

JavaScript

Understanding JavaScript Promises

Modern Asynchronous JavaScript
Part 3/3

May 14, 2025



Source Code

1. `async/await`: The Evolution of Promises

After mastering the basics of promises and their advanced patterns, it's time to explore `async/await`, a modern syntax that makes asynchronous code even more readable and maintainable. This feature, built on top of promises, allows us to write asynchronous code that looks and behaves more like synchronous code.

1.1. Understanding `async/await`

The `async` and `await` keywords provide a more elegant way to work with promises:

- `async`: Declares that a function returns a promise
- `await`: Pauses execution until a promise settles

Let's see how this transforms our promise-based code:

How to Run:


- Save the code as `08_async_await_basics.js`
- Open your terminal and run: `node 08_async_await_basics.js`

```
1 // --- 08_async_await_basics.js ---
2 // Simulating an API call that returns user data
3 async function fetchUserData(userId) {
4   console.log('Fetching user data for ID: ${userId}...');
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


5
6  // Simulate API delay
7  await new Promise(resolve => setTimeout(resolve, 1000));
8
9  // Simulate success/failure
10 if (userId <= 0) {
11   throw new Error('Invalid user ID');
12 }
13
14 return {
15   id: userId,
16   name: 'Alex',
17   email: `user${userId}@example.com`
18 };
19 }
20
21 // Using async/await with try/catch
22 async function displayUserProfile(userId) {
23   console.log('Starting user profile retrieval...');
24
25   try {
26     const userData = await fetchUserData(userId);

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```

27     console.log('User data retrieved successfully:');
28     console.log(userData);
29   } catch (error) {
30     console.error('Error fetching user data:',
31       error.message);
32   }
33   console.log('Profile display operation completed.');
```

```

34 }
35
36 // Execute our async function
37 console.log('Before calling async function');
```

```

38 displayUserProfile(123)
39   .then(() => console.log('Async operation chain
40     completed.'));
41 console.log('After calling async function (executes
42   immediately)');
```


```

43
44 // Try with an invalid ID to see error handling
45 setTimeout(() => {
46   console.log('\nTrying with invalid ID:');
47   displayUserProfile(-1);

```


Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```
46 }, 2000);
```

Key points about `async/await`:

- An `async` function always returns a promise
- `await` can only be used inside an `async` function
- The function pauses at each `await` until the promise resolves
- Error handling uses familiar `try/catch` syntax
- The code looks more like traditional synchronous code

1.2. Error Handling Patterns with `async/await`

When working with multiple asynchronous operations, proper error handling becomes crucial. Let's explore some practical patterns:

How to Run:

- Save the code as `09_error_handling_patterns.js`
- Open your terminal and run: `node 09_error_handling_patterns.js`




```
1 // --- 09_error_handling_patterns.js ---
2 // Simulating database operations
3 async function connectToDatabase() {
4   console.log('Connecting to database...');
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


5   await new Promise(resolve => setTimeout(resolve, 500));
6   return { connected: true };
7 }
8
9 async function queryDatabase(connection, query) {
10  console.log('Executing query: ${query}');
11  await new Promise(resolve => setTimeout(resolve, 800));
12
13  if (!connection.connected) {
14    throw new Error('Database connection lost');
15  }
16
17  if (query.includes('invalid')) {
18    throw new Error('Invalid SQL query');
19  }
20
21  return ['Result 1 for ${query}', 'Result 2 for
    ${query}'];
22 }
23
24 async function processResults(results) {
25  console.log('Processing results...');

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


26   await new Promise(resolve => setTimeout(resolve, 300));
27   return results.map(r => r.toUpperCase());
28 }
29
30 // Pattern 1: Sequential operations with proper cleanup
31 async function performDatabaseOperation(query) {
32   let connection = null;
33
34   try {
35     // Establish connection
36     connection = await connectToDatabase();
37
38     // Execute query
39     const results = await queryDatabase(connection, query);
40
41     // Process results
42     const processedResults = await processResults(results);
43
44     return processedResults;
45   } catch (error) {
46     console.error('Operation failed:', error.message);
47     throw error; // Re-throw to let caller handle it

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```

48   } finally {
49     if (connection) {
50       console.log('Closing database connection...');
51       connection.connected = false;
52     }
53   }
54 }
55
56 // Pattern 2: Parallel operations with Promise.all
57 async function executeMultipleQueries(queries) {
58   try {
59     const connection = await connectToDatabase();
60
61     console.log('Executing queries in parallel...');
62     const results = await Promise.all(
63       queries.map(query => queryDatabase(connection,
64         query))
65     );
66
67     const processedResults = await Promise.all(
68       results.map(result => processResults(result))
69     );

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com


```


69
70     return processedResults;
71 } catch (error) {
72     console.error('Batch operation failed:',
73         error.message);
74     throw error;
75 }
76
77 // Pattern 3: Retry mechanism
78 async function executeWithRetry(operation, maxAttempts =
79     3) {
80     for (let attempt = 1; attempt <= maxAttempts; attempt++)
81     {
82         try {
83             return await operation();
84         } catch (error) {
85             if (attempt === maxAttempts) throw error;
86
87             console.log('Attempt ${attempt} failed,
88                 retrying...');
89             await new Promise(resolve =>

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


87         setTimeout(resolve, Math.pow(2, attempt) * 100)
88     );
89 }
90 }
91 }
92
93 // Demo the patterns
94 async function demonstratePatterns() {
95     console.log('--- Pattern 1: Sequential with Cleanup
96         ---');
97     try {
98         const results = await performDatabaseOperation('SELECT
99             * FROM users');
100         console.log('Success:', results);
101     } catch (error) {
102         console.log('Handler caught:', error.message);
103     }
104
105     console.log('\n--- Pattern 2: Parallel Operations ---');
106     try {
107         const queries = [
108             'SELECT * FROM users',

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


107     'SELECT * FROM posts',
108     'SELECT * FROM comments'
109 ];
110 const results = await executeMultipleQueries(queries);
111 console.log('Batch results:', results);
112 } catch (error) {
113     console.log('Batch handler caught:', error.message);
114 }
115
116 console.log('\n--- Pattern 3: Retry Mechanism ---');
117 let failCount = 0;
118 const unreliableOperation = async () => {
119     failCount++;
120     if (failCount < 3) throw new Error('Temporary
121     failure');
122     return 'Operation succeeded!';
123 };
124 try {
125     const result = await
126     executeWithRetry(unreliableOperation);
127     console.log('Final result:', result);

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```
127   } catch (error) {  
128     console.log('Retry handler caught:', error.message);  
129   }  
130 }  
131  
132 demonstratePatterns();
```

These patterns demonstrate several important concepts:

- Resource cleanup using `try/finally` blocks
- Parallel execution while maintaining error handling
- Retry mechanisms for transient failures
- Proper error propagation to calling code

2. Best Practices and Advanced Patterns

When working with `async/await`, following certain practices can make your code more maintainable and robust:


2.1. Common Pitfalls and Solutions

Let's look at some common mistakes and their solutions:



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


1 // --- 10_best_practices.js ---
2 // [BAD] WRONG: Not handling errors
3 async function wrongErrorHandling() {
4     const data = await riskyOperation(); // Unhandled
      promise rejection!
5 }
6
7 // [GOOD] RIGHT: Proper error handling
8 async function rightErrorHandling() {
9     try {
10         const data = await riskyOperation();
11         return data;
12     } catch (error) {
13         console.error('Operation failed:', error);
14         throw error; // Re-throw if you want callers to handle
           it
15     }
16 }
17
18 // [BAD] WRONG: Sequential when parallel is possible

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


19 async function wrongSequential() {
20   const users = await fetchUsers();
21   const posts = await fetchPosts();
22   const comments = await fetchComments();
23 }
24
25 // [GOOD] RIGHT: Parallel execution when possible
26 async function rightParallel() {
27   const [users, posts, comments] = await Promise.all([
28     fetchUsers(),
29     fetchPosts(),
30     fetchComments()
31   ]);
32 }
33
34 // [BAD] WRONG: await in a loop
35 async function wrongLoop() {
36   const ids = [1, 2, 3, 4, 5];
37   const results = [];
38   for (const id of ids) {
39     results.push(await fetchData(id)); // Sequential
                                     execution

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


40   }
41 }
42
43 // [GOOD] RIGHT: Map and Promise.all
44 async function rightLoop() {
45   const ids = [1, 2, 3, 4, 5];
46   const results = await Promise.all(
47     ids.map(id => fetchData(id))
48   );
49 }
50
51 // [BAD] WRONG: Not considering race conditions
52 let data;
53 async function wrongRaceCondition() {
54   data = await fetchData(); // Global state modification
55 }
56
57 // [GOOD] RIGHT: Proper state management
58 class DataManager {
59   constructor() {
60     this.data = null;
61     this.loading = false;

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```

62   }
63
64   async fetchData() {
65     if (this.loading) return this.data;
66
67     this.loading = true;
68     try {
69       this.data = await fetchData();
70       return this.data;
71     } finally {
72       this.loading = false;
73     }
74   }
75 }

```


Best practices to follow:

- Always handle errors appropriately
- Use `Promise.all` for parallel operations when possible
- Avoid `await` in loops unless sequential execution is required
- Consider race conditions in shared state
- Use proper abstraction and encapsulation



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

3. Real-World Examples


Let's explore some practical examples that combine everything we've learned about promises and async/await:

```
1 // --- 11_real_world_examples.js ---
2 // Example 1: API Request with Timeout and Retry
3 async function fetchWithTimeout(url, timeout = 5000) {
4   const controller = new AbortController();
5   const timeoutId = setTimeout(() => controller.abort(),
6     timeout);
7
8   try {
9     const response = await fetch(url, { signal:
10       controller.signal });
11     const data = await response.json();
12     return data;
13   } finally {
14     clearTimeout(timeoutId);
15   }
16 }
```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


16 async function fetchWithRetry(url, retries = 3) {
17   for (let i = 0; i < retries; i++) {
18     try {
19       return await fetchWithTimeout(url);
20     } catch (error) {
21       if (i === retries - 1) throw error;
22       await new Promise(resolve =>
23         setTimeout(resolve, Math.pow(2, i) * 1000)
24       );
25     }
26   }
27 }
28
29 // Example 2: Resource Pool
30 class ResourcePool {
31   constructor(factory, poolSize = 5) {
32     this.resources = Array(poolSize).fill(null);
33     this.factory = factory;
34     this.available = [...Array(poolSize).keys()];
35     this.waiting = [];
36   }
37

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


38   async acquire() {
39     if (this.available.length > 0) {
40       const index = this.available.pop();
41       if (!this.resources[index]) {
42         this.resources[index] = await this.factory();
43       }
44       return { resource: this.resources[index], index };
45     }
46
47     return new Promise(resolve => {
48       this.waiting.push(resolve);
49     });
50   }
51
52   release({ resource, index }) {
53     if (this.waiting.length > 0) {
54       const resolve = this.waiting.shift();
55       resolve({ resource, index });
56     } else {
57       this.available.push(index);
58     }
59   }

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


60 }
61
62 // Example 3: Batch Processing with Rate Limiting
63 async function processBatch(items, batchSize = 3, delay =
    1000) {
64     const results = [];
65
66     for (let i = 0; i < items.length; i += batchSize) {
67         const batch = items.slice(i, i + batchSize);
68         const batchResults = await Promise.all(
69             batch.map(item => processItem(item))
70         );
71         results.push(...batchResults);
72
73         if (i + batchSize < items.length) {
74             await new Promise(resolve => setTimeout(resolve,
                delay));
75         }
76     }
77
78     return results;
79 }

```



Alejandro Sánchez Yalí


Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```
80
81 // Example 4: Event to Promise conversion
82 function eventToPromise(emitter, successEvent, errorEvent)
83 {
84   return new Promise((resolve, reject) => {
85     const success = (...args) => {
86       cleanup();
87       resolve(...args);
88     };
89     const error = (...args) => {
90       cleanup();
91       reject(...args);
92     };
93
94     const cleanup = () => {
95       emitter.removeListener(successEvent, success);
96       emitter.removeListener(errorEvent, error);
97     };
98
99     emitter.on(successEvent, success);
100    emitter.on(errorEvent, error);
```

**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


101   });
102 }
103
104 // Example usage for demonstration
105 async function demonstrateExamples() {
106   // Example 1: API Request with Timeout and Retry
107   try {
108     const data = await
      fetchWithRetry('https://api.example.com/data');
109     console.log('API Response:', data);
110   } catch (error) {
111     console.error('API Error:', error);
112   }
113
114   // Example 2: Resource Pool
115   const pool = new ResourcePool(async () => {
116     await new Promise(resolve => setTimeout(resolve, 100));
117     return { id: Math.random() };
118   });
119
120   // Use multiple resources concurrently
121   const operations = Array(8).fill().map(async (_, i) => {

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```


122     const { resource, index } = await pool.acquire();
123     console.log('Using resource ${resource.id} at index
    ${index}');
124     await new Promise(resolve => setTimeout(resolve, 200));
125     pool.release({ resource, index });
126     return `Operation ${i} complete`;
127 });
128
129 const results = await Promise.all(operations);
130 console.log('Resource pool results:', results);
131
132 // Example 3: Batch Processing
133 const items = Array(10).fill().map((_, i) => ({ id: i
    }));
134 const batchResults = await processBatch(items);
135 console.log('Batch processing results:', batchResults);
136
137 // Example 4: Event to Promise
138 const EventEmitter = require('events');
139 const emitter = new EventEmitter();
140
141 // Simulate an async event

```



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

```

142   setTimeout(() => emitter.emit('success', 'Operation
      completed'), 500);
143
144   const result = await eventToPromise(emitter, 'success',
      'error');
145   console.log('Event result:', result);
146 }
147
148 // Simulated processItem function for the batch processing
      example
149 async function processItem(item) {
150   await new Promise(resolve => setTimeout(resolve, 100));
151   return { ...item, processed: true };
152 }
153
154 // Run the examples if this file is executed directly
155 if (require.main === module) {
156   demonstrateExamples().catch(console.error);
157 }

```


These examples demonstrate:

- Combining timeouts with fetch requests
- Managing resource pools asynchronously



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

- Rate-limiting batch operations
- Converting event-based APIs to promises

4. Performance Considerations

When working with `async/await` and promises, keep these performance aspects in mind:

- **Memory Usage:** Promises keep references to their results/errors until all handlers complete
- **Microtasks:** Promise callbacks run as microtasks, which have priority over regular tasks
- **Stack Traces:** `async/await` provides better stack traces for debugging compared to raw promises
- **Parallel vs Sequential:** Use `Promise.all` when operations can run in parallel

5. Conclusions


Throughout this series on JavaScript Promises, we've covered:

- **Part 1:** Promise fundamentals, states, and basic handling
- **Part 2:** Advanced promise patterns and combination methods



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

- **Part 3:** Modern async/await syntax and real-world applications

Key takeaways:

- Promises provide a robust foundation for handling asynchronous operations
- `async/await` simplifies asynchronous code while maintaining promise benefits
- Error handling becomes more intuitive with `try/catch` syntax
- Real-world applications often combine multiple patterns

Understanding these concepts is crucial for modern JavaScript development, enabling you to write maintainable, efficient, and reliable asynchronous code.


6. References

- MDN Web Docs. (2025). *async function*. [Link](#)
- MDN Web Docs. (2025). *await*. [Link](#)
- ECMA International. (2025). *ECMAScript 2026: Async Functions*. [Link](#)
- Archibald, J. (2023). *JavaScript Async/Await: The Good Parts*. [Link](#)
- Simpson, K. (2023). *You Don't Know JS Yet: Async & Performance (2nd Edition)*. O'Reilly Media.



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com

- This article was translated, edited, and written in collaboration with AI. If you find any inconsistencies or have suggestions for improvement, please don't hesitate to open an issue in our [GitHub](#) repository or reach out directly.

7. Explore My Other Posts

Enjoyed This Content?

Don't miss my previous post about:


Understanding JavaScript Promises: Part 2/3

Learn about advanced Promise patterns like `Promise.all()`, `Promise.race()`, and how to manage multiple asynchronous operations effectively.



Alejandro Sánchez Yalí

Software Developer | AI & Blockchain Enthusiast

 www.asanchezyali.com



Feedback

Found this helpful?

Save, comment and share

May 14, 2025