ARTICLE

JavaScript Promise Error Handling with async/await Patterns

Promise errors in async JavaScript are easy to miss when control flow is split across awaits, callbacks, and parallel work. This article explains how async/await changes error handling, when try/catch is enough, when errors should propagate, and how to validate behavior before production use.

Programming - August 11, 2026

Why Promise errors become operational problems

The practical problem is not that JavaScript promises can fail; it is that failures often surface later, in a different stack frame, or not at all when async code is structured poorly. In production systems, that creates symptoms such as partial API responses, silent background task failures, retry storms, and logs that show an exception without enough context to identify the request that caused it.

With async/await, you can usually make Promise error handling easier to reason about, but only if you understand how rejection propagates. After reading this article, you should be able to decide when try/catch is appropriate, when to let an error bubble up, how to handle parallel Promise work safely, and what to verify before shipping to production.

Key takeaways

Promise rejection is the error path for asynchronous work, and await turns that rejection into a thrown exception in the current async function. That means normal exception handling patterns apply, but only within the scope that actually awaits the Promise.

The safest default is to catch errors at the boundary where you can add context, classify the failure, or decide whether to retry. Inside internal helper functions, it is often better to preserve the original error and let a higher layer decide how to respond.

Parallel Promise operations need special attention. Promise.all() fails fast on the first rejection, while Promise.allSettled() gives you per-item outcomes. Choosing between them is not a style preference; it changes whether you see one failure or a complete result set.

Unhandled rejections are a production risk because they can hide background task failures and obscure the root cause. If you rely on asynchronous side effects, you need explicit error paths, logging, and a decision on whether a rejected Promise should fail the request, the job, or only a subtask.

How async/await changes Promise error handling

async/await does not remove Promise behavior. It changes how the code reads and how errors flow through the function.

An async function always returns a Promise. If the function returns normally, that Promise resolves. If the function throws synchronously or an awaited Promise rejects, the returned Promise rejects. In other words, a rejected Promise and a thrown error become the same operational event at the await boundary.

That makes try/catch useful again, but only around the code that actually awaits the operation.

This is practical because the error now carries both the low-level cause and the higher-level operation. In service code, that usually matters more than the syntactic detail of whether the failure came from a rejection or a thrown exception.

A useful mental model is this: await pauses only the current function, not the event loop globally. The rejected Promise becomes catchable in the same lexical scope, but any code that was started without being awaited will not be protected by that try/catch block. That distinction is where many production bugs start.

If you need a deeper view into the runtime side of this behavior, JavaScript Event Loop Performance Profiling with Async Hooks can help when you are tracing which asynchronous resource created delayed or missing work.

A compact workflow for handling Promise failures

A practical workflow for most service code is:

- Await the operation where the result is needed.
- Catch only where you can add useful context or make a decision.
- Rethrow if the caller should own the response or retry logic.
- For parallel work, choose Promise.all() only when one failure should fail the whole unit.
- Log enough context to identify the request, job, or resource without exposing secrets.

That workflow keeps handling consistent and avoids the common anti-pattern of catching every error at the lowest possible layer and turning meaningful failures into generic messages.

Where try/catch belongs and where it does not

try/catch is most effective at operational boundaries: request handlers, queue consumers, scheduled jobs, and command-line entry points. Those are the places where you know whether a failure should produce a 500 response, a failed job, a retry, or a degraded result.

Inside low-level helper functions, broad try/catch blocks can be harmful if they hide the original error or turn all failures into the same exception type. A helper should usually either:

- handle a specific expected failure and return a controlled fallback, or
- rethrow with added context.

A broad catch that silently returns null, false, or an empty array can make the system appear healthy while downstream code makes decisions on incomplete data.

This becomes especially important when the code is part of an API path that also performs authentication or authorization checks. If you are composing async handlers with Securing JavaScript APIs with JWT Authentication and Role Checks, keep the security decision separate from transport or parsing failures so you can tell whether a request was denied, malformed, or simply unavailable.

Good catch placement

Catch where you can answer one of these questions:

- Is this failure expected and recoverable?
- Should I retry, fail closed, or degrade gracefully?
- What context do I need to make the error actionable?

If the answer is none of those, let the error propagate.

Parallel promises: when one error should fail everything

Parallel promise handling is where many teams misread async/await. The code looks sequential, but the failure model changes once you start multiple asynchronous operations together.

Promise.all() is appropriate when the work is a single logical unit. If any item fails, the aggregate should fail. This is common for fan-out operations where the final response depends on every input succeeding.

This pattern is efficient and clean, but it also means one rejection aborts the aggregate. That is correct when the caller cannot use a partial result.

Promise.allSettled() is a better fit when partial success is useful and each result can be evaluated independently. For example, an enrichment pipeline might continue if one data source fails, as long as it can record which enrichment step was unavailable.

The trade-off is operational clarity versus completeness. Promise.all() gives a simple success-or-failure outcome. Promise.allSettled() gives more information, but you must explicitly inspect each result and decide how to handle failures. Do not choose allSettled just because it seems safer; choose it only when partial results are truly valid.

Practical scenario: the API request that looks successful until a background task fails

A common environment is an internal API that accepts a request, writes to a database, and then triggers one or more side effects such as cache invalidation, audit logging, or downstream event publication.

The request handler might look correct at first glance because the primary write succeeds. But if the side effect Promise is started and not awaited, the HTTP response can return success even when the side effect later rejects. Operationally, that creates an inconsistent system state: the caller thinks the action completed, while caches, logs, or downstream consumers are stale.

The symptom pattern is familiar:

- the API returns 200 or 201,
- one secondary system shows no corresponding update,
- logs contain an asynchronous error without the original request context,
- retries by clients do not fix the inconsistency because the primary write already happened.

The fix is not always to await everything indiscriminately. The real question is whether the side effect is part of the same transaction boundary. If it is required for correctness, await it and fail the request if it fails. If it is optional, make that explicit by capturing and logging the error in a controlled way so the system does not appear healthy when it is not.

What this means in practice

In production JavaScript, good Promise error handling is less about syntax and more about control over failure boundaries.

If the caller needs a hard guarantee, do not hide rejection behind a fire-and-forget Promise. If the caller can accept a partial result, use allSettled or an explicit per-item error model. If you need the original failure to remain useful, rethrow with context instead of replacing it with a generic error.

That also means your logs and metrics should distinguish between expected operational failures and programming defects. A rejected upstream request, a validation error, and a null dereference should not all be counted as the same thing. They require different remediation paths, different alert thresholds, and different runbook actions.

For teams that operate Node.js services at scale, this is where async tracing becomes valuable. When a promise rejection reaches a request handler far from the source, tracing the async chain helps you identify which task started the work and where the failure actually occurred.

Decision guidance

Use try/catch around await when you need to classify the error, attach context, translate it to a domain-specific response, or compensate for the failure.

Let the error propagate when the caller owns the retry, response mapping, or transaction boundary. Propagation is often the right choice in library code and lower-level service helpers.

Use Promise.all() when every subtask is required and one failure should cancel the aggregate outcome.

Use Promise.allSettled() when partial outcomes are acceptable and you have explicit logic for handling success and failure per item.

Use an explicit .catch() only when you are dealing with a standalone Promise expression and there is no surrounding async function boundary. Mixing the two styles in the same function is possible, but it should be intentional rather than habitual.

If a Promise is started but not awaited or returned, treat that as a design decision that needs review. In most service code, unobserved asynchronous work is the shortest path to unhandled rejection and inconsistent state.

Common mistakes that lead to hidden failures

One common mistake is catching errors too early and returning a default value that looks valid. That masks a failed dependency and pushes the problem downstream where the root cause is harder to locate.

Another is assuming try/catch around a block protects every asynchronous operation created inside it. It protects awaited work in that scope, not promises that continue independently after the function returns.

A third mistake is using Promise.all() for operations that are not truly atomic. If partial success is acceptable, failing the entire operation may cause unnecessary retries or user-visible errors.

It is also easy to lose context by throwing a new error without including the original cause. When possible, preserve the causal chain so operational logs can identify the real source of the failure.

Finally, many teams forget to define what should happen for background tasks. If a task is not critical, it should still have an explicit failure path. Silent background failures are difficult to detect and often expensive to debug.

Production readiness checklist

Before you ship async code with Promise handling, verify the following:

- Every awaited operation is inside a boundary that either handles or intentionally propagates failure.
- Fire-and-forget tasks are explicitly justified and monitored.
- Parallel work uses Promise.all() or Promise.allSettled() based on a documented failure rule.
- Errors preserve enough context to identify the operation, request, or job.
- Logging does not leak secrets, tokens, or raw payloads.
- The system has a clear response for rejected work: retry, fail, degrade, or compensate.
- Unhandled rejections are surfaced in test and staging environments.
- Partial success paths are covered by tests, not just the happy path.

If you cannot explain what should happen when one Promise rejects, the code is not ready for production.

Final takeaway

async/await makes Promise error handling easier to read, but it does not make errors disappear. The operational skill is deciding where failures should be caught, where they should be propagated, and when parallel work should fail as a unit versus return partial results. If you can define those boundaries clearly, your JavaScript becomes easier to debug, safer to run, and much less likely to hide the failure that matters most.

Use this guidance together with A* search algorithm to connect the workflow with related operational context already available on the site.