**E-GOV Portal Full Documentation**

**Overview**

This documentation explains the working of the **E-GOV Portal** based on three main components:

* main.py
* app3.py
* app1.html

It describes how the JWT token authentication flow works with Keycloak and how the user is redirected securely between applications.

A detailed explanation of each file and the overall setup is provided, along with requirements and installation instructions.

**1. Requirements**

Make sure the following are installed:

**Python Packages**

* Flask
* Flask-CORS
* python-keycloak (or requests if verifying tokens manually)
* python-dotenv (optional if using .env for configs)

Install via pip:

pip install flask flask-cors python-keycloak python-dotenv

**Keycloak Server**

* A Keycloak server instance should be running.
* Realm, Client ID, and Client Secret should be configured.
* Enable Direct Access Grants and set client access type as confidential or public accordingly.

**Frontend**

* Browser with JavaScript enabled.

**2. File Breakdown**

**2.1 main.py**

**Purpose**: Starts the Flask server for App 1 (Login/Register Portal)

**Key Actions:**

* Hosts the HTML page app1.html.
* Exposes /verify-token endpoint to verify JWT token with Keycloak.
* Verifies incoming tokens from frontend by posting to Keycloak's introspect or userinfo endpoint.

**Main Endpoints:**

* / (GET): Serves the app1.html frontend.
* /verify-token (POST):
  + Receives a JWT token from the frontend.
  + Verifies the token with Keycloak.
  + Returns {valid: true/false} based on validity.

**Example verify-token Logic:**

@app.route('/verify-token', methods=['POST'])

def verify\_token():

token = request.json.get('token')

# Post token to Keycloak to check if it is active

response = requests.post(KEYCLOAK\_URL, headers={"Authorization": f"Bearer {token}"})

return jsonify({"valid": response.status\_code == 200})

**Summary:** Mainly provides a backend that frontend can use to verify if the user's token is still valid.

**2.2 app3.py**

**Purpose**: Main Backend for E-Gov Login and Database Linkage

**Key Actions:**

* Stores the User Details onto Database integrated.
* Accepts the JWT token passed as a query parameter.
* Validates the JWT Token in the backend before reaching app2.

**2.3 app1.html**

**Purpose**: Frontend interface where the user starts.

**Key Actions:**

* Loads and shows logs on the page.
* On button click, verifies token with /verify-token API.
* If token is valid, redirects to http://localhost:5016/app2.html?token=...
* If no valid token, redirects back to /.

**Functions Explained:**

1. **logMessage(msg)**:
   * Displays messages in the console and on the web page.
2. **verifyWithKeycloak(token)**:
   * Sends the stored JWT token to the /verify-token endpoint.
   * Waits for a success/failure response.
3. **redirectToApp2()**:
   * Checks if a JWT token exists in localStorage.
   * Verifies the token.
   * Redirects to App2 if valid.
   * Otherwise shows an alert and redirects to login.
4. **checkAuthOnLoad()**:
   * Checks if the page was loaded with a token in the URL.
   * If yes, verifies it and saves it into localStorage.

**Summary:** Controls the token verification and redirection logic purely on the client-side, with additional verification at the backend for safety.

**3. Overall Workflow**

1. **User accesses App1 (Login/Portal)**.
2. **If they log in, Keycloak issues a JWT token**.
3. **Token is saved into localStorage**.
4. **User clicks 'Go to App 2'**.
5. **Before redirecting, App1 verifies the JWT token via the backend (/verify-token)**.
6. **If valid, user is redirected to App2 with token passed in the URL**.
7. **App2 can optionally verify again**.

**Security Measures:**

* Always verify token server-side even if client verifies it.
* Consider expiring the token from localStorage when session ends.

**4. Folder Structure**

project/

|

|-- main.py

|-- app3.py

|-- templates/

| |-- app1.html

| |-- index.html

| |-- index3.html

|-- requirements.txt

**5. Future Improvements**

* Refresh token implementation.
* Protect Flask routes with a decorator to require valid JWTs.
* Use HTTPS in production.
* Handle token expiration with Keycloak refresh tokens.
* Improve UI with proper login/logout flow.

**6. Conclusion**

This system shows a simple and secure way to:

* Authenticate with Keycloak.
* Pass tokens securely.
* Verify tokens backend and frontend.
* Move between micro-applications while maintaining authentication state.

With the above setup, you can easily extend the portal to include multiple microservices with centralized authentication via Keycloak.

**End of Documentation**