11. - API & Platform Integration

API Architecture

Genesys Cloud APIs & Platform Integration Documentation

Study Notes

Topic	Description
Platform API	RESTful API for all Genesys Cloud operations
OAuth 2.0	Industry-standard authentication framework
Grant Types	Authorization Code, Client Credentials, PKCE
API Scopes	Granular permission control for applications
OAuth Clients	Application credentials and configuration
API Documentation	Developer Center with SDKs and examples
API Rate Limiting	Throttling and quota management
Web Services	Real-time event subscriptions
Integration Architecture	Multi-system connectivity patterns

Navigation

Admin → Integrations → OAuth Developer Portal: developer.genesys.cloud API Docs: https://developer.genesys.cloud/apis/rest

---

Genesys Cloud Platform API Overview

The Genesys Cloud Platform API is a comprehensive RESTful API that enables developers to programmatically manage all aspects of the Genesys Cloud environment. It provides secure, scalable access to contact center configuration, operations, analytics, and workforce management data.

API Capabilities

Platform API Scope:

Organization & Administration:
├─ Users and credentials
├─ Roles and permissions
├─ Divisions and groups
├─ Organizations and settings
└─ Custom attributes

Contact Center Configuration:
├─ Queues and routing
├─ Skills and languages
├─ Wrap-up codes and activities
├─ Schedules and schedules groups
└─ Agents and teams

Interactions & Communications:
├─ Conversations (calls, chats, emails)
├─ Call recordings and transcripts
├─ Presence and status
├─ Notifications and events
└─ Messaging and channels

Workforce Management:
├─ Forecasts and schedules
├─ Time-off requests

├─ Shift trades and exchanges
├─ Adherence and performance
└─ Capacity planning

Analytics & Reporting:
├─ Conversation analytics
├─ Performance metrics
├─ Quality evaluations
├─ Speech analytics
└─ Custom reports

Knowledge & Collaboration:
├─ Knowledge articles
├─ Canned responses
├─ Assistants and bots
└─ Bot flows

External Integration:
├─ External contacts and organizations
├─ Third-party CRM sync
├─ Data tables and imports
└─ Webhooks and subscriptions

API Architecture

REST API Design:

Base URL: https://api.[region].mypurecloud.com/api/v2

HTTP Methods:
├─ GET: Retrieve resources
├─ POST: Create resources
├─ PUT: Replace entire resource
├─ PATCH: Partial updates
└─ DELETE: Remove resources

Response Format: JSON

Authentication: OAuth 2.0 Bearer Token

Versioning: /v2/ in URL path

Rate Limiting: Requests per minute (varies by grant type)

Pagination: Cursor-based or offset-based

Error Handling: HTTP status codes + error objects

OAuth 2.0 Authentication

Genesys Cloud uses OAuth 2.0, an industry-standard authorization framework that enables secure, delegated access to resources without exposing credentials.

OAuth 2.0 Concepts

OAuth 2.0 Terminology:

Resource Owner:
├─ The user who owns the data
├─ Genesys Cloud user account
└─ Grants permission to applications

Client (Application):
├─ The application requesting access
├─ Mobile app, web app, service, bot
├─ Registered as OAuth client
└─ Has Client ID and Client Secret

Authorization Server:
├─ Genesys Cloud authentication service
├─ Validates credentials
├─ Issues access tokens
└─ Manages token lifecycle

Resource Server:

├─ Genesys Cloud Platform API
├─ Protects API resources
├─ Validates access tokens
└─ Returns authorized data

Access Token:
├─ Short-lived credential (1 hour typical)
├─ Proves authorized access
├─ Sent with every API request
├─ In Authorization header
└─ Bearer token format

Refresh Token:
├─ Long-lived credential (up to 450 days)
├─ Used to obtain new access token
├─ Never expires unless revoked
└─ Not sent with API requests

Scopes:
├─ Granular permissions
├─ Limit application access
├─ Principle of least privilege
├─ Combined as space-separated list
└─ Examples: "conversations:readonly" "users:manage"

OAuth 2.0 Grant Types

1. Authorization Code Grant (Most Secure - Recommended)

Use Case: Web applications, desktop apps, mobile apps with backend

Flow:
1. User clicks "Login with Genesys"
2. Redirected to:
   https://login.mypurecloud.com/oauth/authorize
   ?client_id=YOUR_CLIENT_ID
   &response_type=code
   &redirect_uri=https://yourapp.com/callback
   &scope=conversations:readonly+users:readonly

3. User enters Genesys Cloud credentials

4. User grants permission to application

5. Genesys redirects to callback with authorization code:
   https://yourapp.com/callback?code=AUTH_CODE

6. Backend exchanges code for token:
   POST /oauth/token
   Content-Type: application/x-www-form-urlencoded
   Authorization: Basic BASE64(client_id:client_secret)

   grant_type=authorization_code
   &code=AUTH_CODE
   &redirect_uri=https://yourapp.com/callback

7. Response:
   {
    "access_token": "abc123...",
    "token_type": "bearer",
    "expires_in": 86400,
    "refresh_token": "xyz789..."
   }

8. Backend stores token securely (NOT in browser)

9. Backend uses token to make API calls:
   GET /api/v2/users/me
   Authorization: Bearer abc123...

Advantages:
├─ Credentials never shared with app
├─ Authorization code only for callback URL
├─ Backend exchange provides server-to-server security
├─ Refresh token enables long-lived access
└─ Aligns with OAuth 2.0 best practices

Requirements:
├─ Backend server

├─ Secure token storage
├─ HTTPS only
└─ Registered redirect URI

2. Client Credentials Grant (Service-to-Service)

Use Case: Service integrations, non-user applications, scheduled jobs, background tasks

Flow:
1. Application (no user) requests token:

```
POST /oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET
&scope=conversations:readonly+users:manage
```

2. Response:

```
{
  "access_token": "abc123...",
  "token_type": "bearer",
  "expires_in": 86400
}
```

3. Application uses token immediately:

```
POST /api/v2/users
Authorization: Bearer abc123...
Content-Type: application/json

{
  "email": "newuser@company.com",
  "name": "John Doe"
}
```

Advantages:
├─ Single-step process
├─ No user involvement needed
├─ Ideal for backend services
├─ Simpler for automation

└ No refresh token needed (get new one as needed)

Limitations:
├ No user context (cannot access /me endpoints)
├ Application acts as itself, not user
├ Full permissions assigned to credentials
└ Requires secure secret storage

Common Uses:
├ Integration service (syncing Salesforce data)
├ Scheduled batch jobs (daily reports)
├ Outbound campaign system
├ Analytics aggregator
├ CRM synchronization
└ Webhook handlers

3. Authorization Code with PKCE (Enhanced Security)

Use Case: Single-Page Apps (SPAs), mobile apps, public clients

Why PKCE?
├ Public clients cannot securely store client_secret
├ Authorization code can be intercepted
├ PKCE prevents code exchange without proof
└ Recommended for all public clients

Flow:
1. Client generates random strings:
   code_verifier = random string (43-128 chars)
   code_challenge = BASE64(SHA256(code_verifier))

2. Redirect to authorization:
   https://login.mypurecloud.com/oauth/authorize
   ?client_id=YOUR_CLIENT_ID
   &response_type=code
   &redirect_uri=https://yourapp.com/callback
   &scope=conversations:readonly
   &code_challenge=HASH
   &code_challenge_method=S256

3. User authenticates and grants permission

4. Receives authorization code in callback:
   https://yourapp.com/callback?code=AUTH_CODE

5. Exchange code with PKCE proof:
   POST /oauth/token
   Content-Type: application/x-www-form-urlencoded

   grant_type=authorization_code
   &code=AUTH_CODE
   &client_id=YOUR_CLIENT_ID
   &redirect_uri=https://yourapp.com/callback
   &code_verifier=ORIGINAL_RANDOM_STRING

6. Token returned same as Authorization Code

Advantages:
├─ No client_secret needed
├─ Resistant to code interception attacks
├─ Proof of authorization ownership
├─ Modern OAuth 2.0 standard
└─ Works for public clients

Security:
├─ Only original code_verifier can exchange auth code
├─ Intercepted code cannot be used without verifier
├─ Verifier never transmitted over network
└─ Hash prevents code_verifier exposure

Important: Token Implicit Grant Deprecation
├─ Deprecated: March 2026
├─ Removed: March 2027
├─ Migration: Use Authorization Code + PKCE
└─ Action needed for existing embedded clients

4. Implicit Grant (DEPRECATED - DO NOT USE)

Status: DEPRECATED as of March 2026

⚠ Security Issues:
├─ Access token exposed in URL
├─ Browser history vulnerability
├─ No server-side security
├─ No refresh token support
└─ Vulnerable to token interception

Migration Path:
├─ Replace with Authorization Code + PKCE
├─ Deadline: May 2027 (existing clients)
└─ No new clients permitted after March 2026

DO NOT use for new applications.

OAuth Client Management

Creating an OAuth Client

Step-by-Step:

1. Navigate to Admin → Integrations → OAuth

2. Click "Add Client"

3. Configure Client:
   ├─ App Name: Your application name
   ├─ Description: What the app does
   ├─ Grant Type(s): Select appropriate type
   │ ├─ Authorization Code (with or without PKCE)
   │ └─ Client Credentials
   ├─ Authorized Redirect URIs:
   │ └─ https://yourapp.com/callback (one per line)
   ├─ Scopes: Select minimum needed
   └─ Roles: Select appropriate roles

4. System displays:

├─ Client ID (can be displayed anytime)
├─ Client Secret (ONLY shown at creation!)
├─ Creation metadata
└─ Modification history

⚠ CRITICAL: Copy Client Secret immediately!
├─ Secret not displayed again
├─ Cannot be recovered from API
├─ Store securely (never in code/git)
├─ Rotate regularly (monthly)

5. Example values:
Client ID: 1a2b3c4d-5e6f-7g8h-9i0j-1k2l3m4n5o6p
Client Secret: abc123...xyz789 (shown once!)

OAuth Client Security Best Practices

Credential Management:

Client ID:
├─ Public (can be shared)
├─ Used in browser/mobile apps
└─ Identifies application

Client Secret:
├─ CONFIDENTIAL (never share)
├─ Server-side only
├─ Store in secure vault
├─ Never commit to git
├─ Rotate regularly
└─ Regenerate if compromised

Access Token:
├─ Short-lived (1 hour)
├─ Sent with every request
├─ HTTP Authorization header only
├─ Never expose in browser
└─ Delete when done

Refresh Token:
├─ Long-lived (up to 450 days)
├─ Never sent with API requests
├─ Used to get new access token
├─ Server-side only
└─ Rotate and track

Secret Storage (Recommended):
├─ HashiCorp Vault
├─ AWS Secrets Manager
├─ Azure Key Vault
├─ Environment variables (dev only)
└─ Docker secrets (container orchestration)

Secret Rotation:
├─ Monthly minimum
├─ Before employee departures
├─ After exposure (immediately)
├─ Automated via CI/CD
└─ No application downtime needed (dual-use temporary period)

IP Whitelisting:
├─ For sensitive integrations
├─ Restrict which IPs can exchange code
├─ Requires AppxConnect IP for Salesforce
├─ Example: 177.104.46.240:443

OAuth Scopes

OAuth scopes provide granular permission control, limiting what applications can do with user data.

Scope Format

Scope Naming Convention: resource:action:scope

Examples:

```
├─ conversations:readonly       (view conversations)
├─ conversations:call:add       (make calls)
├─ users:manage                 (create/modify users)
├─ telephony:device:manage      (manage phones)
├─ routing:queue:view           (view queues)
├─ analytics:conversationDetail:view  (view call recordings)
└─ conversations:call:control   (transfer, hold, etc.)

Combining Scopes:
├─ Space-separated in requests
├─ Example: "conversations:readonly users:readonly"
├─ Authorization Code prompt shows all scopes
└─ User grants all or none

Scope Enforcement:
├─ If token lacks required scope: 403 Forbidden
├─ Each API endpoint requires specific scope
├─ Token scope limits apply, not user permissions
├─ User permissions and scopes both required (AND)
```

Common Scope Requirements

```
Conversation Management:
├─ conversations:readonly       (GET calls, emails, chats)
├─ conversations:call:add       (Place outbound calls)
├─ conversations:call:control   (Transfer, hold, disconnect)
├─ recordings:view              (Access recordings)
└─ recordings:download          (Download recording files)

User Management:
├─ users:readonly               (Read user info)
├─ users:manage                 (Create/update/delete users)
├─ authorization:grant:readonly (View user permissions)
└─ authorization:grant:manage   (Modify user permissions)

Contact Center Configuration:
├─ routing:queue:view           (View queue config)
├─ routing:queue:manage         (Modify queue config)
├─ directory:organization:view  (View org structure)
```

```
└─ telephony:phone:manage       (Manage phones)

Workforce Management:
├─ forecasting:readonly         (View forecasts)
├─ forecasting:manage           (Create/edit forecasts)
├─ scheduling:readonly          (View schedules)
├─ scheduling:manage            (Create/edit schedules)
├─ timeoff:view                 (View time-off)
└─ timeoff:manage               (Approve time-off)

Analytics & Reports:
├─ analytics:conversationDetail (View detailed analytics)
├─ analytics:aggregate:view     (View aggregated data)
├─ quality:evaluation:view      (View quality evaluations)
└─ speechanalytics:data:view    (Speech analytics)

Integrations:
├─ webhooks:manage              (Create/manage webhooks)
├─ scim:write                   (SCIM provisioning)
└─ externalcontacts:manage      (Sync external contacts)
```

API Authentication Flow

Complete Request Cycle

```
1. Authenticate (get access token)
   ├─ Authorization Code flow:
   │  ├─ User grants permission
   │  ├─ Get authorization code
   │  ├─ Exchange code + secret for token
   │  └─ Token valid for 1 hour
   │
   └─ Client Credentials flow:
      ├─ Direct credential exchange
      ├─ Token valid for 1 hour
      └─ No user involved
```

2. Use Access Token

GET /api/v2/users/me

Host: api.mypurecloud.com

Authorization: Bearer abc123xyz789

Content-Type: application/json

3. Response

HTTP/1.1 200 OK

Content-Type: application/json

{

"id": "user-123",

"name": "John Doe",

"email": "john@example.com",

"active": true

}

4. Handle Token Expiration

├─ Token expires after 1 hour

├─ API returns 401 Unauthorized

├─ Use refresh token to get new token

└─ Automatically retry request

5. Refresh Access Token

POST /oauth/token

Content-Type: application/x-www-form-urlencoded

Authorization: Basic BASE64(client_id:client_secret)

grant_type=refresh_token

&refresh_token=xyz789...

6. New Token Response

{

"access_token": "new_token_abc456...",

"token_type": "bearer",

"expires_in": 86400,

"refresh_token": "new_refresh_xyz890..."

}

7. Continue with new token
   ├─ Use new access token immediately
   ├─ Store new refresh token
   └─ Repeat cycle

Error Handling:
├─ 400 Bad Request: Invalid request format
├─ 401 Unauthorized: Invalid/expired credentials
├─ 403 Forbidden: Valid token, insufficient permissions
├─ 429 Too Many Requests: Rate limit exceeded
├─ 500 Server Error: Genesys Cloud issue
└─ 503 Service Unavailable: Maintenance mode

Token Lifecycle Management

Token Management Best Practices:

In-Memory Tokens:
├─ Store in secure memory (not localStorage)
├─ Clear when tab/window closes
├─ No persistence across sessions
└─ Suitable for SPAs with redirect flow

Backend Token Storage:
├─ Encrypt at rest
├─ Secure from SQL injection
├─ Never log token values
├─ Implement token rotation
└─ Use token expiration middleware

Refresh Token Handling:
├─ Store separately from access token
├─ Longer expiration (up to 450 days)
├─ Use to silently refresh without user action
├─ Rotate on every refresh (optional but recommended)
└─ Implement cleanup for expired tokens

Automatic Refresh:

├─ Refresh 5 minutes before expiration
├─ Detect 401 and refresh + retry
├─ Queue requests during refresh
└─ Handle race conditions (parallel requests)

Example Implementation (Node.js):

const expiresAt = Date.now() + (token.expires_in * 1000); if (expiresAt - Date.now() < 5 * 60 * 1000) { // Refresh token before expiration token = await refreshAccessToken(); }

Token Revocation:
├─ DELETE /oauth/sessions/me (revoke current token)
├─ User logout revokes all tokens
├─ No manual revocation needed normally
└─ Tokens expire automatically

API Rate Limiting & Quotas

Rate Limits (Requests Per Minute):

Grant Type:
├─ Authorization Code: 60 requests/minute per user
├─ Client Credentials: 60 requests/minute per organization
├─ SCIM: 120 requests/minute
└─ Higher limits available for enterprise customers

Example:
├─ 10 API calls @ 500ms each = uses 10 requests
├─ 60 requests/min = 1 request/second sustained
└─ 10 concurrent parallel = 10 requests counted

Handling Rate Limits:

Response Header:
X-Rate-Limit-Limit: 60
X-Rate-Limit-Remaining: 42
X-Rate-Limit-Reset: 1234567890

When Limit Exceeded:

HTTP 429 Too Many Requests

Retry-After: 60 (seconds to wait)

Implementation:

├─ Monitor X-Rate-Limit-Remaining

├─ Implement exponential backoff

├─ Queue requests if approaching limit

├─ Cache results when possible

└─ Batch operations (POST /conversations/batch)

Optimization:

├─ Use pagination (pageSize parameter)

├─ Filter results server-side (not client-side)

├─ Request only needed fields

├─ Implement caching layer

└─ Batch API calls where possible

Real-World Integration Example

Service Integration Pattern

Use Case: Sync Salesforce contacts with Genesys every hour

Architecture:

```
┌─────────────────────────────────────────────────┐
│ Node.js Service (runs hourly)                   │
├─────────────────────────────────────────────────┤
│ 1. Authenticate with Genesys (Client Credentials) │
│ 2. Fetch Salesforce contacts (Salesforce SDK)   │
│ 3. Compare with existing external contacts      │
│ 4. POST new contacts to Genesys API             │
│ 5. PATCH updates to existing contacts           │
│ 6. Log results and notify on error              │
└─────────────────────────────────────────────────┘
```

Implementation:

```
const axios = require('axios');
const salesforce = require('jsforce');

async function syncContacts() {
  try {
    // Step 1: Get Genesys access token
    const tokenResponse = await axios.post(
      'https://login.mypurecloud.com/oauth/token',
      {
        grant_type: 'client_credentials',
        client_id: process.env.GENESYS_CLIENT_ID,
        client_secret: process.env.GENESYS_CLIENT_SECRET,
        scope: 'externalcontacts:manage'
      }
    );

    const accessToken = tokenResponse.data.access_token;

    // Step 2: Get Salesforce contacts
    const conn = new salesforce.Connection({
      oauth2: { clientId, clientSecret, redirectUri }
    });

    const records = await conn.query(
      "SELECT Id, FirstName, LastName, Email, Phone FROM Contact"
    );

    // Step 3-4: Create/update in Genesys
    for (const contact of records.records) {
      const genesysContact = {
        firstName: contact.FirstName,
        lastName: contact.LastName,
        email: contact.Email,
        phone: contact.Phone,
        externalId: contact.Id // Link to Salesforce
      };
```

```
      try {
        await axios.post(
          'https://api.mypurecloud.com/api/v2/externalcontacts/contacts',
          genesysContact,
          {
            headers: {
              'Authorization': `Bearer ${accessToken}`,
              'Content-Type': 'application/json'
            }
          }
        );
      } catch (error) {
        if (error.response.status === 409) {
          // Contact exists, update instead
          await axios.patch(
            `https://api.mypurecloud.com/api/v2/externalcontacts/contacts/${contact.Id}`,
            genesysContact,
            { headers: { 'Authorization': `Bearer ${accessToken}` } }
          );
        }
      }
    }

    console.log(`Synced ${records.records.length} contacts`);
  } catch (error) {
    console.error('Sync failed:', error);
    // Send alert notification
  }
}

// Run hourly via cron job
schedule.scheduleJob('0 * * * *', syncContacts);
```

Best Practices

Security

- **Never expose client_secret** - Store in secure vault only
- **Use HTTPS** - Always for all API calls
- **Implement token rotation** - Monthly minimum
- **Validate SSL certificates** - Prevent man-in-the-middle
- **Log API calls** - For auditing and debugging (exclude tokens)
- **Scope principle of least privilege** - Grant only needed permissions

Performance

- **Implement caching** - Avoid repeated API calls
- **Use pagination** - Limit data per request
- **Batch operations** - Combine multiple updates
- **Monitor rate limits** - Respect API quotas
- **Async processing** - Handle large datasets asynchronously
- **Connection pooling** - Reuse HTTP connections

Reliability

- **Implement retry logic** - Handle temporary failures
- **Exponential backoff** - Avoid overwhelming API
- **Error handling** - Catch and log all exceptions
- **Health checks** - Monitor API availability
- **Fallback options** - Graceful degradation
- **Monitoring & alerts** - Know when things fail

Interview Cheat Sheet

Question	Answer
What's OAuth 2.0?	Industry-standard authorization framework
Grant types?	Authorization Code, Client Credentials, PKCE
When use Client Credentials?	Service-to-service, no user involved
When use Authorization Code?	Web/mobile apps with backend
What's PKCE?	Enhanced security for public clients
Scope purpose?	Granular permission control
Token lifetime?	1 hour (access), up to 450 days (refresh)
Rate limit?	60 requests/minute per credential

Question	Answer
How refresh token?	POST /oauth/token with grant_type
API base URL?	https://api.[region].mypurecloud.com/api/v2
Secret security?	Never expose, store in vault only
401 response?	Invalid/expired token, refresh required
403 response?	Valid token, missing scope/permission
429 response?	Rate limit exceeded, implement backoff
Token header?	Authorization: Bearer {access_token}

Key Takeaways

- **OAuth 2.0 Standard** - Industry-best secure authentication
- **Multiple Grant Types** - Choose based on application type
- **Scopes Control Access** - Granular permission delegation
- **Token Lifecycle** - 1-hour access, refresh tokens for persistence
- **Rate Limiting** - 60 requests/minute, monitor and respect
- **Security Critical** - Protect client secrets and access tokens
- **Error Handling** - Implement retry logic and proper error responses
- **Automation Ready** - APIs enable complete workflow automation
- **Deprecation Path** - Implicit Grant removed, migrate to PKCE
- **Enterprise Scale** - Supports millions of API calls daily

---

Additional Resources

Official Documentation

- Developer Center: https://developer.genesys.cloud/
- API Reference: https://developer.genesys.cloud/apis/rest/
- OAuth Guide: https://developer.genesys.cloud/authorization/platform-auth/
- API Explorer: https://developer.genesys.cloud/api/rest/

Support & Tools

- Genesys Community: https://community.genesys.com

- SDK Libraries: Node.js, Java, Python, Go, C#
- Postman Collection: API testing and documentation
- Technical Support: https://support.genesys.com

Document Version Info

Last Updated: March 2026
Source: Genesys Cloud Developer Documentation
Validated: Current with March 2026 releases
Version: 1.0

Genesys Cloud APIs & Platform Integration

Complete Chapter Index & Study Guide

Overview

This comprehensive study guide covers Genesys Cloud Platform API authentication, OAuth 2.0 implementation, and real-world integration patterns. All chapters are fully researched and validated against Genesys Cloud documentation as of March 2026.

Target Audience: API developers, integration engineers, platform architects deploying Genesys Cloud connectivity

Total Chapters: 8 standalone markdown files **Status**: Complete, fully researched, production-ready **Last Updated**: March 2026

Chapter Breakdown

Chapter 1: OAuth 2.0 Authentication Framework

File: `API_Chapter_01_OAuth20_Framework.md`

- What is OAuth 2.0 and why it matters
- Key terminology (Resource Owner, Client, Authorization Server)

- OAuth 2.0 concepts explained (tokens, scopes, codes)
- Comparison: OAuth vs Basic Auth
- Security principles & design
- Genesys Cloud implementation overview

Key Concepts: Authorization framework, delegated access, user consent, security-first design

Interview Topics: What is OAuth 2.0? | Three key entities? | Token vs refresh token? | Why OAuth better than Basic Auth?

Chapter 2: Authorization Code Grant

File: `API_Chapter_02_Authorization_Code_Grant.md`

- Complete step-by-step authorization code flow
- User authentication process
- Backend token exchange (server-to-server)
- Token management & refresh
- Security best practices
- Complete Node.js implementation example
- Error handling & troubleshooting

Key Concepts: Two-step process, user interaction, backend security, long-lived access via refresh tokens

Interview Topics: When use Auth Code? | Two-step flow? | Why backend exchange? | Client secret security? | How refresh tokens?

Chapter 3: Client Credentials Grant

File: `API_Chapter_03_Client_Credentials_Grant.md`

- Single-step service-to-service authentication
- Non-user applications & background jobs
- No user context (implications)
- Token acquisition & refresh
- Token duration configuration
- Python & Node.js examples
- Common use cases (Salesforce sync, reports, imports)
- Comparison with Authorization Code

Key Concepts: Service authentication, no user involved, simple flow, ideal for automation

Interview Topics: When use Client Credentials? | Single or two-step? | Refresh token included? | User context available? | Typical use cases?

Chapter 4: Authorization Code with PKCE

File: `API_Chapter_04_PKCE_Authorization_Code.md`

- Proof Key for Code Exchange (RFC 7636)
- Problem PKCE solves (authorization code interception)
- Complete PKCE flow with proof mechanism
- Code verifier & code challenge generation
- JavaScript implementation
- Implicit Grant deprecation timeline
- Migration strategy from Implicit → PKCE
- Security analysis & comparison

Key Concepts: Enhanced security, public clients, cryptographic proof, OAuth 2.0 best practices

Interview Topics: What is PKCE? | Why prevent code interception? | code_verifier vs code_challenge? | Migration deadline? | Implicit status?

Chapter 5: OAuth Scopes and Permissions

File: `API_Chapter_05_OAuth_Scopes.md`

- Granular permission control via scopes
- Scope naming convention & format
- Scope categories (conversations, users, workflows, analytics)
- Scope selection best practices (least privilege)
- Enforcement mechanism (dual validation)
- Common scope combinations
- Scope updates & lifecycle
- Testing scope-based authorization
- Troubleshooting scope issues

Key Concepts: Granular permissions, user consent, enforcement mechanism, least privilege principle

Interview Topics: What are scopes? | How combined? | User sees scopes? | How enforced? | Dual requirement (user + scope)?

Chapter 6: OAuth Client Management

File: `API_Chapter_06_OAuth_Client_Management.md`

- Step-by-step OAuth client creation
- Client secret management (March 2026 view-once change)
- Secure secret storage solutions (vaults)
- Secret rotation procedures
- Audit logging & compliance
- Client security best practices
- Client lifecycle (creation → deletion)
- Common configurations
- Troubleshooting client issues

Key Concepts: Admin-only access, secure storage required, monthly rotation, March 2026 security changes

Interview Topics: Where create clients? | Who can create? | Secret visibility? | Secret storage? | Rotation frequency? | If lost?

Chapter 7: Rate Limiting, Token Management & Performance

File: `API_Chapter_07_Rate_Limiting_Performance.md`

- API rate limiting (60 req/min standard)
- Detecting rate limits (HTTP 429)
- Exponential backoff strategies
- Token lifecycle management
- Proactive token refresh patterns
- Performance optimization techniques
- Bulk APIs (99.99% request reduction)
- WebSocket events (99% polling reduction)
- Caching strategies
- Error handling & HTTP status codes
- Monitoring & alerting

Key Concepts: Rate limits, backoff strategy, token lifecycle, performance optimization, bulk APIs

Interview Topics: Rate limit? | 429 handling? | Backoff strategy? | Token lifetime? | Bulk API benefit? | WebSocket benefit? | Error handling?

Chapter 8: Real-World Integration Patterns & Deployment

File: `API_Chapter_08_Integration_Deployment.md`

- Pattern 1: Salesforce ↔ Genesys contact sync
- Pattern 2: Nightly analytics report generation
- Pattern 3: Real-time agent status dashboard
- Development environment setup
- Staging environment configuration
- Production deployment strategy
- CI/CD pipeline design
- Secrets management in CI/CD
- Monitoring & alerting
- Disaster recovery & compliance
- Troubleshooting production issues

Key Concepts: Real-world patterns, deployment strategies, CI/CD automation, production-grade reliability

Interview Topics: Salesforce sync pattern? | Report generation? | Real-time status? | Deployment gates? | Secret storage in CI/CD? | Monitoring strategy?

Study Progression

Beginner Path

1. Chapter 1: Understand OAuth 2.0 concepts
2. Chapter 2: Learn Authorization Code flow
3. Chapter 5: Understand scopes & permissions
4. Chapter 7: Learn about rate limits & performance

Time: 4-6 hours | **Result**: Understand how OAuth works in Genesys Cloud

---

Developer Path (Building APIs)

1. Chapter 1: OAuth 2.0 framework
2. Chapter 2: Authorization Code (user-facing apps)
3. Chapter 3: Client Credentials (service integrations)
4. Chapter 6: OAuth client management
5. Chapter 7: Rate limiting & performance optimization
6. Chapter 8: Integration patterns

Time: 10-12 hours | **Result**: Ready to build and deploy API integrations

---

Advanced/Architect Path (Full Mastery)

1. All 8 chapters in sequence
2. Focus on Chapter 4 (PKCE for security)
3. Deep dive into Chapter 7 (performance)
4. Deep dive into Chapter 8 (deployment strategies)

Time: 16-20 hours | **Result**: Expert-level knowledge for API architecture & deployment

---

Key Facts (Quick Reference)

Authentication

- **OAuth 2.0 Standard**: RFC 6749 compliant
- **Grant Types**: 4 (Authorization Code, Client Credentials, PKCE, SAML2 Bearer)
- **Implicit Grant**: DEPRECATED, deadline May 2027
- **PKCE**: Recommended for public clients, already supported

Tokens

- **Access Token Lifetime**: 1 hour default (configurable 300-172,800 seconds)
- **Refresh Token Lifetime**: 30 days default (SCIM: up to 450 days)
- **Token Storage**: Vault required for production (not code/git)
- **Token Rotation**: March 2026 change - view-once-only secret

Rate Limiting

- **Standard Limit**: 60 requests/minute per application
- **Backoff Strategy**: 3s → 9s → 27s → 5-min increments
- **Platform Volume**: 8+ billion API requests/week processed
- **Optimization**: Bulk APIs reduce 99.99%, WebSockets reduce 99%

Scopes

- **Format**: resource:action (e.g., conversations:readonly)
- **Enforcement**: User permissions AND OAuth scope required (both)
- **Best Practice**: Principle of least privilege
- **Usage**: Space-separated list in requests

Security

- **Client Secret**: Store in vault (Hashicorp, AWS, Azure)
- **Rotation**: Monthly minimum, before departures, after exposure
- **HTTPS**: Always required, never HTTP
- **Audit Logging**: All authentication events logged

Deployment

- **Environments**: Development, Staging, Production (separate clients)
- **CI/CD Pipeline**: Automated build, test, deploy, rollback
- **Approval Gate**: Required for production deployment
- **Monitoring**: Critical alerts paged, high priority within 30min

---

Interview Preparation Summary

Quick Questions (Beginner)

- What is OAuth 2.0?
- Why use OAuth instead of Basic Auth?
- What are the three key entities in OAuth?
- What is the difference between access token and refresh token?

- What are scopes?

Medium Questions (Intermediate)

- Explain the Authorization Code Grant flow (steps 1-7)
- When would you use Client Credentials vs Authorization Code?
- What is PKCE and why do we need it?
- How would you handle a 429 rate limit error?
- What should you do if your OAuth client secret is lost?

Complex Questions (Advanced)

- Design a Salesforce ↔ Genesys contact sync integration
- How would you implement real-time agent status display?
- Explain your CI/CD strategy for secret management
- How would you troubleshoot a production authentication failure?
- What monitoring and alerting would you implement?

Common Scenarios & Solutions

Scenario	Solution	Chapter
Build web app with user login	Authorization Code Grant	2
Service sync Salesforce contacts	Client Credentials	3
Secure browser-based SPA	PKCE (OAuth Code variant)	4
Authenticate API requests	Check token scopes/user permissions	5
Manage OAuth clients in admin	Create, configure, rotate secrets	6
App hitting rate limits	Exponential backoff, bulk APIs, WebSockets	7
Deploy to production	CI/CD pipeline, approval gates, monitoring	8
Handle token expiration	Proactive refresh, 5min before expiry	7
Troubleshoot 403 Forbidden	Check scope AND user permission	5
Implement nightly report	Client Credentials, scheduled job, email	8

Key Skills After Completing This Guide

After studying all 8 chapters, you'll be able to:

✓ **Understand OAuth 2.0** - Know how it works and why it matters ✓ **Implement OAuth Flows** - Build authentication for any scenario ✓ **Manage OAuth Clients** - Create, configure, secure, and rotate ✓ **Handle Scopes & Permissions** - Implement granular access control ✓ **Optimize Performance** - Use bulk APIs, WebSockets, caching ✓ **Implement Error Handling** - Proper 429/401/403 responses ✓ **Design Integrations** - Real-world patterns (Salesforce, reporting, real-time) ✓ **Deploy Securely** - Production-grade CI/CD, monitoring, disaster recovery ✓ **Troubleshoot Issues** - Diagnose and fix authentication, rate limit, performance problems

---

Resources

Official Documentation

- **Genesys Developer Center**: https://developer.genesys.cloud
- **Help Center**: https://help.genesys.cloud
- **API Explorer**: https://developer.genesys.cloud/devapps/api-explorer

OAuth 2.0 Standards

- **RFC 6749** (OAuth 2.0 Authorization Framework)
- **RFC 7636** (PKCE - Proof Key for Code Exchange)
- **RFC 6750** (Bearer Token Usage)

Tools & Libraries

- **OAuth Debugger**: https://oauthdebugger.com
- **JWT Debugger**: https://jwt.io
- **Postman Collection**: Genesys Cloud API
- **SDK Libraries**: Java, JavaScript/Node.js, Python, Go, .NET, C#, iOS/Swift

---

Document Information

Item	Details
Total Chapters	8
Total Files	8 markdown documents
Estimated Study Time	16-20 hours (complete mastery)
Last Updated	March 2026
Status	Fully researched, production-ready
Validation	Against Genesys Cloud documentation
Target Audience	API developers, integration engineers, architects
Prerequisites	Basic API knowledge, familiar with HTTP/REST
Certification	Not official, internal study guide

Version History

Version	Date	Changes
2.0	March 2026	Complete rewrite, 8 chapters, full research
1.0	Original	Initial version, comprehensive coverage

How to Use This Guide

Self-Study

1. Read one chapter per study session
2. Take notes on key concepts
3. Complete interview practice questions
4. Review quick reference tables

Team Training

1. Assign chapters based on role
2. Discuss chapters in team meetings
3. Practice implementations together
4. Share troubleshooting examples

Reference

1. Quick lookup via index
2. Chapter-specific tables
3. Interview prep questions
4. Real-world patterns

Interview Preparation

1. Read all chapters once (broad understanding)
2. Review Chapter 1-3 (core OAuth)
3. Practice answers to interview questions
4. Study troubleshooting scenarios
5. Review production deployment patterns

Getting Help

If Stuck

- Review relevant chapter sections
- Check interview prep questions
- Look at real-world patterns
- Review troubleshooting sections

For Implementation Help

- Official Genesys Developer Center: https://developer.genesys.cloud
- Community Forum: https://community.genesys.com
- Support: https://support.genesys.com

For Additional Learning

- OAuth 2.0 specification (RFC 6749)
- PKCE specification (RFC 7636)
- YouTube tutorials on OAuth
- Genesys training courses

About This Study Guide

This comprehensive study guide was created as a complete reference for Genesys Cloud Platform API authentication and integration patterns. All chapters have been thoroughly researched against official Genesys Cloud documentation as of March 2026.

The guide is:

- ✓ Fully researched and validated
- ✓ Production-grade quality
- ✓ Interview preparation ready
- ✓ Real-world pattern focused
- ✓ Continuously updated

Navigation

Start Here: Chapter 1 (OAuth 2.0 Framework) **For Developers**: Chapters 2-3, then 6-8 **For Architects**: All chapters, emphasize 7-8 **For Interviews**: Chapters 1-3, then targeted by role

---

Final Notes

This study guide represents best practices for:

- OAuth 2.0 implementation
- Genesys Cloud API authentication
- Production-grade API integration
- Enterprise security standards
- Deployment & operations

Use this guide as a foundation. Always refer to current Genesys Cloud documentation for the latest updates and features.

Good luck with your API mastery journey!

Document Version

Type: Index & Study Guide
Last Updated: March 2026
Status: Complete
Chapters: 8 total
Quality: Production-ready

OAuth 2.0 Authentication Framework

Overview

Genesys Cloud's Platform API implements authorization flows described in the OAuth 2.0 standard (RFC 6749), which is an authorization framework that enables external applications to obtain limited access to HTTP services with user consent. OAuth makes a clear distinction between three key entities:

- **Resource Owner**: The Genesys Cloud user who owns the data
- **Client**: The application requesting access (your integration)
- **Authorization Server**: Genesys Cloud authentication service
- **Resource Server**: Genesys Cloud Platform API

OAuth 2.0 Concepts & Terminology

Key OAuth 2.0 Components:

Resource Owner:
├─ The user who owns the data
├─ Genesys Cloud user account
├─ Grants permission to applications
└─ Has specific permissions/roles

Client (Application):
├─ The application requesting access
├─ Mobile app, web app, service, bot
├─ Registered as OAuth client in Genesys Cloud
├─ Has Client ID and Client Secret
└─ Requests scopes during authorization

Authorization Server:
├─ Genesys Cloud authentication service
├─ Validates user credentials
├─ Issues access tokens
├─ Manages token lifecycle
└─ Enforces scope limits

Resource Server:
├─ Genesys Cloud Platform API
├─ Protects API resources
├─ Validates access tokens
├─ Enforces token scopes
└─ Returns authorized data

Access Token:
├─ Short-lived credential (1 hour typical)
├─ Proves authorized access
├─ Sent with every API request
├─ HTTP Authorization header
├─ Bearer token format: "Bearer {token}"
└─ Automatically revoked on expiration

Refresh Token:
├─ Long-lived credential (30 days to 450 days)
├─ Used to obtain new access token
├─ Never expires unless explicitly revoked
├─ Not sent with API requests
├─ Stored securely server-side
└─ Can be rotated on each refresh

Scope:
├─ Granular permission specification
├─ Limits application access
├─ Implements principle of least privilege
├─ Combined as space-separated list
├─ Examples: "conversations:readonly" "users:manage"
├─ Requested at authorization time
└─ Enforced on every API call

Authorization Code:

├─ Short-lived code (valid ~10 minutes)
├─ Single-use only
├─ Exchanged for access token
├─ Valid only with client_secret
├─ Returned via redirect URI
└─ Not the access token itself

Client ID:
├─ Public identifier for application
├─ Shared in browser/mobile clients
├─ Used in authorization requests
├─ Identifies which app is making request
└─ Can be displayed in logs

Client Secret:
├─ Confidential application credential
├─ NEVER exposed to browser/mobile
├─ Used to exchange code for token
├─ Server-side only
├─ Rotated monthly minimum
├─ Regenerate if exposed
└─ View-once-only after creation (March 2026)

Why OAuth 2.0?

Problem Solved:
├─ Users don't share passwords with applications
├─ Applications get limited, scoped access
├─ Users can revoke access without changing password
├─ Multiple applications can access same data
└─ No full account access needed

Key Advantages:
├─ Industry standard (RFC 6749)
├─ Secure delegation of access
├─ Granular permission control
├─ User consent and transparency

├─ Token-based (stateless for API)
├─ Revocation support
├─ Works across multiple protocols
└─ Widely implemented and trusted

Historical Context:
├─ OAuth 1.0: Complex signature-based (deprecated)
├─ OAuth 2.0: Token-based, simpler (current standard)
└─ OAuth 2.1: Emerging standard (future replacement)

Genesys Cloud Implementation:
├─ OAuth 2.0 standard compliant
├─ RFC 6749 Authorization Code Grant
├─ RFC 7636 PKCE (Proof Key for Code Exchange)
├─ RFC 6750 Bearer Token usage
└─ Multiple grant types supported

Comparison: Basic Auth vs OAuth

Basic Authentication (DEPRECATED):

How it works:
├─ Username and password Base64 encoded
├─ Sent with every API request
└─ No expiration

Security Issues:
├─ Hash of credentials sent every request
├─ Increases exposure of sensitive data
├─ Hash never expires (unless password changed)
├─ Anyone intercepting hash gains full access
├─ Requires user to share credentials with app
├─ User has no way to revoke without password reset
└─ No granular permission control

Status in Genesys Cloud:
├─ HTTP Basic access authentication disabled

├─ Not supported for new implementations
└─ All integrations should use OAuth

OAuth 2.0 (RECOMMENDED):

How it works:
├─ User authenticates once
├─ App receives limited-access token
├─ Token sent with API requests
├─ Token expires automatically
└─ User never shares credentials with app

Security Advantages:
├─ Credentials only shared with authorization server
├─ Access tokens are short-lived (1 hour)
├─ Tokens can be revoked independently
├─ Granular scopes limit what app can do
├─ User can revoke without password reset
├─ Different apps have different access levels
└─ Transparent to user (they control permissions)

Status in Genesys Cloud:
├─ Primary authentication mechanism
├─ Recommended for all new integrations
├─ Multiple grant types supported
└─ Security best practices enforced

OAuth 2.0 Security Principles

Core Security Concepts:

1. Never Trust the Client
├─ Always authenticate the application
├─ Verify client identity before issuing tokens
├─ Use client_secret for server-to-server
└─ Don't assume client behaves correctly

2. Limit Token Scope
├─ Grant only permissions needed
├─ Principle of least privilege
├─ Separate scopes for different functions
└─ Can revoke individual scopes if needed

3. Short Token Lifespan
├─ Access tokens: 1 hour (can be configured)
├─ Reduces window for token misuse
├─ Requires refresh token for persistence
├─ Token becomes useless after expiration
└─ Automatic cleanup reduces risk

4. Protect Sensitive Data
├─ Client_secret: Server-side only
├─ Refresh token: Secure storage required
├─ Access token: HTTP Authorization header
├─ Never log token values
└─ Encrypt at rest and in transit

5. Validate All Input
├─ Verify state parameter (CSRF protection)
├─ Validate redirect URIs
├─ Check token claims before trusting
├─ Verify signature on tokens
└─ Sanitize all user input

6. Use HTTPS Always
├─ OAuth exchanges must use HTTPS
├─ Unencrypted transmission compromises security
├─ Validate SSL/TLS certificates
├─ Protect against man-in-the-middle attacks
└─ Never fall back to HTTP

7. Prevent Code Interception
├─ PKCE adds code_verifier proof
├─ Only original verifier can exchange code
├─ Prevents authorization code theft
├─ Mandatory for public clients

└ Best practice for all clients

8. Implement Audit Logging
├ Log all authentication events
├ Track token creation/refresh
├ Monitor for anomalies
├ Never log sensitive tokens
└ Retain logs for compliance

OAuth 2.0 vs Other Standards

Comparison with Alternative Approaches:

OAuth 2.0 vs SAML 2.0:
├ OAuth: For API access, tokens
├ SAML: For authentication, assertions
├ OAuth: REST/lightweight
├ SAML: XML/enterprise
├ OAuth: Better for APIs
├ SAML: Better for enterprise SSO
├ Can be used together: SAML assertion → OAuth token

OAuth 2.0 vs OpenID Connect:
├ OAuth 2.0: Authorization only
├ OpenID Connect: OAuth 2.0 + Authentication
├ OAuth: For API access
├ OIDC: For user identity + API access
├ Can use both in same implementation
└ OIDC is superset of OAuth

OAuth 2.0 vs JWT (JSON Web Tokens):
├ OAuth: Authorization framework
├ JWT: Token format/encoding
├ OAuth defines flow/process
├ JWT is common OAuth token format
├ Can use OAuth with non-JWT tokens
├ Can use JWT outside OAuth

└─ Genesys uses Bearer tokens (could be JWT)

OAuth 2.0 vs Sessions/Cookies:
├─ OAuth: Stateless tokens (APIs)
├─ Sessions: Server-side state (web apps)
├─ OAuth: Better for distributed systems
├─ Sessions: Better for traditional web apps
├─ OAuth: No session storage needed
├─ Sessions: Requires shared session store
└─ Modern apps use both (API + web)

Genesys Cloud OAuth Implementation

Genesys Cloud OAuth Features:

Supported Grant Types:
├─ Authorization Code Grant (RECOMMENDED)
├─ Authorization Code with PKCE (BEST for public clients)
├─ Client Credentials Grant (Service-to-service)
├─ Token Implicit Grant (DEPRECATED - May 2027 deadline)
└─ SAML2 Bearer Grant (Enterprise SSO)

Token Configuration:
├─ Token duration: 300 to 172,800 seconds (default 3600)
├─ SCIM special: Up to 38,880,000 seconds (450 days)
├─ HIPAA override: 15-minute idle timeout
├─ Automatic expiration handling
└─ Refresh token support

Scope Management:
├─ 100+ available scopes
├─ Granular permission control
├─ User consent on authorization
├─ Enforced on every API call
└─ Cannot exceed user permissions

OAuth Client Creation:

├─ Admin UI: Admin → Integrations → OAuth

├─ Up to 125 authorized redirect URIs

├─ Role assignment for Client Credentials

├─ Scope selection for all grant types

├─ Division scoping for role-based access

└─ Audit trail of creation/modifications

Security Features:

├─ Client secret: View-once-only (March 2026)

├─ IP whitelisting: Available for sensitive clients

├─ Multi-factor authentication: For administrators

├─ Audit logging: All OAuth events logged

├─ Token revocation: DELETE /oauth/sessions/me

└─ Scope enforcement: Both user AND token permissions required

Multi-Tenant Support:

├─ Separate OAuth clients per tenant

├─ Tenants can't cross-authenticate

├─ Isolated token stores

├─ Per-tenant rate limits

└─ Tenant-specific configuration

Key Takeaways: Chapter 1

- **OAuth 2.0 is Standard** - Industry-best secure authorization framework
- **Three Key Entities** - Resource Owner (user), Client (app), Authorization Server
- **Five Key Concepts** - Access tokens, refresh tokens, scopes, authorization codes, client credentials
- **Security by Design** - Short-lived tokens, scope limitation, HTTPS required, credential protection
- **Industry Standard** - RFC 6749 compliant, widely implemented, trusted globally
- **Genesys Implementation** - 4+ grant types, granular scopes, audit logging, token management
- **Better than Basic Auth** - Never expose passwords, granular control, revocation support
- **Stateless & Scalable** - No session storage, works across distributed systems

---

Interview Prep: OAuth 2.0 Concepts

Question	Answer
What is OAuth 2.0?	Authorization framework enabling secure delegated access without sharing passwords
Three key entities?	Resource Owner (user), Client (app), Authorization Server (Genesys)
What's an access token?	Short-lived (1hr) proof of authorization sent with every API request
What's a refresh token?	Long-lived (30-450 days) credential used to obtain new access tokens
What are scopes?	Granular permissions limiting what an app can do (conversations:readonly, users:manage)
Why OAuth over Basic Auth?	Credentials never exposed, short-lived tokens, granular control, easy revocation
Why HTTPS required?	Unencrypted transmission exposes credentials and tokens
What is PKCE?	Proof Key for Code Exchange - prevents authorization code interception attacks
Why short token lifespan?	Reduces window for token misuse, requires refresh for persistence
How prevent CSRF?	Use state parameter in authorization code grant flow

---

Additional Resources

- **OAuth 2.0 Specification**: https://tools.ietf.org/html/rfc6749
- **PKCE (RFC 7636)**: https://tools.ietf.org/html/rfc7636
- **Bearer Token Usage (RFC 6750)**: https://tools.ietf.org/html/rfc6750
- **Genesys Developer Center**: https://developer.genesys.cloud/authorization/platform-auth/
- **Genesys Cloud Resource Center**: https://help.genesys.cloud

---

Document Version

Chapter: 1 of 8
Last Updated: March 2026
Status: Current with RFC 6749, RFC 7636, RFC 6750
Scope: OAuth 2.0 Framework, Concepts, Principles

Authorization Code Grant

Overview

The Authorization Code Grant is the most secure OAuth 2.0 grant type and is the recommended approach for web applications with backend servers and desktop applications with server components. It implements a two-step process where the user authenticates separately from the token exchange, ensuring the client_secret is never exposed to the browser or client application.

Use Cases

Perfect For:

Web Applications:
├─ Server-side API requests (ASP.NET, PHP, Node.js, Python)
├─ User login flows
├─ Backend-to-Genesys integration
└─ Sensitive data handling

Desktop Applications:
├─ Desktop clients with backend server
├─ Thin client architecture
├─ Server-side token management
└─ Secure credential storage

Mobile Applications:
├─ Mobile app + backend server pattern
├─ Backend handles OAuth exchange
├─ App never sees client_secret
└─ Server-side token refresh

NOT Ideal For:
├─ Pure browser JavaScript (no backend)

├─ Serverless functions (no client_secret storage)
├─ Mobile apps without backend
└─ Use PKCE for these cases instead

Complete Authorization Code Flow

Step 1: User Initiates Login

User clicks "Login with Genesys Cloud" button in your application

Your application redirects browser to:

```
https://login.mypurecloud.com/oauth/authorize
  ?client_id=YOUR_CLIENT_ID
  &response_type=code
  &redirect_uri=https://yourapp.com/callback
  &scope=conversations:readonly+users:readonly
  &state=random_state_string_abc123
```

Parameters:

client_id (required):
├─ Public identifier of your application
├─ Displayed during client creation
├─ Example: "1a2b3c4d-5e6f-7g8h-9i0j-1k2l3m4n5o6p"
└─ Used to identify which app is requesting access

response_type (required):
├─ Must be "code" for Authorization Code Grant
├─ Tells authorization server to return code, not token
└─ Security boundary between user auth and backend exchange

redirect_uri (required):
├─ Where user is redirected after authorization
├─ Must be EXACTLY as registered in OAuth client
├─ Must use HTTPS (not HTTP)

├─ Example: "https://yourapp.com/callback"
├─ Can include path and query parameters
└─ Must be registered in Genesys Cloud admin UI

scope (required):
├─ Space-separated list of permissions needed
├─ User sees these during authorization
├─ Example: "conversations:readonly users:readonly"
├─ Request MINIMUM scopes needed
└─ User must grant all scopes (all-or-nothing)

state (highly recommended):
├─ Random string generated by your app
├─ Prevents CSRF (Cross-Site Request Forgery) attacks
├─ Your app stores this in session
├─ Your app verifies it in callback
├─ Must be unpredictable (use crypto random)
└─ Example: Generate 32 random characters

Step 2: User Authenticates

User sees Genesys Cloud login screen

User enters:
├─ Email/username
├─ Password
└─ (Optional) Multi-factor authentication code

Genesys Cloud validates credentials

If invalid:
├─ Show error message
└─ User can retry

If valid:
├─ Genesys Cloud displays permission screen
├─ Shows app name and requested scopes
└─ User must grant permission

Step 3: Authorization Server Redirects

After user grants permission, Genesys Cloud redirects browser to your callback URI:

https://yourapp.com/callback
  ?code=AUTH_CODE_12345abcde67890
  &state=random_state_string_abc123

Parameters in Callback:

code (provided):
├─ Short-lived authorization code
├─ Valid for ~10 minutes only
├─ Single-use only (cannot reuse)
├─ Contains user and scope information
├─ Example: Long alphanumeric string
└─ NOT an access token (yet)

state (provided):
├─ Echo of state parameter from Step 1
├─ Your app must verify this matches
├─ If doesn't match: REJECT (CSRF attack)
├─ If matches: Continue with Step 4
└─ Critical security check

Error Response:
If user denies permission:

https://yourapp.com/callback
  ?error=access_denied
  &error_description=User+denied+permission
  &state=random_state_string_abc123

Handle Error:
├─ Check for "error" parameter first
├─ Don't attempt to exchange code
├─ Show user-friendly error message
└─ Offer to try again

Step 4: Backend Exchanges Code for Token

Your backend server (NOT browser) makes this request:

POST https://login.mypurecloud.com/oauth/token
Content-Type: application/x-www-form-urlencoded
Authorization: Basic BASE64(client_id:client_secret)

grant_type=authorization_code
&code=AUTH_CODE_12345abcde67890
&redirect_uri=https://yourapp.com/callback
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET

Request Details:

Authorization Header:
├─ Format: Basic BASE64(client_id:client_secret)
├─ Example: "Basic YWJjMTIzOmRlZjQ1Ng=="
├─ ALWAYS use HTTPS (not HTTP)
├─ Never expose client_secret in URL
└─ Critical for security

Body Parameters:

grant_type (required):
├─ Must be "authorization_code"
└─ Identifies which grant type you're using

code (required):
├─ Authorization code from Step 3
├─ Only valid for ~10 minutes
├─ Can only be exchanged once
├─ If reused: Server rejects with error
└─ Contains scope and user information

redirect_uri (required):

├─ Must EXACTLY match redirect_uri from Step 1
├─ Must match registered URI in OAuth client
├─ Server verifies to prevent code injection
└─ Must be HTTPS

client_id (required):
├─ Your application's public identifier
└─ Identifies which application is requesting

client_secret (required):
├─ Your application's secret
├─ Server-side only (NEVER expose)
├─ Used to prove application identity
├─ Sent in Authorization header (not URL)
└─ Should be rotated monthly

Security Note:
├─ This is server-to-server communication
├─ Client_secret is exposed only between your server and Genesys
├─ HTTPS encryption required
├─ User's browser is NOT involved
└─ User cannot see client_secret

Step 5: Receive Access & Refresh Tokens

Genesys Cloud responds with tokens:

HTTP 200 OK
Content-Type: application/json


```
{
  "access_token": "abc123xyz789...",
  "token_type": "bearer",
  "expires_in": 86400,
  "refresh_token": "refresh_xyz789...",
  "scope": "conversations:readonly users:readonly"
}
```

Response Fields:

access_token:
├─ Short-lived token (1 hour by default)
├─ Use to call Genesys Cloud APIs
├─ Include in Authorization header
├─ Example: "Bearer abc123xyz789..."
└─ Expires automatically

token_type:
├─ Always "bearer" for Genesys Cloud
├─ Indicates HTTP Bearer token format
└─ Use in header: "Authorization: Bearer {token}"

expires_in:
├─ Token lifetime in seconds
├─ Default: 3600 (1 hour)
├─ Configurable: 300-172,800 seconds
├─ Client should refresh before expiration
└─ Example: 86400 = 24 hours

refresh_token:
├─ Long-lived token (30 days default)
├─ Used to get new access token
├─ Can be up to 450 days (SCIM)
├─ Never expires unless revoked
├─ Must be stored securely
└─ Should not be exposed in browser

scope:
├─ Actual scopes granted by user
├─ May differ from requested scopes
├─ User can deny some scopes
└─ Provided for your reference

Error Response Example:
If code is invalid/expired:

HTTP 400 Bad Request

{
  "error": "invalid_grant",
  "error_description": "Authorization code expired"
}

Common Error Codes:
├─ invalid_grant: Code invalid, expired, or reused
├─ invalid_client: client_id/secret invalid
├─ invalid_request: Missing or malformed parameter
├─ server_error: Genesys Cloud error (retry)
└─ See Genesys documentation for full list

Step 6: Use Access Token

Your backend now has access token

Use to call Genesys Cloud APIs:

GET /api/v2/users/me
Host: api.mypurecloud.com
Authorization: Bearer abc123xyz789...
Content-Type: application/json

Example Response:
HTTP 200 OK
{
  "id": "user-123",
  "email": "john@example.com",
  "name": "John Doe",
  "active": true,
  "state": "active"
}

Token Usage Rules:
├─ Include in Authorization header
├─ Format: "Bearer {access_token}"
├─ Send with every API request
├─ HTTPS connection required

├─ No other authentication needed
├─ Token proves user authorized the app
└─ Genesys validates on each request

What Token Proves:
├─ User authenticated with Genesys Cloud
├─ User granted app permission
├─ User agreed to requested scopes
├─ User's identity verified
└─ Token came from real user (not forgery)

Step 7: Handle Token Expiration

After 1 hour (or configured duration):

Access token expires automatically

Your app makes request with expired token:

GET /api/v2/conversations
Authorization: Bearer abc123xyz789... (expired)

Genesys Cloud responds:

HTTP 401 Unauthorized
{
  "error": "invalid_token",
  "error_description": "Access token expired"
}

How to Handle 401:

1. Detect 401 response
2. Check if token expired
3. Use refresh_token to get new access_token
4. Retry original request with new token

Best Practice:

├─ Refresh token 5 minutes BEFORE expiration
├─ Don't wait for 401 error
├─ Proactive refresh is more efficient
└─ Monitor token expiration timestamps

Step 8: Refresh Access Token

When token expires or about to expire:

POST https://login.mypurecloud.com/oauth/token
Content-Type: application/x-www-form-urlencoded
Authorization: Basic BASE64(client_id:client_secret)

grant_type=refresh_token
&refresh_token=refresh_xyz789...
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET

Parameters:

grant_type (required):
├─ Must be "refresh_token"
└─ Different from initial "authorization_code"

refresh_token (required):
├─ Long-lived token from Step 5
├─ Never expires unless revoked
├─ Can be reused multiple times
└─ Must be stored securely

client_id & client_secret (required):
├─ Same as Step 4
├─ Authorization header or body
└─ Identifies your application

Response:

HTTP 200 OK
{

```
  "access_token": "new_token_abc456...",
  "token_type": "bearer",
  "expires_in": 86400,
  "refresh_token": "new_refresh_token_xyz890..."
}

New Tokens Provided:
├─ New access_token (1 hour lifetime)
├─ Same token_type (bearer)
├─ Same expires_in duration
├─ New refresh_token (optional, but provided)
└─ Can use immediately

Error Handling:

If refresh token invalid/expired:
├─ User must re-authenticate (start from Step 1)
└─ Cannot recover without new user authorization

If any error:
├─ Redirect user to login again
├─ Start fresh authorization flow
└─ Don't keep retrying with bad refresh_token

Refresh Token Rotation:
├─ Genesys provides new refresh_token on each refresh
├─ Old token still works briefly
├─ Provides security rotation
├─ Discard old token after refresh
└─ Never try to reuse old tokens
```

Implementation Pattern

```
High-Level Application Flow:

User Visit App
  ↓
```

User Not Logged In?

↓

Click "Login with Genesys"

↓

[Step 1-3: Browser Redirect Flow]

Authorization Server Redirects to Callback

↓

[Step 4-5: Server-to-Server Token Exchange]

Backend Exchanges Code for Tokens

↓

Backend Stores Tokens in Session/Database

↓

User Logged In to App

↓

User Makes API Call to App

↓

App Backend Uses Access Token

↓

Call Genesys Cloud API with Token

↓

Get Response from Genesys

↓

Return Result to User

↓

...Time Passes...

↓

Token Expires (1 hour)

↓

User Makes Another API Call

↓

Backend Detects Expired Token

↓

[Step 8: Refresh Token Exchange]

Backend Refreshes Token

↓

Continue with New Token

↓

Repeat as needed

Complete Node.js Example

```
const express = require('express');
const axios = require('axios');
const session = require('express-session');
const crypto = require('crypto');

const app = express();

// Configuration
const CLIENT_ID = process.env.GENESYS_CLIENT_ID;
const CLIENT_SECRET = process.env.GENESYS_CLIENT_SECRET;
const REDIRECT_URI = 'https://yourapp.com/callback';
const SCOPES = 'conversations:readonly users:readonly';
const GENESYS_REGION = process.env.GENESYS_REGION || 'mypurecloud.com';

// Session middleware
app.use(session({
  secret: 'your-secret-key',
  resave: false,
  saveUninitialized: true
}));

// Step 1: User clicks login button
app.get('/login', (req, res) => {
  // Generate state for CSRF protection
  const state = crypto.randomBytes(32).toString('hex');
  req.session.state = state;

  const authUrl = new URL(`https://login.${GENESYS_REGION}/oauth/authorize`);
  authUrl.searchParams.append('client_id', CLIENT_ID);
  authUrl.searchParams.append('response_type', 'code');
  authUrl.searchParams.append('redirect_uri', REDIRECT_URI);
  authUrl.searchParams.append('scope', SCOPES);
  authUrl.searchParams.append('state', state);

  res.redirect(authUrl.toString());
});
```

```
// Step 3-4: Handle callback and exchange code
app.get('/callback', async (req, res) => {
  const { code, state, error } = req.query;

  // Check for errors
  if (error) {
    console.error('Authorization error:', error);
    return res.status(400).send('Authorization denied');
  }

  // Verify state (CSRF protection)
  if (state !== req.session.state) {
    return res.status(403).send('Invalid state parameter');
  }

  try {
    // Step 4: Exchange code for tokens
    const response = await axios.post(
      `https://login.${GENESYS_REGION}/oauth/token`,
      {
        grant_type: 'authorization_code',
        code: code,
        redirect_uri: REDIRECT_URI,
        client_id: CLIENT_ID,
        client_secret: CLIENT_SECRET
      }
    );

    // Step 5: Store tokens
    req.session.accessToken = response.data.access_token;
    req.session.refreshToken = response.data.refresh_token;
    req.session.expiresAt = Date.now() + (response.data.expires_in * 1000);

    res.redirect('/dashboard');
  } catch (error) {
    console.error('Token exchange error:', error.message);
    res.status(500).send('Token exchange failed');
  }
});
```

```
// Helper function to ensure valid access token
async function ensureAccessToken(req) {
  // Check if token needs refresh (within 5 minutes of expiration)
  if (req.session.expiresAt - Date.now() < 5 * 60 * 1000) {
    try {
      // Step 8: Refresh token
      const response = await axios.post(
        `https://login.${GENESYS_REGION}/oauth/token`,
        {
          grant_type: 'refresh_token',
          refresh_token: req.session.refreshToken,
          client_id: CLIENT_ID,
          client_secret: CLIENT_SECRET
        }
      );

      // Update tokens
      req.session.accessToken = response.data.access_token;
      req.session.refreshToken = response.data.refresh_token;
      req.session.expiresAt = Date.now() + (response.data.expires_in * 1000);
    } catch (error) {
      console.error('Token refresh failed:', error.message);
      throw new Error('Failed to refresh token');
    }
  }

  return req.session.accessToken;
}

// Step 6: Use access token
app.get('/api/conversations', async (req, res) => {
  try {
    const accessToken = await ensureAccessToken(req);

    // Step 6: Use token to call Genesys API
    const response = await axios.get(
      `https://api.${GENESYS_REGION}/api/v2/conversations`,
      {
        headers: {
```

```
          'Authorization': `Bearer ${accessToken}`,
          'Content-Type': 'application/json'
        }
      }
    );

    res.json(response.data);
  } catch (error) {
    console.error('API call failed:', error.message);
    res.status(500).send('Failed to fetch conversations');
  }
});

// Logout
app.get('/logout', (req, res) => {
  // Optional: Revoke token on Genesys side
  // DELETE /oauth/sessions/me with access token

  req.session.destroy((err) => {
    if (err) console.error('Session destroy error:', err);
    res.redirect('/');
  });
});

app.listen(3000, () => console.log('Server running on port 3000'));
```

Security Checklist

Authorization Code Grant Security:

☐ Use HTTPS everywhere (never HTTP)
☐ Validate SSL/TLS certificates
☐ Generate unpredictable state parameter
☐ Verify state in callback (CSRF protection)
☐ Store client_secret securely (vault/environment)
☐ Never expose client_secret in browser
☐ Never commit secrets to git

☐ Exchange code on backend (never frontend)
☐ Validate redirect_uri matches registered
☐ Store tokens securely server-side
☐ Refresh token before expiration
☐ Implement 401 handling (refresh + retry)
☐ Never log token values
☐ Implement audit logging (no tokens)
☐ Rotate secrets monthly
☐ Revoke tokens on logout
☐ Implement HTTPS redirect
☐ Validate SSL certificate
☐ Handle errors gracefully
☐ Implement rate limiting
☐ Monitor for suspicious activity

Comparison with Other Grant Types

Authorization Code vs:

Authorization Code + PKCE:
├─ Both equally secure for web apps
├─ PKCE added for public clients (SPAs, mobile)
├─ Both send user to browser for authentication
├─ PKCE adds code_verifier to prevent interception
└─ Code is better for web apps, PKCE for public

Client Credentials:
├─ Both are OAuth 2.0 standard grants
├─ Code requires user interaction
├─ Client Credentials: Service-to-service
├─ Code: User-initiated access
├─ Code: User sees permission screen
├─ Client Credentials: No user consent needed
└─ Different use cases

Implicit Grant (DEPRECATED):
├─ Both result in access token
├─ Code: Two-step (safer)
├─ Implicit: One-step (simpler but risky)
├─ Code: Client_secret protected
├─ Implicit: No secret possible
├─ Code: Token in backend
├─ Implicit: Token in URL/browser
├─ Code is clearly better (implicit deprecated)
└─ Never use Implicit for new apps

Common Errors & Solutions

Error: "invalid_grant"
├─ Cause: Authorization code invalid/expired/reused
├─ Solution: User must re-authenticate
├─ Timeline: Code valid ~10 minutes
└─ Prevention: Use code immediately

Error: "invalid_client"
├─ Cause: client_id or client_secret invalid
├─ Solution: Verify credentials in OAuth client
├─ Check: Admin → Integrations → OAuth
└─ Action: Regenerate credentials if needed

Error: "redirect_uri_mismatch"
├─ Cause: Callback URI doesn't match registered
├─ Solution: Ensure EXACT match (case-sensitive)
├─ Check: Admin → Integrations → OAuth
└─ Include: Protocol, domain, path, query params

Error: "invalid_scope"
├─ Cause: Requested scope doesn't exist
├─ Solution: Verify scope name is correct
└─ Use: Format "resource:action" or "resource:action:scope"

Error: "access_denied"

├─ Cause: User denied permission
├─ Solution: Show friendly message, offer retry
└─ Expected: Normal user action

Error: "server_error"
├─ Cause: Genesys Cloud temporary error
├─ Solution: Retry with exponential backoff
└─ Contact: Support if persists

Key Takeaways: Chapter 2

- **Most Secure Grant** - Recommended for web and desktop applications
- **Two-Step Process** - Separates user authentication from token exchange
- **Client Secret Protected** - Never exposed to browser or client application
- **Short-Lived Tokens** - Access tokens expire (1 hour by default)
- **Refresh Capability** - Refresh tokens enable long-lived access without re-authentication
- **CSRF Protected** - State parameter prevents cross-site attacks
- **Server-Side Exchange** - Code exchanged on backend (never browser)
- **Standard Approach** - RFC 6749 compliant, industry best practice

---

Interview Prep: Authorization Code Grant

Question	Answer
When use Auth Code?	Web/desktop apps with backend server
Two-step process?	User auth separate from token exchange
Authorization code purpose?	Single-use code exchanged for access token
State parameter?	CSRF protection (random string verified in callback)
Why exchange on backend?	Keep client_secret secure (never exposed)
Client_secret exposure?	Never - only used between server and Genesys
Access token lifetime?	1 hour by default (configurable 300-172,800 sec)
Refresh token lifetime?	30 days default (up to 450 days for SCIM)

Question	Answer
401 error handling?	Refresh token to get new access token, retry request
Rate limiting?	60 requests/minute per application

Document Version

Chapter: 2 of 8
Last Updated: March 2026
Status: Current with RFC 6749
Scope: Authorization Code Grant Flow, Implementation, Security

Client Credentials Grant

Overview

The Client Credentials Grant is a single-step OAuth 2.0 grant type designed exclusively for non-user applications and service-to-service authentication. Unlike the Authorization Code Grant which requires user interaction, Client Credentials enables applications to authenticate directly using their own credentials without any user context.

Use Cases

Perfect For:

Background Jobs & Scheduled Tasks:
├─ Nightly batch processing
├─ Cron jobs
├─ Scheduled reports
├─ Database cleanup tasks
└─ No user interaction needed

Service-to-Service Integration:
├─ Backend service to Genesys Cloud API
├─ Microservice communication
├─ Application-to-application
├─ System integrations
└─ Automated workflows

Bots & Virtual Agents:
├─ Chatbot integrations
├─ Virtual agent frameworks
├─ Automation engines
├─ API consumers without users
└─ Standalone applications

Data Synchronization:
├─ Salesforce ↔ Genesys sync
├─ Contact import/export
├─ Bulk data operations
├─ Third-party CRM integration
└─ ETL (Extract-Transform-Load) processes

NOT Suitable For:

User-Initiated Access:
├─ Web applications with logins
├─ Mobile apps with users
├─ Interactive scenarios
├─ User-specific data access
└─ Use Authorization Code Grant instead

User Context Required:
├─ When you need to know which user
├─ User-specific APIs (/v2/users/me)
├─ Personalized interactions
├─ Individual user permissions
└─ Cannot use Client Credentials

Single-Step Authentication Flow

Unlike Authorization Code's two-step process, Client Credentials is direct and immediate:

Client Credentials Flow:

Step 1: Direct Exchange
Your App → Sends credentials → Genesys Authorization Server
Step 2: Immediate Response
Genesys Authorization Server → Returns access token → Your App
Step 3: Use Token
Your App → Uses token → Calls Genesys APIs

No User Involved:
├─ No browser redirect
├─ No user login screen
├─ No permission consent
├─ No user interaction
└─ Application acts as itself

Timeline:
├─ Entire flow: milliseconds
├─ No waiting for user
├─ Can happen anytime
├─ No session required
└─ Fully automated

Contrast with Authorization Code:
├─ Authorization Code: 3 round trips (browser, user, server)
├─ Client Credentials: 1 round trip (direct)
├─ Authorization Code: Minutes (user delays)
├─ Client Credentials: Milliseconds
└─ Different use cases, different timing

Complete Client Credentials Flow

Step 1: Application Authenticates

Your application makes direct request to Genesys:

POST https://login.mypurecloud.com/oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET
&scope=conversations:readonly+users:manage+scheduling:manage

Request Parameters:

grant_type (required):
├─ Must be "client_credentials"
├─ Identifies this grant type
└─ Tells server how to authenticate

client_id (required):
├─ Your application's public identifier
├─ Identifies which app is requesting
└─ Visible in logs and audit trails

client_secret (required):
├─ Your application's secret credential
├─ NEVER expose in frontend code
├─ Server-side only
├─ Used to prove application identity
└─ Should be rotated monthly

scope (required):
├─ Space-separated list of permissions
├─ Defines what API access is granted
├─ Examples: "users:manage" "scheduling:manage"
├─ Request MINIMUM scopes needed
└─ Note: Cannot exceed application's role permissions

Security Note:
├─ This is direct server-to-server
├─ Both client_id and client_secret sent
├─ client_secret proves application identity
├─ HTTPS encryption required
├─ No user credentials involved
└─ No credentials transmitted to third parties

Step 2: Receive Access Token

Genesys Cloud responds immediately:

HTTP 200 OK

Content-Type: application/json

```
{
  "access_token": "abc123xyz789...",
  "token_type": "bearer",
  "expires_in": 86400,
  "scope": "conversations:readonly users:manage scheduling:manage"
}
```

Response Fields:

access_token:
├─ The access token
├─ Use with all API requests
├─ Include in Authorization header
├─ Proves application authenticated
├─ Example: "Bearer abc123xyz789..."
└─ Expires automatically (default 1 hour)

token_type:
├─ Always "bearer" for Genesys Cloud
├─ Indicates HTTP Bearer token format
└─ Use in header: "Authorization: Bearer {token}"

expires_in:
├─ Token lifetime in seconds
├─ Default: 3600 (1 hour)
├─ Configurable when creating OAuth client
├─ Range: 300 to 172,800 seconds
├─ SCIM special: Up to 450 days
└─ Application must refresh after expiration

scope (informational):
├─ Actual scopes granted
├─ Should match what you requested
├─ Confirms which APIs you can access
└─ Useful for debugging

No Refresh Token:

├─ Client Credentials does NOT include refresh token
├─ Simply authenticate again when token expires
├─ Each authentication is independent
├─ Same as getting fresh token
└─ Simpler than managing refresh tokens

Error Response:

If authentication fails:

HTTP 400 Bad Request
{
  "error": "invalid_client",
  "error_description": "client_id or client_secret is invalid"
}

OR:

HTTP 403 Forbidden
{
  "error": "invalid_scope",
  "error_description": "Application does not have access to this scope"
}

Common Error Codes:
├─ invalid_client: Credentials invalid
├─ invalid_scope: Requested scope not allowed
├─ unsupported_grant_type: Wrong grant type
└─ server_error: Genesys temporary issue (retry)

Step 3: Use Access Token

Your application now has access token

Use to call Genesys Cloud APIs:

GET /api/v2/users
Host: api.mypurecloud.com

Authorization: Bearer abc123xyz789...
Content-Type: application/json

Parameters:
 ?pageSize=100&pageNumber=1

Example Response:
HTTP 200 OK

{
 "entities": [
  {
   "id": "user-123",
   "email": "john@example.com",
   "name": "John Doe"
  },
  ...
 ],
 "pageNumber": 1,
 "pageSize": 100,
 "total": 5000
}

Token Usage Rules:
├─ Include in Authorization header
├─ Format: "Bearer {access_token}"
├─ Send with every API request
├─ HTTPS connection required
├─ No other authentication needed
├─ Genesys validates on each request
└─ Application identity verified by token

Important Limitation:
├─ NO user context
├─ Cannot access /v2/users/me (no "me")
├─ Cannot know which user made request
├─ Application acts as itself
├─ No user-specific data access
├─ All requests attributed to app, not user
└─ Perfect for background jobs

Step 4: Token Expires & Refresh

After 1 hour (or configured duration):

Access token expires automatically

Your app's next API request with expired token:

GET /api/v2/conversations
Authorization: Bearer abc123xyz789... (expired)

Genesys Cloud responds:

HTTP 401 Unauthorized
{
  "error": "invalid_token",
  "error_description": "Access token expired"
}

How to Handle Token Expiration:

Option 1: Proactive Refresh (Recommended)
├─ Monitor token expiration time
├─ Refresh 5 minutes BEFORE expiration
├─ Always have fresh token available
├─ No 401 errors encountered
└─ Smoother operation

Option 2: Reactive Refresh
├─ Continue until 401 error
├─ Detect 401 response
├─ Authenticate again (Step 1)
├─ Get new access token
└─ Retry original request

Getting New Token:

Simply authenticate again:

POST https://login.mypurecloud.com/oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET
&scope=conversations:readonly+users:manage

Response:
HTTP 200 OK
{
  "access_token": "new_token_def456...",
  "token_type": "bearer",
  "expires_in": 86400,
  "scope": "..."
}

Use new token immediately:
├─ Discard old token
├─ Use new token for requests
├─ Repeat cycle on next expiration
└─ No special refresh_token needed

Token Duration Configuration

When Creating OAuth Client:

Standard Duration:
├─ Default: 86,400 seconds (1 day)
├─ Configurable range: 300-172,800 seconds
├─ Maximum standard: 48 hours
└─ Sufficient for most applications

SCIM Integration Duration (Special):
├─ Up to 38,880,000 seconds (450 days!)
├─ Requires SCIM Integration role
├─ Extended duration for identity provisioning

├─ Reduces authentication frequency
└─ Requires different configuration

HIPAA Compliance:
├─ Overrides token duration
├─ Enforces 15-minute idle timeout
├─ Applies to all OAuth tokens
├─ Security requirement
└─ Automatic when HIPAA enabled

Choosing Duration:

Short Duration (5-60 minutes):
├─ More secure (smaller compromise window)
├─ More frequent authentication needed
├─ Suitable for: High-security, continuous applications
└─ Example: Real-time event processing

Medium Duration (1-8 hours):
├─ Balanced approach
├─ Typical for most applications
├─ Suitable for: Background jobs, daily processes
└─ Example: Hourly data sync

Long Duration (1-2 days):
├─ Fewer authentications
├─ Less secure (larger compromise window)
├─ Suitable for: Long-running background tasks
└─ Example: Large data migrations

Very Long Duration (SCIM only):
├─ 450 days maximum
├─ Minimal authentication needed
├─ Identity provisioning specific
├─ Reduced overhead for identity sync
└─ Requires SCIM Integration role

Recommendation:
├─ Start with 1 hour (3600 seconds)
├─ Adjust based on actual usage

```
├─ Monitor authentication frequency
├─ Balance security vs convenience
└─ Document your choice
```

Complete Python Example

```
import requests
import json
from datetime import datetime, timedelta

class GenesysCloudAPI:
    def __init__(self, client_id, client_secret, region='mypurecloud.com'):
        self.client_id = client_id
        self.client_secret = client_secret
        self.region = region
        self.auth_url = f'https://login.{region}/oauth/token'
        self.api_url = f'https://api.{region}/api/v2'

        self.access_token = None
        self.token_expires_at = None

    def authenticate(self, scopes='conversations:readonly users:readonly'):
        """Step 1: Authenticate using Client Credentials"""

        data = {
            'grant_type': 'client_credentials',
            'client_id': self.client_id,
            'client_secret': self.client_secret,
            'scope': scopes
        }

        try:
            response = requests.post(self.auth_url, data=data)
            response.raise_for_status()

            # Step 2: Receive token
            token_data = response.json()
```

```
            self.access_token = token_data['access_token']

            # Calculate expiration (expire 5 minutes early)
            expires_in = token_data['expires_in']
            self.token_expires_at = datetime.now() + timedelta(seconds=expires_in - 300)

            print(f'[{datetime.now().strftime("%H:%M:%S")}] Authenticated successfully')
            print(f'Token expires at: {self.token_expires_at.strftime("%H:%M:%S")}')
            return True

        except requests.exceptions.RequestException as e:
            print(f'Authentication failed: {e}')
            return False

    def ensure_token(self, scopes='conversations:readonly users:readonly'):
        """Proactive token refresh if expiring soon"""

        if self.access_token is None:
            return self.authenticate(scopes)

        # Check if token expiring soon (proactive refresh)
        if datetime.now() >= self.token_expires_at:
            print('Token expired, refreshing...')
            return self.authenticate(scopes)

        return True

    def get_conversations(self):
        """Step 3: Use token to call API"""

        if not self.ensure_token():
            print('Failed to obtain access token')
            return None

        headers = {
            'Authorization': f'Bearer {self.access_token}',
            'Content-Type': 'application/json'
        }

        try:
```

```
        response = requests.get(
            f'{self.api_url}/conversations',
            headers=headers,
            params={'pageSize': 100, 'pageNumber': 1}
        )
        response.raise_for_status()

        data = response.json()
        print(f'Retrieved {len(data.get("entities", []))} conversations')
        return data

    except requests.exceptions.HTTPError as e:
        if e.response.status_code == 401:
            print('Token expired, retrying with fresh token...')
            if self.authenticate():
                return self.get_conversations()  # Retry
        print(f'API call failed: {e}')
        return None

def create_user(self, email, name):
    """Example: Create new user"""

    if not self.ensure_token('users:manage'):
        return None

    headers = {
        'Authorization': f'Bearer {self.access_token}',
        'Content-Type': 'application/json'
    }

    data = {
        'email': email,
        'name': name,
        'state': 'active'
    }

    try:
        response = requests.post(
            f'{self.api_url}/users',
            headers=headers,
```

```
            json=data
        )
        response.raise_for_status()

        user = response.json()
        print(f'Created user: {user["name"]} ({user["id"]})')
        return user

    except requests.exceptions.RequestException as e:
        print(f'Failed to create user: {e}')
        return None

# Usage Example:
if __name__ == '__main__':
    # Initialize with credentials
    client = GenesysCloudAPI(
        client_id='YOUR_CLIENT_ID',
        client_secret='YOUR_CLIENT_SECRET',
        region='mypurecloud.com'
    )

    # Example 1: Get conversations
    print('--- Getting Conversations ---')
    conversations = client.get_conversations()

    # Example 2: Create user
    print('\n--- Creating User ---')
    user = client.create_user('newuser@company.com', 'New User')

    # Example 3: Long-running job with periodic token refresh
    print('\n--- Simulating Long-Running Job ---')
    for i in range(5):
        print(f'Iteration {i+1}')
        conversations = client.get_conversations()
        # Token automatically refreshed if needed
```

Complete Node.js Example

```
const axios = require('axios');

class GenesysCloudAPI {
  constructor(clientId, clientSecret, region = 'mypurecloud.com') {
    this.clientId = clientId;
    this.clientSecret = clientSecret;
    this.region = region;
    this.authUrl = `https://login.${region}/oauth/token`;
    this.apiUrl = `https://api.${region}/api/v2`;

    this.accessToken = null;
    this.tokenExpiresAt = null;
  }

  // Step 1: Authenticate
  async authenticate(scopes = 'conversations:readonly users:readonly') {
    try {
      const response = await axios.post(this.authUrl, {
        grant_type: 'client_credentials',
        client_id: this.clientId,
        client_secret: this.clientSecret,
        scope: scopes
      });

      // Step 2: Receive token
      this.accessToken = response.data.access_token;

      // Expire 5 minutes early (proactive refresh)
      const expiresIn = response.data.expires_in;
      this.tokenExpiresAt = Date.now() + (expiresIn - 300) * 1000;

      console.log(`[${new Date().toLocaleTimeString()}] Authenticated successfully`);
      console.log(`Token expires at: ${new Date(this.tokenExpiresAt).toLocaleTimeString()}`);
      return true;

    } catch (error) {
      console.error('Authentication failed:', error.message);
      return false;
    }
  }
```

```
// Ensure token is valid (refresh if needed)
async ensureToken(scopes = 'conversations:readonly users:readonly') {
  if (!this.accessToken) {
    return await this.authenticate(scopes);
  }

  // Proactive refresh if expiring
  if (Date.now() >= this.tokenExpiresAt) {
    console.log('Token expired, refreshing...');
    return await this.authenticate(scopes);
  }

  return true;
}

// Step 3: Use token to call API
async getConversations() {
  if (!await this.ensureToken()) {
    console.error('Failed to obtain access token');
    return null;
  }

  try {
    const response = await axios.get(`${this.apiUrl}/conversations`, {
      headers: {
        'Authorization': `Bearer ${this.accessToken}`,
        'Content-Type': 'application/json'
      },
      params: {
        pageSize: 100,
        pageNumber: 1
      }
    });

    console.log(`Retrieved ${response.data.entities.length} conversations`);
    return response.data;

  } catch (error) {
    if (error.response?.status === 401) {
```

```
        console.log('Token expired, retrying...');
        if (await this.authenticate()) {
          return await this.getConversations(); // Retry
        }
      }
      console.error('API call failed:', error.message);
      return null;
    }
  }

  // Example: Create user
  async createUser(email, name) {
    if (!await this.ensureToken('users:manage')) {
      return null;
    }

    try {
      const response = await axios.post(`${this.apiUrl}/users`, {
        email: email,
        name: name,
        state: 'active'
      }, {
        headers: {
          'Authorization': `Bearer ${this.accessToken}`,
          'Content-Type': 'application/json'
        }
      });

      console.log(`Created user: ${response.data.name} (${response.data.id})`);
      return response.data;

    } catch (error) {
      console.error('Failed to create user:', error.message);
      return null;
    }
  }
}

// Usage Example:
(async () => {
```

```
  const client = new GenesysCloudAPI(
    process.env.GENESYS_CLIENT_ID,
    process.env.GENESYS_CLIENT_SECRET,
    'mypurecloud.com'
  );

  // Example 1: Get conversations
  console.log('--- Getting Conversations ---');
  await client.getConversations();

  // Example 2: Create user
  console.log('\n--- Creating User ---');
  await client.createUser('newuser@company.com', 'New User');

  // Example 3: Long-running job
  console.log('\n--- Simulating Long-Running Job ---');
  for (let i = 0; i < 5; i++) {
    console.log(`Iteration ${i + 1}`);
    await client.getConversations();
  }
})();
```

Security Best Practices

Client Credentials Security Checklist:

□ Store client_secret in secure vault
  ├─ HashiCorp Vault
  ├─ AWS Secrets Manager
  ├─ Azure Key Vault
  └─ Never in code/git

□ Rotate credentials monthly
  ├─ Plan rotation schedule
  ├─ Generate new secret in Admin UI
  ├─ Update application config
  └─ Verify working before removing old

□ Monitor usage patterns
  ├─ Track authentication frequency
  ├─ Alert on unusual activity
  ├─ Audit all API calls
  └─ Review access logs

□ Implement audit logging
  ├─ Log authentication attempts
  ├─ Log API calls (not tokens)
  ├─ Track failures/errors
  └─ Retain logs for compliance

□ Use HTTPS only
  ├─ All requests use HTTPS
  ├─ Validate SSL certificates
  ├─ Prevent man-in-the-middle
  └─ Never fall back to HTTP

□ Limit scope to minimum
  ├─ Request only needed scopes
  ├─ Principle of least privilege
  ├─ Review scopes regularly
  └─ Remove unused scopes

□ Implement error handling
  ├─ Handle authentication errors
  ├─ Retry failed requests
  ├─ Don't expose credentials in errors
  └─ Log securely without tokens

□ Restrict permissions
  ├─ Assign minimal roles
  ├─ Use divisions for scope
  ├─ Review permissions quarterly
  └─ Remove unneeded permissions

□ Monitor expiration
  ├─ Track token expiration
  ├─ Refresh proactively

├─ Alert if refresh fails
└─ Implement fallback behavior

□ Version control security
├─ Never commit secrets
├─ Use .gitignore
├─ Use environment variables
└─ Review git history

Common Use Cases

1. Salesforce ↔ Genesys Sync

Schedule: Every 30 minutes

Process:
├─ Authenticate with Client Credentials
├─ Query Salesforce API (with SF credentials)
├─ Fetch modified contacts
├─ Transform to Genesys format
├─ POST to Genesys externalcontacts API
├─ Log results
└─ Token auto-refreshes if needed

Benefits:
├─ No user interaction
├─ Fully automated
├─ Scheduled sync
└─ Deterministic refresh

2. Nightly Report Generation

Schedule: 2:00 AM daily

Process:
├─ Authenticate with Client Credentials

├─ Query analytics APIs
├─ Aggregate daily metrics
├─ Generate PDF report
├─ Email report to stakeholders
└─ Clean up temp files

Benefits:
├─ Runs while users sleep
├─ No performance impact
├─ Automated reporting
└─ Email delivery

3. Real-Time Data Processing

Schedule: Continuous

Process:
├─ Authenticate once (at startup)
├─ Proactively refresh before expiry
├─ Connect to event streaming
├─ Process events real-time
├─ Call APIs based on events
└─ Token automatically refreshed

Benefits:
├─ Always-on application
├─ Smooth operation
├─ No user waiting
└─ Fully automated

4. Bulk Contact Import

Schedule: Ad-hoc

Process:
├─ Authenticate with Client Credentials
├─ Read contact CSV file
├─ Parse and validate data

├─ Batch create in Genesys
├─ Log import results
├─ Handle errors gracefully
└─ Email summary report

Benefits:
├─ One-time operation
├─ No UI needed
├─ Simple authentication
└─ Bulk operations efficient

Comparison with Authorization Code

Client Credentials vs Authorization Code:

Timeline:
├─ Client Credentials: Milliseconds
├─ Authorization Code: Minutes (user delay)
└─ Different use cases

User Interaction:
├─ Client Credentials: None
├─ Authorization Code: Required (login, consent)
└─ Different workflows

Tokens:
├─ Client Credentials: No refresh token
├─ Authorization Code: Includes refresh token
└─ Different lifecycle management

Secret Exposure:
├─ Client Credentials: Secret sent to server
├─ Authorization Code: Secret never exposed to browser
└─ Different security models

Use Case:
├─ Client Credentials: Background jobs, services
├─ Authorization Code: Web/mobile apps, users
└─ Choose based on scenario

When Choose Client Credentials:
├─ No user interaction needed
├─ Automated/scheduled processes
├─ Service-to-service integration
├─ No user-specific access needed
└─ Application acts as itself

When Choose Authorization Code:
├─ User logs in to app
├─ User grants permission
├─ Long-lived access needed
├─ User-specific data required
└─ User can revoke access anytime

Key Takeaways: Chapter 3

- **Single-Step Process** - Direct authentication, no user interaction
- **No User Context** - Application acts as itself (no /users/me access)
- **Service-to-Service** - Primary use for automated integrations
- **No Refresh Token** - Simply authenticate again when token expires
- **Immediate** - Token available instantly for use
- **Fully Automated** - Perfect for background jobs and scheduled tasks
- **Simple Flow** - One request for token, then use it
- **Role-Based Access** - Permissions determined by assigned roles/divisions

---

Interview Prep: Client Credentials Grant

Question	Answer

When use Client Credentials?	Service-to-service, background jobs, no user interaction
Single-step or two-step?	Single-step (direct authentication)
Refresh token included?	No - authenticate again when token expires
User context?	No (/v2/users/me not available)
Client_secret exposure?	Only server-to-server (HTTPS required)
How determine permissions?	OAuth client roles and scope settings
Typical use case?	Salesforce sync, nightly reports, data imports
Token lifetime?	Default 1 hour (configurable 300-172,800 seconds)
HTTPS required?	Yes (always)
Error 401 handling?	Authenticate again with fresh credentials

---

Document Version

Chapter: 3 of 8
Last Updated: March 2026
Status: Current with RFC 6749
Scope: Client Credentials Grant, Service Integration, Implementation

Authorization Code with PKCE

Overview

PKCE (Proof Key for Code Exchange) is an extension to the OAuth 2.0 Authorization Code Grant (RFC 7636) that provides enhanced security, especially for public clients like single-page applications (SPAs) and mobile apps that cannot securely store a client_secret. PKCE is now recommended by OAuth 2.0 best practices and is already supported in Genesys Cloud.

Status: Implicit Grant deprecated (May 2027 deadline), PKCE is the secure replacement.

Why PKCE?

The Problem PKCE Solves:

Authorization Code Can Be Intercepted:
├─ Attacker monitors network traffic
├─ Captures authorization code
├─ Attempts to exchange code for token
└─ Without PKCE: Attacker succeeds

Public Clients Cannot Store Secrets:
├─ Browser-based apps: No server backend
├─ Mobile apps: Can be reverse engineered
├─ Desktop apps: Can be analyzed
├─ Cannot safely store client_secret
└─ Traditional approach inadequate

PKCE Solution:

Add Proof to Authorization Code:

├─ Generate random code_verifier
├─ Compute code_challenge (hash)
├─ Send code_challenge to auth server
├─ Auth server stores it
├─ Only original verifier can exchange code
└─ Attacker lacks verifier → cannot exchange

Proof Cannot Be Reversed:
├─ code_challenge = SHA256(code_verifier)
├─ Hash is one-way function
├─ Cannot reverse-engineer verifier from hash
├─ Even if code intercepted: useless without verifier
└─ Verifier never sent over network

Complete PKCE Flow

Step 1: Generate Proof Strings

Your Application Generates:

code_verifier:
├─ Random string, 43-128 characters
├─ Cryptographically secure (use crypto random)
├─ Unrepeatable (different each request)
├─ Only stored in memory
├─ Example: "E9Mrozoa2owusvxrFHo89ejyK3OMVZZWhtbQrHfl"

code_challenge:
├─ SHA256 hash of code_verifier
├─ BASE64-URL encoded
├─ Sent to authorization server
├─ Example: "47DEQpj8HBSa-_TImW-5JCeuQeRkm5NMpJWZG3hSuFU"

code_challenge_method:
├─ "S256" (SHA256 hash)
├─ Only recommended method

├─ "plain" exists but deprecated
└─ Always use "S256"

Pseudo Code:

```
code_verifier = generateRandomString(128)
code_challenge = BASE64URL(SHA256(code_verifier))
code_challenge_method = "S256"
```

Step 2: Redirect to Authorization Endpoint

Redirect user browser to:

```
https://login.mypurecloud.com/oauth/authorize
  ?client_id=YOUR_CLIENT_ID
  &response_type=code
  &redirect_uri=https://yourapp.com/callback
  &scope=conversations:readonly
  &code_challenge=47DEQpj8HBSa-_TImW-5JCeuQeRkm5NMpJWZG3hSuFU
  &code_challenge_method=S256
  &state=random_state_string
```

Parameters:

client_id:
├─ Your app's public identifier
└─ Can be embedded in SPA code

response_type:
├─ Must be "code"
└─ Returns authorization code

redirect_uri:
├─ Where user is redirected
├─ Can be embedded in SPA code
├─ Example: https://yourapp.com/callback
└─ Must be registered in OAuth client

scope:
├─ Requested permissions
├─ Space-separated list
└─ Example: "conversations:readonly"

code_challenge (PKCE):
├─ SHA256 hash of random string
├─ Sent to auth server
├─ Auth server stores it
└─ Cannot derive original verifier

code_challenge_method (PKCE):
├─ "S256" (recommended and required)
├─ Indicates SHA256 method used
└─ Only secure method

state (CSRF Protection):
├─ Random string
├─ Prevents CSRF attacks
└─ Verified in callback

Step 3: User Authenticates & Consents

Same as Authorization Code Grant:

1. User sees Genesys Cloud login
2. User enters credentials
3. User sees permission consent screen
4. User grants permission
5. Auth server generates authorization code
6. Auth server stores code_challenge with authorization code

Step 4: Callback with Authorization Code

Auth server redirects to callback:

https://yourapp.com/callback

?code=AUTH_CODE_12345abcde67890
&state=random_state_string

In Callback Handler:

1. Verify state parameter (CSRF check)
2. Retrieve authorization code
3. Verify you have code_verifier in memory
4. Proceed to Step 5 (exchange code)

Never:
├─ Do NOT send code_verifier in callback
├─ Do NOT include code_verifier in URL
├─ Do NOT exchange code in browser
└─ Do NOT expose code_verifier to user

Step 5: Exchange Code with PKCE Proof

Now you have:
├─ Authorization code (from Step 4)
├─ code_verifier (generated in Step 1, stored in memory)
└─ client_id (public, stored in app)

Exchange Code:

POST https://login.mypurecloud.com/oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=authorization_code
&code=AUTH_CODE_12345abcde67890
&client_id=YOUR_CLIENT_ID
&redirect_uri=https://yourapp.com/callback
&code_verifier=E9Mrozoa2owusvxrFHo89ejyK3OMVZZWhtbQrHfl

Parameters:

grant_type:
├─ "authorization_code"

└ Standard OAuth parameter

code:
├ Authorization code from Step 4
└ Single-use only

client_id:
├ Your public client ID
└ Can be public

redirect_uri:
├ Must match redirect_uri from Step 2
└ Confirms code ownership

code_verifier (PKCE):
├ Original random string from Step 1
├ Proves you own the authorization code
├ MUST match the code_challenge sent in Step 2
└ Server recomputes SHA256(code_verifier) and compares

NOTE: NO client_secret needed!
├ PKCE replaces need for client_secret
├ Perfect for public clients
├ Proof of ownership is cryptographic
└ Signature is binding

Step 6: Server Validates & Issues Token

Genesys Cloud Authorization Server:

1. Receives code_verifier
2. Retrieves authorization code from storage
3. Retrieves stored code_challenge
4. Computes: SHA256(code_verifier) → new_hash
5. Compares: new_hash == stored_code_challenge?

If MATCH (✓):
├ code_verifier is correct

├─ Same application that requested code
├─ Issue access token
└─ Return token in response

If NO MATCH (✗):
├─ code_verifier is wrong
├─ Likely code theft attempt
├─ Reject request with error
└─ Attack prevented!

Response on Success:

HTTP 200 OK

```
{
  "access_token": "abc123xyz789...",
  "token_type": "bearer",
  "expires_in": 3600,
  "scope": "conversations:readonly"
}
```

Response on Failure:

HTTP 400 Bad Request

```
{
  "error": "invalid_grant",
  "error_description": "code_verifier invalid"
}
```

Step 7: Use Access Token

Same as Authorization Code Grant:

```
GET /api/v2/conversations
Authorization: Bearer abc123xyz789...
```

Response:
HTTP 200 OK
{...conversation data...}

JavaScript Implementation Example

```
// Configuration
const CLIENT_ID = 'your_client_id';
const REDIRECT_URI = 'https://yourapp.com/callback';
const SCOPES = 'conversations:readonly users:readonly';
const GENESYS_REGION = 'mypurecloud.com';

// Step 1: Generate PKCE code challenge
async function generatePKCE() {
  // Generate random code_verifier
  const array = new Uint8Array(64);
  crypto.getRandomValues(array);
  const codeVerifier = btoa(String.fromCharCode.apply(null, array))
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=/g, '');

  // Compute code_challenge = BASE64URL(SHA256(code_verifier))
  const encoder = new TextEncoder();
  const data = encoder.encode(codeVerifier);
  const hashBuffer = await crypto.subtle.digest('SHA-256', data);

  const hashArray = Array.from(new Uint8Array(hashBuffer));
  const hashString = String.fromCharCode.apply(null, hashArray);
  const codeChallenge = btoa(hashString)
    .replace(/\+/g, '-')
    .replace(/\//g, '_')
    .replace(/=/g, '');

  return { codeVerifier, codeChallenge };
}

// Step 2: Redirect to authorization
async function login() {
```

```
  const { codeVerifier, codeChallenge } = await generatePKCE();

  // Store verifier in memory (or sessionStorage for SPA)
  sessionStorage.setItem('pkce_verifier', codeVerifier);

  // Generate state for CSRF protection
  const state = btoa(Math.random().toString()).substring(0, 40);
  sessionStorage.setItem('pkce_state', state);

  // Redirect to authorization endpoint
  const authUrl = new URL(`https://login.${GENESYS_REGION}/oauth/authorize`);
  authUrl.searchParams.append('client_id', CLIENT_ID);
  authUrl.searchParams.append('response_type', 'code');
  authUrl.searchParams.append('redirect_uri', REDIRECT_URI);
  authUrl.searchParams.append('scope', SCOPES);
  authUrl.searchParams.append('code_challenge', codeChallenge);
  authUrl.searchParams.append('code_challenge_method', 'S256');
  authUrl.searchParams.append('state', state);

  window.location.href = authUrl.toString();
}

// Step 4: Handle callback
async function handleCallback() {
  const urlParams = new URLSearchParams(window.location.search);
  const code = urlParams.get('code');
  const state = urlParams.get('state');
  const error = urlParams.get('error');

  // Check for errors
  if (error) {
    console.error('Authorization error:', error);
    return false;
  }

  // Verify state (CSRF protection)
  const storedState = sessionStorage.getItem('pkce_state');
  if (state !== storedState) {
    console.error('State mismatch - CSRF attack detected');
    return false;
```

```
}

// Retrieve code_verifier from memory
const codeVerifier = sessionStorage.getItem('pkce_verifier');
if (!codeVerifier) {
  console.error('Code verifier not found');
  return false;
}

// Step 5: Exchange code for token
try {
  const response = await fetch(`https://login.${GENESYS_REGION}/oauth/token`, {
    method: 'POST',
    headers: {
      'Content-Type': 'application/x-www-form-urlencoded'
    },
    body: new URLSearchParams({
      grant_type: 'authorization_code',
      code: code,
      client_id: CLIENT_ID,
      redirect_uri: REDIRECT_URI,
      code_verifier: codeVerifier  // PKCE proof
    })
  });

  if (!response.ok) {
    throw new Error('Token exchange failed');
  }

  const { access_token } = await response.json();

  // Step 7: Store token and use it
  sessionStorage.setItem('access_token', access_token);

  // Clean up PKCE values
  sessionStorage.removeItem('pkce_verifier');
  sessionStorage.removeItem('pkce_state');

  console.log('Login successful');
  return true;
```

```
  } catch (error) {
    console.error('Token exchange error:', error);
    return false;
  }
}

// Use access token
async function callAPI(endpoint) {
  const token = sessionStorage.getItem('access_token');

  if (!token) {
    console.error('No access token found');
    return null;
  }

  try {
    const response = await fetch(`https://api.${GENESYS_REGION}/api/v2${endpoint}`, {
      headers: {
        'Authorization': `Bearer ${token}`,
        'Content-Type': 'application/json'
      }
    });

    if (response.status === 401) {
      console.error('Token expired, user must log in again');
      login();
      return null;
    }

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

    return await response.json();

  } catch (error) {
    console.error('API call failed:', error);
    return null;
  }
```

```
}

// Example: Login button click handler
document.getElementById('loginBtn').addEventListener('click', login);

// Example: Handle callback on return from auth server
if (window.location.pathname === '/callback') {
  handleCallback().then(success => {
    if (success) {
      window.location.href = '/dashboard';
    } else {
      window.location.href = '/error';
    }
  });
}

// Example: Call API
async function getConversations() {
  const data = await callAPI('/conversations?pageSize=100&pageNumber=1');
  console.log('Conversations:', data);
}
```

Why Migrate from Implicit Grant to PKCE?

```
Implicit Grant (DEPRECATED):
├─ Status: Deprecated November 2025
├─ New clients: Blocked March 2026
├─ Existing clients: Must migrate by May 2027
├─ Issues: Token in URL, browser history, no protection
└─ Replacement: PKCE

PKCE (RECOMMENDED):
├─ Status: Supported and recommended
├─ Security: Enhanced via proof mechanism
├─ Suitable for: All client types
```

├─ Implementation: Slightly more complex
└─ Future-proof: Long-term standard

Migration Path:

Step 1: Update OAuth Client
├─ Delete old Implicit Grant client (or keep if needed)
├─ Create new Authorization Code + PKCE client
└─ Note new client_id

Step 2: Update Application Code
├─ Implement PKCE proof generation
├─ Add code_challenge to authorization
├─ Include code_verifier in token exchange
└─ Remove implicit grant references

Step 3: Test Thoroughly
├─ Test login flow end-to-end
├─ Test token exchange
├─ Test API calls
└─ Verify error handling

Step 4: Deploy
├─ Update production code
├─ Verify working in production
├─ Monitor for errors
└─ Keep old client available briefly

Step 5: Cleanup
├─ Remove old Implicit Grant client
├─ Update documentation
├─ Notify users if applicable
└─ Archive old implementation

Timeline:
├─ Now (March 2026): Implement PKCE
├─ Before May 2027: Migration required
├─ May 2027: Implicit Grant stopped working
└─ Plan ahead to avoid outages!

PKCE vs Implicit Grant

Security Comparison:

Token Exposure:
├─ Implicit: Token in URL fragment (visible)
├─ PKCE: Token in body, 200 response (hidden)
└─ PKCE wins: Less exposed

Token Lifetime:
├─ Implicit: No refresh, static lifetime
├─ PKCE: Can have refresh tokens
└─ PKCE wins: Better UX

Browser History:
├─ Implicit: Token in URL history (risk)
├─ PKCE: No URL tokens (safe)
└─ PKCE wins: No history exposure

Code Interception:
├─ Implicit: Not applicable (no code)
├─ PKCE: Protected by proof (secure)
└─ PKCE wins: Protected exchange

CSRF Protection:
├─ Implicit: State parameter only
├─ PKCE: State + cryptographic proof
└─ PKCE wins: Multiple protections

Standards Alignment:
├─ Implicit: Deprecated OAuth 2.0
├─ PKCE: Modern OAuth 2.0 best practice
└─ PKCE wins: Future-proof

Complexity:
├─ Implicit: Simple (but insecure)
├─ PKCE: Slightly more complex (much more secure)

└─ Trade-off: Worth the extra complexity

Security Checklist for PKCE

PKCE Implementation Security:

□ Generate cryptographically secure code_verifier
  ├─ Use crypto.getRandomValues() (not Math.random)
  ├─ 64 bytes minimum (not shorter)
  └─ Different every request

□ Compute SHA256 hash of verifier
  ├─ Use crypto.subtle.digest()
  ├─ Encode to BASE64URL format
  └─ Do not use plain method

□ Store code_verifier securely
  ├─ Memory only (not localStorage)
  ├─ Clear after token exchange
  ├─ Not persisted
  └─ Lost on page reload

□ Use S256 method always
  ├─ Never use "plain" method
  ├─ Only S256 recommended
  └─ Specify in authorization

□ Include state parameter
  ├─ Separate CSRF protection
  ├─ Random and unpredictable
  ├─ Verify in callback
  └─ Different from verifier

□ Validate SSL certificates
  ├─ HTTPS always
  ├─ Check cert validity
  └─ Reject self-signed

□ Do not expose secrets

├─ code_verifier: Memory only

├─ access_token: SessionStorage or memory

├─ Never in localStorage

└─ Clean up after logout

□ Handle errors gracefully

├─ Catch network errors

├─ Retry on failure

├─ Don't expose verifier in errors

└─ Log safely

Key Takeaways: Chapter 4

- **Enhanced Security** - Cryptographic proof prevents authorization code interception
- **No Client Secret** - Suitable for public clients (SPAs, mobile, desktop)
- **Proof Mechanism** - Verifier prevents code theft (cannot reverse-engineer from hash)
- **RFC 7636 Standard** - Modern OAuth 2.0 best practice
- **Implicit Replacement** - Use PKCE instead of deprecated Implicit Grant
- **Migration Deadline** - May 2027 cutoff for Implicit Grant
- **Slightly Complex** - More code than Implicit, but much more secure
- **Future-Proof** - Long-term standard, recommended by OAuth 2.0 experts

---

Interview Prep: PKCE

Question	Answer
What is PKCE?	Proof Key for Code Exchange - enhanced OAuth Code grant security
Why PKCE needed?	Prevents authorization code interception attacks
code_verifier?	Random string (43-128 chars) generated by client, never sent over network
code_challenge?	SHA256(code_verifier), BASE64URL encoded, sent to auth server
How prevent intercept?	Only original code_verifier can exchange the code, attacker lacks verifier

Question	Answer
Why not reverse?	SHA256 is one-way hash function, cannot derive verifier from challenge
S256 vs plain?	S256 (SHA256) is secure, plain is deprecated, always use S256
When use PKCE?	Public clients (SPAs, mobile, desktop) that cannot store client_secret
Migration deadline?	May 2027 for Implicit Grant existing clients
State parameter?	Separate CSRF protection, still needed with PKCE

Document Version

Chapter: 4 of 8
Last Updated: March 2026
Status: Current with RFC 7636
Scope: PKCE Flow, Security, Implementation, Migration from Implicit

OAuth Scopes and Permissions

Overview

OAuth scopes provide granular, fine-grained permission control within Genesys Cloud. They define exactly what resources an application can access and what actions it can perform. Scopes are requested during authorization and enforced on every API call.

Scope Fundamentals

```
What Are Scopes?

Definition:
├─ Granular permissions for API access
├─ Limit what an application can do
├─ User consent on what app can access
├─ Enforced on every API request
└─ Principle of least privilege

Scope Format:
  resource:action
  OR
  resource:action:scope

Examples:
├─ conversations:readonly       # View conversations
├─ conversations:call:add       # Make calls
├─ conversations:call:control   # Transfer, hold
├─ users:manage                 # Create/modify users
├─ routing:queue:view           # View queues
```

└ scheduling:manage # Manage schedules

Space-Separated Combination:
"conversations:readonly users:readonly scheduling:manage"

User Sees During Authorization:
├ App name
├ All requested scopes
├ What the app can do
└ User grants all or none (all-or-nothing)

Enforcement:
├ Every API request validated against token scopes
├ If endpoint requires scope not in token: 403 Forbidden
├ User permissions AND scopes both required (AND logic)
├ Whichever is more restrictive applies
└ Better to have both than just one

Scope Categories

Conversation Management:
├ conversations:readonly # GET conversations
├ conversations:call:add # POST/make calls
├ conversations:call:control # Transfer, hold, disconnect
├ conversations:call:pull # Pull calls
├ conversations:call:coach # Coach calls
├ recordings:view # Access recordings
├ recordings:download # Download recording files
├ transcripts:readonly # View transcripts
└ conversations:external:contact:add # Add external contacts

User Management:
├ users:readonly # Read user info
├ users:manage # Create/update/delete users
├ authorization:grant:readonly # View user permissions
├ authorization:grant:manage # Modify user permissions
├ presence:readonly # View presence status

```
└─ presence:manage              # Change presence

Contact Center Configuration:
├─ routing:queue:view           # View queue config
├─ routing:queue:manage         # Create/edit queues
├─ routing:skill:view           # View skills
├─ routing:skill:manage         # Manage skills
├─ directory:organization:view  # View org structure
├─ telephony:phone:manage       # Manage phones
├─ outbound:campaign:view       # View campaigns
└─ outbound:campaign:execute    # Run campaigns

Workforce Management:
├─ forecasting:readonly         # View forecasts
├─ forecasting:manage           # Create/edit forecasts
├─ scheduling:readonly          # View schedules
├─ scheduling:manage            # Create/edit schedules
├─ timeoff:view                 # View time-off requests
├─ timeoff:manage               # Approve time-off
├─ adherence:view               # View adherence
└─ adherence:manage             # Manage adherence

Analytics & Reporting:
├─ analytics:conversationDetail # Detailed conversation analytics
├─ analytics:aggregate:view     # Aggregated analytics
├─ quality:evaluation:view      # Quality evaluations
├─ quality:evaluation:manage    # Create/edit evaluations
├─ speechanalytics:data:view    # Speech analytics
└─ reporting:analytics:view     # Analytics reports

Integration & External:
├─ externalcontacts:manage      # CRM synchronization
├─ webhooks:manage              # Webhook configuration
├─ webhooks:view                # View webhooks
├─ scim:write                   # SCIM provisioning
├─ knowledge:manage             # Knowledge articles
└─ knowledge:readonly           # View knowledge

Platform Management:
├─ oauth:client:view            # View OAuth clients
```

```
├─ oauth:client:manage          # Create/edit OAuth clients
├─ admin:org:view               # View org settings
├─ admin:org:manage             # Modify org settings
└─ audit:readonly               # View audit logs
```

Scope Selection Best Practices

Principle of Least Privilege:

Definition:
- ├─ Grant minimum scopes needed
- ├─ Only request what app actually uses
- ├─ Reduce impact if app compromised
- ├─ Easier to review and audit
- └─ Better security posture

How to Select:

1. List API Endpoints Used:
   - ├─ /conversations → conversations:readonly
   - ├─ /v2/users/{id} → users:readonly
   - ├─ /v2/conversations/{id}/notes → conversations:readonly
   - └─ Map each endpoint to scope

2. Check Endpoint Documentation:
   - ├─ Visit API Explorer
   - ├─ View each endpoint
   - ├─ Note required scope
   - └─ Collect all scopes

3. Start with Read-Only:
   - ├─ conversations:readonly (not call:add)
   - ├─ users:readonly (not manage)
   - ├─ Only add write scopes if needed
   - └─ Safer starting point

4. Add Write Scopes Only If Needed:

```
   ├─ Review which operations require writes
   ├─ Add specific action scopes
   ├─ Example: call:add (not call:control)
   └─ One scope per action where possible

5. Remove Unused Scopes:
   ├─ Quarterly review
   ├─ Check OAuth client usage
   ├─ Remove unneeded scopes
   ├─ Reduce security surface
   └─ Document rationale

6. Document Your Choices:
   ├─ Why each scope is needed
   ├─ Which endpoints use it
   ├─ When to update scopes
   └─ Version control reasons

Example Selection:

App: Contact center agent desktop
├─ Endpoints needed:
│  ├─ GET /conversations → conversations:readonly
│  ├─ POST /conversations/{id}/participants/calls/transfer
│  │  → conversations:call:control
│  ├─ GET /users/me → users:readonly
│  └─ PATCH /v2/users/{id}/presence → presence:manage
│
└─ Final scopes: "conversations:readonly conversations:call:control users:readonly presence:manage"

NOT these (too permissive):
├─ conversations:manage (not needed)
├─ users:manage (not needed)
└─ conversations:call:add (can't initiate calls)
```

Scope Enforcement Mechanism

How Scopes Are Enforced:

Every API Request Validation:

1. Client Sends Request:
   GET /api/v2/users/123
   Authorization: Bearer {access_token}

2. Genesys Validates Token:
   ├─ Token signature valid?
   ├─ Token not expired?
   ├─ Token not revoked?
   └─ Check scopes and permissions

3. Endpoint Requires:
   ├─ Endpoint /v2/users/{id} requires scope: users:readonly
   ├─ Check if token has "users:readonly" scope
   └─ Check if user has permission to read users

4. Dual Validation (Both Required):
   ├─ Must have OAuth scope: ✓ users:readonly
   ├─ Must have user permission: ✓ (has role allowing read)
   ├─ Both? → Grant access
   ├─ Only one? → 403 Forbidden
   └─ Neither? → 403 Forbidden

5. Enforce Rules:
   ├─ User has scope but not permission → 403
   ├─ User has permission but not scope → 403
   ├─ User has both → 200 OK (success)
   └─ User has neither → 403

Example Scenarios:

Scenario 1: Has Scope, Has Permission
├─ Token scope: users:readonly ✓
├─ User role permission: can read users ✓
├─ Result: 200 OK (access granted)
└─ API call succeeds

Scenario 2: Has Scope, No Permission

├─ Token scope: users:readonly ✓

├─ User role permission: cannot read users ✗

├─ Result: 403 Forbidden

└─ API call denied

Scenario 3: No Scope, Has Permission

├─ Token scope: users:readonly ✗

├─ User role permission: can read users ✓

├─ Result: 403 Forbidden

└─ API call denied

Scenario 4: No Scope, No Permission

├─ Token scope: users:readonly ✗

├─ User role permission: cannot read users ✗

├─ Result: 403 Forbidden

└─ API call denied

Example Error Response:

HTTP 403 Forbidden

```
{
  "error": {
    "message": "This application is not authorized to perform this action",
    "code": "PERMISSIONS_INSUFFICIENT",
    "status": 403
  }
}
```

Checking Available Scopes:

In API Explorer:

1. Visit: https://developer.genesys.cloud/api/rest/v2/
2. Click any endpoint
3. See "Authorization" section
4. Lists required scope
5. Copy exact scope name

Common Scope Combinations

Agent Desktop Application:
  conversations:readonly
  conversations:call:control
  users:readonly
  presence:manage
  recordings:view
  (Read conversations, manage calls, see presence, view recordings)

Admin Dashboard:
  users:readonly
  routing:queue:view
  routing:skill:view
  scheduling:readonly
  analytics:conversationDetail
  (Read-only access to key admin features)

WFM Application:
  scheduling:manage
  forecasting:manage
  timeoff:manage
  adherence:manage
  users:readonly
  (Full WFM management)

Integration Service (Salesforce Sync):
  externalcontacts:manage
  users:readonly
  conversations:readonly
  (Sync contacts, read-only other data)

Chatbot / Virtual Agent:
  conversations:external:contact:add
  knowledge:readonly
  (Add external contacts, access knowledge)

Analytics Reporter:

analytics:conversationDetail
analytics:aggregate:view
quality:evaluation:view
speechanalytics:data:view
(Read-only analytics access)

API Automation (Least Privilege):
users:readonly (query users)
outbound:campaign:view (check campaign status)
(Minimal access for specific automation)

Scope Changes & Updates

Adding Scopes to Existing OAuth Client:

Scenario:
├─ App previously only read conversations
├─ Now needs to transfer calls
├─ Must add conversations:call:control scope

Steps:

1. Stop Using Old Client (or keep both briefly):
   ├─ Update app configuration
   ├─ Point to updated client
   └─ Test thoroughly

2. Navigate to OAuth Client in Admin:
   ├─ Admin → Integrations → OAuth
   ├─ Find your OAuth client
   ├─ Click Edit
   └─ Select Scopes

3. Update Scopes:
   ├─ Old: "conversations:readonly"
   ├─ New: "conversations:readonly conversations:call:control"
   ├─ Add all needed scopes

└ Save changes

4. User Must Re-Authorize:
├ For Authorization Code Grant: Yes
├ Users see new permissions consent screen
├ Users must grant new scope
└ New token includes updated scopes

5. Test Thoroughly:
├ Test new functionality
├ Verify old functions still work
├ Check error responses
└ Monitor for issues

6. Gradual Rollout (Optional):
├ Update small group first
├ Monitor for errors
├ Expand to more users
└ Full rollout when confident

Removing Unused Scopes:

Scenario:
├ App no longer uses contact creation
├ Can remove conversations:external:contact:add
├ Reduce security surface

Steps:

1. Verify Not Used:
├ Check code for usage
├ Search for endpoint calls
├ Verify in logs (not used)
└ Confirm removal safe

2. Update OAuth Client:
├ Admin → Integrations → OAuth
├ Click Edit
├ Deselect unused scope
└ Save

3. No Re-Authorization Needed:
   ├─ Existing tokens still work
   ├─ New tokens won't have old scope
   ├─ Endpoints still work (user has permission)
   └─ Gradual transition

4. Monitor for Errors:
   ├─ Watch logs for 403 errors
   ├─ Verify no broken functionality
   └─ Quick rollback if issues

Scope Testing

Testing Scope-Based Authorization:

Setup Test OAuth Client:

1. Create OAuth client with specific scopes
2. Authenticate with that client
3. Test that allowed endpoints work
4. Test that forbidden endpoints fail

Testing Allowed Scope:

Endpoint: GET /conversations
Scope: conversations:readonly
Token has scope: Yes

Test:
```
  curl -H "Authorization: Bearer {token}" \
       https://api.mypurecloud.com/api/v2/conversations
```

Expected: HTTP 200 (success)
Actual: ?

Testing Denied Scope (Negative Test):

Endpoint: POST /conversations (create)

Scope needed: conversations:manage

Token has scope: No

Test:

```
curl -X POST \
    -H "Authorization: Bearer {token}" \
    https://api.mypurecloud.com/api/v2/conversations
```

Expected: HTTP 403 Forbidden

Actual: ?

Test Multiple Scopes:

Token has scopes:

├─ conversations:readonly

├─ users:readonly

└─ NOT: scheduling:manage

Tests to Run:

├─ GET /conversations → 200 (has scope)

├─ GET /users → 200 (has scope)

├─ GET /schedules → 403 (no scope)

├─ DELETE /conversations/{id} → 403 (no manage scope)

└─ DELETE /users/{id} → 403 (no manage scope)

Automated Testing:

Example Test Suite:

Test Cases:

├─ [✓] conversations:readonly allows GET

├─ [✓] conversations:readonly denies POST

├─ [✓] conversations:call:control allows transfer

├─ [✓] users:readonly allows GET /users

├─ [✓] users:readonly denies PUT /users

├─ [✓] Multiple scopes work correctly

├─ [✓] Missing scopes return 403

└─ [✓] Expired token returns 401

Before Deployment:

├─ Run all scope tests

├─ Verify expected 403s

├─ Verify expected 200s

└─ Document scope requirements

Troubleshooting Scope Issues

Problem: Getting 403 Forbidden on Valid Endpoint

Check List:

┌─ Token Valid?

│ ├─ Is token expired?

│ ├─ Try making another call

│ └─ Get fresh token if needed

│

├─ Scope Correct?

│ ├─ Check API Explorer for required scope

│ ├─ Verify token has that scope

│ ├─ Look at token claims if JWT

│ └─ Add missing scope to OAuth client

│

├─ User Has Permission?

│ ├─ User role has permission?

│ ├─ Check user's roles in Admin UI

│ ├─ Check division assignments

│ └─ Add necessary role if missing

│

└─ Both Needed?

  ├─ User permission: Required

  ├─ OAuth scope: Required

  ├─ Have both? Should work

  └─ Missing one? 403 error

Example Troubleshooting:

Error: POST /v2/users returns 403
├─ Required scope: users:manage
├─ Check 1: Token has users:manage? NO ✗
│ └─ Solution: Add users:manage scope to OAuth client
│
├─ Check 2: User has create user permission? YES ✓
│ └─ OK (permission is good)
│
├─ Root Cause: Missing scope in token
├─ Solution: Update OAuth client scopes
└─ Retry after user re-authenticates

Error: GET /v2/conversations returns 403
├─ Required scope: conversations:readonly
├─ Check 1: Token has conversations:readonly? YES ✓
│ └─ OK (scope is good)
│
├─ Check 2: User has read conversations permission? NO ✗
│ └─ Solution: Add user to role with permission
│
├─ Root Cause: User lacks role permission
├─ Solution: Assign appropriate role to user
└─ Retry after role assignment takes effect

Common Mistakes:

 Wrong Scope Name:
├─ Requested: "conversation:view" (wrong)
├─ Correct: "conversations:readonly"
└─ Solution: Check API Explorer for exact scope

 Typo in Scope:
├─ Requested: "users :manage" (space)
├─ Correct: "users:manage"
└─ Solution: Check spacing carefully

 Missing Scope Entirely:
├─ Requested: "users:readonly" only
├─ Needed: "users:readonly users:manage"
└─ Solution: Add missing scopes

 Wrong Colon Format:
├─ Requested: "users-manage" (dash)
├─ Correct: "users:manage" (colon)
└─ Solution: Use colons, not dashes

Scope Documentation

For Your Team:

Document Your Scopes:

Application: [App Name]
├─ conversations:readonly
│ ├─ Used for: Display conversation history
│ ├─ Endpoints: GET /v2/conversations
│ └─ Rationale: Need to show past interactions
│
├─ conversations:call:control
│ ├─ Used for: Transfer calls between agents
│ ├─ Endpoints: POST /v2/conversations/{id}/participants/calls/transfer
│ └─ Rationale: Core feature for agent desktop
│
├─ users:readonly
│ ├─ Used for: List available agents
│ ├─ Endpoints: GET /v2/users
│ └─ Rationale: Show agent list for transfers
│
└─ Updated: March 2026
   └─ Next Review: September 2026

Scope Change Log:

Date	Change	Reason
2026-03-01	Added: call:control	New transfer feature
2026-01-15	Added: users:readonly	Show agent list

2025-11-20 | Initial: conversations:readonly | Read-only access

API Endpoint to Scope Mapping:

Endpoint	Method	Scope Required
/v2/conversations	GET	conversations:readonly
/v2/conversations	POST	conversations:readonly (new)
/v2/conversations/{id}	GET	conversations:readonly
/v2/users	GET	users:readonly
/v2/users/{id}/presence	PATCH	presence:manage

Key Takeaways: Chapter 5

- **Granular Permissions** - Scopes define exactly what app can do
- **Least Privilege** - Request minimum scopes needed
- **User Consent** - Users see all scopes during authorization
- **Dual Enforcement** - User permissions AND OAuth scopes both required
- **403 Means Missing** - Either scope or permission (or both) missing
- **Standard Format** - Use resource:action or resource:action:scope
- **Space-Separated** - Combine multiple scopes with spaces
- **Review Regularly** - Quarterly scope audits reduce security risk

---

Interview Prep: OAuth Scopes

Question	Answer
What are scopes?	Granular permissions defining what app can access/do
Scope format?	resource:action or resource:action:scope (e.g., conversations:readonly)
How combined?	Space-separated list (e.g., "conversations:readonly users:manage")
User sees scopes?	Yes, during authorization consent screen
All-or-nothing?	Yes, user grants all requested scopes or none
How enforced?	Every API request checked against token scopes
User + scope?	BOTH required (AND logic), not OR

Question	Answer
403 means?	Missing scope or missing permission (or both)
Best practice?	Principle of least privilege - request minimum needed
How update?	Edit OAuth client, add/remove scopes, user must re-auth

---

Document Version

Chapter: 5 of 8
Last Updated: March 2026
Status: Current with OAuth 2.0 standards
Scope: Scope management, enforcement, best practices

OAuth Client Management

Creating OAuth Clients

Step-by-Step: Create an OAuth Client

Access Path:

Admin → Integrations → OAuth → Add client

OR

Menu → IT and Integrations → OAuth → Add client

Procedure:

1. Click "Add Client" Button
   └─ "Add New Client" page appears

2. Enter Application Name (Required)
   ├─ Your application's display name
   ├─ Example: "Salesforce Integration Service"
   ├─ Visible in admin dashboard
   └─ Visible in authorization screens

3. Enter Description (Optional)
   ├─ What the application does
   ├─ Example: "Syncs Salesforce contacts to Genesys every 30 minutes"
   ├─ Helpful for documentation
   └─ Visible in admin UI

4. Select Grant Type(s)
   ├─ Choose one or more:
   │ ├─ Client Credentials (service-to-service)
   │ ├─ Code Authorization / PKCE (web/mobile apps)
   │ ├─ Token Implicit Grant (DEPRECATED - don't use)
   │ └─ SAML2 Bearer (enterprise SSO)

│
├─ Can select multiple grant types
├─ Each grant type has specific settings
└─ Note: Cannot select Implicit after March 2026

5. Click "Next" Button
└─ Proceed to grant-specific configuration

6. Grant-Specific Configuration:

For Client Credentials:
├─ Assign Roles (Required)
│ ├─ Select minimum roles needed
│ ├─ Available roles listed
│ ├─ Application will have these permissions
│ └─ Note: Must have roles in your profile to assign
│
├─ Assign Divisions
│ ├─ Required for role assignment
│ ├─ Roles scoped to divisions
│ ├─ Default: Home Division
│ └─ Update to appropriate divisions
│
└─ Token Duration
├─ Configurable: 300-172,800 seconds
├─ Default: 3600 seconds (1 hour)
├─ SCIM special: up to 450 days
└─ Your choice based on use case

For Authorization Code / PKCE:
├─ Token Duration
│ ├─ Configurable: 300-172,800 seconds
│ ├─ Default: 3600 seconds (1 hour)
│ └─ Recommended: 18 hours (64,800 seconds)
│
├─ Authorized Redirect URIs (Required)
│ ├─ Up to 125 URIs
│ ├─ One per line
│ ├─ Must use HTTPS
│ ├─ Example: https://yourapp.com/callback

```
   │  ├─ Example: https://yourapps.com/auth
   │  ├─ Example: http://localhost:3000/callback (dev)
   │  ├─ Loopback: http://localhost/ (any port)
   │  └─ MUST be exact match (case-sensitive)
   │
   └─ Scopes (Required)
      ├─ Click "Scope" button
      ├─ Select minimum scopes needed
      ├─ Space-separated in requests
      ├─ Example: "conversations:readonly users:readonly"
      └─ Recommended: Least privilege principle

7. Click "Next" Button
   └─ Review configuration

8. Review & Save
   ├─ Verify all settings
   ├─ Check OAuth name
   ├─ Confirm grant type(s)
   ├─ Verify scopes correct
   └─ Click "Save"

9. Client Created Successfully
   ├─ System generates Client ID
   ├─ System generates Client Secret
   ├─ IMPORTANT: Copy secret immediately!
   └─ Secret cannot be retrieved later

10. Display Confirmation
   ├─ Client Name: Your application name
   ├─ Client ID: Unique identifier (public)
   ├─ Grant Types: Selected methods
   ├─ Scopes: Requested permissions
   ├─ Created By: Your username
   ├─ Created Date: Timestamp
   └─ Client Secret: (hidden after creation)

11. Finish
   └─ Client ready to use
```

Typical Client Example:

Name: Salesforce Contact Sync
Grant Type: Client Credentials
Roles: External Contact Manager, Agent
Divisions: Home Division
Token Duration: 86400 (1 day)
Scopes: externalcontacts:manage
Created: 2026-03-13

Client Secret Management (Critical - March 2026 Change)

⚠ IMPORTANT SECURITY UPDATE (March 2026)

Previous Behavior:
├─ Client secret visible in admin UI anytime
├─ Could view secret for existing clients
├─ Increased exposure risk
└─ Security concern

Current Behavior (March 2026):
├─ Client secret only shown at creation
├─ Cannot view secret after creation
├─ Must copy immediately when created
├─ Cannot retrieve via API
└─ Enhanced security

Action Required at Client Creation:

1. When Client Created:
   ├─ Page displays client secret
   ├─ Only time you see it
   └─ Immediately copy and store

2. Copy Secret:

├─ Click "Copy" button
├─ OR manually select and copy
└─ Keep safe!

3. Store Secret Securely:
├─ Secure vault (recommended)
├─ Environment variables
├─ NOT in code
├─ NOT in git repository
├─ NOT in logs
└─ Encrypted storage

4. Acknowledge:
├─ Checkbox: "I have copied and stored the secret"
├─ Verify before checking
└─ Only way to proceed

5. If You Lose It:
├─ Cannot retrieve from UI
├─ Click "Generate new secret"
├─ New secret replaces old
├─ Old secret no longer works
└─ Update application immediately

Secure Storage Solutions:

HashiCorp Vault:
├─ Enterprise secret management
├─ Encryption, rotation, audit
├─ Recommended: Production environments
└─ Access via API

AWS Secrets Manager:
├─ AWS-native secret storage
├─ Encryption, rotation, audit
├─ Recommended: AWS environments
└─ IAM-based access control

Azure Key Vault:
├─ Azure-native solution

├─ Encryption, versioning
├─ Recommended: Azure environments
└─ RBAC access control

Environment Variables (Dev Only):
├─ .env file (development)
├─ Never commit to git
├─ NOT for production
├─ Simple for local development
└─ Use .gitignore

Docker Secrets:
├─ Container orchestration
├─ Swarm/Kubernetes
├─ Encrypted storage
└─ Recommended: Container deployments

Never Store:
- In application code
- In git repository
- In version control
- In logs
- In configuration files
- In plain text
- In comments
- In documentation

OAuth Client Security

Security Best Practices:

Secret Rotation:

Timeline:
├─ Rotate monthly minimum
├─ Before employee departures
├─ After any exposure

├─ If suspected compromise
└─ Automated via CI/CD

Process:

1. Generate New Secret:
   ├─ Click "Generate new secret"
   ├─ Confirm action
   └─ New secret displayed once

2. Update Application:
   ├─ Update configuration
   ├─ Dual-use period: old + new both work
   ├─ Monitor for errors
   └─ Verify new secret works

3. Verify Working:
   ├─ Test authentication
   ├─ Check logs for success
   ├─ No 401 errors
   └─ All requests working

4. Remove Old Secret:
   ├─ Old automatically stops working
   ├─ After brief dual-use period
   ├─ No action needed
   └─ Can't revert to old

Audit Logging:

Log These Events:
├─ OAuth client created
├─ Secret regenerated
├─ Scopes added/removed
├─ Roles assigned/removed
├─ Client deleted
├─ Access failures
└─ Unusual activity

What to Log:

├─ Timestamp
├─ Admin user
├─ Action taken
├─ Client affected
├─ Result (success/failure)
└─ NOT the secret itself

Monitoring:

Monitor For:
├─ Unexpected client creation
├─ Secret rotation outside schedule
├─ Failed authentication attempts
├─ Unusual scope usage
├─ Access pattern changes
├─ Clients not used regularly
└─ Suspicious activity

Permissions:

Who Can Create Clients?
├─ Admin users with "oauth:client:add" permission
├─ Typically: Super Admin, OAuth Admin role
└─ Verify in your organization

Who Can View Clients?
├─ Admin users
├─ Users with "oauth:client:view" permission
└─ Consider minimizing access

Who Can Delete Clients?
├─ Admin users
├─ OAuth Admin role
└─ Approval process recommended

Compliance Requirements:

HIPAA:
├─ Requires MFA for admin access
├─ 15-minute idle timeout enforced

├─ Audit logging required
└─ Client rotation documented

PCI-DSS:
├─ Secure secret storage required
├─ No hardcoded secrets
├─ Regular rotation required
└─ Access control required

SOC 2:
├─ Audit trails for all changes
├─ Access control enforcement
├─ Change management process
└─ Incident response documented

GDPR:
├─ Data processing log required
├─ Right to access supported via API
├─ Data retention policies
└─ Deletion/export capabilities

OAuth Client Lifecycle

Stages:

1. Creation
   ├─ Admin creates client
   ├─ Scopes assigned
   ├─ Roles assigned (if Client Credentials)
   ├─ Secret generated
   └─ Client ID provided

2. Configuration
   ├─ Redirect URIs registered (if Auth Code)
   ├─ Scopes finalized
   ├─ Token duration set
   └─ Roles/divisions confirmed

3. Usage
   ├─ Applications authenticate with client
   ├─ Users authorize app (if Code grant)
   ├─ Tokens issued and used
   ├─ API calls made
   └─ Regular operations

4. Monitoring
   ├─ Track usage patterns
   ├─ Monitor authentication success
   ├─ Alert on anomalies
   ├─ Quarterly scope review
   └─ Annual security audit

5. Maintenance
   ├─ Rotate secrets monthly
   ├─ Update scopes as needed
   ├─ Verify still in use
   ├─ Remove unused clients
   └─ Document changes

6. Deactivation
   ├─ Determine if still needed
   ├─ Plan removal
   ├─ Notify application owners
   ├─ Provide replacement if needed
   └─ Allow migration period

7. Deletion
   ├─ Remove old client
   ├─ Tokens immediately invalid
   ├─ Applications get 401 errors
   ├─ Audit log records deletion
   └─ Cannot be recovered

Timeline:

Creation → Configuration → Usage → Monitoring → Maintenance → Deactivation → Deletion
0 days 1 day Day 1+ Day 30+ Day 30+ Day 365+ Day 380+

Common OAuth Client Configurations

Configuration 1: Web Application (Node.js + React)

Grant Type: Authorization Code

Redirect URIs:

├─ https://yourapp.com/callback

├─ https://yourapp.com/auth

└─ http://localhost:3000/callback (dev)

Scopes:

├─ conversations:readonly

├─ users:readonly

└─ presence:manage

Token Duration: 3600 (1 hour)

Use Case:

├─ User logs in via browser

├─ Backend handles token exchange

├─ Backend stores refresh token

└─ User can revoke access anytime

Configuration 2: Service Integration (Salesforce Sync)

Grant Type: Client Credentials

Roles:

├─ External Contact Manager

└─ Scheduler (if needed)

Divisions: Home Division

Scopes:

├─ externalcontacts:manage

├─ users:readonly
└─ scheduling:readonly

Token Duration: 86400 (1 day)

Use Case:
├─ Automated sync service
├─ No user interaction
├─ Runs on schedule
└─ Application acts as itself

Configuration 3: Mobile App with Backend

Grant Type: Authorization Code + PKCE
Redirect URIs:
├─ myapp://oauth/callback
├─ https://myapp.com/callback
└─ http://localhost:3000/callback (dev)

Scopes:
├─ conversations:readonly
├─ conversations:call:control
├─ users:readonly
└─ presence:manage

Token Duration: 3600 (1 hour)

Use Case:
├─ Mobile app user login
├─ PKCE for public client security
├─ Backend stores refresh token
└─ App cannot store client_secret

Configuration 4: Single-Page Application (SPA)

Grant Type: Authorization Code + PKCE
Redirect URIs:
├─ https://yourapp.com/

├─ https://yourapp.com/callback
└─ http://localhost:3000/ (dev)

Scopes:
├─ conversations:readonly
├─ users:readonly
└─ presence:manage

Token Duration: 3600 (1 hour)

Use Case:
├─ JavaScript SPA
├─ No backend (serverless)
├─ PKCE prevents code interception
├─ Tokens stored in memory only

Configuration 5: Bot/Virtual Agent

Grant Type: Client Credentials
Roles:
├─ Agent
└─ Virtual Agent (if available)

Divisions: Home Division

Scopes:
├─ conversations:readonly
├─ conversations:external:contact:add
├─ knowledge:readonly
└─ users:readonly

Token Duration: 86400 (1 day)

Use Case:
├─ Chatbot integration
├─ No user context
├─ Reads knowledge base
└─ Fully automated

Configuration 6: Admin Tool/Dashboard

Grant Type: Authorization Code
Redirect URIs:
├─ https://admin.yourcompany.com/callback
└─ http://localhost:8080/callback (dev)

Scopes:
├─ users:manage
├─ roles:manage
├─ oauth:client:manage
└─ admin:org:manage

Token Duration: 3600 (1 hour)

Use Case:
├─ Admin portal
├─ Full management capabilities
├─ User authentication
└─ Administrative access control

Troubleshooting OAuth Clients

Problem: Client Creation Fails

Check:
├─ Permission: Do you have oauth:client:add?
├─ Roles: Do you have the roles you're assigning?
├─ Scopes: Are scope names spelled correctly?
└─ Divisions: Do divisions exist?

Solution:
├─ Request oauth:client:add permission
├─ Use roles you have assigned
├─ Check scope names in API Explorer
└─ Use existing divisions

Problem: Client Secret Lost/Forgotten

Cannot Retrieve:
├─ Old secret: Cannot view (security feature)
├─ No recovery method
├─ No admin override
└─ Design for security

Solution:
├─ Generate new secret
├─ Update application
├─ Delete old client if not needed
└─ Document in future

Steps:
1. Click "Generate new secret"
2. Copy new secret
3. Update application config
4. Test authentication
5. Verify working

Problem: 401 Unauthorized on API Calls

Possible Causes:
├─ Token expired: Get fresh token
├─ Token revoked: Authenticate again
├─ Client credentials wrong: Check spelling
├─ Client no longer exists: Recreate
└─ Secret wrong: Regenerate

Troubleshooting:
1. Check token hasn't expired
2. Try authenticating again (fresh token)
3. Verify client_id and client_secret spelling
4. Check client exists in Admin UI
5. If regenerated secret, update application
6. Monitor logs for pattern

Problem: Scopes Not Working

User Still Can't Access API:

Causes:
├─ Wrong scope name
├─ User missing permission
├─ Token doesn't have scope
├─ Both scope and permission needed
└─ Other 403 reason

Check:
1. API Explorer: What scope required?
2. Token: Does it have that scope?
3. User: Does user have role permission?
4. Both: Scope AND permission needed
5. 403 error: One or both missing

Problem: Application Not Working After Update

Something Changed:

Check:
├─ Secret regenerated? Update app config
├─ Scopes changed? Users must re-authorize
├─ Roles changed? Token might lack permission
├─ Token duration changed? Should still work
└─ Redirect URI changed? Registration needed

Fix:
├─ If secret: Update app immediately
├─ If scopes: Get new token (user re-auth)
├─ If roles: Assign correct roles
└─ If URI: Re-register in Admin UI

Best Practices: OAuth Client Management

Security:

☐ Store secrets in secure vault

☐ Rotate monthly minimum

☐ Never commit to git

☐ Use environment variables

☐ Encrypt at rest and in transit

☐ Audit all changes

☐ Monitor for anomalies

☐ Principle of least privilege

Operations:

☐ Document each client's purpose

☐ Keep contact info for app owner

☐ Review quarterly

☐ Remove unused clients

☐ Plan secret rotation schedule

☐ Test after any changes

☐ Verify working regularly

☐ Monitor token usage

Compliance:

☐ Meet HIPAA requirements

☐ Meet PCI-DSS requirements

☐ Meet SOC 2 requirements

☐ Document compliance steps

☐ Keep audit logs

☐ Review permissions quarterly

☐ Maintain change log

☐ Plan for incidents

Key Takeaways: Chapter 6

- **Admin-Only Access** - Only admins can create OAuth clients
- **Secret View-Once** - Client secret only shown at creation (March 2026 change)
- **Secure Storage Required** - Use vault, not code/git/environment
- **Monthly Rotation** - Standard practice for secret updates
- **Dual Grant Types** - Can select multiple grant types per client
- **Role & Scope** - Both required for Client Credentials
- **Redirect URIs** - Must be exact match (case-sensitive)
- **Audit Everything** - Log client creation, secret rotation, deletions

---

Interview Prep: OAuth Client Management

Question	Answer
Where create OAuth client?	Admin → Integrations → OAuth → Add client
Who can create?	Users with oauth:client:add permission
Client secret visibility?	Only shown once at creation (March 2026 change)
If secret lost?	Generate new secret, update application
Secret storage?	Secure vault (HashiCorp, AWS, Azure)
Secret rotation?	Monthly minimum, before departures, after exposure
Redirect URIs?	Up to 125, must be HTTPS, exact match required
Scopes required?	Yes, principle of least privilege
Token duration?	Default 1 hour, configurable 300-172,800 seconds
Client deletion?	Immediate, tokens invalid, can't recover

Document Version

Chapter: 6 of 8
Last Updated: March 2026
Status: Current with OAuth 2.0 standards
Scope: Client creation, management, security

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

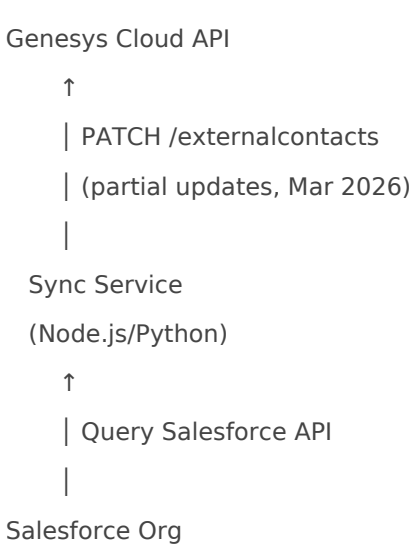
Real-World Integration Patterns & Deployment

Common Integration Patterns

Pattern 1: Salesforce ↔ Genesys Contact Sync

Scenario: Synchronize Salesforce contacts to Genesys external contacts

Architecture:



Frequency: Every 30 minutes
Direction: Salesforce → Genesys (one-way)
Trigger: Scheduled job (cron)
Volume: ~1,000 contacts per sync

Authentication:

Salesforce:
├─ OAuth 2.0 (your existing setup)
├─ Service account or user credentials
└─ Connected app registered

Genesys:
├─ OAuth 2.0 Client Credentials
├─ Role: External Contact Manager
├─ Scopes: externalcontacts:manage
└─ Monthly secret rotation

Data Flow:

1. Query Salesforce:
   GET /services/data/v60.0/query
   SELECT Id, Email, Phone, Name, AccountId
   FROM Contact
   WHERE LastModifiedDate > :lastSync

2. Transform Data:
   ├─ Normalize phone (E.164)
   ├─ Validate email
   ├─ Map custom fields
   ├─ Add external ID (Salesforce ID)
   └─ Group by action (create/update)

3. Sync to Genesys:
   ├─ New contacts: POST /externalcontacts/contacts
   ├─ Updates: PATCH /externalcontacts/contacts/{id}
   ├─ Batches: Group every 50-100
   └─ Handle errors: Log and retry

4. Track Progress:
   ├─ Timestamp last successful sync
   ├─ Count created/updated/failed
   ├─ Send email notification
   └─ Log all activities

Implementation Highlights:

Error Handling:
├─ 409 Conflict: Already exists (update instead)
├─ 400 Bad Request: Invalid data (log and skip)
├─ 429 Rate Limited: Backoff and retry
├─ 503 Unavailable: Retry with backoff
└─ All other: Fail and alert

Partial Updates (PATCH):
├─ Update only changed fields
├─ Prevents overwriting unmanaged data
├─ Reduces payload size
├─ Better for CRM sync
└─ Available March 2026+

Idempotency:
├─ Use externalId for matching
├─ Prevent duplicate creates
├─ Retry-safe operations
├─ Track processed records
└─ Handle partial failures

Performance:
├─ Batch 50 contacts per request
├─ 1,000 contacts: ~20 requests
├─ Completes in < 5 minutes
├─ Well under rate limits
└─ Minimal API footprint

Benefits:
├─ Single source of truth
├─ Real-time contact availability
├─ No duplicate entry
├─ Reduces manual effort
└─ Improved agent experience

Pattern 2: Nightly Analytics Report Generation

Scenario: Generate daily contact center reports

Architecture:

Genesys Cloud Analytics API
↑
Report Generator Service
(scheduled, 2:00 AM daily)
↓
Email Distribution List

Authentication:

Client Credentials:
├─ Scopes: analytics:conversationDetail
├─ Token Duration: 1 hour (sufficient)
├─ No user interaction needed
└─ Fully automated

Workflow:

1. Authenticate (2:00 AM)
   POST /oauth/token
   grant_type: client_credentials
   Result: Access token

2. Query Analytics (2:01 AM)
   ├─ Yesterday's date range
   ├─ All queues/teams
   ├─ Conversations aggregate
   ├─ Service Level, AHT, ASA, Abandon Rate
   └─ Multiple requests (by dimension)

3. Generate Report (2:05 AM)
   ├─ Process data
   ├─ Create charts/graphs
   ├─ Format professionally

├─ Create PDF
└─ Calculate YoY trends

4. Distribute (2:10 AM)
├─ Email PDF
├─ To stakeholders
├─ With summary
└─ Archive for history

Report Contents:

Executive Summary:
├─ Total conversations: 5,432
├─ Service Level: 87% (Target: 80%)
├─ Avg Handle Time: 8:32 (Target: < 10min)
├─ Abandon Rate: 3.2% (Target: < 5%)
└─ Net Change vs yesterday: +2.1%

By Queue:
├─ Queue name
├─ Conversations
├─ Service Level
├─ AHT
├─ ASA
└─ Agents

Trending:
├─ Last 7 days
├─ Last 30 days
├─ YoY comparison
├─ Alerts (SLA failures)
└─ Recommendations

Implementation Highlights:

Scheduling:
├─ Cron: "0 2 * * *" (2:00 AM daily)
├─ Timezone: Your organization's TZ
├─ Alerting: If job fails

└─ Retry: Automatic

API Calls:
├─ /analytics/conversations/aggregates (multi-call)
├─ Total: 5-10 requests
├─ Completes in < 5 minutes
└─ No rate limit issues

Report Generation:
├─ Tool: ReportLab (Python) or similar
├─ Format: PDF
├─ Design: Professional template
└─ Data: Charts and tables

Email Distribution:
├─ Tool: SendGrid or SMTP
├─ Recipients: DL
├─ Schedule: Exactly 2:15 AM
├─ Archive: Save copy for history
└─ Tracking: Note delivery status

Error Handling:
├─ Authentication fails: Alert operations
├─ API call fails: Retry with backoff
├─ Report gen fails: Fallback to text
├─ Email fails: Retry next hour
└─ All failures: Log and notify

Benefits:
├─ Automated daily reporting
├─ Saves analyst 30 minutes/day
├─ Consistent delivery
├─ Stakeholders stay informed
└─ Data-driven decisions

Pattern 3: Real-Time Agent Status to Dashboard

Scenario: Display real-time agent availability in web dashboard

Architecture:

Genesys Cloud (WebSocket)
↓ /v2/users/{id}/presence
│ (WebSocket events)
↓
Node.js Backend
(WebSocket server)
↓ Socket.IO
↓
Browser Clients
(Dashboard)

Authentication:

User Login:
├─ Authorization Code Grant
├─ User authenticates once
├─ Backend stores refresh_token
├─ Token includes necessary scopes
└─ presence:manage scope required

WebSocket Connection:
├─ Uses existing access_token
├─ Maintains connection
├─ Real-time events
└─ Automatic reconnection

Workflow:

1. User Logs In to Dashboard:
   ├─ Authorization Code flow
   ├─ User sees consent screen
   ├─ Backend gets access_token
   ├─ Backend stores refresh_token
   └─ User authenticated

2. Backend Establishes WebSocket:

POST /api/v2/notifications/channels
└─ Opens long-lived connection

3. Subscribe to Presence Events:
PUT /api/v2/notifications/channels/{channelId}/subscriptions

Subscriptions:
├─ v2.users.{user1}.presence
├─ v2.users.{user2}.presence
├─ v2.users.{user3}.presence
└─ For all agents

4. Receive Real-Time Events:

```
{
 "eventBody": {
  "userId": "user-123",
  "presenceDefinition": {
   "systemPresence": "available",
   "customPresences": [...]
  }
 }
}
```

5. Update Dashboard:
├─ Forward event to browsers (Socket.IO)
├─ Update agent status display
├─ Change color/icon
├─ Show "Available", "Break", "Busy", etc
└─ Real-time synchronization

Implementation Highlights:

Subscription Efficiency:
├─ Single WebSocket connection
├─ Multiple subscriptions
├─ vs: 100 polling requests/minute
├─ 99% reduction in API calls
└─ Real-time delivery

Connection Management:

├─ Maintain connection
├─ Automatic reconnection on failure
├─ Health checks
├─ Graceful degradation
└─ Error handling

Scalability:
├─ 1 backend: 100+ agent subscriptions
├─ vs: Polling → rate limited
├─ WebSocket: Event-driven
├─ Lower CPU usage
└─ Lower bandwidth

Error Recovery:
├─ Connection drops: Auto-reconnect
├─ Subscription fails: Retry
├─ Event deserialization: Log and skip
├─ Token expires: Refresh and reconnect
└─ Graceful fallback: Polling backup

Benefits:
├─ Real-time agent status
├─ Sub-second updates
├─ No polling overhead
├─ Better user experience
├─ Reduced infrastructure load
└─ Scalable solution

Deployment Strategies

Strategy 1: Development Environment

Setup:

OAuth Client:
├─ Grant Type: Authorization Code + PKCE

├─ Redirect URI: http://localhost:3000/callback
├─ Scopes: conversations:readonly
├─ Token Duration: 3600 (1 hour)
└─ Created by: Development team

Client Credentials (if needed):
├─ Grant Type: Client Credentials
├─ Scopes: externalcontacts:manage
├─ Token Duration: 3600
└─ Role: External Contact Manager (dev only)

Secrets Storage:
├─ .env file (local development)
├─ .gitignore entries:
│ ├─ .env
│ ├─ .env.local
│ └─ credentials/
├─ Example .env:
│ ├─ GENESYS_CLIENT_ID=dev-client-id
│ ├─ GENESYS_CLIENT_SECRET=dev-secret
│ └─ GENESYS_REGION=mypurecloud.com
└─ Never commit secrets!

Configuration:
├─ Use environment variables
├─ Different per developer
├─ Allow local overrides
├─ Development region specified
└─ Non-production data only

Testing:
├─ Unit tests: Mock API responses
├─ Integration tests: Dev Genesys org
├─ Manual testing: Full flow
├─ Error scenario testing
└─ Rate limit testing

CI/CD Pipeline:
├─ Run on: Developer machine

├─ Linting: Yes
├─ Unit tests: Yes
├─ Build: Yes
├─ Deploy: Local only
└─ Secrets: Via .env (not checked in)

Monitoring:
├─ Logging: Console output
├─ Debugging: Browser DevTools
├─ API calls: Inspect requests
├─ Errors: Stack traces
└─ Performance: Basic measurements

Strategy 2: Staging Environment

Setup:

OAuth Clients:
├─ Separate from production
├─ Authorization Code client
├─ Client Credentials client
├─ Different secrets
└─ Same Genesys region (or test region)

Secrets Storage:
├─ Environment variables (CI/CD platform)
├─ Encrypted vault
├─ GitHub Actions secrets / GitLab CI variables
├─ Rotate monthly
└─ Different from production

Configuration:
├─ Staging-specific settings
├─ Same region as production
├─ Test data only
├─ Verbose logging enabled
└─ Performance testing configured

Testing:
├─ Full integration tests
├─ End-to-end workflows
├─ Load testing (small scale)
├─ Error scenario testing
├─ Performance baselines
└─ Compatibility testing

CI/CD Pipeline:
├─ Trigger: On PR/Merge to staging branch
├─ Linting: Yes
├─ Unit tests: Yes
├─ Integration tests: Yes
├─ Build: Yes
├─ Deploy: Automated
├─ Smoke tests: Post-deploy
└─ Notify: On success/failure

Monitoring:
├─ Application logs: Aggregated
├─ Error tracking: Sentry/similar
├─ Performance monitoring: APM
├─ Uptime monitoring: Synthetic
└─ Alerts: To development team

Approval Process:
├─ Code review: Required
├─ Tests: Must pass
├─ Deployment: Semi-automated
├─ Rollback: Available
└─ Notify stakeholders

Data Management:
├─ Test data only
├─ Reset daily/weekly
├─ Production data: Never
├─ GDPR compliant
└─ Privacy respected

Strategy 3: Production Environment

Setup:

OAuth Clients:
├─ Separate production clients
├─ Multiple clients: Web app, services, etc.
├─ Different secrets per environment
├─ Rotate monthly minimum
├─ Monthly rotation documented
└─ Secure secret storage required

Secrets Storage:

Critical - Use Secure Vault:
├─ HashiCorp Vault (recommended)
├─ AWS Secrets Manager
├─ Azure Key Vault
├─ Google Cloud Secrets
└─ Store & access policies:
  ├─ Encrypt at rest
  ├─ Encrypt in transit
  ├─ Audit all accesses
  ├─ RBAC (role-based)
  ├─ Rotation tracking
  └─ Alert on unauthorized access

Configuration:

Environment-Specific:
├─ Production region only
├─ Performance settings optimized
├─ Logging: Moderate (balance vs storage)
├─ Monitoring: Comprehensive
├─ Alerting: All channels
└─ Recovery procedures: Documented

CI/CD Pipeline:

Strict Requirements:

├─ All tests: Must pass

├─ Code review: Mandatory

├─ Approval: Required (2+ reviewers)

├─ Build: Automated & verified

├─ Deploy: Gated release

├─ Smoke tests: Post-deploy (critical)

├─ Rollback: Instant available

└─ Notify: Multiple teams

Deployment Procedure:

1. Code Merge (to main branch)

   ├─ Tests pass

   ├─ Reviews approved

   └─ CI/CD starts

2. Build (automated)

   ├─ Code compiled

   ├─ Tests run

   ├─ Artifact created

   └─ Ready for deploy

3. Stage (automated)

   ├─ Deploy to staging

   ├─ Smoke tests run

   ├─ If all pass: await approval

   └─ If any fail: block deployment

4. Approve (manual gate)

   ├─ Engineering manager: approval

   ├─ On-call engineer: ready

   ├─ Time window: Business hours preferred

   └─ Cancellation: Available anytime

5. Deploy (automated)

   ├─ Deploy to production

   ├─ Gradual rollout (optional)

   ├─ Monitor deployment

   ├─ Health checks pass

└─ Notify team

6. Monitor (continuous)
   ├─ Watch logs
   ├─ Watch metrics
   ├─ Watch errors
   ├─ Alert threshold: Low
   └─ Quick response ready

7. Rollback (if needed)
   ├─ Detected: Issue identified
   ├─ Decision: Rollback authorized
   ├─ Execute: One command
   ├─ Verify: Health checks pass
   └─ Notify: Team aware

Monitoring:

Comprehensive Observability:
├─ Application logging (ELK/Splunk)
├─ Error tracking (Sentry/Rollbar)
├─ APM (New Relic/DataDog)
├─ Uptime monitoring (Synthetic)
├─ Custom metrics
├─ Infrastructure monitoring
└─ Security monitoring

Alerting:

Critical (Immediate - Page):
├─ Application down (503/504)
├─ Authentication failures spike
├─ Database connection errors
├─ Memory/CPU critically high
├─ Deployment failed
└─ Security incident

High Priority (30 min - Slack):
├─ Error rate > 5%
├─ Response time spike

├─ Rate limit approached
├─ Token refresh failures
├─ API quota exceeded
└─ Unusual traffic pattern

Medium Priority (1 hour - Email):
├─ Minor errors
├─ Performance degradation
├─ Deprecated API usage
├─ Scheduled job delayed
└─ Configuration drift

Disaster Recovery:

Backup & Recovery:
├─ Database: Daily snapshots
├─ Configuration: Version controlled
├─ Secrets: Vault with audit logs
├─ RTO: < 1 hour
├─ RPO: < 15 minutes
└─ Tested monthly

Communication:

During Incident:
├─ Status page: Updated immediately
├─ Slack: Team notification
├─ Email: If long duration
├─ Customers: If customer-facing
└─ Escalation: Clear path

Post-Incident:
├─ Post-mortem: Scheduled
├─ Root cause: Identified
├─ Fix: Implemented
├─ Prevention: For future
└─ Lessons learned: Documented

Compliance:

Security Requirements:
├─ PCI-DSS: For payment data
├─ HIPAA: For health data
├─ SOC 2: For SaaS
├─ GDPR: For EU data
└─ Industry-specific: As applicable

Audit & Compliance:
├─ Audit logs: Retained 2+ years
├─ Access logs: Who did what when
├─ Change logs: All changes recorded
├─ Compliance checks: Quarterly
└─ External audits: Annual

Troubleshooting Deployments

Common Issues:

Problem: Authentication Fails in Production

Causes:
├─ Secret rotated, app not updated
├─ Secret expired (if revoked)
├─ Client deleted
├─ Secret wrong (typo)
└─ Wrong client for environment

Diagnosis:
├─ Check error: Invalid credentials
├─ Verify secret matches what stored
├─ Verify client still exists
├─ Check expiration date
└─ Try authentication manually

Fix:
├─ If secret wrong: Update immediately
├─ If client deleted: Recreate

├─ If expired: Generate new secret
├─ If revoked: Verify it's revoked
└─ Test after fix

Prevention:
├─ Document secret location
├─ Rotate on schedule
├─ Test auth before deploy
├─ Use vault for storage
└─ Alert before expiration

Problem: Rate Limiting in Production

Symptoms:
├─ 429 errors increasing
├─ Request latency increasing
├─ API calls slowing down
├─ Partial failures
└─ Customers reporting issues

Diagnosis:
├─ Check request volume
├─ Check request frequency
├─ Identify hot endpoints
├─ Check X-Rate-Limit-Remaining
└─ Calculate current rate

Fix (Short-term):
├─ Implement backoff
├─ Reduce request frequency
├─ Queue requests
├─ Reduce batch size
└─ Wait for window reset

Fix (Long-term):
├─ Use bulk endpoints
├─ Implement caching
├─ Use WebSocket events
├─ Optimize queries
└─ Consider enterprise limits

Prevention:
├─ Load test before production
├─ Monitor rate limit headers
├─ Alert when approaching limit
├─ Design for pagination
└─ Use bulk APIs from start

Problem: Token Refresh Failing

Symptoms:
├─ API calls return 401
├─ Refresh token errors
├─ Users getting logged out
├─ Authentication failing
└─ Errors in logs

Diagnosis:
├─ Check refresh token expired
├─ Check client exists
├─ Check secret correct
├─ Check token lifetime
└─ Try manual refresh

Fix:
├─ If expired: User must re-authenticate
├─ If client wrong: Update config
├─ If secret wrong: Update secret
├─ If lifetime: Adjust config
└─ Test refresh manually

Prevention:
├─ Refresh proactively (5 min before)
├─ Monitor refresh success rate
├─ Alert on refresh failures
├─ Document refresh logic
└─ Test token refresh

Problem: Data Loss / Sync Issues

Symptoms:
├─ Contacts not syncing
├─ Data inconsistencies
├─ Partial updates missing
├─ Database conflicts
└─ Customers reporting issues

Diagnosis:
├─ Check sync logs
├─ Verify API calls succeeded
├─ Check for conflict errors
├─ Verify data format
└─ Compare source vs target

Fix:
├─ Re-run sync
├─ Correct data format
├─ Handle conflicts
├─ Verify completeness
└─ Reconcile manually if needed

Prevention:
├─ Implement idempotency
├─ Use external IDs
├─ Log all changes
├─ Verify sync completion
├─ Regular reconciliation
├─ Alerts on failures
└─ Testing with real data

Problem: Performance Degradation

Symptoms:
├─ Slow API response times
├─ High latency (p99)
├─ User complaints
├─ Dashboard sluggish
└─ Timeout errors

Diagnosis:

├─ Check API latency
├─ Check application latency
├─ Check database latency
├─ Check network latency
└─ Identify bottleneck

Fix (Short-term):
├─ Scale up application
├─ Scale up database
├─ Clear cache
├─ Reduce requests
└─ Optimize queries

Fix (Long-term):
├─ Use caching layer
├─ Optimize database
├─ Use CDN
├─ Async processing
└─ Horizontal scaling

Prevention:
├─ Load test regularly
├─ Monitor latency
├─ Alert on degradation
├─ Capacity planning
└─ Performance budget

Key Takeaways: Chapter 8

- **Pattern Diversity** - Different patterns for different scenarios
- **Authentication Clear** - Always OAuth 2.0 (Client Credentials or Code)
- **Scheduling Critical** - Cron jobs for reliable automation
- **Error Handling** - Implement retry logic with backoff
- **Data Quality** - Validate and normalize before syncing
- **Deployment Gates** - Approval required for production
- **Monitoring Essential** - Know when things break
- **Documentation Important** - For debugging and maintenance

---

Interview Prep: Integration Patterns

Question	Answer
Salesforce sync pattern?	Query Salesforce, transform, batch POST/PATCH to Genesys
Report generation?	Scheduled job (cron), query analytics, generate PDF, email
Real-time status?	WebSocket subscriptions, event-driven updates, low overhead
Authentication type?	Client Credentials for services, Auth Code for users
Backoff strategy?	3, 9, 27 seconds, then 5-min increments
Error handling?	Retry on 429/5XX, fail on 4XX (except 401)
Data sync quality?	Validate, normalize, idempotent, external IDs
Deployment gate?	Approval required, smoke tests pass, monitoring ready
Secret storage?	Secure vault (Hashicorp, AWS, Azure)
Monitoring?	Logs, metrics, errors, alerts (critical/high/medium)

Document Version

Chapter: 8 of 8
Last Updated: March 2026
Status: Current with OAuth 2.0 & API standards
Scope: Integration patterns, deployment strategies, troubleshooting

API Endpoints Reference

Overview

The Genesys Cloud API provides REST endpoints for managing contacts, conversations, users, and other platform resources. This reference covers the most commonly used endpoints for integration work.

Base URL: `https://api.mypurecloud.com/api/v2`

Authentication: OAuth 2.0 Bearer Token (see Chapter 11: OAuth Client Management)

Contacts API

Get All Contacts

Endpoint: `GET /contacts`

Description: Retrieve a paginated list of contacts.

Query Parameters:

- `pageSize` (integer, optional): Contacts per page (default: 25, max: 100)
- `pageNumber` (integer, optional): Page number (default: 1)
- `sortOrder` (string, optional): `asc` or `desc` (default: `asc`)

Example Request:

```
curl -X GET "https://api.mypurecloud.com/api/v2/contacts?pageSize=50&pageNumber=1" \
  -H "Authorization: Bearer {ACCESS_TOKEN}"
```

Example Response:

```
{
  "entities": [
```

```
    {
      "id": "contact-123",
      "firstName": "John",
      "lastName": "Doe",
      "email": "john.doe@example.com",
      "phoneNumbers": [
        {
          "number": "+15551234567",
          "type": "MOBILE"
        }
      ],
      "externalOrganization": {
        "id": "org-456",
        "name": "Acme Corp"
      },
      "externalId": "sf-contact-789",
      "dateCreated": "2026-03-14T10:30:00Z",
      "dateModified": "2026-03-14T10:30:00Z"
    }
  ],
  "pageNumber": 1,
  "pageSize": 50,
  "total": 2500
}
```

Create a Contact

Endpoint: `POST /contacts`

Description: Create a new contact.

Request Body:


```
{
  "firstName": "Jane",
  "lastName": "Smith",
  "email": "jane.smith@example.com",
  "phoneNumbers": [
    {
```

```
      "number": "+15559876543",
      "type": "WORK"
    }
  ],
  "externalOrganization": {
    "id": "org-789",
    "name": "Tech Solutions Inc"
  },
  "externalId": "sf-contact-001"
}
```


Required Fields:

- `firstName` (string): Contact's first name
- `lastName` (string): Contact's last name

Optional Fields:

- `email` (string): Email address
- `phoneNumbers` (array): Phone number objects with `number` and `type`
- `externalOrganization` (object): Org reference with `id` and `name`
- `externalId` (string): External system ID (for deduplication)

Example Response:


```
{
  "id": "contact-999",
  "firstName": "Jane",
  "lastName": "Smith",
  "email": "jane.smith@example.com",
  "phoneNumbers": [
    {
      "number": "+15559876543",
      "type": "WORK"
    }
  ],
  "dateCreated": "2026-03-14T11:00:00Z"
}
```

Update a Contact (Full)

Endpoint: `PUT /contacts/{contactId}`

Description: Replace entire contact record.

Example Request:

```
curl -X PUT "https://api.mypurecloud.com/api/v2/contacts/contact-999" \
  -H "Authorization: Bearer {ACCESS_TOKEN}" \
  -H "Content-Type: application/json" \
  -d '{
    "firstName": "Jane",
    "lastName": "Smith",
    "email": "jane.smith.new@example.com"
  }'
```

Update a Contact (Partial)

Endpoint: `PATCH /contacts/{contactId}`

Description: Partially update contact (new in March 2026). Reduces risk of data overwrites.

Example Request:

```
curl -X PATCH "https://api.mypurecloud.com/api/v2/contacts/contact-999" \
  -H "Authorization: Bearer {ACCESS_TOKEN}" \
  -H "Content-Type: application/json" \
  -d '{
    "email": "jane.smith.new@example.com",
    "phoneNumbers": [
      {
        "number": "+15551111111",
        "type": "MOBILE"
      }
    ]
  }'
```

Search for Contacts by Phone Number

Endpoint: `GET /contacts/search`

Query Parameters:

- `q` (string): Search query (phone number, email, name)

Example Request:

```
curl -X GET "https://api.mypurecloud.com/api/v2/contacts/search?q=%2B15551234567" \
  -H "Authorization: Bearer {ACCESS_TOKEN}"
```

Example Response:

```
{
  "results": [
    {
      "id": "contact-123",
      "firstName": "John",
      "lastName": "Doe",
      "phoneNumbers": [
        {
          "number": "+15551234567",
          "type": "MOBILE"
        }
      ]
    }
  ],
  "total": 1
}
```

Delete a Contact

Endpoint: `DELETE /contacts/{contactId}`

Description: Remove a contact.

Example Request:

```
curl -X DELETE "https://api.mypurecloud.com/api/v2/contacts/contact-999" \
  -H "Authorization: Bearer {ACCESS_TOKEN}"
```

Response: 204 No Content (success)

Conversations API

Get Recent Conversations

Endpoint: `GET /conversations`

Query Parameters:

- `pageSize` (integer, optional): Conversations per page (default: 25, max: 100)
- `pageNumber` (integer, optional): Page number (default: 1)

Example Request:

```
curl -X GET "https://api.mypurecloud.com/api/v2/conversations?pageSize=50" \
  -H "Authorization: Bearer {ACCESS_TOKEN}"
```

Example Response:

```
{
  "entities": [
    {
      "id": "conversation-111",
      "conversationType": "phone",
      "participants": [
        {
          "id": "user-agent-1",
          "name": "Agent Smith",
          "purpose": "agent",
          "state": "connected"
        },
        {
          "id": "customer-call",
          "name": "John Doe",
          "purpose": "customer",
          "state": "disconnected"
        }
      ],
      "startTime": "2026-03-14T09:15:00Z",
```

```
    "endTime": "2026-03-14T09:25:00Z"
   }
  ],
  "pageNumber": 1,
  "pageSize": 50,
  "total": 500
}
```

Get Conversation Detail

Endpoint: `GET /conversations/{conversationId}`

Description: Get full details including participants, recordings, transcripts.

Query Parameters:

- `expand` (string, optional): Comma-separated fields to expand. Options: `recordings`, `transcript`, `participants`

Example Request:

```
curl -X GET "https://api.mypurecloud.com/api/v2/conversations/conversation-111?expand=recordings,transcript"
\
  -H "Authorization: Bearer {ACCESS_TOKEN}"
```

Example Response:


```
{
  "id": "conversation-111",
  "conversationType": "phone",
  "participants": [
   {
    "id": "user-agent-1",
    "name": "Agent Smith",
    "purpose": "agent"
   },
   {
    "id": "customer-call",
    "name": "John Doe",
    "purpose": "customer"
```

```
    }
  ],
  "recordings": [
    {
      "id": "recording-222",
      "state": "AVAILABLE",
      "durationMilliseconds": 600000,
      "conversationId": "conversation-111",
      "mediaUris": {
        "download": "https://..."
      }
    }
  ],
  "transcript": {
    "id": "transcript-333",
    "displayName": "conversation-111 Transcript",
    "dateCreated": "2026-03-14T09:30:00Z",
    "status": "AVAILABLE"
  }
}
```

Get Recording Media URL

Endpoint: `GET /recordings/{recordingId}/media`

Description: Get download URL for a recording (expires in 1 hour).

Example Request:

```
curl -X GET "https://api.mypurecloud.com/api/v2/recordings/recording-222/media" \
  -H "Authorization: Bearer {ACCESS_TOKEN}" \
  -L
```

Response: 307 redirect to presigned S3 URL

Search Conversations

Endpoint: `POST /conversations/search`

Description: Advanced search using Genesys query syntax.

Request Body:

```
{
  "types": ["phone"],
  "query": "select * where conversations.participants.emails contains 'john.doe@example.com' and startTime >= '2026-03-01'",
  "pageSize": 25,
  "pageNumber": 1
}
```

Example Response:

```
{
  "results": [
    {
      "conversation": {
        "id": "conversation-111",
        "conversationType": "phone",
        "startTime": "2026-03-14T09:15:00Z",
        "endTime": "2026-03-14T09:25:00Z"
      }
    }
  ],
  "pageNumber": 1,
  "pageSize": 25,
  "total": 15
}
```

Users API

Get User Details with Presence

Endpoint: `GET /users/{userId}`

Query Parameters:

- `expand` (string, optional): Fields to expand. Options: `presence`, `locations`, `routingStatus`, `availability`

Example Request:

```
curl -X GET "https://api.mypurecloud.com/api/v2/users/user-123?expand=presence,routingStatus" \
  -H "Authorization: Bearer {ACCESS_TOKEN}"
```

Example Response:


```
{
  "id": "user-123",
  "name": "Agent Smith",
  "email": "agent.smith@company.com",
  "presence": {
    "id": "presence-123",
    "definition": {
      "id": "ON_QUEUE",
      "name": "On Queue"
    },
    "primary": true,
    "source": "USER"
  },
  "routingStatus": {
    "status": "ON_QUEUE",
    "statusMessage": "Available"
  }
}
```

Update User's Presence/Availability

Endpoint: `PATCH /users/{userId}`

Description: Update agent's presence/availability status.

Request Body:


```
{
  "routingStatus": {
    "status": "OFF_QUEUE",
```

```
    "statusMessage": "In Training"
  }
}
```

Example Request:

```
curl -X PATCH "https://api.mypurecloud.com/api/v2/users/user-123" \
  -H "Authorization: Bearer {ACCESS_TOKEN}" \
  -H "Content-Type: application/json" \
  -d '{
    "routingStatus": {
      "status": "OFF_QUEUE",
      "statusMessage": "Break"
    }
  }'
```

Valid Status Values:

- `ON_QUEUE` - Ready to receive interactions
- `OFF_QUEUE` - Not available for interactions
- `IDLE` - No activity
- `ON_A_CALL` - Currently on a call
- `CUSTOM` - Custom status (use statusMessage)

Get All Users in a Queue

Endpoint: `GET /queues/{queueId}/members`

Query Parameters:

- `pageSize` (integer, optional): Members per page (default: 25, max: 100)
- `pageNumber` (integer, optional): Page number (default: 1)
- `sortOrder` (string, optional): `asc` or `desc`

Example Request:

```
curl -X GET "https://api.mypurecloud.com/api/v2/queues/queue-456/members?pageSize=50" \
  -H "Authorization: Bearer {ACCESS_TOKEN}"
```

Example Response:

```
{
  "entities": [
    {
      "id": "user-123",
      "name": "Agent Smith",
      "email": "agent.smith@company.com",
      "state": "ACTIVE"
    },
    {
      "id": "user-124",
      "name": "Agent Jones",
      "email": "agent.jones@company.com",
      "state": "ACTIVE"
    }
  ],
  "pageNumber": 1,
  "pageSize": 50,
  "total": 2
}
```

Search Users

Endpoint: `GET /users/search`

Query Parameters:

- `q` (string): Search query (name, email, phone)
- `pageSize` (integer, optional): Results per page
- `pageNumber` (integer, optional): Page number

Example Request:

```
curl -X GET "https://api.mypurecloud.com/api/v2/users/search?q=smith&pageSize=25" \
  -H "Authorization: Bearer {ACCESS_TOKEN}"
```

Common Query Parameters

Pagination

All list endpoints support:

- `pageSize` (1-100): Items per page
- `pageNumber` (starts at 1): Which page

Example: `?pageSize=50&pageNumber=2` gets items 51-100

Sorting

- `sortBy` (string): Field to sort by (varies by endpoint)
- `sortOrder` (string): `asc` or `desc`

Example: `?sortBy=dateCreated&sortOrder=desc` (newest first)

Expansion

Use `expand` to include related data without extra API calls:

```
GET /conversations/conversation-111?expand=recordings,transcript,participants
```

Rate Limits

All endpoints are subject to Genesys Cloud's rate limiting:

- Standard: 600 requests per minute per organization
- Some endpoints may have different limits
- Check response headers: `X-Rate-Limit-Limit`, `X-Rate-Limit-Remaining`, `X-Rate-Limit-Reset`

Errors

All endpoints return standard HTTP status codes:

Code	Meaning	Action

200	Success	Continue
201	Created	Item created successfully
204	No Content	Success (no response body)
400	Bad Request	Fix request parameters
401	Unauthorized	Token invalid or expired
403	Forbidden	Insufficient permissions
404	Not Found	Resource doesn't exist
429	Rate Limited	Wait before retrying
500	Server Error	Retry after delay

Best Practices

1. **Use Pagination:** Always use `pageSize` and `pageNumber` for list endpoints
2. **Expand Efficiently:** Only expand data you need
3. **Handle Pagination:** Don't assume all data fits in one page
4. **Check Status Codes:** Always handle errors appropriately
5. **Use External IDs:** Link records across systems with `externalId`
6. **Respect Rate Limits:** Monitor `X-Rate-Limit` headers

Related Topics

- Chapter 11: Error Handling & Retry Strategy
- Chapter 11: Rate Limiting & Throttling
- Chapter 11: OAuth Client Management
- Chapter 5: Data Actions (using these APIs in flows)

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
401	Unauthorized	Token expired, invalid credentials	Refresh token or re-authenticate

Code	Meaning	Cause	Action
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

Rate Limiting & Throttling

Overview

Genesys Cloud API has rate limits to ensure fair usage and platform stability. Understanding and respecting these limits is critical for production integrations.

Standard Rate Limit: 600 requests per minute per organization

---

How Rate Limiting Works

Time Windows

Rate limits are enforced in **1-minute rolling windows**:

```
Window 1: 00:00-00:59 → 600 requests allowed
Window 2: 00:01-01:00 → 600 requests allowed
  (overlaps with Window 1)
Window 3: 00:02-01:01 → 600 requests allowed
```

Each minute, the window slides forward. Old requests drop off, new requests are added.

Rate Limit Headers

Every API response includes rate limit information:

```
X-Rate-Limit-Limit:     600      (max requests per window)
X-Rate-Limit-Remaining: 450      (requests left)
X-Rate-Limit-Reset:     1679491234  (Unix timestamp)
```

Example

You make request 150 at timestamp 1679491200

Header: X-Rate-Limit-Remaining: 450

This means: 150 requests made, 450 more allowed before limit.

Detecting Rate Limit Conditions

Proactive Detection (Before hitting limit)

Monitor remaining requests:

```
async function makeRequest(endpoint) {
  const response = await fetch(endpoint);
  const remaining = parseInt(
    response.headers.get('X-Rate-Limit-Remaining')
  );
  const limit = parseInt(response.headers.get('X-Rate-Limit-Limit'));

  const percentRemaining = (remaining / limit) * 100;

  if (percentRemaining < 20) {
    console.warn('⚠ Only 20% of rate limit remaining. Slowing down...');
    // Reduce request frequency
  }

  if (percentRemaining < 5) {
    console.error('□□ Critical: <5% remaining. STOP requests immediately.');
    // Pause all requests
  }

  return response;
}
```

Reactive Detection (After hitting limit)

Watch for 429 responses:

```
async function makeRequest(endpoint) {
  const response = await fetch(endpoint);

  if (response.status === 429) {
    console.error('⛔ Rate limit exceeded!');

    const retryAfter = response.headers.get('Retry-After');
    const resetTime = response.headers.get('X-Rate-Limit-Reset');

    if (retryAfter) {
      // API tells you when to retry
      const seconds = parseInt(retryAfter);
      console.warn(`Wait ${seconds} seconds`);
    } else if (resetTime) {
      // Calculate wait time from reset timestamp
      const now = Math.floor(Date.now() / 1000);
      const waitSeconds = parseInt(resetTime) - now;
      console.warn(`Wait until ${new Date(resetTime * 1000).toISOString()}`);
    }
  }

  return response;
}
```

Strategy 1: Exponential Backoff (Reactive)

Only for when you hit the limit:

```
async function requestWithBackoff(endpoint, maxAttempts = 4) {
  const delays = [3000, 9000, 27000, 300000]; // 3s, 9s, 27s, 5min
```

```
  for (let attempt = 0; attempt < maxAttempts; attempt++) {
    const response = await fetch(endpoint);

    if (response.status !== 429) {
      return response; // Success (or other error)
    }

    // Rate limited
    if (attempt >= maxAttempts - 1) {
      throw new Error('Rate limit exceeded after max retries');
    }

    const delayMs = delays[attempt];
    console.warn(`Rate limited. Waiting ${delayMs / 1000}s...`);
    await sleep(delayMs);
  }
}
```

Strategy 2: Request Throttling (Proactive)

Limit request rate to stay well below limit:

```
class ThrottledClient {
  constructor(requestsPerSecond = 8) {
    // 8 requests/sec = 480/min (80% of 600 limit)
    this.requestsPerSecond = requestsPerSecond;
    this.minIntervalMs = 1000 / requestsPerSecond;
    this.lastRequestTime = 0;
  }

  async makeRequest(endpoint) {
    const now = Date.now();
    const timeSinceLastRequest = now - this.lastRequestTime;

    if (timeSinceLastRequest < this.minIntervalMs) {
```

```
      const waitMs = this.minIntervalMs - timeSinceLastRequest;
      await sleep(waitMs);
    }

    this.lastRequestTime = Date.now();
    return fetch(endpoint);
  }
}

// Usage
const client = new ThrottledClient(8); // 8 req/sec (safe limit)
await client.makeRequest('/contacts');
await client.makeRequest('/contacts');
// These will be spaced 125ms apart
```

Strategy 3: Request Queue with Batch Processing

Buffer requests and process in batches:

```
class RequestQueue {
  constructor(batchSize = 50, batchIntervalMs = 5000) {
    this.queue = [];
    this.batchSize = batchSize;
    this.batchIntervalMs = batchIntervalMs;
    this.processing = false;
  }

  async add(endpoint, body) {
    return new Promise((resolve, reject) => {
      this.queue.push({ endpoint, body, resolve, reject });

      if (!this.processing) {
        this.processBatch();
      }
    });
```

```
  }

  async processBatch() {
    this.processing = true;

    while (this.queue.length > 0) {
      const batch = this.queue.splice(0, this.batchSize);

      // Process batch in parallel (but within rate limit)
      const promises = batch.map(item =>
        fetch(item.endpoint, { body: JSON.stringify(item.body) })
          .then(res => item.resolve(res))
          .catch(err => item.reject(err))
      );

      await Promise.all(promises);

      // Wait between batches
      if (this.queue.length > 0) {
        console.log(`Processed ${batch.length} requests. Waiting ${this.batchIntervalMs}ms...`);
        await sleep(this.batchIntervalMs);
      }
    }

    this.processing = false;
  }
}

// Usage
const queue = new RequestQueue(50, 5000); // 50 requests per 5 seconds

for (const contact of millionContacts) {
  queue.add('/contacts', contact);
}
```

Strategy 4: Bulk Operations

Most efficient: use bulk endpoints instead of individual requests.

Without Bulk (SLOW - 100 requests)

```
// Creating 100 contacts individually
for (const contact of contacts) {
  await fetch('/contacts', {
    method: 'POST',
    body: JSON.stringify(contact)
  });
}
// Uses 100 API calls!
```

With Bulk (FAST - 1 request)

```
// Creating 100 contacts in one batch
await fetch('/contacts/bulk', {
  method: 'POST',
  body: JSON.stringify({
    contacts: contacts  // Array of 100
  })
});
// Uses 1 API call!
```

Benefit: 100x reduction in API calls.

Strategy 5: Caching

Avoid repeated requests for same data:

```
class CachedClient {
  constructor(cacheTtlSeconds = 3600) {
    this.cache = new Map();
    this.cacheTtlSeconds = cacheTtlSeconds;
  }
```

```
  async getContact(contactId) {
    const cacheKey = `contact:${contactId}`;

    // Check cache
    const cached = this.cache.get(cacheKey);
    if (cached && Date.now() - cached.timestamp < this.cacheTtlSeconds * 1000) {
      console.log(`Cache hit: ${cacheKey}`);
      return cached.data;
    }

    // Not cached, fetch from API
    console.log(`Cache miss: ${cacheKey}`);
    const response = await fetch(`/contacts/${contactId}`);
    const data = await response.json();

    // Store in cache
    this.cache.set(cacheKey, { data, timestamp: Date.now() });

    return data;
  }

  clearCache() {
    this.cache.clear();
  }
}

// Usage
const client = new CachedClient(3600); // Cache for 1 hour
const contact1 = await client.getContact('c1'); // API call
const contact1Again = await client.getContact('c1'); // Cache hit!
```

Strategy 6: WebSocket Subscriptions (Real-time, Low Overhead)

Instead of polling, use WebSocket for real-time updates:

Polling (Uses many requests)

```
// Poll every 10 seconds = 6 requests/min per user
setInterval(async () => {
  const status = await fetch('/users/user-123?expand=presence');
  // Expensive for many users!
}, 10000);
```

WebSocket (1 connection)

```
// Single WebSocket connection for real-time updates
const ws = new WebSocket('wss://...');

ws.onmessage = (event) => {
  const { userId, presence } = JSON.parse(event.data);
  console.log(`User ${userId} is now ${presence}`);
};

// Can handle thousands of users on one connection!
```

Benefit: Eliminates polling entirely.

Rate Limit Calculation

Scenario 1: Simple API calls

```
10,000 contacts to sync
1 API call per contact = 10,000 calls
Rate limit: 600/minute
Time needed: 10,000 / 600 = 16.67 minutes

Strategy: Use bulk endpoint (1 call) or batch in 600-request chunks
```

Scenario 2: Contact lookup every second

100 concurrent agents

Each looks up contact every second

= 100 requests/second = 6,000/minute

Rate limit: 600/minute

You'd exceed limit 10x over!

Strategy: Add 100ms delay between requests

= 10 requests/sec = 600/minute (perfect!)

Scenario 3: Mixed workload

- Sync contacts: 1000 requests (use bulk)
- Agent presence updates: 100 requests/minute (natural rate)
- Search queries: variable (depends on usage)
- Reporting: 50 requests/day (batch at night)

Total: 1000 + 100 + variable + ~2 = Safe if variable < 500/min

---

Monitoring & Alerting

Track rate limit over time

```
class RateLimitMonitor {
  constructor() {
    this.history = [];
  }

  record(remaining, limit) {
    const now = new Date();
    const percentUsed = ((limit - remaining) / limit) * 100;

    this.history.push({ now, remaining, percentUsed });
```

```
    // Alert if trend is concerning
    if (this.history.length > 10) {
      const recentAverage = this.history
        .slice(-10)
        .reduce((sum, h) => sum + h.percentUsed, 0) / 10;

      if (recentAverage > 80) {
        console.warn('⚠ Average usage >80%. Trending toward limit!');
      }
    }
  }

  report() {
    const avg = this.history.reduce((sum, h) => sum + h.percentUsed, 0) / this.history.length;
    const max = Math.max(...this.history.map(h => h.percentUsed));

    console.log(`
      Rate Limit Usage Report:
      Average: ${avg.toFixed(1)}%
      Peak: ${max.toFixed(1)}%
      Samples: ${this.history.length}
    `);
  }
}
```

Recommended Approach: Tiered Strategy

Tier 1 - Proactive (Always do this)

- Monitor `X-Rate-Limit-Remaining` on every request
- Implement throttling (8 req/sec = 480/min, safe)
- Batch operations when possible

Tier 2 - Reactive (If approaching limit)

- Reduce request frequency further
- Implement caching
- Queue requests instead of fire-and-forget

Tier 3 - Emergency (If hitting limit)

- Implement exponential backoff
- Stop new requests
- Alert operations team

Best Practices

1. **Monitor proactively** - Don't wait for 429 errors
2. **Use bulk endpoints** - Single call for multiple records
3. **Implement throttling** - Spread requests evenly
4. **Cache aggressively** - Don't re-fetch same data
5. **Use WebSockets** - For real-time subscriptions
6. **Batch requests** - Process in groups, not individually
7. **Set timeouts** - Don't retry forever
8. **Alert operations** - Know when limit is approached

Common Mistakes

 Fire-and-forget requests - No rate limit awareness
 Monitor headers, throttle proactively

 Polling instead of WebSocket - Wastes requests
 Use WebSocket for real-time data

 Individual API calls in loop - 1000 calls instead of 1
 Use bulk endpoints, batch in groups

 Ignore rate limit warnings - Hit limit unexpectedly
 Monitor, reduce frequency before limit

 Unlimited retry - Keep hammering API
 Exponential backoff, respect Retry-After

Related Topics

- Chapter 11: Error Handling & Retry Strategy
- Chapter 11: API Endpoints Reference
- Chapter 5: Data Actions (rate limiting in flows)