Coding & Development / JavaScript

Convert callback-based or Promise.then() code to modern async/await syntax.
Difficulty: Intermediate
Model: GPT-4 / Claude / Gemini
Use Case: Code Modernization, Async/Await, Refactoring
Updated: May 2026
Why This Prompt Exists
Most legacy JavaScript uses callbacks or Promise chains — which are hard to read and debug.

You get:

- callback hell (nested functions, hard to follow)
- Promise.then() chains with .catch() at the end
- no error handling for specific steps
- hard to debug (stack traces are messy)
- difficult to add conditional logic between async steps

But async/await is not just syntax sugar.

It is readability and maintainability.

- Convert callbacks to async functions
- Replace Promise.then() with await
- Use try/catch for error handling
- Add proper error messages
- Preserve original behavior

Without conversion, async code is hard to maintain.

This framework forces AI to modernize async patterns.

The Prompt

```
Assume the role of a code modernization engineer who converts legacy
async patterns to async/await.

Your task is to convert callback or Promise-based code to async/await.

Generate:

1. ORIGINAL CODE (for reference)
   - As provided

2. CONVERTED CODE
   - Async/await syntax
   - Try/catch error handling

3. CHANGE SUMMARY
   - What was changed and why

4. ERROR HANDLING NOTES
   - How errors are now handled

5. TEST SUGGESTIONS
   - How to test the converted code

INPUTS:
```

```
Original Code (paste):
[PASTE CALLBACK OR PROMISE CODE]

Code Type:
[CALLBACK / PROMISE.THEN / MIXED]

Environment:
[NODE.JS / BROWSER]

Preserve Behavior:
[YES / NO]

RULES:
- Convert to async function (add async keyword)
- Replace .then() with await
- Wrap in try/catch for error handling
- Handle errors appropriately (don't just console.log)
- Preserve original behavior exactly
- Keep variable names the same
- Add proper error messages
- Test the converted code before deploying
```

How To Use It

- Paste the exact callback or Promise code you want to convert.
- Include error handling in the original so the conversion preserves it.
- Test the converted code thoroughly (async/await changes execution order subtly).
- Add try/catch for any operation that can throw (network, file I/O).
- Use this as a learning tool to understand async/await patterns.

Example Input

Original Code:

```
function getUserData(userId) {
return fetch(`/api/users/${userId}`)
.then(response => response.json())
.then(data => {
if (data.error) {
throw new Error(data.error);
}
return data;
})
.catch(error => {
console.error('Failed to fetch user:', error);
return null;
});
}
```

Code Type: PROMISE.THEN

Environment: BROWSER

Preserve Behavior: YES

Why It Works

Most legacy async code is hard to read.

This framework improves outcomes by forcing:

- callback/Promise → async/await (modern syntax)
- try/catch error handling (clarity)

- preserved behavior (safety)
- change summary (learning)
- test suggestions (validation)

Great async/await code isn’t just newer — it’s more readable, debuggable, and maintainable.

Build Better AI Systems

Subscribe for advanced prompt engineering, AI coding tools, JavaScript frameworks, and practical strategies for developers and engineers.

Carefully engineered prompts for people doing real work.

Share this:

- Share on Facebook (Opens in new window) Facebook
- Share on X (Opens in new window) X

See also The DOM Manipulation Script