# Understanding JavaScript Promises

Handling Asynchronous Operations Like a Pro
Part 2/3

May 2, 2025

# 1. Chaining Promises: Building Complex Async Flows

In the first part of this series, we learned how to create and handle basic promises. Now, we'll dive into more advanced techniques that reveal the full potential of promises in JavaScript.

One of the most powerful features of promises is the ability to chain them together to create elegant complex asynchronous flows.

## 1.1. The Power of Chaining

When we use the `.then()` method, it returns a new promise that resolves with the value returned by the provided callback function. This allows us to chain multiple asynchronous operations sequentially, passing data from one to the next.

**How to Run:**

- Save the code as `03_promise_chaining.js`.

- Open your terminal and run: `node 03_promise_chaining.js`

```
1  // --- 03_promise_chaining.js ---
2  // Simulates retrieving a user ID from a database
3  function getUserId(username) {
4    console.log(`Looking up ID for user: ${username}...`);
5    return new Promise((resolve, reject) => {
```

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

```
 6      setTimeout(() => {
 7        // Simulates database lookup
 8        if (username.toLowerCase() === 'alejandro') {
 9          resolve(123);
10        } else {
11          reject(new Error('User not found'));
12        }
13      }, 1000);
14    });
15  }
16
17  // Simulates retrieving profile details based on an ID
18  function getUserProfile(userId) {
19    console.log('Getting profile for ID: ${userId}...');
20    return new Promise((resolve, reject) => {
21      setTimeout(() => {
22        // Simulates database lookup
23        resolve({
24          id: userId,
25          name: 'Alejandro',
26          role: 'Developer',
27          city: 'Medellin'
```

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

```
28        });
29      }, 1000);
30    });
31  }
32
33  // Simulates checking permissions for a profile
34  function checkPermissions(userProfile) {
35    console.log(`Verifying permissions for:
      ${userProfile.name}...`);
36    return new Promise((resolve, reject) => {
37      setTimeout(() => {
38        if (userProfile.role === 'Developer') {
39          resolve({
40            ...userProfile,
41            permissions: ['read', 'write', 'deploy']
42          });
43        } else {
44          resolve({
45            ...userProfile,
46            permissions: ['read']
47          });
48        }
```

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

```
49      }, 800);
50    });
51 }
52
53 // Using promise chaining
54 console.log("Starting authentication flow...");
55 getUserId('Alejandro')
56    .then(userId => {
57      console.log(`User ID found: ${userId}`);
58      return getUserProfile(userId); // Returns a new promise
59    })
60    .then(userProfile => {
61      console.log(`Profile obtained:`, userProfile);
62      return checkPermissions(userProfile); // Returns
      another promise
63    })
64    .then(profileWithPermissions => {
65      console.log(`Authentication complete!`);
66      console.log(`User ${profileWithPermissions.name} has
      permissions:
      ${profileWithPermissions.permissions.join(', ')}`);
67    })
```

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

```
68    .catch(error => {
69      console.error(`Error in process: ${error.message}`);
70    })
71    .finally(() => {
72      console.log("Authentication process finished.");
73    });
74
75  console.log("Code continues to run while promises
        resolve...");
```

**Key points:**

- Each `.then()` receives the result of the previous one and can return a new value or promise.

- Chaining is sequential: each operation waits for the previous one to complete.

- A single `.catch()` can capture errors from any part of the chain.

- Return values are handled automatically: if we return a simple value, it's automatically wrapped in a resolved promise.

## 2. Managing Multiple Promises in Parallel

In many situations, we need to execute multiple asynchronous operations simultaneously and react when they all complete or when any fails. JavaScript

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

provides static methods in the Promise class for these scenarios.

## 2.1. Promise.all(): Waiting for Everything to Complete

When we need to run multiple operations in parallel and wait for all of them to complete, `Promise.all()` is the ideal tool.
**How to Run:**

- Save the code as `04_promise_all.js`.

- Open your terminal and run: `node 04_promise_all.js`

```js
// --- 04_promise_all.js ---
// Simulates API requests for different resources
function fetchUserData(userId) {
  console.log(`Requesting user data ${userId}...`);
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`User data ${userId} received`);
      resolve({ id: userId, name: `User ${userId}`, email:
    `user${userId}@example.com` });
    }, 1000 + Math.random() * 1000); // Random time
    between 1-2 seconds
  });
```

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

```
11 }
12
13 function fetchUserPosts(userId) {
14   console.log('Requesting posts for user ${userId}...');
15   return new Promise((resolve) => {
16     setTimeout(() => {
17       console.log('Posts for user ${userId} received');
18       resolve([
19         { id: 1, title: 'Post 1 from user ${userId}' },
20         { id: 2, title: 'Post 2 from user ${userId}' }
21       ]);
22     }, 1200 + Math.random() * 1000); // Random time
       between 1.2-2.2 seconds
23   });
24 }
25
26 function fetchUserFollowers(userId) {
27   console.log('Requesting followers for user
     ${userId}...');
28   return new Promise((resolve) => {
29     setTimeout(() => {
30       console.log('Followers for user ${userId} received');
```

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

```
31          resolve([101, 102, 103]);
32        }, 800 + Math.random() * 1000); // Random time between
      0.8-1.8 seconds
33    });
34  }
35
36  console.time('Total loading time');
37
38  // Using Promise.all to execute all requests in parallel
39  const userId = 42;
40  Promise.all([
41    fetchUserData(userId),
42    fetchUserPosts(userId),
43    fetchUserFollowers(userId)
44  ])
45    .then(([userData, userPosts, userFollowers]) => {
46      console.log('\nAll data was successfully loaded!');
47      console.log('User data:', userData);
48      console.log('Posts:', userPosts);
49      console.log('Followers:', userFollowers.length);
50
51      console.timeEnd('Total loading time');
```

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

```
52   })
53   .catch(error => {
54     console.error('Error loading data:', error);
55     console.timeEnd('Total loading time');
56   });
57
58 console.log('\nRequesting all data in parallel...');
```

**Important features of Promise.all():**

- Takes an array of promises and returns a single promise.

- The resulting promise resolves with an array of all results, in the same order as the original promises.

- If **any** of the promises is rejected, `Promise.all()` is immediately rejected with that error, without waiting for the others.

- All promises execute in parallel, potentially improving performance compared to sequential execution.

## 2.2. Promise.race(): The First Winner

Sometimes, we only need the result of the first promise that completes, whether with success or error. `Promise.race()` is perfect for these cases.
**How to Run:**

- Save the code as `05_promise_race.js`.

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

● Open your terminal and run: `node 05_promise_race.js`

```javascript
// --- 05_promise_race.js ---
// Simulates searching for a product across different
   services with varying times
function searchAmazon(productName) {
  console.log(`Searching for "${productName}" on
  Amazon...`);
  return new Promise((resolve) => {
    setTimeout(() => {
      console.log(`Amazon responded`);
      resolve({
        source: 'Amazon',
        price: 29.99,
        inStock: true
      });
    }, 1500 + Math.random() * 1000); // 1.5-2.5 seconds
  });
}

function searchEbay(productName) {
```

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

```javascript
18    console.log(`Searching for "${productName}" on eBay...`);
19    return new Promise((resolve) => {
20      setTimeout(() => {
21        console.log(`eBay responded`);
22        resolve({
23          source: 'eBay',
24          price: 26.50,
25          inStock: true
26        });
27      }, 800 + Math.random() * 2000); // 0.8-2.8 seconds
28    });
29  }
30
31  function searchLocalStore(productName) {
32    console.log(`Searching for "${productName}" in local
      store...`);
33    return new Promise((resolve) => {
34      setTimeout(() => {
35        console.log(`Local store responded`);
36        resolve({
37          source: 'Local Store',
38          price: 32.99,
```

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

```
39          inStock: true
40        });
41      }, 500 + Math.random() * 1000); // 0.5-1.5 seconds
42    });
43  }
44
45  // Adding a timeout to cancel if all searches take too long
46  function timeout(ms) {
47    return new Promise((_, reject) => {
48      setTimeout(() => {
49        reject(new Error(`Timeout after ${ms}ms`));
50      }, ms);
51    });
52  }
53
54  const productName = "Bluetooth Headphones";
55  console.log(`\nSearching for the best price for:
      ${productName}`);
56  console.time('Search time');
57
58  // Race between sources and a timeout
59  Promise.race([
```

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

```
60    searchAmazon(productName),
61    searchEbay(productName),
62    searchLocalStore(productName),
63    timeout(2000) // Cancel after 2 seconds
64 ])
65    .then(result => {
66      console.log('\nWe have a winner!');
67      console.log('${result.source} was the fastest to
     respond');
68      console.log('Price: $${result.price}');
69      console.timeEnd('Search time');
70    })
71    .catch(error => {
72      console.error('\nError: ${error.message}');
73      console.timeEnd('Search time');
74    });
75
76 console.log('\nSearch in progress across all sources...');
```

**Important features of Promise.race():**

- Takes an array of promises and returns a single promise.

- The resulting promise resolves or rejects with the value of the first promise that resolves or rejects.

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

- Useful for setting timeouts, choosing the fastest data source, or implementing redundancy patterns.

- Unlike `Promise.all()`, only the first settled promise matters; the others are ignored.

# 3. Additional Promise Methods: allSettled, any

JavaScript has added additional methods to address specific use cases beyond `Promise.all()` and `Promise.race()`.

## 3.1. Promise.allSettled(): Waiting for Everything to Finish

`Promise.allSettled()` waits for all promises to complete, regardless of whether they resolve or reject, and returns an array with the results of each one. It's ideal when we need to process all results, even if some fail.

**How to Run:**

- Save the code as `06_promise_allsettled.js`.

- Open your terminal and run: `node 06_promise_allsettled.js`

```
1  // --- 06_promise_allsettled.js ---
2  // We simulate different network operations that may
       succeed or fail
```

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

```javascript
3  function fetchDataFromServer(serverId) {
4    return new Promise((resolve, reject) => {
5      const serverSuccessRate = {
6        'EU': 0.95, // 95% success rate
7        'US': 0.98, // 98% success rate
8        'ASIA': 0.85, // 85% success rate
9      };
10
11     setTimeout(() => {
12       // Simulate success or failure based on predefined
     rates
13       const success = Math.random() <
     (serverSuccessRate[serverId] || 0.5);
14       if (success) {
15         resolve({
16           serverId,
17           data: 'Data from server ${serverId}',
18           timestamp: new Date().toISOString()
19         });
20       } else {
21         reject(new Error('Error connecting to server
     ${serverId}'));
```

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

```
22        }
23      }, 1000 + Math.random() * 1000);
24    });
25 }
26
27 const serverRequests = [
28    fetchDataFromServer('EU'),
29    fetchDataFromServer('US'),
30    fetchDataFromServer('ASIA'),
31    // Adding a promise that will always fail to demonstrate
32    new Promise((_, reject) => setTimeout(() => reject(new
       Error('Server offline')), 1500)),
33 ];
34
35 console.log('Requesting data from multiple servers...\n');
36
37 Promise.allSettled(serverRequests)
38    .then(results => {
39      console.log('All requests completed (successful or
       failed)');
40
41      // Count successful and failed results
```

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

```
42      const fulfilled = results.filter(r => r.status ===
    'fulfilled');
43      const rejected = results.filter(r => r.status ===
    'rejected');
44
45      console.log('\nResults: ${fulfilled.length}
    successful, ${rejected.length} failed');
46
47      // Processing successful results
48      console.log('\nRetrieved data:');
49      fulfilled.forEach((result, index) => {
50        console.log('${index + 1}. Server
    ${result.value.serverId}: ${result.value.data}');
51      });
52
53      // Logging errors
54      console.log('\nErrors found:');
55      rejected.forEach((result, index) => {
56        console.log('${index + 1}. Error:
    ${result.reason.message}');
57      });
58    });
```

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

```
59
60  console.log('Code continues to execute while requests are
          being processed...');
```

**Important features of Promise.allSettled():**

- Waits for all promises to finish, regardless of the outcome.

- The resulting promise never rejects.

- Returns an array of objects describing the result of each promise:

  - For resolved promises: `{ status: 'fulfilled', value: result }`
  - For rejected promises: `{ status: 'rejected', reason: error }`

- Ideal for operations where we want to try everything and then process the results and errors together.

## 3.2. Promise.any(): The First Success

`Promise.any()` is similar to `Promise.race()` but only considers promises that resolve, ignoring rejected ones until all are rejected.
**How to Run:**

- Save the code as `07_promise_any.js`.

- Open your terminal and run: `node 07_promise_any.js`

**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

🌐 www.asanchezyali.com

```
1  // --- 07_promise_any.js ---
2  // Simulate loading images from different CDNs with
   varying times and reliability
3  function loadImageFromCDN(cdnName, reliability = 1.0) {
4    console.log('Attempting to load image from
     ${cdnName}...');
5    return new Promise((resolve, reject) => {
6      setTimeout(() => {
7        // Simulate success or failure based on reliability
     factor
8        const success = Math.random() <= reliability;
9
10       if (success) {
11         console.log('[SUCCESS] ${cdnName}: Image loaded
     successfully');
12         resolve({
13           cdn: cdnName,
14           url:
     'https://${cdnName.toLowerCase()}.example.com/image.jpg',
15           loadTime: Math.floor(Date.now() - startTime) +
```

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

```
         'ms'
16          });
17        } else {
18          console.log('[ERROR] ${cdnName}: Error loading
     image');
19          reject(new Error('Error loading from ${cdnName}'));
20        }
21      }, 1000 + Math.random() * 2000); // 1-3 seconds
22    });
23 }
24
25 const startTime = Date.now();
26 console.log('Starting image loading from multiple
     CDNs...\n');
27
28 // Try loading from various CDNs with different reliability
29 Promise.any([
30   loadImageFromCDN('PrimeCDN', 0.7),    // 70% reliability
31   loadImageFromCDN('FastNetwork', 0.5), // 50% reliability
32   loadImageFromCDN('BackupCDN', 0.9)    // 90% reliability
33 ])
34   .then(result => {
```

**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

🌐 www.asanchezyali.com

```
35      console.log('\n[IMAGE] Image loaded successfully!');
36      console.log('Source: ${result.cdn}');
37      console.log('URL: ${result.url}');
38      console.log('Load time: ${result.loadTime}');
39    })
40    .catch(error => {
41      // AggregateError is a new type that groups all errors
      when all promises fail
42      console.error('\n[ERROR] No CDN could load the image');
43      console.error('Errors found: ${error.errors.length}');
44      error.errors.forEach((err, i) => {
45        console.error('  ${i + 1}. ${err.message}');
46      });
47    });
48
49 console.log('Attempting to load the image...');
```

**Important features of Promise.any():**

- Resolves as soon as one of the promises resolves successfully.

- Ignores rejections until all promises are rejected.

- If all promises are rejected, it rejects with an `AggregateError` that contains all the errors.

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

- Useful when we have multiple sources for the same resource and want to use the first one that succeeds.

## 4. Conclusions from Part 2

In this second part of our series on JavaScript Promises, we've explored advanced techniques that allow managing complex asynchronous flows:

- **Promise chaining:** Allows creating sequences of ordered and clean asynchronous operations, passing data from one operation to the next.

- **Promise.all():** Executes multiple promises in parallel and waits for all of them to complete successfully, optimizing the total wait time.

- **Promise.race():** Returns the result of the first promise that settles, ideal for setting timeouts or choosing the fastest source.

- **Promise.allSettled():** Waits for all promises to complete regardless of outcome, allowing processing of both successes and failures.

- **Promise.any():** Returns the result of the first promise that resolves successfully, ignoring ones that fail until all fail.

These tools provide the necessary flexibility to implement complex and robust asynchronous patterns that can adapt to different scenarios and requirements of modern applications.

In the next part, we'll explore how promises integrate with modern `async/await` syntax, which further simplifies writing and reading asynchronous code, making it almost as intuitive as traditional synchronous code.

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

## 5. References

- MDN Web Docs. (2025). *Using Promises*. Link

- MDN Web Docs. (2025). *Promise methods*. Link

- ECMA International. (2025). *ECMAScript 2026 Language Specification: Promise Objects*. Link

- Archibald, J. (2023). *JavaScript Promises: An Introduction*. Google Developers. 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.

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

# 6. Explore My Other Posts

### Enjoyed This Content?

Don't miss my previous post about:

### Understanding JavaScript Promises:  Part 1/3

Learn the fundamental concepts of promises in JavaScript and how they help you effectively manage asynchronous operations.

**Alejandro Sánchez Yalí**

Software Developer | AI & Blockchain Enthusiast

🌐 www.asanchezyali.com

# Found this **helpful?**

Save, comment and share

May 2, 2025