Rate Limiting, Token Management & Performance

API Rate Limiting

Genesys Cloud Scale:

Volume:
├─ 8+ billion API requests per week
├─ Automatically scaling infrastructure
├─ Multiple microservices (hundreds)
├─ Global deployment (multiple regions)
└─ Protection against abuse

Rate Limiting Purpose:

Protect Platform Stability:
├─ Prevent denial-of-service attacks
├─ Ensure fair access for all
├─ Protect against runaway applications
├─ Maintain performance for everyone
└─ Distribute resources fairly

Standard Rate Limits:

Authorization Code: 60 requests/minute per user session
Client Credentials: 60 requests/minute per application
SCIM Integration: 120 requests/minute per application
Enterprise: Custom limits available

Per-Microservice Limits:
├─ Each service has own limits

├─ Different services, different limits
├─ Aggregate across all services
└─ Total impact depends on mix

Detecting Rate Limits

Response Headers:

X-Rate-Limit-Limit: 60
├─ Maximum requests allowed per minute
└─ Standard: 60, SCIM: 120

X-Rate-Limit-Remaining: 42
├─ Requests remaining in current window
├─ Monitor this value
├─ When low: Start proactive management
└─ At 0: Next request will be rate limited

X-Rate-Limit-Reset: 1234567890
├─ Unix timestamp when limits reset
├─ Countdown until fresh window
└─ Use for retry calculations

Rate Limited Response:

HTTP 429 Too Many Requests

```
{
  "error": {
    "message": "Rate limit exceeded",
    "code": "RATE_LIMIT",
    "status": 429
  }
}
```

Retry-After: 60
├─ Seconds to wait before retry

├─ Recommended wait time
├─ Should respect this value
└─ Minimum wait suggested

Handling 429 Response:

1. Detect Status Code:
   ├─ Check for 429
   └─ Act immediately

2. Check Retry-After Header:
   ├─ Wait specified seconds
   └─ Example: 60 seconds

3. Implement Backoff:
   ├─ First retry: 3 seconds
   ├─ Second retry: 9 seconds
   ├─ Third retry: 27 seconds
   └─ Increment: 5-minute intervals

4. Retry Request:
   ├─ After waiting
   ├─ Same request parameters
   └─ Should succeed

5. If Still Failed:
   ├─ Contact Genesys
   ├─ Provide correlation ID
   ├─ May need custom limit
   └─ Provide usage data

Token Management

Token Lifecycle:

Creation:
├─ User authenticates

├─ Access token issued
├─ Refresh token provided (Auth Code only)
├─ Timestamp noted
└─ Token valid from this point

Active Use:
├─ Included in every API request
├─ Format: "Authorization: Bearer {token}"
├─ Server validates on each request
├─ Token proves authenticated access
└─ Scopes enforced

Expiration Tracking:

Store Expiration Time:
```
expiresAt = now + (expires_in * 1000)  // milliseconds
```

Proactive Check (Recommended):
```
if (now + 5min >= expiresAt) {
  refresh_token()  // Get new token
}
```

Reactive Check (Less Ideal):
```
if (401_response) {
  refresh_token()  // Try again
  retry_request()
}
```

Refresh Token Lifecycle:

Obtained:
├─ With access token (Authorization Code Grant)
├─ Not with Client Credentials
├─ Long-lived (30 days default, 450 days max)
└─ Stored securely

Refresh:
├─ Use refresh_token to get new access_token
├─ Happens on demand
├─ New refresh_token provided (optional)

└─ Keep updating refresh_token

Expiration:
├─ After configured duration
├─ Automatic cleanup
├─ Cannot extend manually
└─ Must re-authenticate if needed

Revocation:

On Logout:

```
DELETE /oauth/sessions/me
Authorization: Bearer {access_token}
```

Effect:
├─ Access token immediately invalid
├─ Refresh token immediately invalid
├─ User logged out
└─ New login required

Manual Revocation:
├─ Admin can delete OAuth client
├─ All tokens from that client invalid
├─ Immediate effect
└─ Audit log records action

Token Storage Best Practices:

Browser Applications:

DO:
├─ Store in memory (volatile)
├─ SessionStorage (cleared on close)
├─ Secure HTTP-only cookies
└─ Temporary locations only

DON'T:
├─ LocalStorage (persistent, exposed)
├─ Plain text
├─ Unencrypted

└─ Accessible to JavaScript

Backend Applications:

DO:
├─ Encrypted storage (database)
├─ Cache with expiration
├─ Environment variables
├─ Secure vault
└─ Limited lifetime

DON'T:
├─ Plain text
├─ Version control
├─ Logs
├─ Comments
└─ Hardcoded

Token Refresh Implementation:

JavaScript Example:
```javascript
const tokenExpiry = Date.now() + (response.expiresIn * 1000);

// Proactive refresh (recommended)
setInterval(() => {
  if (Date.now() >= tokenExpiry - 5*60*1000) {
    // Token expiring in 5 minutes
    refreshToken();
  }
}, 60000); // Check every minute

async function refreshToken() {
  const response = await fetch(tokenUrl, {
    method: 'POST',
    body: new URLSearchParams({
      grant_type: 'refresh_token',
      refresh_token: savedRefreshToken,
      ...credentials
    })
```

```
  });

  const data = await response.json();
  saveAccessToken(data.access_token);
  tokenExpiry = Date.now() + (data.expiresIn * 1000);
}
```

Python Example:

```
from datetime import datetime, timedelta

token_expiry = datetime.now() + timedelta(seconds=response['expires_in'])

# Proactive refresh
if datetime.now() >= token_expiry - timedelta(minutes=5):
    # Token expiring in 5 minutes
    refresh_token()

def refresh_token():
    response = requests.post(token_url, data={
        'grant_type': 'refresh_token',
        'refresh_token': saved_refresh_token,
        **credentials
    })

    data = response.json()
    global token_expiry
    saved_access_token = data['access_token']
    token_expiry = datetime.now() + timedelta(seconds=data['expires_in'])
```

```
---

## Performance Optimization
```

Optimization Strategy Hierarchy:

Priority 1: Use Bulk/Batch APIs ├─ Reduce 10,000 requests to 1-2 ├─ 99.99% reduction ├─ Most important optimization └─ Example: POST /conversations/batch

Priority 2: Use WebSocket Notifications ├─ Replace polling with events ├─ 99% reduction in requests ├─ Real-time data delivery └─ Subscribe to /v2/users/{id}/presence

Priority 3: Implement Caching ├─ Avoid repeat requests ├─ Cache with expiration ├─ 50-90% reduction └─ Example: Cache user list for 1 hour

Priority 4: Use Pagination ├─ Don't retrieve all records at once ├─ Request only needed fields ├─ Reduce payload size └─ Server-side filtering

Priority 5: Asynchronous Processing ├─ Don't block on API calls ├─ Queue requests ├─ Process in background └─ Better overall throughput

Specific Use Cases:

Use Case: Query 10,000 Conversations

Naive Approach: for i in 1..10000: GET /api/v2/conversations/{i} // 10,000 requests!

Problem: ├─ Rate limited at 60 requests/minute ├─ Takes 166+ minutes ├─ 99.99% inefficient └─ Cannot meet deadline

Optimized Approach: GET /api/v2/analytics/conversations/details?... // 1-2 requests!

Benefits: ├─ 1-2 requests vs 10,000 ├─ Completes in seconds ├─ No rate limiting └─ 99.99% better

Use Case: Monitor Agent Presence

Naive Approach: every 1 second: GET /api/v2/users/{id}/presence // 60 req/min per agent

Problem: ├─ 60 requests/minute per agent ├─ 100 agents = 6,000 req/min ├─ Hits rate limit immediately └─ Wasted bandwidth

Optimized Approach: WebSocket subscribe: /v2/users/{id}/presence

Benefits: ├─ Real-time event delivery ├─ 0 polling requests ├─ Lower latency ├─ No rate limiting impact └─ Event-driven architecture

Use Case: Create 10,000 Contacts

Naive Approach: for contact in contacts: POST /api/v2/externalcontacts/contacts // 10,000 requests

Problem: ├─ 10,000 individual requests ├─ Lock contention in database ├─ High API overhead ├─ Slow execution (hours) └─ Rate limiting likely

Optimized Approach: POST /api/v2/externalcontacts/contacts/bulk [array of 500 contacts] // 20 requests!

Benefits: ├─ 20 requests vs 10,000 ├─ Completes in minutes ├─ Reduced overhead ├─ No lock contention └─ Within rate limits

Field Selection Optimization:

Naive: GET /api/v2/users

Returns ALL fields (large payload)

Optimized: GET /api/v2/users?fields=id,email,name

Returns ONLY needed fields (smaller payload)

Benefits: ├─ Reduced bandwidth ├─ Faster response ├─ Lower processing └─ Better performance

Filter Server-Side:

Naive: GET /api/v2/users // Get all filter in code // Filter locally

Optimized: GET /api/v2/users?q=active:true // Server filters

Benefits: ├─ Smaller response ├─ Faster network ├─ Server-side indexes └─ Better performance

---

Backoff Strategies

Exponential Backoff Standard:

Recommended Timing: ├─ First retry: 3 seconds ├─ Second retry: 9 seconds ├─ Third retry: 27 seconds ├─ Fourth retry: 5 minutes + retry ├─ Fifth retry: 10 minutes + retry └─ Continue as needed

Real-Time Applications: ├─ Tolerance: Few retries (3-5 max) ├─ Max wait: 10-30 seconds ├─ Then: Alert user or fail gracefully └─ Example: UI interactions

Batch Applications: ├─ Tolerance: Many retries (10-20+) ├─ Max wait: Hours if needed ├─ Continue retrying: Until success └─ Example: Nightly sync jobs

Implementation:

JavaScript:

```
async function apiCallWithBackoff(url, options, maxRetries = 5) {
  for (let attempt = 0; attempt <= maxRetries; attempt++) {
    try {
      const response = await fetch(url, options);

      if (response.status === 429) {
        // Rate limited
        const retryAfter = response.headers.get('Retry-After') || 60;
        const waitTime = Math.pow(3, attempt) * 1000;
        const finalWait = Math.max(waitTime, retryAfter * 1000);

        console.log(`Rate limited, waiting ${finalWait}ms`);
        await sleep(finalWait);
        continue; // Retry
      }

      if (!response.ok) {
        throw new Error(`HTTP ${response.status}`);
      }

      return response.json(); // Success

    } catch (error) {
      if (attempt === maxRetries) {
        throw error; // Give up
      }

      const waitTime = Math.pow(3, attempt) * 1000;
      console.log(`Attempt ${attempt + 1} failed, retrying in ${waitTime}ms`);
      await sleep(waitTime);
    }
  }
}

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

Python:

```
import time
from requests.adapters import HTTPAdapter
from urllib3.util.retry import Retry

def requests_with_backoff(session, method, url, **kwargs):
    # Configure retry strategy
    retry = Retry(
        total=5,
        backoff_factor=3,  # 3, 9, 27, ... seconds
        status_forcelist=[429, 502, 503, 504]
    )

    adapter = HTTPAdapter(max_retries=retry)
    session.mount('http://', adapter)
    session.mount('https://', adapter)

    # Make request with automatic backoff
    return session.request(method, url, **kwargs)

# Usage
import requests
session = requests.Session()
response = requests_with_backoff(session, 'GET', api_url)
```

Monitoring & Alerting

What to Monitor:

Rate Limit Health: ├─ X-Rate-Limit-Remaining per request ├─ Alert if below 10 ├─ Alert if 429 responses ├─ Track pattern over time └─ Identify bottlenecks

Token Expiration: ├─ Track token age ├─ Alert on tokens near expiry ├─ Monitor refresh failures ├─ Detect refresh token issues └─ Plan proactive refreshes

API Performance: ├─ Request latency (p50, p95, p99) ├─ Response times trending ├─ Error rates by type ├─ Endpoint-specific metrics └─ Regional differences

Application Health: ├─ Authentication success rate ├─ Failed requests trend ├─ Retry frequency ├─ Unusual access patterns └─ Error type distribution

Metrics to Track:

Requests/Minute: ├─ Actual vs limit ├─ Trend over time ├─ Peak times ├─ Per endpoint └─ Per user/application

Success Rate: ├─ 2XX responses ├─ 4XX errors (auth, scope) ├─ 5XX errors (server issues) ├─ 429 rate limit └─ 401 token expired

Average Response Time: ├─ Overall ├─ By endpoint ├─ By region ├─ Trend detection └─ Outlier identification

Token Metrics: ├─ Tokens created ├─ Tokens refreshed ├─ Refresh failures ├─ Token age └─ Expiration near events

Alerting Thresholds:

Critical (Immediate): ├─ 429 rate limited for 5+ min ├─ 401 auth failures spike ├─ 503 service unavailable ├─ High error rate (>10%) └─ P99 latency spike

Warning (Within 30 min): ├─ 429 rate limited ├─ Approaching rate limit ├─ Token refresh failures ├─ Error rate increasing └─ Latency degrading

Informational (Daily): ├─ API usage report ├─ Performance summary ├─ Token refresh count ├─ Endpoint breakdown └─ Trend analysis

Dashboard Example:

Current Status: ├─ Requests/min: 45 of 60 (75%) ├─ Success rate: 99.8% ├─ Avg latency: 245ms ├─ Active tokens: 3 └─ Last refresh: 2 hours ago

Recent Alerts: ├─ None currently active ├─ Last alert: 3 days ago (resolved) └─ Next review: Today 5:00 PM

Trending: ├─ Requests/min: ↑ +5% this week ├─ Latency: ↓ -10% this month ├─ Errors: ↓ -3% this week └─ Status: Healthy

```
---

## Error Handling
```

HTTP Status Codes:

Retryable (Use Backoff): ├─ 429 Too Many Requests (rate limit) ├─ 502 Bad Gateway (temp infrastructure) ├─ 503 Service Unavailable (maintenance) ├─ 504 Gateway Timeout (temp slowness) └─ Action: Wait and retry

Client Errors (Don't Retry): ├─ 400 Bad Request (fix format) ├─ 401 Unauthorized (refresh token) ├─ 403 Forbidden (add scope/permission) ├─ 404 Not Found (resource missing) ├─ 405 Method Not Allowed (wrong HTTP verb) └─ Action: Fix and retry (or fail)

Server Errors (Usually Retryable): ├─ 500 Internal Server Error (try again) ├─ 502 Bad Gateway (usually temp) ├─ 503 Service Unavailable (usually temp) ├─ 504 Gateway Timeout (usually temp) └─ Action: Backoff and retry

Decision Tree:

Is Response Successful (2XX)? ├─ Yes → Return data, success └─ No → Check status code

Is Status 401 (Unauthorized)? ├─ Yes → Refresh token, retry └─ No → Continue

Is Status 403 (Forbidden)? ├─ Yes → Check scope/permission, error └─ No → Continue

Is Status 4XX (Other Client)? ├─ Yes → Log error, fail (don't retry) └─ No → Continue

Is Status 429 (Rate Limited)? ├─ Yes → Backoff, retry └─ No → Continue

Is Status 5XX or Other? ├─ Yes → Backoff, retry └─ No → Log, fail

Implementation:

```
def handle_api_response(response):
    if response.status_code in [200, 201, 204]:
        return response.json() if response.text else None

    elif response.status_code == 401:
        # Unauthorized - refresh token
        refresh_token()
        raise RetryException("Token refreshed, retry")

    elif response.status_code == 403:
        # Forbidden - permission/scope issue
        log_error(f"Access denied: {response.json()}")
        raise PermissionException("Insufficient permissions")

    elif response.status_code == 404:
        # Not found
```

```
        raise NotFoundException("Resource not found")

    elif response.status_code == 429:
        # Rate limited
        retry_after = response.headers.get('Retry-After', 60)
        raise RateLimitException(f"Retry after {retry_after}s")

    elif response.status_code >= 500:
        # Server error - retry
        raise ServerException(f"HTTP {response.status_code}")

    else:
        # Other error
        raise APIException(f"HTTP {response.status_code}")
```

Key Takeaways: Chapter 7

- **Rate Limits Exist** - 60 req/min standard, per application
- **Monitor Headers** - X-Rate-Limit-Remaining tells story
- **Exponential Backoff** - 3, 9, 27 second strategy
- **Token Lifespan** - 1 hour (configurable), proactive refresh recommended
- **Bulk APIs Critical** - 99.99% reduction in requests
- **WebSocket Events** - 99% reduction in polling
- **Caching Helps** - 50-90% reduction in repeat queries
- **Error Handling** - 429/5XX retry, 4XX (except 401) fail

Interview Prep

Question	Answer
Rate limit?	60 requests/minute per application (standard)
429 handling?	Exponential backoff: 3s → 9s → 27s
Token lifetime?	1 hour default (configurable 300-172,800 sec)
Token refresh?	When expiring, use refresh_token to get new
Bulk API benefit?	Reduce 10,000 requests to 1-2 (99.99% saving)
WebSocket benefit?	Replace polling, event-driven, 99% reduction

| Caching benefit? | Avoid repeat queries, 50-90% reduction |

| 401 handling? | Refresh token, get new access_token |

| 403 handling? | Add missing scope or permission, fail |

| Backoff factor? | Exponential: 3^(attempt) seconds |

Document Version

Chapter: 7 of 8

Last Updated: March 2026

Status: Current with OAuth 2.0 standards

Scope: Rate limiting, token management, performance, error handling

---

Revision #1
Created 14 March 2026 19:33:28 by Cesar Gzz
Updated 14 March 2026 19:33:42 by Cesar Gzz