OAuth Custom Claims Implementation Guide

Executive Summary

Consideration for implementing custom claims in JupyterHealth Exchange's OAuth ID tokens to include user permissions (studies, organizations, roles) directly in the authentication token. This enables a **Direct MCP Server architecture** that simplifies deployment and improves performance compared to the Bridge Web App pattern.

Decision:

- Option A (Bridge Web App with /userinfo API) or
- Option B (Direct MCP with custom ID token claims)

Architecture Comparison

Option A: Bridge Web App Pattern

```
┌──────────────┐
│ User / LLM   │
│ Client       │
└──────┬───────┘
       │ 1. Query Request
       ▼
┌───────────────────────────────────────────────────────┐
│ Bridge Web App (FastAPI)                              │
│                                                       │
│  A. User Authentication (First Time)                  │
│     ↓                                                 │
│  ┌────────────────────────────────────────────────┐   │
│  │ OAuth Flow                                     │   │
│  │ → JHE Authorization Server                     │   │
│  │ ← Access Token (opaque)                        │   │
│  └────────────────────────────────────────────────┘   │
│     ↓                                                 │
│  B. Permission Lookup (Per Query)                     │
│     ↓                                                 │
│  ┌────────────────────────────────────────────────┐   │
│  │ API Call: /userinfo                            │   │
│  │ → Send access token                            │   │
│  │ ← Get organization IDs, roles                  │   │
│  └────────────────────────────────────────────────┘   │
│     ↓                                                 │
│  C. Forward Query with Permission Filter              │
└───────┬───────────────────────────────────────────────┘
        │ 2. Query + org filter
        ▼
┌────────────────────────────────────┐
│ MCP-Postgres Server                │
│  - Appends WHERE clauses           │
│  - Filters by organization IDs    │
└───────┬───────────────────────────┘
        │ 3. SQL Query
        ▼

┌───────────────────┐
│ PostgreSQL        │
│ Database          │
└───────┬───────────┘
        │ 4. All matching rows
        ▼

┌───────────────────────────────────┐
│ MCP-Postgres Server               │
│  ← Returns data                   │
└───────┬───────────────────────────┘
        │ 5. Data
        ▼

┌───────────────────────────────────────────────────────┐
│ Bridge Web App                                        │
│  ← May apply additional filtering                     │
└───────┬───────────────────────────────────────────────┘
        │ 6. Filtered data
        ▼

┌──────────────┐
│ User / LLM   │
│ Client       │
└──────────────┘
```

Components:

- Bridge Web App
- MCP-Postgres Server
- PostgreSQL Database
- JHE OAuth Server

Data Flow:

1. Request: User → Bridge App → JHE OAuth (get access token)
2. Per-query: Bridge App → /userinfo API (fetch permissions)
3. Request: Bridge App → MCP Server → PostgreSQL (filtered query)
4. Response: PostgreSQL → MCP Server → Bridge App → User
5. **Filtering location**: MCP Server adds WHERE clauses based on permissions from /userinfo

---

Option B: Direct MCP Server Pattern

```
┌──────────────┐
│ User / LLM   │
│ Client       │
└──────┬───────┘
       │ 1. Query Request
       ▼
```

```
┌──────────────────────────────────────────────────────────────┐
│ JHE Universal MCP Server                                     │
│                                                              │
│  A. First Use: OAuth Flow                                    │
│     ↓                                                        │
│  ┌────────────────────────────────────────────────────────┐  │
│  │ OAuth Flow (opens browser)                             │  │
│  │ → JHE Authorization Server                             │  │
│  │ ← Access Token (opaque)                                │  │
│  │ ← ID Token (JWT with custom claims) ◄── NEW: Claims!   │  │
│  └────────────────────────────────────────────────────────┘  │
│     ↓                                                        │
│  B. Extract Permissions from ID Token (Cached)               │
│     ↓                                                        │
│  ┌────────────────────────────────────────────────────────┐  │
│  │ Decode ID Token (local, no API call)                   │  │
│  │ {                                                      │  │
│  │   "sub": "20001",                                      │  │
│  │   "jhe_permissions": {                                 │  │
│  │     "studies": [30001, 30002],                         │  │
│  │     "organizations": [...]                             │  │
│  │   }                                                    │  │
│  │ }                                                      │  │
│  └────────────────────────────────────────────────────────┘  │
│     ↓                                                        │
│  C. Build Permission-Filtered SQL Query                      │
│     ↓                                                        │
│  ┌────────────────────────────────────────────────────────┐  │
│  │ Appends WHERE clauses with user-permissioned data only │  │
│  │ (e.g., WHERE study_id IN (30001, 30002))               │  │
│  └────────────────────────────────────────────────────────┘  │
└──────────────┬───────────────────────────────────────────────┘
               │ 2. SQL Query (pre-filtered)
               ▼
      ┌────────────────────┐
      │ PostgreSQL         │
      │ Database           │
      └────────┬───────────┘
               │ 3. Only authorized rows
               ▼
┌──────────────────────────────────────────────────────────────┐
│ JHE Universal MCP Server                                     │
│  ← Returns data (already filtered at SQL level)              │
└──────────────┬───────────────────────────────────────────────┘
               │ 4. Filtered data
               ▼
      ┌──────────────┐
      │ User / LLM   │
      │ Client       │
      └──────────────┘
```

Components:

- JHE Universal MCP Server (single component)
- PostgreSQL Database
- JHE OAuth Server (modified to include custom claims)

Data Flow:

1. First-time: User → MCP Server → JHE OAuth (get access token + ID token with claims)
2. Cached: MCP Server decodes ID token locally (no API call needed)
3. Request: MCP Server → PostgreSQL (pre-filtered SQL query)
4. Response: PostgreSQL (only authorized rows) → MCP Server → User
5. **Filtering location**: MCP Server adds WHERE clauses based on claims from ID token (SQL-level filtering)

Side-by-Side Comparison

Aspect	Option A: Bridge App	Option B: Direct MCP
Architecture	3 components (Bridge + MCP + DB)	2 components (MCP + DB)
Deployment	Deploy and monitor 2 services	Deploy and monitor 1 service
User Setup	Configure Bridge App + MCP	Configure MCP only
Permission Lookup	API call to /userinfo	Read from ID token (local)
API Calls per Query	2+ (/userinfo + query)	1 (query only)
Permission Freshness	Always current (real-time)	Stale until token expires (1 hour)
Revocation Speed	Immediate	Up to 1 hour delay
Performance	Slower (network calls)	Faster (local token read)
Offline Support	No (requires /userinfo API)	Yes (permissions in token)
Complexity	Higher (2 apps to maintain)	Lower (1 app to maintain)
MCP Pattern	Non-standard (Bridge pattern)	Standard (Direct MCP pattern)
JHE Changes Required	UserInfo endpoint	Custom Validator
Security: Metadata Disclosure	None (permissions server-side)	Study IDs visible in ID token
Token Size	Small (~200 bytes)	Medium (~1-2KB)

Recommendations: Option B (Direct MCP) - Simpler architecture, better performance, follows standard MCP patterns

---

Background

Current Implementation

- Django OAuth Toolkit with OIDC enabled
- ID tokens contain only standard OIDC claims (sub, email, iat, exp)
- MCP server makes separate API calls to fetch user's studies and organizations
- Access token is opaque (secure random string)

Why Add Custom Claims?

Benefits:

- ✅ **Performance**: Eliminates 2+ API calls per MCP query (organizations, studies)
- ✅ **Simplicity**: MCP server gets all permissions in one token
- ✅ **Offline capability**: Permissions available without API access
- ✅ **Standard pattern**: Many OAuth providers include custom claims (Auth0, Okta, etc.)

Security Consideration:

- ID tokens are JWTs (signed but not encrypted) - anyone can decode and read them
- This means study IDs, organization names, and roles are **readable** if token is intercepted
- **However**: This is metadata disclosure, not a security breach
- Access token (opaque) is still required to actually query patient data
- Risk assessment: **Acceptable** - study IDs are not considered sensitive information

Decision: Custom OAuth2Validator

We are using Django OAuth Toolkit's built-in extension point for adding custom claims.

Trade-offs

Aspect	Current (API-based)	Method 1 (Claims in Token)
Permission freshness	Always current	Stale until token expires
API calls needed	2+ per query	0
Token size	Small (~200 bytes)	Medium (~1-2KB)
Performance	Slower (API calls)	Faster (local lookup)
Revocation speed	Immediate	Up to token expiry
Metadata disclosure	None	Study IDs, org names, roles visible

Mitigation for stale permissions: Reduce token expiry from 2 weeks to 1 hour (see Configuration section)

MCP Server Handles Authentication & Authorization

The MCP server reads claims from ID token instead of making `/userinfo` API calls. This is one of the core benefit of adding custom claims - eliminating API calls for permission lookups.

```python
# In jhe-universal-mcp/src/auth/auth_context.py

def __init__(self, token: str, id_token: str):
    """
    Initialize auth context from OAuth tokens.

    Args:
        token: Access token (opaque, used for API authentication)
        id_token: ID token (JWT with custom claims containing permissions)
    """
    self.token = token

    # Extract permissions from ID token claims
    import jwt
    claims = jwt.decode(id_token, options={"verify_signature": False})

    self.user_id = claims.get('user_id')
    self.user_type = claims.get('user_type')
    self.is_superuser = (self.user_type == 'superuser')

    # Get permissions from custom claims
    permissions = claims.get('jhe_permissions', {})
    self._accessible_studies = set(permissions.get('studies', []))
    self._organizations = permissions.get('organizations', [])

    # Build role set for permission checks
    self._roles = {org['role'] for org in self._organizations}
```

Key changes:

- ✅ Removed `/userinfo` API calls entirely
- ✅ Read all permissions from ID token claims
- ✅ Simplified initialization - one source of truth (ID token)

Configuration

Token Expiry Recommendation

Before: 2 weeks (1209600 seconds) **After:** 1 hour (3600 seconds)

Rationale:

- Shorter expiry = fresher permissions
- Users re-authenticate every hour (transparent with refresh tokens)
- Reduced window for stale permissions

Alternative: Keep 2 weeks but implement token refresh with claim updates

Implementation

Step 1: Create Custom OAuth2 Validator

Create file: `jupyterhealth-exchange/core/oauth_validators.py`

```
"""
Custom OAuth2 Validator for JupyterHealth Exchange
Adds user permissions to OIDC ID tokens
"""
from oauth2_provider.oauth2_validators import OAuth2Validator
from core.models import Study, PractitionerOrganization


class JHEOAuth2Validator(OAuth2Validator):
    """
    Custom validator that adds JHE-specific claims to ID tokens
    """

    def get_additional_claims(self, request):
        """
        Add custom claims to the ID token.

        This method is called during token generation and adds:
        - user_type: "patient" or "practitioner"
        - user_id: JheUser primary key
        - jhe_permissions: Object containing accessible studies and
organizations

        Args:
            request: OAuthlib request object with authenticated user

        Returns:
            dict: Custom claims to add to ID token
        """
        user = request.user

        # Don't add claims for patient users (they don't have org/study
access)
        if user.user_type == 'patient':
            return {
                'user_type': 'patient',
                'user_id': user.id,
            }

        # For practitioners, fetch accessible studies and organizations
        try:
            # Get all studies accessible via practitioner's organizations
            # Uses the related_name 'practitioner_links' from
PractitionerOrganization
            accessible_studies = list(
                Study.objects.filter(

organization__practitioner_links__practitioner__jhe_user=user
                ).values_list('id', flat=True).distinct()
            )
            # Get practitioner's organizations with roles
            practitioner_orgs = list(
                PractitionerOrganization.objects.filter(
                    practitioner__jhe_user=user
                ).select_related('organization').values(
                    'organization_id',
                    'organization__name',
                    'role'
                )
            )

            organizations = [
                {
                    'id': org['organization_id'],
                    'name': org['organization__name'],
                    'role': org['role']
                }
                for org in practitioner_orgs
            ]

        except Exception as e:
            # Log error but don't fail token generation
            import logging
            logger = logging.getLogger(__name__)
            logger.error(f"Error fetching permissions for user {user.id}: {e}")
            accessible_studies = []
            organizations = []

        return {
            'user_type': user.user_type,
            'user_id': user.id,
            'jhe_permissions': {
                'studies': accessible_studies,
                'organizations': organizations,
            }
        }
```

Step 2: Update Settings

Edit `jupyterhealth-exchange/jhe/settings.py`:

```
# BEFORE (line 204-209):
OAUTH2_PROVIDER = {
    "OIDC_ENABLED": True,
    "OIDC_RSA_PRIVATE_KEY": os.getenv("OIDC_RSA_PRIVATE_KEY"),
    "SCOPES": {"openid": "OpenID Connect scope"},
    "ACCESS_TOKEN_EXPIRE_SECONDS": 1209600,  # 2 weeks
}

# AFTER:
```

```
OAUTH2_PROVIDER = {
    "OIDC_ENABLED": True,
    "OIDC_RSA_PRIVATE_KEY": os.getenv("OIDC_RSA_PRIVATE_KEY"),
    "SCOPES": {"openid": "OpenID Connect scope"},
    "ACCESS_TOKEN_EXPIRE_SECONDS": 3600,  # 1 hour (reduced from 2 weeks)
    "OAUTH2_VALIDATOR_CLASS": "core.oauth_validators.JHEOAuth2Validator",
# NEW
}
```

Step 3: Test the Implementation

Verify ID Token Contains Claims

```
# In Django shell or test
import jwt
from oauth2_provider.models import AccessToken

# Get a token for a test user
token = AccessToken.objects.filter(user__email='sam@example.com').first()

# Decode ID token (if you have access to it)
# In production, the MCP server would do this
decoded = jwt.decode(id_token, options={"verify_signature": False})

print(decoded)
# Expected output:
# {
#   # Standard OIDC claims (always present):
#   "iss": "https://jhe.fly.dev/o",
#   "sub": "20001",
#   "aud": "Ima7rx8D6eko0PzlU1jK28WBUT2ZweZj7mqVG2wm",
#   "exp": 1729283400,
#   "iat": 1729197000,
#   "auth_time": 1729197000,
#
#   # Custom claims added by JHEOAuth2Validator:
#   "user_type": "practitioner",
#   "user_id": 20001,
#   "jhe_permissions": {
#     "studies": [30001, 30002, 30003, 30004, 30005, 30006, 30007, 30008],
#     "organizations": [
#       {"id": 50001, "name": "Berkeley Institute for Data Science
(BIDS)", "role": "manager"},
#       {"id": 50002, "name": "Cardiology", "role": "member"}
#     ]
#   }
#
#   # Optional claims (may be present depending on OAuth flow):
#   # "at_hash": "dGhpcyBpcyBhIGhhc2g",  # Access token hash
#   # "nonce": "xyz789..."                # If nonce was in auth request
# }
```

Security Considerations

What's in the ID Token (**Readable**)

- User email (already in standard OIDC claims)
- User ID
- Study IDs (e.g., 30001, 30002)
- Organization names and IDs
- User's role in each organization

What's Protected (**Not Readable**)

- Access token (opaque, required for API calls)
- Patient data (requires valid access token)
- User passwords (never in any token)

Security Trade-offs

What's Disclosed	Risk Level	Why It's Acceptable
Study IDs (integers)	Low	Study IDs are not PHI; cannot be used to access patient data without valid access token
Organization IDs (integers)	Low	Organization IDs are internal identifiers; cannot be used to access data without valid access token
User roles (member/manager)	Low	Role information doesn't grant access; only describes what access user already has
User ID	None	Already included in standard `sub` claim

Key Security Facts:

- ✅ ID token is signed (tamper-proof) - cannot be modified to gain additional access
- ✅ Access token (opaque) is still required for all API calls - ID token alone grants no data access
- ✅ Study and Organization IDs are not considered sensitive - they're internal identifiers, not PHI
- ❌ If Study and Organization IDs were considered sensitive, use encrypted JWE tokens (significant development effort)

Future Enhancements

1. **Claim size limits**: Cap studies at 50, return "many_studies": true flag
2. **Claim versioning**: Add `claims_version: 1` to support future schema changes
3. **Token refresh**: Implement refresh token flow that updates claims

Questions & Answers

Q: Can someone use the ID token to query patient data? A: No. The access token (opaque, secure) is required for all API calls. ID token is only for identification.

Q: What if study IDs are considered sensitive? A: Consider including only a count (`study_count: 8`) instead of IDs, or use encrypted JWE tokens (requires significant development).

Q: How do we handle users with 100+ studies? A: Implement size limits - include first 50 studies and add `partial: true` flag, with MCP server falling back to API calls.

Q: What happens to existing tokens after deployment? A: Old tokens (without custom claims) remain valid until expiry. MCP server should handle both formats gracefully.

References

- Django OAuth Toolkit - OIDC Support
- OAuth2Validator API
- JWT.io - Decode and inspect JWTs