Integration of Django Web Applications with RAP and GMS via OIDC

Issue/Rev. No.: 1.0.0
Date: March 27, 2026

Authors: Andrea Bignamini, Cristiano Urban
Affiliations: INAF - Osservatorio Astronomico di Trieste

Approved by: Cristina Knapic

Contents

Introduction

The **Remote Authentication Portal (RAP)** and the **Group Management Service (GMS)** are fundamental components of the **IA2 (Italian center for Astronomical Archives)** infrastructure, designed to provide centralized authentication and authorization services for the astronomical community. RAP acts as a Single Sign-On (SSO) provider, while GMS manages user permissions and group memberships.

The integration of web applications with this ecosystem is typically achieved via the **OpenID Connect (OIDC)** protocol. However, the specific implementation of RAP requires certain architectural adjustments to ensure compatibility with standard libraries. For instance, typical OIDC flows rely on a `userinfo_endpoint` that may not be standard in the RAP environment, necessitating the extraction of user claims directly from the Identity Token (ID Token).

This report presents a reference implementation and a "Starter Kit" for **Django** web applications. The solution utilizes the `mozilla-django-oidc` library with custom backend overrides to facilitate seamless authentication via RAP and efficient retrieval of user groups from the GMS API using OIDC Access Tokens.

Acronyms

GMS Group Management Service
IA2 Italian center for Astronomical Archives
OIDC OpenID Connect
RAP Remote Authentication Portal
SSO Single Sign-On

References

1. Django Framework Documentation. https://docs.djangoproject.com/
2. mozilla-django-oidc Documentation. https://mozilla-django-oidc.readthedocs.io/
3. IA2 Django RAP/GMS Integration Starter Kit Repository. https://www.ict.inaf.it/gitlab/ia2/django-rap-gms

Architecture

The integration follows a standard OpenID Connect (OIDC) Authorization Code Flow, specifically tailored to interact with the IA2 infrastructure. The architecture involves three main entities: the **Django Web Application**, the **RAP Service** (Authentication Service), and the **GMS Service** (Authorization Service).

The following steps describe the authentication and authorization lifecycle:

1. **Authentication Request**: When a user attempts to log in, the Django application redirects the browser to the RAP Authorization Endpoint.
2. **User Authentication**: The user authenticates directly on the RAP SSO platform. Upon success, RAP redirects the user back to the application with an authorization code.
3. **Token Exchange**: The Django application (via the `mozilla-django-oidc` library) contacts the RAP Token Endpoint to exchange the authorization code for three tokens:
   - **ID Token**: Contains user identity information (claims).
   - **Access Token**: A Bearer token used to access protected resources (GMS).
   - **Refresh Token**: Used to obtain new tokens without re-authenticating (if configured).
4. **Identity Processing (Custom Backend)**: Due to the specific configuration of the RAP environment, the standard OIDC "UserInfo" request is bypassed. Instead, the application uses a custom authentication backend to extract user details (such as email, name, and unique identifier) directly from the validated ID Token payload.
5. **Authorization (GMS Integration)**: Once authenticated, the application uses the stored Access Token to perform a server-to-server request to the GMS API. This request retrieves the list of groups or "Virtual Organizations" (VO) the user belongs to.
6. **Session Management**: User identity and GMS groups are stored in the Django session, allowing the application to implement fine-grained access control based on group membership.

This architecture ensures that the application never handles user credentials directly, relying instead on secure, short-lived tokens.

Installation & Setup

This section provides detailed instructions for preparing the development environment and installing the necessary components to integrate a Django application with the IA2 infrastructure.

System Requirements

Before proceeding, ensure that the following prerequisites are met:

- **Python**: Version 3.8 or higher.
- **Django**: Version 4.2 or higher.
- **RAP Credentials**: A valid **Client ID** and **Client Secret** provided by IA2 administrators.

- **Network Configuration**: Authorized Redirect URIs (e.g., `http://localhost:8000/oidc/callback/`) must be registered within the RAP provider.

Environment Preparation

It is strongly recommended to use a Python virtual environment to isolate the project dependencies and prevent conflicts with system-level packages.

1. **Clone the Repository**: Retrieve the starter kit files from the INAF ICT GitLab instance: `bash git clone --depth 1 [https://www.ict.inaf.it/gitlab/ia2/django-rap-gms.git](https://www.ict.inaf.it/gitlab/ia2/django-rap-gms.git)cd django-rap-gms`
2. **Create and Activate a Virtual Environment**:

```
# Create the environment
python -m venv venv

# Activate the environment (Linux/macOS)
source venv/bin/activate
```

Installing Dependencies

The project relies on a specific set of libraries to handle the OIDC flow, secure token verification, and API communication. Once the virtual environment is active, install the required packages using the provided `requirements.txt` file:

```
pip install -r requirements.txt
```

Core Components

The installation includes the following essential libraries:

- **Django**: The primary web framework.
- **mozilla-django-oidc**: The standard library for OIDC integration in Django, used as the foundation for RAP authentication.
- **requests**: Employed for performing server-side HTTP calls to the GMS API.
- **python-dotenv**: Used to securely manage sensitive credentials via environment variables.
- **PyJWT[crypto]**: A security dependency required for the cryptographic verification of tokens.

Configuration

The configuration of the Django application is divided into two parts: the management of sensitive credentials through environment variables and the integration of the OIDC protocol within the Django settings.

Environment Variables

To ensure security and portability, all sensitive information and provider-specific endpoints are stored in a `.env` file located in the project root. This file should never be committed to version control.

The configuration requires the following variables:

Provider Credentials

These values are unique to each application and must be requested from the IA2 team: * **OIDC_RP_CLIENT_ID**: The unique identifier for your application. * **OIDC_RP_CLIENT_SECRET**: The secret key used for authentication. * **OIDC_OP_LOGOUT_ENDPOINT**: The URL where users are redirected after logging out (e.g., `http://localhost:8000/logout`).

IA2 Service Endpoints

These endpoints are standard for the IA2 infrastructure and typically do not require modification: * **OIDC_OP_AUTHORIZATION_ENDPOINT**: `https://sso.ia2.inaf.it/rap-ia2/auth/oauth2/authorize` * **OIDC_OP_TOKEN_ENDPOINT**: `https://sso.ia2.inaf.it/rap-ia2/auth/oauth2/token` * **OIDC_OP_JWKS_ENDPOINT**: `https://sso.ia2.inaf.it/rap-ia2/auth/oidc/jwks` * **OIDC_OP_USER_ENDPOINT**: A placeholder URL (e.g., `https://sso.ia2.inaf.it/rap-ia2/userinfo-placeholder`) required to satisfy the library's internal checks.

Django Settings Configuration

The `settings.py` file must be updated to include the `mozilla-django-oidc` library and point to the custom authentication logic.

Loading Environment Variables

At the beginning of the settings file, the `python-dotenv` library is used to load the variables defined in the `.env` file:

```
import os
from dotenv import load_dotenv

load_dotenv()
```

Installed Apps and Authentication Backends

The OIDC library must be added to the `INSTALLED_APPS`. Furthermore, the `AUTHENTICATION_BACKENDS` must be updated to prioritize the custom RAP backend:

```
INSTALLED_APPS = [
    # ...
```

```
    "mozilla_django_oidc",
    "core", # Your application name
]

AUTHENTICATION_BACKENDS = [
    'core.backends.RAPBackend',
    'django.contrib.auth.backends.ModelBackend',
]
```

OIDC Protocol Settings

Specific parameters are required to ensure compatibility with the RAP implementation:

- **Signature Algorithm**: Set to `OIDC_RP_SIGN_ALGO = "RS256"`.
- **Nonce Handling**: `OIDC_USE_NONCE = False` is configured to disable the standard OIDC nonce verification, ensuring compatibility with the RAP provider's authentication flow.
- **Authentication Method**: `OIDC_TOKEN_USE_BASIC_AUTH = True` is necessary to pass credentials correctly during the token exchange.
- **Scopes**: The scopes `OIDC_RP_SCOPES` must include `openid email read:gms read:rap` to allow access to user identity and GMS group information.
- **Token Storage**: `OIDC_STORE_ACCESS_TOKEN = True` must be enabled to allow the application to reuse the token for subsequent GMS API calls.
- **ID Token Persistence**: `OIDC_STORE_ID_TOKEN = True` must be enabled to store the Identity Token in the user session, which is required for the custom backend to extract user details without a UserInfo endpoint.

```
OIDC_OP_AUTHORIZATION_ENDPOINT = os.getenv('OIDC_OP_AUTHORIZATION_ENDPOINT')
OIDC_OP_TOKEN_ENDPOINT = os.getenv('OIDC_OP_TOKEN_ENDPOINT')
OIDC_OP_JWKS_ENDPOINT = os.getenv('OIDC_OP_JWKS_ENDPOINT')
OIDC_OP_USER_ENDPOINT = os.getenv('OIDC_OP_USER_ENDPOINT')

OIDC_RP_CLIENT_ID = os.getenv('OIDC_RP_CLIENT_ID')
OIDC_RP_CLIENT_SECRET = os.getenv('OIDC_RP_CLIENT_SECRET')

OIDC_RP_SIGN_ALGO = "RS256"
OIDC_USE_NONCE = False
OIDC_TOKEN_USE_BASIC_AUTH = True
OIDC_RP_SCOPES = "openid email read:gms read:rap"

OIDC_STORE_ACCESS_TOKEN = True
OIDC_STORE_ID_TOKEN = True

LOGIN_REDIRECT_URL = '/'
LOGOUT_REDIRECT_URL = '/'
OIDC_OP_LOGOUT_ENDPOINT = os.getenv('OIDC_OP_LOGOUT_ENDPOINT')
```

URL Configuration

Finally, the OIDC authentication routes must be included in the project's `urls.py`:

```
from django.urls import path, include

urlpatterns = [
    # ...
    path('oidc/', include('mozilla_django_oidc.urls')),
]
```

Implementation Details

The core of the integration lies in two main components: a custom authentication backend to handle RAP-specific identity flows and a retrieval logic to interact with the GMS API.

Custom Authentication Backend

Standard OIDC libraries, such as `mozilla-django-oidc`, typically follow a two-step process to retrieve user information: first, they validate the ID Token, and then they perform a call to a `userinfo_endpoint` to fetch additional profile details. In the RAP environment, the `userinfo_endpoint` is not standard, which would cause the authentication process to fail with a `404 Not Found` error.

The solution implemented in this project is the `RAPBackend` class, which extends the standard `OIDCAuthenticationBackend`.

Key Overrides

The following methods are overridden to ensure compatibility:

- **`verify_token`**: This method is updated to accept additional keyword arguments (`**kwargs`). This prevents a common `TypeError` when the RAP provider sends a `nonce` parameter during token verification.
- **`get_or_create_user`**: This is the most critical override. It is modified to skip the automated call to the `userinfo_endpoint`. Instead, it passes the validated claims directly to the user creation logic.
- **`create_user`**: Since the user info call is skipped, this method extracts user details—such as `email`, `given_name` (first name), and `family_name` (last name)—directly from the payload of the ID Token. It uses the email or the unique `sub` (subject) identifier to create or retrieve the user in the Django database.

```
import logging
from mozilla_django_oidc.auth import OIDCAuthenticationBackend

# Logger configuration for debugging
logger = logging.getLogger(__name__)

class RAPBackend(OIDCAuthenticationBackend):
    """Custom OIDC authentication backend for RAP."""
```

```python
    def verify_claims(self, claims):
        """Verify that the token contains minimum required data."""
        logger.debug(f"DEBUG - Claims received from RAP: {claims}")

        verified = super().verify_claims(claims)
        # Check for 'email' or 'sub' if RAP does not send standard fields
        return verified and ('email' in claims or 'sub' in claims)

    def get_or_create_user(self, access_token, id_token, payload):
        """
        Called by authenticate() immediately after token verification.
        Overrides default to skip user info endpoint calls (avoid 404).
        """
        logger.debug(f"DEBUG: get_or_create_user called. Payload: {payload}"
            ↪ )

        # Payload contains token claims (email, sub, etc.)
        user = self.create_user(payload)
        return user

    def create_user(self, claims):
        """Create or retrieve user based on claims."""
        logger.debug(f"DEBUG: Creating user for: {claims.get('email')}")

        username = claims.get('email', claims.get('sub'))
        email = claims.get('email', '')
        logger.debug(claims)

        user, created = self.UserModel.objects.get_or_create(
            username=username,
            defaults={
                'email': email,
                'first_name': claims.get('given_name', ''),
                'last_name': claims.get('family_name', '')
            }
        )
        return user

    def verify_token(self, token, **kwargs):
        """Verify token validity, accepting additional kwargs to prevent
            ↪ TypeError."""
        return super().verify_token(token, **kwargs)
```

GMS Group Retrieval

`RAPBackend` Class Authorization within the IA2 ecosystem is managed through GMS groups. While authentication confirms *who* the user is, GMS identifies the groups or Virtual Organizations (VO) associated with the account; the application then uses these group memberships to determine *what* the user is allowed to do.

The Retrieval Process

The retrieval of groups is performed server-side once the user has successfully authenticated:

1. **Access Token Retrieval**: The application retrieves the OIDC Access Token stored in the user's session. This token acts as a Bearer credential.
2. **API Call**: The application performs an HTTP GET request to the GMS search endpoint: `https://sso.ia2.inaf.it/gms/vo/search`.
3. **Request Headers**: The request must include the following headers:
   - `Authorization: Bearer <access_token>`
   - `Accept: text/plain`
4. **Data Processing**: The GMS API returns the list of groups in a plain text format (one group per line). The application parses this response into a Python list and cleans the data by removing whitespace and empty lines.
5. **Caching**: To optimize performance and avoid redundant network calls, the list of groups is cached in the Django session (`user_groups_cache`). This allows the application to perform group-based permission checks throughout the user's session without re-querying the GMS server.

```
import requests
from django.shortcuts import render
from django.http import HttpResponse

def home(request):

    # Try to retrieve from session to avoid redundant calls
    user_groups = request.session.get('user_groups_cache')

    if user_groups is None and request.user.is_authenticated:

        # Retrieve Access Token (ensure OIDC_STORE_ACCESS_TOKEN = True in
        ↪ settings)
        access_token = request.session.get('oidc_access_token')

        if access_token:
            # GMS URL endpoint
            gms_url = "https://sso.ia2.inaf.it/gms/vo/search"

            # Mimic headers
            headers = {
                'Accept': 'text/plain',
                'Authorization': f'Bearer␣{access_token}'
            }

            try:
                response = requests.get(gms_url, headers=headers, timeout
                ↪ =10)

                if response.status_code == 200:
                    # Since it's text/plain, read text and split into a list
```

```
                    # .strip() removes whitespace, .splitlines() creates a
                        ↪ line-by-line list
                    raw_text = response.text
                    user_groups = [line.strip() for line in raw_text.
                        ↪ splitlines() if line.strip()]
                else:
                    print(f"GMS Error: {response.status_code} - {response.
                        ↪ text}")

            except requests.exceptions.RequestException as e:
                print(f"GMS connection error: {e}")

    request.session['user_groups'] = user_groups

    return render(request, 'core/home.html', {'user_groups': user_groups})
```

Troubleshooting

Example `view.py` to Retrieve User Groups Integrating with a specialized OIDC provider like RAP can lead to specific configuration issues. Below are the most common errors and their solutions as identified during the development of this starter kit.

Common Issues

- **ImproperlyConfigured: Setting OIDC_OP_USER_ENDPOINT not found** The `mozilla-django-oidc` library expects a standard UserInfo endpoint. Since RAP authentication relies on the ID Token payload, a placeholder URL must still be defined in the `settings.py` or `.env` file to satisfy the library's internal requirements.
- **Unauthorized: Missing Authorization header** This error often occurs during the token exchange phase. It is necessary to ensure that `OIDC_TOKEN_USE_BASIC_AUTH` is set to `True` in the Django settings to allow the application to send its credentials correctly to the RAP server.
- **TypeError: verify_token() ... unexpected keyword argument 'nonce'** This is a frequent compatibility issue where the RAP provider sends a `nonce` parameter that the default library method does not expect. The solution is to use the provided `RAPBackend` in `backends.py`, which includes a signature for `verify_token` that correctly handles additional keyword arguments (`**kwargs`).
- **GMS Connection or Authorization Errors** If the application fails to retrieve groups, verify that `OIDC_STORE_ACCESS_TOKEN` is enabled. Without this setting, the Access Token is not persisted in the session and cannot be used for subsequent requests to the GMS API.

Conclusion

The integration of Django applications with the RAP and GMS infrastructure via OIDC provides a secure and centralized way to manage astronomical user identities and access rights. By utilizing the custom backend and logic provided in this reference implementation, developers can bypass common compatibility hurdles related with RAP and GMS.