Contents

OAuth 2.0 Developer Guide 2
- Table of Contents 2
- Overview 2
  - Supported Flows 2
- Registering Your Application 3
  - For Developer/Testing 3
  - For Production/Platform-Wide 3
- Authorization Code Flow 3
  - Step 1: Redirect to Authorization 3
  - Step 2: User Grants Permission 4
  - Step 3: Receive Authorization Code 4
  - Step 4: Exchange Code for Tokens 5
- Token Management 5
  - Access Tokens 5
  - Refresh Tokens 6
  - Token Revocation 6
- Scopes and Permissions 7
  - Available Scopes 7
  - Scope Combinations 7
- API Requests 7
  - Base URL 7
  - Authentication 7
  - Example Requests 8
- Consent Screens 8
  - What Users See 8
  - Improving Consent Experience 9
- Error Handling 9
  - Authorization Errors 9
  - Token Errors 9
  - API Errors 10
- Security Best Practices 10
  - Must Do 10
  - Token Storage 10
  - Redirect URI Security 10
- Testing 11
  - Development Redirect URIs 11
  - Test Users 11
  - Debugging Tips 11
  - Token Inspection Endpoint 11
- Code Examples 11
  - Complete Node.js Example 11
- Related Documentation 13

OAuth 2.0 Developer Guide

Version: 5.52.29 | **Last Updated**: January 25, 2026 | **Audience**: Developers

This guide covers building third-party applications that integrate with RADIANT using OAuth 2.0, including authorization flows, scopes, token management, and best practices.

Table of Contents

1. Overview
2. Registering Your Application
3. Authorization Code Flow
4. Token Management
5. Scopes and Permissions
6. API Requests
7. Consent Screens
8. Error Handling
9. Security Best Practices
10. Testing

Overview

RADIANT implements OAuth 2.0 to allow third-party applications to access user data with their consent.

```
sequenceDiagram
    participant User
    participant YourApp as Your App
    participant RADIANT

    User->>YourApp: Click "Connect to RADIANT"
    YourApp->>RADIANT: Redirect to authorization
    RADIANT->>User: Show consent screen
    User->>RADIANT: Grant permission
    RADIANT->>YourApp: Authorization code
    YourApp->>RADIANT: Exchange code for tokens
    RADIANT->>YourApp: Access + refresh tokens
    YourApp->>RADIANT: API requests with access token
    RADIANT->>YourApp: Protected resources
```

Supported Flows

Flow	Use Case	Client Type
Authorization Code + PKCE	Web apps, mobile apps, SPAs	Public & Confidential

Flow	Use Case	Client Type
Client Credentials	Server-to-server (M2M)	Confidential only

> **Note**: Implicit flow is not supported due to security concerns.

Registering Your Application

For Developer/Testing

1. Sign in to your RADIANT account (must be a tenant admin)
2. Navigate to **Admin** → **Integrations** → **OAuth Applications**
3. Click **Create Application**
4. Enter application details:

Field	Description	Example
Name	Displayed to users	"My Awesome App"
Description	What your app does	"Sync your sessions with…"
Website URL	Your app's homepage	`https://myapp.com`
Redirect URIs	Where to return after auth	`https://myapp.com/callback`
Logo	256x256 PNG/SVG	Upload file

5. Click **Create**
6. **Save your credentials**:
   - **Client ID**: Public identifier
   - **Client Secret**: Keep this secret! (shown only once)

For Production/Platform-Wide

Contact RADIANT to register a verified application that can be used across all tenants.

Authorization Code Flow

The recommended flow for most applications.

Step 1: Redirect to Authorization

Redirect the user to RADIANT's authorization endpoint:

```
GET https://{radiant-domain}/oauth/authorize
```

Query Parameters:

Parameter	Required	Description
`client_id`	Yes	Your application's client ID

Parameter	Required	Description
`redirect_uri`	Yes	Must exactly match a registered URI
`response_type`	Yes	Always `code`
`scope`	Yes	Space-separated list of scopes
`state`	Yes	Random string for CSRF protection
`code_challenge`	Yes*	PKCE challenge (required for public clients)
`code_challenge_method`	Yes*	Always `S256`

Example:

```
// Generate PKCE verifier and challenge
const codeVerifier = generateRandomString(64);
const codeChallenge = base64URLEncode(sha256(codeVerifier));

// Build authorization URL
const authUrl = new URL('https://app.radiant.ai/oauth/authorize');
authUrl.searchParams.set('client_id', 'your-client-id');
authUrl.searchParams.set('redirect_uri', 'https://myapp.com/callback');
authUrl.searchParams.set('response_type', 'code');
authUrl.searchParams.set('scope', 'openid profile read:sessions');
authUrl.searchParams.set('state', generateRandomString(32));
authUrl.searchParams.set('code_challenge', codeChallenge);
authUrl.searchParams.set('code_challenge_method', 'S256');

// Store verifier and state for later
sessionStorage.setItem('pkce_verifier', codeVerifier);
sessionStorage.setItem('oauth_state', state);

// Redirect user
window.location.href = authUrl.toString();
```

Step 2: User Grants Permission

RADIANT displays a consent screen showing: - Your application's name and logo - Requested permissions (scopes) - The user's identity

The user can **Allow** or **Deny** the request.

Step 3: Receive Authorization Code

After the user grants permission, RADIANT redirects back to your `redirect_uri`:

```
https://myapp.com/callback?code=AUTH_CODE&state=YOUR_STATE
```

Verify the state parameter matches what you sent!

Step 4: Exchange Code for Tokens

```
POST https://{radiant-domain}/oauth/token
Content-Type: application/x-www-form-urlencoded
```

Body Parameters:

Parameter	Required	Description
`grant_type`	Yes	`authorization_code`
`code`	Yes	The authorization code received
`redirect_uri`	Yes	Same URI used in authorization
`client_id`	Yes	Your client ID
`client_secret`	Conditional	Required for confidential clients
`code_verifier`	Conditional	Required if PKCE was used

Example (Node.js):

```javascript
const response = await fetch('https://app.radiant.ai/oauth/token', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/x-www-form-urlencoded',
  },
  body: new URLSearchParams({
    grant_type: 'authorization_code',
    code: authorizationCode,
    redirect_uri: 'https://myapp.com/callback',
    client_id: 'your-client-id',
    code_verifier: storedCodeVerifier, // From Step 1
  }),
});

const tokens = await response.json();
```

Response:

```json
{
  "access_token": "eyJhbGciOiJSUzI1NiIs...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "dGhpcyBpcyBhIHJlZnJlc2g...",
  "scope": "openid profile read:sessions",
  "id_token": "eyJhbGciOiJSUzI1NiIs..."
}
```

Token Management

Access Tokens

Property	Value
Type	JWT
Lifetime	1 hour (3600 seconds)
Use	Authorization header for API requests

Using the access token:

```
const response = await fetch('https://api.radiant.ai/v1/sessions', {
  headers: {
    'Authorization': `Bearer ${accessToken}`,
  },
});
```

Refresh Tokens

Property	Value
Type	Opaque string
Lifetime	30 days (configurable)
Use	Obtain new access tokens

Refreshing tokens:

```
POST https://{radiant-domain}/oauth/token
Content-Type: application/x-www-form-urlencoded

grant_type=refresh_token
&refresh_token=YOUR_REFRESH_TOKEN
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET
```

Response:

```
{
  "access_token": "new_access_token...",
  "token_type": "Bearer",
  "expires_in": 3600,
  "refresh_token": "new_or_same_refresh_token..."
}
```

Token Revocation

Revoke tokens when users disconnect your app:

```
POST https://{radiant-domain}/oauth/revoke
Content-Type: application/x-www-form-urlencoded

token=TOKEN_TO_REVOKE
&token_type_hint=refresh_token
```

```
&client_id=YOUR_CLIENT_ID
&client_secret=YOUR_CLIENT_SECRET
```

Scopes and Permissions

Request only the scopes your application needs.

Available Scopes

Scope	Access	Description
`openid`	Identity	Required for OIDC, returns ID token
`profile`	Identity	User's name and avatar
`email`	Identity	User's email address
`read:sessions`	Sessions	List and view Think Tank sessions
`write:sessions`	Sessions	Create and modify sessions
`read:files`	Files	Access uploaded files
`write:files`	Files	Upload and delete files
`read:artifacts`	Artifacts	View generated artifacts
`write:artifacts`	Artifacts	Create and modify artifacts
`offline_access`	Tokens	Receive refresh tokens

Scope Combinations

Minimal (identity only):

```
scope=openid profile email
```

Read-only access:

```
scope=openid profile read:sessions read:files read:artifacts
```

Full access:

```
scope=openid profile email read:sessions write:sessions read:files write:files read:artifacts
```

API Requests

Base URL

```
https://api.radiant.ai/v1
```

Authentication

Include the access token in the `Authorization` header:

```
GET /v1/sessions HTTP/1.1
Host: api.radiant.ai
Authorization: Bearer eyJhbGciOiJSUzI1NiIs...
```

Example Requests

Get current user:

```
const user = await fetch('https://api.radiant.ai/v1/me', {
  headers: { 'Authorization': `Bearer ${accessToken}` },
}).then(r => r.json());

// Response:
// {
//   "id": "user_abc123",
//   "email": "user@example.com",
//   "name": "Jane Doe",
//   "avatar_url": "https://..."
// }
```

List sessions:

```
const sessions = await fetch('https://api.radiant.ai/v1/sessions', {
  headers: { 'Authorization': `Bearer ${accessToken}` },
}).then(r => r.json());

// Response:
// {
//   "data": [
//     { "id": "sess_123", "title": "Project Planning", "created_at": "..." },
//     ...
//   ],
//   "has_more": true,
//   "next_cursor": "..."
// }
```

See the API Reference for complete endpoint documentation.

Consent Screens

What Users See

The consent screen displays:

```
            [Your App Logo]

      "My Awesome App" wants to
      access your RADIANT account

      This will allow the app to:
        See your profile information
        View your Think Tank sessions
      Create new sessions

     [Deny]              [Allow]

     Signed in as: user@example.com
```

Improving Consent Experience

Do	Don’t
Request minimal scopes	Request all scopes “just in case”
Use a clear app name	Use technical/internal names
Provide a recognizable logo	Use a generic placeholder
Explain why you need access	Leave users guessing

Error Handling

Authorization Errors

Errors during authorization redirect to your `redirect_uri` with error parameters:

```
https://myapp.com/callback?error=access_denied&error_description=User%20denied%20access&state=
```

Error	Description
`invalid_request`	Missing or invalid parameters
`unauthorized_client`	Client not allowed to use this flow
`access_denied`	User denied permission
`unsupported_response_type`	Invalid `response_type`
`invalid_scope`	Unknown or invalid scopes
`server_error`	Internal error

Token Errors

Token endpoint returns JSON errors:

```json
{
  "error": "invalid_grant",
  "error_description": "Authorization code has expired"
}
```

Error	Description
`invalid_request`	Missing required parameter
`invalid_client`	Client authentication failed
`invalid_grant`	Code expired, already used, or invalid

Error	Description
`unauthorized_client`	Client not authorized for this grant
`unsupported_grant_type`	Invalid grant type

API Errors

API requests return standard HTTP errors:

Status	Meaning	Action
`401`	Token invalid or expired	Refresh the token
`403`	Insufficient scope	Request additional scopes
`429`	Rate limited	Back off and retry

Security Best Practices

Must Do

Practice	Why
Always use HTTPS	Protect tokens in transit
Always use PKCE	Prevent code interception attacks
Validate state parameter	Prevent CSRF attacks
Store secrets securely	Never expose client secret in frontend code
Use short-lived tokens	Limit damage from token theft

Token Storage

Client Type	Recommended Storage
Web (SPA)	Memory (not localStorage) or secure httpOnly cookie
Web (Server)	Server-side session or encrypted database
Mobile	Secure keychain (iOS) / Keystore (Android)
Desktop	OS credential storage

Redirect URI Security

- Use exact matching (no wildcards in production)
- Always use HTTPS (except `http://localhost` for development)
- Avoid open redirectors

Testing

Development Redirect URIs

You can register `http://localhost:*` for development:

```
http://localhost:3000/callback
http://localhost:8080/auth/callback
```

Test Users

Create test users in your tenant for development:

1. Navigate to **Admin** → **Users** → **Invite User**
2. Invite yourself with a `+test` email alias (e.g., `you+test@company.com`)
3. Use this account for OAuth testing

Debugging Tips

1. **Inspect tokens**: Use jwt.io to decode and inspect JWTs
2. **Check scopes**: Verify the token contains expected scopes in the `scope` claim
3. **Verify signatures**: Ensure tokens are signed correctly
4. **Monitor logs**: Check your app logs for OAuth errors

Token Inspection Endpoint

```
GET https://{radiant-domain}/oauth/tokeninfo?token=ACCESS_TOKEN
```

Response:

```json
{
  "active": true,
  "client_id": "your-client-id",
  "scope": "openid profile read:sessions",
  "sub": "user_abc123",
  "exp": 1706234567,
  "iat": 1706230967
}
```

Code Examples

Complete Node.js Example

```javascript
const express = require('express');
const crypto = require('crypto');

const app = express();

const CLIENT_ID = 'your-client-id';
const CLIENT_SECRET = 'your-client-secret';
const REDIRECT_URI = 'http://localhost:3000/callback';
const RADIANT_DOMAIN = 'https://app.radiant.ai';

// Generate PKCE challenge
function generatePKCE() {
  const verifier = crypto.randomBytes(32).toString('base64url');
  const challenge = crypto
    .createHash('sha256')
    .update(verifier)
    .digest('base64url');
  return { verifier, challenge };
}

// Step 1: Start OAuth flow
app.get('/login', (req, res) => {
  const { verifier, challenge } = generatePKCE();
  const state = crypto.randomBytes(16).toString('hex');

  // Store for later verification
  req.session.pkceVerifier = verifier;
  req.session.oauthState = state;

  const authUrl = new URL(`${RADIANT_DOMAIN}/oauth/authorize`);
  authUrl.searchParams.set('client_id', CLIENT_ID);
  authUrl.searchParams.set('redirect_uri', REDIRECT_URI);
  authUrl.searchParams.set('response_type', 'code');
  authUrl.searchParams.set('scope', 'openid profile read:sessions');
  authUrl.searchParams.set('state', state);
  authUrl.searchParams.set('code_challenge', challenge);
  authUrl.searchParams.set('code_challenge_method', 'S256');

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

// Step 2-4: Handle callback
app.get('/callback', async (req, res) => {
  const { code, state, error } = req.query;

  // Check for errors
  if (error) {
    return res.status(400).send(`OAuth error: ${error}`);
  }

  // Verify state
  if (state !== req.session.oauthState) {
    return res.status(400).send('State mismatch');
  }

  // Exchange code for tokens
  const tokenResponse = await fetch(`${RADIANT_DOMAIN}/oauth/token`, {
    method: 'POST',
    headers: { 'Content-Type': 'application/x-www-form-urlencoded' },
    body: new URLSearchParams({
      grant_type: 'authorization_code',
      code,
      redirect_uri: REDIRECT_URI,
      client_id: CLIENT_ID,
      client_secret: CLIENT_SECRET,
      code_verifier: req.session.pkceVerifier,
    }),
  });

  const tokens = await tokenResponse.json();

  if (tokens.error) {
    return res.status(400).send(`Token error: ${tokens.error}`);
  }

  // Store tokens securely
  req.session.accessToken = tokens.access_token;
  req.session.refreshToken = tokens.refresh_token;

  res.redirect('/dashboard');
});

app.listen(3000);
```

Related Documentation

- Authentication Overview
- API Reference
- Tenant Admin Guide
- Security Architecture