Async JS & Promises

GUIDE

Waiting, without waiting — how JS handles time!

Synchronous vs Asynchronous · Callbacks

Promises · .then() .catch() · async/await

fetch() API · Error Handling · Real Examples

Learn With Asad

learnwithasad.com • prepnex.in

Contents

01 Sync vs Async

The restaurant vs. the pizza delivery

Imagine ordering a pizza

SYNCHRONOUS: You stand at the pizza shop and wait. Nobody behind you can order until your pizza is ready. Everyone is blocked! ASYNCHRONOUS: You order, they give you a buzzer, and you go sit down. Other people can order. When pizza is ready, the buzzer goes off and you collect it. Nothing is blocked!

Synchronous Code — One at a time

```
// Each line waits for the previous one to finish:
console.log('Step 1: Wake up');        // runs first
console.log('Step 2: Brush teeth');    // then this
console.log('Step 3: Have breakfast'); // then this

// Output:
// Step 1: Wake up
// Step 2: Brush teeth
// Step 3: Have breakfast
```

Synchronous = sequential = predictable. But if one step takes 10 seconds, EVERYTHING waits.

Asynchronous Code — Don't wait!

```
console.log('1. Start order');        // runs now

// setTimeout = wait X milliseconds, THEN run:
setTimeout(() => {
  console.log('3. Pizza is ready! (2 sec later)');
}, 2000);

console.log('2. Waiting... (but not blocking)'); // runs NOW

// Output ORDER:
// 1. Start order
// 2. Waiting... (but not blocking)
// 3. Pizza is ready! (2 sec later)
```

Notice: line 3 is set to run after 2 seconds, but line after it runs IMMEDIATELY. The code doesn't stop and wait. That is asynchronous!

Tip: JavaScript is single-threaded — it can only do one thing at a time. Async lets it SCHEDULE work and keep moving while waiting.

02 Callbacks

The old way of handling async

A **callback** is a function you pass to another function, saying: 'When you're done with your work, call THIS function.'

A Simple Callback

```
function orderPizza(type, onReady) {
  console.log(`Ordering ${type} pizza...`);

  setTimeout(() => {
    const pizza = `Hot ${type} pizza`;
    onReady(pizza);  // call the callback when done!
  }, 2000);
}

// Pass a function as the callback:
orderPizza('Margherita', (pizza) => {
  console.log(`Received: ${pizza}! Eating now!`);
});

console.log('Waiting for pizza...');
```

The Problem: Callback Hell!

When you need multiple async steps in order, callbacks lead to deeply nested, unreadable code. This is called Callback Hell:

```
// Nightmare version:
loginUser('asad', (user) => {
  getProfile(user.id, (profile) => {
    getFriends(profile.id, (friends) => {
      getMessages(friends[0].id, (messages) => {
        // 4 levels deep... and growing!
        console.log(messages);
      });
    });
  });
});
// This is hard to read, hard to fix, and hard to love!
```

Real Life: Imagine getting directions and being told: turn left, then when you see a red house, ask inside for more directions, then that person will tell you...

Tip: Callback Hell is also called the 'Pyramid of Doom' because of its triangle shape. Promises were invented to fix this!

03 Promises

A better way to handle the future

A Promise is exactly what it sounds like

When someone makes you a promise, they say: 'I will do this — and when I'm done, I'll tell you if it worked or not.' A JavaScript Promise says the same: 'I'll try to get your data. If I succeed — here it is! If I fail — here's the error.'

A Promise has 3 states

`pending`	Working on it... not done yet (like waiting for pizza)
`fulfilled`	Done! Success! Here's your result (pizza arrived)
`rejected`	Failed. Something went wrong (shop was closed)

Creating a Promise

```
const myPromise = new Promise((resolve, reject) => {
  // Do some work...
  const success = true;

  if (success) {
    resolve('Great news! It worked!');  // fulfilled
  } else {
    reject('Oops! Something failed.');  // rejected
  }
});

console.log(myPromise);  // Promise { 'pending' }
```

Simulating an Async Promise

```
function orderPizza(type) {
  return new Promise((resolve, reject) => {
    console.log(`Making your ${type} pizza...`);

    setTimeout(() => {
      const shopOpen = true;

      if (shopOpen) {
        resolve(`Hot ${type} pizza is ready!`);
      } else {
        reject('Shop is closed. Sorry!');
      }
    }, 2000);
  });
}
```

Tip: A function that returns a Promise can be used with .then()/.catch() or async/await. This is the standard pattern in JavaScript!

04 .then() / .catch() / .finally()

Reacting to the Promise result

.then() — What to do on success

```
orderPizza('Margherita')
  .then((result) => {
    // runs if the promise was FULFILLED (success)
    console.log(result);  // 'Hot Margherita pizza is ready!'
    return result.toUpperCase();  // can return new value
  })
  .then((upperResult) => {
    // chain another .then()!
    console.log(upperResult);  // 'HOT MARGHERITA PIZZA IS READY!'
  });
```

.catch() — What to do on failure

```
orderPizza('Pepperoni')
  .then((result) => {
    console.log('Pizza arrived:', result);
  })
  .catch((error) => {
    // runs if the promise was REJECTED (failed)
    console.log('Order failed:', error);
    // error = 'Shop is closed. Sorry!'
  });
```

.finally() — Runs no matter what

```
orderPizza('BBQ Chicken')
  .then((result) => {
    console.log('Eating:', result);
  })
  .catch((error) => {
    console.log('Error:', error);
  })
  .finally(() => {
    // ALWAYS runs — success OR failure
    console.log('Payment processed.');
    hideLoadingSpinner();
  });
```

Real Life: finally is like paying the delivery driver — whether the pizza was perfect or wrong, you still pay!

Promise Chaining — The Clean Version

```
// Compare to Callback Hell from Ch02:
loginUser('asad')
  .then(user => getProfile(user.id))
  .then(profile => getFriends(profile.id))
  .then(friends => getMessages(friends[0].id))
  .then(messages => console.log(messages))
  .catch(err => console.log('Something failed:', err));

// Same logic — but now it's READABLE! Left to right, top to bottom.
```

Tip: .then() chains are flat and readable. Each .then() receives what the previous one returned.

05 async / await

The cleanest way to write async code!

async/await makes async code look synchronous

Instead of chaining .then() blocks, async/await lets you write asynchronous code that looks and reads like normal, top-to-bottom synchronous code. It's just a prettier way to use Promises — under the hood, it's still Promises!

async function

```
// Adding 'async' makes a function return a Promise automatically:
async function greet() {
  return 'Hello World!';  // auto-wrapped in Promise.resolve()
}

greet().then(msg => console.log(msg));  // Hello World!

// You can also use await inside:
async function example() {
  const result = await greet();  // wait for the Promise
  console.log(result);           // 'Hello World!'
}
```

await — Wait for a Promise

```
async function getPizza() {
  console.log('Placing order...');

  const pizza = await orderPizza('Margherita'); // WAIT here

  // This line only runs AFTER the pizza promise resolves:
  console.log('Got pizza:', pizza);
  console.log('Eating now!');
}

getPizza();
console.log('This runs while waiting for pizza!');
```

await pauses ONLY inside the async function. The rest of the program keeps running!

Error Handling with try/catch

```
async function getStudentData(id) {
  try {
    const student = await fetchStudent(id);      // might fail
    const grades  = await fetchGrades(student);  // might fail
    return { student, grades };

  } catch (error) {
    console.log('Failed to get data:', error.message);
    return null;  // return something safe

  } finally {
    console.log('Done fetching — hiding loader');
    hideLoader();  // always clean up
  }
}
```

Tip: try/catch with async/await is cleaner than .catch(). One catch block handles errors from any await line above it.

06 fetch() — Talking to the Internet

Load real data from APIs!

fetch() lets JavaScript get data from the internet

An API is like a waiter at a restaurant. You tell the waiter what you want (your request), and they bring back data from the kitchen (the server). fetch() is how JavaScript places that order!

Basic fetch()

```
// Fetch returns a Promise!
fetch('https://api.example.com/students')
  .then(response => response.json())  // convert to JS object
  .then(data => console.log(data))
  .catch(error => console.log('Failed:', error));
```

fetch() with async/await (cleaner)

```
async function getStudents() {
  const response = await fetch('https://api.example.com/students');

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

  const students = await response.json();
  return students;
}

getStudents()
  .then(students => {
    console.log(`Got ${students.length} students!`);
    students.forEach(s => console.log(s.name));
  })
  .catch(err => console.log('Error:', err.message));
```

Sending Data (POST Request)

```
async function createStudent(name, age) {
  const response = await fetch('/api/students', {
    method:  'POST',
    headers: { 'Content-Type': 'application/json' },
    body:    JSON.stringify({ name, age })  // convert to JSON text
  });

  if (!response.ok) throw new Error('Failed to create student');

  const newStudent = await response.json();
  console.log('Created:', newStudent);
  return newStudent;
}

await createStudent('Asad', 15);
```

Watch Out! Always check response.ok after fetch! A 404 or 500 from the server does NOT automatically trigger .catch() — only network failures do.

07 Promise.all() & friends

Running Promises together!

Promise.all() — Run in Parallel

If you have multiple independent Promises, run them at the same time instead of one after another!

```
// Sequential (slow):
const maths   = await fetchScore('maths');    // wait...
const science = await fetchScore('science');  // wait again...
const english = await fetchScore('english');  // wait again...

// Parallel with Promise.all (3x faster!):
const [maths, science, english] = await Promise.all([
  fetchScore('maths'),
  fetchScore('science'),
  fetchScore('english'),
]);
// ALL three run at the same time!
```

Real Life: Sequential = ordering 3 dishes one at a time and waiting for each. Promise.all = ordering all 3 at once!

Promise.all() Behaviour

```
const results = await Promise.all([
  fetch('/api/user'),
  fetch('/api/posts'),
  fetch('/api/comments'),
]);

// results = [userResponse, postsResponse, commentsResponse]
// If ANY ONE fails → the whole Promise.all rejects!
```

Watch Out! If even ONE Promise inside Promise.all fails, the entire thing fails. Use Promise.allSettled if you want all results regardless.

Promise.allSettled() — Never Give Up

```
const results = await Promise.allSettled([
  fetch('/api/user'),
  fetch('/api/broken-endpoint'),
  fetch('/api/posts'),
]);

results.forEach(result => {
  if (result.status === 'fulfilled') {
    console.log('Success:', result.value);
  } else {
    console.log('Failed:', result.reason);
  }
});
// Gets all results — both successes AND failures
```

Promise.race() — First One Wins

```
// Use case: timeout! If fetch takes > 5 seconds, give up.
const timeout = new Promise((_, reject) =>
  setTimeout(() => reject(new Error('Timeout!')), 5000)
);
```

```
const data = await Promise.race([
  fetch('/api/slow-endpoint'),
  timeout,
]);
// Whichever resolves/rejects FIRST wins the race
```

08 Real Project: Weather App

fetch + async/await in action!

Let's build a simple weather app that fetches real weather data for any city. This uses everything from this guide: fetch, async/await, error handling, and DOM!

The Full Code

```
// Using the free Open-Meteo API (no key needed!)
async function getWeather(city) {
  // Step 1: Get coordinates for the city
  const geoUrl = `https://geocoding-api.open-meteo.com/v1/search?name=${city}&count=1`;

  const geoRes  = await fetch(geoUrl);
  const geoData = await geoRes.json();

  if (!geoData.results || geoData.results.length === 0) {
    throw new Error(`City '${city}' not found!`);
  }

  const { latitude, longitude, name } = geoData.results[0];

  // Step 2: Get weather for those coordinates
  const weatherUrl = `https://api.open-meteo.com/v1/forecast
    ?latitude=${latitude}&longitude=${longitude}
    &current_weather=true`;

  const weatherRes  = await fetch(weatherUrl);
  const weatherData = await weatherRes.json();

  return {
    city: name,
    temp: weatherData.current_weather.temperature,
    wind: weatherData.current_weather.windspeed,
  };
}
```

The UI Handler

```
const btn    = document.querySelector('#searchBtn');
const input  = document.querySelector('#cityInput');
const result = document.querySelector('#result');

btn.addEventListener('click', async () => {
  const city = input.value.trim();
  if (!city) return;

  result.textContent = 'Loading...';
  btn.disabled = true;

  try {
    const weather = await getWeather(city);
    result.innerHTML = `
      <h2>${weather.city}</h2>
      <p>Temperature: ${weather.temp} C</p>
      <p>Wind Speed: ${weather.wind} km/h</p>
    `;
  } catch (err) {
    result.textContent = `Error: ${err.message}`;
```

```
  } finally {
    btn.disabled = false;  // always re-enable button
  }
});
```

Tip: This project uses async/await, try/catch/finally, fetch, DOM selection, textContent, and event listeners — the full package!