

Error handling in async code

Error Handling in Asynchronous JavaScript — In Detail

Asynchronous code can fail in many ways — a failed network request, an exception in a callback, or a rejected promise. Proper error handling ensures your app behaves gracefully.

1 Error Handling with Callbacks

Traditionally, in **callback-based** code, errors are handled using the **Node.js-style error-first pattern**:

```
function fetchData(callback) {
  setTimeout(() => {
    const error = false;
    if (error) {
      callback("Something went wrong", null);
    } else {
      callback(null, { data: "Success!" });
    }
  }, 1000);
}

fetchData((err, result) => {
  if (err) {
    console.error("Error:", err);
  } else {
    console.log("Data:", result);
  }
});
```

Pros:

- Simple for small cases

❌ Cons:

- Doesn't scale well
- Leads to **callback hell**
- No `try...catch` support

2 Error Handling with Promises

With **Promises**, errors are caught using `.catch()`:

```
const fetchData = () => {
  return new Promise((resolve, reject) => {
    let success = false;
    if (success) {
      resolve("Data received");
    } else {
      reject("Failed to fetch data");
    }
  });
};

fetchData()
  .then((data) => console.log("Success:", data))
  .catch((error) => console.error("Error:", error));
```

❗ If any `.then()` throws, the error flows to the nearest `.catch()`.

3 Error Handling with `async/await`

In `async/await`, use `try...catch` blocks to handle errors:

```
const fetchData = async () => {
  try {
    const response = await fetch("https://api.example.com/data");
    if (!response.ok) throw new Error("Network response was not ok");
  }
}
```

```
const data = await response.json();
console.log("Data:", data);
} catch (err) {
  console.error("Caught Error:", err.message);
}
};

fetchData();
```

✓ Advantages:

- Synchronous-like syntax
- Easier to follow and debug
- Works seamlessly with `throw` and `try...catch`

4 Global Error Handling for Promises

In production, always handle unhandled promise rejections:

```
process.on("unhandledRejection", (reason, promise) => {
  console.error("Unhandled Rejection:", reason);
});
```

Or in browsers:

```
window.addEventListener("unhandledrejection", event => {
  console.error("Unhandled promise rejection:", event.reason);
});
```

5 Chained Promise Error Handling

You can handle multiple errors in a chain:

```

fetch("invalid-url")
  .then((res) => res.json())
  .then((data) => {
    console.log(data);
  })
  .catch((err) => {
    console.error("Error caught at the end:", err.message);
  });

```

! Only one `.catch()` is needed at the end if you're not handling errors in each step.

Best Practices

Tip	Description
✓ Always use <code>.catch()</code> with Promises	To prevent unhandled errors
✓ Wrap <code>await</code> in <code>try...catch</code>	Cleaner error boundaries
✗ Don't swallow errors silently	Always log or surface them
✓ Use meaningful messages	Helps debugging
✓ Validate response status	e.g., <code>if (!res.ok) throw new Error(...)</code>

Example with Async Function

```

async function getUser() {
  try {
    const res = await fetch("https://api.example.com/user");
    if (!res.ok) throw new Error("User not found");
    const user = await res.json();
    console.log("User:", user);
  } catch (error) {
    console.error("Fetch failed:", error.message);
  }
}

```

```
}  
}
```

← Summary

Async Technique	Error Handling Style
Callback	<code>if (err) {...}</code>
Promises	<code>.catch()</code>
Async/Await	<code>try...catch</code>

Proper error handling makes your asynchronous code **robust, reliable, and maintainable**.

Let me know if you want hands-on exercises or a mini project using async error handling!