CIA Final Project - SSO and RBAC with KeyCloak

Ahmed Nouralla - a.shaaban@innopolis.university

Introduction

- **Identity and Access Management (IAM)** is a crucial topic for the security of any system that every developer should understand. Let's review the basics:
  - **Identification:** user claiming an identity (e.g., by specifying their name/email)
  - **Authentication:** user proves their identity (e.g., by means of a password/access key)
  - **Authorization:** user gets access to certain system resources/functionalities based on their proven identity (e.g., access to the admin portal is only available for system administrators).
- **Access Control** defines who should have access to what and how to enforce such access
  - As systems get more complex, access control can get very difficult to implement correctly and securely, and it's the reason why **"Broken Access Control"** is the most common security issue according to OWASP TOP 10 (2021).
- **Keycloak** is an open-source IAM solution that helps software developers "add authentication to applications and secure services with minimum effort."
  - It provides a feature-packed management console to quickly deploy functional login forms supporting Single-Sign On (SSO), social login, integration with existing directory servers, role-based access control and much more.
  - Keycloak is based on and supports standard protocols: OAuth 2.0, OpenID Connect, and SAML
- Two popular approaches for authorizing users (in the context of modern web applications) are:
  - **Server sessions:** stateful way in which the web server maintains a dynamic storage storing information about active client sessions.
  - **JSON Web Tokens (JWTs):** stateless way in which the web server provides clients with a token upon login (typically through HTTP cookies) and verifies it whenever they try accessing protected resources.

Methods

This work implements a minimal authentication/authorization scenario with three main components

- **Keycloak server:** pre-configured to authenticate web users through OpenID Connect.
- **Web server (Relying Party):** Python (Flask) application outsourcing authentication functionalities to Keycloak.
- **Web client:** a pure HTML/CSS/JS web page to communicate with the server.

Keycloak Deployment

> Official guide: https://www.keycloak.org/getting-started/getting-started-docker

- Create a compose file to quickly run a keycloak development instance with volumes for persisting data and easier configuration from the host.
  - Containers are the industry standard for reproducibility and easy migration to the cloud.

```yaml
name: demo

services:
  keycloak:
    container_name: keycloak
    ports:
        - "8080:8080"
    environment:
        - KC_BOOTSTRAP_ADMIN_USERNAME=admin
        - KC_BOOTSTRAP_ADMIN_PASSWORD=admin
    image: quay.io/keycloak/keycloak:26.0
    volumes:
      - keycloak_data:/opt/keycloak/data
    command: start-dev
```

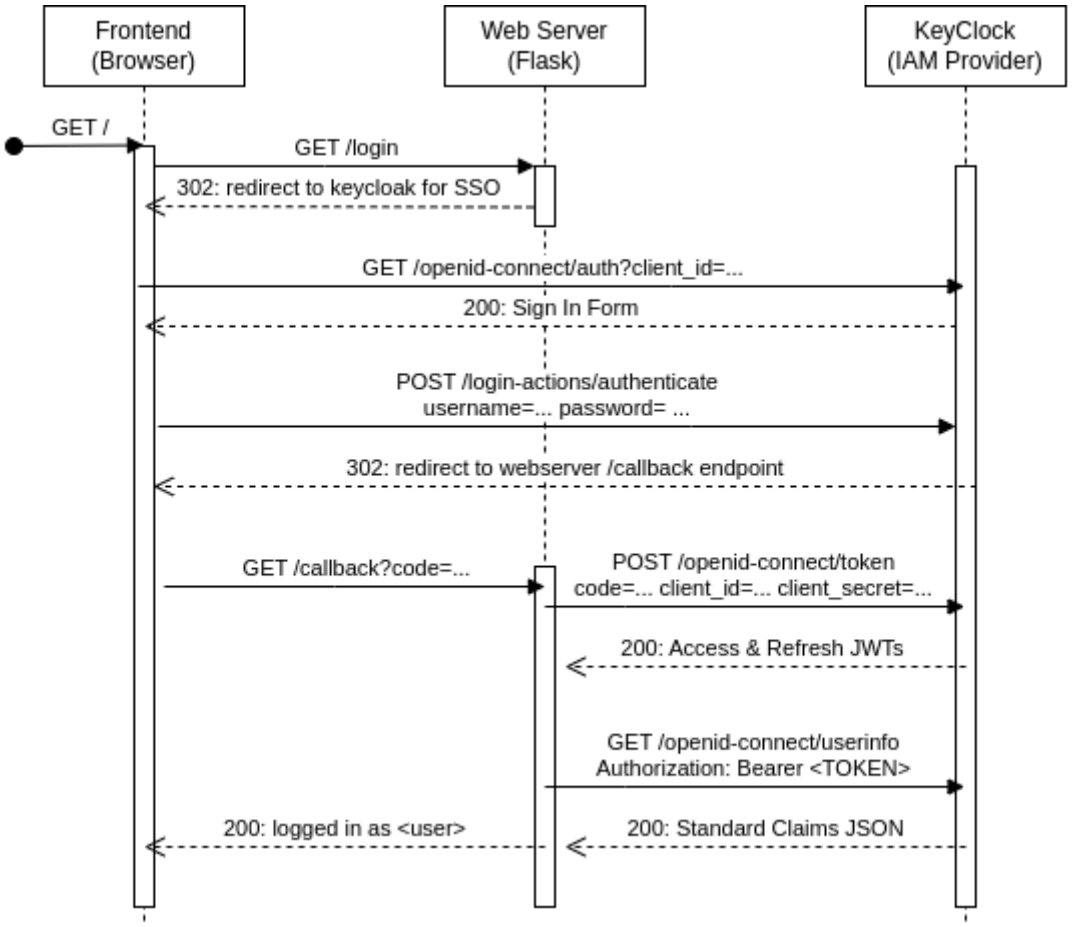
- Configure keycloak at http://localhost:8080 (login with default credentials from the compose file then change them for security). [Check results section below for screenshots of the process].
  - Create a realm (like a namespace for managing objects) called `myrealm`.
  - Create an OpenID Connect client (i.e., our web server). Enable authentication and authorization.
    - Note: including another service under SSO is only a matter of adding and configuring another client!
  - Configure RBAC: create 3 roles (`admin`, `editor`, and `viewer`) under the client as shown
  - Create 3 users (`user1`, `user2`, and `user3`) to simulate end-users for the application.
- Set a credential (i.e., password) and a role mapping (i.e., admin/editor/viewer) for each user.

Web Server Deployment

- Create a Python virtual environment with needed dependencies (Flask, PyJWT, requests, and python-dotenv). Export dependencies to `requirements.txt` for reproducibility.
- Write application logic handling the following endpoints
  - `/`: returns HTML content of application homepage
  - `/login`: called when user clicks login
  - `/callback`: called after a successful login
  - `/logout`: sign the logged-in user out
- Create a Dockerfile to containerize application for easier deployment, then add a service entry in the compose file to run the webserver along with keycloak.

```
app:
  container_name: app
  build: .
  ports:
    - "5000:5000"
```

- The following diagram illustrates the typical interaction between system components. It shows the exact HTTP requests and responses being exchanged for OpenID connect to work.



- Explanation for the process (adapted from https://openid.net/developers/how-connect-works/)

  1. User accesses the home page of the application via the browser.

2. User clicks "Log In" button and gets redirected to keycloak login form (with client id in query params).
    - Keycloak recognizes that the user is trying to login to the service `demo` in our example.
3. User logs in with their credentials (username and password) and gets redirected to the `/callback` endpoint (with authorization code)
4. Browser informs the web server of the code, which in turn contacts keycloak with the code, client id, and client secret to obtain the access token (JWT in our case).
3. The webserver may then:
    1. Supply the token in the `Authorization: Bearer <TOKEN>` to obtain standard claims about the end-user (e.g., user name and email) to process them in any way (e.g., send personalized greeting emails).
    2. Return the JWT to the browser (for localStorage or as HTTP-only cookie) so it can supply it for subsequent requests to the API (e.g., for accessing protected resources based on the user's logged-in status and role).

- To keep things simple, our demo application only returns the successful login status and basic info obtained about the client (name, email and role).
- This can be extended on to implement granular access control depending on the required functionality of the app.

Enabling TLS

- To illustrate how the infrastructure can be secured, sample self-signed certificates were generated using mkcert and used to configure HTTPS to/between webserver and keycloak.

```
mkdir certs
cd certs/
mkcert "*.internal.test" # Generate certs
mkcert -install          # Trust them in system and browsers

# Shorter file names used in configs
mv _wildcard.internal.test-key.pem tls.key
mv _wildcard.internal.test.pem tls.crt

# Rertieve issuing CA certificate from system trust store
cp /usr/local/share/ca-certificates/*.crt ca.crt
```

- Configured local hostnames at `/etc/hosts` for testing

```
127.0.0.1       app.internal.test
127.0.0.1       keycloak.internal.test
```

- Mount certs directory into keycloak and app containers then configure cert usage as follows:
    - **HTTPS to Keycloak:** set the following options in `keycloak.conf` with corresponding certificate locations and mount the config to `/opt/keycloak/conf/keycloak.conf`

```
# The file path to a server certificate or certificate chain in PEM format.
https-certificate-file=/certs/tls.crt

# The file path to a private key in PEM format.
https-certificate-key-file=/certs/tls.key

# HTTPS Port
https-port=443

# Hostname for the Keycloak server.
hostname=keycloak.internal.test
```

- **HTTPS to webserver:** use a Web Server Gateway Interface like `gunicorn` with the following command

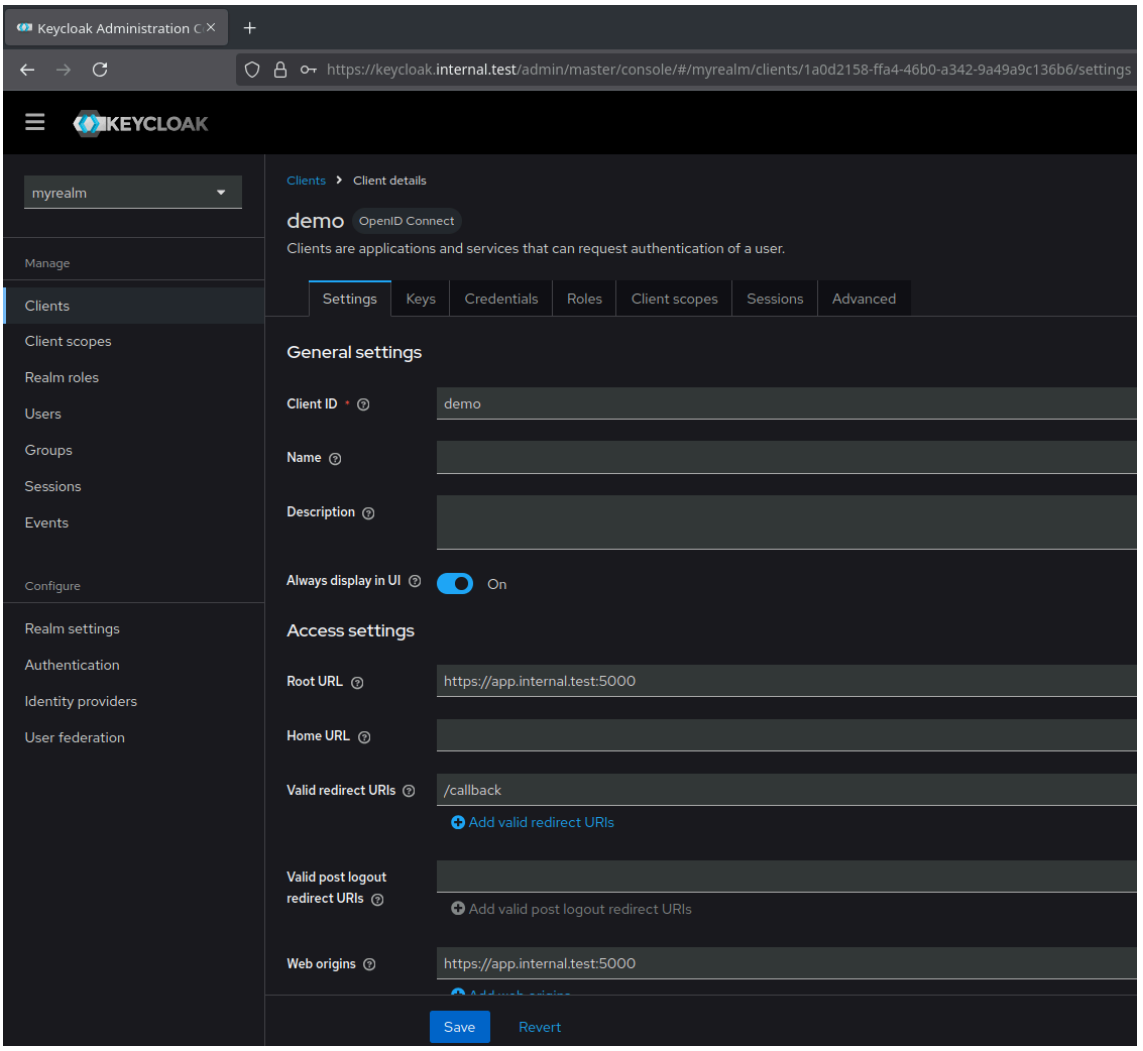
```
gunicorn -b 0.0.0.0:5000 --certfile=/app/certs/tls.crt --keyfile=/app/certs/tls.crt
```

- **HTTPS between webserver and keycloak:** for requests originating from the web server, one must add the option to explicitly trust the issuer of server certificate. This step will not be needed for globally trusted certificates.

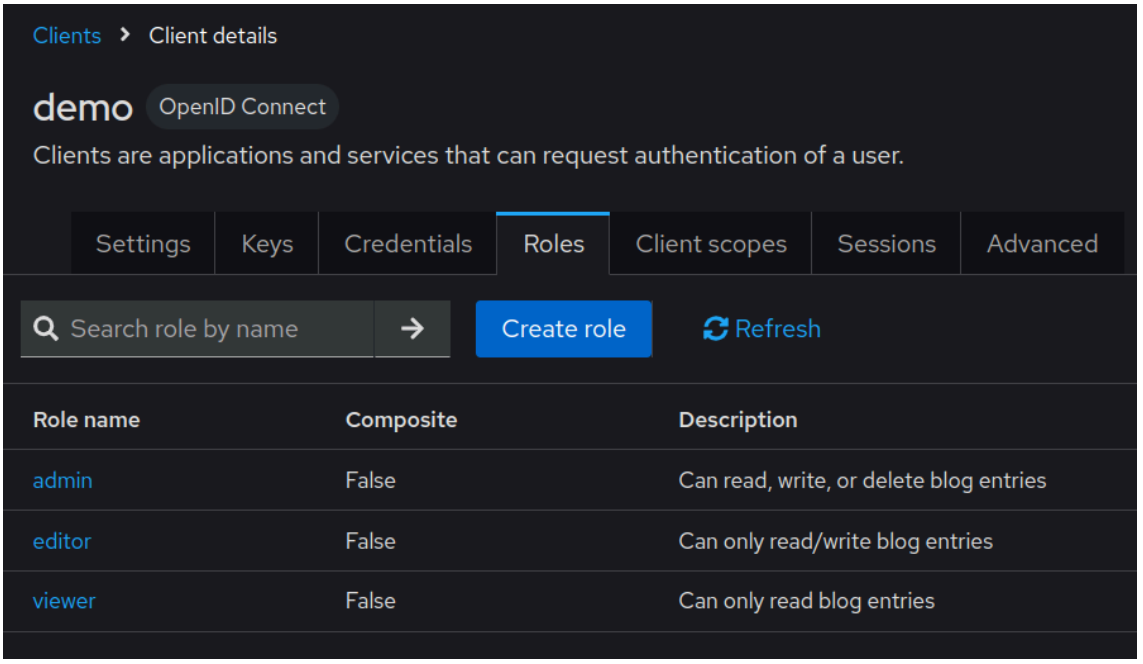
```
requests.get(..., verify="/app/certs/ca.crt")
```

Results

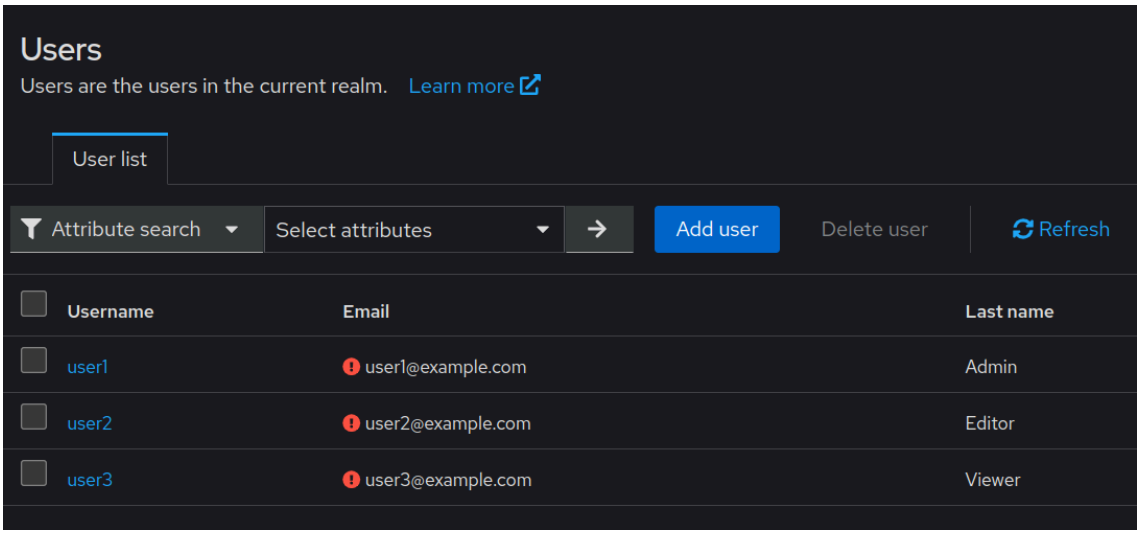
- Local Keycloak server showing the created realm and client configuration. The server is also accessed over HTTPS.



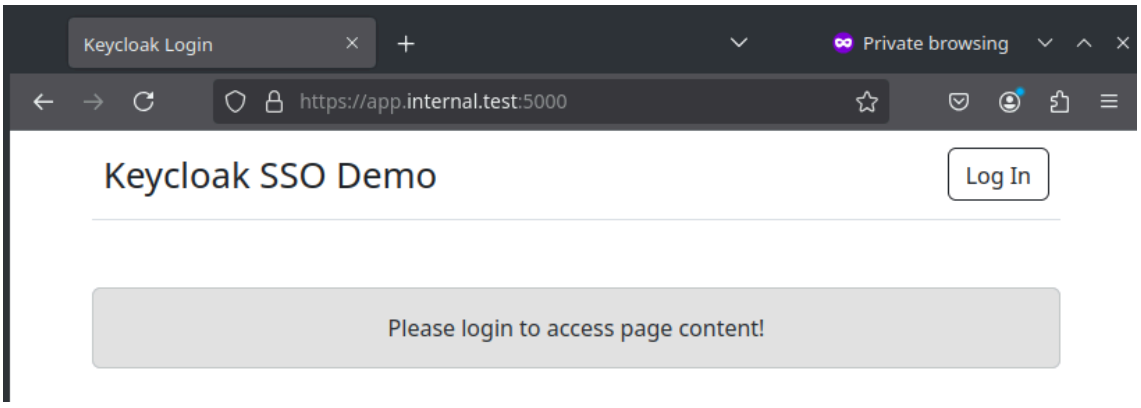
- Service roles for RBAC:



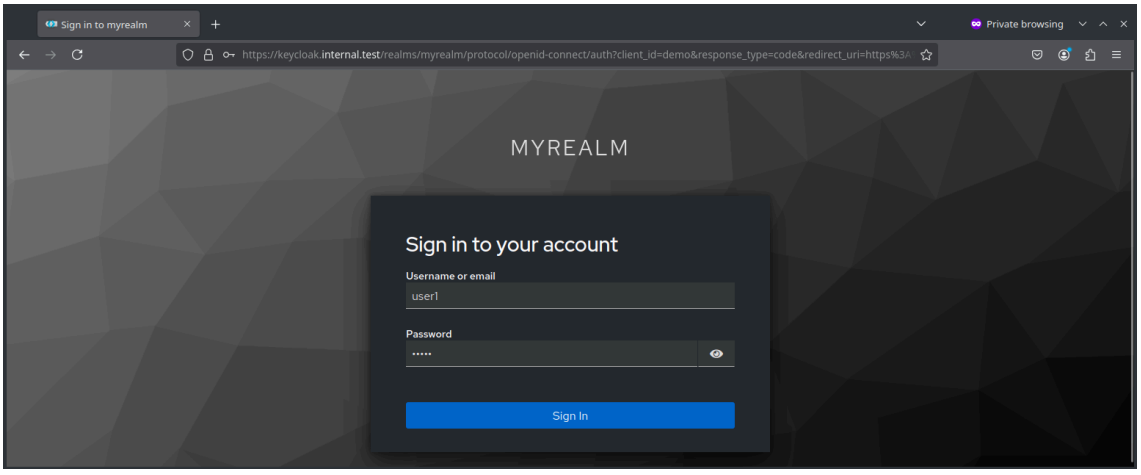
- Sample users for testing different roles:



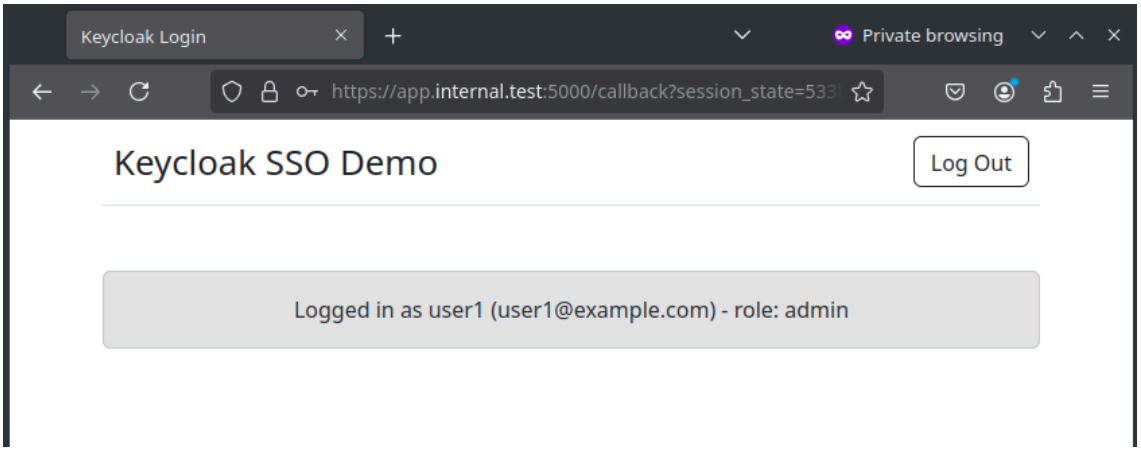
- Application webpage before logging in. Also accessed over HTTPS.



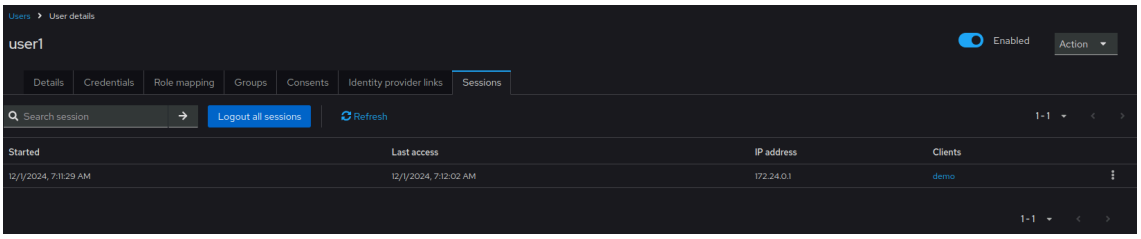
- Clicking "Log In" redirects to Keyclock login form



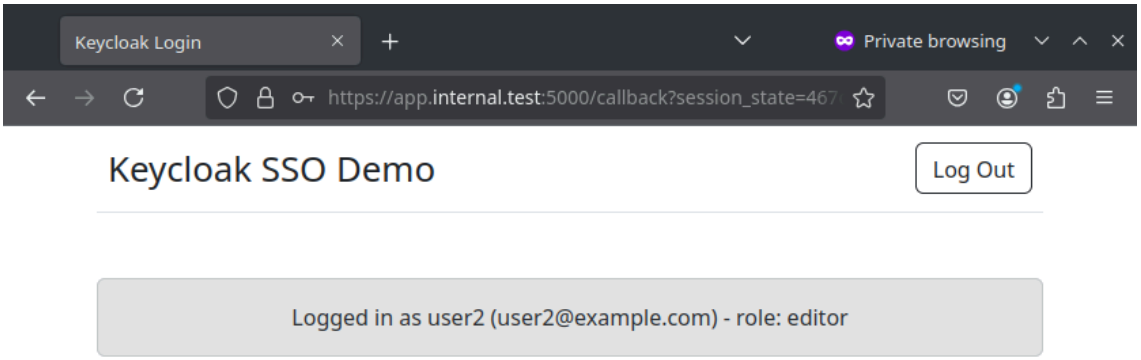
- Successful login and redirection

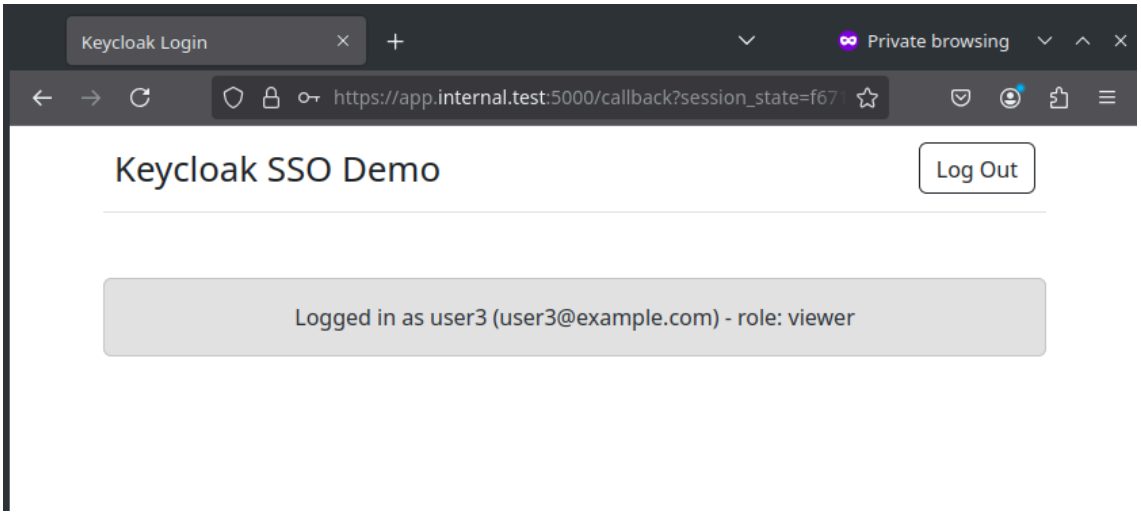


- The client session can be inspected at the server



- Logged in status for other users





Discussion

- Even for the demonstrated simple application, implementing login correctly in the frontend and backend can take time and be error-prone.
- Keycloak starts to prove more useful as the number of clients (services) increase and the need for centralized IAM solution (e.g., for SSO) is essential.
- Support for additional features (e.g., social login, OTP/email verification, password resets, user registration, requesting additional user info, etc.) can also be configured faster through Keycloak.
- Additional measures should be taken into account when deploying this infrastructure in production as recommended in Keycloak production guide:
    - Using TLS for secure communication.
    - Configuring UI and admin endpoints under different hostnames (with the latter being only exposed internally to reduce attack surface).
    - The use of reverse proxy in distributed environments.

- Limiting the number of queued requests.
- Replacing the default `dev-file` with a production-grade database (e.g., PostgreSQL or MySQL).
- Enable observability (e.g., Prometheus metrics and alerts for monitoring).

References

- https://www.keycloak.org/securing-apps/oidc-layers
- https://openid.net/developers/how-connect-works/
- https://openid.net/specs/openid-connect-basic-1_0.html