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)

Question	Answer
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)
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

Revision #1
Created 14 March 2026 19:31:22 by Cesar Gzz
Updated 14 March 2026 19:31:39 by Cesar Gzz