Async/Await With Try-Catch — Deep Dive (Continuation of Previous Content)

This document builds on the earlier explanation of callbacks → promises → async/await and provides:

- Basic to advanced **async/await try-catch examples**
- Why `try/catch` is needed
- Internal behavior of error handling
- Interview questions and answers

---

1. Why Do We Use Try/Catch With Async/Await?

`async/await` makes asynchronous code look synchronous.

But synchronous-looking code needs **synchronous-style error handling**, and that's where `try/catch` becomes important.

Promises handle errors using:

```
promise.catch(errorHandler)
```

Async/Await handles errors using:

```
try {
  await someAsyncTask();
} catch (error) {
  console.error(error);
}
```

Why?

Because:

- Any promise rejection inside an `await` behaves like a **thrown error**.
- Without `try/catch`, the function will produce an **UnhandledPromiseRejection**.

---

2. Basic Async/Await Example WITHOUT Try/Catch (Not Recommended)

```
async function getData() {
  const data = await fetchData(); // If this rejects → crash
  console.log(data);
}

getData();
```

Problem:

If `fetchData()` fails → app crashes.

---

3. Basic Async/Await Example WITH Try/Catch (Recommended)

```
async function getData() {
  try {
    const data = await fetchData();
    console.log("Data received:", data);
  } catch (error) {
    console.error("Error while fetching data:", error);
  }
}

getData();
```

Explanation:

- `try` protects async code
- If any awaited promise fails → error goes to `catch`
- Prevents application from crashing

---

4. Intermediate Example — Multiple Awaits Inside One Try Block

```
async function processUser() {
  try {
    const user = await getUser();
    const orders = await getOrders(user.id);
    const shipping = await getShipping(orders[0]);

    console.log(shipping);
  } catch (err) {
    console.error("Something went wrong in processing the user:", err);
  }
}

processUser();
```

Explanation:

- Any of the three await calls can fail.
- Instead of writing `.catch()` three times, you handle errors once.
- Cleaner, production-friendly approach.

---

5. Advanced Example — Using Multiple Try/Catch Blocks for Granular Handling

```
async function processOrder() {
  let user, orders, shipping;

  try {
    user = await getUser(1);
  } catch (err) {
    console.error("User fetch failed:", err);
    return; // stop processing
  }

  try {
    orders = await getOrders(user);
  } catch (err) {
    console.error("Order fetch failed:", err);
    return;
  }

  try {
    shipping = await getShipping(orders[0]);
  } catch (err) {
    console.error("Shipping fetch failed:", err);
    return;
  }

  console.log("Shipping: ", shipping);
}
```

```
processOrder();
```

Why do this?

- You want **more precise error messages**.
- You want to stop execution only at specific points.
- You want to log meaningful messages for monitoring.

---

6. Expert Example — Custom Error Classes With Async/Await

```
class NotFoundError extends Error {}
class PermissionError extends Error {}

async function getUserProfile(id) {
  const user = await db.findUser(id);
  if (!user) throw new NotFoundError("User not found");
  if (!user.isActive) throw new PermissionError("Inactive user access 
denied");
  return user;
}

async function run() {
  try {
    const profile = await getUserProfile(5);
    console.log(profile);
  } catch (error) {
    if (error instanceof NotFoundError) {
      console.error("404 - User not found");
    } else if (error instanceof PermissionError) {
      console.error("403 - Forbidden: inactive user");
    } else {
      console.error("500 - Server Error", error);
    }
  }
}

run();
```

Why this is senior-level:

- Real applications use **error classes**.
- Helps differentiate between business logic errors.
- Helps APIs return correct HTTP statuses.

---

7. Key Behaviors of Error Handling in Async/Await

1. Any rejection = thrown error

```
await Promise.reject("fail"); // behaves like: throw "fail"
```

2. Try/Catch catches all async errors

Unlike callbacks, error handling is centralized.

3. If no try/catch is used, the async function returns a rejected promise

```
async function demo() {
  throw new Error("fail");
}

// Equivalent to
Promise.reject("fail");
```

---

8. Interview Questions (Async/Await + Try/Catch)

Q1. Why do we need try/catch in async/await?

Answer: Because `await` unwraps promise rejections as thrown errors. To prevent the function from failing unexpectedly, errors must be caught.

Q2. Is try/catch mandatory in async/await?

Answer: No, but it's strongly recommended. If omitted, the function returns a rejected promise which must be caught externally.

Example:

```
async function test() {
  const data = await fetch(); // If fails → test() returns a rejected promise
}

test().catch(console.error);
```

Q3. How does error propagation work in async/await?

Answer:

- If an error is not caught inside the async function, it **bubbles up** like a normal thrown error.
- But technically it returns a rejected promise.

Q4. Should we wrap every await in its own try/catch?

Answer:

- No for most apps → one global try/catch is enough.
- Yes when you need **granular error messages**.

Q5. What is the difference between throwing an error vs rejecting a promise?

Answer: With async/await, they behave the same:

```
throw new Error("fail");
// Same as
return Promise.reject("fail");
```

Q6. Can try/catch handle synchronous and asynchronous errors?

Answer: Yes.

```
try {
  // synchronous error
  JSON.parse("invalid");

  // async error
  await Promise.reject("Async fail");
} catch (err) {
  console.error(err);
}
```

This is one reason async/await + try/catch is powerful.

---

9. Summary Table — Error Handling Styles

Pattern	Error Handling	Difficulty	Best Use
Callback	callback(err, data)	Hard	Legacy code
Promise	.catch()	Medium	Modern async flows
Async/Await	try/catch	Easy	Cleanest production code
