Error Handling & Retry Strategy

Overview

API calls fail for various reasons: network timeouts, rate limits, server errors, authentication issues, and invalid inputs. This guide covers how to identify errors, decide whether to retry, and implement resilient integration patterns.

HTTP Status Codes

2xx - Success (No retry needed)

Code	Meaning	Action
200	OK	Request succeeded, response body included
201	Created	Resource created successfully
204	No Content	Success, no response body (DELETE, PATCH)

Action: Continue normally.

4xx - Client Errors (Don't retry)

Code	Meaning	Cause	Action
400	Bad Request	Invalid parameters, malformed JSON	Fix request, don't retry

Code	Meaning	Cause	Action
401	Unauthorized	Token expired, invalid credentials	Refresh token or re-authenticate
403	Forbidden	User lacks required permissions	Check roles/permissions
404	Not Found	Resource doesn't exist	Verify ID, don't retry
409	Conflict	Duplicate record (same email, phone)	Check for existing record

Action: Fix the request and try again. Retrying won't help.

5xx - Server Errors (Retry)

Code	Meaning	Typical Cause	Action
500	Internal Server Error	Server-side exception	Retry with backoff
502	Bad Gateway	Proxy/gateway error	Retry with backoff
503	Service Unavailable	Temporary outage, maintenance	Retry with backoff
504	Gateway Timeout	Timeout during processing	Retry with longer backoff

Action: Retry with exponential backoff.

---

429 - Rate Limit Exceeded (Retry with timing)

Code: 429
Meaning: Too many requests in time window
Response Headers:

- `X-Rate-Limit-Limit`: Max requests per window
- `X-Rate-Limit-Remaining`: Requests left
- `X-Rate-Limit-Reset`: Unix timestamp when window resets
- `Retry-After`: Seconds to wait before retrying

Action: Wait and retry. Use `Retry-After` header if present, otherwise calculate from `X-Rate-Limit-Reset`.

Retry Decision Matrix

```
Is the error...

├─ 2xx (Success)?
│  └─ No: continue
│
├─ 4xx (Client error)?
│  ├─ 401 (Auth)?
│  │  └─ Yes: Refresh token, retry once
│  ├─ 409 (Conflict)?
│  │  └─ Yes: Check for existing record, skip or update
│  └─ Other 4xx?
│     └─ No: Fix request, don't retry
│
├─ 429 (Rate limit)?
│  └─ Yes: Wait (see Retry-After), retry
│
└─ 5xx (Server error)?
   └─ Yes: Exponential backoff, retry 3-4 times
```

Exponential Backoff Pattern

Strategy

1. **First retry:** 3 seconds
2. **Second retry:** 9 seconds (3^2)
3. **Third retry:** 27 seconds (3^3)
4. **Fourth retry:** 300 seconds (5 minutes)
5. **Fifth+:** Give up

Formula: `delay = 3^(attempt_number)` capped at 5 minutes

Why Exponential?

- Gives server time to recover

- Reduces load during outages
- Prevents thundering herd (everyone retrying at same time)
- Each retry waits progressively longer

Implementation: Exponential Backoff

JavaScript/Node.js

```
/**
 * Retry logic with exponential backoff
 */
async function requestWithRetry(
  method,
  endpoint,
  body = null,
  maxAttempts = 4
) {
  const delays = [3000, 9000, 27000, 300000]; // 3s, 9s, 27s, 5min

  for (let attempt = 0; attempt < maxAttempts; attempt++) {
    try {
      const response = await makeRequest(method, endpoint, body);

      // Check status code
      if (response.ok) {
        return response; // Success
      }

      const status = response.status;

      // Determine if retryable
      const isRetryable = [429, 500, 502, 503, 504].includes(status);

      if (!isRetryable || attempt >= maxAttempts - 1) {
```

```
      // Non-retryable error or last attempt
      throw new Error(
        `API Error ${status}: ${response.statusText}`
      );
    }

    // Is 401? Try refreshing token
    if (status === 401 && attempt === 0) {
      console.log('Token expired, refreshing...');
      await refreshToken();
      // Retry immediately
      continue;
    }

    // Calculate wait time
    const delayMs = delays[attempt];
    const delaySec = delayMs / 1000;

    // Check Retry-After header
    const retryAfter = response.headers.get('Retry-After');
    if (retryAfter) {
      const waitSec = parseInt(retryAfter);
      console.warn(
        `Rate limited. Waiting ${waitSec}s before retry (attempt ${attempt + 1}/${maxAttempts})`
      );
      await sleep(waitSec * 1000);
    } else {
      console.warn(
        `Error ${status}. Retrying in ${delaySec}s (attempt ${attempt + 1}/${maxAttempts})`
      );
      await sleep(delayMs);
    }

  } catch (error) {
    // Network error (no response)
    const isRetryable = [
      'ECONNREFUSED', // Connection refused
      'ENOTFOUND',    // DNS lookup failed
      'ETIMEDOUT'     // Request timeout
    ].some(e => error.code?.includes(e));
```

```
      if (!isRetryable || attempt >= maxAttempts - 1) {
        throw error;
      }

      const delayMs = delays[attempt];
      const delaySec = delayMs / 1000;
      console.warn(
        `Network error: ${error.code}. Retrying in ${delaySec}s (attempt ${attempt + 1}/${maxAttempts})`
      );
      await sleep(delayMs);
    }
  }
}

function sleep(ms) {
  return new Promise(resolve => setTimeout(resolve, ms));
}
```

Implementation: Rate Limit Detection

```
/**
 * Monitor rate limit status
 */
async function monitorRateLimit(response) {
  const limit = parseInt(response.headers.get('X-Rate-Limit-Limit'));
  const remaining = parseInt(response.headers.get('X-Rate-Limit-Remaining'));
  const reset = parseInt(response.headers.get('X-Rate-Limit-Reset'));

  const percentRemaining = (remaining / limit) * 100;

  console.log(`Rate Limit: ${remaining}/${limit} (${percentRemaining.toFixed(1)}%)`);

  // Warning at 20% remaining
  if (percentRemaining < 20) {
```

```
    console.warn('⚠ Approaching rate limit! Slow down requests.');
  }

  // Critical at 5% remaining
  if (percentRemaining < 5) {
    console.error('□□ Critical: Rate limit nearly exceeded!');
    const waitSeconds = reset - Math.floor(Date.now() / 1000);
    console.error(`Must wait ${waitSeconds} seconds.`);
  }

  return {
    remaining,
    limit,
    percentRemaining,
    resetAt: new Date(reset * 1000)
  };
}
```

Handling Specific Errors

401 - Token Expired

```
async function handleAuthError() {
  console.log('Refreshing authentication token...');

  try {
    const newToken = await refreshAccessToken(
      process.env.GENESYS_CLIENT_ID,
      process.env.GENESYS_CLIENT_SECRET
    );

    this.accessToken = newToken;
    console.log('□  Token refreshed successfully');

    // Retry the original request
    return await makeRequest(...originalRequest);
```

```
  } catch (error) {
    console.error('❌ Token refresh failed:', error);
    throw new Error('Authentication failed. Manual re-authentication required.');
  }
}
```

409 - Duplicate Record

```
async function handleConflictError(contactData) {
  console.log('Contact may already exist. Searching...');

  try {
    const existing = await searchContact(contactData.email);

    if (existing) {
      console.log(`Found existing contact: ${existing.id}`);
      // Update instead of create
      return await updateContact(existing.id, contactData);
    } else {
      console.log('No duplicate found. Retrying create...');
      return await createContact(contactData);
    }
  } catch (error) {
    console.error('Could not resolve conflict:', error);
    throw error;
  }
}
```

429 - Rate Limit

```
async function handleRateLimit(response) {
  let waitSeconds = 60; // Default

  // Check Retry-After header first
  const retryAfter = response.headers.get('Retry-After');
  if (retryAfter) {
    waitSeconds = parseInt(retryAfter);
```

```
  } else {
    // Calculate from X-Rate-Limit-Reset
    const reset = parseInt(response.headers.get('X-Rate-Limit-Reset'));
    const now = Math.floor(Date.now() / 1000);
    waitSeconds = Math.max(1, reset - now);
  }

  console.warn(`Rate limited. Waiting ${waitSeconds}s...`);
  await sleep(waitSeconds * 1000);
  console.log('Resuming requests...');
}
```

5xx - Server Error

```
async function handleServerError(status, response) {
  if (status === 503) {
    // Service Unavailable - check Retry-After
    const retryAfter = response.headers.get('Retry-After');
    if (retryAfter) {
      const seconds = parseInt(retryAfter);
      console.warn(`Service unavailable. Retry after ${seconds}s`);
      return { retryAfter: seconds };
    }
  }

  if (status === 504) {
    // Gateway Timeout - likely temporary
    console.warn('Gateway timeout. Retrying with longer backoff...');
    return { shouldRetry: true, backoff: 'long' };
  }

  // Generic 5xx
  console.error(`Server error ${status}. Retrying...`);
  return { shouldRetry: true, backoff: 'exponential' };
}
```

Data Validation (Catch Errors Early)

Validate data BEFORE calling API to avoid 400 errors:

```
/**
 * Validate contact before creating
 */
function validateContact(contact) {
  const errors = [];

  // Required fields
  if (!contact.firstName || contact.firstName.trim() === '') {
    errors.push('firstName is required');
  }
  if (!contact.lastName || contact.lastName.trim() === '') {
    errors.push('lastName is required');
  }

  // Email format
  if (contact.email) {
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!emailRegex.test(contact.email)) {
      errors.push('email format is invalid');
    }
  }

  // Phone format (if present)
  if (contact.phoneNumbers && contact.phoneNumbers.length > 0) {
    contact.phoneNumbers.forEach((phone, i) => {
      if (!/^\+?[1-9]\d{1,14}$/.test(phone.number)) {
        errors.push(`phoneNumbers[${i}].number is not E.164 format`);
      }
      if (!['WORK', 'MOBILE', 'HOME', 'OTHER'].includes(phone.type)) {
        errors.push(`phoneNumbers[${i}].type must be WORK, MOBILE, HOME, or OTHER`);
      }
    });
```

```
  }

  return {
    valid: errors.length === 0,
    errors
  };
}

// Usage
const validation = validateContact(contactData);
if (!validation.valid) {
  console.error('Invalid contact:', validation.errors);
  return; // Don't call API
}

// Safe to call API
await createContact(contactData);
```

Idempotency: Preventing Duplicates on Retry

Use `externalId` to link records and prevent duplicates:

```
/**
 * Create contact with idempotency
 */
async function createContactIdempotent(sfContact) {
  const contactData = {
    firstName: sfContact.FirstName,
    lastName: sfContact.LastName,
    email: sfContact.Email,
    externalId: sfContact.Id  // ← Salesforce ID
  };

  try {
    return await createContact(contactData);
```

```
  } catch (error) {
    if (error.status === 409) {
      // Conflict - might already exist
      // Search by externalId
      const existing = await getContactByExternalId(sfContact.Id);
      if (existing) {
        console.log(`Contact already exists: ${existing.id}`);
        return existing;
      }
    }
    throw error;
  }
}
```

Logging & Monitoring

Log ALL API calls for debugging:

```
/**
 * Log API request and response
 */
async function loggedRequest(method, endpoint, body) {
  const startTime = Date.now();

  console.log(`[${new Date().toISOString()}] ${method} ${endpoint}`);
  if (body && method !== 'GET') {
    console.log('  Request:', JSON.stringify(body).substring(0, 200));
  }

  try {
    const response = await makeRequest(method, endpoint, body);
    const duration = Date.now() - startTime;

    console.log(`  ${response.status} (${duration}ms)`);

    // Log rate limit status
    const remaining = response.headers.get('X-Rate-Limit-Remaining');
```

```
    if (remaining) {
      console.log(`  Rate limit: ${remaining} remaining`);
    }

    return response;
  } catch (error) {
    const duration = Date.now() - startTime;
    console.error(`  ${error.status || 'NETWORK'} (${duration}ms)`);
    console.error(`  Error: ${error.message}`);
    throw error;
  }
}
```

Complete Example: Safe Contact Sync

```
/**
 * Complete contact sync with error handling
 */
async function safeSyncContact(sfContact) {
  // 1. Validate
  const validation = validateContact({
    firstName: sfContact.FirstName,
    lastName: sfContact.LastName,
    email: sfContact.Email
  });

  if (!validation.valid) {
    console.error(`Skip contact ${sfContact.Id}: ${validation.errors.join(', ')}`);
    return { status: 'SKIPPED', reason: validation.errors[0] };
  }

  // 2. Prepare
  const contactData = {
    firstName: sfContact.FirstName,
    lastName: sfContact.LastName,
```

```
    email: sfContact.Email,
    externalId: sfContact.Id  // For idempotency
  };

  // 3. Try to create with retry logic
  try {
    const result = await requestWithRetry('POST', '/contacts', contactData);
    console.log(`✅ Created: ${result.id}`);
    return { status: 'CREATED', id: result.id };
  } catch (error) {
    if (error.status === 409) {
      // Might already exist - check
      const existing = await getContactByExternalId(sfContact.Id);
      if (existing) {
        console.log(`ℹ️ Already exists: ${existing.id}`);
        return { status: 'EXISTS', id: existing.id };
      }
    }

    console.error(`❌ Failed: ${error.message}`);
    return { status: 'ERROR', reason: error.message };
  }
}
```

Best Practices

1. **Always check status codes** before retrying
2. **Use exponential backoff** for 5xx errors
3. **Respect rate limits** - check headers, slow down if needed
4. **Validate early** - catch 400 errors before calling API
5. **Use external IDs** - prevent duplicates on retry
6. **Log everything** - need data for debugging
7. **Implement timeouts** - don't wait forever
8. **Monitor rate limits** - adjust request frequency proactively

Common Mistakes

 Retrying on 400 - Invalid input won't be fixed by retrying
 Only retry on 429, 5xx

 Immediate retry on error - Doesn't fix server problems
 Wait with exponential backoff

 Creating duplicate records - No idempotency
 Use external IDs for deduplication

 Silent failures - No visibility into errors
 Log all requests and responses

 Ignoring rate limits - Keep hammering API
 Monitor headers, slow down proactively

Related Topics

- Chapter 11: API Endpoints Reference
- Chapter 11: Rate Limiting & Throttling
- Chapter 11: OAuth Client Management

Revision #1
Created 15 March 2026 00:41:58 by Cesar Gzz
Updated 15 March 2026 00:42:15 by Cesar Gzz