TypeScript Async & Concurrency

1. Promises — construction & settling

the executor runs synchronously — new Promise(executor) runs the executor DURING construction, before new Promise returns; only .then callbacks defer

three states, settle once — pending → fulfilled or rejected, irreversibly; a second resolve or a later reject is silently ignored

.then on a settled promise — still defers to a microtask, never synchronous — Promise.resolve(1).then(log) prints after the whole sync tail

a promise that never settles — a code path that calls neither res nor rej waits forever — the missing else, or a try whose catch never calls rej

unhandled rejection timing — node checks AFTER the sync code finishes and the whole microtask queue drains, BEFORE timers: a .catch on the next line or from inside a queueMicrotask is in time; one inside setTimeout is too late (crash, exit 1)

combinators attach handlers — Promise.all / race call .then on every member as they run — the losers of multiple rejections never warn

.finally — runs on both outcomes, does NOT catch, and its return value is DISCARDED — the original settlement passes straight through; only a throw from the callback changes it

2. async / await

throw becomes a rejection — an async function converts a throw into a rejected promise — a try/catch around the bare (un-awaited) call sees nothing, and the rejection goes unhandled

return await inside try — keeps the rejection catchable by this function's catch; return f() without the await settles the outer promise and the catch never sees it

sequential vs parallel — two awaits with no dependency serialize for nothing — const [u, o] = await Promise.all([getUser(), getOrders()])

race / any / allSettled — race = first to SETTLE (a rejection wins too); any = first to RESOLVE, and only when all reject it rejects with AggregateError carrying .errors; allSettled waits for all, never rejects, {status:'fulfilled', value} / {status:'rejected', reason}

forEach + async doesn't wait — forEach discards the returned promises — 'after' prints before any body finishes; Promise.all(arr.map(async ...)) is the parallel form, for...of with await the sequential one

3. The event loop

microtasks drain completely — the ENTIRE microtask queue drains after each macrotask — a microtask that re-queues itself starves the timers forever; setImmediate cannot starve the loop (it yields to the check phase)

the order — process.nextTick queue → microtasks → timers; .then and queueMicrotask share ONE FIFO queue — arrival order decides, neither outranks; a nextTick queued from inside a microtask waits for the running microtask queue to finish

await resumes on a microtask — an async function runs synchronously up to the first await; everything after it is a microtask

Node phases — I/O callbacks run in the poll phase and check comes right after — so setImmediate inside an I/O callback ALWAYS beats setTimeout(0); at top level the two have no guaranteed order

timers & process lifetime — a pending timer keeps the node process alive; setInterval never lets it exit until clearInterval(id) or id.unref(); an unref'd timer stops holding the loop open (and may never fire)

4. Utilities & cancellation

node:timers/promises — setTimeout(delay, value, { ref, signal }) — the SECOND argument is the resolved value; { signal } is the abortable sleep for free; { ref: false } ≡ unref(); it rejects with node's own AbortError (.code ABORT_ERR) and DISCARDS signal.reason

sleep — const sleep = (ms: number): Promise<void> => new Promise(res => setTimeout(res, ms))

timeout a promise — new Promise<never>((_, rej) => setTimeout(() => rej(new Error('timeout')), ms)) — the <never> keeps Promise.race([work(), t]) from widening to Promise<unknown>; race settles the CALLER and never cancels the loser

retry with backoff — await the sleep (a fire-and-forget sleep is no backoff at all), no delay after the last attempt, return await work() inside the try, delay = base * 2 ** attempt

debounce vs throttle — debounce stores a TIMER ID and clears it on every call (the last call wins after quiet); throttle stores a TIMESTAMP assigned INSIDE the firing if — assigned unconditionally it fires once per burst and never again

worker pool — read the shared cursor into a local BEFORE the await (const idx = i++), write results by index; Promise.all receives LIMIT workers, not one promise per item; a worker pulls its next item itself (while (i < items.length))

structured concurrency — child tasks bound to a parent scope: leaving the scope cancels and awaits every child, so nothing outlives its parent; JS has only the manual version — an AbortSignal threaded by hand — and nothing stops a Promise.all loser from running on

abort identity — every API rejects with .name === 'AbortError' but DIFFERENT classes (fetch: DOMException; node timers/fs: node's AbortError) — the name check is the only one that survives both; a custom c.abort(reason) replaces the whole value and silently breaks it

AbortController shape — the controller keeps .abort(); the work receives only the read-only .signal; an abort surfaces as a REJECTION, so try/catch is the whole mechanism; combine causes with AbortSignal.any([user.signal, AbortSignal.timeout(5000)])

listener teardown with { signal } — addEventListener(type, fn, { signal }) — one abort() removes every listener registered with that signal, no stored function reference; { once: true } is different: it removes after the first firing

5. Async iteration

for await unwraps, it does not promote — over a sync array with promises inside, each value is awaited on the way out — ['a', load(), 'c'] binds three plain strings; over a sync iterable it uses Symbol.iterator and awaits each value itself

for await paces DELIVERY, not start — [...].map(start) has started everything before the loop begins; only a lazy source (an async generator) lets the loop decide when work begins

early exit calls .return() — break or throw in the body resumes the generator at the yield as though it returned — the finally runs before the error propagates, which is why cleanup is reliable

async function* — returns AsyncGenerator<T>; the work happens per pull; yield* spreads a batch one item at a time

Symbol.asyncIterator — async *[Symbol.asyncIterator]() on a class makes it for await-able; lookup order: Symbol.asyncIterator, else Symbol.iterator with each value awaited, else TypeError

backpressure — Promise.all(all.map(f)) starts everything and buffers everything; for await alone is flat but SERIAL; the production shape is a lazy source (cursor / async generator) plus a bounded pool pulling from it — flat memory, N in flight

6. Implementations — the checklists

sleep(ms, signal?) — check already-aborted BEFORE creating the timer (an aborted signal never fires abort again); clearTimeout inside the abort listener; REJECT with signal.reason (resolving on abort runs the caller's next line); { once: true } or the listener leaks per call

timeout(ms) — Promise<never> written at construction; (_, rej); a bare one leaks a pending timer the caller cannot clear — return { promise, cancel } and cancel in a finally; the losing rejection needs a handler

withTimeout(fn, ms, signal?) — the parameter is fn: (signal) => Promise<T>, never an already-started promise (nothing can cancel one); combine AbortSignal.any([signal, AbortSignal.timeout(ms)]); TimeoutError and AbortError must stay distinguishable; guard with signal?.throwIfAborted(), which preserves the caller's reason

retry(fn, attempts, base) — return await fn() in the try; no sleep after the last attempt; rethrow the last error; throwIfAborted in the catch or a cancel just starts the next attempt; the production rungs: shouldRetry(err), cap + full jitter, an overall deadline, a per-attempt timeout, onRetry hook, cause chaining

callable object (debounce.cancel/.flush/.pending) — an interface with a nameless call signature plus properties (an arrow type cannot carry them); build with Object.assign(run, { cancel, flush }); a getter inside the Object.assign literal is copied ONCE — a live .pending needs defineProperty or a method

single-flight — cache the PROMISE, not the value; .finally(() => map.delete(key)) so a rejection cannot poison the cache; insert before awaiting or two sync callers both miss

semaphore / withLock — a counter plus a queue of pending resolvers; the caller releases in a finally so a throw cannot deadlock the rest; Promise.withResolvers<void>() builds the waiter (ES2024)

Promise.all from scratch — results written by index, never push (completion order ≠ input order); a remaining COUNTER resolved at 0, not a results.length check; rejects on the first rejection (settle-once makes later settles no-ops); [] resolves immediately; Promise.resolve(x) every member

7. Recall — names & return values

.then returns a NEW promise — p.then(f) and p.then(g) are two independent chains; .catch(fn) is sugar for .then(undefined, fn)

setTimeout's return — a handle — a number in browsers, a Timeout object in node; store it for clearTimeout; type it ReturnType<typeof setTimeout>

queueMicrotask(fn) — ≡ Promise.resolve().then(fn); process.nextTick runs BEFORE the microtask queue; setImmediate runs in node's check phase, after timers

signal.aborted / signal.reason — aborted = is it aborted RIGHT NOW (not "was this rejection an abort"); reason defaults to a DOMException named 'AbortError'; throwIfAborted() throws signal.reason — the one place cancellation throws; abort() itself throws nothing at the call site

'abort' listeners — fire SYNCHRONOUSLY inside abort(); the type string is 'abort' (not 'aborted'); removeEventListener needs the type string and the SAME function reference

AbortSignal.timeout(ms) — a self-aborting signal whose reason is named 'TimeoutError' — the literal that lets a catch tell a timeout from a user cancel ('AbortError')

8. Failure modes

GC of pending awaits — a suspended await frame and its pending promise reference each other; a pair nothing else points at is collectable — the leak is the OUTSIDE reference (a Map of pending requests, a listener, a cache); unreachable ≠ resolved: the code after the await never runs