FAQ

JavaScript FAQ: How to Handle Async Errors with Promises

Async errors in JavaScript Promises are easy to miss until they break production workflows. This FAQ explains how to catch them reliably, when to use try/catch with await, and what to verify before you ship.

Programming - July 05, 2026

Async errors in Promise-based JavaScript matter because failures often happen outside the main execution flow, where they can bypass simple try/catch blocks and surface as unhandled rejections, partial writes, or silent logic errors. That creates operational risk in scripts, services, and automation where a missed rejection can leave systems in an inconsistent state. After reading this FAQ, you will know how Promise errors propagate, when to use .catch() versus await with try/catch, how to structure safe workflows, and what to verify before production use.

What is an async error in a Promise workflow?

An async error is a failure that occurs after the current synchronous call stack has already returned, typically during a Promise resolution or rejection. In practice, that means the error does not behave like a normal immediate exception unless you explicitly await the Promise or attach a rejection handler. A common example is a failed network request, a rejected database call, or a thrown error inside an async function that becomes a rejected Promise.

The key decision rule is simple: if a function returns a Promise, you must handle both the resolved and rejected path. For example, fetchData().then(...) is incomplete without .catch(...), and await fetchData() is incomplete without try/catch or a higher-level rejection boundary. In operational code, this is the difference between a controlled failure and a process that appears to succeed until later steps break.

How do Promise errors propagate?

Promise errors propagate by rejection, not by synchronous throw once the async operation has started. That means errors move through the chain until a rejection handler consumes them, and any link in the chain can transform or rethrow the error. If no handler exists by the time the chain settles, you get an unhandled rejection.

A practical example is a three-step workflow: read input, call an API, then write output. If the API call rejects and you do not attach a handler, the output step never runs, but the failure may still be easy to miss if logging is weak. This is why a consistent JavaScript reporting pattern can help when async jobs need auditable outcomes; the same principle applies to error handling because every failure should produce a predictable status or record.

A useful validation point is to confirm where the error is caught. If a Promise chain has multiple .then() calls, the nearest .catch() handles rejections from any earlier step unless they are handled locally. That makes the chain concise, but it also means one missing catch can let a rejection escape farther than expected.

Should you use .catch() or try/catch with await?

Use .catch() when you are staying in Promise chain style, and use try/catch when you are using async/await. Both handle rejections; the better choice depends on readability, flow control, and how much surrounding logic you need to protect. The mistake is mixing the two styles without a reason, because that often creates gaps where errors are either swallowed or handled twice.

In a simple chain, .catch() is direct and clear:

When you need sequential steps with local error handling, async/await with try/catch is usually easier to reason about:

The decision rule is: use the style that keeps the error boundary closest to the operational action. If you need to handle an error once for the whole workflow, try/catch is often simpler. If you are composing small Promise utilities, .catch() keeps the chain explicit.

How do you catch errors inside a Promise chain?

You catch errors inside a Promise chain by adding a .catch() at the end of the chain or by returning a rejected Promise from any step when you need to abort the workflow. Errors thrown inside a .then() callback are also converted into rejections and will flow to the nearest .catch(). That behavior is useful because it lets you centralize failure handling without wrapping every step manually.

For example, if validation fails after an API response, you can throw an error inside the chain and let the final catch handle it:

A useful caveat is that .catch() only sees errors that occur in the chain leading to it. If you create a Promise and forget to return it, the outer chain may resolve before that inner work completes, and the rejection can escape the intended handler. That is one of the most common causes of false confidence in async code.

What is the safest way to handle multiple async steps?

The safest way is to keep each step explicit, return every Promise you start, and place one clear failure boundary around the full operation. For multi-step automation, that usually means validating inputs first, awaiting each step in order, and stopping on the first failure unless you intentionally want partial success. This approach is especially important when the workflow touches stateful systems such as files, queues, or security controls.

A practical pattern is:

- Validate the input before any side effect.
- Await each async operation in sequence.
- Catch errors once at the orchestration layer.
- Log enough context to identify the failing step.
- Preserve the original error or stack when rethrowing.

This is also where operational discipline matters. If the workflow will be deployed as part of a broader implementation, a JavaScript implementation checklist can help ensure error handling, test coverage, and rollback expectations are verified before production use.

The validation point here is whether the workflow can leave partial state behind. If it can, you need compensating logic, idempotency, or a retry strategy. If it cannot, a simple fail-fast design is usually safer and easier to support.

How do you avoid swallowing Promise errors?

You avoid swallowing Promise errors by never using empty catch blocks, never returning success after a failed await, and always deciding whether the error should be logged, transformed, or rethrown. A catch handler that only prints a message and then allows execution to continue can hide production defects. In security-sensitive or operational code, hidden failures are often worse than visible crashes because they can make controls appear healthy when they are not.

A safer pattern is to add context and then rethrow when the caller must know about the failure:

Use this carefully. If you wrap errors, keep the original cause when your runtime supports it or preserve the original error in logs, because losing stack context makes incident triage slower. The practical test is whether a caller can still tell what failed without reading the implementation.

How should you handle errors from parallel Promise operations?

When running parallel operations with Promise.all(), one rejection fails the whole aggregate promise. That is useful when every task is required for success, but it is not ideal if you need per-item results or partial completion. The choice depends on whether failure in one task should cancel the entire job.

For required-all-or-fail workflows, Promise.all() is the right fit:

If you need to inspect each result independently, consider collecting per-task outcomes and handling rejections in a controlled way. A common validation check is to confirm whether one failed operation should roll back the rest, continue independently, or trigger a retry. If the answer is not obvious, do not use parallel execution until the business rule is defined.

The caveat is that Promise.all() does not wait for the other Promises to finish after one rejects; it rejects as soon as the first failure occurs. That is efficient, but it means you should not assume every side effect completed before the catch handler runs.

What should you verify before production use?

Before production use, verify that every async entry point has a rejection path, that logs include enough context to identify the failing operation, and that failures produce the expected exit code or status response. You should also confirm that your runtime and linting rules surface unhandled rejections during testing, because defaults can vary by environment and version. If behavior depends on a specific Node.js or browser runtime, verify that version explicitly rather than assuming consistency across deployments.

A practical pre-production check looks like this:

- Every Promise-returning function is either awaited or returned.
- Every awaited call is inside a meaningful try/catch boundary.
- Every .catch() either handles the error or rethrows it.
- Parallel work has an agreed failure policy.
- Logs, metrics, or audit output make the failure easy to trace.

For teams that need repeatable output and traceability, aligning the error path with a reliable reporting template can make failure records easier to compare during troubleshooting. The operational goal is not just to catch the error, but to prove that the system reacted in the intended way.

What is a practical example of handling async errors correctly?

A practical example is a service startup script that loads configuration, validates it, connects to a backend, and exits with a non-zero status if anything fails. This is a common pattern in automation because startup errors should be immediate and visible rather than deferred. The safest structure is to keep the entire startup in one async function with a top-level error boundary.

This pattern works because it keeps control flow simple: one entry point, one failure boundary, and one clear outcome. The validation point is that the process should fail fast and visibly when startup cannot complete. If the script continues after a failed connection, the error handling is not doing its job.

When is a Promise-based error strategy not enough?

A Promise-based strategy is not enough when you need stronger guarantees than basic rejection handling, such as idempotent retries, transactional behavior, or guaranteed cleanup after partial failure. In those cases, error handling must be paired with compensating actions, state tracking, or orchestration logic. Promise handling alone can tell you that something failed, but it cannot by itself decide how to restore system consistency.

That distinction matters in security and infrastructure automation. If a failed step leaves a lock, credential, or partial policy update behind, you need explicit cleanup and verification before retrying. The practical rule is to treat Promise error handling as the detection layer, not the whole recovery strategy.

A good final check is to ask whether the caller can safely retry the operation after failure. If the answer is yes, your handler should support idempotency. If the answer is no, your handler must preserve state carefully and stop further automation until a human or a compensating workflow verifies the environment.

The main takeaway is straightforward: handle Promise errors at the right boundary, keep failure paths explicit, and verify the operational outcome before production use. If you can explain where the rejection is caught, what gets logged, and how the system behaves afterward, your async error handling is usually in good shape.

Use this guidance together with git branch cleanup script and Python JSONDecodeError to connect the workflow with related operational context already available on the site.

> Part of the Programming: JavaScript Insights content cluster.