Ego Documentation

Release 0.1.0

overture.bio

Sep 11, 2019

First Steps

CHAPTER 1

Introduction

1.1 What is Ego?

EGO is an OAuth2 based authentication and authorization management microservice. It allows users to login and authenticate themselves using their existing logins from sites such as Google and Facebook, create and manage authorization tokens, and use those tokens to interact with Ego-aware third party applications which they are authorized for.

OAuth single sign-on means that Ego does not need to manage users and their passwords; and similarly, none of the services that use Ego need to worry about how to manage users, logins, authentication or authorization. The end user simply sends them a token, and the service checks with Ego to learn who the token is for, and what permissions the token grants. EGO is one of many products provided by Overture and is completely open-source and free for everyone to use.

See also:

For additional information on other products in the Overture stack, please visit https://overture.bio

1.2 Features

- Single sign-on for microservices
- User authentication through federated identities such as Google, Facebook, Linkedin, Github (Coming Soon), ORCID (Coming Soon)
- Provides stateless authorization using JSON Web Tokens (JWT)
- Can scale very well to large number of users
- Provides ability to create permission lists for users and/or groups on user-defined permission entities
- Standard REST API that is easy to understand and work with
- Interactive documentation of the API is provided using Swagger UI. When run locally, this can be found at : http://localhost:8080/swagger-ui.html

- Built using well established Frameworks - Spring Boot, Spring Security

1.3 License

CHAPTER 2

Getting Started

The easiest way to understand EGO, is to simply use it!

Below is a description of how to get Ego quickly up and running, as well as a description of how Ego works and some important terms.

2.1 Quick Start

The goal of this quick start is to get a working application quickly up and running.

Using Docker:

1. Download the latest version of Ego.
2. From the Ego root directory, set the API_HOST_PORT where Ego is to be run, then run docker-compose:

```
$ API_HOST_PORT=8080 docker-compose up -d
```

Ego should now be deployed locally with the Swagger UI at http://localhost:8080/swagger-ui.html

Alternatively, see the Installation instructions.

2.2 How Ego Works

1. An Ego administrator configures Ego.

- Registers a unique client-id and application password for each application that will use Ego for Authorization.
- Creates a policy for every authorization scope that an application will use.
- Registers users and groups, and sets them up with appropriate permissions for policies and applications.

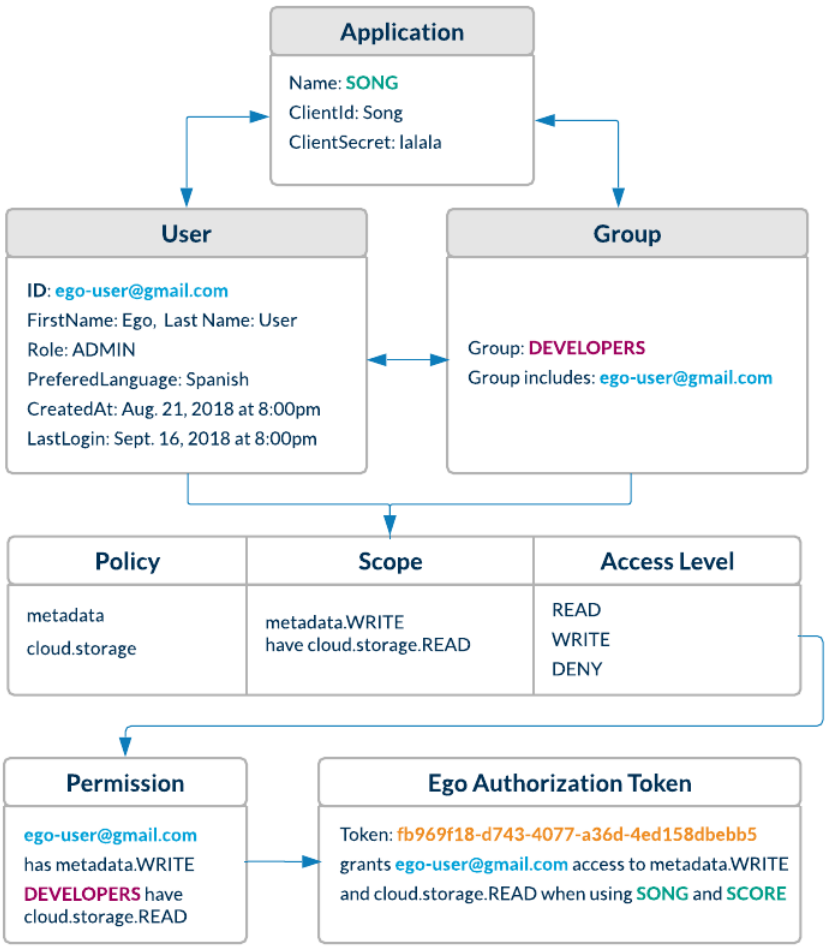
2. Ego grants secret authorization tokens to individual users to represent their permissions.

- Authorization tokens expire, and can be revoked if compromised.
- Individuals can issue tokens for part or all of their authority, and can limit the authority to specific applications.
- Users (and programs operating on their behalf) can then use these tokens to access services.

3. Individual services make a REST call to EGO to determine the user and authority represented by a token.

- Makes a call to Ego's check_token endpoint and validates the user's authorization to access the requested services.

2.3 Terms Used in Ego



User A user is any individual registered in Ego who needs to authorize themselves with Ego-aware applications.

Admin An admin is a power user whose role is set to 'ADMIN'. Only admins are authorized to register users, groups, applications & policies using Ego's REST endpoints.

Group A group of users with similar properties. Admins can create new groups and add users to them. They can then assign permissions to an entire group which will be reflected for each user in that group.

Policy A policy is a scope or context for which an application may want to grant a user or group READ/WRITE/DENY permissions.

Permission A user or group can be given READ/WRITE/DENY permissions for a particular policy.

Application An application is a third party service that registers itself with EGO so that EGO can authorize users on its behalf. Upon registration, the service must provide a client_id and client secret.

Application Authentication Token This a Basic JWT token which encodes a client id and secret, and authorizes an application to interact with Ego. This is passed in the authorization request header when an application uses the check_token endpoint in order to check a user's token.

User Authentication Token This is a Bearer token which encodes user information, and is passed to a user when they are authenticated through OAuth single sign-on. This Bearer token is passed in the request authorization header whenever the user wants to access Ego's resources. If the JWT denotes that a user has an ADMIN role, they are permitted to create and modify resources (users, groups, permissions, policies).

User Authorization Token This is a random token which is generated to authorize a user for a specific scope, in the context of an application.

2.4 Play with the REST API from your browser

If you want to play with EGO from your browser, you can visit the Swagger UI located here :

https://ego.overture.cancercollaboratory.org/swagger-ui.html

CHAPTER 3

Ego for Administrators

3.1 Tutorial

To administer Ego, the admin must:

1. Install Ego.

View the installation instructions.

2. Insert a new user with the admin's Oauth Id into the "egousers" table, with role ADMIN.

3. A developer creates a new Ego-aware application

a. Admin creates a new application in Ego with the client_id and password.

b. Admin creates new policies with new policy names

c. Admin assigns permissions to users/groups to permit/deny them access to the new application and policies

4. Admin creates or deletes groups, assigns user/group permissions, revoke tokens, etc. as necessary.

For example, an administrator might want to:

- Create a new group called **"QA"**, whose members are all the people in the "QA department"
- Create a group called "Access Denied" with access level "DENY" set for every policy in Ego
- Grant another user administrative rights (role ADMIN)
- Add a former employee to the group "AccessDenied", and revoke all of their active tokens.
- In general, manage permissions and access controls within Ego.

3.2 Using the Admin Portal

Ego provides an intuitive GUI for painless user management.

CHAPTER 4

Ego for Application Developers

To create an Ego-aware application, a developer must:

1. Pick a unique policy name for each type of authorization that the application requires.
2. Write the application. Ensure that the application does it's authorization by performing a call to Ego's "check_token" REST endpoint, and only grants access to the service for the user id returned by "check_token" if the permissions returned by "check_token" include the required permission.
3. Configure the program with a meaningful client_id and a secret password.
4. Give the client_id, password, and policy names to an Ego administrator, and ask them to configure Ego for you.

CHAPTER 5

Tokens

5.1 User Authentication Tokens

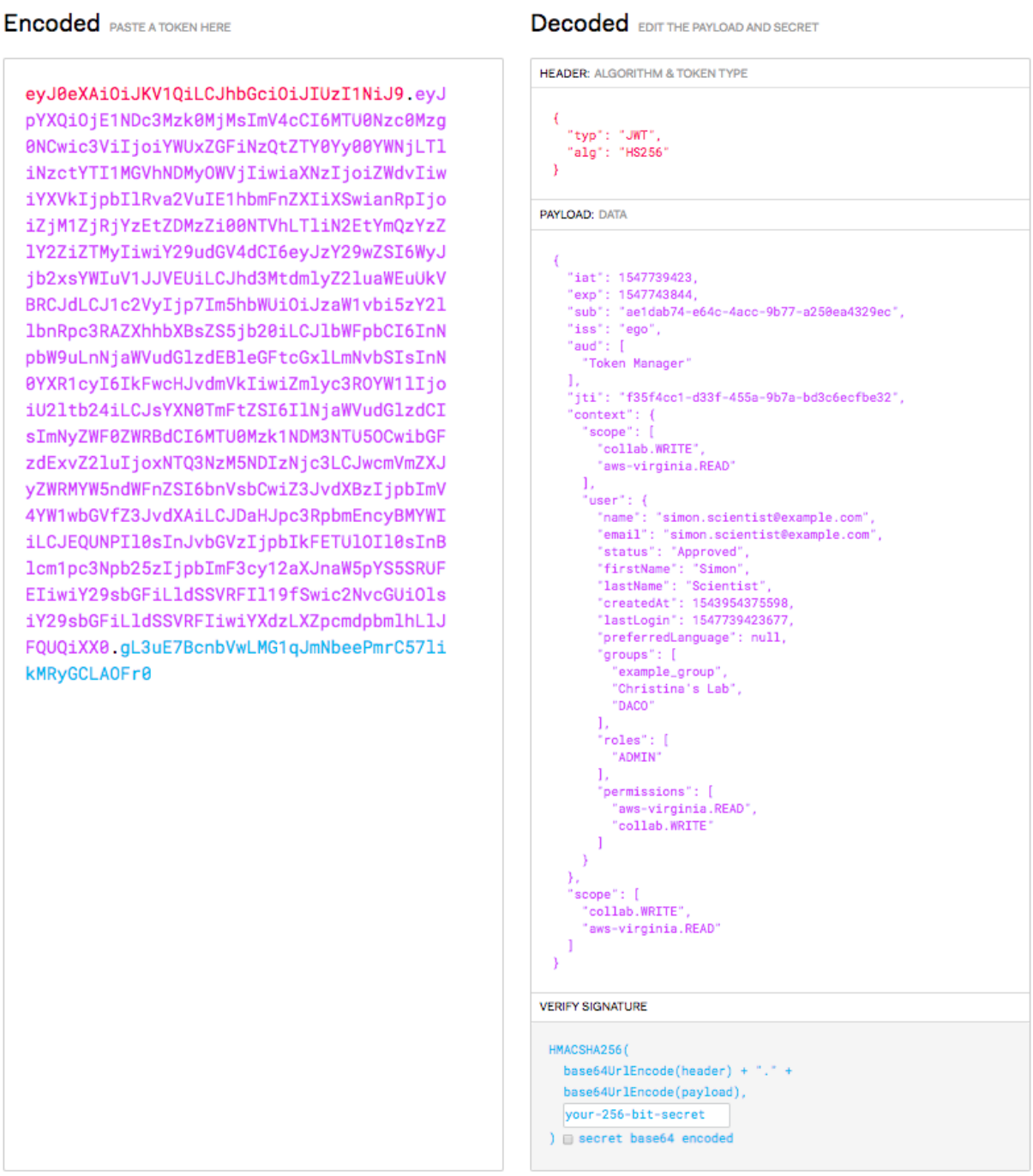
Authentication concerns *who the user is*.

User Authentication tokens are used to verify a user's identity.

Ego's User Authentication tokens are signed JSON Web Tokens (see http://jwt.io) that Ego issues when a user successfully logs into Ego using their Google or Facebook credentials.

Ego's authentication tokens confirm the user's identity, and contain information about a user's name, their role (user/administrator), and any applications, permissions, and groups associated with their Ego account etc.

This data is current as of the time the token is issued, and the token is digitally signed by Ego with a publicly available signing key that applications have to use to verify that an authentication token is valid. Most of Ego's REST endpoints require an Ego authentication token to be provided in the authorization header, in order to validate the user's identity before operating on their data.



5.2 User Authorization Tokens

Authorization concerns *what a user is allowed to do*.

User Authorization tokens are used to verify a user's permissions to execute on a desired scope.

Ego's User Authorization tokens are random numbers that Ego issues to users so they can interact with Ego-aware applications with a chosen level of authority.

Each token is a unique secret password that is associated with a specific user, permissions, and optionally, an allowed set of applications.

Unlike passwords, Authorization tokens automatically expire, and they can be revoked if the user suspects that they have been compromised.

The user can then use their token with Ego-authorized applications as proof of who they are and what they are allowed to do. Typically, the user will configure a client program (such as SING, the client program used with SONG, the ICGC Metadata management service) with their secret token, and the program will then operate with the associated level of authority.

In more detail, when an Ego-aware application wants to know if it is authorized to do something on behalf of a given user, it just sends their user authorization token to Ego, and gets back the associated information about who the user is (their user id), and what they are allowed to do (the permissions associated with their token). If the permissions that the user have include the permission the application wants, the application know it is authorized to perform the requested service on behalf of the user.

5.3 Application Authentication Tokens

For security reasons, applications need to be able to prove to Ego that they are the legitimate applications that Ego has been configured to work with.

For this reason, every Ego-aware application must be configured in Ego with it's own unique CLIENT ID and CLIENT SECRET, and the application must send a token with this information to Ego whenever it makes a request to get the identity and credentials associated with a user's authorization token.

CHAPTER 6

Installation

6.1 Step 1 - Setup Database

1. Install Postgres
2. Create a Database: ego with user postgres and empty password
3. Execute SQL Script to setup tables.

6.1.1 Database Migrations with Flyway

Database migrations and versioning is managed by flyway.

6.2 Step 2 - Run

EGO currently supports three Profiles:

- default: Use this to run the most simple setup. This lets you test various API endpoints without a valid JWT in authorization header.
- auth: Run this to include validations for JWT.
- secure: Run this profile to enable https

Run using Maven. Maven can be used to prepare a runnable jar file, as well as the uber-jar for deployment:

```
$ mvn clean package ; ./fly migrate
```

To run from command line with maven:

```
$ mvn spring-boot:run
```

CHAPTER 7

Architecture

CHAPTER 8

Technology Stack

This application is written in JAVA using Spring Boot and the Spring Security Frameworks.

8.1 JSON Web Token

8.1.1 Basics

Ego makes use of JSON Web Tokens (JWTs) for providing users with a Bearer token.

The RFC for JWTs can be found here: https://tools.ietf.org/html/rfc7519

The following is a useful site for understanding JWTs: https://jwt.io/

The following is the structure of an ego JWT:

```
{
    "alg": "HS512"
}
.
{
    "sub": "1234567",
    "iss": "ego:56fc3842ccf2c1c7ec5c5d14",
    "iat": 1459458458,
    "exp": 1459487258,
    "jti": "56fd919accf2c1c7ec5c5d16",
    "aud": [
        "service1-id",
        "service2-id",
        "service3-id"
    ],
    "context": {
        "user": {
            "name": "Demo.User@example.com",
            "email": "Demo.User@example.com",
```

(continues on next page)

(continued from previous page)

```
            "status": "Approved",
            "firstName": "Demo",
            "lastName": "User",
            "createdAt": "2017-11-23 10:24:41",
            "lastLogin": "2017-11-23 11:23:58",
            "preferredLanguage": null,
            "roles": ["ADMIN"]
        }
    }
}
.
[signature]
```

Notes

- "aud" field can contain one or more client IDs. This field indicates the client services that are authorized to use this JWT.
- "groups" will differ based on the domain of client services - each domain of service should get list of groups from that domain's ego service.
- "permissions" will differ based on domain of client service - each domain of service should get list of permissions from that domain's ego service.

Unit Tests using testcontainers will also run flyway migrations to ensure database has the correct structure

8.1.2 Library Support

The Java JWT library is used in Ego for providing support for encoding, decoding, and validating JWTs: https://github.com/jwtk/jjwt

8.2 Spring-Boot

Ego is a microservice written in Java 8 and Spring-Boot. It makes use of the following parts of the Spring and Spring-Boot framework:

- Web / Spring MVC w/ embedded tomcat
- Spring Security
- JDBC
- Spring Data JPA

Swagger docs are generated by Springfox : https://springfox.github.io/springfox/docs/current/

8.3 Ego Design Notes

1. OAuth Single Sign-On means that Ego doesn't need to manage users and their passwords; users don't need a new username or password, and don't need to trust any service other than Google / Facebook.
2. Ego lets users be in charge of the authority they give out; so they can issue secret tokens that are limited to the exact authority level they need to do a given task.

Even if a such a token becomes publicly known, it can't grant an outsider accesses to services or permissions that the token doesn't have – regardless of whether the user has more authority that they could have granted.

Tokens also automatically expire (by default, within 24 hours), and if a user suspects that a token may have become known to outsiders, they can simply revoke the compromised token, removing all of it's authority, then issue themselves a new secret token, and use it.

3. None of the services that use Ego uses need to manage worry about how to manage users, logins, authentication, or authorization. The end user simply sends them a token, and the service checks with Ego to learn who the token is for, and what permissions the token grants. If the permissions granted don't include the permissions the service needs, it denies access; otherwise, it runs the service for the given user.

CHAPTER 9

Contributing to the Ego Project

CHAPTER 10

Contribute

If you'd like to contribute to this project, it's hosted on github.

See https://github.com/overture-stack/ego

10.1 Indices and tables

- genindex
- search

Index