Micro Task | After HTTP response is received |
| setTimeout() | Macro Task | Scheduled after current tasks finish |

**🔁 Real Use Cases of Macro Tasks with fetch:**

* Fetching data from APIs (weather, recipes, users)
* Chained with UI updates after network response
* Used with loading spinners (start spinner → fetch → stop spinner)
* Event-based workflows (click → fetch → display)

**2. Micro‑tasks (Jobs)**

**What are they?**

Micro‑tasks are smaller, more immediate callbacks that run **after** the currently executing script or macro‑task, but **before** the next macro‑task starts. (like setTimeout, fetch, or DOM events)

Examples include:

* **Promise callbacks**: every .then, .catch, .finally
* **process.nextTick** in Node.js
* **queueMicrotask()**
* **MutationObserver** callbacks in browsers

They’re meant for things that should happen “very soon,” before yielding back to the browser to render or handling the next timer.

**Execution model**

* Whenever a micro‑task is queued, it doesn’t run immediately—instead, it waits until:
  1. The current macro‑task (or script) completes, **then**
  2. All micro‑tasks are run in FIFO order, one after another, until the micro‑task queue is empty, **then**
  3. Control returns to the event loop to pick the next macro‑task.

**🍩 Real-World Scenario: Order Processing with Priority**

Imagine a bakery app:

* You place an order (console.log)
* Your **reward points** are updated (important, so done immediately as a microtask)
* Your receipt is printed later (macrotask, not urgent)

💻 Real-World Code Example: (Run in browser or JSFiddle)

console.log("📦 Order placed");

Promise.resolve().then(() => {

console.log("🎁 Reward points updated (Microtask)");

});

queueMicrotask(() => {

console.log("📈 Loyalty score recalculated (Microtask)");

});

setTimeout(() => {

console.log("🧾 Receipt printed (Macrotask)");

}, 0);

console.log("🧁 Preparing your order...");

**🧾 Output:**

📦 Order placed

🧁 Preparing your order...

🎁 Reward points updated (Microtask)

📈 Loyalty score recalculated (Microtask)

🧾 Receipt printed (Macrotask)

**🔍 Explanation of Task Flow:**

| **Task** | **Type** | **When It Executes** |
| --- | --- | --- |
| console.log("Order placed") | Synchronous | Immediate |
| Promise.then() | Microtask | After sync code, before macrotask |
| queueMicrotask() | Microtask | Same as above |
| setTimeout() | Macrotask | After microtasks finish |

**✅ Real-World Use Cases of Microtasks:**

* Updating UI immediately after data change (e.g., state change in frameworks)
* Validating form fields after input event
* Updating application state (e.g., React re-render triggers)
* Ensuring higher priority tasks finish before slower external ones (e.g., animations, logging)

**3. Putting It Together: Event Loop Tick**

1. **Execute** your script or a macro‑task from the queue.
2. **Drain** the micro‑task queue completely (including micro‑tasks that queue more micro‑tasks).
3. **Render/UI updates** (in browsers).
4. **Pick next** macro‑task and repeat.

**Summary**

| **Aspect** | **Callbacks** | **Promises** |
| --- | --- | --- |
| Syntax | fn(arg1, arg2, callback) | fn(...).then(...).catch(...) |
| Error handling | Manual if (err) checks at each level | Single .catch() for whole chain |
| Control flow | Nested, pyramid of doom | Flat, chainable |
| Composability | Difficult | Promise.all, Promise.race, etc. |

**1. Async & Await**

**What & Why**

* **What:** Syntactic sugar over Promises that lets you write asynchronous code in a sequential, “top‑to‑bottom” style.
* **Why:** Chaining .then() can get hard to read and reason about—async/await flattens the flow, making error handling and control flow simpler.

**Rules & Best Practices**

1. **async functions always return a Promise.**

async function foo() { return 10; }

foo().then(x => console.log(x)); // 10

1. **await only works inside async functions** (or modern module top‑level).
2. **await pauses** the function until the Promise settles (fulfilled or rejected).
3. **Parallelize independent tasks** by starting Promises before awaiting:

// ❌ sequential (slow)

const a = await fetchA();

const b = await fetchB();

// ✅ concurrent (faster)

const pA = fetchA();

const pB = fetchB();

const [a, b] = await Promise.all([pA, pB]);

1. **Always handle errors** with try…catch (or let the caller .catch it).

**When & Where**

* **Network calls** (fetching from REST endpoints).
* **Delays** (wrapping setTimeout in a Promise).
* **IndexedDB** or other browser APIs that return Promises.

🌍 Real-World Examples of async/await

1. 🔍 **Fetching User Data from an API**

<!DOCTYPE html>

<html>

<body>

<h3>User Info</h3>

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

<script>

async function fetchUser() {

try {

const response = await fetch('https://jsonplaceholder.typicode.com/users/1');

const user = await response.json();

document.getElementById('user').textContent = `${user.name} (${user.email})`;

} catch (err) {

document.getElementById('user').textContent = '❌ Failed to load user';

}

}

fetchUser();

</script>

</body>

</html>

2. 🛒 **Placing an Order in a Shopping Cart System**

async function placeOrder(itemId) {

try {

const inventoryCheck = await checkInventory(itemId);

const paymentStatus = await processPayment(inventoryCheck);

const confirmation = await confirmOrder(paymentStatus);

console.log("✅ Order Confirmed:", confirmation);

} catch (error) {

console.error("❌ Order failed:", error.message);

}

}

// Dummy async functions

function checkInventory(itemId) {

return new Promise((resolve) => {

setTimeout(() => resolve({ itemId, available: true }), 1000);

});

}

function processPayment(inventory) {

return new Promise((resolve) => {

setTimeout(() => resolve({ paymentId: "PAY123" }), 1000);

});

}

function confirmOrder(payment) {

return new Promise((resolve) => {

setTimeout(() => resolve({ orderId: "ORD456" }), 1000);

});

}

placeOrder(101);

3. 💾 **Uploading a File and Getting a Download Link**

async function uploadFile(file) {

try {

const uploadResult = await simulateUpload(file);

const downloadLink = await getDownloadLink(uploadResult.fileId);

console.log("📎 Download Link:", downloadLink);

} catch (error) {

console.error("❌ Upload failed:", error.message);

}

}

// Simulate async functions

function simulateUpload(file) {

return new Promise((resolve) => {

setTimeout(() => resolve({ fileId: "abc123" }), 1500);

});

}

function getDownloadLink(fileId) {

return new Promise((resolve) => {

setTimeout(() => resolve(`https://example.com/download/${fileId}`), 1000);

});

}

uploadFile("resume.pdf");

4. 📊 **Chained API Calls (e.g., User → Posts)**

<!DOCTYPE html>

<html>

<body>

<h2>🔗 User & Posts</h2>

<pre id="output">Loading...</pre>

<script>

async function fetchUserPosts() {

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

try {

const userRes = await fetch("https://jsonplaceholder.typicode.com/users/1");

const user = await userRes.json();

const postsRes = await fetch(`https://jsonplaceholder.typicode.com/posts?userId=${user.id}`);

const posts = await postsRes.json();

let result = `🧑 User: ${user.name}\n\n📝 Posts:\n`;

posts.slice(0, 3).forEach(post => {

result += `- ${post.title}\n`;

});

output.textContent = result;

} catch (err) {

output.textContent = `❌ Error fetching data: ${err.message}`;

}

}

fetchUserPosts();

</script>

</body>

</html>

**2. Error Handling in JavaScript**

**What & Why**

* **What:** Mechanisms to catch runtime exceptions and rejected Promises so your app doesn’t crash or stay in a broken state.
* **Why:** Unhandled errors either bubble up to the console/user or break subsequent code paths; proper handling makes apps robust.

**Mechanisms & Rules**

| **Mechanism** | **Use Case** | **Key Rules** |
| --- | --- | --- |
| **try…catch…finally** | Synchronous code or await calls | Keep try blocks small; clean up in finally |
| **.catch()** | Promise chains | Always end a chain with .catch() |
| **Global handlers** | Last‑resort crash reporting | window.onerror / window.onunhandledrejection |

1. **try…catch** only wraps the code you expect to throw.
2. **Don’t swallow errors:** log or rethrow so callers know something went wrong.
3. **Use .finally()** to release resources or reset UI state.

**When & Where**

* **Parsing:** JSON.parse can throw SyntaxError.
* **Network:** fetch rejects on network failure; responses may have non‑OK status.
* **User input:** validating forms before processing.

**Real‑World Example**

function loadSettings() {

try {

const raw = localStorage.getItem('settings');

if (!raw) {

console.warn('No settings found—using defaults.');

return { theme: 'light', alerts: true };

}

return JSON.parse(raw);

} catch (err) {

console.error('Could not parse settings:', err);

// Return safe defaults

return { theme: 'light', alerts: true };

}

}

// Promise‑style fetch with catch

fetch('/api/data')

.then(resp => {

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

return resp.json();

})

.then(data => {

render(data);

})

.catch(err => {

console.error('Data load failed:', err);

showError('Failed to load data.');

});

**3. HTTP Methods Using fetch**

**What & Why**

* **What:** Browser API to make HTTP requests, returning a Promise for the Response.
* **Why:** Cleaner, Promise‑based replacement for XMLHttpRequest, with streaming support.

**Rules & Best Practices**

1. **Check response.ok** (status 200–299) before parsing.
2. **Set appropriate headers**, e.g. Content-Type: application/json.
3. **Serialize/deserialize** JSON with JSON.stringify and response.json().
4. **Abort long requests** via AbortController if needed.

**When & Where**

* **RESTful CRUD** on backend resources.
* **Submitting forms** or file uploads.
* **Polling** or long‑polling endpoints.

**HTTP Methods Cheat‑Sheet**

| **Method** | **Purpose** | **Example Use** |
| --- | --- | --- |
| **GET** | Read data | Fetching a list of items |
| **POST** | Create data | Submitting a new record |
| **PUT** | Replace data | Overwriting an entire entity |
| **PATCH** | Update partial data | Changing one field |
| **DELETE** | Remove data | Deleting an item |

**Real‑World Examples**

**✅ 1. GET – Simulated User Fetch**

<h3>GET User</h3>

<pre id="getOutput">Loading...</pre>

<script>

function getUser() {

return new Promise(resolve => {

setTimeout(() => {

resolve({ id: 1, name: "Leanne Graham", email: "leanne@example.com" });

}, 1000);

});

}

getUser().then(user => {

document.getElementById("getOutput").textContent =

`👤 Name: ${user.name}\n📧 Email: ${user.email}`;

});

</script>

**✅ 2. POST – Simulated Create Post**

<h3>POST Create Post</h3>

<pre id="postOutput">Creating...</pre>

<script>

function createPost(data) {

return new Promise(resolve => {

setTimeout(() => {

resolve({ id: Math.floor(Math.random() \* 1000), ...data });

}, 1000);

});

}

createPost({ title: "New Post", body: "Some content" })

.then(post => {

document.getElementById("postOutput").textContent =

`✅ Post Created:\n🆔 ID: ${post.id}\n📄 Title: ${post.title}`;

});

</script>

**✅ 3. PUT – Simulated Full Update**

<h3>PUT Update Post</h3>

<pre id="putOutput">Updating...</pre>

<script>

function updatePost(id, data) {

return new Promise(resolve => {

setTimeout(() => {

resolve({ id, ...data });

}, 1000);

});

}

updatePost(101, { title: "Updated Title", body: "Updated Body" })

.then(post => {

document.getElementById("putOutput").textContent =

`🔁 Updated Post:\nID: ${post.id}\nTitle: ${post.title}`;

});

</script>

**✅ 4. PATCH – Simulated Partial Update**

<h3>PATCH Post Title</h3>

<pre id="patchOutput">Patching...</pre>

<script>

function patchPost(id, patchData) {

return new Promise(resolve => {

setTimeout(() => {

resolve({ id, ...patchData });

}, 1000);

});

}

patchPost(101, { title: "Patched Title" })

.then(post => {

document.getElementById("patchOutput").textContent =

`🧩 Patched:\nID: ${post.id}\nNew Title: ${post.title}`;

});

</script>

**✅ 5. DELETE – Simulated Post Deletion**

<h3>DELETE Post</h3>

<pre id="deleteOutput">Deleting...</pre>

<script>

function deletePost(id) {

return new Promise(resolve => {

setTimeout(() => {

resolve({ success: true, id });

}, 1000);

});

}

deletePost(101)

.then(result => {

document.getElementById("deleteOutput").textContent =

result.success ? `🗑 Post ${result.id} deleted.` : `❌ Delete failed`;

});

</script>

**✅ Topic 1: Async & Await**

**Quiz 1**

**Q:** What does an async function always return?

**A:** A Promise.

**Quiz 2**

**Q:** What will be the output?

async function test() {

return 42;

}

test().then(console.log);

**A:** 42 — because async functions return a Promise that resolves with the return value.

**Quiz 3**

**Q:** Can you use await outside of an async function?

**A:** No, unless you’re in a top-level ES module (in modern browsers).

**Quiz 4**

**Q:** What will happen here?

async function fetchData() {

let data = await fetch('invalid-url');

console.log('done');

}

fetchData();

**A:** It throws a network error and skips console.log('done').

**✅ Topic 2: Error Handling**

**Quiz 5**

**Q:** What is the output?

try {

JSON.parse("{invalid}");

} catch (err) {

console.log("Caught error:", err.name);

}

**A:** Caught error: SyntaxError

**Quiz 6**

**Q:** What does finally do?

**A:** It runs **no matter what**—whether an error occurs or not.

**Quiz 7**

**Q:** True or False? try...catch can catch both sync and async errors.

**A:** False. It can only catch **synchronous errors** unless you await inside an async function.

**Quiz 8**

**Q:** What's wrong with this code?

fetch('/api/data')

.then(data => JSON.parse(data))

.catch(console.error);

**A:** data is a Response object, not JSON. You should use data.json() instead of JSON.parse(data).

**✅ Topic 3: HTTP Methods Using Fetch**

**Quiz 9**

**Q:** Which HTTP method is used to **update part** of an existing resource?

**A:** PATCH

**Quiz 10**

**Q:** What does fetch return?

**A:** A Promise that resolves to a Response object.

**Quiz 11**

**Q:** What’s wrong with this code?

fetch('/api/products', {

method: 'POST',

body: JSON.stringify({ name: "Pen" })

});

**A:** Missing 'Content-Type': 'application/json' header.

**Quiz 12**

**Q:** How do you check if a fetch request succeeded?

**A:**

fetch('/api')

.then(res => {

if (!res.ok) throw new Error('Request failed');

return res.json();

});