Using EG1 for M2M authentication

Contents

Implementing authentication in a client

EG1 uses the **client credentials flow** to obtain tokens for machine-to-machine authentication, this flow uses a shared secret to authenticate clients and obtain time limited tokens.

Technical details

The **client credentials flow** is a standardized OAuth 2.0 flow designed for server-to-server communication, where no user is directly involved. In this model, the client authenticates directly with the authorization server using its client_id and client_secret to obtain an access token.

This token is then used to authenticate API requests to downstream services.

1. **Token Request**

   The client initiates the flow by sending a POST request to the identity provider's token endpoint. The request must include:

   - grant_type=client_credentials
   - client_id and client_secret (as basic auth header or in the request body)

```
POST /account/api/oauth/token HTTP/1.1
Host: account-public-service-gamedev.ol.epicgames.net
Authorization: Basic Base64(client_id:client_secret)
Content-Type: application/x-www-form-urlencoded

grant_type=client_credentials
```

2. **Token Response**
   If the credentials are valid, the authorization server responds with an access token (and optional expiration information):

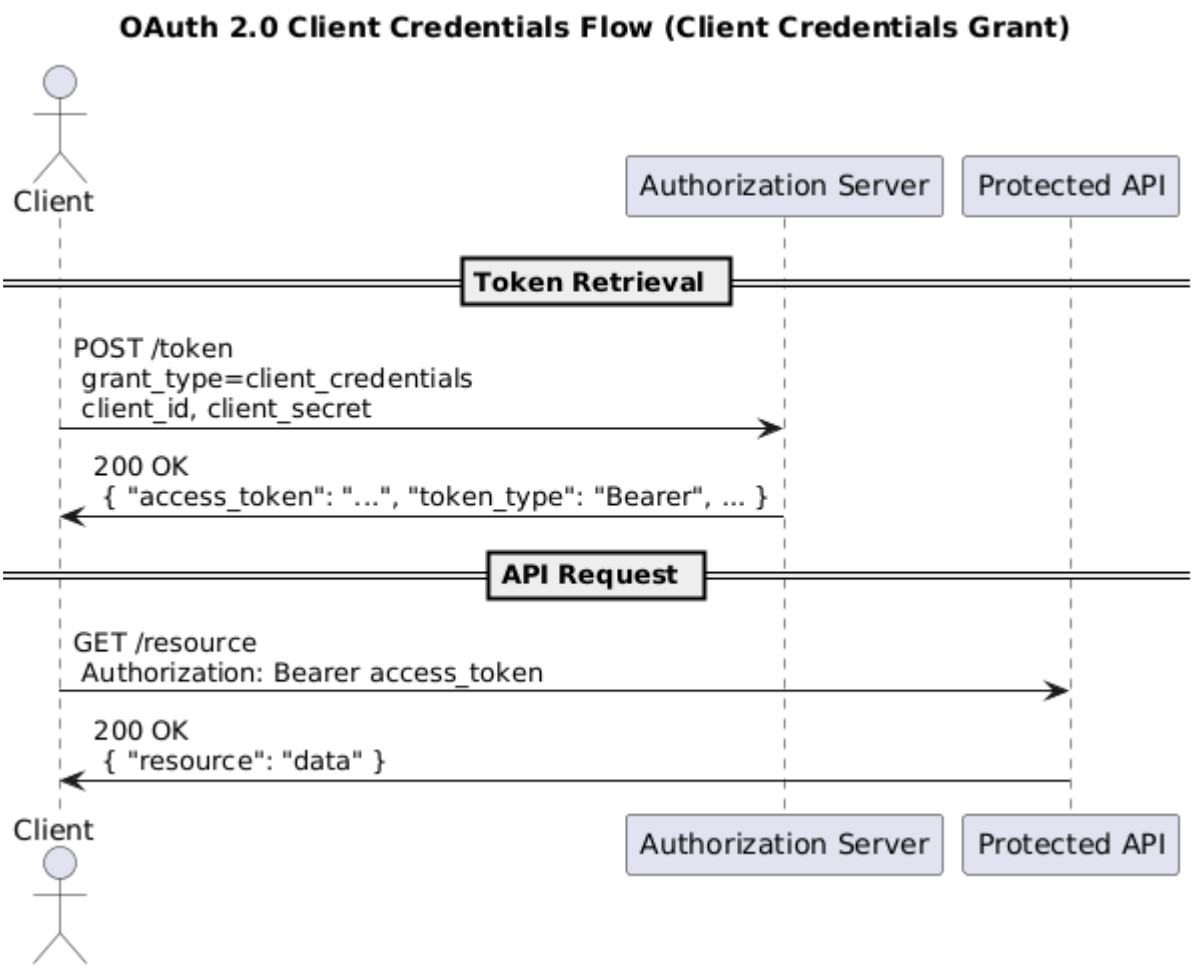
```
{
  "access_token": "eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...",
  "token_type": "Bearer",
  "expires_in": 3600
}
```

3. **Authenticated Requests**
   The client includes the access token in the Authorization header of subsequent API requests:

```
GET /resource HTTP/1.1
Host: api.service.local
Authorization: Bearer eyJhbGciOiJSUzI1NiIsInR5cCI6IkpXVCJ9...
```

Tokens are short lived so its the clients responsibility to cache and refresh them as needed.

Click for diagram...



Implementation guide

The first requirement for implementing a client is a client-id and client-secret. A client can be obtained via a request in the #on-identity-support-ext Slack channel where it will provisioned within Epicenter. Once the client-id and client-secret is obtained it is recommended they are stored in a secure location such as Vault.

Once a client-id and client-secret have been obtained the language specific changes can be made.

Language specific implementation

Mosaic macros cannot be exported to this format.

Implementing authentication/ authorization in a server

In order to ensure access is only granted to allowed clients the server must implement two steps:

- The token must be validated to **authenticate** the client.
- The tokens contents can be used to make a decision of if the client is **authorized**.

The validation can be achieved in two ways, "online" by using the authentications servers token introspection endpoint, or "offline" by validating the tokens signature inside the application.

Technical details

Online validation

Online validation refers to the process of verifying an access token by querying the **token introspection endpoint** of the authentication server. This endpoint is a part of the authorization server that returns metadata

about the token, such as its validity, scope, expiration, and associated user information.

How it works:

1. The client sends the access token to the authorization server's introspection endpoint.

```
curl -X GET https://account-public-service-gamedev.ol.epicgames.net
  -H "Authorization: Bearer <ACCESS_TOKEN>"
```

2. The server verifies the token and responds with a JSON object indicating whether the token is active and providing relevant token details (e.g., user ID, scopes).

```
{
  "token": "v2:xyz123exampletoken",
  "session_id": "abc123-session",
  "token_type": "bearer",
  "client_id": "my-client-id",
  "internal_client": false,
  "client_service": "game-service",
  "account_id": "user-456-account-id",
  "expires_in": 3600,
  "expires_at": "2025-05-19T14:30:00.000Z",
  "auth_method": "password",
  "lastPasswordValidation": "2025-05-19T13:30:00.000Z",
  "app": "my-game-app",
  "in_app_id": "user-game-id",
  "device_id": "device-789",
  "perms": [
    {
      "resource": "profile",
      "action": "read"
    },
    {
```

```
      "resource": "game_state",
      "action": "write"
    }
  ]
}
```

3. The application uses this information to authenticate the client and make authorization decisions.

Pros:

- Centralized control: Token revocation is immediately effective.
- Simple implementation: The server does not need to manage key material or token parsing logic.
- Real-time verification ensures tokens are still valid and not revoked.

Cons:

- Introduces latency due to network calls.
- Depends on the availability and performance of the authorization server.
- May become a bottleneck under high request volumes.

Offline validation

Offline validation means verifying the token **locally**, without contacting the authorization server. This is typically done by validating the **token's digital signature** using a public key (for JWTs, usually via RS256 or similar algorithms).

How it works:

1. The application receives the access token (typically a JWT).
2. It verifies the token's signature using a pre-shared or discovered public key (e.g., from a JWKS endpoint).

3. If the signature is valid and the token hasn’t expired, the application extracts claims and uses them for authentication and authorization.

Pros:

- No external calls: Faster and more scalable for high-throughput applications.
- Works even if the authorization server is temporarily unavailable.
- Suitable for distributed systems and microservices.

Cons:

- Tokens cannot be easily revoked before expiration.
- Requires proper handling of key rotation and secure key management.
- More complex implementation compared to introspection.

Implementation guide

Mosaic macros cannot be exported to this format.