JavaScript

Understanding JavaScript Promises

Modern Asynchronous JavaScript
Part 3/3

May 2, 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.

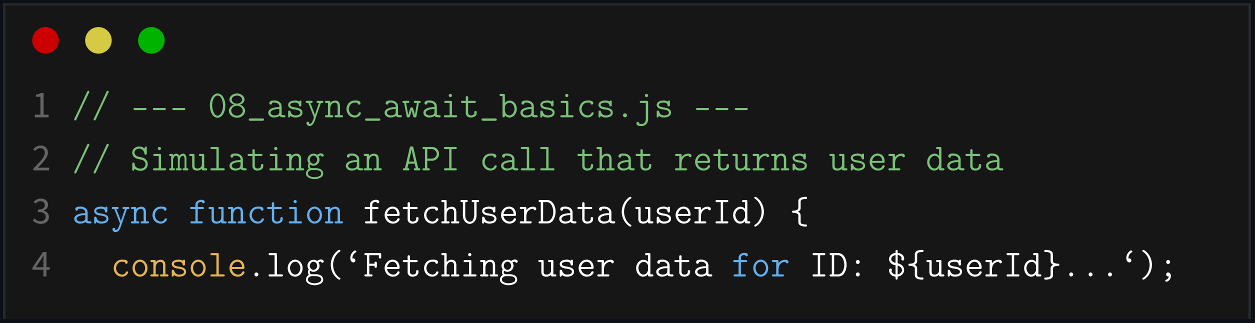
1.1. Understanding async/await

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`



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

  // Simulate API delay
  await new Promise(resolve => setTimeout(resolve, 1000));

  // Simulate success/failure
  if (userId <= 0) {
    throw new Error(’Invalid user ID’);
  }

  return {
    id: userId,
    name: ’Alex’,
    email: ‘user${userId}@example.com‘
  };
}

// Using async/await with try/catch
async function displayUserProfile(userId) {
  console.log(’Starting user profile retrieval...’);

  try {
    const userData = await fetchUserData(userId);
```

```
    console.log('User data retrieved successfully:');
    console.log(userData);
  } catch (error) {
    console.error('Error fetching user data:', error.message);
  }

  console.log('Profile display operation completed.');
}

// Execute our async function
console.log('Before calling async function');
displayUserProfile(123)
  .then(() => console.log('Async operation chain completed.'));
console.log('After calling async function (executes immediately)');

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

```
}, 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`

```
// --- 09_error_handling_patterns.js ---
// Simulating database operations
async function connectToDatabase() {
  console.log('Connecting to database...');
  await new Promise(resolve => setTimeout(resolve, 500));
  return { connected: true };
}

async function queryDatabase(connection, query) {
  console.log(‘Executing query: ${query}‘);
  await new Promise(resolve => setTimeout(resolve, 800));

  if (!connection.connected) {
    throw new Error(’Database connection lost’);
  }

  if (query.includes(’invalid’)) {
    throw new Error(’Invalid SQL query’);
  }

  return [‘Result 1 for ${query}‘, ‘Result 2 for ${query}‘];
}

async function processResults(results) {
  console.log(’Processing results...’);
```

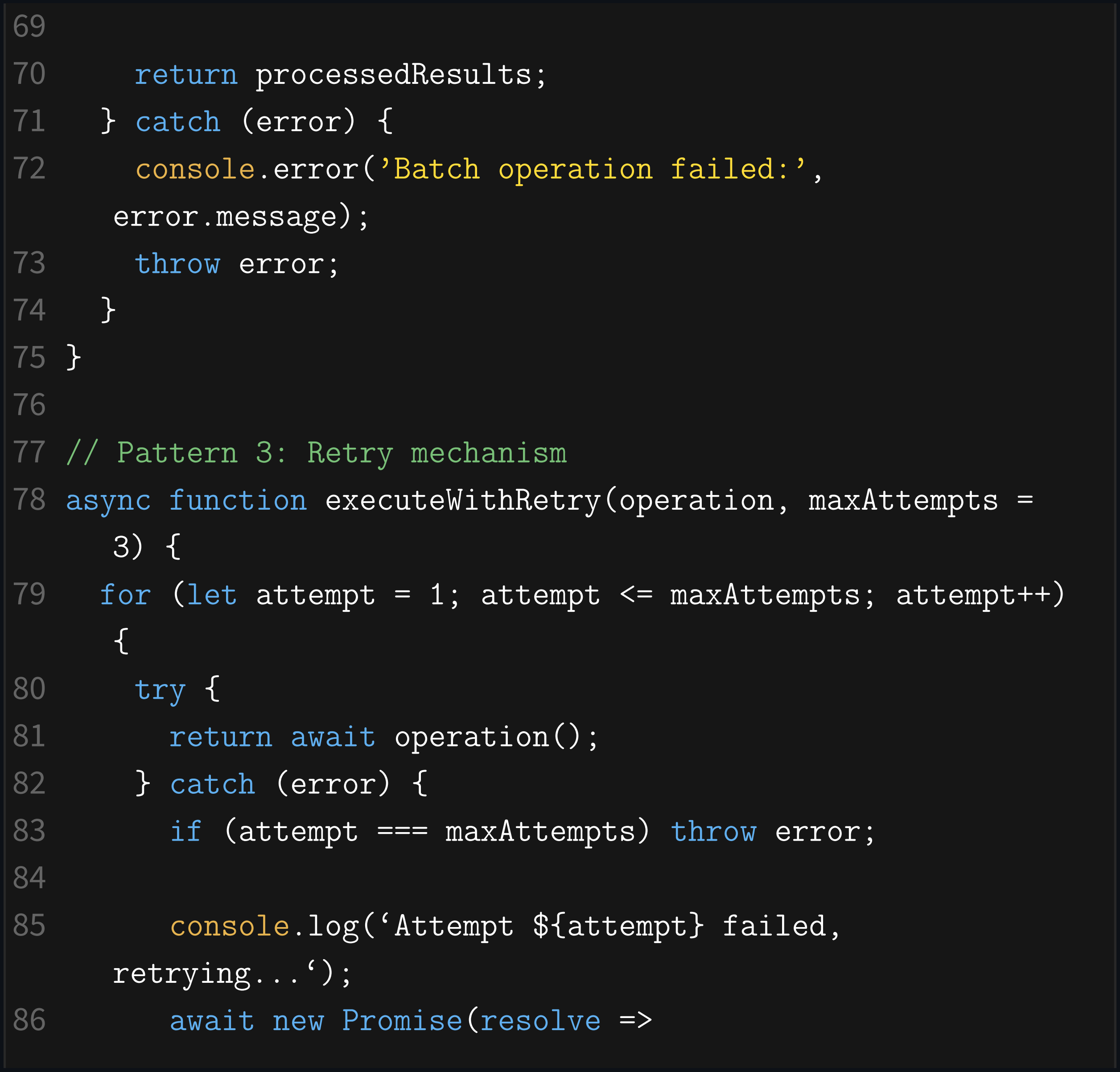
Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
🌐 www.asanchezyali.com

```
  await new Promise(resolve => setTimeout(resolve, 300));
  return results.map(r => r.toUpperCase());
}

// Pattern 1: Sequential operations with proper cleanup
async function performDatabaseOperation(query) {
  let connection = null;

  try {
    // Establish connection
    connection = await connectToDatabase();

    // Execute query
    const results = await queryDatabase(connection, query);

    // Process results
    const processedResults = await processResults(results);

    return processedResults;
  } catch (error) {
    console.error(’Operation failed:’, error.message);
    throw error; // Re-throw to let caller handle it
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```javascript
  } finally {
    if (connection) {
      console.log('Closing database connection...');
      connection.connected = false;
    }
  }
}

// Pattern 2: Parallel operations with Promise.all
async function executeMultipleQueries(queries) {
  try {
    const connection = await connectToDatabase();

    console.log('Executing queries in parallel...');
    const results = await Promise.all(
      queries.map(query => queryDatabase(connection, query))
    );

    const processedResults = await Promise.all(
      results.map(result => processResults(result))
    );
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com



```

    return processedResults;
  } catch (error) {
    console.error(’Batch operation failed:’, error.message);
    throw error;
  }
}

// Pattern 3: Retry mechanism
async function executeWithRetry(operation, maxAttempts = 3) {
  for (let attempt = 1; attempt <= maxAttempts; attempt++) {
    try {
      return await operation();
    } catch (error) {
      if (attempt === maxAttempts) throw error;

      console.log(‘Attempt ${attempt} failed, retrying...‘);
      await new Promise(resolve =>
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

NODE.JS | PROMISES

```
        setTimeout(resolve, Math.pow(2, attempt) * 100)
      );
    }
  }
}

// Demo the patterns
async function demonstratePatterns() {
  console.log(’--- Pattern 1: Sequential with Cleanup ---’);
  try {
    const results = await performDatabaseOperation(’SELECT * FROM users’);
    console.log(’Success:’, results);
  } catch (error) {
    console.log(’Handler caught:’, error.message);
  }

  console.log(’\n--- Pattern 2: Parallel Operations ---’);
  try {
    const queries = [
      ’SELECT * FROM users’,
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
    ’SELECT * FROM posts’,
    ’SELECT * FROM comments’
  ];
  const results = await executeMultipleQueries(queries);
  console.log(’Batch results:’, results);
} catch (error) {
  console.log(’Batch handler caught:’, error.message);
}

console.log(’\n--- Pattern 3: Retry Mechanism ---’);
let failCount = 0;
const unreliableOperation = async () => {
  failCount++;
  if (failCount < 3) throw new Error(’Temporary
 failure’);
  return ’Operation succeeded!’;
};

try {
  const result = await
 executeWithRetry(unreliableOperation);
  console.log(’Final result:’, result);
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
  } catch (error) {
    console.log(’Retry handler caught:’, error.message);
  }
}

demonstratePatterns();
```

These patterns demonstrate several important concepts:

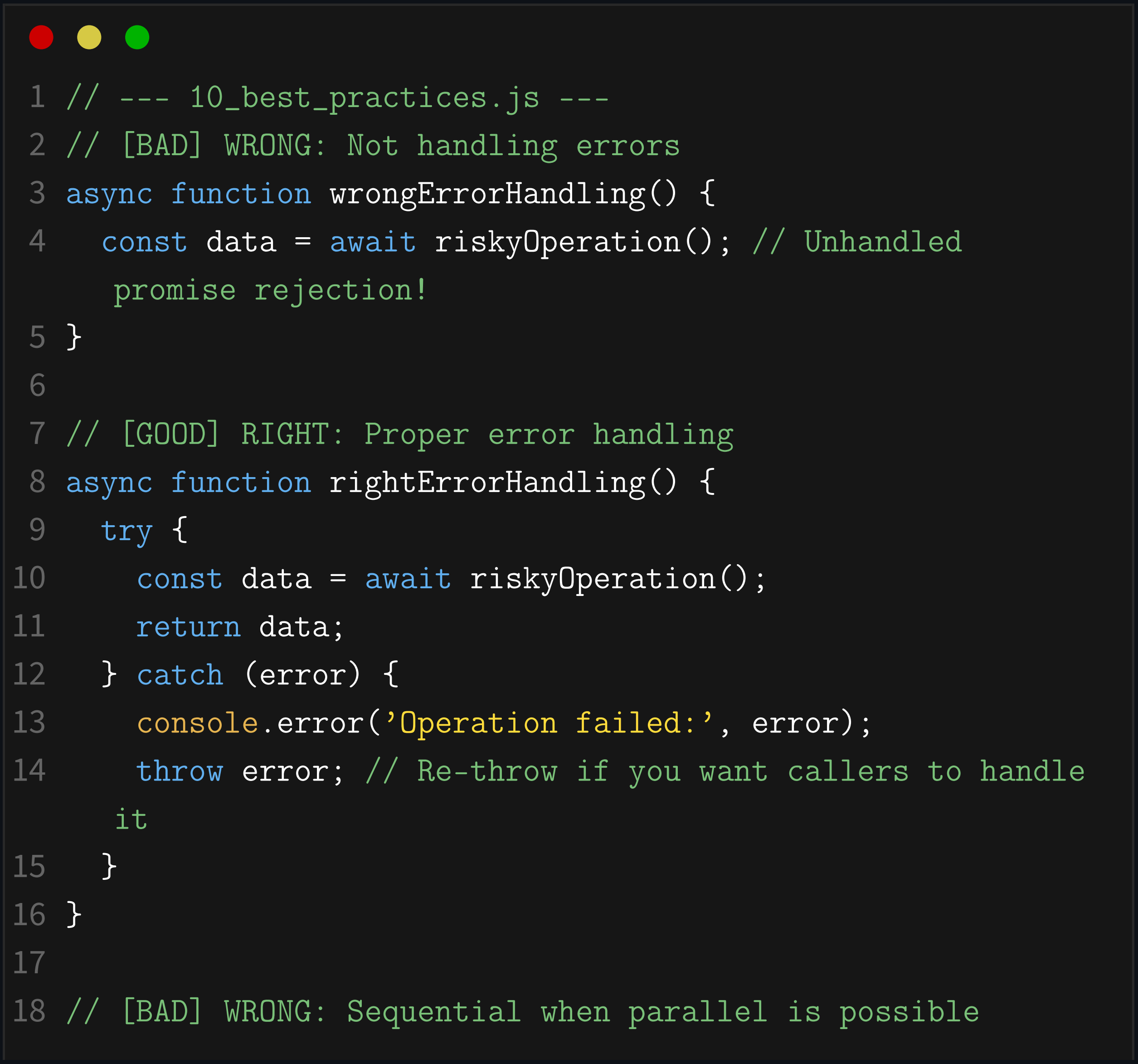
- Resource cleanup using `try/finally` blocks
- Parallel execution while maintaining error handling
- Retry mechanisms for transient failures
- Proper error propagation to calling code

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

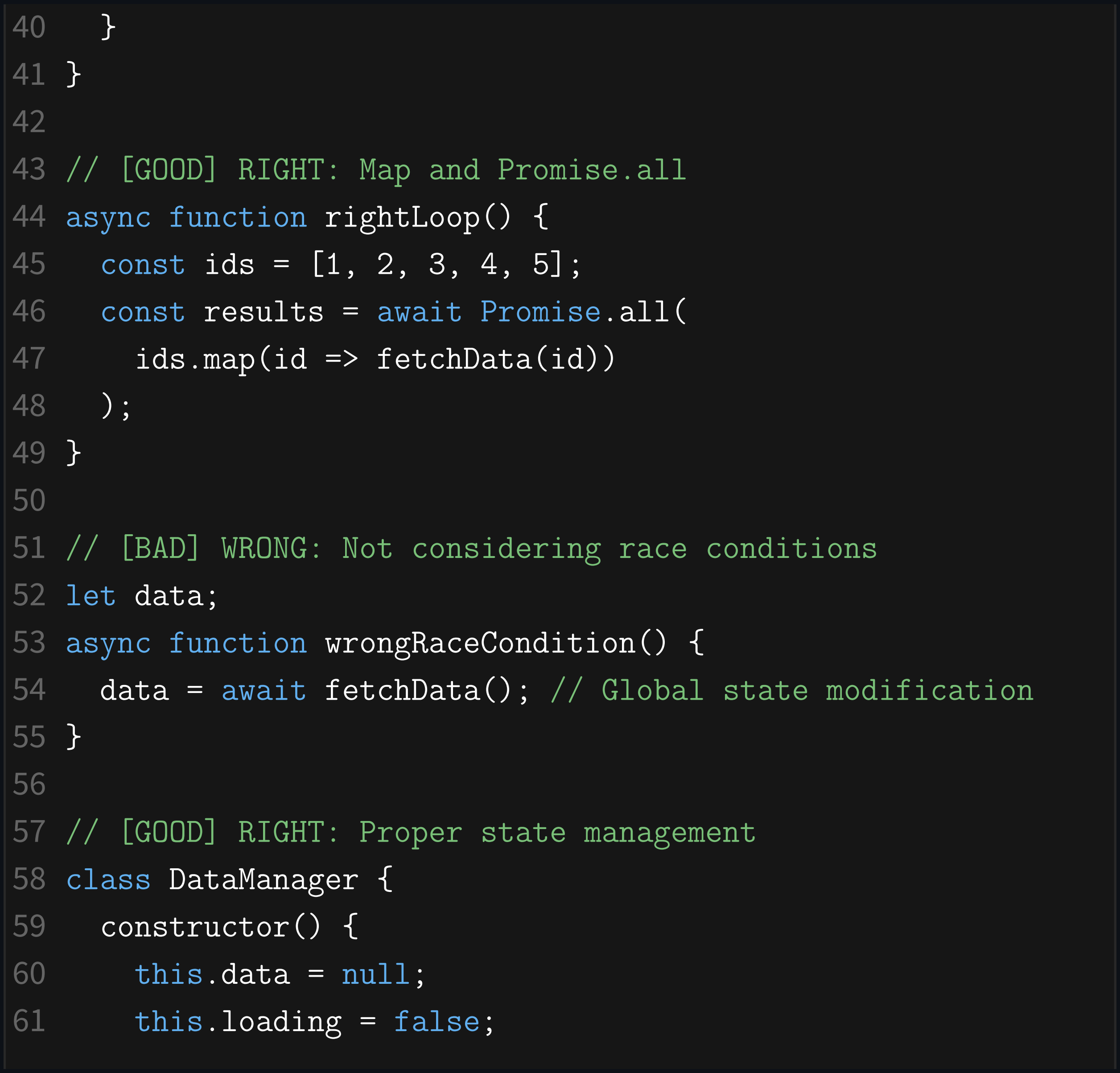
Let’s look at some common mistakes and their solutions:



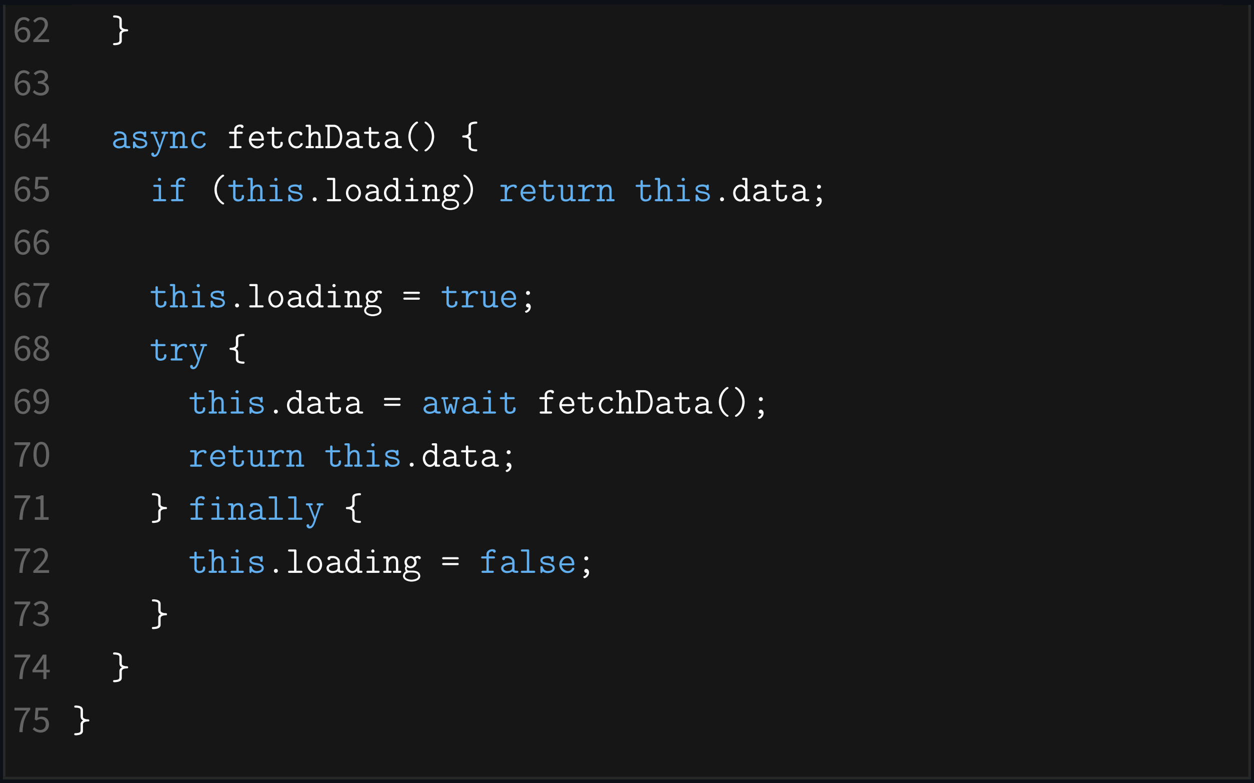
```javascript
// --- 10_best_practices.js ---
// [BAD] WRONG: Not handling errors
async function wrongErrorHandling() {
  const data = await riskyOperation(); // Unhandled promise rejection!
}

// [GOOD] RIGHT: Proper error handling
async function rightErrorHandling() {
  try {
    const data = await riskyOperation();
    return data;
  } catch (error) {
    console.error(’Operation failed:’, error);
    throw error; // Re-throw if you want callers to handle it
  }
}

// [BAD] WRONG: Sequential when parallel is possible
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```javascript
async function wrongSequential() {
  const users = await fetchUsers();
  const posts = await fetchPosts();
  const comments = await fetchComments();
}

// [GOOD] RIGHT: Parallel execution when possible
async function rightParallel() {
  const [users, posts, comments] = await Promise.all([
    fetchUsers(),
    fetchPosts(),
    fetchComments()
  ]);
}

// [BAD] WRONG: await in a loop
async function wrongLoop() {
  const ids = [1, 2, 3, 4, 5];
  const results = [];
  for (const id of ids) {
    results.push(await fetchData(id)); // Sequential execution
```



```
  }
}

// [GOOD] RIGHT: Map and Promise.all
async function rightLoop() {
  const ids = [1, 2, 3, 4, 5];
  const results = await Promise.all(
    ids.map(id => fetchData(id))
  );
}

// [BAD] WRONG: Not considering race conditions
let data;
async function wrongRaceCondition() {
  data = await fetchData(); // Global state modification
}

// [GOOD] RIGHT: Proper state management
class DataManager {
  constructor() {
    this.data = null;
    this.loading = false;
```



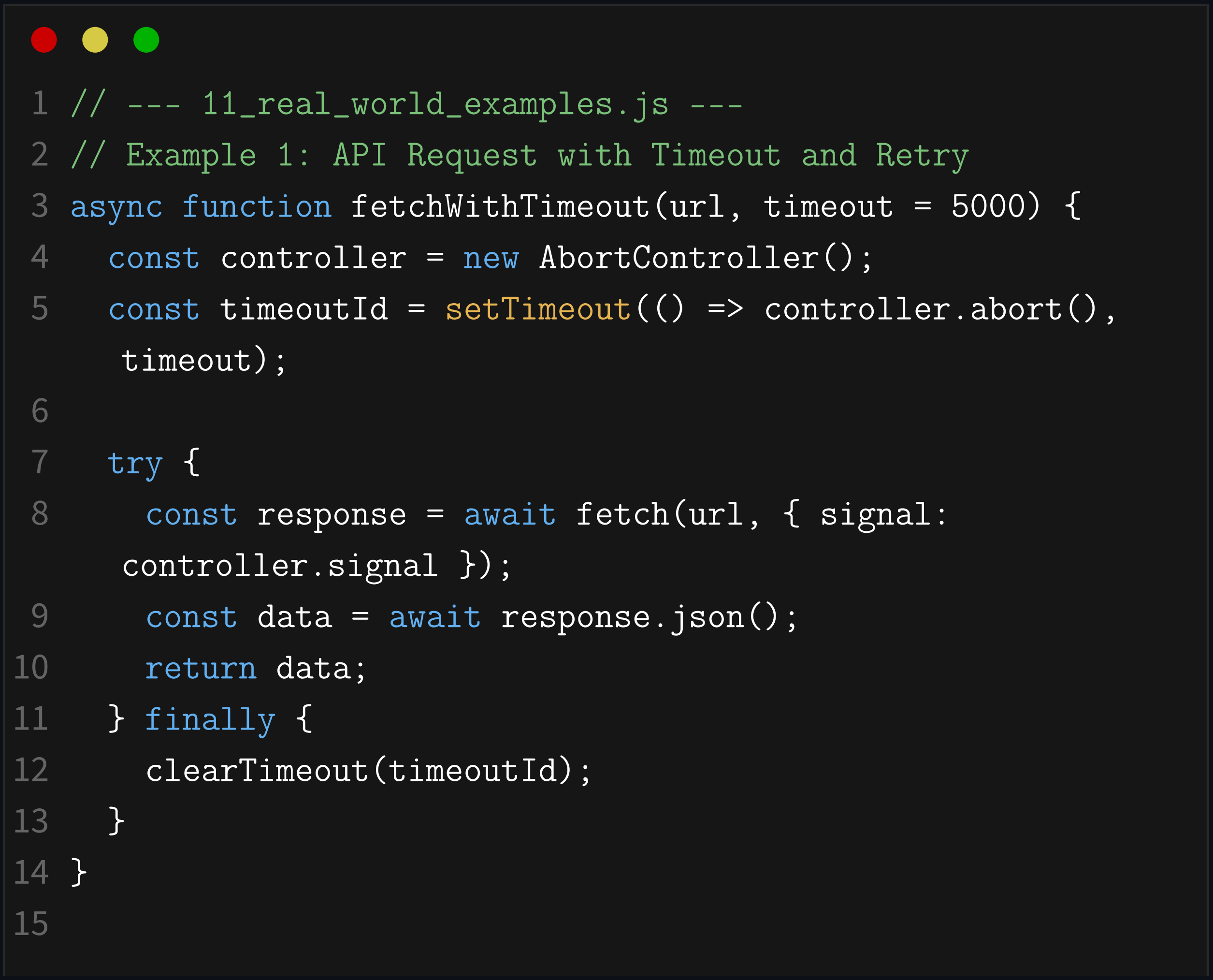
```
  }

  async fetchData() {
    if (this.loading) return this.data;

    this.loading = true;
    try {
      this.data = await fetchData();
      return this.data;
    } finally {
      this.loading = false;
    }
  }
}
```

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

3. Real-World Examples

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



```javascript
// --- 11_real_world_examples.js ---
// Example 1: API Request with Timeout and Retry
async function fetchWithTimeout(url, timeout = 5000) {
  const controller = new AbortController();
  const timeoutId = setTimeout(() => controller.abort(), timeout);

  try {
    const response = await fetch(url, { signal: controller.signal });
    const data = await response.json();
    return data;
  } finally {
    clearTimeout(timeoutId);
  }
}

```

```javascript
async function fetchWithRetry(url, retries = 3) {
  for (let i = 0; i < retries; i++) {
    try {
      return await fetchWithTimeout(url);
    } catch (error) {
      if (i === retries - 1) throw error;
      await new Promise(resolve =>
        setTimeout(resolve, Math.pow(2, i) * 1000)
      );
    }
  }
}

// Example 2: Resource Pool
class ResourcePool {
  constructor(factory, poolSize = 5) {
    this.resources = Array(poolSize).fill(null);
    this.factory = factory;
    this.available = [...Array(poolSize).keys()];
    this.waiting = [];
  }

```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
🌐 www.asanchezyali.com

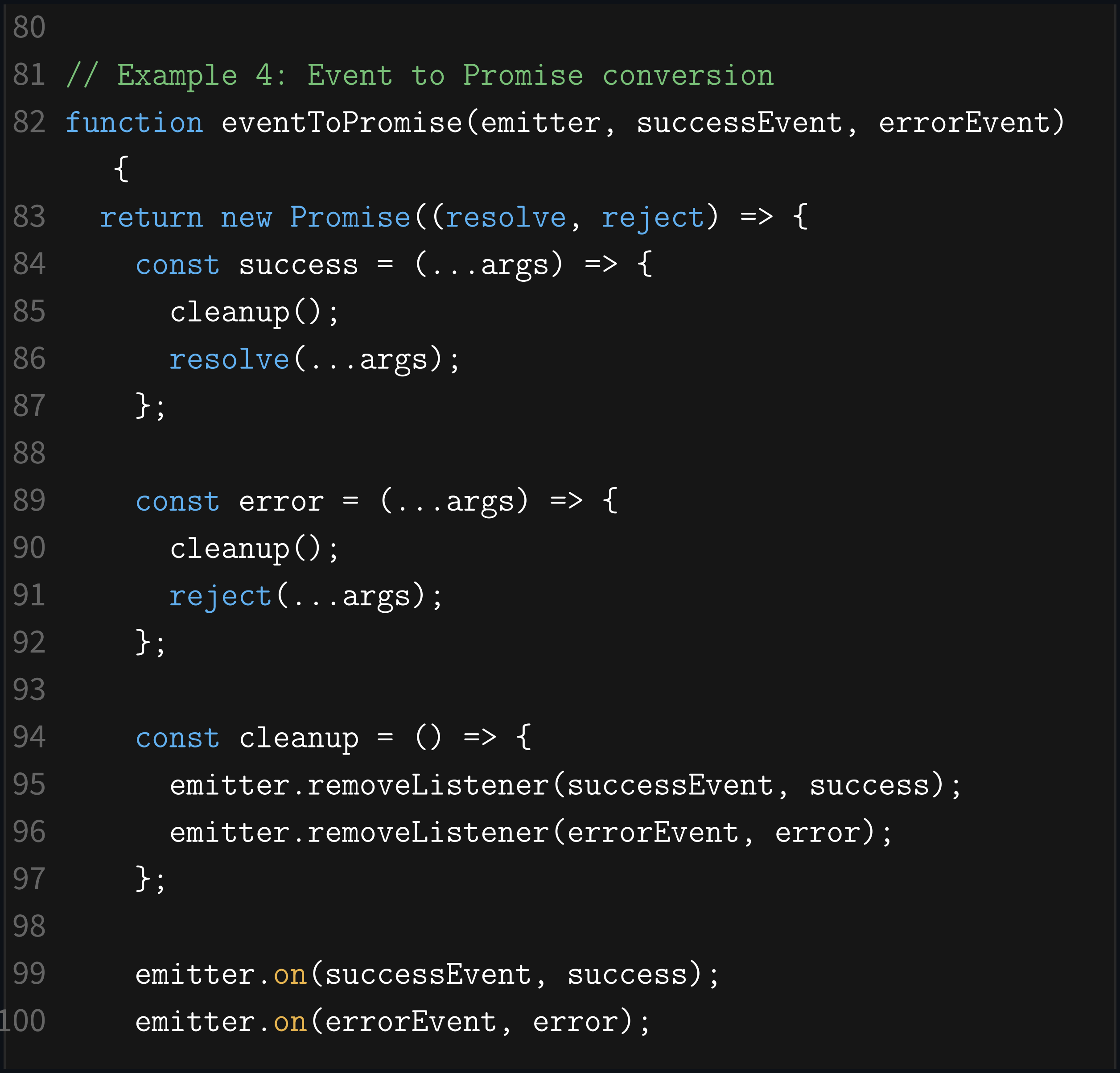
NODE.JS | PROMISES

```javascript
  async acquire() {
    if (this.available.length > 0) {
      const index = this.available.pop();
      if (!this.resources[index]) {
        this.resources[index] = await this.factory();
      }
      return { resource: this.resources[index], index };
    }

    return new Promise(resolve => {
      this.waiting.push(resolve);
    });
  }

  release({ resource, index }) {
    if (this.waiting.length > 0) {
      const resolve = this.waiting.shift();
      resolve({ resource, index });
    } else {
      this.available.push(index);
    }
  }
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```javascript
}

// Example 3: Batch Processing with Rate Limiting
async function processBatch(items, batchSize = 3, delay = 1000) {
  const results = [];

  for (let i = 0; i < items.length; i += batchSize) {
    const batch = items.slice(i, i + batchSize);
    const batchResults = await Promise.all(
      batch.map(item => processItem(item))
    );
    results.push(...batchResults);

    if (i + batchSize < items.length) {
      await new Promise(resolve => setTimeout(resolve, delay));
    }
  }

  return results;
}
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com



```javascript

// Example 4: Event to Promise conversion
function eventToPromise(emitter, successEvent, errorEvent) {
  return new Promise((resolve, reject) => {
    const success = (...args) => {
      cleanup();
      resolve(...args);
    };

    const error = (...args) => {
      cleanup();
      reject(...args);
    };

    const cleanup = () => {
      emitter.removeListener(successEvent, success);
      emitter.removeListener(errorEvent, error);
    };

    emitter.on(successEvent, success);
    emitter.on(errorEvent, error);
```

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

```
  });
}
```

These examples demonstrate:

- Combining timeouts with fetch requests
- Managing resource pools asynchronously
- 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
- **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

Alejandro Sánchez Yalí
Software Developer | AI & Blockchain Enthusiast
www.asanchezyali.com

- 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.

- 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.

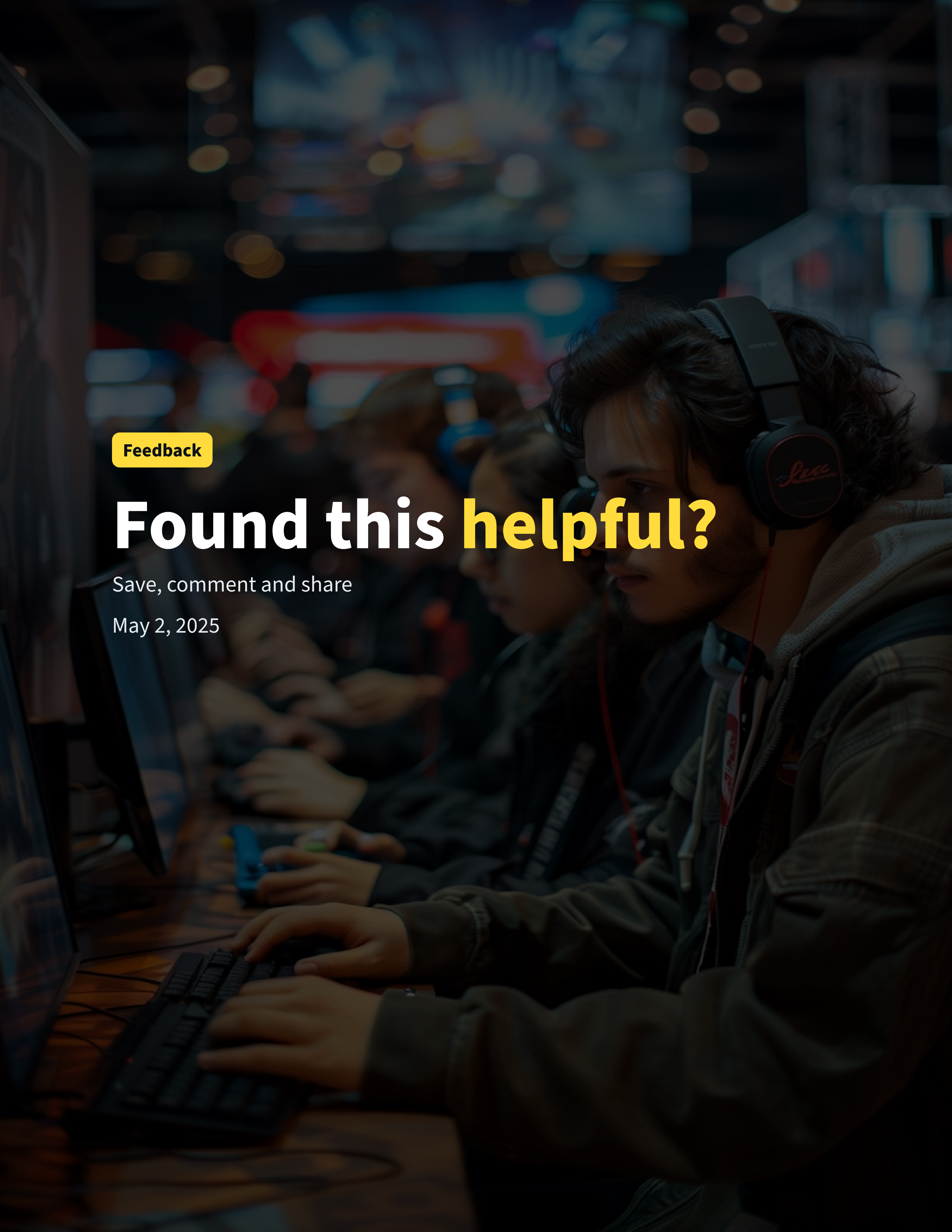
7. Explore My Other Posts

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.



Feedback

Found this helpful?

Save, comment and share

May 2, 2025